DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Número 1

6 de julio de 2016 — Teoría de Lenguajes

Grupo n_n

Integrante	LU	Correo electrónico
Gasco, Emilio	171/12	gascoe@gmail.com
Gatti, Mathias	477/14	mathigatti@gmail.com
Patané, Federico	683/10	fede_river_8e@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

0. Introducción

En el presente trabajo práctico hemos llevado a cabo la construcción por completo de un parser. La problemática planteada por la cátedra ha sido la de poder, a partir de un texto de entrada con líneas de código del lenguaje descripto por el Trabajo Práctico, poder si este era tanto sintáctica como semánticamente válido, darle una tabulación correcta según términos de legibilidad conocidos por los programadores.

Para realizar este trabajo, tuvimos que dividir nuestras tareas en varias etapas. La primer etapa, es lograr conseguir los tokens de nuestra entrada, que es tener una lista con todos los símbolos que vinieron en las líneas. Para poder realizar esto, hizo falta crear un lexer, que es un tokenizador, que a través de reglas sabrá dividir la entrada en los tokens, dandole un significado a cada uno de ellos para que luego podamos parsear a los mismos.

Cabe destacar que para la realización de este lexer, decidimos utilizar la herramienta de ply, del lenguaje de programación Python.

Luego de realizada esta etapa, deberemos crear una gramática que pueda parsear correctamente cualquier entrada esperada de nuestro lenguaje. Muy importante destacar la necesidad de tener una gramática no ambigua, para que en el parseo posterior se genere solo un árbol sintáctico, lo cuál nos dará la posibilidad de utilizar el parser que la herramienta ply nos brinda, ya que trabaja con un parser LALR, que es un tipo de parser ascendente, osea que genera este árbol desde las hojas hacia la raìz.

Una vez parseado esto, y generado el árbol de derivación debemos darle semántica a las producciones, para poder hacer que este lenguaje tenga también significado.

1. Analizador léxico

Para configurar el analizar léxico es necesario proveer expresiones regulares que reconozcan cada uno de los tokens. Para poder reconocer las palabras reservadas del lenguaje fue necesario agregar lógica extra al analizador. Para evitar que las palabras reservadas sean usadas como variables se creo un diccionario donde cada elemento es el par (palabra reservado,token palabra reservada). Cuando una cadena del código siendo analizado se corresponde con la expresión regular del token VARIABLE, antes de aceptar el token se buscar la cadena en el diccionario. De encontrar la cadena en el diccionario se utiliza el tipo de token definido en el mismo, en caso contrario se devuelve un token VARIABLE.

1.1. Expresiones regulares para tokens

```
                DO ->    'do'
             WHILE ->    'while'
               FOR ->    'for'
                IF ->    'if'
              ELSE ->    'else'
               RES ->    'res'
            RETURN ->    'return'
             BEGIN ->    'begin'
               END ->    'end'
        CAPITALIZAR ->    'capitalizar'
            LENGTH ->    'length'
             PRINT ->    'print'
MULTIPLICACIONESCALAR ->    'multiplicacionescalar'
         COLINEALES ->    'colineales'
               AND ->    'and'
                OR ->    'or'
               NOT ->    'not'
             MINUS ->    '-'
           ELEVADO ->    '^'
            MODULO ->    '%'
               DIV ->    '/'
             MAYOR ->    '>'
             MENOR ->    '<'
              PLUS ->    '+'
             TIMES ->    '*'
            LPAREN ->    '('
            RPAREN ->    ')'
         LCORCHETE ->    '['
         RCORCHETE ->    ']'
          LLAVEIZQ ->    '{'
          LLAVEDER ->    '}'
      INTERROGACION ->    '?'
```

```
      PUNTO ->    '.'
   DOSPUNTOS ->    ':'
  PUNTOYCOMA ->    ';'
       COMA ->    ','
      IGUAL ->    '='('\n')*'='
    DISTINTO ->    '!'('\n')*'='
    AGREGAR ->    '+'('\n')*'='
      SACAR ->    '-'('\n')*'='
     DIVIDI ->    '/'('\n')*'='
    MULTIPL ->    '*'('\n')*'='
     MASMAS ->    '+'('\n')*'+'
  COMENTARIO ->    '#'.*
     CADENA -> '"' .*? '"'
       BOOL ->    'true' | 'false' | 'FALSE' | 'TRUE'
   VARIABLE -> ([a-z][A-Z])([a-z][A-Z]|'_'|[0-9])*
  ASGINACION ->   '='
```

2. Análisis sintáctico y chequeo de tipos

La herramienta utilizada para construir el analizador sintáctica crea un analizador LALR.

2.1. Gramática

```
<programa>      :   <sentencia> <programa'>
                |   <control> <programa'>
                |   COMENTARIO <programa>

<programa'>     :   <sentencia> <programa'>
                |   <control> <programa'>
                |   COMENTARIO <programa'>
                |   <empty>

<sentencia>     :   <var_asig> ';'
                |   <funcion> ';'

<control>       :   <ifelse>
                |   <loop>

<control_cond>  :   <var_asig_l>
                |   <exp_bool>
                |   <comparacion>
                |   <op_ternario>
```

```
    <loop>          :   'while' '(' <control_cond> ')' <bloque>
                    |   'do' <bloque> 'while' '(' <control_cond> ')' ';'
                    |   <for>

    <for>           :   'for' '(' <for_term> ',' <form_term_2> ',' <for_term> ')'
                            <bloque>


    <for_term>      :   <var_asig>
                    |   <empty>

    <form_term_2>   :   <valores>
                    |   <comparacion>

    <ifelse>        :   'if' '(' <control_cond> ')' <bloque>
                    |   'if' '(' <control_cond> ')' <bloque> else <bloque>

    <bloque>        :   COMENTARIO <bloque>
                    |   <sentencia>
                    |   <control>
                    |   '{' <programa'> '}'

    <funcion>       :   func_ret
                    |   func_void

    <func_void>     :   'print' '(' <valores> ')'

    <func_ret>      :   <func_ret_int>
                    |   <func_ret_cadena>
                    |   <func_ret_bool>
                    |   <func_ret_arreglo>

<func_ret_arreglo>  :   'multiplicarEscalar' '(' <valores> ',' <valores> ')'
                    |   'multiplicarEscalar' '(' <valores> ',' <valores> ','
                                                 <valores>')'

<func_ret_bool>     :   'colineales' '(' <valores> ',' <valores> ')'

<func_ret_cadena>   :   'capitalizar' '(' <valores> ')'

    <func_ret_int>  :   'length' '(' valores ')'


    <valores>       :   <exp_arit>
                    |   <exp_bool>
```

```
                |   <exp_cadena>
                |   <exp_arreglo>
                |   <registro>
                |   <registro> '.' VARIABLE
                |   <var_asig_l>
                |   <op_ternario>

<exp_arreglo>   :   '[' <list_valores> ']'
                |   '[' <list_valores> ']' <exp_arreglo>
                |   '['']'
                |   func_ret_arreglo

<lista_valores> :  <valores>
                |  <valores> ',' <lista_valores>

<registro>      :   '{' <reg_item> '}'

<reg_item>      :   VARIABLE ':' <valores> ',' <reg_item>
                |   VARIABLE ':' <valores>

<var_asig_l>    :   VARIABLE
                |   RES
                |   VARIABLE <var_member>


<var_member>    :   '[' var_asig_l ']' <var_member>
                |   '[' <exp_arit> ']'  <var_member>
                |   '.' VARIABLE <var_member>
                |   '[' <exp_arit> ']'
                |   '[' <var_asig_l> ']'
                |   '.' VARIABLE


<var_asig>      :   <var_asig_l> '++'
                |   '++' <var_asig_l>
                |   <var_asig_l> '--'
                |   '--' <var_asig_l>
                |   <var_asig_l> '*=' <valores>
                |   <var_asig_l> '/=' <valores>
                |   <var_asig_l> '+=' <valores>
                |   <var_asig_l> '-=' <valores>
                |   <var_asig_l> '=' <valores>
                |   <var_asig_l> '=' <comparacion>
                |   <var_asig_l> '=' <operador_ternario>

<op_ternario>   :   <valores> '?' <valores> ':' <valores>
```

```
                |   <comparacion> '?' <valores> ':' <valores>
                |   <valores> '?' <valores> ':' <op_ternario>
                |   <comparacion> '?' <valores> ':' <op_ternario>

<var_oper>      :   <var_asig_l>
                |   '(' <op_ternario> ')'
                |   <exp_arreglo>
                |   <registro> '.' VARIABLE


<exp_arit>      :   <exp_arit> '+' <term>
                |   <exp_arit> '+' <var_oper>
                |   <var_oper> '+' <term>
                |   <var_oper> '+' <var_oper>
                |   <exp_arit> '-' <term>
                |   <exp_arit> '-' <var_oper>
                |   <var_oper> '-' <term>
                |   <var_oper> '-' <var_oper>
                |   <term>


<arit_oper_2>   :   '*'
                |   '/'
                |   '%'

<term>          :   <term>  <arti_oper_2> <factor>
                |   <var_oper> <arit_oper_2> <factor>
                |   <term> <arit_oper_2> <var_oper>
                |   <var_oper> <arit_oper_2> <var_oper>
                |   <factor>

<factor>        :   <base> '^' <sigexp>
                |   <var_oper> '^' <sigexp>
                |   '-' <base>
                |   '-' <var_oper>
                |   <base> '++'
                |   <base> '--'
                |   '++' <base>
                |   '--'<base>
                |   <var_oper> '++'
                |   '++' <var_oper>
                |   <var_oper> '--'
                |   '--' <var_oper>
                |   <base>

<base>          :   '(' <exp_arit> ')'
```

```
                |   '(' <var_oper> ')'
                |   NUMBER
                |   <func_int>

<sigexp>        :   '-' <exp>
                |   <exp>

<exp>           :   <var_oper>
                |   NUMBER
                |   '(' <exp_arit> ')'

<exp_cadena>    :   <exp_cadena> '+' <term_cadena>
                |   <var_oper> '+' <term_cadena>
                |   <term_cadena> '+' <var_oper>
                |   <term_cadena>

<term_cadena>   :   CADENA
                |   <func_ret_cadena>
                |   '(' <exp_cadena> ')'



<exp_bool>      :   <exp_bool> AND <term_bool>
                |   <var_oper> AND <term_bool>
                |   <exp_bool> AND <var_oper>
                |   <var_oper> AND <var_oper>
                |   <term_bool>

<term_bool>     :   <term_bool> OR <factor_bool>
                |   <var_oper> OR <factor_bool>
                |   <term_bool> OR <var_oper>
                |   <var_oper> OR <var_oper>
                |   'not' <factor_bool>
                |   'not' <var_oper>

<factor_bool>   :   BOOL
                |   '(' <exp_bool> ')'
                |   '(' <comparacion> ')'
                |   <func_bool>

<op_comp>       :   '=='
                |   '>'
                |   '<'
                |   '!='

<comparacion>   :   <valores> <op_comp>  <valores>
```

3. Análisis semántico

Con lo descrito hasta ahora podemos reconocer y aceptar textos escritos con una sintaxis correcta bajo las reglas enunciadas en las consignas del trabajo pero hay posibles expresiones que no queremos que sean aceptadas a pesar de que cumplan las reglas de la gramática, porque pueden carecer de significado para el proposito del lenguaje. Para esta situación extenderemos lo que tenemos a una gramática de tipos. De esta manera podremos detectar, por ejemplo, cuando se intenta realizar un cálculo aritmetico con una variable que no fue declarada.

3.1. Introduccion

Para hacer esto utilizamos algunas estructuras de datos. La principal es una tupla [STRING, TIPO] la cual tiene como primer componente el texto de salida y como segunda componente su tipo. Si una variable no esta definida su tipo será 'ND'. Si en algún momento se le asigna un valor entonces almacenaremos su nuevo tipo en un diccionario llamado *variables_dict*. Entonces al recorrer las reglas de la gramática que forman el texto iremos armando el texto de sálida al mismo tiempo que verificaremos que no haya ningún error semántico. Como ejemplo se puede ver como se describe el tipo de la siguiente regla.

```
def p_factor_var_op_pp(p):
    'factor : var_oper MASMAS'
    p[0] = [p[1][0] + '++', p[1][1]]

    if not esNumber(p[1][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))
```

El valor del tipo de factor es $p[0]$ el cual como dijimos queda definido por una tupla, la primer componente es el valor de la variable *var_oper* seguido del signo ++, el segundo es el tipo, el cual dependerá unicamente del tipo de *var_oper*, al cual se accede escribiendo $p[1][1]$. En este caso sería un problema que la variable no fuera de tipo INT o $FLOAT$, para verificar esto se utiliza la funcion *esNumber* y dependiendo su resultado se expondrá un mensaje de error de tipo o no.

3.2. Registros y Arreglos

Otro posible error semántico es el de los registros y el acceso a variables que no pertenecen al mismo. Para esto, de manera muy similar al caso anterior, utilizamos un

diccionario llamado *reg_dic*. Aquí se guardan todas las variables de cada registro que se define en el código de entrada. Y se verifica en caso de haber un intento de acceder al valor de la variable de algún registro si la misma fue definida. En la siguiente regla se ve un caso en el que esto ocurre.

```
def p_valores_reg2(p):
    'valores : reg PUNTO VARIABLE'
    p[0] = [p[1][0] + '.' + p[3], 'CUALQUIER_TIPO']
    if estaReg(p[3]) == 'ND':
        pass
        raise SemanticException('REGISTRO',p.lineno(1),p.lexpos(1))
```

Aquí verificamos utilizando la función *estaReg* si $VARIABLE$ fue definida como clave de algún registro. De no ser así devolvemos un error de tipo $'REGISTRO'$. Notar que para no complicar demasiado el código decidimos no verificar cual es el tipo exacto de esa variable en el registro. Por eso le asignamos como tipo $'CUALQUIER\_TIPO'$ y en estos caso permitimos que esta sea parte de cualquier tipo de expresiones, ya sea booleana, aritmetica o cadena. Lo mismo sucede cuando accedemos al valor de un arreglo.

3.3. Comentarios

Para reconocer si un comentario es inline o no utilizamos la funcion *lineno* la cual nos proporciona la linea en la cual esta escrita una sentencia, si dicha sentencia esta escrita en la misma que un comentario, luego a la salida imprimimos ambos a la misma altura, en caso contrario los separamos en distintas lineas. Un ejemplo de como hacemos esto se puede ver en el siguiente fracmento de codigo extraido de *parser_rules.py* .

```
def p_programa_s_pp(p):
    'p : sentencia pp'
    if (p.lineno(1) == p.lineno(2)) and p[2][1] == 'COMENTARIO':
        p[0] = [p[1][0] + ' '  + p[2][0], 'ND']
    else:
        p[0] = [p[1][0] + '\n'  + p[2][0], 'ND']
```

El if verifica que ambas expresiones esten en la misma linea como dijimos pero a parte, obviamente, que el tipo de la segunda expresión sea $'COMENTARIO'$, de no ser así, deberiamos imprimir ambas sentencias en distintas lineas por mas que originalmente hubieran estado juntas.

3.4. Tabulaciones

Para manejar las tabulaciones usamos la función *find_and_replace* que recibe como entrada un string y la agrega a todos los saltos de linea una tabulacion. De esta manera cuando sea necesario, por ejemplo en los bloques de los while, for e if's llamarremos a esta función para que les agregue un espaciado extra.

```
def p_ifSinElse(p):
    'ifelse : IF LPAREN control_cond_term RPAREN bloque'
    p[0] = ['If(' + p[3][0] + ')\n    ' + find_and_replace(p[5][0]) + '\n', 'ND']
```

3.5. Detalles Finales

Como el código esta comentado y pensamos que será de fácil lectura ya que no tiene mayores dificultades no extenderemos demasiado esta sección, pero creemos que vale la pena mencionar algunos detalles finales para facilitar el entendimiento del lector. El valor 'ND' es un poco ambigüo ya que lo utilizamos mayormente para simbolizar el tipo de una variable que no esta definida pero también en algunas partes está asignado al tipo de algunas expresiones las cuales no necesitamos chequear nunca. Los posibles errores semánticos son variados, se puede encontrar el listado de todos los errores posibles en el archivo *semantic_error.py*

4. El Código

4.1. Código de lexer_rules.py

```
# Si el codigo esta vacio o solo tiene espacios, saltos de linea y demas caracteres no

#
reserved = {
'do':'DO',
'while':'WHILE',
'for':'FOR',
'if':'IF',
'else':'ELSE',
'res':'RES',
'true':'TRUE',
'false':'FALSE',
'return':'RETURN',
'begin':'BEGIN',
'end':'END',
'capitalizar':'CAPITALIZAR',
'length':'LENGTH',
'print':'PRINT',
'multiplicacionescalar':'MULTIPLICACIONESCALAR',
'colineales':'COLINEALES',
'and':'AND',
'or':'OR',
'not':'NOT',

}

notokens = {
'begin':'BEGIN',
'end':'END',
'true':'TRUE',
'false':'FALSE',
'return':'RETURN',
}

tokens = [
# Numeros
   'NUMBER',
   'PLUS',
   'TIMES',
   'MINUS',
   'ELEVADO',
```

```
    'MODULO',
    'DIV',
    'IGUAL',
    'DISTINTO',
    'MAYOR',
    'MENOR',
# +=, -=, /=, *=, ++, --
    'AGREGAR',
    'SACAR',
    'DIVIDI',
    'MULTIPL',
    'MASMAS',
    'LESSLESS',
# Booleanos
'BOOL',

#
    'INTERROGACION',
    'PUNTO',
    'DOSPUNTOS',
    'PUNTOYCOMA',
    'COMA',
    'COMENTARIO',
    'LLAVEDER',
    'LLAVEIZQ',
    'LPAREN',
    'RPAREN',
    'RCORCHETE',
    'LCORCHETE',
#
    'VARIABLE',
    'CADENA',

# Estos actualmente estan siendo ignorados, son tomados como tipo VARIABLE
    'ASIGNACION',
] + [valor for valor in list(reserved.values()) if valor not in list(notokens.values()

def t_NUMBER(token):
r"([0-9]+(\.[0-9][0-9]*)?)"
return token

def t_NEWLINE(token):
  r"\n+"
  token.lexer.lineno += len(token.value)

def t_IGUAL(token):
```

```
    r"=(\n)*="
    token.value = '=='
    return token


def t_DISTINTO(token):
    r"\!(\n)*="
    token.value = '!='
    return token;

def t_AGREGAR(token):
    r"\+(\n)*="
    token.value = '+='
    return token;

def t_SACAR(token):
    r"\-(\n)*="
    token.value = '-='
    return token

def t_DIVIDI(token):
    r"/(\n)*="
    token.value = '/='
    return token;

def t_MULTIPL(token):
    r"\*(\n)*="
    token.value = '*='
    return token;

def t_MASMAS(token):
    r"\+(\n)*\+"
    token.value = '++'
    return token

def t_LESSLESS(token):
    r"-(\n)*-"
    token.value = '--'
    return token

t_MINUS = r"\-"
t_ELEVADO = r"\^"
t_MODULO = r"\%"
t_DIV = r"\/"
t_MAYOR = r">"
t_MENOR = r"<"
```

```
t_PLUS = r"\+"
t_TIMES = r"\*"

t_LPAREN = r"\("
t_RPAREN = r"\)"
t_LCORCHETE = r"\["
t_RCORCHETE = r"\]"
t_LLAVEIZQ = r"\{"
t_LLAVEDER = r"\}"
t_INTERROGACION = r"\?"
t_PUNTO = r"\."
t_DOSPUNTOS = r"\:"
t_PUNTOYCOMA = r"\;"
t_COMA = r"\,"
t_COMENTARIO = r"\#.*"


t_CADENA = r"\" .*? \" "

def t_BOOL(token) :
    r"true | false | FALSE | TRUE"
    return token

def t_VARIABLE(token):
  r"([a-z]|[A-Z]) ([a-z]|[A-Z]|\_|[0-9])*"
  token.type = reserved.get(token.value.lower(),'VARIABLE')
  if token.type in notokens.values():
   raise Exception('Su codigo contiene palabras reservadas')
  else:
return token

t_ASIGNACION = r"="

t_ignore = " \t"


def t_error(token):
    message = "Token desconocido:"
    message += "\ntype:" + token.type
    message += "\nvalue:" + str(token.value)
    message += "\nline:" + str(token.lineno)
    message += "\nposition:" + str(token.lexpos)
    raise Exception(message)
```

4.2. Código de parser_rules.py

```
from lexer_rules import tokens
import ply.yacc as yacc
from semantic_error import SemanticException

# Diccionario donde se almacenaran las variables declaradas junto con su tipo
variables_dict = dict()

# Diccionario donde se almacenaran los nombres de las variables de todos los registros
reg_dict = dict()

# Funcion que reemplaza los '\n' por '\n    ' o sea agrega un tab en cada salto de lin
def find_and_replace(palabra):
    j = 0
    res = ''
    for i in xrange(len(palabra)):
        if palabra[i] == '\n':
            res = res + palabra[j:i] + '\n    '
            j = i+1
    return res + palabra[j:]

# Funcion que devuelve lo que esta despues de un guion bajo en un string, por ejemplo
def tipo(palabra):
j = len(palabra)
for i in xrange(len(palabra)):
if palabra[i] == '_':
j = i + 1
break
if j == len(palabra):
return 'SINTIPO'
return palabra[j:len(palabra)]

# Funcion que devuelve NUMBER_FLOAT o NUMBER_INT segun si en su entrada tiene algun Fl
def tipoNumber(*args):
if len(args) == 2:
palabra1 = args[0]
palabra2 = args[1]
if type(palabra1) == type(1.0) or type(palabra2) == type(1.0):
return 'NUMBER_FLOAT'
else: return 'NUMBER_INT'
else:
palabra1 = args[0]
if type(palabra1) == type(1.0):
return 'NUMBER_FLOAT'
```

```
else: return 'NUMBER_INT'

# Funcion que devuelve True si es o puede llegar a ser NUMBER
def esNumber(palabra):
return (palabra[0:6] == 'NUMBER' or palabra == 'CUALQUIER_TIPO')

# Funcion que devuelve True si es o puede llegar a ser REGISTRO
def esRegistro(palabra):
return (palabra[0:8] == 'REGISTRO' or palabra == 'CUALQUIER_TIPO')

# Funcion que devuelve True si es o puede llegar a ser VECTOR
def esVector(palabra):
    return (palabra[0:6] == 'VECTOR' or palabra == 'CUALQUIER_TIPO')

# Funcion que devuelve True si es o puede llegar a ser BOOL
def esBool(palabra):
    return (palabra == 'BOOL' or palabra == 'CUALQUIER_TIPO')

# Funcion que devuelve True si es o puede llegar a ser STRING
def esString(palabra):
    return (palabra == 'STRING' or palabra == 'CUALQUIER_TIPO')

# Funcion que accede al diccionario de variables y devuelve su tipo (Si esta definida
# Si no esta definida devuelve ND
def estaDefinida(key):
if key in variables_dict:
return variables_dict[key]
else: return 'ND'

# Funcion que accede al diccionario de variables y devuelve su tipo (Si esta definida
# Si no esta definida devuelve ND
def estaReg(key):
    if key in reg_dict:
        return reg_dict[key]
    else: return 'ND'

# Funcion que convierte a str su entrada en caso que sea un int
def toStrIfInt(var):
    if isinstance( var, int ):
       return str(var)
    else:
       return var

class ParserException(Exception):
        pass
```

```
#Producciones Generales

def p_programa_s_pp(p):
    'p : sentencia pp'
    if (p.lineno(1) == p.lineno(2)) and p[2][1] == 'COMENTARIO':
        p[0] = [p[1][0] + ' '  + p[2][0], 'ND']
    else:
        p[0] = [p[1][0] + '\n'  + p[2][0], 'ND']

def p_programa_coment_pp(p):
    'p : COMENTARIO p'
    p[0] = [p[1] + '\n'  + p[2][0], 'COMENTARIO']

def p_programa_ctl_p(p):
    'p : control pp'
    if (p.lineno(1) == p.lineno(2)) and p[2][1] == 'COMENTARIO':
        p[0] = [p[1][0] + ' '  + p[2][0], 'ND']
    else:
        p[0] = [p[1][0] + '\n'  + p[2][0], 'ND']

def p_pp_s_pp(p):
    'pp : sentencia pp'
    if (p.lineno(1) == p.lineno(2)) and p[2][1] == 'COMENTARIO':
        p[0] = [p[1][0] + ' '  + p[2][0], 'ND']
    else:
        p[0] = [p[1][0] + '\n'  + p[2][0], 'ND']

def p_pp_ctl_pp(p):
    'pp : COMENTARIO pp'
    p[0] = [p[1] + '\n'  + p[2][0], 'COMENTARIO']

def p_pp_comentario_p(p):
    'pp : control pp'
    if (p.lineno(1) == p.lineno(2)) and p[2][1] == 'COMENTARIO':
        p[0] = [p[1][0] + ' '  + p[2][0], 'ND']
    else:
        p[0] = [p[1][0] + '\n'  + p[2][0], 'ND']

def p_pp_empty(p):
    'pp : empty'
    p[0] = [p[1][0], 'ND']

def p_empty(p):
    'empty :'
    p[0] = ['', 'ND']
```

```
def p_sentencia_var_asig(p):
    'sentencia : var_asig PUNTOYCOMA'
    p[0] = [p[1][0] + ';', p[1][1]]


def p_sentencia_func(p):
    'sentencia : funcion PUNTOYCOMA'
    p[0] = [p[1][0] + ';', p[1][1]]

#Producciones para estructuras de control
def p_control_ifelse(p):
    'control : ifelse'
    p[0] = [p[1][0], 'ND']

def p_control_loop(p):
    'control : loop'
    p[0] = [p[1][0], 'ND']

def p_control_cond_term_var_asig_l(p):
    'control_cond_term : var_asig_l'
    p[0] = [ p[1][0],p[1][1]]

def p_control_cond_term_e_bool(p):
    'control_cond_term : exp_bool'
    p[0] = [ p[1][0],p[1][1]]

def p_control_cond_term_comp(p):
    'control_cond_term : comparacion'
    p[0] = [ p[1][0],p[1][1]]

def p_control_cond_term_ternario(p):
    'control_cond_term : operador_ternario'
    p[0] = [ p[1][0],p[1][1]]

def p_loop_while(p):
    'loop : WHILE LPAREN control_cond_term RPAREN bloque'
    p[0] = ['while('+ p[3][0] + ')\n    ' + find_and_replace(p[5][0]), 'ND']

    if not esBool(p[3][1]):
        pass
        raise SemanticException('LOOP',p.lineno(1),p.lexpos(1))

def p_loop_do(p):
    'loop : DO bloque WHILE LPAREN control_cond_term RPAREN PUNTOYCOMA'
    p[0] = ['do\n    ' + find_and_replace(p[2][0]) + '\nwhile(' + p[5][0] + ');' +'\n
```

```
    if not esBool(p[5][1]):
        pass
        raise SemanticException('LOOP',p.lineno(1),p.lexpos(1))

def p_loop_for(p):
    'loop : for'
    p[0] = [ p[1][0],'ND']

def p_for_main(p):
    'for : FOR LPAREN form_term PUNTOYCOMA form_term_2 PUNTOYCOMA form_term RPAREN blo
    p[0] = ['for(' + p[3][0] + ';' + p[5][0] + ';' + p[7][0] +')\n    ' + find_and_rep

    if not esBool(p[5][1]):
        pass
        raise SemanticException('LOOP',p.lineno(1),p.lexpos(1))

def p_for_term(p):
    'form_term : var_asig '
    p[0] = [ p[1][0],p[1][1]]

def p_for_term_2_val(p):
    'form_term_2 : valores '
    p[0] = [ p[1][0],p[1][1]]

def p_for_term_2_comp(p):
    'form_term_2 : comparacion'
    p[0] = [ p[1][0],p[1][1]]

def p_for_term_empty(p):
    'form_term : '
    p[0] = ['','ND']

def p_ifelse(p):
    'ifelse : IF LPAREN control_cond_term RPAREN bloque ELSE bloque'
    p[0] = ['If(' + p[3][0] + ')\n    ' + find_and_replace(p[5][0]) + '\nelse\n    ' +

    if not esBool(p[3][1]):
        pass
        raise SemanticException('IF',p.lineno(1),p.lexpos(1))

def p_ifSinElse(p):
    'ifelse : IF LPAREN control_cond_term RPAREN bloque'
    p[0] = ['If(' + p[3][0] + ')\n    ' + find_and_replace(p[5][0]) + '\n', 'ND']

    if not esBool(p[3][1]):
        pass
```

```
        raise SemanticException('IF',p.lineno(1),p.lexpos(1))

def p_bloque_cb(p):
    'bloque : COMENTARIO bloque'
    p[0] = [p[1] + '\n'  + p[2][0], 'COMENTARIO']

def p_bloque_s(p):
    'bloque : sentencia'
    p[0] = [p[1][0], 'ND']

def p_bloque_c(p):
    'bloque : control'
    p[0] = [p[1][0], 'ND']

def p_bloque_p(p):
    'bloque : LLAVEIZQ p LLAVEDER'
    p[0] = ['{' + p[2][0] + '}', 'ND']

#Producciones para funciones
def p_funcion_ret(p):
    'funcion : func_ret'
    p[0] = [p[1][0], p[1][1]]

def p_funcion_void(p):
    'funcion : func_void '
    p[0] = [p[1][0], 'ND']

def p_func_void(p):
    'func_void : PRINT LPAREN valores RPAREN'
    p[0] = ['print(' + p[3][0] + ')', 'ND']

def p_funcion_ret_int(p):
    'func_ret : func_ret_int'
    p[0] = [p[1][0], p[1][1]]

def p_funcion_ret_cadena(p):
    'func_ret : func_ret_cadena'
    p[0] = [p[1][0], p[1][1]]

def p_funcion_ret_bool(p):
    'func_ret : func_ret_bool'
    p[0] = [p[1][0], p[1][1]]

def p_funcion_ret_arreglo(p):
    'func_ret : func_ret_arreglo'
    p[0] = [p[1][0], p[1][1]]
```

```
def p_funcion_ret_arreglo_3(p):
    'func_ret_arreglo : MULTIPLICACIONESCALAR LPAREN valores COMA valores COMA valores
    p[0] = ['multiplicacionEscalar(' + p[3][0] + ',' + p[5][0] + ',' + p[7][0] + ')',

    if (not esNumber(tipo(p[3][1])) and p[3][1] != "VECTOR_VACIO") or not esNumber(p[5
        pass
        raise SemanticException('MULTIPLICACIONESCALAR',p.lineno(3),p.lexpos(3))

def p_funcion_ret_arreglo_2(p):
    'func_ret_arreglo : MULTIPLICACIONESCALAR LPAREN valores COMA valores RPAREN'
    p[0] = ['multiplicacionEscalar(' + p[3][0] + ',' + p[5][0] + ')', 'VECTOR_NUMBER_

    if (not esNumber(tipo(p[3][1])) and p[3][1] != "VECTOR_VACIO") or not esNumber(p[5
        pass
        raise SemanticException('MULTIPLICACIONESCALAR',p.lineno(3),p.lexpos(3))

def p_funcion_ret_int_length(p):
    'func_ret_int : LENGTH LPAREN valores RPAREN'
    p[0] = ['length(' + p[3][0] + ')', 'NUMBER_INT']

    if (not esString(p[3][1]) and not esVector(p[3][1])):
        pass
        raise SemanticException('LENGTH',p.lineno(3),p.lexpos(3))


def p_funcion_ret_string(p):
    'func_ret_cadena : CAPITALIZAR LPAREN valores RPAREN'
    p[0] = ['capitalizar(' + p[3][0] + ')', 'STRING']

    if not esString(p[3][1]):
        pass
        raise SemanticException('CAPITALIZAR',p.lineno(3),p.lexpos(3))

def p_funcion_ret_string_2(p):
    'func_ret_cadena : CAPITALIZAR LPAREN operador_ternario RPAREN'
    p[0] = ['capitalizar(' + p[3][0] + ')', 'STRING']

    if not esString(p[3][1]):
        pass
        raise SemanticException('CAPITALIZAR',p.lineno(3),p.lexpos(3))

def p_funcion_ret_bool_f(p):
    'func_ret_bool : COLINEALES LPAREN valores COMA valores RPAREN '
    p[0] = ['colineales(' + p[3][0] + ',' + p[5][0] + ')', 'BOOL']
```

```
    if (not esNumber(tipo(p[3][1])) and p[3][1] != "VECTOR_VACIO") or (not esNumber(t
        pass
        raise SemanticException('COLINEALES',p.lineno(2),p.lexpos(2))

#Producciones para vectores y variables
def p_valores_exp_arit(p):
    'valores : exp_arit'
    p[0] = [toStrIfInt(p[1][0]), p[1][1]]

def p_valores_exp_bool(p):
    'valores : exp_bool'
    p[0] = [p[1][0], 'BOOL']

def p_valores_exp_cadena(p):
    'valores : exp_cadena'
    p[0] = [p[1][0], 'STRING']

def p_valores_exp_arreglo(p):
    'valores : exp_arreglo'
    p[0] = [p[1][0], p[1][1]]


def p_valores_reg(p):
    'valores : reg'
    p[0] = [p[1][0], p[1][1]]

def p_valores_reg2(p):
    'valores : reg PUNTO VARIABLE'
    p[0] = [p[1][0] + '.' + p[3], 'CUALQUIER_TIPO']
    if estaReg(p[3]) == 'ND':
        pass
        raise SemanticException('REGISTRO',p.lineno(1),p.lexpos(1))

def p_valores_variables(p):
    'valores : var_asig_l'
    p[0] = [p[1][0], p[1][1]]


def p_valor_perador(p):
    'valores : LPAREN operador_ternario RPAREN '
    p[0] = [ '(' + p[2][0] + ')',p[2][1]]

def p_exp_arreglo(p):
    'exp_arreglo : LCORCHETE lista_valores RCORCHETE'
    p[0] = ['[' + toStrIfInt(p[2][0]) +  ']', 'VECTOR_' + p[2][1]]
```

```
def p_exp_arreglo_vacio(p):
    'exp_arreglo : LCORCHETE RCORCHETE'
    p[0] = ['[]', 'VECTOR_VACIO']

def p_exp_arreglo_mult_escalar(p):
    'exp_arreglo : func_ret_arreglo'
    p[0] = [p[1][0], p[1][1]]

def p_lista_valores_end(p):
    'lista_valores : valores'
    p[0] = [p[1][0], p[1][1]]

def p_lista_pt_end(p):
    'lista_valores : operador_ternario'
    p[0] = [p[1][0], p[1][1]]

def p_lista_valores_lista(p):
    'lista_valores : valores COMA lista_valores'

    tipo_lista = p[3][1]
    if p[1][1] != 'ND':
     tipo_lista = p[1][1]
    p[0] = [p[1][0] + ',' + p[3][0], tipo_lista]

    if (p[1][1] != p[3][1] and p[1][1] != 'ND' and p[3][1] != 'ND' and p[1][1] != 'CU
        pass
        raise SemanticException('LISTAINCORRECTA',p.lineno(1),p.lexpos(1))

def p_lista_ot_lista(p):
    'lista_valores : operador_ternario COMA lista_valores'

    tipo_lista = p[3][1]
    if p[1][1] != 'ND':
     tipo_lista = p[1][1]

    p[0] = [p[1][0] + ',' + p[3][0], tipo_lista]

    if (p[1][1] != p[3][1] and p[1][1] != 'ND' and p[3][1] != 'ND'):
        pass
        #raise SemanticException('LISTAINCORRECTA',p.lineno(1),p.lexpos(1))

#Producciones Registros
def p_reg(p):
    'reg : LLAVEIZQ reg_item LLAVEDER'
    p[0] = ['{' + p[2][0] + '}', 'REGISTRO']
```

```
def p_reg_item_list(p):
    'reg_item : VARIABLE DOSPUNTOS valores COMA reg_item'
    p[0] = [p[1] + ":" + toStrIfInt(p[3][0]) + ',' + p[5][0], 'ND']
    reg_dict[p[1]] = p[3][1]


def p_reg_item(p):
    'reg_item : VARIABLE DOSPUNTOS valores'
    p[0] = [p[1] + ":" + toStrIfInt(p[3][0]), 'ND']
    reg_dict[p[1]] = p[3][1]


#Producciones de asignaciones
def p_var_asig_l_var(p):
    'var_asig_l : VARIABLE'
    p[0] = [p[1],estaDefinida(p[1])]


def p_var_asig_l_res(p):
    'var_asig_l : RES'
    p[0] = [p[1],estaDefinida(p[1])]

def p_var_asig_l_var_mem(p):
    'var_asig_l : VARIABLE var_member'
    p[0] = [p[1] + p[2][0],'CUALQUIER_TIPO']
    if not esVector(estaDefinida(p[1])) and not esRegistro(estaDefinida(p[1])):
        pass
        raise SemanticException('NODEFINIDA',p.lineno(1),p.lexpos(1))


def p_var_member_vec_item_rec(p):
    'var_member : LCORCHETE var_asig_l RCORCHETE var_member'
    p[0] = ['[' + p[2][0] + ']' + p[4][0],'ND']
    if tipo(p[2][1]) != 'INT' and p[2][1] != 'CUALQUIER_TIPO':
        pass
        raise SemanticException('INDEX_NOT_NAT',p.lineno(2),p.lexpos(2))


def p_var_member_vec_item_2(p):
    'var_member : LCORCHETE exp_arit RCORCHETE'
    p[0] = ['[' + p[2][0] + ']','ND']
    if tipo(p[2][1]) != 'INT':
        pass
        raise SemanticException('INDEX_NOT_NAT',p.lineno(2),p.lexpos(2))

def p_var_member_vec_item_2_rec(p):
```

```
    'var_member : LCORCHETE exp_arit RCORCHETE var_member'
    p[0] = ['[' + p[2][0] + ']' + p[4][0], 'ND']
    if tipo(p[2][1]) != 'INT':
        pass
        raise SemanticException('INDEX_NOT_NAT',p.lineno(2),p.lexpos(2))

def p_var_member_vec_item_3(p):
    'var_member : LCORCHETE var_asig_l RCORCHETE'
    p[0] = ['[' + p[2][0] + ']', 'ND']
    if tipo(p[2][1]) != 'INT' and p[2][1] != 'CUALQUIER_TIPO':
        pass
        raise SemanticException('INDEX_NOT_NAT',p.lineno(2),p.lexpos(2))

def p_var_member_reg_item(p):
    'var_member : PUNTO VARIABLE'
    p[0] = [ '.' + p[2], 'ND']

    if estaReg(p[2]) == 'ND':
        pass
        raise SemanticException('REGISTRO',p.lineno(2),p.lexpos(2))


def p_var_member_reg_item_rec(p):
    'var_member : PUNTO VARIABLE var_member'
    p[0] = [ '.' + p[2] + p[3][0], 'ND' ]

    if estaReg(p[2]) == 'ND':
        pass
        raise SemanticException('REGISTRO',p.lineno(2),p.lexpos(2))

def p_var_asig_base_mm(p):
    'var_asig : var_asig_l LESSLESS'
    p[0] = [toStrIfInt(p[1][0]) + '--', p[1][1]]

    if not esNumber(p[1][1]):
        pass
        raise SemanticException('NODEFINIDA',p.lineno(1),p.lexpos(1))


def p_var_asig_mm_base(p):
    'var_asig : LESSLESS var_asig_l'
    p[0] = ['--' + toStrIfInt(p[2][0]), p[2][1]]

    if not esNumber(p[2][1]):
        pass
        raise SemanticException('NODEFINIDA',p.lineno(1),p.lexpos(1))
```

```
def p_var_asig_base_pp(p):
    'var_asig : var_asig_l MASMAS'
    p[0] = [toStrIfInt(p[1][0]) + '++', p[1][1]]

    if not esNumber(p[1][1]):
        pass
        raise SemanticException('NODEFINIDA',p.lineno(1),p.lexpos(1))


def p_var_asig_pp_base(p):
    'var_asig : MASMAS var_asig_l '
    p[0] = ['++' + toStrIfInt(p[2][0]), p[2][1]]

    if not esNumber(p[2][1]):
        pass
        raise SemanticException('NODEFINIDA',p.lineno(1),p.lexpos(1))


def p_var_asig_multipl(p):
    'var_asig : var_asig_l MULTIPL valores'
    p[0] = [p[1][0] + '=*' + toStrIfInt(p[3][0]), tipoNumber(p[1][1],p[3][1])]

    if not esNumber(p[1][1]) or not esNumber(p[3][1]):
        pass
        raise SemanticException('NODEFINIDA',p.lineno(1),p.lexpos(1))

def p_var_asig_dividi(p):
    'var_asig : var_asig_l DIVIDI valores'
    p[0] = [p[1][0] + '=/' + toStrIfInt(p[3][0]), tipoNumber(p[1][1],p[3][1])]

    if not esNumber(p[1][1]) or not esNumber(p[3][1]):
        pass
        raise SemanticException('NODEFINIDA',p.lineno(1),p.lexpos(1))

def p_var_asig_agregar(p):
    'var_asig : var_asig_l AGREGAR valores'
    p[0] = [p[1][0] + '+=' + toStrIfInt(p[3][0]), tipoNumber(p[1][1],p[3][1])]

    if (esNumber(p[1][1]) and esNumber(p[1][1])) or (p[1][1] == "STRING" and p[3][1] =
        pass
    else:
        pass
        raise SemanticException('NODEFINIDA',p.lineno(1),p.lexpos(1))
```

```
def p_var_asig_sacar(p):
    'var_asig : var_asig_l SACAR valores'
    p[0] = [p[1][0] + '-=' + toStrIfInt(p[3][0]), tipoNumber(p[1][1],p[3][1])]

    if not esNumber(p[1][1]) or not esNumber(p[3][1]):
        pass
        raise SemanticException('NODEFINIDA',p.lineno(1),p.lexpos(1))

def p_var_asig(p):
    'var_asig : var_asig_l ASIGNACION valores'
    p[0] = [p[1][0] + '=' + toStrIfInt(p[3][0]), 'ASIGNACION']
    variables_dict[p[1][0]] = p[3][1]

def p_var_comparacion(p):
    'var_asig : var_asig_l ASIGNACION comparacion'
    p[0] = [p[1][0] + '=' + toStrIfInt(p[3][0]), 'ASIGNACION']
    variables_dict[p[1][0]] = p[3][1]

def p_var_op_ternario(p):
    'var_asig : var_asig_l ASIGNACION operador_ternario'
    p[0] = [p[1][0] + '=' + toStrIfInt(p[3][0]), 'ASIGNACION']
    variables_dict[p[1][0]] = p[3][1]

# En asignacion no importa el tipo, por mas que tengas una variable 'aux' del tipo que
# aux = 10; deberia ser valido

def p_operador_ternario(p):
    'operador_ternario : valores INTERROGACION valores DOSPUNTOS valores'
    p[0] = [  p[1][0] + ' ? ' + p[3][0] + ':' + p[5][0],p[3][1]]
    if not esBool(p[1][1]) or (p[3][1] != p[5][1] and not(esNumber(p[3][1]) and esNumb
            raise SemanticException('OPTERNARIO',p.lineno(1),p.lexpos(1))

def p_operador_ternario_2(p):
    'operador_ternario : comparacion INTERROGACION valores DOSPUNTOS valores'
    p[0] = [  p[1][0] + ' ? ' + p[3][0] + ':' + p[5][0],p[3][1]]
    if not esBool(p[1][1]) or (p[3][1] != p[5][1] and not(esNumber(p[3][1]) and esNumb
            raise SemanticException('OPTERNARIO',p.lineno(1),p.lexpos(1))

def p_operador_ternario_3(p):
    'operador_ternario : valores INTERROGACION valores DOSPUNTOS operador_ternario'
    p[0] = [  p[1][0] + ' ? ' + p[3][0] + ':' + p[5][0],p[3][1]]
    if not esBool(p[1][1]) or (p[3][1] != p[5][1] and not(esNumber(p[3][1]) and esNumb
        raise SemanticException('OPTERNARIO',p.lineno(1),p.lexpos(1))

def p_operador_ternario_4(p):
    'operador_ternario : comparacion INTERROGACION valores DOSPUNTOS operador_ternario
```

```
    p[0] = [  p[1][0] + ' ? ' + p[3][0] + ':' + p[5][0],p[3][1]]
    if not esBool(p[1][1]) or (p[3][1] != p[5][1] and not(esNumber(p[3][1]) and esNumb
            raise SemanticException('OPTERNARIO',p.lineno(1),p.lexpos(1))

def p_oper_var(p):
    'var_oper : var_asig_l'
    p[0] = [p[1][0] , p[1][1]]

def p_oper_ternaerio(p):
    'var_oper : LPAREN operador_ternario RPAREN '
    p[0] = [ '(' + p[2][0] + ')', p[2][1]]

def p_oper_arreglo(p):
    'var_oper : exp_arreglo'
    p[0] = [p[1][0] , p[1][1]]

def p_oper_reg_correcto(p):
    'var_oper : reg PUNTO VARIABLE'
    p[0] = [p[1][0] , 'CUALQUIER_TIPO']

def p_exp_arregloValor(p):
    'exp_arreglo : LCORCHETE lista_valores RCORCHETE exp_arreglo'
    p[0] = ['[' + toStrIfInt(p[2][0]) +  ']' + p[4][0], 'CUALQUIER_TIPO']

#Producciones operaciones binarias con Numeros


def p_exp_arit_ept(p):
    'exp_arit : exp_arit PLUS term'
    p[0] = [p[1][0] + ' + ' + toStrIfInt(p[3][0]), tipoNumber(p[1][1],p[3][1])]


def p_exp_arit_epv2(p):
    'exp_arit : exp_arit PLUS var_oper'
    p[0] = [p[1][0] + ' + ' + p[3][0], tipoNumber(p[1][1],p[3][1])]

    if not esNumber(p[3][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))

def p_exp_arit_v2pt(p):
    'exp_arit : var_oper PLUS term'
    p[0] = [p[1][0] + ' + ' + toStrIfInt(p[3][0]), tipoNumber(p[1][1],p[3][1])]

    if not esNumber(p[1][1]):
        pass
```

```
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))

def p_exp_arit_v2pv2(p):
    'exp_arit : var_oper PLUS var_oper'
    p[0] = [p[1][0] + ' + ' + toStrIfInt(p[3][0]), tipoNumber(p[1][1],p[3][1])]

    if not esNumber(p[1][1]) or not esNumber(p[3][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))


def p_exp_arit_emt(p):
    'exp_arit : exp_arit MINUS term'
    p[0] = [p[1][0] + ' - ' + toStrIfInt(p[3][0]), tipoNumber(p[1][1],p[3][1])]


def p_exp_arit_emv2(p):
    'exp_arit : exp_arit MINUS var_oper'
    p[0] = [p[1][0] + ' - ' + p[3][0], tipoNumber(p[1][1],p[3][1])]

    if not esNumber(p[3][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))


def p_exp_arit_v2mt(p):
    'exp_arit : var_oper MINUS term'
    p[0] = [p[1][0] + ' - ' + toStrIfInt(p[3][0]), tipoNumber(p[1][1],p[3][1])]

    if not esNumber(p[1][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))


def p_exp_arit_v2mv2(p):
    'exp_arit : var_oper MINUS var_oper'
    p[0] = [p[1][0] + ' - ' + toStrIfInt(p[3][0]),  tipoNumber(p[1][1],p[3][1])]

    if not esNumber(p[1][1]) or not esNumber(p[3][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))

def p_exp_arit_term(p):
    'exp_arit : term'
    p[0] = [toStrIfInt(p[1][0]), p[1][1]]
```

```
def p_arit_oper2_times(p):
    'arit_oper_2 : TIMES'
    p[0] = [p[1], 'ND' ]

def p_arit_oper2_div(p):
    'arit_oper_2 : DIV'
    p[0] = [p[1], 'ND' ]

def p_arit_oper2_mod(p):
    'arit_oper_2 : MODULO'
    p[0] = [p[1], 'ND' ]

def p_term_tmf(p):
    'term : term arit_oper_2 factor'
    p[0] = [p[1][0] + p[2][0] + toStrIfInt(p[3][0]), tipoNumber(p[1][1],p[3][1])]

    if p[2][0] == '/' and p[3][0] == 0:
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))

def p_term_tmv2(p):
    'term : term arit_oper_2 var_oper'
    p[0] = [p[1][0] + p[2][0] + p[3][0], tipoNumber(p[1][1],p[3][1])]

    if not esNumber(p[3][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))

def p_term_v2mf(p):
    'term : var_oper arit_oper_2 factor'
    p[0] = [p[1][0] + p[2][0] + toStrIfInt(p[3][0]), tipoNumber(p[1][1],p[3][1])]

    if not esNumber(p[1][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))

def p_term_v2mv2(p):
    'term : var_oper arit_oper_2 var_oper'
    p[0] = [p[1][0] + p[2][0] + p[3][0], tipoNumber(p[1][1],p[3][1])]

    if not esNumber(p[1][1]) or not esNumber(p[3][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))

def p_term_factor(p):
    'term : factor'
```

```
    p[0] = [toStrIfInt(p[1][0]), p[1][1]]

def p_factor_base_exp(p):
    'factor : base ELEVADO sigexp'
    p[0] = [p[1][0] + ' ^' + toStrIfInt(p[3][0]), tipoNumber(p[1][1],p[3][1])]


def p_factor_var_op__exp(p):
    'factor : var_oper ELEVADO sigexp'
    p[0] = [p[1][0] + ' ^' + toStrIfInt(p[3][0]), tipoNumber(p[1][1],p[3][1])]

    if not esNumber(p[1][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))

def p_factor_base(p):
    'factor : base '
    p[0] = [toStrIfInt(p[1][0]), p[1][1]]

def p_factor_m_base(p):
    'factor : MINUS base '
    p[0] = ['-' + toStrIfInt(p[2][0]), 'NUMBER_FLOAT']

def p_factor_m_var_oper(p):
    'factor : MINUS var_oper'
    p[0] = ['-' + p[2][0], 'NUMBER_FLOAT']

    if not esNumber(p[2][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))

def p_factor_var_op_mm(p):
    'factor : var_oper LESSLESS'
    p[0] = [toStrIfInt(p[1][0]) + '--', p[1][1]]

    if not esNumber(p[1][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))


def p_factor_base_mm(p):
    'factor : base LESSLESS'
    p[0] = [toStrIfInt(p[1][0]) + '--', p[1][1]]

def p_factor_mm_var_op(p):
    'factor : LESSLESS var_oper'
```

```
    p[0] = ['--' + toStrIfInt(p[2][0]), p[2][1]]

    if not esNumber(p[2][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))

def p_factor_mm_base(p):
    'factor : LESSLESS base '
    p[0] = ['--' + p[2][0], p[2][1]]

def p_factor_var_op_pp(p):
    'factor : var_oper MASMAS'
    p[0] = [p[1][0] + '++', p[1][1]]

    if not esNumber(p[1][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))

def p_factor_base_pp(p):
    'factor : base MASMAS'
    p[0] = [toStrIfInt(p[1][0]) + '++', p[1][1]]

def p_factor_pp_var_op(p):
    'factor : MASMAS var_oper'
    p[0] = ['++' + p[2][0], p[2][1]]

    if not esNumber(p[2][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))

def p_factor_pp_base(p):
    'factor : MASMAS base '
    p[0] = ['++' + p[2][0], p[2][1]]

def p_base_expr(p):
    'base : LPAREN exp_arit RPAREN'
    p[0] = ['(' + p[2][0] + ')', p[2][1]]

def p_base_paren_var_oper(p):
    'base : LPAREN var_oper RPAREN'
    p[0] = ['(' + p[2][0] + ')', p[2][1]]

    if not esNumber(p[2][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))
```

```
def p_base_valor(p):
    'base : NUMBER'
    p[0] =  [toStrIfInt(p[1]), tipoNumber(p[1])]

def p_base_func_ret_int(p):
    'base : func_ret_int'
    p[0] =  [p[1][0], p[1][1]]

def p_sigexp_m(p):
    'sigexp : MINUS exp'
    p[0] = ['-' + p[2][0], p[2][1]]

def p_sigexp_exp(p):
    'sigexp : exp'
    p[0] =  [p[1][0], p[1][1]]

def p_exp_var_op(p):
    'exp : var_oper'
    p[0] =  [p[1][0], p[1][1]]

    if not esNumber(p[1][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))


def p_exp_valor(p):
    'exp : NUMBER'
    p[0] =  [toStrIfInt(p[1]), tipoNumber(p[1])]

def p_exp__expr(p):
    'exp : LPAREN exp_arit RPAREN'
    p[0] = ['(' + p[2][0] + ')', p[2][1]]


#Producciones operaciones con Strings
def p_exp_cadena_concat(p):
    'exp_cadena : exp_cadena PLUS term_cadena'
    p[0] = [p[1][0] + ' + ' +  p[3][0], 'STRING']

def p_exp_cadena_concat_1(p):
    'exp_cadena : var_oper PLUS term_cadena'
    p[0] = [p[1][0] + ' + ' +  p[3][0], 'STRING']

    if not esString(p[1][1]):
        pass
```

```
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))


def p_exp_cadena_concat_2(p):
    'exp_cadena : exp_cadena PLUS var_oper'
    p[0] = [p[1][0] + ' + ' +  p[3][0], 'STRING']

    if not esString(p[3][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))


def p_exp_cadena_term(p):
    'exp_cadena : term_cadena'
    p[0] = [p[1][0], 'STRING']

def p_exp_cadena_cadena(p):
    'term_cadena : CADENA'
    p[0] = [p[1], 'STRING' ]

def p_exp_cadena_funct_ret_string(p):
   'term_cadena : func_ret_cadena'
   p[0] =  [p[1][0], p[1][1]]

def p_exp_cadena_parent(p):
    'term_cadena : LPAREN exp_cadena RPAREN'
    p[0] = ['(' + p[2][0] + ')', 'STRING']

#Producciones de operaciones booleanas

def p_bool_expr_eat(p):
    'exp_bool : exp_bool AND term_bool'
    p[0] = [p[1][0] + ' and ' + p[3][0], 'BOOL']


def p_bool_expr_v2af(p):
    'exp_bool : var_oper AND term_bool'
    p[0] = [p[1][0] + ' and ' + p[3][0], 'BOOL']

    if not esBool(p[1][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))

def p_bool_expr_eav2(p):
    'exp_bool : exp_bool AND var_oper'
    p[0] = [p[1][0] + ' and ' + p[3][0], 'BOOL']
```

```
    if not esBool(p[3][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))


def p_bool_expr_v2av2(p):
    'exp_bool : var_oper AND var_oper'
    p[0] = [p[1][0] + ' and ' + p[3][0], 'BOOL']

    if not esBool(p[3][1]) or not esBool(p[1][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))


def p_bool_expr_term(p):
    'exp_bool : term_bool'
    p[0] = [p[1][0], 'BOOL' ]

def p_bool_tof(p):
    'term_bool : term_bool OR factor_bool'
    p[0] = [p[1][0] + ' or ' + p[3][0], 'BOOL']

def p_bool_tov2(p):
    'term_bool : term_bool OR var_oper'
    p[0] = [p[1][0] + ' or ' + p[3][0], 'BOOL']

    if not esBool(p[3][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))


def p_bool_v2of(p):
    'term_bool : var_oper OR factor_bool'
    p[0] = [p[1][0] + ' or ' + p[3][0], 'BOOL']

    if not esBool(p[1][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))


def p_bool_v2ov2(p):
    'term_bool : var_oper OR var_oper'
    p[0] = [p[1][0] + ' or ' + p[3][0], 'BOOL']

    if not esBool(p[1][1]) or not esBool(p[3][1]):
```

```
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))

def p_bool_term_factor(p):
    'term_bool : factor_bool'
    p[0] = [p[1][0], 'BOOL']

def p_term_not(p):
    'term_bool : NOT factor_bool'
    p[0] = ['not ' + p[2][0], 'BOOL']

def p_term_not_var_oper(p):
    'term_bool : NOT var_oper'
    p[0] = ['not ' + p[2][0], 'BOOL']

    if not esBool(p[2][1]):
        pass
        raise SemanticException('ERRORTIPO',p.lineno(1),p.lexpos(1))


def p_term_bool_parentesis(p):
    'factor_bool : LPAREN comparacion RPAREN'
    p[0] = ['(' + p[2][0] + ')', 'BOOL']

def p_term_bool_parentesis2(p):
    'factor_bool : LPAREN exp_bool RPAREN'
    p[0] = ['(' + p[2][0] + ')', 'BOOL']

def p_term_bool_bool(p):
    'factor_bool : BOOL'
    p[0] = [str(p[1]), 'BOOL']

def p_term_bool_func(p):
    'factor_bool : func_ret_bool'
    p[0] = [str(p[1][0]), 'BOOL']

def p_operador_comparacion_igual(p):
    'operador_comp : IGUAL'
    p[0] = [' == ', 'ND']

def p_operador_comparacion_mayor(p):
    'operador_comp : MAYOR'
    p[0] = [' > ', 'ND']

def p_operador_comparacion_menor(p):
    'operador_comp : MENOR'
```

```
    p[0] = [' < ', 'ND']

def p_operador_comparacion_dif(p):
    'operador_comp : DISTINTO'
    p[0] = [' != ', 'ND']

def p_comparcion(p):
    'comparacion : valores operador_comp valores'
    p[0] = [p[1][0] + p[2][0] + p[3][0], 'BOOL']

def p_error(token):
    message = "[Syntax error]"
    if token is not None:
        message += "\ntype:" + token.type
        message += "\nvalue:" + str(token.value)
        message += "\nline:" + str(token.lineno)
        message += "\nposition:" + str(token.lexpos)
    raise Exception(message)
```

4.3. Código de semantic_error.py

```
semantic_errors =  {
        'INDEX_NOT_NAT' : 'El indice del vector debe ser un numero natural',
        'MATH_ERROR' : 'No esta definida esa operacion matematica',
        'CAPITALIZAR' : 'Argumentos incorrectos, Capitalizar(STRING)',
        'MULTIPLICACIONESCALAR' : 'Argumentos incorrectos, multiplicacionEscalar(VECT
        'COLINEALES' : 'Argumentos incorrectos, colineales(VECTOR[NUMBER],VECTOR[NUMB
        'PRINT' : 'Argumentos incorrectos, la funcion print recibe solo un argumento'
        'LENGTH' : 'Argumentos incorrectos, length(STRING) o length(VECTOR)',
        'LISTAINCORRECTA' : 'El vector debe tener todos sus elementos del mismo tipo'
        'OPTERNARIO' : 'El operador ternario debe devolver en ambos casos un elemento
        'COMPINVALIDA' : 'Comparacion invalida, error de tipos.',
        'ERRORTIPO' : 'Operacion invalida, error de tipos.',
        'NODEFINIDA' : 'Variable no definida.',
        'REGISTRO' : 'Esta intentando acceder a un valor que no esta en el registro.'
        'LOOP' : 'Error en un ciclo. La guarda no es de tipo BOOL',
        'IF' : 'Error en un IF. La guarda no es de tipo BOOL',
        }

class SemanticException(Exception):
    def __init__(self,msg_id,lineno,lexpos):
```

```
        super(SemanticException,self).__init__('Error semantico en la linea ' + str(li
```

5. Casos de Prueba

A continuación presentaremos una serie de casos correctos e incorrectos respecto a lo descrito en las cosignas del trabajo práctico. Cada test tiene como propósito corroborar el buen funcionamiento de los distintas partes de nuestro programa. Intentamos poner a prueba la capacidad del parser para discernir entre un texto con buena sintaxis y uno mal escrito, también con el lexer probamos que se excluyan las palabras reservadas y por último que los posibles errores semánticos sean detectados.

A parte de estos tests, utilizamos algunos mas creados por nosotros y tambien los proporcionados por la catedra. Nuestro programa funciona correctamente para todos ellos.

5.1. Casos Correctos.

5.1.1. Caso 1:

```
#NADA
z = [a.edad + length("doce"+capitalizar("2")) / 2, 1.0
+ (siete ? ((aa OR bb AND (2!=length(""))?1:2)):
g*12 %3) , NOT aa ? b / -5 : c]; while(dos);

a = 0;
b = 1;
a = b;
c = 2.5;
d = "Hola";
e = d;

f = [1, 2, 3];
a = f[1];
b = f[2];

g = [1, a, 2, b, 3];
g[a] = b;
g[b] = a;

h = [f[g[a]], g[g[b]]];
h[f[g[a]]] = g[f[g[g[b]]]];
```

5.1.2. Caso 2:

```
for (i=0; i<10; i++) {
while(true) {
for(;true;) {
do {
a=0;
}
while(false);
}
}
}
```

5.1.3. Caso 3:

```
a = -10;
b = 2 + 3 - 8 + 1;
c = 3 * 5 % 4 / 6;
d = 2.0 ^ 9.0;
e = -3 * (4 / ((3) + 8 ^ ((1))) - -5 + -(7) * 6);

f = -a * (b / ((a) + d ^ ((c))) - -b + -(c) * a);

g = [1.5, 2 * (7) + 5 / 0, a * d, 8 ^ 1.0, (6)
 / (6), (3 + 3) + 0.0, e - 5, 2 * -f];
h = [1,b,3];

g[2 * 4] = 10 + (3 * 2/1);
g[b ^ b] = (g[7 * (5) + (h[(2)])]);
g[9 - (5 % 3)] = g[b % h[b]];

while (true) {
d = 1;
a = 0;
}

do {
# Co
# men
# tario
b = 2;
c = 3;
} while ((d > 10) AND NOT false);
```

```
usuario = {nombre:"Al", edad:50};
print(capitalizar(usuario.nombre));
nacimiento = 2016;
nacimiento -= usuario.edad;
usuarios = [{nombre:"Mr.X", edad:10}, usuario];

suma = 0;
for (i = 0; i < length(usuarios); i++) {
print(usuarios[i].nombre);
suma += usuarios[i].edad;
}
```

5.2. Casos incorrectos.

5.2.1. Caso 1.

```
else
  b=0;
```

5.2.2. Caso 2.

```
while(true){
#comentario sin sentencias
}
```

5.2.3. Caso 3.

```
for(true){}
```

5.2.4. Caso 4.

```
While = 10;
```

5.2.5. Caso 5.

```
a = 5
```

5.2.6. Caso 6.

```
multiplicacionEscalar("1,2,3");
```

5.2.7. Caso 7.

```
multiplicacionEscalar();
```

6. Conclusión