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

Crearea unei proiect Haskell folosind `stack`

Exercițiu (proiect nou)

Folosiți `stack` pentru a crea un proiect nou

Exercițiu (dependințe parser)

Dacă vreți, puteți folosi biblioteca de parsare pe care ați scris-o în laboratorul precedent. în acest caz, adăugați-o în directorul de surse al proiectului.

Alternativ puteți folosi o bibliotecă externă de parsare, care are probabil câteva avantaje față de ce ați dezvoltat voi deja (e.g., mai mulți combinatori de parsare, parsarea comentariilor ca spații, etc.). Puteți alege între mai multe alternative, dar laboratorul de azi e scris având în minte parsec. Adăugați `parsec` la lista de dependințe a proiectului (`package.yaml`).

Exercițiu (dependințe readline)

O abilitate utilă pentru un REPL este aceea de a putea edita comanda (textul) introdusă la prompt si de a putea accesa istoricul acestor comenzi. Pentru a putea face acest lucru putem folosi o nouă bibliotecă. Din nou, puteți alege între mai multe alternative, dar laboratorul de azi e scris având în minte `isocline` deoarece este independentă de sistemul de operare și disponibilă pe `stackage`.

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

Dacă nu vreți să folosiți biblioteca de parsare pe care ați scris-o laboratorul trecut, puteți folosi funcția `makeTokenParser` din biblioteca `Parsec` pentru a obține un obiect de tip înregistrare GenTokenParser, prin care putem accesa combinatori de parsare similari tuturor celor definiți în laboratorul precedent.

Funcția `makeTokenParser` ia ca argument un obiect de tipul GenLanguageDef, care descrie lucruri cum ar fi:

- cum sunt formate comentariile (`commentStart`, `commentEnd`, `commentLine`, `nestedComments`)
- cum sunt formați identificatorii (`identStart`, `identStart`) și operatorii (`opStart`, `opLetter`)
- cuvintele (`reservedNames`) și operatorii (`reservedOpNames`) cheie
- dacă limbajul face diferența între litere mari și mici (`caseSensitive`)

Putem să creem un obiect de felul acesta de la zero, sau putem particulariza unul din obiectele predefinite în modul `Language`. Deoarece limbajul nostru e bazat pe Haskell, putem începe cu `haskellStyle` la care vom adăuga:

- cuvintele cheie din miniHaskell: `let`, `letrec`, `in`
- operatorii cheie din miniHaskell: `\`, `->`, `=`

Exercițiu (Analiză lexicală)

```
module Parsing where

import Exp
```

```
import Text.ParserCombinators.Parsec
import Text.ParserCombinators.Parsec.Language
    ( haskellStyle, LanguageDef )
import Text.ParserCombinators.Parsec.Token
import Control.Applicative (some)
```

Definiți un obiect `LanguageDef` pentru `miniHaskell` (bazat pe `haskellStyle`):

```
miniHaskellDef :: LanguageDef st
miniHaskellDef = undefined
```

Folosind acesta, putem defini un analizor lexical pentru miniHaskell:

```
miniHs :: TokenParser st
miniHs = makeTokenParser miniHaskellDef
```

Gata! Acum puteți folosi toți combinatorii pe care i-am definit săptămâna trecută, ba chiar mai buni și chiar mai mulți.

Pentru testare, putem defini următoarea funcție (folosind funcția parse):

```
testParse :: Parser a -> String -> a
testParse p s
  = case parse p "<input>" s of
      Left err -> error (show err)
      Right a -> a
```

Exercițiu (Identificatorii miniHaskell)

Definiți un analizor sintactic pentru identificatorii `miniHaskell`, care să accepte atât identificatorii, cât și operatorii din limbajul `Haskell`.

```
var :: Parser Var
var = undefined
-- >>> testParse var "b is a var"
-- Var {getVar = "b"}
```

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

```
varExp :: Parser ComplexExp
varExp = undefined
-- >>> testParse varExp "b is a var"
-- 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
-- >>> testParse lambdaExp "\\x -> x"
-- 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
-- >>> testParse letExp "let x := y in z"
-- Let (Var {getVar = "x"}) (CX (Var {getVar = "y"})) (CX (Var {getVar
= "z"}))
```

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

```
letrecExp :: Parser ComplexExp
letrecExp = undefined
-- >>> testParse letrecExp "letrec x := y in z"
-- 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
-- >>> testParse listExp "[a,b,c]"
```

```
-- 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
-- >>> testParse natExp "223 a"
-- Nat 223
```

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

```
parenExp :: Parser ComplexExp
parenExp = undefined
-- >>> testParse parenExp "(a)"
-- 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
-- >>> testParse basicExp "[a,b,c]"
-- 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
-- >>> testParse expr "\\x -> [x,y,z]"
-- CLam (Var {getVar = "x"}) (List [CX (Var {getVar = "x"}),CX (Var
{getVar = "y"}),CX (Var {getVar = "z"})])
```

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 miniHs *> expr <* eof
-- >>> testParse exprParser "let x := 28 in \\y -> + x y"
-- 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 Text.Parsec.String (Parser)
import Text.Parsec.Language (emptyDef, LanguageDef)
import qualified Text.Parsec.Token as Token
import Text.Parsec (anyChar)
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 System.Console.Isocline

import Exp
import Parsing
import Printing
import REPLCommand

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