Universidad Nacional de La Plata
Facultad de Informática
Programación Funcional

LIS Parser: un parser para el lenguaje imperativo simple LIS.

Autor
Ary Pablo Batista

9 de noviembre de 2014

Resumen

Este trabajo documenta el desarrollo de una biblioteca de parsers y un parser para el lenguaje de programación LIS, lenguaje creado por Fidel (Pablo E. Martínez López) para la enseñanza de tipos de datos algebraicos recursivos en la materia de Programación Funcional que se dicta en la Universidad Nacional de La Plata y la Universidad Nacional de Quilmes.

Índice

1. Introducción

A partir de la construcción de procesadores capaces de ejecutar conjuntos de instrucciones de bajo nivel, constituidas por cálculos aritméticos y lógicos simples, los hombres de ciencia comenzaron a buscar formas de representar sus complejos algoritmos, expresados en los lenguajes de programación primitivos o sistemas formales, mediante estas instrucciones de manera que puedan ser calculadas automáticamente. Construyeron estas representaciones a través de un conjunto de reglas de transformación ligadas a cada uno de los elementos que componen el lenguaje, proceso al que llamaron análisis sintáctico.

Este trabajo aborda el diseño de un analizador sintáctico (o *parser*[1]) para el lenguaje de programación LIS creado por Fidel (Pablo E. Martínez López) para la enseñanza de tipos de datos algebraicos recursivos en la materia de Programación Funcional que se dicta en la Universidad Nacional de La Plata y la Universidad Nacional de Quilmes.

2. Objetivo

El objetivo de este trabajo es el desarrollo de una biblioteca para el análisis sintáctico de lenguajes y un analizador sintáctico para el lenguaje de programación LIS y su documentación mediante este escrito, con la esperanza de que sea utilizado en el marco de la materia como material adicional para la enseñanza de la técnica de análisis sintáctico aquí empleada.

3. El lenguaje imperativo simple LIS

El lenguaje de programación LIS[2][3] es un lenguaje imperativo simple que define un conjunto de comandos reducidos y expresiones booleanas y numéricas. Los comandos presentes en LIS son la instrucción sin efectos (`skip`), la asignación de expresiones numéricas (`:=`), la alternativa condicional (`if`) y la repetición condicional (`while`). Las expresiones numéricas están presentes en las asignaciones y en las comparaciones mientras que las expresiones booleanas solo pueden ser utilizadas como condiciones de la alternativa condicional o la repetición condicional. A continuación se muestra un programa LIS que calcula el factorial de un número.

[1] *parser* es una palabra del inglés que significa analizador sintáctico.

[2] Definido por Pablo E. Martínez López en su clase "Tipos de Datos: Tipos Recursivos" que forma parte de la materia de Programación Funcional que dicta en la Universidad Nacional de La Plata y la Universidad Nacional de Quilmes.

[3] El lenguaje presentado es una variación de LIS que renombra el tipo de dato algebraico y constructor `P` a `Program`, el constructor `Bloque` a `Block`, el tipo de dato algebraico `Comand` a `Command`, el constructor `RelOp` a `Cmp` y el constructor `Vble` a `Variable` siendo el antiguo alias de tipo `Variable` renombrado a `VarName`.

```
-- Código LIS
program
 {
   -- Parámetros
   n  := 6

   -- Cuerpo
   a  := n;
   fn := 1;
   while (a > 0)
    {
      fn := a * fn;
      a := a - 1
    }

    -- Retorno
    return := fn
 }
```

Este programa está organizado en tres secciones, cada una indicada por un comentario. La sección indicada como *Parámetros* contiene aquellas variables que son parámetros utilizados en la sección indicada como *Cuerpo* que contiene el algoritmo. Por último la sección indicada como *Retorno* contiene todas las variables que representan el resultado del algoritmo. Esta organización se mantendrá en todos los programas LIS que se presenten en este trabajo.

3.1. Representación de programas LIS mediante tipos de datos algebraicos

Para la realización de distintas manipulaciones internas de un programa LIS, creamos una representación para estos programas a partir de la definición de los tipos de datos algebraicos que se presentan a continuación.

Un programa LIS se define como un tipo de dato algebraico `Program` construido a partir del constructor `Program` y un bloque.

```
data Program = Program Block
```

El tipo `Block` es un alias para el tipo lista de comandos.

```
type Block = [Command]
```

Un comando puede ser una instrucción sin efectos `Skip`, una asignación numérica `Assign`, una alternativa condicional `If` o una repetición condicional `While`.

```
data Command = Skip
             | Assign VarName NExp
             | If BExp Block Block
             | While BExp Block
```

```
type VarName = String
```

Tanto la alternativa condicional como la repetición condicional hacen uso de expresiones booleanas (`BExp`) tanto atómicas, como una constante (`BCte`) `False` o `True`, como compuestas, expresadas utilizando los operadores lógicos *y* (`And`), *o* (`Or`) y *no* (`Not`) o los operadores relacionales (`ROp`) *igual* (`Equal`), *distinto* (`NotEqual`), *mayor* (`Greater`), *mayor o igual* (`GreaterEqual`), *menor* (`Lower`) y *menor o igual* (`LowerEqual`).

```
data BExp = BCte Bool
          | And BExp BExp
          | Cmp ROp NExp NExp
          | Not BExp
          | Or BExp BExp

data ROp = Equal
         | Greater
         | GreaterEqual
         | NotEqual
         | Lower
         | LowerEqual
```

Por último, las expresiones numéricas (`NExp`) incluyen las contantes enteras (`NCte`), la suma (`Add`), la resta (`Sub`), la multiplicación (`Mul`), la división entera (`Div`) y la operación módulo (`Mod`).

```
data NExp = Variable VarName
          | NCte Int
          | Add NExp NExp
          | Sub NExp NExp
          | Mul NExp NExp
          | Div NExp NExp
          | Mod NExp NExp
```

Los tipos de datos algebraicos definidos nos permiten modelar un programa LIS completo. El siguiente ejemplo muestra el programa LIS para el cálculo del factorial de un número, introducido en la sección 3, y su correspondiente representación mediante estas estructuras.

```
-- Código LIS
program
 {
   -- Parámetros
   n  := 6

   -- Cuerpo
   a  := n;
   fn := 1;
   while (a > 0)
    {
      fn := a * fn;
      a := a - 1
    }

    -- Return
    return := fn
 }
```

```
-- Representación Haskell

lisFactN =
 Program
 [
    -- Parámetros
    Assign "n"    (NCte 6),

    -- Cuerpo
    Assign "a"    (Variable "n"),
    Assign "fn"   (NCte 1),
    While (Cmp Greater (Variable "a")
                       (NCte 0))
     [
       Assign "fn"
              (Mul (Variable "a")
                   (Variable "fn")),
       Assign "a"
              (Sub (Variable "a")
                   (NCte 1))
     ],

   -- Retorno
   Assign "return" (Variable "fn"),
 ]
```

A partir de esta representación es posible implementar un evaluador que nos permita evaluar el resultado de la ejecución del programa, como se verá en la sección 3.2.

3.2. Evaluación de LIS

Para la evaluación de una representación de un programa LIS implementaremos funciones que transformen cada una de los elementos del lenguaje representados en el valor denotado y, adicionalmente, utilizaremos un tipo de dato abstracto `Memory`, que es una memoria de variables.

La memoria permite realizar escrituras y lecturas mediante las funciones `memoryRead` y `memoryWrite` y listar las variables inicializadas de la memoria empleando la función `memoryVariables`. Adicionalmente es posible crear una nueva memoria vacía a través de la función `memoryNew`. Su interfaz se muestra a continuación.

```
memoryNew       :: Memory
memoryRead      :: Memory -> VarName -> Int
memoryWrite     :: Memory -> VarName -> Int -> Memory
memoryVariables :: Memory -> [VarName]
```

La implementación de la representación interna y operaciones de este tipo de dato abstracto se encuentra en el apéndice A.

Definimos la función evaluadora de programa `evalProgram` recibe un programa y evalúa su bloque.

```
evalProgram (Program block) = evalBlock block
```

La evaluación del bloque se realiza mediante la función `evalBlock` que recursivamente evalúa cada unos de sus comandos, actualizando la memoria a partir de cada evaluación.

```
evalBlock [] = id
evalBlock (c:cs) =
    \mem -> let mem' = evalCom c mem
            in evalBlock cs mem'
```

La evaluación de un comando evalúa cada uno de sus casos particulares. La evaluación del comando `Skip` no altera la memoria. La evaluación del comando `Assign` asocia el valor de una expresión a una variable en la memoria tras la evaluación de dicha expresión. La evaluación del comando `If` evalúa una expresión booleana y a partir de ella evalúa el bloque correspondiente. Finalmente, el comando `While` se implementa como una recursión en función de la evaluación de un comando `If` que controla la recursión y la finaliza con un comando `Skip`.

```
evalCom Skip = id
evalCom (Assign x ne) = \mem -> memoryWrite mem x (evalN ne mem)
evalCom (If be bt bf)
    = \mem -> if (evalB be mem)
              then (evalBlock bt mem)
              else (evalBlock bf mem)
evalCom (While be blk)
    = evalCom (If be (blk ++ [While be blk]) [Skip])
```

La evaluación de las expresiones tanto booleanas como numéricas se realiza traduciendo los constructores que representan operaciones en funciones Haskell que las implementan (`+`, `-`, `==`, `&&`, etc.) y los constructores que representan constantes en los valores Haskell correspondientes.

```
-- Evaluación de expresiones booleanas

evalB (BCte b) mem = b
evalB (Cmp rop e1 e2) mem = evalROp rop (evalN e1 mem) (evalN e2 mem)
evalB (And e1 e2) mem = evalB e1 mem && evalB e2 mem


evalROp Equal        = (==)
evalROp NotEqual     = (/=)
evalROp Greater      = (>)
evalROp GreaterEqual = (>=)
evalROp Lower        = (<)
evalROp LowerEqual   = (<=)
```

```
-- Evaluación de expresiones numéricas

evalN (Variable x) = \mem ->
    case (memoryRead mem x) of
        Nothing -> error ("Variable '" ++ x ++ "' indefinida")
        Just v  -> v
evalN (NCte n) = \mem -> n
evalN (Add e1 e2) = evalAndApply (+) e1 e2
evalN (Sub e1 e2) = evalAndApply (-) e1 e2
evalN (Mul e1 e2) = evalAndApply (*) e1 e2
evalN (Div e1 e2) = evalAndApply div e1 e2
evalN (Mod e1 e2) = evalAndApply mod e1 e2

evalAndApply f e1 e2 mem = f (evalN e1 mem) (evalN e2 mem)
```

Utilizando estos evaluadores es posible evaluar el resultado de ejecución de una representación de un programa LIS dada. A continuación se muestra el resultado de evaluar la ejecución del programa que calcula el factorial de un número a partir de la invocación de `evaluarProgram` con este programa como argumento y una memoria nueva.

```
Main> evalProgram lisFactN memoryNew
[
  ("return", 720)
  ("n"     ,   6),
  ("a"     ,   0),
  ("fn"    , 720)
]
```

4. Biblioteca de parsers

Para la definición de LIS es preciso contar con una biblioteca de parsers que faciliten su implementación. Estos parsers funcionales [1] utilizan la técnica de *Monadic Parser Combinators*, técnica de análisis sintáctico propuesta por Graham Hutton y Erik Meijer en su artículo *"Monadic Parser Combinators"* [2]. A continuación se describe la implementación de cada uno de los parsers, funciones y tipos que componen esta biblioteca, muchos de ellos basados en los presentados en la bibliografía mencionada.

4.1. El tipo `Parser`

El tipo `Parser` representa el tipo de una función que a partir de una cadena de caracteres denota una lista de todos los posibles análisis sintácticos.

```
newtype Parser a = Parser (String -> [(a, String)])
```

Un análisis sintáctico en particular se representa mediante una tupla donde su primer valor denota la estructura generada y la segunda el remanente de la

cadena de caracteres recibida como parámetro. El análisis sintáctico resultará fallido si a partir de una cadena de caracteres el parser devuelve una lista vacía.

4.2. Mónadas

Para los parsers siguientes es necesario introducir la idea de mónadas. Las mónadas son abstracciones matemáticas que permiten aplicar una secuencia de operaciones sobre un estado que se actualiza con cada operación. En este trabajo utilizaremos las mónadas para aplicar una secuencia de parsers sobre una cadena de caracteres.

```
instance Monad Parser where
    return v         =  Parser (\inp -> [(v, inp)])
    Parser p >>= f   =  Parser (\inp ->
                            concat [ parse (f v) out |
                                     (v, out) <- p inp ])
```

La instancia de mónada `Monad Parser` define la función `return` como un parser que siempre tiene éxito sin importar la cadena de caracteres analizada y la función de secuenciación `>>=` (o *bind*) como la concatenación de todos los análisis sintácticos generados a partir del la utilización de cada análisis sintácticos de `p` como argumento de un segundo parser (generado por `f`).

También definimos la clase `MonadFailureOr` que representa una mónada con las funciones `failure` y `<|>` (se lee *"o"*) y una instancia `MonadFailureOr Parser`.

```
class Monad m => MonadFailureOr m where
    failure  ::  m a
    (<|>)    ::  m a -> m a -> m a

instance MonadFailureOr Parser where
    failure                  =  Parser(\inp -> [])
    (Parser p) <|> (Parser q)  =  Parser(\inp -> p inp ++ q inp)
```

Esta instancia implementa la función `failure` como un parser que falla para cualquier cadena de caracteres y la función `<|>` como la concatenación de los análisis sintácticos resultados de aplicar la cadena de caracteres a dos parsers distintos.

4.3. Parsers básicos

Entre los parsers básicos incluidos en esta biblioteca podemos encontrar al parser `item`, que simplemente consume el primer carácter de una cadena de caracteres fallando en el caso en que la cadena sea vacía,

```
item = Parser (\inp -> case inp of
                         []     -> []
                         (x:xs) -> [(x, xs)])
```

el parser `satisfy`, que consume un carácter retornando el carácter si satisface el predicado `p` o fallando en caso contrario,

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do
              c <- item
              if p c then return c else failure
```

el parser `symbol`, que consume un carácter si el mismo es igual a un carácter dado,

```
symbol :: Char -> Parser Char
symbol x = satisfy (==x)
```

y el parser `token` que consume una cadena de caracteres que coincide con la dada.

```
token :: String -> Parser String
token ""     = return ""
token (c:cs) = do
                 symbol c
                 token cs
                 return (c:cs)
```

Estos parsers por sí solos no poseen utilidad alguna. En las siguientes secciones introduciremos parsers que permiten combinar parsers y así definir parsers para repeticiones de elementos y listas de elementos, entre otros.

4.4. Repetición de elementos

Para realizar el análisis sintáctico de repeticiones de elementos definiremos el parser `many`, que aplica el parser dado, tantas veces como sea posible, denotando una lista con los resultados obtenidos.

```
many p  = (do
             x <- p
             xs <- many p
             return (x:xs))
       <|> return []
```

Para aquellas repeticiones que comienzan con un elemento concreto definiremos el parser `manyBeginWith`, que aplica un parser de comienzo (`start`) antes de aplicar sucesivas veces el parser para los elementos de la repetición.

```
manyBeginWith p start
    = do
        x  <- start
        xs <- many p
        return (x:xs)
```

En base a este, definiremos la versión estricta `many1` que requiere que el parser dado logre al menos una aplicación exitosa, de lo contrario falla.

```
many1 p = manyBeginWith p p
```

Para aquellas repeticiones de elementos separados por elementos de separación definiremos el parser `listOf`. Este parser aplica de manera sucesiva el parser `p` intercalando aplicaciones del parser separador `sep` e ignorando el resultado de este último.

```
listOf p sep = do
                 x  <- p
                 xs <- rest p sep
                 return (x:xs)
               where
                 rest p sep
                     = many (ignoreFirst sep p)
                 ignoreFirst sep p
                     = do
                         _ <- sep
                         x <- p
                         return x
```

4.5. Delimitadores

Para aquellos elementos del lenguaje que se encuentran delimitados por dos elementos (posiblemente distintos) definiremos el parser `pack` que aplica los parsers `open` y `close` antes y después de la aplicación del parser `p`, retornando únicamente el resultado de este último.

```
pack open p close = do
                      open
                      x <- p
                      close
                      return x
```

Los parsers `parenthesized`, `bracketized` y `braced` esperan consumir paréntesis, corchetes y llaves respectivamente antes y después de la aplicación del parser dado.

```
parenthesized p = pack (symbol '(') p (symbol ')')
bracketized p   = pack (symbol '[') p (symbol ']')
braced p        = pack (symbol '{') p (symbol '}')
```

4.6. Elementos opcionales

En ocasiones los lenguajes de programación presentan elementos opcionales. Los parsers `optionDef` y `option` permiten definir parsers para tales elementos. El parser `optionDef` que aplica el parser retornando su resultado de ser exitoso o devuelve un parser que siempre tiene éxito con un valor por defecto `def`.

```
optionDef p def  =  p
                <|> return def
```

El parser `option` define un parser que tiene éxito o, si falla, obtiene el valor `[]` indicando que no hubo resultado.

```
option p = optionDef p []
```

4.7. Palabras e identificadores

Definiremos parsers para el análisis sintáctico de caracteres y cadenas de caracteres que nos permitirán realizar el análisis sintáctico de palabras e identificadores. Entre los parsers que consumen un único carácter podemos encontrar: `lower`, que consume una letra en minúscula, `upper` que consume una letra en mayúscula, `letter` consume una letra cualquiera, `digit` que consume un dígito y `alphanum` que consume una letra o un dígito.

```
lower    = satisfy (between 'a' 'z')
upper    = satisfy (between 'A' 'Z')
letter   = lower <|> upper
digit    = satisfy (between '0' '9')
alphanum = letter <|> digit

between low high = (\x -> low <= x && x <= high)
```

Por otro lado definimos parsers que consumen cadenas de caracteres de cualquier longitud, ellos son: `alphanumWord` que consume una palabra escrita con caracteres alfanuméricos,

```
alphanumWord = many1 alphanum
```

y dos parsers de identificadores `lowerId` y `upperId` que consumen identificadores que comienzan con una letra minúscula y mayúscula respectivamente.

```
lowerId = manyBeginWith alphanum lower
upperId = manyBeginWith alphanum upper
```

4.8. Números naturales y enteros

Si deseamos trabajar con expresiones numéricas es necesario contar con parsers para números. A partir del parser de dígito se definimos el parser `natural`, que consume un número natural, y el parser `sign`, que consume un signo `+` o `-`.

```
natural = many1 digit
sign    = symbol '+' <|> symbol '-'
```

A partir de estos es posible definir el parser que consume un número entero.

```
integer = do
            s <- optionDef sign '+'
            n <- natural
            return (s:n)
```

4.9. Transformación de resultados

Otro parser del que haremos uso es el parser `<@` (o *apply*) que aplica un parser transforma su resultado mediante una función dada.

```
p <@ f = do
           x <- p
           return (f x)
```

El parser `tokenAs` (también `op`) que consume una cadena de caracteres y devuelve un valor ignorando el resultado original. Este parser facilita la definición de parsers para operadores como ser verá en la sección 5.

```
op = tokenAs
tokenAs s v       = replaceResult (token s) v
replaceResult p v = p <@ const v
```

4.10. Secuencia de operadores y operandos

Por último, agregaremos a la biblioteca los parsers `chainl1` y `chainr1` que están pensados para el análisis sintáctico de una sucesión de operandos separados por operadores.

```
p `chainl1` op = do
                   x <- p
                   rest x
                 where
                   rest x = (do
                               f <- op
                               y <- p
                               rest (f x y)) <|> return x
```

```
p `chainr1` op = do
                    x <- p
                    (do
                        f <- op
                        y <- p `chainr1` op
                        return (f x y)) <|> return x
```

El parser `chainl` asocia a izquierda y el parser `chainr1` asocia a derecha. En ambos casos, el resultado del análisis sintáctico del parser de operadores debe retornar una función responsable de combinar los operandos ligados a éste.

4.11. Definiciones auxiliares

Además de los parsers definiremos tres funciones que simplifican su utilización. La función `parse` aplica un parser a una cadena de caracteres.

```
parse :: Parser a -> String -> [(a, String)]
parse (Parser x) = x
```

La función `bestParse` aplica un parser y devuelve sólo los resultados del análisis sintáctico que terminaron por consumir toda la cadena de caracteres.

```
parse :: Parser a -> String -> [(a, String)]
bestParse p = (filter (\(v, inp') -> inp' == "")) . parse p
```

La función `bestFirst` devuelve el primero de los resultados obtenidos mediante `bestParse` levantando un error si no se produjeron resultados que consumen el total de su entrada.

```
parse :: Parser a -> String -> (a, String)
bestFirst p = case bestParse p of
                [] -> error "Parsing error"
                xs -> head xs
```

Sobre este conjunto de parsers y funciones auxiliares se construye el parser del lenguaje de programación LIS.

5. El parser de LIS

Comenzaremos por definir los parsers para los comandos simples seguido de los parsers para los comandos compuestos y luego nos dedicaremos a definir los parsers de comandos, bloque, expresiones booleanas y expresiones numéricas.

5.1. Comandos simples

El primer parser definido es `skip` que consume el token `"skip"` y devuelve el constructor `Skip`.

```
skip = tokenAs "skip" Skip
```

Luego podemos definir el parser `assign` que consume la sintaxis de la asignación generando un comando `Assign` a partir del nombre de variable y la expresión obtenidos del análisis sintáctico.

```
assign = do
           v <- varname
           token ":="
           e <- nExpr
           return (Assign v e)

varname = lowerId
```

El parser `varname` consume un nombre de variable y es equivalente a `lowerId`. El parser de expresiones numéricas `nExpr` se define más adelante en esta sección.

5.2. Comandos compuestos

El parser `iff` realiza el análisis sintáctico de la alternativa condicional.

```
iff = do
        token "if"
        bexp      <- parenthesized bExpr
        blockThen <- block
        blockElse <- option iffElse
        return (If bexp blockThen blockElse)

iffElse = do
            token "else"
            blockElse <- block
            return blockElse
```

El parser de la repetición condicional, similar al de la alternativa condicional, construye un comando `While` a partir de una expresión booleana y un bloque.

```
while = do
          token "while"
          bexp  <- parenthesized bExpr
          blck  <- block
          return (While bexp blck)
```

Habiendo definido los parsers para cada comando, definimos el parser `command` para cualquiera de ellos y el parser `commands` que realiza el análisis sintáctico una lista de comandos separados, opcionalmente, por punto y coma.

```
commands = do
             cs <- listOf command (option (token ";"))
             option (token ";")
             return cs

command  =  skip
        <|> assign
        <|> iff
        <|> while
```

El parser `block` realiza el análisis sintáctico de una secuencia de comandos delimitada por llaves.

```
block = braced commands
```

5.3. Expresiones booleanas

El parser `bExpr` define el análisis sintáctico de una expresión booleana, que no es más que una sucesión de términos booleanos intercalados por operadores lógicos.

```
bExpr  =  bTerm `chainl1` logop
bTerm  =  bConstant
      <|> nCmp

logop  =  op "&&" And
      <|> op "||" Or
```

El parser para un término booleano consume una constante booleana o una comparación. La comparación construye un valor `BExp` a partir del constructor `Cmp`, de un operador relacional `relop` y dos expresiones numéricas.

```
nCmp   =  do
            n1 <- nExpr
            r  <- relop
            n2 <- nExpr
            return (Cmp r n1 n2)

relop  =  op ">"  Greater
      <|> op ">=" GreaterEqual
      <|> op "<"  Lower
      <|> op "<=" LowerEqual
      <|> op "==" Equal
      <|> op "/=" NotEqual
```

El parser `bConstant` consume la constante `True` o `False` mediante los parsers `bTrue` o `bFalse` respectivamente.

```
bConstant  =  bTrue
          <|> bFalse
bTrue      = tokenAs "True"  (BCte True )
bFalse     = tokenAs "False" (BCte False)
```

5.4. Expresiones numéricas

Definimos el parser `nExpr` que consume una sucesión de términos separados por operadores de suma y resta, y el parser `term` que consume factores separados por operadores de multiplicación, división y módulo. El parser `factor` aplica cualquiera de los parsers para enteros (`nConstant`), para variables `variable` y para expresiones numéricas rodeadas por paréntesis (`parenthesized nExpr`).

```
nExpr   = term   `chainl1` addop
term    = factor `chainr1` mulop
factor  =  nConstant
       <|> variable
       <|> parenthesized nExpr
```

El parser `addop` consume un operador de suma o resta y devuelve un constructor `Add` o `Sub` respectivamente. De manera similar al anterior, el parser `mulop` consume un operador de multiplicación, división o módulo y devuelve un constructor `Mul`, `Div` o `Mod`.

```
addop  =  op "+" Add
      <|> op "-" Sub

mulop  =  op "*" Mul
      <|> op "/" Div
      <|> op "%" Mod
```

El parser `nConstant` consume un número entero y construye una constante numérica `NCte`. El parser `variable` consume un nombre de variable y con él construye un valor `Variable`.

```
nConstant = integer <@ (NCte . strToInt)

variable = varname <@ Variable

strToInt (c:cs) = applySign c (toNat cs)
                  where
                    applySign s = case s of
                                    '+' -> (*1)
                                    '-' -> (* -1)
                    toNat cs
                      = foldl (\a x -> a*10 + toDigit x) 0 cs
                    toDigit x = ord x - ord '0'
```

5.5. Parser de programa

El último parser es `lisParser`, que dado un programa LIS realiza su análisis sintáctico retornando una representación de programa.

```
lisParser = do
              token "program"
              blk <- block
              return (Program blk)
```

La función `parseLIS` aplica el parser de LIS sobre una entrada devolviendo el mejor resultado o un error indicando que el análisis sintáctico fue fallido.

```
parseLIS = bestFirst lisParser
```

5.6. Eliminación de comentarios y espacios en blanco

Definimos además una función `junkOut` que limpiará una cadena de caracteres de espacios en blanco (`"\n"`, `"\t"` y `" "`) y comentarios de línea comenzados con `"--"`.

```
junkOut []  = []
junkOut inp = let
                 inp' = (removeWhites . removeComments) inp
              in
                 if inp' /= inp then
                   junkOut inp'
                 else
                   inp'

removeWhites = dropWhile (\c -> c `inn` ['\n', '\t', ' '])

removeComments ('-':'-':inp) = dropWhile (\x -> x /= '\n') inp
removeComments inp           = inp

inn :: Eq a => a -> [a] -> Bool
x `inn` xs = foldr (\y r -> r || (y == x)) False xs
```

Modificaremos la mónada `Monad Parser` donde aplicaremos la función `junkOut` para limpiar la entrada de cada parser.

```
instance Monad Parser where
    return v         =  Parser (\inp -> [(v, junkOut inp)])
    Parser p >>= f   =  Parser (\inp ->
                          concat [ parse (f v) (junkOut out) |
                                   (v, out) <- p (junkOut inp)])
```

6. Evaluación de un programa LIS

La función `executeLIS` realiza el análisis sintáctico de un programa LIS y luego evalúa la representación obtenida devolviendo el estado final de la memoria empleada con los resultados de la evaluación.

```
evaluateLIS = evalProgram . fst . parseLIS
```

Un ejemplo de esta ejecución es el siguiente programa LIS que dado un número `n`, calcula en la variable `res` su conversión a número binario.

```
-- Number to Binary
program
 {
    -- Parameters
    n := 235;

    -- Body
    num := n;
    bit := 0;
    res := 0;
    offset := 1
    while (num > 0)
     {
       bit := num % 2;
       num := num / 2;
       res := bit * offset + res;
       offset := offset * 10
     }

     -- Retorno
     return := res / 10
 }
```

A partir del análisis sintáctico de este programa, el parser de LIS genera la siguiente representación.

```
-- Number to Binary
Program
 [
   -- Parameters
   Assign "n"      (NCte 235),

   -- Body
   Assign "num"    (Variable n),
   Assign "bit"    (NCte 0),
   Assign "res"    (NCte 0),
   Assign "offset" (NCte 1),
   While (Cmp Greater (Variable num ) (NCte 0 ))
    [
      Assign "bit"    (Mod (Variable num) (NCte 2)),
      Assign "num"    (Div (Variable num) (NCte 2)),
      Assign "res"    (Add (Mul (Variable bit)
                               (Variable offset))
                           (Variable res)),
      Assign "offset" (Mul (Variable offset) (NCte 10)),
    ],

   -- Return
   Assign "return" (Div (Variable res) (NCte 10))
 ]
```

Y su evaluación devuelve el estado final de la memoria como se muestra a continuación.

```
[
  ("return",   1110101),
  ("offset", 100000000),
  ("res"   ,  11101011),
  ("bit"   ,         1),
  ("num"   ,         0),
  ("n"     ,       235)
]
```

7. Conclusiones

Tras la finalización de este trabajo, sus objetivos fueron satisfechos y resultaron en un parser del lenguaje LIS cuyo código se encuentra disponible en la dirección `https://github.com/arypbatista/LIS-Parser`.

La realización de este trabajo representó un destacado aprendizaje respecto de la implementación de lenguajes que me permitirá, el día de mañana, adentrarme en el mundo de los intérpretes, compiladores y transformaciones de programas con un importante conocimiento previo.

La implementación de parsers es una herramienta muy útil al momento de aprender conceptos relacionados con el diseño e implementación de los lenguajes. El proceso de transformación (y abstracción) de un programa a una represen-

tación simbólica facilita la comprensión de la estructura de los lenguajes de programación.

El desarrollo técnico requerido para esta implementación favoreció a afianzar mis conocimientos sobre el lenguaje de programación funcional Haskell y comprender en profundidad la idea de mónadas.

Como trabajo a futuro se propone implementar un compilador de LIS, que transforme la representación del programa en una representación más simple en términos de ejecución, y una máquina virtual completa para la ejecución del resultado de la compilación.

Referencias

[1] Fokker, Jeroen: *Functional Parsers*.

[2] Hutton, Graham y Erik Meijer: *Monadic Parser Combinators*. Informe técnico, Department of Computer Science, University of Nottingham, 1996.

A. Implementación del tipo de datos abstracto `Memory`

El código que se muestra en esta sección define la representación interna del tipo de datos abstracto `Memory` e implementa la interfaz para manipularlo.

```
module Memory (
    Memory,
    memoryNew,
    memoryWrite,
    memoryRead,
    memoryVariables,
) where

type VarName = String
type VarValueAssoc = (VarName, Int)

data Memory = Memory [VarValueAssoc]

memoryNew :: Memory
memoryNew = (Memory [])

memoryWrite :: Memory -> VarName -> Int -> Memory
memoryWrite mem v x = case memoryRead mem v of
                        Nothing -> memoryAdd mem v x
                        Just _  -> memoryReplace mem v x

memoryRead :: Memory -> VarName -> Maybe Int
memoryRead (Memory ss) v = case (find (\(v', x) -> v' == v) ss) of
                             Nothing    -> Nothing
                             Just (v,x) -> Just x

memoryVariables :: Memory -> [VarName]
memoryVariables (Memory ss) = map fst ss

-- Internal functions

memoryReplace :: Memory -> VarName -> Int -> Memory
memoryReplace (Memory ss) v x
  = Memory (repl (\(v', x) -> v' == v) x ss)

memoryAdd :: Memory -> VarName -> Int -> Memory
memoryAdd (Memory ss) v x = Memory ((v,x):ss)

memoryDump :: Memory -> [VarValueAssoc]
memoryDump (Memory ss) = ss
```

```
find :: (VarValueAssoc -> Bool) -> [VarValueAssoc] -> Maybe VarValueAssoc
find p []     = Nothing
find p (x:xs) = if p x then
                  Just x
                else
                  find p xs

repl :: (VarValueAssoc -> Bool) -> Int -> [VarValueAssoc] -> [VarValueAssoc]
repl p y []         = error "Not found"
repl p y ((v,x):xs) = if p (v,x) then
                        (v, y) : xs
                      else
                        (v, x) : repl p y xs

instance Show Memory where
    show = show . memoryDump
```

B. Visualización de la representación de un programa LIS

El siguiente código define la visualización de una representación de un programa LIS.

```
join :: String -> [String] -> String
join sep [] = ""
join sep (s:ss) = s ++ sep ++ join sep ss

wrap s = "(" ++ s ++ ")"

wNJ = wrap . (join " ")

instance Show ROp where
    show Equal        = "Equal"
    show NotEqual     = "NotEqual"
    show Greater      = "Greater"
    show GreaterEqual = "GreaterEqual"
    show Lower        = "Lower"
    show LowerEqual   = "LowerEqual"


instance Show BExp where
    show (BCte b)          = wNJ ["BCte", show b]
    show (And b1 b2)       = wNJ ["And",  show b1,  show b2]
    show (Or b1 b2)        = wNJ ["Or",   show b1,  show b2]
    show (Cmp rop e1 e2)   = wNJ ["Cmp",  show rop, show e1, show e2]
    show (Not b)           = wNJ ["Not",  show b]


instance Show NExp where
    show (Variable name) = wNJ ["Variable", name]
    show (NCte n)        = wNJ ["NCte", show n]
    show (Add e1 e2)     = wNJ ["Add",  show e1, show e2]
    show (Sub e1 e2)     = wNJ ["Sub",  show e1, show e2]
    show (Mul e1 e2)     = wNJ ["Mul",  show e1, show e2]
```

```
    show (Div e1 e2)     = wNJ ["Div",  show e1, show e2]
    show (Mod e1 e2)     = wNJ ["Mod",  show e1, show e2]


instance Show Comando where
    show Skip = "Skip"
    show (Assign v ne) = wNJ ["Assign ", show v, show ne]
    show (If b tb fb)  = wNJ ["If ", show b, show tb, show fb]
    show (While b block) = wNJ ["While ", show b, show block]

instance Show Program where
    show (Program cs) = "Program \n" ++ show cs
```