Laboratorul 4 – Evaluarea λ-termenilor

În acest laborator vom defini mașinăria de bază pentru evaluarea λ-termenilor:

- variabile și variabile libere
- variabile noi (față de o mulțime)
- α-redenumirea variabilelor
- substituție
- β-reducție
- normalizare

Setup

În cazul în care continuați laboratoarele anterioare, trebuie să faceți câteva modificări: - din folderul `src`, copiați fișierele `Eval.hs`, `Sugar.hs` și `Exp.hs` în folderul `src` al proiectului vostru. Primele două fișiere sunt noi, iar `Exp.hs` este actualizat cu noi tipuri de date. - în modulul `Main`, importați cele două module noi `Eval` și `Sugar`. - pentru a verifica dacă totul este în regulă, rulați `stack build`.

Reducerea expresiilor la expresii (mai) simple

Mai întâi, vom începe prin a reduce tipul expresiilor la λ-calcul simplu (cu variabile indexate).

```
module Sugar where

import Exp
```

Deoarece va trebui să creăm variabile noi, cel mai simplu este ca o variabilă să aibă pe lângă un nume și o valoare numerică care ne spune a câta variabilă cu acel nume este:

```
data IndexedVar = IndexedVar
  { ivName :: String
  , ivCount :: Int
  } deriving (Eq, Read, Show)

makeIndexedVar :: String -> IndexedVar
makeIndexedVar name = IndexedVar name 0
```

Cu acest nou tip de variabile, putem defini tipul λ-expresiilor simple/standard, astfel:

```
data Exp
  = X IndexedVar
  | Lam IndexedVar Exp
  | App Exp Exp
  deriving (Show)
```

Exercițiu (de la `Var` la `IndexedVar`)

Implementați o funcție care transformă un obiect de tip `Var` într-unul de tip `IndexedVar` cu același nume și index `0`:

```
desugarVar :: Var -> IndexedVar
desugarVar = undefined

-- >>> desugarVar (Var "x")
-- IndexedVar {ivName = "x", ivCount = 0}
```

Exercițiu (de la `IndexedVar` la `Var`)

Implementați o funcție care transformă un obiect de tip `IndexedVar` într-unul de tip `Var` (pentru a-l putea afișa). Pentru variabilele indexate cu un index diferit de `0`, să zicem `n`, folosiți `_n` ca prefix după numele variabilei.

```
sugarVar :: IndexedVar -> Var
sugarVar = undefined

-- >>> sugarVar (IndexedVar "x" 0)
-- Var {getVar = "x"}

-- >>> sugarVar (IndexedVar "x" 3)
-- Var {getVar = "x_3"}
```

Pentru transformarea expresiilor complexe în expresii simple vom folosi câteva variabile predefinite:

```
consExp, nilExp, zeroExp, succExp, fixExp :: Exp
consExp = X (makeIndexedVar ":")  -- : :: a -> List a -> List a  list
constructor
```

```
nilExp = X (makeIndexedVar "Nil") -- Nil :: List a                empty
list
zeroExp = X (makeIndexedVar "Z")  -- Z :: Natural                 zero
succExp = X (makeIndexedVar "S")  -- S :: Natural -> Natural      successor
fixExp = X (makeIndexedVar "fix") -- fix :: (a -> a) -> a         fixpoint
fn.
```

Într-unul din laboratoarele următoare vom adăuga definiții pentru variabilele de mai sus, bazate pe Codările Church descrise la curs.

Exercițiu (de la `ComplexExp` la `Exp`)

Implementați următoarea funcție care transfomră un `ComplexExp` într-un `Exp`, astfel:

- `CApp`, `CLam` și `CX` se transformă în `App`, `Lam`, și respectiv `X`
- listele se transformă în aplicări repetate ale lui `:` peste `Nil`
  - de exemplu, `[a,b]` se transformă în `: a (: b Nil)`
- numerele naturale se transformă în aplicări repetate ale lui `S` peste `Z`
  - de exemplu, `3` se transformă în `S (S (S Z))`
- `let x = ex in e` se transformă în `(\x -> e) ex`
- `letrec f = ef in e` se transformă în `let f = fix (\f -> ef) in e`, deci în `(\f -> e)(fix (\f -> ef))`

```
desugarExp :: ComplexExp -> Exp
desugarExp = undefined

-- >>> desugarExp (CApp (CLam (Var "x") (CX (Var "y"))) (CX (Var "z")))
-- App (Lam (IndexedVar {ivName = "x", ivCount = 0}) (X (IndexedVar
{ivName = "y", ivCount = 0}))) (X (IndexedVar {ivName = "z", ivCount
= 0}))

-- >>> desugarExp (Nat 3)
-- App (X (IndexedVar {ivName = "S", ivCount = 0})) (App (X (IndexedVar
{ivName = "S", ivCount = 0})) (App (X (IndexedVar {ivName = "S", ivCount
= 0})) (X (IndexedVar {ivName = "Z", ivCount = 0}))))

-- >>> desugarExp (List [CX (Var "y"), CX (Var "x")])
-- App (App (X (IndexedVar {ivName = ":", ivCount = 0})) (X (IndexedVar
{ivName = "y", ivCount = 0}))) (App (App (X (IndexedVar {ivName = ":",
ivCount = 0})) (X (IndexedVar {ivName = "x", ivCount = 0}))) (X (IndexedVar
{ivName = "Nil", ivCount = 0})))
```

```
-- >>> desugarExp (Let (Var "y") (CX (Var "x")) (CX (Var "z")))
-- App (Lam (IndexedVar {ivName = "y", ivCount = 0}) (X (IndexedVar
{ivName = "z", ivCount = 0}))) (X (IndexedVar {ivName = "x", ivCount
= 0}))

-- >>> desugarExp (LetRec (Var "y") (CX (Var "x")) (CX (Var "z")))
-- App (Lam (IndexedVar {ivName = "y", ivCount = 0}) (X (IndexedVar
{ivName = "z", ivCount = 0}))) (App (X (IndexedVar {ivName = "fix",
ivCount = 0})) (Lam (IndexedVar {ivName = "y", ivCount = 0}) (X (IndexedVar
{ivName = "x", ivCount = 0}))))
```

Exercițiu (de la `Exp` la `ComplexExp`)

Implementați o funcție care traduce o expresie de tip `Exp` într-una de tip `ComplexExp`, traducând `App`, `Lam` și `X` în `CApp`, `CLam` și respectiv `CX`.

```
sugarExp :: Exp -> ComplexExp
sugarExp = undefined

-- >>> sugarExp (App (X (IndexedVar "x" 0)) (X (IndexedVar "y" 1)))
-- CApp (CX (Var {getVar = "x"})) (CX (Var {getVar = "y_1"}))

-- >>> sugarExp (App (Lam (IndexedVar {ivName = "x", ivCount = 0}) (X
(IndexedVar {ivName = "y", ivCount = 0}))) (X (IndexedVar {ivName =
"z", ivCount = 0})))
-- (CApp (CLam (Var "x") (CX (Var "y"))) (CX (Var "z")))
```

Variabile

```
module Eval where

import Exp
import Data.List ( union, delete )
```

Exercițiu (variabile)

Definiți o funcție care dat fiind un λ-termen, calculează mulțimea (ca listă) variabilelor care apar în termen.

```
vars :: Exp -> [IndexedVar]
vars = undefined
```

```
-- >>> vars (Lam (makeIndexedVar "x") (X (makeIndexedVar "y")))
-- [IndexedVar {ivName = "x", ivCount = 0},IndexedVar {ivName = "y",
ivCount = 0}]

-- >>> vars (App (Lam (IndexedVar {ivName = "x", ivCount = 0}) (X (IndexedVar
{ivName = "y", ivCount = 0}))) (X (IndexedVar {ivName = "z", ivCount
= 0})))
-- [IndexedVar {ivName = "x", ivCount = 0},IndexedVar {ivName = "y",
ivCount = 0},IndexedVar {ivName = "z", ivCount = 0}]

-- >>> vars (App (Lam (IndexedVar {ivName = "x", ivCount = 0}) (X (IndexedVar
{ivName = "x", ivCount = 0}))) (X (IndexedVar {ivName = "x", ivCount
= 0})))
-- [IndexedVar {ivName = "x", ivCount = 0}]
```

Exercițiu (variabile libere)

Definiți o funcție care dat fiind un λ-termen, calculează mulțimea (ca listă) variabilelor libere care apar în termen.

```
freeVars :: Exp -> [IndexedVar]
freeVars = undefined

-- >>> freeVars (Lam (makeIndexedVar "x") (X (makeIndexedVar "y")))
-- [IndexedVar {ivName = "y", ivCount = 0}]

-- >>> freeVars  (App (Lam (IndexedVar {ivName = "x", ivCount = 0})
(X (IndexedVar {ivName = "y", ivCount = 0}))) (X (IndexedVar {ivName
= "z", ivCount = 0})))
-- [IndexedVar {ivName = "y", ivCount = 0},IndexedVar {ivName = "z",
ivCount = 0}]

-- >>> freeVars (App (Lam (IndexedVar {ivName = "x", ivCount = 0}) (X
(IndexedVar {ivName = "x", ivCount = 0}))) (X (IndexedVar {ivName =
"x", ivCount = 0})))
-- [IndexedVar {ivName = "x", ivCount = 0}]

-- >>> freeVars (Lam (IndexedVar {ivName = "x", ivCount = 0}) (App (X
(IndexedVar {ivName = "x", ivCount = 0})) (X (IndexedVar {ivName = "x",
ivCount = 0}))))
-- []
```

Definiți o funcție care dată fiind o variabilă și un λ-termen, verifică dacă variabila apare liberă în termen.

```
occursFree :: IndexedVar -> Exp -> Bool
occursFree = undefined

-- >>> makeIndexedVar "x" `occursFree` Lam (makeIndexedVar "x") (X (makeIndexedVar
"y"))
-- False

-- >>> makeIndexedVar "y" `occursFree` Lam (makeIndexedVar "x") (X (makeIndexedVar
"y"))
-- True
```

Exercițiu (variabilă nouă)

Scrieți o funcție care dată fiind o variabilă și o listă de variabile produce o variabilă nouă care are același nume cu variabila dată dar e diferită de ea și de orice altă variabilă din lista dată.

```
freshVar :: IndexedVar -> [IndexedVar] -> IndexedVar
freshVar = undefined

-- >>> freshVar (makeIndexedVar "x") [makeIndexedVar "x"]
-- IndexedVar {ivName = "x", ivCount = 1}

-- >>> freshVar (makeIndexedVar "x") [makeIndexedVar "x", IndexedVar
{ivName = "x", ivCount = 1}, IndexedVar {ivName = "y", ivCount = 2}]
-- IndexedVar {ivName = "x", ivCount = 2}
```

Substituții

Exercițiu (redenumiri de variabile)

Scrieți o funcție care dată fiind o variabilă de redenumit, variabila care o redenumește și un λ-termen, redenumește toate aparițiile variabilei de redenumit cu cea care o redenumește.

```
renameVar :: IndexedVar -> IndexedVar -> Exp -> Exp
renameVar toReplace replacement = undefined
```

```
-- >>> renameVar (IndexedVar {ivName = "x", ivCount = 0}) (IndexedVar
{ivName = "z", ivCount = 0}) (App (Lam (IndexedVar {ivName = "x", ivCount
= 0}) (X (IndexedVar {ivName = "x", ivCount = 0}))) (X (IndexedVar {ivName
= "x", ivCount = 0})))
-- App (Lam (IndexedVar {ivName = "z", ivCount = 0}) (X (IndexedVar
{ivName = "z", ivCount = 0}))) (X (IndexedVar {ivName = "z", ivCount
= 0}))
```

Exercițiu (substituție)

Scrieți o funcție care definește substituția unei variabile cu un termen într-un λ-termen, implementând definiția matematică din curs (redenumind variabilele dacă există pericolul capturării).

```
substitute :: IndexedVar -> Exp -> Exp -> Exp
substitute toReplace replacement = undefined

-- >>> substitute (IndexedVar {ivName = "x", ivCount = 0}) (X (IndexedVar
{ivName = "z", ivCount = 0})) (App (Lam (IndexedVar {ivName = "x", ivCount
= 0}) (X (IndexedVar {ivName = "x", ivCount = 0}))) (X (IndexedVar {ivName
= "x", ivCount = 0})))
-- App (Lam (IndexedVar {ivName = "x", ivCount = 0}) (X (IndexedVar
{ivName = "x", ivCount = 0}))) (X (IndexedVar {ivName = "z", ivCount
= 0}))

-- >>> substitute (IndexedVar {ivName = "y", ivCount = 0}) (X (IndexedVar
{ivName = "x", ivCount = 0})) (App (Lam (IndexedVar {ivName = "x", ivCount
= 0}) (X (IndexedVar {ivName = "y", ivCount = 0}))) (X (IndexedVar {ivName
= "z", ivCount = 0})))
-- App (Lam (IndexedVar {ivName = "x", ivCount = 1}) (X (IndexedVar
{ivName = "x", ivCount = 0}))) (X (IndexedVar {ivName = "z", ivCount
= 0}))
```

Strategii de evaluare

Exercițiu (normalizare)

Implementați strategia normală de beta-reducție.

```
normalize :: Exp -> Exp
```

```
normalize = undefined

-- >>> normalize (X (makeIndexedVar "x"))
-- X (IndexedVar {ivName = "x", ivCount = 0})

-- >>> normalize (App (Lam (IndexedVar {ivName = "y", ivCount = 0})
(X (IndexedVar {ivName = "x", ivCount = 0}))) (App (Lam (IndexedVar
{ivName = "y", ivCount = 0}) (App (X (IndexedVar {ivName = "y", ivCount
= 0})) (X (IndexedVar {ivName = "y", ivCount = 0})))) (Lam (IndexedVar
{ivName = "y", ivCount = 0}) (App (X (IndexedVar {ivName = "y", ivCount
= 0})) (X (IndexedVar {ivName = "y", ivCount = 0}))))))
-- X (IndexedVar {ivName = "x", ivCount = 0})
```

Intergrare cu REPL

Exercițiu (REPL)

Importați modulele `Sugar` și `Eval` în `Main` și recrieți linia

```
Right e -> putStrLn (showExp e) >> main
```

astfel încât să facă următoarele:

- să transforme expresia din `ComplexExp` în `Exp`
- să normalizeze expresia
- să transforme expresia înapoi în `ComplexExp`
- să afișeze expresia
- să ruleze din nou `main`