Untyped.hs

```
module Untyped where

import Data.List
import Text.ParserCombinators.Parsec
import Text.Parsec.Token
import Text.Parsec.Language

import Common

------------------------
-- Ejercicio 1
------------------------

num :: Integer -> LamTerm
num 0    =  Abs "s" (Abs "z" (LVar "z"))
num n = let (Abs "s" (Abs "z" t )) = num (n-1)
          in Abs "s" (Abs "z" (App (LVar "s") t) )

-----------------------
--- Sección 2 Parsers
-----------------------

lam :: [String] -> LamTerm -> LamTerm
lam [x] t = Abs x t
lam (x:xs) t = Abs x (lam xs t)

app :: [LamTerm] -> LamTerm
app xs = app' (reverse xs)

app' :: [LamTerm] -> LamTerm
app' [x] = x
app' (x:xs) = App (app' xs) x

var_int :: Parser LamTerm
var_int = do
        i <- identifier untyped
        return (LVar i)
       <|> do
          n <- natural untyped
          return (num n)

lamb = do reservedOp untyped "\\"
        vars <- many1 (identifier untyped)
        reservedOp untyped "."
        t <- parseLamTerm
        return (lam vars t)
```

```haskell
par = do reservedOp untyped "("
         t <- parseLamTerm
         reservedOp untyped ")"
         return t


-- Parser para LamTerms
parseLamTerm :: Parser LamTerm
parseLamTerm = do
               t <- many1 (var_int <|> lamb <|> par)
               return (app t)


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 '_',
                                          reservedNames = ["def"], opLetter = return ' ' })


-- 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)

-- conversion a términos localmente sin nombres
conversion  :: LamTerm -> Term
conversion t = conversion' t []

conversion' (LVar s) b = case (findv s b) of
                           Just i -> Bound i
                           Nothing -> Free (Global s)
conversion' (App t1 t2) b = (conversion' t1 b) :@: (conversion' t2 b)
conversion' (Abs s t) b = Lam (conversion' t (s:b))

findv :: String -> [String] -> Maybe Int
findv s b = findv' s b 0
```

```
findv' :: String -> [String] -> Int-> Maybe Int
findv' _ [] _ = Nothing
findv' s (x:xs) i = if s==x then Just i else findv' s xs (i+1)


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

-------------------------------
-- Sección 3
-------------------------------

vapp :: Value -> Value -> Value
vapp (VLam f) v = f v
vapp (VNeutral n) v = VNeutral (NApp n v)


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

eval' :: Term -> (NameEnv Value,[Value]) -> Value
eval' (Bound  ii)  d  =  (snd d) !! ii
eval' (Free s) d  = find' (fst d) s
eval' (t1:@:t2) d = vapp (eval' t1 d) (eval' t2 d)
eval' (Lam t) d = VLam (\v -> (eval' t (fst d, v:(snd d))))

find' :: Eq a => [(a,b)] -> a -> b -- asumo que aparece exactamente una tupla (Nombre, valor) por
cada Nombre que se busca
find' [(x,v)] _ = v
find' ((x,v):xs) y = if x==y then v else find' xs y


-------------------------------
-- Sección 4
-------------------------------

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

quote' :: Value -> Int -> Term
quote' (VNeutral vn) n = quote_neutral vn n
quote' (VLam f) n = Lam (quote' (f  (VNeutral (NFree (Quote n)))) (n+1) )


quote_neutral :: Neutral -> Int -> Term
quote_neutral (NFree (Global s)) _ = Free (Global s)
quote_neutral (NFree (Quote k)) i = Bound (i-k-1)
quote_neutral (NApp n v) c = (quote_neutral n c) :@: (quote' v c)
```

Sqrt.lam

```
-- resta
def resta =\n m s z . m pred n s z

-- menor igual
def mi = \ n m. (is0 (resta n m))

-- cua
def cua = \ n. mult n n

--sqrt
def sqrt = \ n. raiz n zero

--raiz
def raiz = Y (\f .\n m. (mi (cua m) n ) (f n (suc m)) (pred m))
```

Ejercicio 2

Gramática para el λ-cálculo

Term := Number | Identifier | '\' Identifier '.' Term | Term Term | '(' Term ')'

Le vamos agregando las convenciones para extendarla:

1) La aplicación asocia a izquierda

   NApp := Number | Identifier | '\' Identifier '.' Term | '(' Term ')'
   Term := NApp | Term Napp

2) La abstracción tiene el alcance más grande posible posible

   Abs := '\' Identifier '.' Term
   NApp := Number | Identifier | '(' Term ')'
   Term := Abs | Term' | Term' Abs
   Term' := NApp | Term' NApp

3) Varias abstracciones consecutivas bajo el mismo λ

   Abss := \' Identifier | Abss Identifier
   Abs := Abss '.' Term
   NApp := Number | Identifier | '(' Term ')'
   Term := Abs | Term' | Term' Abs
   Term' := NApp | Term' NApp

La gramática extendida para el λ-cálculo sin reducción a izquierda queda asi:

Abss := '\' Identifier | Abss Identifier
Abs := Abss '.' Term
NApp := Number | Identifier | '(' Term ')'
Term := Abs | Term' | Term' Abs
Term' := NApp T
T := ε | Napp T

Aclaración: En la gramática extendida sin reducción a la izquierda la aplicación no aoscia a izquierda.