Lambda Calculus Interpreter

Schelet: lambda-interpreter.zip

Deadline: miercuri 29 mai, ora 23:59

- Temele trebuie submise pe curs.upb.ro, în assignment-ul `Tema 3`.
- Pentru întrebări folosiți forum-ul dedicat de pe curs.upb.ro.

În cadrul acestei teme va trebui să realizezi un interpretor de expresii lambda în *Haskell*.

Remember: Lab 7. Lambda Calculus.

O să definim o expresie lambda cu ajutorul următorului **TDA**:

```
data Lambda = Var String
            | App Lambda Lambda
            | Abs String Lambda
```

În schelet, definiția **TDA**-ului conține și un constructor pentru macro-uri, pentru cerințele **1** și **2** îl puteți ignora, o să puteți lua punctaj maxim fără să faceți pattern matching pe el, o să fie introdus în cadrul cerinței **3**.

Variabilele sunt declarate de tipul `String`, pentru simplitate o să considerăm variabilă orice șir de caractere format numai din litere mici ale alfabetului englez.

1. Evaluation

Reminder:

- **redex** - o expresie reductibilă, **i.e.** are forma $(\lambda x.e1\ e2(\lambda x.e1\ e2)$
- **normal-form** - expresie care nu mai poate fi redusa (nu contine niciun **redex**)

Evaluarea unei *expresii lambda* constă în realizarea de $\beta\beta$-reducerii până ajungem la o expresie echivalentă în formă normală.
Un detaliu de implementare este că înainte de a realiza $\beta\beta$-reducerea, va trebui să rezolvăm posibilele ***coliziuni de nume***.
Dacă am încerca să reducem un **redex** fără a face substituții textuale există riscul de a pierde întelesul original al expresiei.
Spre exemplu **redex**-ul: $(\lambda x.\lambda y.(x\ y)\ \lambda x.y)(\lambda x.\lambda y.(x\ y)\ \lambda x.y)$, ar fi redus la: $\lambda y.(\lambda x.y\ y)\lambda y.(\lambda x.y\ y)$. Acest efect nedorit are denumirea intuitivă de *variable-capture*:
Variabila inițial liberă y a devenit legată după reducere.
Puteți observa că expresia și-a pierdut sensul original, pentru că **y**-ul liber din $\lambda x.y\lambda x.y$ e

acum *bound* de $\lambda y.\lambda y.$ din expresia în care a fost înlocuit.
Astfel, reducerea corectă ar fi: $\lambda a.(\lambda x.y\ a)\lambda a.(\lambda x.y\ a)$.
Pentru a detecta și rezolva *variable capture*, o să pregătim câteva funcții ajutătoare:
1.1. (*5p*) Implementați funcția auxiliară `vars` care returnează o listă cu toate `String`-urile care reprezintă variabile într-o expresie.
1.2. (*5p*) Implementați funcția auxiliară `freeVars` care returnează o listă cu toate `String`-urile care reprezintă variabile libere într-o expresie. (**notă**: dacă o variabilă este liberă în expresie în mai multe contexte, o să apară o singură dată în listă).
1.3. (*10p*) Implementați funcția auxiliară `newVars` care primește o listă de `String`-uri și intoarce cel mai mic `String` lexicografic care nu apare în listă (**e.g.** `new_vars ["a", "b", "c"]` o să întoarcă `"d"`).

1.4. (*5p*) Implementați funcția `isNormalForm` care verifică daca o expresie este în formă normală.

1.5. (*20p*) Implementați funcția `reduce` care realizează $\beta\beta$-reducerea unui **redex** luând în considerare și ***coliziunile de nume***. Funcția primește **redex**-ul 'deconstruit' și returnează expresia rezultată.

```
reduce :: String -> Lambda -> Lambda -> Lambda

reduce x e_1 e_2 = undefined

-- oriunde apare variabila x în e_1, este inlocuită cu e_2
```

Acum că putem reduce un **redex**, vrem să reducem o expresie la forma ei normală. Pentru asta trebuie să implementăm o strategie de alegere a **redex**-ului care urmează să fie redus, și să o aplicăm până nu mai există niciun **redex**. În această temă o să implementăm 2 strategii: **Normală** și **Aplicativă** (studiate la curs).

- **Normală**: se alege cel mai exterior, cel mai din stânga **redex**
- **Aplicativă**: se alege cel mai interior, cel mai din stânga **redex**

O să facem reducerea „step by step", implementăm o funcție care reduce doar următorul **redex** comform unei strategii. Apoi aplicăm acești pași până expresia rămasă este în formă normală.
1.6. (*10p*) Implementați funcția `normalStep` care aplică un pas de reducere după strategia Normală.
1.7. (*10p*) Implementați funcția `applicativeStep` care aplică un pas de reducere după strategia Aplicativă.
1.8. (*5p*) Implementați funcția `simplify`, care primeste o funcție de step și o aplică până expresia rămâne în formă normală, și întoarce o listă cu toți pași intermediari ai reduceri.

2. Parsing

Momentan putem să evaluăm expresii definite tot de noi sub formă de cod. Pentru a avea un interpretor funcțional, trebuie să putem lua expresii sub forma de șiruri de caractere și să le transformăm în **TDA**-uri (acest proces se numește **parsare**).

O gramatică pentru expresii lambda ar putea fi:

```
<lambda> ::= <variable> | '\' <variable> '.' <lambda> | (<lambda> <lambda>)

<variable> ::= <variable><alpha> | <alpha>

<alpha> ::= 'a' | 'b' | 'c' | ... | 'z'
```

2.1. (*40p*) Implementați funcția `parseLambda` care parsează un `String` și returnează o expresie

NU aveți voie să schimbați structura parserului, o soluție care nu se folosește de tipul de date `Parser` din schelet, nu o să fie punctată pentru cerințele de parsare.

Parserul care trebuie să îl implementați are definiția:

```
newtype Parser a = Parser {
    parse :: String -> Maybe(a, String)
}
```

Obervați că tipul care îl întoarce funcția de parsare este `Maybe(a, String)`, el întoarce `Nothing` dacă nu a putut parsa expresia sau `Just (x, s)` dacă a parsat `x`, iar din String-ul original a rămas sufixul `s`.

3. Steps towards a programming language

Folosind parserul și evaluatorul anterior, putem să evaluăm orice rezultat computabil, expresiile lambda fiind suficient de expresive, însă, cum probabil ați văzut la curs și laborator, este foarte greu să scrii astfel de expresii. Pentru a fi mai ușor de folosit, vrem să putem denumi anumite sub-expresii pentru a le putea refolosi ulterior. Pentru asta o să folosim conceptul de **macro**. Primul pas ar fi să extindem definiția unei expresii cu un constructor `Macro` care acceptă un `String` ca parametru (denumirea macro-ului). O să introducem și sintaxa: orice șir de caractere format numai din litere mari ale alfabetului englez si cifre e considerat un macro.

Câteva exemple de expresii cu macro-uri sunt:

TRUE$TRUE$
λx.FALSE$\lambda x.FALSE$
λx.(NOT λy.AND)$\lambda x.(NOT\ \lambda y.AND)$
Pentru a putea folosi macro-uri, trebuie să introducem noțiunea de ***context computațional***. Contextul în care evaluăm o expresie este pur și simplu un dicționar de nume de macro-uri și expresii pe care aceste nume le înlocuiesc. Astfel când evaluăm un macro, facem pur și simpu substituție textuală cu expresia găsită în dicționar.

În cazul în care nu găsim macro-ul în context, nu o să știm cum să evaluăm expresia, asa că am vrea să întoarcem o eroare. O să extindem tipul de date întors la `Either String [Lambda]` și o să întoarce `Left` în caz de eroare și `Right` în cazul în care evaluarea se termina cu succes.

3.1. (*15p*) Implementați funcția `simplifyCtx` care ia un context și o expresie care poate să conțină macro-uri, face substituțiile macro-urilor (sau returnează eroare dacă nu reușește) și evaluează expresia rezultată folosind strategia de step primită. (**Hint:** putem refolosi `simplify` ca să nu rescriem logica?)

Codul atunci când lucrezi cu `Maybe` sau `Either` poate să devina complicat dacă folosim `case`-uri pe toate variabilele, pentru a ușura lucrul cu ele există monade definite atât peste tipul de date `Maybe` cât și peste `Either`, poți folosi `do` notation să îți ușurezi viața.

funcția `lookup` este foarte utilă pentru lucrul cu dicționare (liste de perechi)

Ultimul pas ca să ne putem folosi de macro-uri e să găsim o metodă de a le defini. Pentru asta o sa definim conceptul de linie de cod:

```
data Line = Eval Lambda
          | Binding String Lambda
```

O linie de cod poate să fie ori o expresie lambda, ori o definiție de macro. Astfel daca o sa evaluam mai multe linii de cod, în expresii o sa ne putem folosi de macro-urile definite anterior.

3.2. (*5p*) Modificați parser-ul vostru astfel încât să parsați și expresii care conțin macro-uri.

3.3. (*5p*) Implementați funcția `parseLine` care să parseze o linie de cod, dacă găsește erori o să întoarcă o eroare (sub formă de `String`).

4.Default Library

Acum că avem un interpretor funcțional pentru calcul lambda, hai să definim și câteva expresii uzuale, ca să le putem folosi ca un context default pentru interpretorul nostru (un fel de standard library).

În fișierul `Default.hs` sunt deja definiti câțiva combinatori. Definiții restul expresiilor.

4.1. (*6p*) Definiți ca expresii lambda câteva macro-uri utile pentru lucrul cu Booleene (`TRUE`, `FALSE`, `AND`, `OR`, `NOT`, `XOR`).

4.2. (*4p*) Definiți ca expresii lambda câteva macro-uri utile pentru lucrul cu perechi (`PAIR`, `FIRST`, `SECOND`).

4.3. (*5p*) Definiti ca expresii lambda câteva macro-uri utile pentru lucrul cu numere naturale (`N0`, `N1`, `N2`, `SUCC`, `PRED`, `ADD`, `SUB`, `MULT`).

Pentru a fi punctați pentru cerința **4** este nevoie ca cerința **1** sa fie completată, pentru ca o sa ne folosim de `simplify` implementat de voi să testăm expresiile, deoarece vrem să testăm comportamentul lor, nu structura.

REPL

La finalul temei, puteți rula `runhaskell main.hs` pentru a vedea aplicația creată de voi :). O să pornească un **REPL** în care puteți scrie expresii lambda pentru a le evalua (main-ul se folosește de evaluarea normală implementată de voi), puteți crea binding-uri noi sau folosi binding-uri din contextul default creat.

Există și câteva comenzi utile:

- `:q` - pentru a ieși din **REPL**
- `:r` - pentru a sterge contextul, reluând contextul default
- `:ctx` - pentru a afișa contextul curent

Punctare

Tema are un punctaj total de **1.5p** din nota finală, împărțit pe subpuncte:

1. Evaluation
    - 5p - **1.1.** - aflarea variabilelor
    - 5p - **1.2.** - aflarea variabilelor libere
    - 10p - **1.3.** - generarea unei noi variabile
    - 5p - **1.4.** - verificarea formei normale
    - 20p - **1.5.** - reducerea unui redex
    - 10p - **1.6.** - step normal
    - 10p - **1.7.** - step aplicativ

    - 5p - **1.8.** - reducerea unei expresii step by step
2. Parsing
    - 40p - **2.1.** parsare
3. Steps towards a programming language
    - 15p - **3.1.** evaluarea unei expresii cu macro-uri
    - 5p - **3.2.** parsarea expresiilor cu macro-uri
    - 5p - **3.3.** parsarea liniilor de cod
4. Code
    - 6p - **4.1.** expresii boolene
    - 4p - **4.2.** expresii perechi
    - 5p - **4.2.** expresii numere naturale

După cum s-a anunțat la începutul semestrului, pentru studenții care au punctaj maxim pe toate 3 temele de pe parcursul semestrului, o să se echivaleze examenul din sesiune cu punctaj maxim.

Pot să existe depunctări de până la **1.5p** pentru implementări hardcodate sau plagiat.

Testing

Pentru testare puteți rula un set de teste unitare cu `runhaskell test.hs`. Pentru a testa doar o cerință, puteți da unul din argumentele [lambda | parser | binding | default] pentru a rula testele doar pentru cerința 1, 2, 3 sau 4.

Pentru fiecare test v-a aparea **PASSED** / **FAILED**, și în caz de **FAILED**, diferențele între rezultatul vostru și cel dorit.

Dacă implementarea unei funcții lipsește (sau apar alte erori) o să apară *„Error: …”* în loc de **PASSED** / **FAILED**.

Trimitere

Temele trebuie submise pe curs.upb.ro, în assignment-ul `Tema 3`.

În arhivă trebuie să se regăsească *cel puțin*:

- `Lambda.hs`
- `Parser.hs`
- `Binding.hs`
- `ID.txt` - acest fișier va conține o singură linie, formată din ID-ul unic al fiecărui student