Laboratorul 4: Evaluare leneșă, memoizare, agregarea ca operator universal

ATENTIE Fișierul `lab4literate.lhs` are extensia `.lhs` și este scris folosind "Literate Haskell" Fișierele `lab4.hs` și `lab4.pdf` sunt generate automat din fișierul `.lhs`

Puteti lucra direct în acest fisier (atât `ghci` cât și `ghc` înțeleg formatul literate Haskell) sau puteti lucra ca și până acum folosind fișierul generat `lab4.hs`.

Dacă hotărâți să folosiți fișierul `.lhs`, trebuie să respectați regula că liniile de cod incep cu `>`.

RECOMANDARE Înainte de a începe să lucrați exercițiile din acest laborator finalizați exercițiile din laboratoarele precedente.

În acest laborator vom exersa conceptele prezentate în cursul 3.

Dar mai întâi, să ne familiarizăm cu tipul de date `Natural`.

```haskell
module Lab4 where

import Numeric.Natural
```

Tipul de date Natural este un tip numeric, asemănător cu Integer în toate aspectele, inclusiv acela că poate reprezenta numere oricât de mari, cu excepția faptului că nu acceptă numere negative. Acest lucru este implementat printr-o excepție la rulare. Astfel, dacă rezultatul unei operații pe tipul Natural este negativ, va fi generată o excepție de tipul "arithmetic underflow". Puteți testa acest lucru evaluând expresii de forma `1 - 3 :: Natural` sau, mai simplu, `(-1)::Natural`.

În acest laborator, tipul `Natural` va fi folosit ca domeniu pentru indecșii unor șiruri recursive.

Tot pentru a putea opera cu indecși arbitrar de mari, vom folosi și o generalizare a operatorului `(!!)` pe liste, numită `genericIndex`.

```haskell
import Data.List (genericIndex)
```

`genericIndex` are aceeași signatură ca `(!!)`, însă al doilea argument (corespunzător indexului) nu mai este constrâns să fie `Int` ci doar din clasa de tipuri

`Integral` (din care fac parte și `Integer` și `Natural`).

Partea I. Funcția `foldr`.

Funcția `foldr` este folosită pentru agregarea unei colecții. O definiție intuitivă a lui `foldr` este:

```
foldr op unit [a1, a2, a3, ... , an] == a1 `op` a2 `op` a3 `op` .. `op` an `op` unit
```

Vom exersa folosirea funcției `foldr` scriind câteva funcții, mai întâi folosind recursie, apoi folosind foldr. Pentru fiecare pereche de funcții testați ca sunt echivalente folosind `QuickCheck`.

Exercițiul 1

(a) Scrieți o funcție recursivă care calculează produsul numerelor dintr-o listă.

```haskell
produsRec :: [Integer] -> Integer
produsRec = undefined
```

(b) Scrieți o funcție echivalentă care folosește `foldr` în locul recursiei.

```haskell
produsFold :: [Integer] -> Integer
produsFold = undefined
```

(c) Scrieți o proprietate `QuickCheck` că cele două funcții sunt echivalente

```haskell
prop_produs :: [Integer] -> Bool
prop_produs = undefined
```

Exercițiul 2

(a) Scrieți o funcție recursivă care verifică faptul că toate elementele dintr-o listă sunt `True`.

```haskell
andRec :: [Bool] -> Bool
andRec = undefined
```

(b) Scrieți o funcție echivalentă care folosește `foldr` în locul recursiei.

```haskell
andFold :: [Bool] -> Bool
andFold = undefined
```

(c) Scrieți o proprietate `QuickCheck` că cele două funcții sunt echivalente

```haskell
prop_and :: [Bool] -> Bool
prop_and = undefined
```

Exercițiul 3

(a) Scrieți o funcție recursivă care concatenează o listă de liste.

```haskell
concatRec :: [[a]] -> [a]
concatRec = undefined
```

(b) Scrieți o funcție echivalentă care folosește `foldr` în locul recursiei.

```
concatFold :: [[a]] -> [a]
concatFold = undefined
```

(c) Scrieți o proprietate `QuickCheck` că cele două funcții sunt echivalente

```
prop_concat :: Eq a => [[a]] -> Bool
prop_concat = undefined
```

Exercițiul 4

(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.

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

test_rmchars :: Bool
test_rmchars = rmCharsRec ['a'..'l'] "fotbal" == "ot"
```

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

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

(d) Scrieți o proprietate `QuickCheck` că cele două funcții sunt echivalente

```
prop_rmChars :: String -> String -> String
prop_rmChars = undefined
```

Partea II. Universalitatea funcției `foldr`

Lectură recomandată: slide-urile 20-32 din curs.

O posibilă definiție a funcției `foldr` ar putea fi cam așa:

```
foldr_ :: (a -> b -> b) -> b -> ([a] -> b)
foldr_ op unit = f
  where
    f []     = unit
    f (a:as) = a `op` f as
```

Această definiție ne dă și o indicație despre ce funcții recursive pe liste pot fi definite folosind `foldr` și cum putem să derivăm aceste definiții, astfel:

Dată fiind o funcție `f :: [a] -> b` pentru care putem descoperi `unit :: b` și `op :: a -> b -> b` astfel încât `f [] = unit` și `f (a:as) = op a (f as)`, atunci avem că `f = foldr op unit`.

Exemplul 1: Suma pătratelor elementelor impare

```
sumaPatrateImpare :: [Integer] -> Integer
sumaPatrateImpare []     = 0
sumaPatrateImpare (a:as)
  | odd a = a * a + sumaPatrateImpare as
  | otherwise = sumaPatrateImpare as
```

Aplicând algoritmul de mai sus, putem defini varianta ei folosind `foldr` în locul recursiei:

```
sumaPatrateImpareFold :: [Integer] -> Integer
sumaPatrateImpareFold = foldr op unit
  where
    unit = 0
    a `op` suma
      | odd a     = a * a + suma
      | otherwise = suma
```

Exemplul 2: funcția `map`

```
map_ :: (a -> b) -> [a] -> [b]
map_ f []     = []
map_ f (a:as) = f a : map_ f as
```

Aplicăm algoritmul de mai sus pentru a obține `map_ f`:

```
mapFold :: (a -> b) -> [a] -> [b]
mapFold f = foldr op unit
  where
    unit = []
    a `op` l = f a : l
```

Exemplul 3: funcția `filter`

```
filter_ :: (a -> Bool) -> [a] -> [a]
filter_ p [] = []
filter_ p (a:as)
  | p a       = a : filter_ p as
  | otherwise = filter_ p as
```

Aplicăm algoritmul de mai sus pentru a obține `filter_ p`:

```
filterFold :: (a -> Bool) -> [a] -> [a]
filterFold p = foldr op unit
```

```haskell
  where
    unit = []
    a `op` filtered
      | p a       = a : filtered
      | otherwise = filtered
```

Exercițiul 1

(a) Folosind doar recursie și funcții de bază, scrieți o funcție `semn` care ia ca argument o listă de întregi și întoarce un șir de caractere care conține semnul numerelor din intervalul `-9..9` (inclusiv), ignorându-le pe celelalte.

Indicație: `String = [Char]`

```haskell
semn :: [Integer] -> String
semn = undefined
```

```haskell
test_semn :: Bool
test_semn = semn [5, 10, -5, 0] == "+-0" -- 10 este ignorat
```

(c) Folosiți algoritmul descris mai sus pentru a defini funcția semn folosind `foldr` în locul recursiei

```haskell
semnFold :: [Integer] -> String
semnFold = foldr op unit
  where
    unit = undefined
    op = undefined
```

Funcții cu acumulatori

Lectură recomandată: Slide-urile 24-31 din curs

Un fenomen mai interesant se intâmplă atunci când funcția pe care o definim folosește argumente adiționale pentru stocarea de informații adiționale în timpul recursiei.

```haskell
medie :: [Double] -> Double
medie l = f l 0 0
  where
    f :: [Double] -> Double -> Double-> Double
    f [] n suma = suma / n
    f (a:as) n suma = f as (n + 1) (suma + a)
```

În acest caz deoarece parametrii funcției `f` sunt modificați în timpul recursiei, ei vor deveni parte a rezultatului, citind tipul lui `f` ca `f :: [a] -> (a -> a -> a)`. Astfel:

```haskell
medieFold :: [Double] -> Double
medieFold l = (foldr op unit l) 0 0  -- paranteze doar pentru claritate
  where
```

```haskell
    unit :: Double -> Double -> Double
    unit n suma = suma / n
    op :: Double -> (Double -> Double -> Double) -> (Double -> Double -> Double)
    (a `op` r) n suma = r (n + 1) (suma + a)
```

Exercițiul 2

(a) Folosind doar recursie și funcții de bază, scrieți o funcție `pozitiiPare` care ia ca argument o listă de întregi și întoarce lista pozițiilor elementelor pare.

```haskell
pozitiiPare :: [Integer] -> [Int]
pozitiiPare l = pozPare l 0 -- al doilea argument tine minte pozitia curenta
  where
    pozPare [] _ = []
    pozPare (a:as) i
      | even a = i:pozPare as (i+1)
      | otherwise = pozPare as (i+1)

test_pozitiiPare :: Bool
test_pozitiiPare = pozitiiPare [5, 10, -5, 0] == [1,3]
```

(b) [optional] Folosiți algoritmul descris mai sus pentru a defini funcția `pozitiiPare` folosind `foldr` în locul recursiei

```haskell
pozitiiPareFold :: [Integer] -> [Int]
pozitiiPareFold l = (foldr op unit l) 0
  where
    unit :: Int -> [Int]
    unit = undefined
    op :: Integer -> (Int -> [Int]) -> (Int -> [Int])
    (a `op` r) p = undefined
```

Exercițiul 3 [optional]

Definiți funcția `zipFold`, cu același comportament ca funcția `zip` folosind `foldr` în locul recursiei

```haskell
zipFold :: [a] -> [b] -> [(a,b)]
zipFold as bs = (foldr op unit as) bs
  where
    unit :: [b] -> [(a,b)]
    unit = undefined
    op :: a -> ([b] -> [(a,b)]) -> [b] -> [(a,b)]
    op = undefined
```

Partea III: Evaluarea leneșă

Introducere

Haskell este un limbaj leneș. Asta înseamnă că:

1. Evaluarea unei expresii este amânată până când devine necesară pentru continuarea execuției programului. In particular, argumentele unei funcții nu sunt evaluate înainte de apelul funcției.
2. Chiar și atunci când devine necesară pentru continuarea execuției programului, evaluarea se face parțial, doar atât cât e necesar pentru a debloca execuția programului.
3. Pentru a evita evaluarea aceluiaș argument al unei funcții de fiecare dată cănd e folosit în corpul funcției, toate aparițiile unei variabile sunt partajate, expandarea parțială a evaluării făcându-se pentru toate simultan.

Vom folosi în continuare o funcție intenționat definită ineficient pentru a testa ipotezele de mai sus. Funcția `logistic` simulează o lege de evoluție și a fost propusă ca generator de numere aleatoare.

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

Pentru simplificare vom lucra cu o variantă a ei în care `rate` și `start` au fost instanțiate:

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

Exercițiul 1

Pentru exercițiile de mai jos avem nevoie de o expresie a cărei execuție durează foarte mult timp, pentru a putea observa dacă este evaluată sau nu (și pentru a nu folosi `undefined`).

Testați că evaluarea funcției `logistic0` crește exponențial cu valoarea argumentului de intrare. Alegeți o valoare a acestuia `ex1` suficient de mare pentru a putea fi siguri dacă expresia se evaluează sau nu.

```
ex1 :: Natural
ex1 = undefined
```

Observație: chiar dacă nu rezolvați acest exercițiu, puteți observa dacă `logistic0 ex1` se evaluează deoarece `undefined` va arunca o excepție.

Amânarea evaluării expresiilor

Exercițiul 2

Evaluarea cărora dintre expresiile definite mai jos va necesita evaluarea expresiei `logistic0 ex1`?

Încercați să răspundeți singuri la întrebare, apoi testați în interpretor.

```
ex20 :: Fractional a => [a]
ex20 = [1, logistic0 ex1, 3]

ex21 :: Fractional a => a
ex21 = head ex20

ex22 :: Fractional a => a
ex22 = ex20 !! 2

ex23 :: Fractional a => [a]
ex23 = drop 2 ex20

ex24 :: Fractional a => [a]
ex24 = tail ex20
```

Evaluarea parțială a expresiilor

Exercițiul 3

Definim următoarele funcții auxiliare:

```
ex31 :: Natural -> Bool
ex31 x = x < 7 || logistic0 (ex1 + x) > 2

ex32 :: Natural -> Bool
ex32 x = logistic0 (ex1 + x) > 2 || x < 7
```

Evaluarea cărora dintre expresiile definite mai jos va necesita evaluarea expresiei `logistic0 (ex1 + x)`?

Încercați să răspundeți singuri la întrebare, apoi testați în interpretor.

```
ex33 :: Bool
ex33 = ex31 5

ex34 :: Bool
ex34 = ex31 7

ex35 :: Bool
ex35 = ex32 5

ex36 :: Bool
ex36 = ex32 7
```

Exercițiul 4

Evaluarea parțială a expresiilor este esențială în lucrul cu structuri (potențial) infinite de date.

(a)

Scrieți o funcție `findFirst` care ia ca argument un predicat și o listă de elemente și întoarce primul element din listă pentru care predicatul e adevărat.

```
findFirst :: (a -> Bool) -> [a] -> Maybe a
findFirst = undefined
```

(b)

Funcția `findFirst` poate fi folosită pentru a găsi primul număr natural care satisface o proprietate dată:

```
findFirstNat :: (Natural -> Bool) -> Natural
findFirstNat p = n
  where Just n = findFirst p [0..]
```

Observați că folosim o listă infinită. Dar, deoarece `findFirst` se oprește după ce găsește primul element, faptul că lista e infinită nu contează, dacă elementul este găsit (într-un timp rezonabil).

Dacă nu ați rezolvat punctul (a) puteți folosi funcția `find` din `Data.List`.

Calculați parte întreagă superioară din radical din `12347`:

```
ex4b :: Natural
ex4b = findFirstNat (\n -> n * n >= 12347)
```

(c) [optional]

Folosind punctul (b) ca inspirație, scrieți o funcție `inversa` care calculează "inversa" unei funcții monotone:

```
inversa :: Ord a => (Natural -> a) -> (a -> Natural)
inversa = undefined
```

Astfel, dată fiind `f :: Natural -> a` și `y :: a`, `inversa f x` reprezintă cel mai mic număr natural `n` pentru care `f n >= y`. Observați că, în particular, `inversa f (f x) == x`.

Partajarea subexpresiilor (subterm sharing)

Exercițiul 5

Rescrieți funcția logistic pentru a profita de partajarea expresiilor. (folosiți `where` sau `let` pentru a da un nume lui `f (n - 1)`.

Memoizare

Lectură obligatorie: slide-urile 6-11 de la curs

Lectură recomandată: HaskelWiki: Memoization (structuri avansate de memoizare, regăsire în timp logaritmic)

Funcția `memoize` definită mai jos "tabelează" funcția dată ca argument (reamintiți-vă din introducere `genericIndex`).

```
memoize :: (Natural -> a) -> (Natural -> a)
memoize f = genericIndex tabela
  where
    tabela = map f [0..]
```

Această funcție nu pare în sine foarte utilă, dar devine o construcție puternică atunci când e folosită în felul următor:

Dată fiind o funcție recursivă definită direct, cum ar fi funcția Fibonacci:

```
fibonacci :: Natural -> Natural
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)
```

O transformăm în varianta sa care folosește memoizarea, astfel:

```
fibonacciM :: Natural -> Natural
fibonacciM = memoize f
  where
    f 0 = 0
    f 1 = 1
    f n = fibonacciM (n - 1) + fibonacciM (n - 2)
```

De ce funcționează? Este un exemplu foarte bun al ideii partajării subexpresiilor combinată cu principiul evaluării leneșe. Astfel, `fibonacciM` este definit ca fiind `memoize f`, și toate referirile ulterioare la `fibonacciM` din definiția funcției `f` vor partaja această definiție, deci și tabela care memorează valorile lui `f`. De asemeni, elementele listei `map f [0..]` sunt calculate doar la nevoie, dar odată ce au fost calculate, ele sunt partajate de toate referirile la `fibonacciM`.

Exercițiul 6

Verificați că `fib` este exponențială, în timp ce `fibM` nu.

Exercitiul 7

Fie următoarea definiție a numerelor Catalan.

```
catalan :: Natural -> Natural
catalan 0 = 1
catalan n = sum [catalan i * catalan (n - 1 - i) | i <- [0..n-1]]
```

(a) Verificați exponențialitatea funcției `catalan`.
(b) Rescrieți `catalan` folosind memoizarea și testați creșterea performanței.

Exercițiul 8

Fie următoarea secvență Hofstadter-Conway

```
conway :: Natural -> Natural
conway 1 = 1
conway 2 = 1
conway n = conway (conway (n - 1)) + conway (n - conway (n - 1))
```

(a) Verificați exponențialitatea funcției `conway`.
(b) Rescrieți `conway` folosind memoizarea și testați creșterea performanței.

Exercițiul 9 (opțional)

Deși tehnica de memoizare prezentată mai sus elimină calcularea acelorași subexpresii de mai multe ori, ea necesită totuși o căutare în tabelă pentru fiecare apel recursiv.

Estimați complexitatea variantelor memoizate ale funcțiilor `fib`, `catalan` și `conway`.