

Parse.y

```
{
module Parse where
import Common
import Data.Maybe
import Data.Char

}

%monad { P } { thenP } { returnP }
%name parseStmt Def
%name parseStmts Defs
%name term Exp

%tokentype { Token }
%lexer {lexer} {TEOF}

%token
  '='    { TEquals }
  ':'    { TColon }
  '\\'   { TAbs }
  '.'    { TDot }
  '('    { TOpen }
  ')'    { TClose }
  '->'   { TArrow }
  VAR    { TVar $$ }
  TYPE   { TType }
  DEF    { TDef }
  LET    { TLet }
  IN     { TIn }
  AS     { TAs }
  ','    { TComa }
  UNIT   { Tunit }
  TUNIT  { TTUnit }
  FST    { TFst }
  SND    { TSnd }
  REC    { TR }
  SUC    { TSuc }
  ZERO   { TZero }
  NAT    { TTNat }

%right VAR
%left '='
%right '->'
%right '\\' '.' LET IN
%left AS
%right REC
%right SUC
%right SND FST
```

%%

```
Def    : Defexp                { $1 }
      | Exp                    { Eval $1 }
Defexp : DEF VAR '=' Exp       { Def $2 $4 }

Exp    :: { LamTerm }
      : LET VAR '=' Exp IN Exp { LetIn $2 $4 $6 }
      | '\\' VAR ':' Type '.' Exp { Abs $2 $4 $6 }
      | Exp AS Type              { Asc $1 $3 }
      | FST Exp                  { Pri $2 }
      | SND Exp                  { Seg $2 }
      | SUC Exp                  { Succ $2 }
      | REC Atom Atom Atom      { Rec $2 $3 $4 }
      | NAbs                     { $1 }

NAbs   :: { LamTerm }
      : NAbs Atom               { App $1 $2 }
      | Atom                     { $1 }

Atom   :: { LamTerm }
      : VAR                     { LVar $1 }
      | '(' Exp ')'             { $2 }
      | UNIT                    { LUnit }
      | ZERO                    { Cero }
      | '(' Exp ',' Exp ')'      { Par $2 $4 }

Type   : TYPE                  { Base }
      | Type '->' Type          { Fun $1 $3 }
      | '(' Type ')'            { $2 }
      | TUNIT                   { TUnit }
      | '(' Type ',' Type ')'    { TPar $2 $4 }
      | NAT                     { TNat }

Defs   : Defexp Defs           { $1 : $2 }
      |                          { [] }
{
```

```
data ParseResult a = Ok a | Failed String
    deriving Show
type LineNumber = Int
type P a = String -> LineNumber -> ParseResult a
```

```
getLineNo :: P LineNumber
getLineNo = \s l -> Ok l
```

```
thenP :: P a -> (a -> P b) -> P b
m `thenP` k = \s l -> case m s l of
    Ok a    -> k a s l
    Failed e -> Failed e
```

```
returnP :: a -> P a
returnP a = \s l-> Ok a
```

```
failP :: String -> P a
failP err = \s l -> Failed err
```

```
catchP :: P a -> (String -> P a) -> P a
catchP m k = \s l -> case m s l of
    Ok a    -> Ok a
    Failed e -> k e s l
```

```
happyError :: P a
happyError = \s i -> Failed $ "Línea "++(show (i::LineNumber))++": Error de parseo\n"++(s)
```

```
data Token = TVar String
            | TType
            | TDef
            | TAbs
            | TDot
            | TOpen
            | TClose
            | TColon
            | TArrow
            | TEquals
            | TEOF
            | TLet
            | TIn
            | TAs
            | Tunit
            | TTUnit
            | TComa
            | TFst
            | TSnd
            | TTNat
            | TZero
            | TR
            | TSuc
            deriving Show
```

```
-----
lexer cont s = case s of
    [] -> cont TEOF []
    ('\n':s) -> \line -> lexer cont s (line + 1)
    (c:cs)
        | isSpace c -> lexer cont cs
        | isAlpha c -> lexVar (c:cs)
    ('-':('-':cs)) -> lexer cont $ dropWhile ((/=) '\n') cs
    ('{':('-':cs)) -> consumirBK 0 0 cont cs
    ('-':('{}':cs)) -> \line -> Failed $ "Línea "++(show line)++": Comentario no abierto"
    ('-':('>':cs)) -> cont TArrow cs
    ('\':cs)-> cont TAbs cs
```

```

('.:cs) -> cont TDot cs
('(:cs) -> cont TOpen cs
('):cs) -> cont TClose cs
('.:cs) -> cont TColon cs
('=:cs) -> cont TEquals cs
(',: cs) -> cont TComa cs
('0':cs) -> cont TZero cs
unknown -> \line -> Failed $ "Línea "++(show line)++": No se puede reconocer "++
(show $ take 10 unknown)++ "..."
where lexVar cs = case span isAlpha cs of
    ("B",rest) -> cont TType rest
    ("def",rest) -> cont TDef rest

    ("let", rest) -> cont TLet rest
    ("in", rest) -> cont TIn rest
    ("as",rest) -> cont TAs rest
    ("unit", rest) -> cont Tunit rest
    ("Unit",rest) -> cont TTUnit rest
    ("fst", rest) -> cont TFst rest
    ("snd", rest) -> cont TSnd rest
    ("suc", rest) -> cont TSuc rest
    ("Nat",rest) -> cont TTNat rest
    ("R", rest) -> cont TR rest
    (var,rest) -> cont (TVar var) rest
consumirBK anidado cl cont s = case s of
    ('-':('':cs)) -> consumirBK anidado cl cont $ dropWhile
((/=) '\n') cs
    ('{':('':cs)) -> consumirBK (anidado+1) cl cont cs
    ('-':('}')':cs)) -> case anidado of
        0 -> \line -> lexer cont cs (line+cl)
        _ -> consumirBK (anidado-1) cl
cont cs
    ('\n':cs) -> consumirBK anidado (cl+1) cont cs
    (_:cs) -> consumirBK anidado cl cont cs

stmts_parse s = parseStmts s 1
stmt_parse s = parseStmt s 1
term_parse s = term s 1
}

```