Laboratorul 10: ADT, QuickCheck

Înainte de a începe să lucrați exercițiile din acest laborator, să vă amintiți noțiunile din cursurile 4, 6, 7 și 8.

Pentru a lucra exercițiile din acest laborator folosiți fișierul lab10.hs.

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)
```

0. Să se instanțieze clasa `Show` pentru tipul de date `Expr`, astfel încât să se afiseze mai simplu expresiile.
1. 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
```

2. Să se scrie o funcție `evalArb :: Tree -> Int` care evaluează o expresie determinând valoarea acesteia.

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

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

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

4. Să se instanțieze clasa `MySmallCheck` (Clasa tipurilor "mici" - ***cursul 8***) pentru tipul de date `Expr`, lista de valori conținând câteva expresii definite de voi.

```
class MySmallCheck a where
     smallValues :: [a]
     smallCheck :: ( a -> Bool ) -> Bool
     smallCheck prop = and [ prop x | x <- smallValues ]
```

5. Să se scrie un predicat care verifică faptul că evaluarea unei expresii este egala cu evaluarea arborelui asociat expresiei. Folosind funcția `smallCheck` să se verifice ca predicatul este adevărat pentru toate valorile `smallValues`.

```
checkExp :: Expr -> Bool
checkExp = undefined
```

Testare folosind QuickCheck

În Haskell avem la dispoziție o librărie care, în anumite situații, generează teste automate. Modificați fișierul lab10.hs astfel:

1. Importați modulul `Test.QuickCheck`, i.e. adăugați la începutul fișierului `import Test.QuickCheck`. Dacă acest modul nu este instalat, puteți instala folosind următoarele comenzi direct în terminal (cmd, powershell):

```
cabal update
cabal install QuickCheck
```

2. Definiți următoarele funcții care calculează dublul, triplul, respectiv, de cinci ori numărul dat ca parametru.

```
double :: Int -> Int
double = undefined
triple :: Int -> Int
triple = undefined
penta :: Int -> Int
penta = undefined
```

3. Observați în fișier următoarea funcție test:

```
test x = (double x + triple x) == (penta x)
```

4. Ce tip are funcția `test`?
5. În interpretor evaluați

```
*Main> quickCheck test
```

și observați rezultatul.

6. Scrieți un alt test care să verifice o proprietate falsa, verificați cu `quickCheck` și observați rezultatul.

7. Scrieți o funcție

```
myLookUp :: Int -> [(Int,String)]-> Maybe String
myLookUp = undefined
```

care caută un element întreg într-o listă de perechi cheie-valoare și întoarce valoarea gasită folosind un răspuns de tip `Maybe String`.

Scrieți un test

```
testLookUp :: Int -> [(Int,String)] -> Bool
testLookUp = undefined
```

care verifică faptul că funcția `myLookUp` are aceleași rezultate ca funcția `lookUp` predefinită.

QuickCheck cu constrângeri

Să încercăm să testăm că `myLookUp` este echivalentă cu `lookUp` predefinită doar pentru chei pozitive și divizibile cu 5.

```
-- testLookUpCond :: Int -> [(Int,String)] -> Property
-- testLookUpCond n list = n > 0 && n `div` 5 == 0 ==> testLookUp n list
```

Observați faptul că `testLookUpCond` are ca rezultat `Property`. Constrângerea din stânga „implicației” `==>` selecționeaza din valorile de intrare generate doar pe acelea care satisfac condiția dată. Evaluați în interpretor `quickCheck testLookUpCond` și observați rezultatul.

Testare pentru tipuri de date algebrice

Definiți o instanță a clasei `Arbitrary` pentru tipul de date `ElemIS` (instanțe similare în **cursul 8**)

```
data ElemIS = I Int | S String
     deriving (Show,Eq)
```

8. Definiți o funcție `myLookUpElem` care funcționează similar cu `lookUp`, doar ca funcționează pentru chei de tip întreg și valori de tip ElemIS.

```
myLookUpElem :: Int -> [(Int,ElemIS)]-> Maybe ElemIS
myLookUpElem = undefined
```

Scrieți un test

```
testLookUpElem :: Int -> [(Int,ElemIS)] -> Bool
testLookUpElem = undefined
```

Și rulați în consolă `quickCheck testLookUpElem`.

MicroHaskell

Fișierul microHaskell.hs conține un mini-limbaj funcțional, împreună cu semantica lui denotațională, așa cum afost definit în **cursul 7**. Definim comanda:

```
run :: Hask -> String
run pg = showV (hEval pg [])
```

Astfel, `run pgm` va întoarce rezultatul evaluării (rulării) programului `pgm`.

4.1) Scrieți mai multe programe si rulați-le pentru a vă familiariza cu sintaxa.

4.2) Adăugați operația de înmulțire pentru expresii, cu precedență mai mare decât a operației de adunare. Definiți semantica operației de înmulțire.

4.3) Folosind funcția `error`, înlocuiți acolo unde este posibil valoarea `VError` cu o eroare care să precizeze motivul apariției erorii.

4.4) Adăugați expresia `HLet Name Hask Hask` ca alternativă în definirea tipului `Hask`. Semantica acestei expresii este cea uzuală: `HLet x ex e` va evalua `e` într-un mediu în care `x` are valoarea lui `ex` în mediul curent. De exemplu, dacă definim

h1 = HLet “x” (HLit 3) ((HLit 4) :+: HVar “x”)

atunci `run h1` va întoarce “7”.