Análisis de Lenguajes de Programación
Trabajo Práctico 2

Carlini Milton Vitali Franco

6 de Octubre de 2016

Parser.hs

```
module Parser where

import Text.Parsec.Prim
import Text.ParserCombinators.Parsec
import Text.Parsec.Token
import Text.Parsec.Language
import Control.Applicative hiding ((<|>))

import Common
import Untyped

----------------------------------------------
-- Seccon 2 - Representacion de Lambda Terminos
-- Ejercicio 1
----------------------------------------------
num :: Integer -> LamTerm
num 0 = Abs "s" (Abs "z" (LVar "z"))
num n = let (Abs _ (Abs _ w)) = num (n-1)
      in (Abs "s" (Abs "z" (App (LVar "s") w)))
-------------------------------------------------
-- Parser de Lambda Calculo (Gramatica Extendida)
-------------------------------------------------
totParser :: Parser a -> Parser a
totParser p = do
              whiteSpace untyped
              t <- p
              eof
              return t

-- Analizador de Tokens
untyped :: TokenParser u
untyped = makeTokenParser (haskellStyle { identStart = letter <|> char ’_’
                                    , opStart    = oneOf "=.\\"
                                    , opLetter = parserZero
                                    , reservedOpNames = ["=",".","\\"]
                                    , reservedNames = ["def"]
                                    })

-- Parser para comandos
parseStmt :: Parser a -> Parser (Stmt a)
parseStmt p = do
         reserved untyped "def"
         x <- identifier untyped
         reservedOp untyped "="
         t <- p
         return (Def x t)
   <|> fmap Eval p


parseTermStmt :: Parser (Stmt Term)
parseTermStmt = fmap (fmap conversion) (parseStmt parseLamTerm)

-- Parser para LamTerms
parseLamTerm :: Parser LamTerm
parseLamTerm =
   do bs <- many1 parseVarAbs
      return (foldl1 App bs) --foldl1 garantiza la asociacion izquierda de la aplicacion

```

```
parseVarAbs :: Parser LamTerm
parseVarAbs = do ide <- identifier untyped
                 return (LVar ide)
--lexeme toma un parser y devuelve otro que consume todos los espacios ( tabuladores y  newlines), a la derecha, del token parseado-}
         <|> do x <- lexeme untyped (decimal untyped)
                return (num (fromInteger x))
         <|> do reservedOp untyped "\\"
                vars <- many1 (identifier untyped)
                reservedOp untyped "."
                lt <- parseLamTerm
                return (foldr Abs lt vars)
         <|> parens untyped parseLamTerm

-- para testear el parser interactivamente.
testParser :: Parser LamTerm
testParser = totParser parseLamTerm
```

Untyped.hs

```
module Untyped where

import Control.Monad
import Data.List

import Common

--------------------------------------------------
-- Seccion 2 - Representacion de Terminos Lambda
-- Ejercicio 2: Conversion de Terminos
--------------------------------------------------
conversion :: LamTerm -> Term
conversion lt = conversion' lt []

conversion' :: LamTerm -> [(String, Int)] -> Term
conversion' (App t1 t2) xs = (conversion' t1 xs) :@: (conversion' t2 xs)
conversion' (Abs c t)   xs = Lam ((conversion' t) (add c xs))
conversion' (LVar v)    xs = case null $ lookUp v xs of
                              False  -> Bound (head (lookUp v xs))
                              True   -> Free (Global v)

add :: String -> [(String, Int)] -> [(String, Int)]
add c [] = [(c,0)]
add c ((c',i):xs)
   | c == c'   = (c,0):[(p,q+1) | (p,q)<-xs]
   | otherwise = (c', i+1) : (add c xs)

lookUp :: String -> [(String, Int)] -> [Int]
lookUp _ [] = []
lookUp c ((c',i):xs)
   | c == c'   = [i]
   | otherwise = lookUp c xs

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

vapp :: Value -> Value -> Value
vapp (VLam f) p     = f p
vapp (VNeutral p) q = VNeutral (NApp p q)

```

```
eval :: [(Name,Value)] -> Term -> Value
eval nvs t = eval' t (nvs,[])

eval' :: Term -> (NameEnv Value, [Value]) -> Value
eval' (Bound  p) (_,zs) = zs !! p
eval' (Free i) (ys,_)   = head $ [ q | (p,q)<-ys, p==i]
eval' (p :@: q) xs      = vapp (eval' p xs) (eval' q xs)
eval' (Lam p) (ys,zs)   = VLam (\x -> eval' p (ys,x:zs))

--------------------------------
-- Seccion 4 - Mostrando Valores
--------------------------------

quote  :: Value -> Term
quote v = quote' v 0

quote'  :: Value -> Int -> Term
quote' (VLam f) i               = Lam (quote' (f (VNeutral (NFree (Quote i)))) (i+1))
quote' (VNeutral (NFree v)) i  = case v of
                                    Global xs -> Free (Global xs)
                                    Quote k   -> Bound (i-k-1)
quote' (VNeutral (NApp n v)) i = (quote' (VNeutral n) i) :@: (quote' v i)
```

sqrt.lam

```
--------------------------------------------------------------------
-- Seccion 5 - Programando en Lamda Calculo
-- Ejercicio 6: Implementacion de la raiz cuadrada en lamda-calculo
--------------------------------------------------------------------

-- Resta en los naturales

def res = \n m . m pred n

-- sqrt

def sqrt' = Y (\f n i. is0 (res (mult i i) n) i (f n (pred i)))

def sqrt = \x . sqrt' x x
```