Práctica 3
Cálculo Lambda sin Tipos

Favio E. Miranda Perea (favio@ciencias.unam.mx)
Diego Carrillo Verduzco (dixego@ciencias.unam.mx)
Pablo G. González López (pablog@ciencias.unam.mx)

Miércoles 19 de septiembre de 2018

Fecha de entrega: Miércoles 3 de octubre de 2018 a las 23:59:59.
El cálculo lambda consiste simplemente de tres términos y todas las combinaciones recursivas válidas de estos mismos.

- **Var** - Una variable
- **Lam** - Una abstracción lambda
- **App** - Una aplicación

Se dice que el cálculo lambda es el "lenguaje ensamblador" de la programación funcional, y sus distintas variaciones y extensiones forman la base de muchos compiladores funcionales para lenguajes como `Haskell`, `OCaml`, `Standard ML`, etc.

1 Sintaxis

Definiremos las expresiones del cálculo lambda del siguiente modo:

```
type Identifier = String

data Expr = Var Identifier
          | Lam Identifier Expr
          | App Expr Expr
```

Existen varias opciones de para representar la sintaxis de las expresiones lambda, sin embargo nosotros elegiremos la convención de `Haskell` que denota lambda por la diagonal inversa (\), el cuerpo con (`->`), y la aplicación con espacio encerrando las expresiones en un paréntesis (`(e e)`). Las variables serán nombradas únicamente con caracteres alfabéticos.
Ejemplo:

- **Notación Lógica:** $\lambda x.\lambda y.xy$

- **Notación de `Haskell`:** `\x -> \y -> (x y)`

1. (1 punto) Crea una instancia de la clase `Show` para las expresiones lambda utilizando la convención anterior. Ejemplo:

```
*Main> Lam "x" (Lam "y" (App (Var "x") (Var "y")))
\x -> \y -> (x y)
```

2 Sustitución y α-equivalencia

2.1 Sustitución

La evaluación de un término lambda $(\lambda x.e)a$ consiste en sustituir todas las ocurrencias libres de x en e por el argumento a. A este paso de la sustitución se le llama reducción. La sustitución se denota como $[x := a]$ y se define del siguiente modo:

$x[x := a] = a$
$y[x := a] = y$ si $x \neq y$
$ee'[x := a] = (e[x := a])(e'[x := a])$
$\lambda x.e[x := a] = \lambda x.e$
$\lambda y.e[x := a] = \lambda y.(e[x := a])$ si $x \neq y$ y $y \notin frVars(a)$
Definiremos el tipo sustitución del siguiente modo:

```
type Substitution = (Identifier, Expr)
```

2.2 α-equivalencia

La alfa equivalencia es la propiedad de cambiar la variable ligada, junto con todas sus ocurrencias libres dentro del cuerpo sin cambiar el significado de la expresión.

$\lambda x.e \equiv^{\alpha} \lambda y.(e[x := y])$
Implementa las siguientes funciones:

1. (0.5 puntos) `frVars`. Obtiene el conjunto de variables libres de una expresión.

```
frVars :: Expr -> [Identifier]
```

   Ejemplo:

```
*Main> frVars (App (Lam "x" (App (Var "x") (Var "y")))
(Lam "z" (Var "z")))
["y"]
*Main> frVars (Lam "f" (App (App (Var "f") (Lam "x"
(App (App (Var "f") (Var "x")) (Var "x")))) (Lam "x"
(App (App (Var "f") (Var "x")) (Var "x")))))
[]
```

2. (0.5 puntos) `lkVars`. Obtiene el conjunto de variables ligadas de una expresión.

```
lkVars :: Expr -> [Identifier]
```

Ejemplo:

```
*Main> lkVars (App (Lam "x" (App (Var "x") (Var "y")))
(Lam "z" (Var "z")))
["x", "z"]
*Main> (Lam "f" (App (App (Var "f") (Lam "x" (App (App
(Var "f") (Var "x")) (Var "x")))) (Lam "x" (App (App
(Var "f") (Var "x")) (Var "x")))))
["f", "x"]
```

3. (1 puntos) `incrVar`. Dado un identificador, si este no termina en número le agrega el sufijo 1, en caso contrario toma el valor del número y lo incrementa en 1.

```
incrVar :: Identifier -> Identifier
```

Ejemplo:

```
*Main> incrVar "elem"
"elem1"
*Main> incrVar "x97"
"x98"
```

4. (2 puntos) `alphaExpr`. Toma una expresión lambda y devuelve una α-equivalente utilizando la función `incrVar` hasta encontrar un nombre que no aparezca en el cuerpo.

```
alphaExpr :: Expr -> Expr
```

Ejemplo:

```
*Main> alphaExpr (Lam "x" (Lam "y" (App (Var "x") (Var "y"))))
\x1 -> \y -> (x1 y)
*Main> alphaExpr (Lam "x" (Lam "x1" (App (Var "x") (Var "x1"))))
\x2 -> \x1 -> (x2 x1)
```

5. (2 puntos) `subst`. Aplica la sustitución a la expresión dada.

```
subst :: Expr -> Substitution -> Expr
```

Ejemplo:

```
*Main> subst (Lam "x" (App (Var "x") (Var "y")))
("y", Lam "z" (Var "z"))
\x -> (x \z -> z)
*Main> subst (Lam "x" (Var "y")) ("y", Var "x")
\x1 -> x
```

3 β-reducción

Como se mencionó anteriormente la beta reducción es simplemente un paso de sustitución, remplazando la variable ligada por una expresión lambda por el argumento de la aplicación.

$(\lambda x.a)y \rightarrow^{\beta} a[x := y]$

4 Evaluación

La estrategia de evaluación de una expresión consistirá en aplicar la beta reducción hasta que ya no sea posible, usando las siguientes reglas:

$$\frac{t \rightarrow t'}{\lambda x.t \rightarrow \lambda x.t'}\ Lam$$

$$\frac{t_1 \rightarrow t_1'}{t_1 t_2 \rightarrow t_1' t_2}\ App1$$

$$\frac{t_1 \rightarrow t_1'}{(\lambda x.t) t_1 \rightarrow (\lambda x.t) t_1'}\ App2$$

$$\frac{}{(\lambda x.t)y \rightarrow^{\beta} t[x := y]}\ Beta$$

Implementa las siguientes funciones:

1. (2 puntos) `beta`. Aplica un paso de la beta reducción.

   ```
   beta :: Expr -> Expr
   ```

   Ejemplo:

   ```
   *Main> beta (App (Lam "x" (App (Var "x") (Var "y")))
   (Lam "z" (Var "z")))
   (\z -> z y)
   ```

2. (0.5 puntos) `locked`. Determina si una expresión esta bloqueada, es decir, no se pueden hacer más beta reducciones.

   ```
   locked :: Expr -> Bool
   ```

   Ejemplo:

   ```
   *Main> locked (Lam "s" (Lam "z" (Var "z")))
   True
   *Main> locked (Lam "x" (App (Lam "x" (Var "x")) (Var "y")))
   False
   ```

3. (0.5 puntos) `eval`. Evalúa una expresión lambda aplicando beta reducciones hasta quedar bloqueada.

```
eval :: Expr -> Expr
```

Ejemplo:

```
*Main> eval (App (L "n" (L "s" (L "z" (App (Var "s")
(App (App (Var "n") (Var "s")) (Var "z")))))) (L "s"
(L "z" (Var "z"))))
\s -> \z -> (s z)
*Main> eval (App (L "n" (L "s" (L "z" (App (Var "s")
(App (App (Var "n") (Var "s")) (Var "z")))))) (L "s"
(L "z" (App (Var "s") (Var "z")))))
\s -> \z -> (s (s z))
```

¡Suerte!