Laboratorul 3 – Implementarea unui Read-Eval-Print-Loop (REPL)

Scopul acestul laborator este de a trece prin pașii necesari implemntării unui REPL în Haskell.

Întrucât de partea de evaluare ne vom ocupa în laboratoarele următoare, de fapt azi vom implementa doar un read-print-loop:

1. vom citi de la prompt o comandă REPL / expresie,
   –vom implementa o funcție care va analiza sintactic șirul de intrare pentr a-l "traduce" într-o structură internă
2. vom implementa o funcție care formatează expresia pentru afișare din structura internă
   –vom afișa rezultatul
3. vom relua procesul (până la introducerea unei comenzi de terminare)

Crearea unui proiect pe GitHub (sau GitLab)

Exercițiu (cont GitHub/GitLab, dacă e cazul)

Creați-vă un cont pe GitHub/GitLab

Exercițiu (proiect nou)

Creați un proiect (repository) nou pe GitHub/GitLab

Setați limbajul

Exercițiu (cheie SSH – opțional, pe calculatorul propriu)

Configurați-vă o cheie SSH pentru accesul GitHub/GitLab.

Cum sa generezi o cheie SSH

Pentru a adauga o noua cheie in contul de GitHub, accesati meniul `Settings` -> `SSH and GPG Keys` -> `New SSH Key`.

Exercițiu (copie locală a proiectului)

Faceți o copie locală (`clone`) a proiectului

Posibil să trebuiască să vă instalați Git / GitHub for Windows
în acest caz, adăugați-o în directorul de surse al proiectului.

Analiză sintactică pentru expresii

Fie tipul de date al expresiilor:

```
module Exp where

import Numeric.Natural

newtype Var = Var { getVar :: String }
  deriving (Show)

data ComplexExp                          --  ComplexExp ::= "(" ComplexExp
")"
  = CX Var                               --          |   Var
  | Nat Natural                            --        |   Natural
  | CLam Var ComplexExp                  --          |   "\" Var "->"
ComplexExp
  | CApp ComplexExp ComplexExp           --          |   ComplexExp ComplexExp
  | Let Var ComplexExp ComplexExp        --          |   "let" Var ":="
ComplexExp "in"
  | LetRec Var ComplexExp ComplexExp     --          |   "letrec" Var
":=" ComplexExp "in"
  | List [ComplexExp]                    --          |   "[" {ComplexExp
","}* "]"
  deriving (Show)
```

Prin următoarele exerciții vom defini un parser care poate fi folosit pentru analiza sintactică a expresiilor prezentate în Laboratorul 1.

În acest scop ne va ajuta biblioteca de parsare pe care ați scris-o în laboratorul precedent.

```
module Parsing where

import Exp
import Lab2
import Control.Applicative (some, many, (<|>))
import Data.Char (isAlpha, isAlphaNum)
```

Pentru testare, putem defini următoarele funcții:

```
parseFirst :: Parser a -> String -> Maybe a
parseFirst p s
  = case apply p s of
      [] -> Nothing
      (a,_):_ -> Just a
```

Exercițiu (Identificatorii miniHaskell)

Definiți un analizor sintactic pentru identificatorii `miniHaskell`, care să accepte atât identificatorii (care încep cu literă și se continuă cu literă sau cifră), cât și operatorii din limbajul `Haskell` (care sunt formați cu caractere din mulțimea `~!@#$%^&*_+=|:<>.?/`).

```
var :: Parser Var
var = undefined
-- >>> parseFirst var "b is a var"
-- Just (Var {getVar = "b"})
```

Definiți un analizor sintactic pentru variabile ca λ-expresii (folosiți `var`)

```
varExp :: Parser ComplexExp
varExp = undefined
-- >>> parseFirst varExp "b is a var"
-- Just (CX (Var {getVar = "b"}))
```

Exercițiu (λ-abstracții)

În cele ce urmeaza vom presupune că există deja un analizor sintactic `expr :: Parser ComplexExp`, pe care vom încerca să îl definim recursiv. Pentru a putea testa definițiile de mai jos, puteți, pentru început să îl definiți pe `expr` ca `varExp`.

Folosind `expr` și `var` definiți un analizor sintactic care știe să recunoască o λ-expresie de forma `ComplexExp ::= "\" Var "->" ComplexExp`

```
lambdaExp :: Parser ComplexExp
lambdaExp = undefined
-- >>> parseFirst lambdaExp "\\x -> x"
-- Just (CLam (Var {getVar = "x"}) (CX (Var {getVar = "x"})))
```

Exercițiu (expresiile `let` și `letrec`)

Folosind `expr` și `var` definiți analizoare sintactice care știu să recunoască λ-expresii de forma:

- `ComplexExp ::= "let" Var ":=" ComplexExp "in"`

```
letExp :: Parser ComplexExp
letExp = undefined
-- >>> parseFirst letExp "let x := y in z"
-- Just (Let (Var {getVar = "x"}) (CX (Var {getVar = "y"})) (CX (Var
{getVar = "z"})))
```

- `ComplexExp ::= "letrec" Var ":=" ComplexExp "in"`

```
letrecExp :: Parser ComplexExp
letrecExp = undefined
-- >>> parseFirst letrecExp "letrec x := y in z"
-- Just (LetRec (Var {getVar = "x"}) (CX (Var {getVar = "y"})) (CX (Var
{getVar = "z"})))
```

Exercițiu (expresii liste)

folosind `brackets`, `commaSep` și `expr`, definiți un analizor sintactic care știe să recunoască o listă de expresii separate de virgulă, între paranteze pătrate:

```
listExp :: Parser ComplexExp
listExp = undefined
-- >>> parseFirst listExp "[a,b,c]"
-- Just (List [CX (Var {getVar = "a"}),CX (Var {getVar = "b"}),CX (Var
{getVar = "c"})])
```

Execițiu (alte mici expresii)

Definiți un analizor sintactic pentru

- numere naturale ca expresii (folosiți `natural`)

```
natExp :: Parser ComplexExp
natExp = undefined
-- >>> parseFirst natExp "223 a"
-- Just (Nat 223)
```

- expresii în paranteze ca expresii (folosiți `expr` și `parens`)

```
parenExp :: Parser ComplexExp
parenExp = undefined
-- >>> parseFirst parenExp "(a)"
-- Just (CX (Var {getVar = "a"}))
```

Exercițiu (Expresii de bază fără aplicație)

O expresie de bază este una din următoarele

- o expresie `letrec` sau `let`
- o λ-abstracție
- o variabilă (ca expresie)
- un număr natural
- o listă de expresii
- o expresie între paranteze

Folosind analizoarele sintactice definite mai sus precum și `natural` și `parens`, definiți un nou analizor lexical care acceptă toate definițiile de mai sus ca alternative.

```
basicExp :: Parser ComplexExp
basicExp = undefined
-- >>> parseFirst basicExp "[a,b,c]"
-- Just (List [CX (Var {getVar = "a"}),CX (Var {getVar = "b"}),CX (Var
{getVar = "c"})])
```

Toate expresiile (incluzând aplicația)

În sfârșit, o expresie este o succesiune de aplicații de expresii de bază. Totuși, în tipul `ComplexExp`, aplicația este construită doar din două expresii; de aceea, după ce veți obține lista de expresii corespunzătoare aplicărilor succesive (indicație: folosiți `some` și `basicExp`), va trebui să o transformați într-un arbore de aplicații binare, ținând cont de faptul că aplicația se grupează la stânga.

Astfel, din șirul de intrare `"x y z t"` va trebui să obțineți `CApp (CApp (CApp (Var "x") (Var "y")) (Var "z")) (Var "t")`

```
expr :: Parser ComplexExp
expr = varExp
-- >>> parseFirst expr "\\x -> x y z t"
-- Just (CLam (Var {getVar = "x"}) (CApp (CApp (CApp (CX (Var {getVar
= "x"})) (CX (Var {getVar = "y"}))) (CX (Var {getVar = "z"}))) (CX (Var
{getVar = "t"}))))
```

Și gata! am închis cercul și am obținut un analizor sintactic pentru tipul λ-expresiilor. Tot ce mai trebuie să facem, pentru a ne asigura că nu există spații înaintea expresiei în șirul de intrare, este să definim un analizor sintactic care le elimină înainte de a analiza expresia:

```
exprParser :: Parser ComplexExp
exprParser = whiteSpace *> expr <* endOfInput
-- >>> parseFirst exprParser "let x := 28 in \\y -> + x y"
-- Just (Let (Var {getVar = "x"}) (Nat 28) (CLam (Var {getVar = "y"})
(CApp (CApp (CX (Var {getVar = "+"})) (CX (Var {getVar = "x"}))) (CX
(Var {getVar = "y"})))))
```

Formatarea expresiilor

```
module Printing (showExp) where

import Exp
import Data.List (intercalate)
```

Exercițiu (formatare)

Implementați o funcție care formatează pentru afișare obiectele de tipul `ComplexExp`.

```
showVar :: Var -> String
showVar = undefined

showExp :: ComplexExp -> String
showExp = undefined
```

Interfața REPL (interacțiunea cu utilizatorul)

Exercițiu (Parser pentru comenzi REPL)

Creați un fișier nou, numit `REPLCommand.hs` care să conțină următoarea definiție de tip (inductiv):

```
module REPLCommand where

import Lab2
import Control.Applicative (many, (<|>))
```

```
data REPLCommand
  = Quit
  | Load String
  | Eval String
```

Scrieți (în același fișier) un mini-parser care dată fiind o comandă obține un obiect de tipul `REPLCommand`:

```
replCommand :: Parser REPLCommand
replCommand = undefined
```

Acest parser va trebui să înțeleagă următoarele comenzi:

- `:q` sau `:quit` pentru `Quit`
- `:l` sau `:load`, urmate de un șir de caractere pentru `Load`
- dacă nu e nici unul din cazurile de mai sus, tot șirul de intrare va fi pus într-un `Eval`.

Punem totul cap la cap

Exercițiu (programul principal)

Implementați repl-ul ca parte a funcției `main`.

- Afisează un prompt și citește o comandă
- parsează comanda într-un REPLComand
- în funcție de ea,
  - dacă e `Quit` termină programul
  - dacă e `Load`, deocamdată nu face nimic și reapelează `main`
  - dacă e `Eval`, atunci:
    - ⋆ transformați șirul de intrare într-un obiect de tip expresie
    - ⋆ transformați obiectul de tip expresie într-un șir de caractere prin formatare
    - ⋆ afișați rezultatul
    - ⋆ executați `main` din nou

Notă: În laboratoarele următoare vom insera funcția de evaluare între primele două de mai sus.

```
module Main where

import System.IO
```

```
import Lab2
import Exp
import Parsing
import Printing
import REPLCommand

main :: IO ()
main = undefined
```