Tema 3 Paradigme de Programare (`Haskell`)

> Link enunt: https://ocw.cs.pub.ro/ppcarte/doku.php?id=pp:2024:tema3

ABS (Abstractizarea, definirea unui functii)

In calculul lambda, o abstractizare este procesul de definire a unei functii.

Ea se compune din λ (lambda), urmat de un nume de variabila si o expresie.

```
Abs var expr
```

APP (Aplicatia, apel de functie)

O applicatie este procesul de aplicare a unei functii definite (prin abstractizre) la un argument.

```
App func valApp lamdaExpr1 lambdaExpr2
```

Aceasta s compune prin simpla plasare a argumetnului alaturi de fucntie. parseLine = undefined

1.1 `vars`

Implementați funcția auxiliară vars care returnează o listă cu toate String-urile care reprezintă variabile într-o expresie.

Daca expresia este:

- o `variabila`, atunci o va returna pe aceasta
- un `MACRO`, acesta se ve ignora
- o `APP` (aplicatie) intre doua expresii, atunci se vor returna variabilele din ele, eliminand duplicatele
- o `ABS` (abstractie) intre o variabila si o alta epresie, atunci se va returna variabila alaturi de toate variabilele expresie

> PS: functia `nub` elimina duplicatele dintr-o lista

1.2 `freeVar`

Implementați funcția auxiliară newVar care primește o listă de String-uri și intoarce cel mai mic String lexicografic care nu apare în listă.

Functia genereaza toate combinatiile posibile de litere mici ale alfabetului englez si gaseste prima variabila care nu e deja prezenta in lista data.

1.8 `simplify`

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.

Daca expresia este deja in forma normala, o vom returna pe aceasta, altfel:

- adaugam expresia curenta la pasii returnati
- simplifcam recursiv expresia, de data aceasta aplicand functia `step` pe expresie

4. Default Library

4.1 Booleans

`TRUE` = λx.λy.x

- primeste o expresie cu **doi parametri** si-l returneaza mereu pe **primul**
- `K`-Combinator (`Kestrel`)

`FALSE` = λx.λy.y

- primeste o expresie cu **doi parametri** si-l returneaza mereu pe **al doilea**
- `KI`-Combinator (`Kite`)

`NOT` = λx.λa.λb.((x b) a)

> `NOT` = λx.((x FALSE) TRUE)

- primeste o expresie cu **un parametru** si reutneaza FALSE daca este TRUE, altfel returneaza TRUE
- `C`-Combinator (`Cardinal`)

`AND` = λx.λy.((x y) x)

> `AND` = λx.λy.((x y) FALSE)

`OR` = λx.λy.((x x) y)

> `OR` = λx.λy.((x TRUE) y)

`XOR` = λx.λy.(NOT (OR x y))

`NAND` = λx.λy.(NOT (AND x y))

`NOR` = λx.λy.(NOT (OR x y))

4.2 Pereachi

`PAIR` = λa.λb. ((z a) b)

`FIRST` = λp. (p TRUE)

`SECOND` = λp. (p FALSE)

`PAIR`

- functie de ordin superior
- `V`-Combinator (`Vireo`)

4.3 Numere Naturale

`N0` = λf.λx.x

`N1` = λf.λx.(f x)

`N2` = λf.λx.(f (f x))

> `N0` este echivalentul lui `FALSE` daca folosim `alfa-reducere`

`SUCC` = λn.λf.λx.(f ((n f) x))

`ADD` = λn.λm.λf.λx.((n f) ((m f) x))

`MULT` = λm.λn.λf.(m (n f))

`PRED` = λn.λf.λx.((n λg.λh.(h (g f)) λu.x) λu.u)

Exemplu utilizare interpretor:

```
runhaskell main.hs
λ> :ctx
λ> \x.x
λx.x
λ> \x.(x y)
λx.(x y)
λ> \x.\y.y
λx.λy.y
λ> \x.((x FALSE) TRUE)
λx.((x λx.λy.y) λx.λy.x)

λ> M=\x.x
λ> M
λx.x
λ> MD5 = \x.\y.x
λ> MD5
λx.λy.x
λ> M = \x.\y.x
λ> M
λx.λy.x

λ> :q
```