

Laboratorul 1: Introducere în Haskell

Pentru început, vă veți familiariza cu mediul de programare **GHC** (Glasgow Haskell Compiler). Acesta include doua componente: **GHCi** (care este un interpretor) și **GHC** (care este un compilator).

Descărcare și instalare

Pentru instalare puteți citi mini-tutorialul de la acest [https://docs.google.com/document/d/1lMvx4dRw1rXQ1KiW80poZJwG6F0v6FQU/edit\[LINK](https://docs.google.com/document/d/1lMvx4dRw1rXQ1KiW80poZJwG6F0v6FQU/edit[LINK)

De asemenea, este recomandată folosirea unui stil standard de formatare a fișierelor sursă, spre exemplu <https://github.com/tibbe/haskell-style-guide/blob/master/haskell-style.md>.

GHCi

1. Deschideți un terminal si introduceți comanda **ghci** (în Windows este posibil să aveți instalat WinGHCi). După câteva informații despre versiunea instalată va apare

Prelude>

Prelude este librăria standard: <http://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html>

În interpretor puteți:

- să introduceți expresii, care vor fi evaluate atunci cand este posibil:

```
Prelude> 2+3
5
Prelude> False || True
True
Prelude> x
<interactive>:10:1: error: Variable not in scope: x
Prelude> x=3
Prelude> x
3
Prelude> y=x+1
Prelude> y
4
Prelude> head [1,2,3]
1
Prelude> head "abcd"
'a'
Prelude> tail "abcd"
'bcd'
```

să introduceți comenzi, orice comandă fiind precedată de ":"

- `?:` - este comanda *help*

:q - este comanda *quit*

:cd - este comanda *change directory*

:t - este comanda *type*

```
Prelude> :t True
```

```
True :: Bool
```

https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html

2. Fișierele sursă sunt fișiere cu extensia **.hs**, pe care le puteți edita cu un editor la alegerea voastră. Deschideți fișierul **lab1.hs** care conține următorul cod:

[illegible]

Fără încărcarea fișierului, încercați să calculați `double myInt`:

```
Prelude> double myInt
```

Observați mesajele de eroare. Acum încărcați fișierul folosind comanda **load** (:!) și încercați din nou să calculați `double myInt`:

```
Prelude> :l lab1.hs
[1 of 1] Compiling Main                ( lab1.hs, interpreted )
Ok, 1 module loaded.
*Main> double myInt
*Main> double 2000
```

Modificați fișierul adăugînd o funcție `triple`. Dacă fișierul este deja încărcat, puteți să îl reîncărați folosind comanda ***reload*** (:r).

Puteti reveni în **Prelude** folosind `:m` -

```
Prelude> :l lab1.hs
[1 of 1] Compiling Main                ( lab1.hs, interpreted )
Ok, 1 module loaded.
*Main> :m - Main
Prelude>
```

Ați observat că în mesajele primite a apărut noțiunea de **modul**. Practic, fișierul lab1.hs conține un modul care se numește **Main**, definit automat.

Elemente de limbaj

3. Există numeroase librării foarte utile. Cum putem să le identificăm? O sursă de informații foarte bună este **Hoogle** <https://hoogle.haskell.org/>

Căutați funcția `head` folosită anterior. Observăm că se găsește atât în librăria `{Prelude}`, cât și în librăria `Data.List`.

Să presupunem că vrem să generăm toate permutările unei liste. Căutați funcția `permutation` (sau ceva asemănător) și observăm că în librăria `Data.List` se găsește o funcție `permutations`. Faceți click pe numele funcției (sau al librăriei) pentru a putea citi detalii despre această funcție. Pentru a o folosi în interpretor va trebui să încărcăm librăria `Data.List` folosind comanda `import`

```
Prelude> :t permutations
<interactive>:1:1: error: Variable not in scope: permutations
Prelude> import Data.List
Prelude Data.List> :t permutations
permutations :: [a] -> [[a]]
Prelude Data.List> permutations [1,2,3]
[[1,2,3],[2,1,3],[3,2,1],[2,3,1],[3,1,2],[1,3,2]]
Prelude Data.List> permutations "abc"
["abc","bac","cba","bca","cab","acb"]
```

Atenție! funcția `permutations` întoarce o listă de liste.

Eliminați librăria folosind

```
Prelude> :m - Data.List
```

Librăriile se includ în fișiere sursă folosind comanda `import`. Descideți fișierul `lab1.hs` și adugați la început

```
import Data.List
```

Încărcați fișierul în interpretor și evaluați

```
*Main> permutations [1..myInt]
```

Ce se întâmplă? `[1..myInt]` este lista `[1,2,3,..., myInt]` care are o dimensiune foarte mare. Observăm că putem folosi valori numerice foarte mari. Evaluarea expresiei o oprim cu `Ctrl+C`.

În librăria `Data.List` căutați funcția `subsequences`, înțelegeți ce face și folosiți-o pe câteva exemple.

Indentare

4. În Haskell se recomandă scrierea codului folosind *indentarea*. În anumite situații, nerespectarea regulilor de indentare poate provoca erori la încărcarea programului.

În fișierul `lab1.hs` deplasați cu câteva spații definiția funcției `double`:

```
double :: Integer -> Integer
double x = x+x
```

Reîncărcați programul. Ce observați?

Atenție! În unele editoare se recomandă înlocuirea tab-urilor cu spații.

Să definim funcția `maxim`

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

Varianta cu indentare este:

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

Dorim acum să scriem o funcție care calculează maximul a trei numere. Evident, o varianta este

```
maxim3 x y z = maxim x (maxim y z)
```

Scrieți funcția `maxim3` fără a folosi `maxim`, utilizând direct `if` și scrierea indentată.

Putem scrie funcția `maxim3` folosind expresia `let...in` astfel

```
maxim3 x y z = let u = (maxim x y) in (maxim u z)
```

Atenție! expresia `let...in` crează scop local.

Varianta cu indentare este

```
maxim3 x y z =
    let
        u = maxim x y
    in
        maxim u z
```

Scrieți o funcție `maxim4` folosind varianta cu `let...in` și indentare.

Scrieți o funcție care testează funcția `maxim4` prin care să verificați ca rezultatul este în relația `>=` cu fiecare din cele patru argumente (operatorii logici în Haskell sunt `||`, `&&`, `not`).

Citiți mai multe despre indentare <https://en.wikibooks.org/wiki/Haskell/Indentation>

Tipuri de date

5. Din exemplele de până acum ați putut observa că în Haskell:

a) există tipuri predefinite: `Integer`, `Bool`, `Char`

b) se pot construi tipuri noi folosind `[]`

```
*Main> :t [1..myInt]
[1..myInt] :: [Integer]

Prelude> :t "abc"
"abc" :: [Char]
```

Evident, `[a]` este tipul *listă de date de tip a*. Tipul `String` este un sinonim pentru `[Char]`.

c) Ați întâlnit tipul `Bool` și valorile `True` și `False`. În Haskell tipul `Bool` este definit astfel

```
data Bool = False | True
```

În această definiție, `Bool` este un *constructor de tip*, iar `True` și `False` sunt *constructori de date*.

d) Sistemul tipurilor în Haskell este mult mai complex. Fără a încărca fișierul `lab1.hs`, definiți direct în GHCi funcția `maxim`:

```
Prelude> maxim x y = if (x > y) then x else y
```

Cu ajutorul comenzii `:t` aflați tipul acestei funcții. Ce observați?

```
Prelude> :t maxim  
maxim :: Ord p => p -> p -> p
```

Răspunsul primit trebuie interpretat astfel: `p` reprezintă un tip arbitrar înzestrat cu o relație de ordine, funcția `maxim` are două argumente de tip `p` și întoarce un rezultat de tip `p`.

Astfel, tipul unei operații poate fi definit de noi sau dedus automat. Vom discuta mai multe în cursurile și laboratoarele următoare.

Exerciții

6. Să se scrie următoarele funcții:

- a) funcție cu 2 parametri care calculează suma pătratelor celor două numere;
- b) funcție cu un parametru ce întoarce mesajul “par” dacă parametrul este par și “impar” altfel;
- c) funcție care calculează factorialul unui număr;
- d) funcție care verifică dacă un primul parametru este mai mare decât dublul celui de-al doilea parametru.

Material suplimentar

- Citiți capitolul *Starting Out* din M. Lipovaca, Learn You a Haskell for Great Good! <http://learnyouahaskell.com/starting-out>

LAB 1

[illegible]

```
par_impar :: Integer -> [Char]
par_impar a = if (mod a 2 == 0) then "par" else "impar"

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

-- d)
dublu :: Integer -> Integer -> Bool
dublu a b = if (a > 2*b) then True else False
-- sau dublu a b = a > 2 * b
```

Laboratorul 2: Funcții

Exerciții

1. Să se scrie o funcție `poly2` care are patru argumente de tip `Double`, `a`, `b`, `c`, `x` și calculează $a \cdot x^2 + b \cdot x + c$. Scrieți și semnatura funcției (`poly :: ceva`).
2. Să se scrie o funcție `eeny` care întoarce “eeny” pentru input par și “meeny” pentru input impar. Hint: puteți folosi funcția `even` (puteți căuta pe <https://hoogle.haskell.org/>).

```
eeny :: Integer -> String
eeny = undefined
```

3. Să se scrie o funcție `fizzbuzz` care întoarce “Fizz” pentru numerele divizibile cu 3, “Buzz” pentru numerele divizibile cu 5 și “FizzBuzz” pentru numerele divizibile cu ambele. Pentru orice alt număr se întoarce șirul vid. Pentru a calcula modulo a două numere puteți folosi funcția `mod`. Să se scrie această funcție în 2 moduri: folosind `if` și folosind gărzi (condiții).

```
fizzbuzz :: Integer -> String
fizzbuzz = undefined
```

Recursivitate

Una dintre diferențele dintre programarea declarativă și cea imperativă este modalitatea de abordare a problemei iterării: în timp ce în programarea imperativă acesta este rezolvată prin bucle (`while`, `for`, ...), în programarea declarativă rezolvarea iterării se face prin conceptul de recursie.

Un avantaj al recursiei față de bucle este acela că ușurează sarcina de scriere și verificare a corectitudinii programelor prin raționamente de tip inductiv: construiește rezultatul pe baza rezultatelor unor subprobleme mai simple (aceeași problemă, dar pe o dimensiune mai mică a datelor).

Un foarte simplu exemplu de recursie este acela al calculării unui element de index dat din secvența numerelor Fibonacci, definită recursiv de:

$$F_n = \begin{cases} n & \text{dacă } n \in \{0, 1\} \\ F_{n-1} + F_{n-2} & \text{dacă } n > 1 \end{cases}$$

Putem transcrie această definiție direct în Haskell:

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

Alternativ, putem folosi o definiție în stil ecuațional (cu șabloane):

```
fibonacciEcuational :: Integer -> Integer
fibonacciEcuational 0 = 0
fibonacciEcuational 1 = 1
```



```
fibonacciEcuational n =
    fibonacciEcuational (n - 1) + fibonacciEcuational (n - 2)
```

4. Numerele tribonacci sunt definite de ecuația

$$T_n = \begin{cases} 1 & \text{dacă } n = 1 \\ 1 & \text{dacă } n = 2 \\ 2 & \text{dacă } n = 3 \\ T_{n-1} + T_{n-2} + T_{n-3} & \text{dacă } n > 3 \end{cases}$$

Să se implementeze funcția `tribonacci` atât cu cazuri cât și ecuațional.

```
tribonacci :: Integer -> Integer
tribonacci = undefined
```

5. Să se scrie o funcție care calculează coeficienții binomiali, folosind recursivitate. Aceștia sunt determinați folosind următoarele ecuații.

$B(n,k) = B(n-1,k) + B(n-1,k-1)$

$B(n,0) = 1$

$B(0,k) = 0$

```
binomial :: Integer -> Integer -> Integer
binomial = undefined
```

Liste

Funcții utile: `head`, `tail`, `take`, `drop`, `length`

6. Să se implementeze următoarele funcții folosind liste:

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

```
verifL :: [Int] -> Bool
verifL = undefined
```

b) `takefinal` - pentru o listă dată ca parametru și un număr `n`, întoarce lista cu ultimele `n` elemente. Dacă lista are mai puțin de `n` elemente, se întoarce lista nemodificată.

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

Cum trebuie să modificăm prototipul funcției pentru a putea fi folosită și pentru șiruri de caractere?

c) `remove` - pentru o listă și un număr `n` se întoarce lista din care se șterge elementul de pe poziția `n`. (Hint: puteți folosi funcțiile `take` și `drop`). Scriți si prototipul funcției.

Recursivitate pe Liste

Listele sunt definite inductiv: - vida `[]` - construită prin adăugarea unui element `head` unei liste existente `tail` (`head:tail`)

Recursivitatea pe liste se bazează pe definiția inductivă a lor.

Exemplu: Dată fiind o listă de numere întregi, să se scrie o funcție `semiPareRec` care elimină numerele impare și le înjumătățește pe cele pare. De exemplu:

```
-- semiPareRec [0,2,1,7,8,56,17,18] == [0,1,4,28,9]
```

```

semiPareRec :: [Int] -> [Int]
semiPareRec [] = []
semiPareRec (h:t)
  | even h    = h `div` 2 : t'
  | otherwise = t'
where t' = semiPareRec t

```

7. Exerciții: să se scrie următoarele funcții folosind recursivitate:

- a) **myreplicate** - pentru un întreg **n** și o valoare **v** întoarce lista de lungime **n** ce are doar elemente egale cu **v**. Să se scrie și prototipul funcției.
- b) **sumImp** - pentru o listă de numere întregi, calculează suma valorilor impare. Să se scrie și prototipul funcției.
- c) **totalLen** - pentru o listă de șiruri de caractere, calculează suma lungimilor șirurilor care încep cu caracterul 'A'.

```

totalLen :: [String] -> Int
totalLen = undefined

```

LAB 2

```
poly :: Double -> Double -> Double -> Double -> Double
```

```
poly a b c x = a*x*x + b*x + c
```

```
eeny :: Integer -> String
```

```
eeny x
```

```
    | even x = "eeny"
```

```
    | otherwise = "meeny"
```

```
fizzbuzz :: Integer -> String
```

```
fizzbuzz x
```

```
    | mod x 15 == 0 = "FizzBuzz"
```

```
    | mod x 3 == 0 = "Fizz"
```

```
    | mod x 5 == 0 = "Buzz"
```

```
    | otherwise = ""
```

```
fizzbuzz2 :: Integer -> String
```

```
fizzbuzz2 x =    if (mod x 15 == 0)
```

```
                then "FizzBuzz"
```

```
                else
```

```
                    if (mod x 3 == 0)
```

```
                    then "Fizz"
```

```
                    else
```

```
                        if (mod x 5 == 0)
```

```
                        then "Buzz"
```

```
                        else ""
```

```
tribonacci :: Integer -> Integer
```

```
tribonacci n
```

```
    | n < 3      = 1
```

```
    | n == 3     = 2
```

```
    | otherwise = tribonacci (n-1) + tribonacci (n-2) +
```

```
tribonacci(n-3)
```

```

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

verifL :: [Int] -> Bool
verifL lst = even (length lst)

takefinal :: [a] -> Int -> [a]
takefinal lst n
    | length lst < n = lst
    | otherwise = drop (length lst - n) lst

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

myreplicate :: Int -> v -> [v]
myreplicate 1 v = [v]
myreplicate n v = [v] ++ myreplicate (n-1) v

sumImp :: [Int] -> Int
sumImp [] = 0
sumImp (h:t)
    | odd h = h + t'
    | otherwise = t'
    where t' = sumImp t

totalLen :: [String] -> Int
totalLen [] = 0
totalLen (h:t)
    | head h == 'A' = length h + t'
    | otherwise = t'
    where t' = totalLen t

```

Laboratorul 3: Liste

Recursivitate pe liste

- 1) Să se scrie o funcție `nrVocale` care pentru o listă de șiruri de caractere, calculează numărul total de vocale ce apar în cuvintele palindrom. Pentru a verifica dacă un șir e palindrom, puteți folosi funcția `reverse`, iar pentru a căuta un element într-o listă puteți folosi funcția `elem`. Puteți defini oricâte funcții auxiliare.

```
nrVocale :: [String] -> Int
nrVocale = undefined
-- nrVocale ["sos", "civic", "palton", "desen", "aerisirea"] = 9
```

- 2) Să se scrie o funcție care primește ca parametru un număr și o listă de întregi, și adaugă elementul dat după fiecare element par din listă. Să se scrie și prototipul funcției.

```
-- f 3 [1,2,3,4,5,6] = [1,2,3,3,4,3,5,6,3]
```

Liste definite prin comprehensiune sau selecție

Haskell permite definirea unei liste prin selectarea și transformarea elementelor din alte liste sursă, folosind o sintaxă asemănătoare definirii mulțimilor matematice:

[expresie | selectori, legari, filtrari]

unde:

selectori una sau mai multe construcții de forma `pattern <- elista` (separate prin virgulă) unde `elista` este o expresie reprezentând o listă iar `pattern` este un șablon pentru elementele listei `elista`

legari zero sau mai multe expresii (separate prin virgulă) de forma `let pattern = expresie` folosind la legarea corespunzătoare a variabilelor din `pattern` cu valoarea `expresie`.

filtrari zero sau mai multe expresii de tip `Bool` (separate prin virgulă) folosite la eliminarea instanțelor selectate pentru care condiția e falsă

expresie expresie descriind elementele listei rezultat

Exemplu Iată cum arată o posibilă implementare a funcției `semiPare` folosind descrieri de liste:

```
semiPareComp :: [Int] -> [Int]
semiPareComp l = [ x `div` 2 | x <- l, even x ]
```

Exercitii

- 3) Să se scrie o funcție care are ca parametru un număr întreg și determină lista de divizori ai acestui număr. Să se scrie și prototipul funcției.

```
-- divizori 4 = [1,2,4]
```

- 4) Să se scrie o funcție care are ca parametru o listă de numere întregi și calculează lista listelor de divizori.

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

```
-- listadiv [1,4,6,8] = [[1],[1,2,4],[1,2,3,6],[1,2,4,8]]
```

- 5) Scrieți o funcție care date fiind limita inferioară și cea superioară (întregi) a unui interval închis și o listă de numere întregi, calculează lista numerelor din listă care aparțin intervalului. De exemplu:

```
-- inInterval 5 10 [1..15] == [5,6,7,8,9,10]
```

```
-- inInterval 5 10 [1,3,5,2,8,-1] = [5,8]
```

- a) Folosiți doar recursie. Denumiți funcția `inIntervalRec`
b) Folosiți descrieri de liste. Denumiți funcția `inIntervalComp`
- 6) Scrieți o funcție care numără câte numere strict pozitive sunt într-o listă dată ca argument.

De exemplu:

```
-- pozitive [0,1,-3,-2,8,-1,6] == 3
```

- a) Folosiți doar recursie. Denumiți funcția `pozitiveRec`
b) Folosiți descrieri de liste. Denumiți funcția `pozitiveComp`.
- Nu puteți folosi recursie, dar veți avea nevoie de o funcție de agregare. (Consultați modulul `Data.List`). De ce nu e posibil să scriem `pozitiveComp` doar folosind descrieri de liste?
- 7) Scrieți o funcție care dată fiind o listă de numere calculează lista pozițiilor elementelor impare din lista originală. De exemplu:

```
-- pozitiiImpare [0,1,-3,-2,8,-1,6,1] == [1,2,5,7]
```

- a) Folosiți doar recursie. Denumiți funcția `pozitiiImpareRec`.
- Indicație: folosiți o funcție ajutătoare, cu un argument în plus reprezentând poziția curentă din listă.
- b) Folosiți descrieri de liste. Denumiți funcția `pozitiiImpareComp`.

- Indicație: folosiți funcția `zip` pentru a asocia poziții elementelor listei (puteți căuta exemplu în curs).

8) Scrieți o funcție care calculează produsul tuturor cifrelor care apar în șirul de caractere dat ca intrare. Dacă nu sunt cifre în șir, răspunsul funcției trebuie să fie 1 . De exemplu:

```
-- multDigits "The time is 4:25" == 40  
-- multDigits "No digits here!" == 1
```

a) Folosiți doar recursie. Denumiți funcția `multDigitsRec`

b) Folosiți descrieri de liste. Denumiți funcția `multDigitsComp`

- Indicație: Veți avea nevoie de funcția `isDigit` care verifică dacă un caracter e cifră și funcția `digitToInt` care transformă un caracter în cifră. Cele 2 funcții se află în pachetul `Data.Char`.

LAB 3

```
import Data.Char

palindrom :: String -> Bool
palindrom s = s == reverse s

voc :: String -> Int
voc "" = 0
voc (h : t)
    | elem h "aeiouAEIOU" = 1 + t'
    | otherwise = t'
    where t' = voc t

-- voc s = sum[1 | c <- s, e `elem` "aeiouAEIOU"]

nrVocale :: [String] -> Int
nrVocale [] = 0
nrVocale (h : t)
    | palindrom h = voc h + t'
    | otherwise = t'
    where t' = nrVocale t

-- nrVocale l = sum[voc s | s <- l, s==reverse s]

f :: Int -> [Int] -> [Int]
f a [] = []
f a (h : t)
    | even h = [h, a] ++ t'
    | otherwise = h : t'
    where t' = f a t

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

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



```

inIntervalComp :: Int -> Int -> [Int] -> [Int]
inIntervalComp a b l = [x | x <- l, x >= a && x <= b]

inIntervalRec :: Int -> Int -> [Int] -> [Int]
inIntervalRec a b [] = []
inIntervalRec a b (h : t)
    | h >= a && h <= b = h : t'
    | otherwise = t'
    where t' = inIntervalRec a b t

pozitiveRec :: [Int] -> Int
pozitiveRec [] = 0
pozitiveRec (h : t)
    | h > 0 = 1 + t'
    | otherwise = t'
    where t' = pozitiveRec t

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

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

pAuxRec :: [Int] -> Int -> [Int]
pAuxRec [] _ = []
pAuxRec (h:t) index
    | odd h = index : t'
    | otherwise = t'
    where t' = pAuxRec t (index+1)

pozitiiiImpareRec :: [Int] -> [Int]
pozitiiiImpareRec l = pAuxRec l 0

```

```
-- importurile se pun la inceputul fisierului
-- mersi

multiDigitsRec :: String -> Int
multiDigitsRec "" = 1
multiDigitsRec (h : t)
    | isDigit h = digitToInt h * t'
    | otherwise = t'
    where t' = multiDigitsRec t

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

Laboratorul 4: Exerciții liste, map, filter

Liste

Reamintiți-vă definirea listelor prin selecție din **Laboratorul 3**. 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]
-}
```

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 metoda prin selecție definiți o funcție

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

astfel încât factori n întoarce lista divizorilor pozitivi ai lui n.

2. Folosind funcția factori, definiți predicatul prim n care întoarce True dacă și numai dacă n este număr prim.

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

3. Folosind numai metoda prin selecție și funcțiile definite anterior, definiți funcția

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

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

Funcția zip

Testați și sesizați diferența:

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

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

4. Definiți funcția 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)]
```

Secțiuni

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.

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 argument o funcție f de tip $a \rightarrow a$ 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 $a \rightarrow b$ 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 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]
```

Exercitii

Rezolvați următoarele exerciții folosind `map` și `filter` (fără recursivitate sau selecție). Pentru fiecare funcție scrieți și prototipul acesteia.

5. 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"
```

6. 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]
```

7. 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]
```

8. 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`).
9. Scrieți o funcție care primește ca argument o listă de întregi și întoarce lista pătratelor numerelor impare.
10. 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`.
11. 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"]
```

12. Definiți recursiv funcțiile `mymap` și `myfilter` cu aceeași funcționalitate ca și funcțiile predefinite.

LAB 4

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

prim :: Int -> Bool
prim x = factori x == [1, x]

prim2 :: Int -> Bool
prim2 x = length(factori x) == 2

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

myzip3 :: [a] -> [b] -> [c] -> [(a, b, c)]
myzip3 a b [c] = []
myzip3 a [] c = []
myzip3 [] b c = []
myzip3 (a1 : a2) (b1 : b2) (c1 : c2) = [(a1, b1, c1)] ++ myzip3 a2
b2 c2

-- myzip3 l1 l2 l3 = [(x, y, z) | i <- [0.. (min (min (length l1)
(length l2)) (length l3)- 1)], let x = l1 !! i, let y = l2 !! i,
let z = l3 !! i]

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

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

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

```

listaSiruri :: Char -> [String] -> [String]
listaSiruri ch = filter (\str -> elem ch str)

-- 9
impare :: [Int] -> [Int]
impare x = filter odd x
patrate :: [Int] -> [Int]
patrate = map (\x -> x*x)
patrateImpare :: [Int] -> [Int]
patrateImpare l = patrate (impare l)

-- 10
patratePozImpare lst = map (\(x1,x2) -> x1 * x1) (filter (\(x1,x2)
-> odd x2) (zip lst [1..]))

-- 11
elimConsoane :: String -> String
elimConsoane str = filter f str where f x = elem x "aeiouAEIOU"
numaiVocale :: [String] -> [String]
numaiVocale = map (\str -> elimConsoane str)

-- 12
mymap :: (a -> b) -> [a] -> [b]
mymap f [] = []
mymap f (h : t) = f h : mymap f t

-- am testat sa vad daca merge bine
sumList2 :: [[Int]] -> [Int]
sumList2 = mymap sum

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

```


Laboratorul 5: Exerciții Fold

FOLD

Funcțiile `foldr` și `foldl` sunt folosite pentru agregarea unei colecții. Definițiile intuitive pentru `foldr` și `foldl` sunt:

```
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 op i xs)
```

```
foldl op unit [a1, a2, a3, ... , an] =  
    (((unit `op` a1) `op` a2) `op` a3) `op` ..) `op` an
```

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

```
ghci> foldr (+) 0 [1..5]  
15  
ghci> foldr (*) 1 [2,3,4]  
24  
ghci> foldr (++) [] ["abc","def","ghi"]  
"abcdefghi"  
ghci> foldl (++) "first" ["abc","def","ghi"]  
"firstabcdefghi"  
ghci> foldr (++) "last" ["abc","def","ghi"]  
"abcdefghilast"
```

Exercitii

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

1. Calculați suma pătratelor elementelor impare dintr-o listă dată ca parametru.
2. Scrieți o funcție care verifică faptul că toate elementele dintr-o listă sunt True, folosind foldr.
3. Scrieți o funcție care verifică dacă toate elementele dintr-o listă de numere întregi satisfac o proprietate dată ca parametru.

```
allVerifies :: (Int -> Bool) -> [Int] -> Bool
allVerifies = undefined
```

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

```
anyVerifies :: (Int -> Bool) -> [Int] -> Bool
anyVerifies = undefined
```

5. Redefiniți funcțiile map și filter folosind foldr. Le puteți numi mapFoldr și filterFoldr.
6. Folosind funcția foldl, definiți funcția listToInt care transformă o lista de cifre (un număr foarte mare stocat sub formă de listă) în numărul întreg asociat. Se presupune ca lista de intrare este dată corect.

```
listToInt :: [Integer] -> Integer
listToInt = undefined
-- listToInt [2,3,4,5] = 2345
```

7.

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

```
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.

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

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

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

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

LAB 5

```
-- 1
sumOdd :: [Int] -> Int
sumOdd l = foldl (+) 0 (map (^ 2) (filter odd l))

-- 2
allTrue :: [Bool] -> Bool
allTrue = foldr (&&) True

-- 3
allVerifies :: (Int -> Bool) -> [Int] -> Bool
allVerifies f l
  | length (filter f l) == length l = True
  | otherwise = False

allVerifiesFold :: (Int -> Bool) -> [Int] -> Bool
allVerifiesFold f l = foldr (&&) True (foldr (\x xs -> f x : xs)
[] l)

-- 4
anyVerifies :: (Int -> Bool) -> [Int] -> Bool
anyVerifies f l
  | length (filter f l) > 0 = True
  | otherwise = False

anyVerifiesFold :: (Int -> Bool) -> [Int] -> Bool
anyVerifiesFold f l = foldr (+) 0 (foldr (\x xs -> if f x then 1 :
xs else 0 : xs) [] l) > 0

-- 5
mapFoldr :: (a -> b) -> [a] -> [b]
mapFoldr _ [] = []
mapFoldr f l = foldr (\x xs -> f x : xs) [] l
```

```

filterFoldr :: (a -> Bool) -> [a] -> [a]
filterFoldr _ [] = []
filterFoldr f l = foldr (\x xs -> if f x then x : xs else xs) [] l

-- 6
listToInt :: [Integer] -> Integer
listToInt l = foldl (\a b -> a * 10 + b) 0 l

-- 7
-- a
rmChar :: Char -> String -> String
rmChar ch = filter (/= ch)

-- b
rmCharsRec :: String -> String -> String
rmCharsRec [] l = l
rmCharsRec (h : t) s2 = rmCharsRec t (rmChar h s2)

-- c
rmCharsFold :: String -> String -> String
rmCharsFold s1 = foldr (\x xs -> if x `elem` s1 then xs else x :
xs) []

```

Laboratorul 6: Tipuri de date

Exercițiul 1

Vom începe prin a scrie câteva funcții 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 și 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) Scrieți o funcție

```
ePortocalaDeSicilia :: Fruct -> Bool
ePortocalaDeSicilia = undefined
```

care indică dacă un fruct este o portocală de Sicilia sau nu. Soiurile de portocale din Sicilia sunt Tarocco, Moro și Sanguinello. De exemplu,

```
test_ePortocalaDeSicilia1 =
  ePortocalaDeSicilia (Portocala "Moro" 12) == True
test_ePortocalaDeSicilia2 =
  ePortocalaDeSicilia (Mar "Ionatan" True) == False
```

b) Scrieți o funcție

```
nrFeliiSicilia :: [Fruct] -> Int
nrFeliiSicilia = undefined
```

```
test_nrFeliiSicilia = nrFeliiSicilia listaFructe == 52
```

care calculează numărul total de felii ale portocalelor de Sicilia dintr-o listă de fructe.

c) Scrieți o funcție

```
nrMereViermi :: [Fruct] -> Int
nrMereViermi = undefined
```

```
test_nrMereViermi = nrMereViermi listaFructe == 2
```

care calculează numărul de mere care au viermi dintr-o lista de fructe.

Exercițiul 2

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

a) Scrieți o funcție

```
vorbeste :: Animal -> String
vorbeste = undefined
```

care întoarce "Meow!" pentru pisică și "Woof!" pentru câine.

b) Vă reamintiți tipul de date predefinit Maybe

```
data Maybe a = Nothing | Just a
```

scrieți o funcție

```
rasa :: Animal -> Maybe String
rasa = undefined
```

care întoarce rasa unui câine dat ca parametru sau Nothing dacă parametrul este o pisică.

Exercițiul 3

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

```
data Linie = L [Int]
    deriving Show
data Matrice = M [Linie]
    deriving Show
```

a) Scrieți o funcție care verifică dacă suma elementelor de pe fiecare linie este egală cu o valoare n. Rezolvați cerința folosind foldr.

```
verifica :: Matrice -> Int -> Bool
```

```
verifica = undefined
```

```
test_veri1 = 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
```

- 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.

```
doarPozN :: Matrice -> Int -> Bool
```

```
doarPozN = undefined
```

```
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
```

- c) Definiți predicatul corect care verifică dacă toate liniile dintr-o matrice au aceeași lungime.

```
corect :: Matrice -> Bool
```

```
corect = undefined
```

```
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
```

LAB 6

```
-- 1

data Fruct = Mar String Bool | Portocala String Int

-- a)

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

-- b)

listaFructe =
  [ Mar "Ionatan" False,
    Portocala "Sanguinello" 10,
    Portocala "Valencia" 22,
    Mar "Golden Delicious" True,
    Portocala "Sanguinello" 15,
    Portocala "Morro" 12,
    Portocala "Tarocco" 3,
    Portocala "Morro" 12,
    Portocala "Valencia" 2,
    Mar "Golden Delicious" False,
    Mar "Golden" False,
    Mar "Golden" True
  ]

getFellii :: Fruct -> Int
getFellii (Mar _ _) = 0
getFellii (Portocala _ f) = f

nrFelliiSicilia :: [Fruct] -> Int
nrFelliiSicilia [] = 0
nrFelliiSicilia (x : xs) =
  if ePortocalaDeSicilia x
  then getFellii x + nrFelliiSicilia xs
```



```

    else nrFeliisicilia xs

-- c)
hasViermi :: Fruct -> Bool
hasViermi (Portocala _ _) = False
hasViermi (Mar _ v) = v

nrMereViermi :: [Fruct] -> Int
nrMereViermi [] = 0
nrMereViermi (x : xs) = if hasViermi x then 1 + nrMereViermi xs
    else nrMereViermi xs

-- 2
type NumeA = String
type Rasa = String
data Animal = Pisica NumeA | Caine NumeA Rasa
    deriving (Show)

-- a)
vorbeste :: Animal -> String
vorbeste (Pisica _) = "Meoow!"
vorbeste (Caine _ _) = "Woof!"

-- b)
rasa :: Animal -> Maybe String
rasa (Pisica _) = Nothing
rasa (Caine _ rasa) = Just rasa

-- 3
data Linie = L [Int]
    deriving (Show)

data Matrice = M [Linie]
    deriving (Show)

-- a)

```

```

matriceaMea = M [L [1, 2, 3], L [4, 5], L [2, 3, 6, 8], L [8, 5,
3]]

-- getSumaPeLinie :: Linie -> Int
-- getSumaPeLinie (L []) = 0
-- getSumaPeLinie (L list) = sum list

getSumaLinii :: Matrice -> [Int]
getSumaLinii (M []) = []
getSumaLinii (M (L x : xs)) = sum x : getSumaLinii (M xs)

verifica :: Matrice -> Int -> Bool
verifica (M m) n = and (foldr (\x y -> if x == n then True : y
else False : y) [] (getSumaLinii (M m)))

-- b
getLiniiDeLungimeN :: Matrice -> Int -> Matrice
getLiniiDeLungimeN (M m) n = M (filter \(L l) -> length l == n)
m)

getCateNegativePeLinii :: Matrice -> [Int]
getCateNegativePeLinii (M m) = map \(L l) -> length (filter (<=
0) l) m

doarPozN :: Matrice -> Int -> Bool
doarPozN (M m) n = sum (getCateNegativePeLinii (getLiniiDeLungimeN
(M m) n)) == 0

-- c)
corect :: Matrice -> Bool
corect (M []) = True
corect (M (L h1 : [])) = True
corect (M (L h1 : L h2 : t)) = length h1 == length h2 && corect (M
(L h2 : t))

```

Laboratorul 7: ADT si Clase de tipuri

1. Expresii și Arbori

Se dau următoarele tipuri de date reprezentând expresii și 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 instanțieze clasa Show pentru tipul de date Expr, astfel încât să se afișeze mai simplu expresiile.

1.2. Să se scrie o funcție evalExp :: Expr -> Int care evaluează o expresie determinând valoarea acesteia.

```
evalExp :: Expr -> Int
evalExp = undefined
```

Exemplu:

```
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)
test11 = evalExp exp1 == 6
test12 = evalExp exp2 == 14
test13 = evalExp exp3 == 13
test14 = evalExp exp4 == 16
```

1.3. Să se scrie o funcție evalArb :: Tree -> Int care evaluează o expresie modelată sub formă de arbore, determinând valoarea acesteia.

```
evalArb :: Tree -> Int
evalArb = undefined
```

```

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))
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

```

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

```

expToArb :: Expr -> Tree
expToArb = undefined

```

2. Clasa Collection

În acest exercitiu vom exersa manipularea listelor și tipurilor de date prin implementarea a catorva colecții de tip tabelă asociativă cheie-valoare.

Aceste colecții vor trebui să aibă următoarele facilitati

- crearea unei colecții vide
- crearea unei colecții cu un element
- adăugarea/actualizarea unui element într-o colecție
- căutarea unui element într-o colecție
- ștergerea (marcarea ca șters a) unui element dintr-o colecție
- obținerea listei cheilor
- obținerea listei valorilor
- obținerea 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
  lookup :: 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. Adăugați definiții implicite (în funcție de funcțiile celelalte) pentru

- keys

- b. values
- c. fromList

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

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

Faceti PairList instantă a clasei Collection.

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 instantă a clasei Collection.

LAB 7

```
import Control.Arrow (ArrowChoice (right))

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

```

evalExp (a **: b) = evalExp a * evalExp b

-- 1.3
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))

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

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

-- 1.4
expToArb :: Expr -> Tree
expToArb (Const x) = Lf x
expToArb (a :+: b) = Node Add (expToArb a) (expToArb b)
expToArb (a **: b) = Node Mult (expToArb a) (expToArb b)

-- 2.1
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

```

```

-- a
keys c = map fst (toList c)

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

-- c
fromList = foldr (uncurry insert) empty

-- 2.2
newtype PairList k v = PairList {getPairList :: [(k, v)]}

instance Collection PairList where
    empty = PairList []
    singleton key value = PairList [(key, value)]
    insert key value (PairList l) = insert key value (delete key
(PairList l))
    toList = getPairList
    clookup key (PairList l) = Main.clookup key (PairList l)
    delete key (PairList l) = PairList $ filter \(k, v) -> k /= key)
l

-- 2.3
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

instance Collection SearchTree where
    empty = Empty
    singleton k value = BNode Empty k (Just value) Empty

```



```

insert k v Empty = singleton k v
insert k v (BNode leftTree key val rightTree)
  | key < k = BNode leftTree key val (insert k v rightTree)
  | key > k = BNode (insert k v leftTree) key val rightTree
  | otherwise = BNode leftTree key val rightTree
toList Empty = []
toList (BNode leftTree key Nothing rightTree) = toList leftTree
++ toList rightTree
toList (BNode leftTree key val rightTree) = (key, case val of {
Just a -> a }) : toList leftTree ++ toList rightTree
clookup k Empty = Nothing
clookup k (BNode leftTree key val rightTree)
  | k == key = val
  | k < key = clookup k leftTree
  | otherwise = clookup k rightTree
delete k (BNode leftTree key val rightTree)
  | k == key = BNode leftTree key Nothing rightTree
  | k < key = delete k leftTree
  | otherwise = delete k rightTree

```

Laboratorul 8: ADT. Clase de Tipuri

Exercițiul 1

Se dau următoarele tipuri de date ce reprezintă puncte cu număr variabil de coordonate întregi:

```
data Punct = Pt [Int]
```

Arbori cu informația în frunze și clasă de tipuri ToFromArb

```
data Arb = Vid | F Int | N Arb Arb
    deriving Show
```

```
class ToFromArb a where
    toArb :: a -> Arb
    fromArb :: Arb -> a
```

- a) Să se scrie o instanță a clasei Show pentru tipul de date Punct, astfel încât lista coordonatelor să fie afișată sub forma de tuplu.

```
-- Pt [1,2,3]
```

```
-- (1, 2, 3)
```

```
-- Pt []
```

```
-- ()
```

- b) Să se scrie o instanță a clasei ToFromArb pentru tipul de date Punct astfel încât lista coordonatelor punctului să coincidă cu frontiera arborelui.

```
-- toArb (Pt [1,2,3])
```

```
-- N (F 1) (N (F 2) (N (F 3) Vid))
```

```
-- fromArb $ N (F 1) (N (F 2) (N (F 3) Vid)) :: Punct
```

```
-- (1,2,3)
```

Exercițiul 2

Se dă următorul tip de date reprezentând figuri geometrice.

```
data Geo a = Square a | Rectangle a a | Circle a
    deriving Show
```

Și clasa GeoOps în care se definesc operațiile perimetru și area.

```
class GeoOps g where
    perimetru :: (Floating a) => g a -> a
    area :: (Floating a) => g a -> a
```

- a) Să se instanțieze clasa `GeoOps` pentru tipul de date `Geo`. Pentru valoarea `pi` există funcția cu același nume (`pi`).

```
-- ghci> pi
```

```
-- 3.141592653589793
```

- b) Să se instanțieze clasa `Eq` pentru tipul de date `Geo`, astfel încât două figuri geometrice să fie egale dacă au perimetrul egal.

LAB 8

```
-- 1
data Punct = Pt [Int]

data Arb = Vid | F Int | N Arb Arb
  deriving (Show)

class ToFromArb a where
  toArb  :: a -> Arb
  fromArb :: Arb -> a

-- a
instance Show Punct where
  show (Pt []) = "()"
  show (Pt l) =
    "("
    ++ show (head l)
    ++ concat
      [ b : show a
        | (a, b) <- zip (tail l) [',', ' ', ','] ..]
    ++ ")"

-- b
getPunctToList :: Punct -> [Int]
getPunctToList (Pt []) = []
getPunctToList (Pt (x : xs)) = x : getPunctToList (Pt xs)

instance ToFromArb Punct where
  fromArb Vid = Pt []
  fromArb (F a) = Pt [a]
  fromArb (N st dr) = Pt (getPunctToList (fromArb st) ++
    getPunctToList (fromArb dr))
```

```

toArb (Pt []) = Vid
toArb (Pt [a]) = F a
toArb (Pt (x : xs)) = N (F x) (toArb (Pt xs))

-- 2
data Geo a = Square a | Rectangle a a | Circle a
  deriving (Show)

class GeoOps g where
  perimeter :: (Floating a) => g a -> a
  area :: (Floating a) => g a -> a

-- a
instance GeoOps Geo where
  perimeter (Square l) = 4 * l
  perimeter (Rectangle lMic lMare) = 2 * lMic + 2 * lMare
  perimeter (Circle r) = 2 * pi * r

  area (Square l) = l ^ 2
  area (Rectangle lMic lMare) = lMic * lMare
  area (Circle r) = pi * (r ^ 2)

-- b
instance (Floating l, Eq l) => Eq (Geo l) where
  a == b = perimeter a == perimeter b

```

Laboratorul 10 - Functor

```
{-  
class Functor f where  
fmap :: ( a -> b ) -> f a -> f b  
-}
```

Scrieți instanțe ale clasei Functor pentru tipurile de date descrise mai jos.

```
newtype Identity a = Identity a
```

```
data Pair a = Pair a a
```

```
data Constant a b = Constant b
```

```
data Two a b = Two a b
```

```
data Three a b c = Three a b c
```

```
data Three' a b = Three' a b b
```

```
data Four a b c d = Four a b c d
```

```
data Four'' a b = Four'' a a a b
```

```
data Quant a b = Finance | Desk a | Bloor b
```

S-ar putea să fie nevoie să adăugați unele constrângeri la definirea instanțelor

```
data LiftItOut f a = LiftItOut (f a)
```

```
data Parappa f g a = DaWrappa (f a) (g a)
```

```
data IgnoreOne f g a b = IgnoringSomething (f a) (g b)
```

```
data Notorious g o a t = Notorious (g o) (g a) (g t)
```

```
data GoatLord a = NoGoat | OneGoat a | MoreGoats (GoatLord a) (GoatLord a) (GoatLord a)
```

```
data TalkToMe a = Halt | Print String a | Read (String -> a)
```

LAB 10

```
newtype Identity a = Identity a

data Pair a = Pair a a

data Constant a b = Constant b

data Two a b = Two a b

data Three a b c = Three a b c

data Three' a b = Three' a b b

data Four a b c d = Four a b c d

data Four'' a b = Four'' a a a b

data Quant a b = Finance | Desk a | Bloor b

instance Functor Identity where
    fmap f (Identity a) = Identity (f a)

instance Functor Pair where
    fmap f (Pair a b) = Pair (f a) (f a)

instance Functor (Constant a) where
    fmap f (Constant b) = Constant (f b)

instance Functor (Two a) where
    fmap f (Two a b) = Two a (f b)

instance Functor (Three a b) where
    fmap f (Three a b c) = Three a b (f c)
```

```
instance Functor (Three' a) where
  fmap f (Three' a b c) = Three' a (f b) (f b)

instance Functor (Four a b c) where
  fmap f (Four a b c d) = Four a b c (f d)

instance Functor (Four'' a) where
  fmap f (Four'' a b c d) = Four'' a a a (f d)

instance Functor (Quant a) where
  fmap f Finance = Finance
  fmap f (Desk a) = Desk a
  fmap f (Bloor b) = Bloor (f b)

data LiftItOut f a = LiftItOut (f a)

data Parappa f g a = DaWrappa (f a) (g a)

data IgnoreOne f g a b = IgnoringSomething (f a) (g b)

data Notorious g o a t = Notorious (g o) (g a) (g t)

data GoatLord a = NoGoat | OneGoat a | MoreGoats (GoatLord a)
                (GoatLord a) (GoatLord a)

data TalkToMe a = Halt | Print String a | Read (String -> a)

instance Functor f => Functor (LiftItOut f) where
  fmap f (LiftItOut fa) = LiftItOut (fmap f fa)

instance (Functor f, Functor g) => Functor (Parappa f g) where
  fmap f (DaWrappa fa ga) = DaWrappa (fmap f fa) (fmap f ga)

instance Functor g => Functor (IgnoreOne f g a) where
```



```
fmap f (IgnoringSomething fa gb) = IgnoringSomething fa (fmap f
gb)

instance Functor g => Functor (Notorious g o a) where
  fmap f (Notorious go ga gt) = Notorious go ga (fmap f gt)

instance Functor GoatLord where
  fmap f NoGoat = NoGoat
  fmap f (OneGoat a) = OneGoat (f a)
  fmap f (MoreGoats a b c) = MoreGoats (fmap f a) (fmap f b) (fmap
f c)

instance Functor TalkToMe where
  fmap f Halt = Halt
  fmap f (Print s a) = Print s (f a)
  fmap f (Read g) = Read (f . g)
```

Laboratorul 11

Amintiți-vă clasele Functor și Applicative, rulați și analizați următoarele exemple.

```
{-
class Functor f where
    fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b

Just length <*> Just "world"

Just (++ " world") <*> Just "hello,"
pure (+) <*> Just 3 <*> Just 5
pure (+) <*> Just 3 <*> Nothing
(++ ) <$> ["ha","heh"] <*> ["?","!"]
-}
```

Exercitii

1. Se dă tipul de date

```
data List a = Nil
            | Cons a (List a)
            deriving (Eq, Show)
```

Să se scrie instanțe Functor și Applicative pentru tipul de date List.

```
instance Functor List where
    fmap = undefined
instance Applicative List where
    pure = undefined
    (<*>) = undefined
```

Exemple

```
f = Cons (+1) (Cons (*2) Nil)
v = Cons 1 (Cons 2 Nil)
test1 = (f <*> v) == Cons 2 (Cons 3 (Cons 2 (Cons 4 Nil)))
```

2. Se dă tipul de date

```
data Cow = Cow {
    name :: String
    , age :: Int
    , weight :: Int
} deriving (Eq, Show)
```

- a) Să se scrie funcțiile `noEmpty`, respectiv `noNegative` care validează un string, respectiv un întreg.

```
noEmpty :: String -> Maybe String
noEmpty = undefined
```

```
noNegative :: Int -> Maybe Int
noNegative = undefined
```

```
test21 = noEmpty "abc" == Just "abc"
test22 = noNegative (-5) == Nothing
test23 = noNegative 5 == Just 5
```

- b) Să se scrie o funcție care construiește un element de tip `Cow` verificând numele, vârsta și greutatea cu funcțiile de la a).

```
cowFromString :: String -> Int -> Int -> Maybe Cow
cowFromString = undefined
```

```
test24 = cowFromString "Milka" 5 100 == Just (Cow {name = "Milka", age = 5, weight = 100})
```

- c) Se scrie funcția de la b) folosind `fmap` și `<*>`.

3. Se dau următoarele tipuri de date:

```
newtype Name = Name String deriving (Eq, Show)
newtype Address = Address String deriving (Eq, Show)
```

```
data Person = Person Name Address
  deriving (Eq, Show)
```

- a) Să se implementeze o funcție `validateLength` care validează lungimea unui șir (să fie mai mică decât numărul dat ca parametru).

```
validateLength :: Int -> String -> Maybe String
validateLength = undefined
```

```
test31 = validateLength 5 "abc" == Just "abc"
```

- b) Să se implementeze funcțiile `mkName` și `mkAddress` care transformă un șir de caractere într-un element din tipul de date asociat, validând stringul cu funcția `validateLength` (numele trebuie să aibă maxim 25 caractere iar adresa maxim 100).

```
mkName :: String -> Maybe Name
mkName = undefined
```

```
mkAddress :: String -> Maybe Address
mkAddress = undefined
```

```
test32 = mkName "Gigel" == Just (Name "Gigel")
test33 = mkAddress "Str Academiei" == Just (Address "Str Academiei")
```

- c) Să se implementeze funcția `mkPerson` care primește ca argument două șiruri de caractere și formează un element de tip `Person` dacă sunt validate condițiile, folosind funcțiile implementate mai sus.

```
mkPerson :: String -> String -> Maybe Person
mkPerson = undefined
```

```
test34 = mkPerson "Gigel" "Str Academiei" == Just (Person (Name "Gigel") (Address "Str Academiei"))
```

- d) Să se implementeze funcțiile de la b) și c) folosind `fmap` și `<*>`.

LAB 11

```
-- 1
data List a
  = Nil
  | Cons a (List a)
  deriving (Eq, Show)

instance Functor List where
  fmap f Nil = Nil
  fmap f (Cons a l) = Cons (f a) (fmap f l)

myConcat :: List a -> List a -> List a
myConcat a Nil = a
myConcat Nil a = a
myConcat (Cons a l) b = Cons a (myConcat l b)

instance Applicative List where
  pure a = Cons a Nil
  f <*> Nil = Nil
  Nil <*> f = Nil
  Cons f fs <*> (Cons a l) = Cons (f a) (myConcat (fmap f l) (fs
<*> Cons a l))

-- 2
data Cow = Cow
  { name :: String,
    age :: Int,
    weight :: Int
  }
  deriving (Eq, Show)

noEmpty :: String -> Maybe String
noEmpty s
  | not (null s) = Just s
  | otherwise = Nothing
```

```

noNegative :: Int -> Maybe Int
noNegative n
  | n < 0 = Nothing
  | otherwise = Just n

cowFromString :: String -> Int -> Int -> Maybe Cow
cowFromString n a w
  | noEmpty n == Just n && noNegative a == Just a && noNegative w
  == Just w = Just Cow {name = n, age = a, weight = w}
  | otherwise = Nothing

cowFromString2 :: String -> Int -> Int -> Maybe Cow
cowFromString2 n a w = Cow <$> noEmpty n <*> noNegative a <*>
noNegative w

-- 3
newtype Name = Name String deriving (Eq, Show)

newtype Address = Address String deriving (Eq, Show)

data Person = Person Name Address
  deriving (Eq, Show)

-- a
validateLength :: Int -> String -> Maybe String
validateLength n sir
  | n > length sir = Just sir
  | otherwise = Nothing

-- b
mkName :: String -> Maybe Name
mkName nume
  | validateLength 26 nume == Just nume = Just (Name nume)
  | otherwise = Nothing

```

```

mkAddress :: String -> Maybe Address
mkAddress adr
  | validateLength 101 adr == Just adr = Just (Address adr)
  | otherwise = Nothing

-- c
mkPerson :: String -> String -> Maybe Person
mkPerson n a
  | mkName n == Just (Name n) && mkAddress a == Just (Address a) =
Just (Person (Name n) (Address a))
  | otherwise = Nothing

-- d
mkPerson2 :: String -> String -> Maybe Person
mkPerson2 n a = Person <$> Just (Name n) <*> Just (Address a)

mkAddress2 :: String -> Maybe Address
mkAddress2 adr = Address <$> validateLength 101 adr

mkName2 :: String -> Maybe Name
mkName2 n = Name <$> validateLength 26 n

```

Laboratorul 12

Exerciții pentru Foldable

1. Implementați următoarele funcții folosind foldMap și/sau foldr din clasa Foldable, apoi testați-le cu mai multe tipuri care au instanță pentru Foldable

```
elem1 :: (Foldable t, Eq a) => a -> t a -> Bool
elem1 = undefined
```

```
null1 :: (Foldable t) => t a -> Bool
null1 = undefined
```

```
length1 :: (Foldable t) => t a -> Int
length1 = undefined
```

```
toList1 :: (Foldable t) => t a -> [a]
toList1 = undefined
```

fold combină elementele unei structuri folosind structura de monoid a acestora.

```
fold1 :: (Foldable t, Monoid m) => t m -> m
fold1 = undefined -- Hint: folosiți foldMap
```

2. Scrieți instanțe ale lui Foldable pentru următoarele tipuri, implementand funcția foldMap.

```
data Constant a b = Constant b
```

```
data Two a b = Two a b
```

```
data Three a b c = Three a b c
```

```
data Three' a b = Three' a b b
```

```
data Four' a b = Four' a b b b
```

```
data GoatLord a = NoGoat | OneGoat a | MoreGoats (GoatLord a) (GoatLord a) (GoatLord a)
```


LAB 12

```
import Data.Monoid

elem1 :: (Foldable t, Eq a) => a -> t a -> Bool
elem1 x = foldr (\h t -> h == x || t) False

elem1' x xs = getAny $ foldMap (\h -> Any (h == x)) xs

null1 :: (Foldable t) => t a -> Bool
null1 = foldr (\x xs -> False) True

null1' xs = not (getAny $ foldMap (\x -> Any True) xs)

length1 :: (Foldable t) => t a -> Int
length1 = foldr (const (1 +)) 0

length1' xs = getSum $ foldMap (const 1) xs

toList1 :: (Foldable t) => t a -> [a]
toList1 = foldMap (: [])

fold1 :: (Foldable t, Monoid m) => t m -> m
fold1 = foldMap id

data Constant a b = Constant b

instance Foldable (Constant a) where
    foldMap f (Constant b) = f b

data Two a b = Two a b

instance Foldable (Two a) where
    foldMap f (Two a b) = f b

data Three a b c = Three a b c
```

```
instance Foldable (Three a b) where
  foldMap f (Three a b c) = f c

data Three' a b = Three' a b b

instance Foldable (Three' a) where
  foldMap f (Three' a b c) = f b <> f c

data Four' a b = Four' a b b b

instance Foldable (Four' a) where
  foldMap f (Four' a b c d) = f b <> f c <> f d

data GoatLord a = NoGoat | OneGoat a | MoreGoats (GoatLord a)
                (GoatLord a) (GoatLord a)

instance Foldable GoatLord where
  foldMap f NoGoat = mempty
  foldMap f (OneGoat a) = f a
  foldMap f (MoreGoats a b c) = foldMap f a <> foldMap f b <>
foldMap f c
```

Laboratorul 13: Monade - Introducere

Lucrați în fișierul `lab13.hs`, care conține și definiția monadei `Maybe`. Definiția este comentată deoarece monada `Maybe` este definită în `GHC.Base`

0. Înțelegeți funcționarea operațiilor monadice (`>=`) și `return`

```
return 3 :: Maybe Int
Just 3
(Just 3) >= (\ x -> if (x>0) then Just (x*x) else Nothing)
Just 9
```

1. Definim

```
pos :: Int -> Bool
pos x = if (x>=0) then True else False

fct :: Maybe Int -> Maybe Bool
fct mx = mx >= (\x -> Just (pos x))
```

- 2.1 Înțelegeți ce face funcția `fct`.

- 2.2 Definiți funcția `fct` folosind notația `do`.

2. Vrem să definim o funcție care adună două valori de tip `Maybe Int`

```
addM :: Maybe Int -> Maybe Int -> Maybe Int
addM mx my = undefined
```

Exemplu de funcționare:

```
addM (Just 4) (Just 3)
Just 7
addM (Just 4) Nothing
Nothing
addM Nothing Nothing
Nothing
```

- 2.1 Definiți `addM` prin orice metodă (de exemplu, folosind șabloane).

- 2.2 Definiți `addM` folosind operații monadice și notația `do`.

3. Să se treacă în notația `do` următoarele funcții:

```
cartesian_product xs ys = xs >= ( \x -> (ys >= \y-> return (x,y)))
```

```

prod f xs ys = [f x y | x <- xs, y<-ys]

myGetLine :: IO String
myGetLine = getChar >>= \x ->
    if x == '\n' then
        return []
    else
        myGetLine >>= \xs -> return (x:xs)

```

4. Să se treacă în notația cu secvențiere următoarea funcție:

```

prelNo noin = sqrt noin
ioNumber = do
    noin <- readLn :: IO Float
    putStrLn $ "Intrare\n" ++ (show noin)
    let noout = prelNo noin
    putStrLn $ "Iesire"
    print noout

```

5. Pentru următoarele exerciții lucrați cu fișierul `mWriter.hs`.

5.1. Fișierul `mWriter.hs` conține o definiție a monadei `Writer String` (puțin modificată pentru a compila fără opțiuni suplimentare):

```

newtype WriterS a = Writer { runWriter :: (a, String) }

```

5.1.1 Definiți funcțiile `logIncrement` și `logIncrement2` din curs și testați funcționarea lor.

5.1.2 Definiți funcția `logIncrementN`, care generalizează `logIncrement2`, astfel:

```

logIncrementN :: Int -> Int -> WriterS Int
logIncrement x n = undefined

```

Exemplu de funcționare:

```

runWriter $ logIncrementN 2 4
(6,"increment:2\nincrement:3\nincrement:4\nincrement:5\n")

```

5.2. Modificați definiția monadei `WriterS` astfel încât să producă lista mesajelor logate și nu concatenarea lor. Pentru a evita posibile confuzii, lucrați în alt fișier. Definiți funcția `logIncrementN` în acest context.

```

newtype WriterLS a = Writer {runWriter :: (a, [String])}

```

Exemplu de funcționare:

```

runWriter $ logIncrementN 2 4
(6,["increment:2","increment:3","increment:4","increment:5"])

```

6. Definim tipul de date

```

data Person = Person { name :: String, age :: Int }

```

6.1 Definiți funcțiile

```
showPersonN :: Person -> String
showPersonA :: Person -> String
```

care afișează “frumos” numele și vârsta unei persoane, după modelul

```
showPersonN $ Person "ada" 20
"NAME: ada"
```

```
showPersonA $ Person "ada" 20
"AGE: 20"
```

6.2 Folosind funcțiile definite la punctul 5.1, definiți funcția

```
showPerson :: Person -> String
```

care afișează “frumos” toate datele unei persoane, după modelul

```
showPerson $ Person "ada" 20
"(NAME: ada, AGE: 20)"
```

6.3 Folosind monada `Reader` (aveți implementarea instanțelor în fișierul `lab13.hs`), definiți variante monadice pentru cele trei funcții definite anterior, fără a folosi funcțiile definite anterior. Variantele monadice vor avea tipul

```
mshowPersonN :: Reader Person String
mshowPersonA :: Reader Person String
mshowPerson :: Reader Person String
```

Exemplu de funcționare:

```
runReader mshowPersonN $ Person "ada" 20
"NAME:ada"
```

```
runReader mshowPersonA $ Person "ada" 20
"AGE:20"
```

```
runReader mshowPerson $ Person "ada" 20
"(NAME:ada,AGE:20)"
```

LAB 13

```
addM :: Maybe Int -> Maybe Int -> Maybe Int
addM mx my = let Just r = mx; Just b = my in Just (r + b)

addM' :: Maybe Int -> Maybe Int -> Maybe Int
addM' mx my = do
  x <- mx
  y <- my
  return (x + y)

cartesianProduct xs ys = do
  x <- xs
  y <- ys
  return (x, y)

prod f xs ys = do
  x <- xs
  y <- ys
  return f x y

myGetLine :: IO String
myGetLine = do
  x <- getChar
  if x == '\n'
    then return []
    else do
      xs <- myGetLine
      return (x : xs)

prelNo = sqrt

ioNumber = readLn >>= (\noin -> (putStrLn $ "Intrare\n" ++ show
noin) >> let noout = prelNo noin in (putStrLn $ "Iesire") >> print
noout)
```

```

--- Monada Writer

newtype WriterS a = Writer {runWriter :: (a, String)}

instance Monad WriterS where
    return va = Writer (va, "")
    ma >>= k =
        let (va, log1) = runWriter ma
            (vb, log2) = runWriter (k va)
        in Writer (vb, log1 ++ log2)

instance Applicative WriterS where
    pure = return
    mf <*> ma = do
        f <- mf
        a <- ma
        return (f a)

instance Functor WriterS where
    fmap f ma = pure f <*> ma

tell :: String -> WriterS ()
tell log = Writer ((), log)

logIncrement :: Int -> WriterS Int
logIncrement x = do
    tell ("increment:" ++ show x ++ "\n")
    return (x + 1)

logIncrement2 :: Int -> WriterS Int
logIncrement2 x = do
    y <- logIncrement x
    logIncrement y

logIncrementN :: Int -> Int -> WriterS Int

```

```

logIncrementN x n = do
  if n > 0
  then do
    a <- logIncrement x
    logIncrementN a (n - 1)
  else return x

data Person = Person {name :: String, age :: Int}

showPersonN :: Person -> String
showPersonN (Person name age) = "NAME:" ++ show name

showPersonA :: Person -> String
showPersonA (Person name age) = "AGE:" ++ show age

showPerson :: Person -> String
showPerson (Person name age) = showPersonN (Person name age) ++ ",
" ++ showPersonA (Person name age)

newtype Reader env a = Reader {runReader :: env -> a}

instance Monad (Reader env) where
  return :: a -> Reader env a
  return x = Reader (\_ -> x)

  (>>=) :: Reader env a -> (a -> Reader env b) -> Reader env b
  ma >>= k = Reader f
    where
      f env =
        let a = runReader ma env
        in runReader (k a) env

instance Applicative (Reader env) where
  pure :: a -> Reader env a
  pure = return

```



```

(<*>) :: Reader env (a -> b) -> Reader env a -> Reader env b
mf <*> ma = do
    f <- mf
    a <- ma
    return (f a)

instance Functor (Reader env) where
    fmap :: (a -> b) -> Reader env a -> Reader env b
    fmap f ma = pure f <*> ma

ask :: Reader env env
ask = Reader id

mshowPersonN :: Reader Person String
mshowPersonN = do
    env <- ask
    return ("NAME: " ++ name env)

mshowPersonA :: Reader Person String
mshowPersonA = do
    env <- ask
    return ("AGE: " ++ show (age env))

mshowPerson :: Reader Person String
mshowPerson = do
    env <- ask
    return ("(NAME: " ++ name env ++ ",AGE: " ++ show (age env) ++
    ") ")

```

MODEL EXAMEN

```
data Point = Pt [Int]
    deriving Show

data Arb = Empty | Node Int Arb Arb
    deriving Show

class ToFromArb a where
    toArb :: a -> Arb
    fromArb :: Arb -> a

instance ToFromArb Point where
    toArb (Pt []) = Empty
    toArb (Pt (x:xs)) = Node x (toArb (Pt (filter (< x) xs))) (toArb
(Pt (filter (>= x) xs)))
    fromArb Empty = Pt []
    fromArb (Node x st dr) = let Pt l1 = fromArb st
                                Pt l2 = fromArb dr
                                in Pt (l1 ++ [x] ++ l2)

-- Subiectul 2
-- cu selectie
getFromIntervalSel a b list = [x | x<-list, x>=a, x<=b]

-- monade
getFromInterval a b list = do
    x<-list
    if a <= x && x <= b then return x else []

-- Subiectul 3
newtype ReaderWriter env a = RW {getRW :: env-> (a,String)}

instance Monad (ReaderWriter env) where
    return va = RW (\_ -> (va,""))
    ma >>= k = RW f
        where f env = let (va, str1) = getRW ma env
```

```
        (vb, str2) = getRW (k va) env
    in (vb, str1 ++ str2)

instance Applicative (ReaderWriter env) where
    pure = return
    mf <*> ma = do
        f <- mf
        va <- ma
        return (f va)

instance Functor (ReaderWriter env) where
    fmap f ma = pure f <*> ma
```