Universidad Nacional de Rosario
Facultad de Ciencias Exactas,
Ingeniería y Agrimensura

Lic. en Cs. de la Computación

Análisis de Lenguajes de Programación

Trabajo Práctico 2

Tomás Fernández De Luco F-3443/6
Ignacio Sebastián Moline M-6466/1

18 de octubre de 2018

Ejercicio 1

```
-----------------------------------------------------
-- Seccion 2 - Representacion de Lambda Terminos
-- Ejercicio 1
-----------------------------------------------------

  num :: Integer -> LamTerm
-- ~ El valor de cero es conocido.
  num 0 = Abs "s" (Abs "z" (LVar "z"))
-- ~ El sucesor de un numeral de Church tendra la misma estructura que
-- ~ su predecesor, teniendo la primer funcion aplicada nuevamente
  num n = let (Abs v1 (Abs v2 lt)) = num (n - 1) in
              Abs v1 (Abs v2 (App (LVar v1) lt))
```

Ejercicio 2

La conversión se realiza en la función convAux, la cual toma un Term y una lista de Strings que corresponde a los nombres de las variables con ocurrencia de ligadura encontradas hasta este momento, ordenadas desde la más cercana a la más lejana.

Si se quiere convertir una variable, se la buscará en el listado con la función lookFor. Si se la encuentra, la misma devuelve la cantidad de ligaduras previas hasta la primera ocurrencia de la variable en el listado. En caso contrario, se devuelve -1, con lo que se dejará a la variable como libre en la conversión de términos.

Si se quiere convertir una abstracción, se convertirá su término interior habiendo agregado la variable ligada por la abstracción en la cabecera del listado.

```
-----------------------------------------------------
-- Seccion 2 - Representacion de Terminos Lambda
-- Ejercicio 2: Conversion de Terminos
-----------------------------------------------------

lookFor :: String -> [String] -> Int -> Int
lookFor _ [] _ = -1
lookFor s (x:xs) dist = if s == x then dist else lookFor s xs (dist + 1)

convAux :: LamTerm -> [String] -> Term
convAux (LVar s) scope = case lookFor s scope 0 of
                           -1 -> Free s
                           n -> Bound n
convAux (App lt1 lt2) scope = (convAux lt1 scope) :@: (convAux lt2 scope)
convAux (Abs var lt) scope = Lam (convAux lt (var:scope))


conversion  :: LamTerm -> Term
conversion lt = convAux lt []
```

Ejercicio 3

Como la función shift es llamada en funciones de ejercicios posteriores, siempre con un argumento C = 0, se optó por definirla para que responda a ese comportamiento. La función que realmente realiza lo pedido en el ejercicio es shiftAux.

```
-----------------------------------
-- Seccion 3 - Evaluacion
-----------------------------------

shiftAux :: Term -> Int -> Int -> Term
shiftAux (Free s) _ _ = Free s
shiftAux (Bound n) c d = if n < c
                         then Bound n
                         else Bound (n + d)
shiftAux (lt1 :@: lt2) c d = (shiftAux lt1 c d) :@: (shiftAux lt2 c d)
shiftAux (Lam lt) c d = Lam (shiftAux lt (c + 1) d)

shift :: Term -> Int -> Term
shift lt d = shiftAux lt 0 d
```

Ejercicio 4

```
subst :: Term -> Term -> Int -> Term
subst (Free s) _ _  = Free s
subst x@(Bound n) lt' i = if n == i
                          then lt'
                          else x
subst (lt1 :@: lt2) lt' i = (subst lt1 lt' i) :@: (subst lt2 lt' i)
subst (Lam lt) lt' i = Lam (subst lt (shift lt' 1) (i + 1))
```

Ejercicio 5

Definimos la función betaRed, que realiza la β-reducción de un término según lo especificado en el enunciado. Para nuestra función de evaluación fuimos considerando cuáles de las reglas de reducción normal deberían aplicarse a cada uno de los valores de nuestro tipo de datos para términos.

```
betaRed :: Term -> Term -> Term
betaRed lt1 lt2 = shift (subst lt1 (shift lt2 1) 0) (-1)

lookInEnv :: Name -> NameEnv Term -> Either Term Term
lookInEnv s [] = Left (Free s)
lookInEnv s ((name, term):xs) = if s == name
                                then (Right term)
                                else lookInEnv s xs
eval :: NameEnv Term -> Term -> Term
-- ~ Una variable libre evalua a si misma si no se encuentra en el
-- ~ entorno, y a la evaluacion de su valor en caso contrario.
eval env (Free s) = case (lookInEnv s env) of
                        Left ft -> ft
                        Right t -> eval env t
-- ~ Una variable ligada evalua a si misma.
eval env (Bound n) = Bound n
-- ~ La evaluacion de una aplicacion de una variable libre a otro termino
-- ~ depende de si la variable se encuentra en el entorno.
eval env ((Free s) :@: lt2) = case (lookInEnv s env) of
                                Left ft -> ft :@: (eval env lt2)
                                Right t -> eval env (t :@: lt2)
-- ~ La aplicacion de una variable ligada a otro termino evalua a la
-- ~ aplicacion de la misma, a la evaluacion del termino.
eval env ((Bound n) :@: lt2) = (Bound n) :@: (eval env lt2)
-- ~ La aplicacion de una abstraccion a un termino evalua a la beta-reduccion,
-- ~ segun la regla E - APPABS.
eval env ((Lam lt1) :@: lt2) = eval env (betaRed lt1 lt2)
-- ~ En caso de que el primer termino de la aplicacion no corresponda a los
-- ~ casos anteriores, se lo debe evaluar. En funcion de si corresponde a una
-- ~ abstraccion u otro termino se aplicara E-APPABS o E-APP2 respectivamente.
eval env (lt1 :@: lt2) = case (eval env lt1) of
                            (Lam lt) -> eval env ((Lam lt) :@: lt2)
                            lt -> lt :@: (eval env lt2)
 -- ~ La evaluacion de una abstraccion sera la abstraccion de la evaluacion del
-- ~ termino interno, segun la regla E - ABS.
eval env (Lam lt) = Lam (eval env lt)
```

Ejercicio 6

```
-- not b = if b then false else true
def not = \b. b false true

-- sub x y = foldn y pred x
def sub = \x y. y pred x

-- greater x y = not (iszero (sub x y))
def greater = \x y. not (is0 (sub x y))

-- div x n = if (greater n x) then 0 else suc (div (sub x n) n)
def div = Y (\f x n. (greater n x) zero (suc (f (sub x n) n)))

-- log2 n = if (iszero (n - 1)) && not (iszero n) then 0 else 1 + log2(divi n 2)
def log2 = Y (\f n. (and (is0 (pred n)) (not (is0 n))) zero (suc (f (div n 2))))
```