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]

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'
```

Funcțiile `head` și `tail` aparțin modulului standard **Prelude**.

- 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
```

Citiți mai mult despre **GHCi**:

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

Fișiere sursă

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:

```
myInt = 55555555555555555555555555555555555555555555555555555555555
double :: Integer -> Integer
double x = x+x
```

Fără încărca fișierul, încercați să calculați `double myInt`:

```
Prelude> double myInt
```

Observați mesajele de eroare. Acum încărcați fișierul folosind comanda ***load*** (:l) 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 reîncărcați folosind comanda ***reload*** (:r).

Puteți 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 asemanător) 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 sa încărcați 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` creaza 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 in 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 arbitar 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) functie cu 2 parametri care calculeaza 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

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*x^2+b*x+c`. Scrieți și signatura funcției (`poly :: ceva`).
2. Să se scrie o funcție `eeny` care întoarce "eeny" pentru input par și "meeny" pentru input impar. Hint: puteti folosi funcția `even` (puteți căuta pe https://hoogle.haskell.org/).

```haskell
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).

```haskell
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 in 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 usurează 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 in Haskell:

```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):

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

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

4. Numerele `tribonacci` sunt definite de ecuatția

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

Să se implementeze functia `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 urmatoarele 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 putin de `n` elemente, se intoarce 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 functiei.

Recursivitate pe Liste

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

Recursivitatea pe liste se bazeaza 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 injumătățește pe cele pare. De exemplu:

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

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

7. Exerciții: să se scrie urmatoarele funcții 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 și prototipul functiei.

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

c) `totalLen` - pentru o listă de șiruri de caractere, calculează suma lungimilor șirurilor care încep cu caracterul 'A'.

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

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 pozitii 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 in cifră. Cele 2 funcții se află în pachetul `Data.Char`.

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 adapatate ușor pentru valori oarecare folosind funcția `abs`.

1. Folosind numai metoda prin selecție definiți o funcție

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

atfel 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 sa le întoarcem ca rezultat.

Să presupunem că vrem să definim o funcție `aplica2` care primește ca argument o funcție `f` de tip `a -> a` și o valoare `x` de tip `a`, rezultatul fiind `f (f x)`. Tipul funcției `aplica2` este

```
aplica2 :: (a -> a) -> a -> a
```

Se pot da mai multe definiții:

```
aplica2 f x = f (f x)
aplica2 f = f . f
aplica2 =\f x -> f (f x)
aplica2 f = \x -> f (f x)
```

MAP

Funcția `map` are ca argumente o funcție de tip `a -> b` și o listă de elemente de tip `a`, rezultatul fiind lista elementelor de tip `b` obținute prin aplicarea funcției date pe fiecare element de tip `a`:

```
map :: (a -> b) -> [a] -> [b]
map f xs =[f x | x <- xs]
```

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 si 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 fillter (fara recursivitate sau selectie). Pentru fiecare functie scrieti si 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.

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 intreg 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` si `foldr` în locul recursiei.

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

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 calcuelază 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 urmatoarele tipuri de date ce reprezintă matrici cu linii de lungimi diferite:

```
data Linie = L [Int]
     deriving Show
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`.

```
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 aceeasi 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
```

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 afiseze 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
```

```haskell
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.

```haskell
expToArb :: Expr -> Tree
expToArb = undefined
```

2. Clasa Collection

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

```haskell
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. Adaugati definitii implicite (in functie de functiile celelalte) pentru

a. `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` instanta 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` instanta a clasei `Collection`.

Laboratorul 8: ADT. Clase de Tipuri

Exercițiul 1

Se dau următoarele tipuri de date ce reprezinta puncte cu numar variabil de coordonate intregi:

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

Arbori cu informația în frunze şi 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 sa fie afisată sub forma de tuplu.

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

-- Pt []
-- ()
```

b) Să se scrie o instanţă a clasei ToFromArb pentru tipul de date Punct astfel incat lista coordonatelor punctului sa 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 `perimeter` și `area`.

```
class GeoOps g where
  perimeter :: (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
-- 3.1415926535897793
```

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

Laboratorul 9: Logică propozițională - exerciții facultative

În acest laborator vom implementa funcții pentru a lucra cu logică propozițională în Haskell. Fie dată următoarea definiție:

```
type Nume = String
data Prop
  = Var Nume
  | F
  | T
  | Not Prop
  | Prop :|: Prop
  | Prop :&: Prop
  deriving Eq
infixr 2 :|:
infixr 3 :&:
```

Tipul `Prop` este o reprezentare a formulelor propoziționale. Variabilele propoziționale, precum `p` și `q` pot fi reprezentate va `Var "p"` și `Var "q"`. În plus, constantele booleene `F` și `T` reprezintă `false` și `true`, operatorul unar `Not` reprezintă negația ($\neg$; a nu se confunda cu funcția `not :: Bool -> Bool`) și operatorii (infix) binari `:|:` și `:&:`reprezintă disjuncția ($\vee$) și conjuncția ($\wedge$).

Exercițiul 1

Scrieți următoarele formule ca expresii de tip `Prop`, denumindu-le `p1`, `p2`, `p3`.

1. $(P \vee Q) \wedge (P \wedge Q)$

```
p1 :: Prop
p1 = (Var "P" :|: Var "Q") :&: (Var "P" :&: Var "Q")
```

2. $(P \vee Q) \wedge (\neg P \wedge \neg Q)$

```
p2 :: Prop
p2 = undefined
```

3. $(P \wedge (Q \vee R)) \wedge ((\neg P \vee \neg Q) \wedge (\neg P \vee \neg R))$

```
p3 :: Prop
p3 = undefined
```

Exercițiul 2

Faceți tipul `Prop` instanță a clasei de tipuri `Show`, înlocuind conectivele `Not`, `:|:` și `:&:` cu `~`, `|` și `&` și folosind direct numele variabilelor în loc de construcția `Var nume`.

```
instance Show Prop where
  show = undefined

test_ShowProp :: Bool
test_ShowProp =
    show (Not (Var "P") :&: Var "Q") == "((~P)&Q)"
```

Evaluarea expresiilor logice

Pentru a putea evalua o expresie logică vom considera un mediu de evaluare care asociază valori `Bool` variabilelor propoziționale:

```
type Env = [(Nume, Bool)]
```

Tipul `Env` este o listă de atribuiri de valori de adevăr pentru (numele) variabilelor propoziționale.

Pentru a obține valoarea asociată unui `Nume` în `Env`, putem folosi funcția predefinită `lookup :: Eq a => a -> [(a,b)] -> Maybe b`.

Deși nu foarte elegant, pentru a simplifica exercițiile de mai jos, vom definit o variantă a funcției `lookup` care generează o eroare dacă valoarea nu este găsită.

```
impureLookup :: Eq a => a -> [(a,b)] -> b
impureLookup a = fromJust . lookup a
```

O soluție mai elegantă ar fi să reprezentăm toate funcțiile ca fiind parțiale (rezultat de tip `Maybe`) și sa controlam propagarea erorilor.

Exercițiul 3

Definiți o funcție `eval` care dat fiind o expresie logică și un mediu de evaluare, calculează valoarea de adevăr a expresiei.

```
eval :: Prop -> Env -> Bool
eval = undefined

test_eval = eval  (Var "P" :|: Var "Q") [("P", True), ("Q", False)] == True
```

Satisfiabilitate

O formulă în logica propozițională este *satisfiabilă* dacă există o atribuire de valori de adevăr pentru variabilele propoziționale din formulă pentru care aceasta se evaluează la `True`.

Pentru a verifica dacă o formulă este satisfiabilă vom genera toate atribuirile posibile de valori de adevăr și vom testa dacă formula se evaluează la `True` pentru vreuna dintre ele.

Exercițiul 4

Definiți o funcție `variabile` care colectează lista tuturor variabilelor dintr-o formulă. *Indicație*: folosiți funcția `nub`.

```
variabile :: Prop -> [Nume]
variabile = undefined

test_variabile =
  variabile (Not (Var "P") :&: Var "Q") == ["P", "Q"]
```

Exercițiul 5

Dată fiind o listă de nume, definiți toate atribuirile de valori de adevăr posibile pentru ea.

```
envs :: [Nume] -> [Env]
envs = undefined

test_envs =
    envs ["P", "Q"]
    ==
    [ [ ("P",False)
      , ("Q",False)
      ]
    , [ ("P",False)
      , ("Q",True)
      ]
    , [ ("P",True)
      , ("Q",False)
      ]
    , [ ("P",True)
      , ("Q",True)
      ]
    ]
```

Exercițiul 6

Definiți o funcție `satisfiabila` care dată fiind o Propoziție verifică dacă aceasta este satisfiabilă. Puteți folosi rezultatele de la exercițiile 4 și 5.

```
satisfiabila :: Prop -> Bool
satisfiabila = undefined
```

```
test_satisfiabila1 = satisfiabila (Not (Var "P") :&: Var "Q") == True
test_satisfiabila2 = satisfiabila (Not (Var "P") :&: Var "P") == False
```

Exercițiul 7

O propoziție este validă dacă se evaluează la `True` pentru orice interpretare a varibilelor. O forumare echivalenta este aceea că o propoziție este validă dacă negația ei este nesatisfiabilă. Definiți o funcție `valida` care verifică dacă o propoziție este validă.

```
valida :: Prop -> Bool
valida = undefined

test_valida1 = valida (Not (Var "P") :&: Var "Q") == False
test_valida2 = valida (Not (Var "P") :|: Var "P") == True
```

Implicație și echivalență

Exercițiul 9

Extindeți tipul de date `Prop` și funcțiile definite până acum pentru a include conectivele logice `->` (implicația) și `<->` (echivalența), folosind constructorii `:->:` și `:<->:`.

Exercițiul 10

Două propoziții sunt echivalente dacă au mereu aceeași valoare de adevăr, indiferent de valorile variabilelor propoziționale. Scrieți o funcție care verifică dacă două propoziții sunt echivalente.

```
echivalenta :: Prop -> Prop -> Bool
echivalenta = undefined

test_echivalenta1 =
  True
  ==
  (Var "P" :&: Var "Q") `echivalenta` (Not (Not (Var "P") :|: Not (Var "Q")))
test_echivalenta2 =
  False
  ==
  (Var "P") `echivalenta` (Var "Q")
test_echivalenta3 =
  True
  ==
  (Var "R" :|: Not (Var "R")) `echivalenta` (Var "Q" :|: Not (Var "Q"))
```

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)
```

Laboratorul 11

Amintiți-vă clasele Functor si 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 valideaza un string, respectiv un intreg.

```
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) Sa se scrie o functie care construieste un element de tip `Cow` verificând numele, varsta si greutatea cu functiile 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 se scrie functia de la b) folosind `fmap` si `<*>`.

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 sir (sa 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 sir de caractere într-un element din tipul de date asociat, validând stringul cu functia `validateLength` (numele trebuie sa aiba 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 primeste ca argument două șiruri de caractere și formeaza un element de tip `Person` daca 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) si c) folosind `fmap` și `<*>`.

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 functia `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)
```

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` urmatoarele 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 urmatoarea 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` (aveti 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)"
```