DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

SLS Parser

Teoría de Lenguajes

Grupo 11: El Niño Rata

Integrante	LU	Correo electrónico
Bonet, Felipe	668/08	fpbonet@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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

1. Introducción

En este trabajo se desarrolló un analizador sintáctico de Simple Lenguaje de Scripting (SLS), un lenguaje ficticio de programación similar a ANSI C. Este analizador permite detectar errores de sintaxis y tipado en cadenas escritas en SLS. Adicionalmente, en caso de no haber errores el programa devuelve un pretty print del código de entrada, es decir, una versión del código con un formato de escritura más legible y estandarizado.

El analizador consta de dos componentes, un Lexer o analizador léxico que divide al input en pequeñas porciones de código (tokens) las cuales serán analizadas por el segundo componente, el Parser. Este último tiene como tarea el análisis sintáctico del programa, detectando posibles errores de sintaxis.

Para el desarrollo del analizador se utilizó el Framework de Python PLY. Este framework consiste de una implementación de LEX y YACC en python. Se provee en este trabajo entonces el código que utiliza esta herramienta.

1.1. Entrega

Junto con este informe se entrega una carpeta con el código fuente de la aplicación, **src**. En esta carpeta se puede encontrar en los archivos **lexer.py** y **parser.py** las fuentes de tanto lexer como parser.
Cada uno de estos elementos utiliza una serie de reglas requeridas por PLY. Dichas reglas se encuentran en **lexer_rules.py** y **parser_rules.py** respectivamente. Los archivos contienen comentarios que intentan facilitar su comprensión.

Adicionalmente, se entrega también un script de bash en el archivo **SLSParser**. Este archivo contiene el código utilizado para llamar al parser. Su funcionamiento se describe en la sección siguiente. El script se encuentra comentado en caso de desear revisarlo.

El código de todos estos archivos se incluye en la sección Apéndice, al final de este informe.

Por último, se entrega tambien una carpeta **testing** con algunos archivos de prueba utilizados para verificar el funcionamiento del parser. En estos archivos se encuentra código SLS, en algunos casos bien formado y en otros no. Los archivos contienen en su nombre el resultado esperado para el parser, ya sea PASA para el correcto funcionamiento o FALLA si el código contenido debería generar error de parseo.

1.2. Requisitos y Modo de uso

Para poder ejecutar este trabajo se requiere:

- Contar con un entorno Linux, con un usuario que tenga permisos de ejecución. Este trabajo fue desarrollado en Ubuntu 14.04.
- Tener instalado python. Este trabajo fue desarrollado usando Python v2.7.5ubuntu3 .
- Tener PLY instalado. Este trabajo fue desarrollado usando PLY v3.4 .
- Dar permiso de ejecución al archivo **SLSParser**

Una vez satisfechos los requisitos, el modo de uso es el siguiente. Se debe abrir una consola en la carpeta **src** y escribir:

```
./SLSParser [-o SALIDA] -c ENTRADA | FUENTE
```

Donde SALIDA es un archivo de output, ENTRADA archivo de input, FUENTE es un string con código SLS.

Este script recibe por parámetro de entrada código SLS. Este código puede estar en un archivo (en ese caso pasárselo como *-c ENTRADA*) o puede estar escrito directamente en la llamada. Para este segundo caso,

insertar el código SLS entre comillas.

Ejemplo de llamada con código SLS:

```
./SLSParser "a=100;if(true)b=100; else b=123;"
```

Este ejemplo imprimirá por pantalla el código pasado por parámetro, formateado.

Es obligatorio proveer el código de entrada por parámetro, en alguno de los dos formatos descriptos. El script puede recibir también por parámetro un archivo de salida de forma opcional. En caso de recibirlo, se escribe la salida del programa allí. La salida del programa es el código de entrada bien formateado. Si se desea proveer un archivo de salida, escribir en la llamda *-o SALIDA*.
Ejemplo de llamada con archivos de input y output:

```
./SLSParser -o output_file -c input_file
```

Este ejemplo analizará el código SLS en el archivo de entrada. En caso de haber un error, lo imprimirá por pantalla. Si no imprime nada por pantalla, el código es correcto y se imprime en el archivo de salida el pretty print.

Por último, es importante respetar el orden de los parámetros descripto en la llamada de ejemplo.

En caso de haber algún tipo de error o problema con el script de bash, se puede ejecutar el parser directamente de la siguiente forma:

```
python parser.py input output
```

Donde input y output son los archivos correspondientes a la cadena de entrada y de salida.

2. Desarrollo

En esta sección comentaré el proceso de desarrollo de la aplicación.

2.1. Lexer

Para el desarrollo de este trabajo fue necesario implementar dos módulos: el lexer y el parser. El primero de ellos, el lexer, se encarga de tomar la cadena de entrada y dividirla en tokens. Estos tokens serán luego utilizados como entrada del parser.

El código del lexer presente en **lexer.py** es exactamente el mismo provisto por la cátedra, así que no creo que merezca una explicación.
Las reglas del lexer son funciones python que definen cada uno de los distintos tokens. Para tomar una fraccion de la cadena de input como token, se definen las distintas regex en cada regla. Las reglas y regex se pueden observar en el código de **lexer_rules.py** en el apéndice.

Notas sobre el lexer:

- Para marcar las palabras reservadas, se utilizó el método recomendado por el manual de PLY y se designó una lista de palabras reservadas, de modo que si un ID matcheaba alguna de los elementos de la lista, el token se marcaba dicha palabra reservada. Se detectan palabras reservadas tanto en uppercase como lowercase y en ambos casos la salida respeta el formato en el que las encontró escritas.
- Se utiliza un mecanismo muy similar para las invocaciones a funciones. Se detectan los nombres de funciones con una lista de nombres espciales.
- Se ignoran los saltos de linea y las tabulaciones. Los primeros si se utilizan para avanzar el atributo "lineno" del lexer. Atributo que será utilizado por el parser para conocer las los números de línea originales de cada expresión.

2.2. Parser

En segundo lugar, el parser es el módulo encargado de analizar sintáctica y semánticamente el código. A grandes rasgos, el parser es una implementación de una TDS, una traducción dirigida por sintaxis. Este tipo de gramáticas fueron estudiadas en el curso a lo largo del cuatrimestre. Básicamente su particularidad es que cada símbolo no terminal cuenta con una serie de atributos utilizados para hacer chequeos semánticos en cada producción.

2.2.1. Gramática

Para poder implementar la TDS, primero fue necesario desarrollar una gramática que interprete cadenas de código SLS. Esta gramática debe aceptar cadenas bien formadas y rechazar las que tienen errores de sintaxis.

Para el desarrollo de mi gramática, tomé varias ideas y elementos de la gramática de ANSI C desarrollada en 1985 por Jeff Lee [1] para el borrador del standard ANSI C. Principalmente el sistema de clasificación de expresiones, el cual se va desarrollando escalonadamente de forma de marcar la precedencia de cada operación mediante las producciones.

Notas sobre la gramática: Aspectos Sintácticos

- La gramática esta casi totalmente libre de conflictos. Únicamente presenta el conficto Shift-reduce clásico del if-then vs if-then-else. Ver sección "Limitaciones Conocidas".
- Básicamente cuento con 3 tipos de elementos: comments, statements y expressions. Los comments representan comentarios, los statements son sentencias, bloques y estructuras de control de flujo ; y las expressions son expresiones más simples como una operacion unaria, binaria, un llamado a una función o un valor.

- Los statements y los comments pueden ir intercalados e incluso puede haber comments dentro de statements.
- Las expressions se van derivando de a niveles, comenzando con la expression más general, asignación, hasta las más básicas, las expresiones primarias. En el medio, los distintos símbolos de expresión marcan también la precedencia de cada operación.
- RETURN se toma como una instrucción. RES, BEGIN y END se toman como variables asignables. Esto último entra en conflicto con algunos tests de la cátedra. Sucede que en C++ Begin y End son expresiones que se usan para manipular punteros de una lista o vector. Al no especificar en el enunciado su utilización, las tomé como variables asignables, al igual que RES.
- No se permiten asignaciones en cadena estilo $a = b = c = d = 100$ ni tampoco $a, b, c = 100$.

Anexo a continuación la gramática desarrollada, en formato PLY-friendly.

```
statements :  COMMENT
           |  COMMENT statements
           |  statement
           |  statement statements

comments_statement : statement
                   | comments statement

comments : COMMENT
         | comments COMMENT

statement : statement_iter
              | statement_expression
              | statement_block
              | statement_if
              | statement_return

statement_iter : WHILE '(' expression ')' comments_statement
statement_iter : DO comments_statement WHILE '(' expression ')' ';'
statement_iter : FOR '(' statement_expression statement_expression
expression ')' comments_statement

statement_iter : FOR '(' statement_expression statement_expression ')'
comments_statement

statement_if :  IF '(' expression ')' comments_statement
                |  IF '(' expression ')' comments_statement ELSE comments_statement

statement_block : '{' statements '}'

statement_return : RETURN ';'
                 | RETURN expression ';'

statement_expression : ';'
                     | expression ';'

expression : assignment_expression
            | expression ',' assignment_expression

assignment_expression :  conditional_expression
                      |  unary_expression assign_operator conditional_expression

assign_operator  : '='
```

```
                | ADDASSIGN
                | SUBASSIGN
                | MULASSIGN
                | DIVASSIGN

conditional_expression : logical_or_expression
                         | logical_or_expression '?' expression ':' conditional_expression

logical_or_expression :  logical_and_expression
                         |  logical_or_expression OR logical_and_expression

logical_and_expression : equality_expression
                         | logical_and_expression AND equality_expression

equality_expression : relational_expression
                       | equality_expression EQ relational_expression
                       | equality_expression NE relational_expression

relational_expression : additive_expression
                         | relational_expression '<' additive_expression
                         | relational_expression '>' additive_expression

additive_expression : multiplicative_expression
                        | additive_expression '+' multiplicative_expression
                        | additive_expression '-' multiplicative_expression

multiplicative_expression : power_expression
                            | multiplicative_expression '*' power_expression
                            | multiplicative_expression '/' power_expression
                            | multiplicative_expression '%' power_expression

power_expression : unary_expression
                    | power_expression '^' unary_expression

unary_expression : suffix_expression

unary_expression : INCREMENT unary_expression
unary_expression : DECREMENT unary_expression
unary_expression : NOT unary_expression
unary_expression : '+' unary_expression
                    | '-' unary_expression

suffix_expression : suffix_expression '[' conditional_expression ']'
suffix_expression : suffix_expression '.' ID
suffix_expression : suffix_expression INCREMENT
suffix_expression : suffix_expression DECREMENT
suffix_expression : primary_expression
suffix_expression : function_call_expression

function_call_expression : FUNC_ME '(' assignment_expression ',' assignment_expression ','
assignment_expression ')'

                    | FUNC_ME '(' assignment_expression ',' assignment_expression ')'

function_call_expression : FUNC_CAP '(' assignment_expression ')'

function_call_expression : FUNC_COL '(' assignment_expression ',' assignment_expression ')'

function_call_expression : FUNC_PR '(' assignment_expression ')'
```

```
function_call_expression : FUNC_LE '(' assignment_expression ')'

primary_expression : '(' expression ')'
primary_expression : ID
primary_expression : RES
primary_expression : BEGIN
primary_expression : END
primary_expression : STRING
primary_expression : FLOAT
primary_expression : INT
primary_expression : '[' expression ']'
primary_expression :  '{' field_value_expressions '}'

field_value_expressions : field_value_expression
                        | field_value_expressions ',' field_value_expression

field_value_expression : ID ':' conditional_expression
```

2.2.2. TDS - Atributos

El segundo aspecto relevante del parser es el aspecto semántico. Así como la gramática básica se encarga de aceptar o rechazar cadenas en base a su sintaxis, al extenderla a una TDS ganamos la posibilidad de rechazar cadenas en base a condiciones semánticas.
El enunciado de este trabajo pedía que el parser realizara type-checking es decir chequeo de las reglas de tipado de C++. Para ello se incorporaron a los símbolos una serie de atributos con ese fin.

La TDS cuenta con varios atributos. Implementativamente, cada símbolo es representado por un $ASTNode$, un objeto que representa un nodo de un árbol de derivación potencial. Este nodo es el que tiene los atributos y métodos asignados a cada símbolo no terminal.

Cada una de las derivaciones es una función python que crea los nodos correspondientes para cada símbolo y realiza los chequeos de tipado. Al encontrar un error de tipos, el parser corta la ejecución y lanza un TypeError, el cual se imprime en pantalla, aclarando el error específico y la línea donde ocurrió.

Todos los atributos aquí mencionados son sintetizados.

Atributos (más relevantes) de la TDS:

- **type**: representa el tipo de cada símbolo. Se representa como un diccionario en python con dos campos, "type_name" que contiene el nombre del tipo, por ejemplo INT, $FLOAT$, $REGISTER$, etc. y "fields" que es a su vez un diccionario. Este segundo diccionario solo se utiliza para los tipos compuestos, registro y vector.
  En los registros, "fields" contiene una entrada para cada campo del registro. Dicha entrada contiene como clave el nombre del campo del registro y el significado es a su vez un diccionario *type*, que representa el tipo del campo.
  En los vectores, "fields" contiene únicamente un diccionario *type* que representa el tipo de cada elemento del vector. En SLS los elementos de un vector son todos del mismo tipo.
  Aclaración: no todos los símbolos tienen tipo, solo los que representan variables, valores o expresiones tipadas de C++.
- **text**: en este atributo se va sintetizando el texto bien formateado que procesa cada nodo. De esta forma, cada símbolo va formando su texto a través de los textos de sus nodos hijos. Desde los nodos más básicos que sólo tienen un valor o una variable, hasta los nodos más complejos que pueden contener un bloque entero de código. Cada producción tiene sus especificaciones sobre qué hacer con el texto de sus símbolos y que sintetizar en el atributo *text* del nodo del lado izquierdo. Ver más detalles en la subsección siguiente, "Impresión".
- **line**: los símbolos de tipo statement y comment deben saber en que línea del código original se encontraban. De esta forma pueden saber si un comentario era inline.
- **descendants**: guardo para cada nodo sus nodos hijos del AST. Únicamente utilizado para graficar ASTs.

2.2.3. TDS - Type-Checking

Para implementar el type-checking, además de utilizar el atributo sintetizado *type*, fue necesario tomar otras medidas. El programa cuenta con un diccionario global de variables llamado $varTypeDict$ en donde se registran todas las variables junto con su *type*. Este diccionario se utiliza para poder recordar que tipo tenía cada variable declarada.

De esta forma, para valores y variables nuevas utilizo el tipo del atributo *type* mientras que para variables ya declaradas anteriormente, se revisa $varTypeDict$ y se les asigna el tipo que tenían allí registrado.

Una producción clave en el funcionamiento del sistema de type-checking es la de $assignment_expression$, ya que aquí es donde se realiza el registro de la variable en $varTypeDict$. Así, en mi sistema una variable se declara cuando se le asigna un valor. Si encuentro una variable (una expresión ID) que no haya tenido una asignación previa, la marcaré con tipo "UNKNOWN" hasta que se le asigne algún tipo de valor.

2.2.4. TDS - Consideraciones Semánticas

El type-checking implementado cumple todas las reglas pedidas por el enunciado. Como además el enunciado dejaba algunas cuestiones a interpretación, voy a enunciar aquí algunas consideraciones y decisiones de diseno que tomé para las reglas semánticas.

Consideraciones Semánticas:

- No se lleva registro de los valores de las variables. Solo de su tipo. No lo consideré necesario para el desarrollo del trabajo y agregaba muchísima complejidad.
- Como mencioné anteriormente, las variables nombradas se registran globalmente junto con su tipo. Si a una variable no se le asigna tipo, será de tipo desconocido y no podrá ser utilizada para operar. Esto tiene algunas consecuencias. Observar este extracto del ejemplo 1 del enunciado:

  ```
  a=100;
  for (i=1; i<a;i++) b = b+10*i;
  ```

  En mi sistema, este código será erróneo, ya que para la primera iteración del for, b tiene tipo (y valor) desconocido y por ende no tiene sentido que se la utilice para una operación.
  Esta se trata de una decision de disenio mía. Considero que no debería ser posible utilizar variables sin valor ni tampoco sería bueno asignarles a las variables un valor default.
- Mi parser utiliza *typeJuggling*. Es decir, bajo ciertas circunstancias, una variable puede cambiar su tipo por otro. Por ejemplo, al usar el operador "+" con un operando STRING y otro INT, el resultado será un STRING. No se considera un error de tipos, se asume que se concatenó el string y el valor string del INT. Otro ejemplo posible, al utilizar operadores matemáticos entre INTs y FLOATs, el resultado será FLOAT. Ver un ejemplo en carpeta **src/test**, archivo *typeJuggling*.

  Esta decisión se implementó debido que consideré que se debía poder operar entre tipos numéricos sin importar si eran puntos flotantes o enteros y además a la presencia de concatenaciones entre valores INT y STRING presentes en los tests de la cátedra, como por ejemplo *all.i*.
- Las asignaciones con operacion, como por ejemplo "+=", "-=", "*=" etc. solo se permiten entre tipos numéricos. A excepcion de "+=" que también permite strings. Ver un ejemplo en carpeta **src/test**, archivo *malaAsignacionOperador*.
- Los errores en la cantidad de parámetros para las funciones son filtrados por la gramática. Pero los errores en el tipo de los parámetros son detectados por el type-checking.
- Al hacer un llamado a un elemento dentro de un vector con un índice, como por ejemplo:

  ```
  a = v[i];
  ```

  No se chequea que el índice esté en rango. No es posible hacerlo ya que no se lleva registro de los valores de las variables como tampoco se registra la longitud de los vectores. Por ende, al hacer un llamado de este tipo, únicamente se sabe que ahora a es del mismo tipo que todos los elementos del vector.
- Al asignar valores dentro de un vector, no se permite modificar su tipo. Si es un vector, por ejemplo, de BOOLs, no se puede insertar un INT. Ver un ejemplo en carpeta **src/test**, archivo $insertandoMalEnVector$.
- Similarmente, si es un vector de registros, no se puede modificar el tipo de sus campos ni agregar nuevos campos a ninguno de los registros contenidos. Esto rompería con el invariante que los vectores tienen todos elementos del mismo tipo. Ver un ejemplo en carpeta **src/test**, archivo *cambioCamposDentroVector*.
- El operador ternario $a?b:c$ requiere que a sea BOOL, y que b y c sean del mismo tipo. Se permite que b y c sean ambos numéricos, uno INT y el otro FLOAT.
- Se considera que si la función multiplicacionEscalar recibe tercer parámetro, éste es $TRUE$. Al no llevar registro de los valores, no puedo corroborar si el booleano pasado se reduce a $TRUE$ o no, asi que asumo que siempre lo es. Por ende al ser siempre verdadero, la función con tres parámetros devuelve siempre un $VECTOR(INT)$.

2.2.5. Impresión

Además del chequeo sintáctico y de tipado, el parser, en caso de no hallar errores, imprime el código procesado en un formato especial. Este formato es el que pide el enunciado: no hay líneas en blanco, se tabulan todas las líneas dentro de un bloque, se respetan los comentarios inline y cada sentencia va en una línea aparte.
Sin embargo, además de estas consignas, mi parser cuenta con algunas salvedades y decisiones agregadas. A saber:

Consideraciones de Impresión:

- Las palabras reservadas se permiten en upper y lowercase y se devuelven igual que como se escribieron.
- En caso de abrirse un bloque de código con llaves, los comentarios pegados a las llaves no se consideran inline. Para facilidad de la implementación, al abrir un bloque, todas las sentencias interiores se bajan un renglón y se tabulan. Es decir, las llaves ("{", "}") no permiten código ni comentarios adicionales a su derecha.
- En la misma sintonía, el $IF - THEN - ELSE$ imprime el $ELSE$ abajo de la llave de cierre del bloque del if. Tomé esta decisión luego de ver el pretty print del ejemplo 4 del enunciado, donde bajan el else a la siguiente línea. Luego, al probar los ejemplos de la cátedra vi que en muchos se deja el else en la misma línea que el cierre del bloque del if. Decidí dejarlo de la forma original.
- Similarmente, el $DO - WHILE$ imprime el *while* abajo del cierre del bloque del DO.

2.3. Limitaciones Conocidas

El parser tiene los siguientes problemas y limitaciones conocidas:

- Existe un conflicto Shift-Reduce para las producciones del IF. Luego de parser una expresión que comienza con un if, su condición y su bloque de código, el parser encuentra este conflicto:

```
    statement_if -> IF ( expression ) comments_statement .
    statement_if -> IF ( expression ) comments_statement . ELSE comments_statement

  ! shift/reduce conflict for ELSE resolved as shift
```

  El conflicto se resuelve siempre tomando el camino del Shift, ya que si hay un $ELSE$, seguramente es el que le corresponde al IF anterior. Por ende la acción por default del YACC, shiftear, es correcta.

- En algunas cadenas, si existe un error en la última linea o si el texto introducido no matchea con absolutamente ninguna producción, la función que retorna error de sintaxis no logra determinar la línea del error. En estos casos, que aún no entiendo por qué suceden, se devuelve un mensaje de error genérico, sin número de línea, que sugiere revisar la última línea.
- El parser no devuelve el código formateado **exactamente igual** a los ejemplos de prueba entregados por la cátedra. Hay leves variaciones, como la posición de un *else* o el cierre de la llave de un bloque. No considero estos errores sino diferencias en el formato de impresión. Todas estas sutilezas fueron aclaradas en la subsection anterior, "Impresión".

3. Resultados

En esta sección comentaré los resultados obtenidos, los casos de testing y las conclusiones del trabajo.

3.1. Testing

Para probar el parser utilicé los múltiples casos de prueba provistos por la cátedra. Además, desarrollé algunos por mi cuenta para probar casos específicos. Incluyo los casos de test más interesantes en la carpeta **src/test**. Básicamente prueban casos bordes, elementos con tipos compuestos dentro de tipos compuestos, códigos difíciles de formatear o ambiguos.

Los archivos cuentan con un nombre descriptivo y dentro mismo del código SLS hay comentarios que resumen la intención de cada test, por lo que no voy a hacer una explicación detallada de cada uno. El nombre del archivo contiene también el resultado esperado (PASA/FALLA) y el problema que presentan.

Tanto para mis tests como para los de la cátedra, el parser no mostró problemas graves. Existen como ya comenté algunas divergencias menores en los resultados esperados, la mayoría por haber tomado yo criterios distintos para el type-checking o el pretty-printing. Todos estas diferencias están mencionadas en la sección Desarrollo.

3.2. Conclusión

Este trabajo fue de utilidad para poder ver en la práctica real temas estudiados a lo largo del curso. Ayuda tener un caso real para poder ver, aunque sea de forma básica, el funcionamiento interno de un compilador.

Valoro la oportunidad de haber podido utilizar conocimientos adquiridos que de otra forma no hubiese podido bajar a la vida real. El trabajo me resultó muy extenso ya que lo desarrollé casi íntegramente solo, aunque fue muy interesante y por momentos hasta disfrutable.

Considero que los conocimientos adquiridos en este curso aportan a mi formación profesional.

4. Apéndice - Código

Incluyo a continuación el código del lexer junto a sus reglas y del parser junto a las suyas.

4.1. Bash Script SLSParser

```
#Codigos especiales que uso en parser.py
#Para determinar destino de input/output

OUTPUT="_DIRECT_OUTPUT"
INPUT="_DIRECT_INPUT"


# Llamada incorrecta, tiene que
# haber al menos un parametro
if [ $# -lt 1 ]
then
    echo
    echo "Llamada Incorrecta! Debe haber al menos un parametro (FUENTE)"
    echo "Uso:"
    echo "./SLSParser [-o SALIDA] -c ENTRADA | FUENTE "
    echo
    echo "Nota: En caso de poner el codigo directamente, hacerlo entre comillas"
    exit 1
fi

# Si hay solo 1 param, es la fuente directa
if [ $# -eq 1 ]
then
    INPUT+="$1"
    python parser.py "$INPUT" "$OUTPUT"
    exit 0
fi

# Si hay solo 2 param, debe ser "-c ENTRADA"
if [ $# -eq 2 ]
then
    # Hay archivo input, lo seteo y llamo al parser
    if [ "$1" = "-c" ]
    then
        INPUT="$2"
        python parser.py "$INPUT" "$OUTPUT"
        exit 0
    fi
fi

# Si hay solo 3 param, debe ser "-o SALIDA FUENTE"
if [ $# -eq 3 ]
then
    # Hay archivo output, lo seteo
    if [ "$1" = "-o" ]
    then
        OUTPUT="$2"
        INPUT+="$3"
        python parser.py "$INPUT" "$OUTPUT"
        exit 0
    fi
fi
```

```
# Si hay 4 param, deben ser "-o SALIDA" y "-c ENTRADA"
if [ $# -eq 4 ]
then
    # Hay archivo output, lo seteo
    if [ "$1" = "-o" ] && [ "$3" = "-c" ]
    then
        OUTPUT="$2"
        INPUT="$4"
        python parser.py "$INPUT" "$OUTPUT"
        exit 0
    fi
fi


#Si no se cumplieron ninguna de las condiciones, llamada incorrecta
echo
echo "Llamada Incorrecta!"
echo "Uso:"
echo "./SLSParser [-o SALIDA] -c ENTRADA | FUENTE "
echo
echo "Nota: En caso de poner el codigo directamente, hacerlo entre comillas"
exit 1
```

4.2. lexer.py

```
import lexer_rules

from sys import argv

from ply.lex import lex


def dump_tokens(lexer, output_file):
    token = lexer.token()

    while token is not None:
        output_file.write("type:" + token.type)
        output_file.write(" value:" + str(token.value))
        output_file.write(" line:" + str(token.lineno))
        output_file.write(" position:" + str(token.lexpos))
        output_file.write("\n")

        token = lexer.token()


if __name__ == "__main__":
    if len(argv) != 3:
        print "Parametros invalidos."
        print "Uso:"
        print "  lexer.py archivo_entrada archivo_salida"
        exit()

    input_file = open(argv[1], "r")
    text = input_file.read()
    input_file.close()

    lexer = lex(module=lexer_rules)

    lexer.input(text)

    output_file = open(argv[2], "w")
    dump_tokens(lexer, output_file)
    output_file.close()
```

4.3. lexer_rules.py

```
# Palabras reservadas del lenguaje
reserved = {
    'for' : 'FOR',
    'while' : 'WHILE',
    'do' : 'DO',
    'if' : 'IF',
    'else' : 'ELSE',
    'begin' : 'BEGIN',
    'end' : 'END',
    'return' : 'RETURN',
    'res' : 'RES',
    'and' :'AND',
    'or' : 'OR',
    'not' : 'NOT',
}

# Tokens
tokens = [
    'STRING',
    'BOOL',
    'FLOAT',
    'INT',
    'INCREMENT',
    'DECREMENT',
    'EQ',
    'NE',
    'ADDASSIGN',
    'SUBASSIGN',
    'MULASSIGN',
    'DIVASSIGN',
    'FUNC_ME',
    'FUNC_CAP',
    'FUNC_COL',
    'FUNC_PR',
    'FUNC_LE',
    'ID',
    'COMMENT',
] + list(reserved.values())

# Literales
literals = [
    '+',
    '-',
    '*',
    '/',
    '^',
    '%',
    '=',
    '>',
    '<',
    '!',
    ';',
    '(',
    ')',
    '{',
    '}',
    '[',
```

```
    ']',
    '?',
    ':',
    '.',
    ',',
    '\'',
    '"'
]

# Tipos de datos
## Cadena, booleano, numerico (con o sin decimales), vectores, y registros
def t_STRING(token):
    r'\"(\\.|[^\\"])*\"'
    token.value = str(token.value)
    return token

def t_BOOL(token):
    r"([Tt][Rr][Uu][Ee]|[Ff][Aa][Ll][Ss][Ee])"
    return token

def t_FLOAT(token):
    r"[0-9]+\.[0-9]*|[-+]?[0-9]*\.[0-9]+"
    token.value = float(token.value)
    return token

def t_INT(token):
    r"[0-9]+"
    token.value = int(token.value)
    return token

# Operadores matematicos unarios
def t_INCREMENT(token):
    r"\+\+"
    return token

def t_DECREMENT(token):
    r"\-{2}"
    return token


# Operadores relacionales
t_EQ = r'=='
t_NE = r'!='


# Operadores de asignacion compuestos
t_ADDASSIGN = r'\+='
t_SUBASSIGN = r'-='
t_MULASSIGN = r'\*='
t_DIVASSIGN = r'/='

# Funciones
functions = {
    'multiplicacionescalar': 'FUNC_ME',
    'capitalizar': 'FUNC_CAP',
    'colineales': 'FUNC_COL',
    'print': 'FUNC_PR',
    'length': 'FUNC_LE',
}
```

```
def t_ID(token):
    r"[a-zA-Z][_a-zA-Z0-9]*"
    if(token.value.lower() in reserved):
        token.type = reserved.get(token.value.lower())

    elif(token.value.lower() in functions):
        token.type = functions.get(token.value.lower())

    else:
        token.value = str(token.value)

    return token

def t_COMMENT(token):
    r"\#.*"
    token.value = str(token.value)
    return token

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

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)
```

4.4. parser.py

```
import lexer_rules
import parser_rules
import pprint

from sys import argv, exit

from ply.lex import lex
from ply.yacc import yacc


def dump_ast(ast, output_file):
    output_file.write("digraph {\n")

    edges = []
    queue = [ast]
    numbers = {ast: 1}
    current_number = 2
    while len(queue) > 0:
        node = queue.pop(0)
        name = node.name().replace('"', "'")
        number = numbers[node]
        output_file.write('node[width=1.5, height=1.5,
            shape="circle", label="%s"] n%d;\n' % (name, number))

        for child in node.children():
            numbers[child] = current_number
            edge = 'n%d -> n%d;\n' % (number, current_number)
            edges.append(edge)
            queue.append(child)
            current_number += 1

    output_file.write("".join(edges))

    output_file.write("}")


# http://stackoverflow.com/questions/2556108/how-to-replace-
the-last-occurence-of-an-expression-in-a-string

# Reemplaza la ultima aparicion de "old" por "new" en "s",
"occurrence" cantidad de veces

def rreplace(s, old, new, occurrence):
    li = s.rsplit(old, occurrence)
    return new.join(li)


if __name__ == "__main__":
    if len(argv) != 3:
        print "Parametros invalidos."
        print "Uso:"
        print "  parser.py archivo_entrada archivo_salida"
        exit()


    if(argv[1].startswith('_DIRECT_INPUT')):
        # Le saco mi marca
```

```
    text = argv[1].replace('_DIRECT_INPUT', '', 1)

else: #Archivo Input
    input_file = open(argv[1], "r")
    text = input_file.read()
    input_file.close()

lexer = lex(module=lexer_rules)
parser = yacc(module=parser_rules)

ast = parser.parse(text, lexer)

if(ast is None):
    # Hubo error, no imprimir texto
    exit()

final_text = ast.text

# Hago un trim del texto final, por si quedaron lineas en blanco al comienzo
#  o al final
if(final_text.startswith('\n')):
    final_text = final_text.replace('\n', '', 1)
if(final_text.endswith('\n')):
    final_text = rreplace(final_text, '\n', '', 1)

if(argv[2] == '_DIRECT_OUTPUT'):
    print "----------------------------------------"
    print final_text
    print "----------------------------------------"
else: #Archivo Input
    output_file = open(argv[2], "w")
    output_file.write(final_text)
    output_file.close()
```

4.5. parser_rules.py

```
from lexer_rules import tokens
import pprint
import re


# tipos basicos = {'type_name': INT, 'fields': {}}
#
# reg = {'type_name': REG, 'fields': {'edad': {'type_name': INT, 'fields': {}},
#                                 'nombre': {'type_name': STR, 'fields': {}},
#                                 'agenda': {'type_name': REG, 'fields':
#                                           {'edad': {'type_name': INT, 'fields': {}},
#                                           'nombre': {'type_name': STR, 'fields': {}},
# }}
# vec = {'type_name': VECTOR_REGISTER, fields':
#            {0: {'type_name': REG, 'fields': {'edad': {'type_name': INT, 'fields': {}},
#                                              'nombre': {'type_name': STR, 'fields': {}} }
#
# vec = {'type_name': VECTOR_INT, 'fields': {} }
# }


class ASTNode: #Representa Nodo con atributos
    def __init__(self, label, children=[], text='',
             type={'type_name': 'UNKNOWN', 'fields': {}},
             line=-1,
             isRegisterCall=False,
             isVectorCall=False):

        self.label = str(label)
        self.text = text
        self.descendants = children
        self.type = type
        self.line = line

        self.isRegisterCall = isRegisterCall
        self.isVectorCall = isVectorCall

        if(not text):
            self.text = self.label

    def name(self):
        return self.label

    def children(self):
        return self.descendants

###################### FUNCIONES AUXILIARES ###################################
varTypeDict = {}

def checkTypes(nodes, types,
        message = 'Error de tipos'):

    for node in nodes:
        if (node.type['type_name'] not in types):
            raise TypeError(message)
            return
```

```
def registerVar(varName, varType):
    # registro variable en diccionario global de variable->tipo
    varTypeDict[varName] = varType

def updateField(varName, fieldName, fieldType):
    # actualizo el valor de un campo de un registro
    varTypeDict[varName]['fields'].update({fieldName: fieldType})

def isVector(varType):
    return varType['type_name'].startswith('VECTOR', 0, 6)

def isNumeric(varType):
    return varType['type_name'] in ['INT', 'FLOAT']


# http://stackoverflow.com/questions/2556108/how-to-replace-the-last-occurence-of-an-expression-in-
# Reemplaza la ultima aparicion de "old" por "new" en "s", "occurrence" cantidad de veces
def rreplace(s, old, new, occurrence):
    li = s.rsplit(old, occurrence)
    return new.join(li)

###################### FIN FUNCIONES AUXILIARES ###################################

def p_statements(p):
    '''statements :  COMMENT
                  |  COMMENT statements
                  |  statement
                  |  statement statements
    '''

    if(len(p) == 2):
        if type(p[1]) is str:
            # statements :  COMMENT

            comment = ASTNode(p[1], text=p[1])

            p[0] = ASTNode('statements1', [comment],
                text= comment.text,
                line=p.lineno(1))

        else:
            # un statement comun
            p[0] = ASTNode('statements1', [p[1]],
                text=p[1].text,
                line=p.lineno(1))

    else:
        if type(p[1]) is str:
            # Varios comments al inicio del archivo o de un bloque
            comment = ASTNode(p[1], text=p[1])

            if(p[2].text.startswith('\n')):
                #Si ya trae un \n (es un statement comun, no un comentario)
                text = comment.text + p[2].text
            else:
                text = comment.text + '\n' + p[2].text

            p[0] = ASTNode('statements1', [comment, p[2]],
                text= text,
```

```
                line=p.lineno(1))

        else:

            if(p[2].text.startswith('#') and p[2].line == p[1].line):
                # comment inline: le saco el \n que todos los comments se ponen
                text = p[1].text + p[2].text
            elif(p[2].text.startswith('\n')):
                # Sea comment o sentencia comun, si ya viene con \n, no agrego newline
                text = p[1].text + p[2].text
            else:
                text = p[1].text + '\n' + p[2].text

            p[0] = ASTNode('statements2', [p[1],p[2]],
                text=text,
                line=p.lineno(1))

def p_comments_statement(p):
    '''comments_statement : statement
                          | comments statement
    '''
    if(len(p) == 2):
        statement = p[1]
        p[0] = ASTNode('comments_statement1', [statement],
            text=statement.text,
            line=p[1].line)
    else:
        comments = p[1]
        statement = p[2]

        p[0] = ASTNode('comments_statement2', [comments, statement],
            text=comments.text + statement.text,
            line=p[2].line)

def p_comments(p):
    '''comments : COMMENT
                | comments COMMENT
    '''
    if(len(p) == 2):

        comment = ASTNode(p[1], text=p[1])
        p[0] = ASTNode('comments_single_comment', [comment],
            text='\n' + comment.text,
            line=p.lineno(1))
    else:
        recursion = p[1]
        comment = ASTNode(p[2], text=p[2])

        p[0] = ASTNode('comments', [recursion,comment],
            text=recursion.text+'\n'+comment.text,
            line=p.lineno(1))

def p_statement(p):
    '''statement : statement_iter
                 | statement_expression
                 | statement_block
                 | statement_if
                 | statement_return
     '''
```

```
    p[0] = ASTNode('statement', [p[1]],
        text=p[1].text,
        line=p[1].line)

def p_statement_iter_while(p):
    '''statement_iter : WHILE '(' expression ')' comments_statement'''

    while_terminal = ASTNode(p[1], text=p[1])
    left_parethesis = ASTNode(p[2], text=p[2])
    condition = p[3]
    right_parethesis = ASTNode(p[4], text=p[4])
    body = p[5]

    checkTypes([condition], ['BOOL'], 'Linea ' + str(p.lexer.lineno) +
        ': la condicion del WHILE debe ser BOOL') #condicion debe ser bool

    tabulated_body = body.text.replace("\n", "\n\t")

    if(body.text.startswith('{')):
        tabulated_body = rreplace(tabulated_body, '\t}', '}', 1)
    # Si lo que sigue es un bloque con {}, le quito 1 tab al ultimo }

    text = ('\n' + while_terminal.text + left_parethesis.text +
        condition.text + right_parethesis.text +
        tabulated_body
    )

    children = [while_terminal, left_parethesis, condition,
        right_parethesis, body]

    p[0] = ASTNode('statement_while', children,
        text=text,
        line=p.lineno(5))

def p_statement_iter_do_while(p):
    '''statement_iter : DO comments_statement WHILE '(' expression ')' ';' '''

    do_terminal = ASTNode(p[1], text=p[1])
    body = p[2]
    while_terminal = ASTNode(p[3], text=p[3])
    left_parethesis = ASTNode(p[4], text=p[4])
    condition = p[5]
    right_parethesis = ASTNode(p[6], text=p[6])
    semicolon = ASTNode(p[7], text=p[7])

    checkTypes([condition], ['BOOL'], 'Linea ' + str(p.lexer.lineno) +
        ': la condicion del WHILE debe ser BOOL') #condicion debe ser bool

    tabulated_body = body.text.replace("\n", "\n\t")

    if(body.text.startswith('{')):
        tabulated_body = rreplace(tabulated_body, '\t}', '}', 1)
    # Si lo que sigue es un bloque con {}, le quito 1 tab al ultimo }

    text = ('\n' + do_terminal.text + tabulated_body +
            '\n' + while_terminal.text +
            left_parethesis.text + condition.text +
            right_parethesis.text + semicolon.text)
```

```
    children = [do_terminal, body, while_terminal, left_parethesis,
        condition, right_parethesis, semicolon]

    p[0] = ASTNode('statement_do_while', children,
        text=text,
        line=p.lineno(7))


def p_statement_iter_for_complete(p):
    '''statement_iter : FOR '(' statement_expression statement_expression expression ')'
        comments_statement '''

    for_terminal = ASTNode(p[1], text=p[1])
    left_parethesis = ASTNode(p[2], text=p[2])

    statement_expression_1 = p[3]
    statement_expression_2 = p[4]
    final_expression = p[5]

    right_parethesis = ASTNode(p[6], text=p[6])
    body = p[7]

    checkTypes([statement_expression_2], ['BOOL'], 'Linea ' +
            str(p.lexer.lineno) +
        ': la condicion del FOR debe ser BOOL') #condicion debe ser bool

    children = [for_terminal, left_parethesis,
        statement_expression_1, statement_expression_2,
        final_expression, right_parethesis, body]

    tabulated_body = body.text.replace("\n", "\n\t")

    if(body.text.startswith('{')):
        tabulated_body = rreplace(tabulated_body, '\t}', '}', 1)
    # Si lo que sigue es un bloque con {}, le quito 1 tab al ultimo }

    p[0] = ASTNode('statement_for_complete', children,
        line=p.lineno(7),
        text='\n' + for_terminal.text + left_parethesis.text +
            statement_expression_1.text.replace("\n", "") + ' '+
            statement_expression_2.text.replace("\n", "") + ' '+
            final_expression.text.replace("\n", "") +
            right_parethesis.text +
            tabulated_body)

def p_statement_iter_for_short(p):
    '''statement_iter : FOR '(' statement_expression statement_expression ')' comments_statement ''

    for_terminal = ASTNode(p[1], text=p[1])
    left_parethesis = ASTNode(p[2], text=p[2])

    statement_expression_1 = p[3]
    statement_expression_2 = p[4]

    right_parethesis = ASTNode(p[5], text=p[5])
    body = p[6]

    checkTypes([statement_expression_2], ['BOOL'], 'Linea ' +
            str(p.lexer.lineno) +
```

```
        ': la condicion del FOR debe ser BOOL') #condicion debe ser bool

    children = [for_terminal, left_parethesis, statement_expression_1,
        statement_expression_2,
        right_parethesis, body]

    tabulated_body = body.text.replace("\n", "\n\t")

    if(body.text.startswith('{')):
        tabulated_body = rreplace(tabulated_body, '\t}', '}', 1)
    # Si lo que sigue es un bloque con {}, le quito 1 tab al ultimo }

    p[0] = ASTNode('statement_for_complete', children,
        line=p.lineno(6),
        text='\n' + for_terminal.text + left_parethesis.text +
            statement_expression_1.text.replace("\n", "") + ' '+
            statement_expression_2.text.replace("\n", "") +
            right_parethesis.text +
            tabulated_body)

def p_statement_if(p):
    '''statement_if :  IF '(' expression ')' comments_statement
                    |  IF '(' expression ')' comments_statement ELSE comments_statement '''


    if_terminal     = ASTNode(p[1], text=p[1])
    left_parethesis = ASTNode(p[2], text=p[2])
    condition = p[3]
    right_parethesis = ASTNode(p[4], text=p[4])
    if_body = p[5]

    checkTypes([condition], ['BOOL'], 'Linea ' + str(p.lexer.lineno) +
        ': la condicion del IF debe ser BOOL') #condicion debe ser bool

    tabulated_body = if_body.text.replace("\n", "\n\t")

    if(if_body.text.startswith('{')):
        tabulated_body = rreplace(tabulated_body, '\t}', '}', 1)
    # Si lo que sigue es un bloque con {}, le quito 1 tab al ultimo }

    if(len(p) == 6):
        children = [if_terminal, left_parethesis, condition, right_parethesis, if_body]

        p[0] = ASTNode('statement_if_no_else', children,
            line=p.lineno(5),
            text='\n' + if_terminal.text + left_parethesis.text + condition.text +
                right_parethesis.text +
                tabulated_body
                )
    else:
        else_terminal = ASTNode(p[6], text=p[6])
        else_body = p[7]

        children = [if_terminal, left_parethesis,
            condition, right_parethesis,
            if_body, else_terminal, else_body]

        tabulated_else_body = else_body.text.replace("\n", "\n\t")
```

```
        if(else_body.text.startswith('{')):
            tabulated_else_body = rreplace(tabulated_else_body, '\t}', '}', 1)
            # Si lo que sigue es un bloque con {}, le quito 1 tab al ultimo }

        p[0] = ASTNode('statement_if', children,
            line=p.lineno(7),
            text='\n' + if_terminal.text + left_parethesis.text +
                condition.text +
                right_parethesis.text +
                tabulated_body +
                '\n' +
                else_terminal.text +
                tabulated_else_body
            )

def p_statement_block(p):
    '''statement_block : '{' statements '}'
    '''

    left_bracket = ASTNode(p[1], text=p[1])

    right_bracket = ASTNode(p[3], text=p[3])

    statements = p[2]

    if(statements.text.startswith('#')):
    # Si viene un comment, bajarlo y tabularlo
        formatted_statements_text = '\n'+ statements.text
    else:
        formatted_statements_text = statements.text

    p[0] = ASTNode('statement_block',
        [left_bracket, statements, right_bracket],
        line=p.lineno(3),
        text=left_bracket.text +
            formatted_statements_text +
            '\n' +
            right_bracket.text)

def p_statement_return(p):
    '''statement_return : RETURN ';'
                        | RETURN expression ';'
    '''

    return_terminal = ASTNode(p[1], text=p[1])

    if(len(p) == 3):
        semicolon = ASTNode(p[2], text=p[2])

        p[0] = ASTNode('statement_return_empty',
            [return_terminal, semicolon],
            text='\n' + return_terminal.text + semicolon.text,
            line=p.lineno(2))
    else:
        expression = p[2]
        semicolon = ASTNode(p[3], text=p[3])

        p[0] = ASTNode('statement_return',
            [return_terminal, expression, semicolon],
```

```
            text='\n' + return_terminal.text + ' ' + expression.text
                + semicolon.text,
            line=p.lineno(3))

def p_statement_expression(p):
    '''statement_expression : ';'
                            | expression ';'
    '''
    if(len(p) == 2):
        semicolon = ASTNode(p[1], text=p[1])

        p[0] = ASTNode('statement_expression1', [semicolon],
            line=p.lineno(1),
            text=semicolon.text)
    else:

        semicolon = ASTNode(p[2], text=p[2])

        p[0] = ASTNode('statement_expression2', [p[1], semicolon],
            text='\n' + p[1].text+semicolon.text,
            line=p.lineno(2),
            type=p[1].type)

def p_expression(p):
    '''expression : assignment_expression
                | expression ',' assignment_expression
    '''
    if(len(p) == 2):
        p[0] = ASTNode('expression1', [p[1]],
            text=p[1].text,
            type=p[1].type)
    else:
        # este es el caso que sea una expresion tipo lista de expresiones separadas por comma
        # para la declaracion de un vector
        # por ende el tipo del primer elemento debe ser el de los siguientes (o numericos)

        if(p[1].type != p[3].type and
            not(isNumeric(p[1].type) and isNumeric(p[3].type)) ) :
            raise TypeError('Linea ' + str(p.lexer.lineno) +
                ': Cuidado, elementos del vector deben ser del mismo tipo!')

        comma = ASTNode(p[2], text=p[2])

        p[0] = ASTNode('expression2', [p[1], comma, p[3]],
            text=p[1].text+comma.text+' '+p[3].text,
            type=p[3].type)

def p_assignment_expression(p):
    '''assignment_expression :  conditional_expression
                             |  unary_expression assign_operator conditional_expression
    '''
    if(len(p) == 2):
        p[0] = ASTNode('assignment_expression1', [p[1]],
            text=p[1].text,
            type=p[1].type)
    else:

        assign_operator = p[2]
```

```
#Debo chequear que si la asignacion es con operacion, los tipos sean correctos
if(assign_operator.text == '+='):

    if(p[1].type['type_name'] == 'INT'
        and p[3].type['type_name'] == 'INT'): # INT + INT = INT
        resulting_type = {'type_name': 'INT', 'fields': {} }

    elif(p[1].type['type_name'] == 'INT'
        and p[3].type['type_name'] == 'FLOAT'): # INT + FLOAT = FLOAT
        resulting_type = {'type_name': 'FLOAT', 'fields': {} }

    elif(p[1].type['type_name'] == 'FLOAT'
        and p[3].type['type_name'] == 'INT'): # FLOAT + INT = FLOAT
        resulting_type = {'type_name': 'FLOAT', 'fields': {} }

    elif(p[1].type['type_name'] == 'FLOAT'
        and p[3].type['type_name'] == 'FLOAT'): # FLOAT + FLOAT = FLOAT
        resulting_type = {'type_name': 'FLOAT', 'fields': {} }

    elif(p[1].type['type_name'] == 'STRING'
        and p[3].type['type_name'] == 'STRING'): # STRING + STRING = STRING
        resulting_type = {'type_name': 'STRING', 'fields': {} }

    elif(p[1].type['type_name'] == 'STRING'
        and isNumeric(p[3].type)): # STRING + Numeric = STRING
        resulting_type = {'type_name': 'STRING', 'fields': {} }

    elif(isNumeric(p[1].type)
        and p[3].type['type_name'] == 'STRING'): # Numeric + STRING = STRING
        resulting_type = {'type_name': 'STRING', 'fields': {} }

    elif(p[1].type['type_name'] == 'UNKNOWN'): # * + UNKNOWN = UNKNOWN
        raise TypeError('Linea ' + str(p.lexer.lineno) + ': "' +
            p[1].text + '" variable no declarada, no tiene valor para operar')

    elif(p[3].type['type_name'] == 'UNKNOWN'): # UNKNOWN + * = UNKNOWN
        raise TypeError('Linea ' + str(p.lexer.lineno) + ': "' +
            p[3].text + '" variable no declarada, no tiene valor para operar')
    else:
        raise TypeError('Linea ' + str(p.lexer.lineno) +
            ': tipos no validos para operacion +=')

elif(assign_operator.text == '-='
    or assign_operator.text == '*='
    or assign_operator.text == '/='):

    if(p[1].type['type_name'] == 'INT' and
        p[3].type['type_name'] == 'INT'): # INT - INT = INT
        resulting_type = {'type_name': 'INT', 'fields': {} }

    elif(p[1].type['type_name'] == 'INT' and
        p[3].type['type_name'] == 'FLOAT'): # INT - FLOAT = FLOAT
        resulting_type = {'type_name': 'FLOAT', 'fields': {} }

    elif(p[1].type['type_name'] == 'FLOAT' and
        p[3].type['type_name'] == 'INT'): # FLOAT - INT = FLOAT
        resulting_type = {'type_name': 'FLOAT', 'fields': {} }

    elif(p[1].type['type_name'] == 'FLOAT' and
```

```
            p[3].type['type_name'] == 'FLOAT'): # FLOAT - FLOAT = FLOAT
            resulting_type = {'type_name': 'FLOAT', 'fields': {} }

        elif(p[1].type['type_name'] == 'UNKNOWN'): # UNKNOWN + * = UNKNOWN
            raise TypeError('Linea ' + str(p.lexer.lineno) +
                p[1].text + ': variable no declarada, no tiene valor para operar')

        elif(p[3].type['type_name'] == 'UNKNOWN'): # * + UNKNOWN = UNKNOWN
            raise TypeError('Linea ' + str(p.lexer.lineno) + ': "' +
                p[3].text + '" variable no declarada, no tiene valor para operar')
        else:
            raise TypeError('Linea ' + str(p.lexer.lineno) +
                ': tipos no validos para operacion '+ assign_operator.text)

    else: #asignacion normal
        resulting_type = p[3].type

    if(p[1].isRegisterCall):
        # si p1 es una llamada a un atributo de registro, ej: register.field
        # debo registrar su nuevo valor para el atributo llamado

        regName = p[1].text.rsplit('.', 1)[0] #nombre del registro

        regField = p[1].text.rsplit('.', 1)[1] #nombre del campo

        if(regName.endswith(']')):
            #Estoy sacando un registro de un vector: no puedo modificar sus campos!
            if(p[1].type['type_name'] == 'UNKNOWN'):
                # El tipo del campo es deconocido, es un campo nuevo
                raise TypeError('Linea ' + str(p.lexer.lineno) +
                    ': No se permite agregar campos nuevos a un
                    registro dentro de un vector')

            elif(resulting_type !=  p[1].type):
                # El campo existe, pero se le esta asignando un tipo distinto al que tenia
                raise TypeError('Linea ' + str(p.lexer.lineno) +
                    ': No se permite modificar tipos de campos a un registro
                        dentro de un vector')
        else:
            p[1].type['fields'].update({regField: resulting_type})
                #Actualizo los campos de p[1]

            if(regName in varTypeDict): #Si era una variable existente
                #Actualizo campos del registro en Dict Global
                updateField(regName, regField, resulting_type)
            else:
                # La registro como REGISTER en Dict Global
                registerVar(regName, {'type_name':'REGISTER',
                    'fields':{regField: resulting_type}})

    elif(p[1].isVectorCall):
        # si el tipo del elemento que estoy metiendo en el vector
        # no es el tipo de los elementos del vector, error

        vecName = p[1].text.rsplit('[', 1)[0] #nombre del vector

        expected_type = p[1].type;
        if(expected_type['type_name'] == 'UNKNOWN'):
            # Si el vector no tenia tipo, ahora lo marco con el tipo de p[3]
```

```
                registerVar(vecName, {'type_name':'VECTOR_'+resulting_type['type_name'],
                    'fields':resulting_type})

            elif(resulting_type != expected_type and
                not (isNumeric(expected_type) and isNumeric(resulting_type)) ):
                raise TypeError('Linea ' + str(p.lexer.lineno) +
                        ': Vector con elementos de tipo '
                        + expected_type['type_name'] +
                            ' se le esta insertando un '
                            + resulting_type['type_name'])

        else:
            # registro una variable normalmente
            registerVar(p[1].text, resulting_type)

        p[0] = ASTNode('assignment_expression2', [p[1], p[2], p[3]],
            text=p[1].text+' '+p[2].text+' '+p[3].text,
            type=resulting_type)

def p_assign_operator(p):
    '''assign_operator  : '='
                            | ADDASSIGN
                            | SUBASSIGN
                            | MULASSIGN
                            | DIVASSIGN
    '''
    assign = ASTNode(p[1])
    if(p[1] == '+=' or p[1] == '-=' or p[1] == '*=' or p[1] == '/='):
        p[0] = ASTNode('assign_operator ADD/SUB/MUL/DIV',
            [assign], text=assign.text)
    else:
        p[0] = ASTNode('assign_operator =', [assign], text=assign.text)

def p_conditional_expression(p):
    '''conditional_expression : logical_or_expression
                              | logical_or_expression '?' expression ':' conditional_expression
    '''
    if(len(p) == 2):
        p[0] = ASTNode('conditional_expression1', [p[1]],
            text=p[1].text,
            type=p[1].type,
            isVectorCall=p[1].isVectorCall,
            isRegisterCall=p[1].isRegisterCall )
    else:
        operator_1 = ASTNode(p[2])
        operator_2 = ASTNode(p[4])

        checkTypes([p[1]], ['BOOL'], 'Linea ' + str(p.lexer.lineno) +
                ': Condicion del operador ternario debe ser BOOL')
                #condicion debe ser bool

        if(p[3].type != p[5].type and
            not (isNumeric(p[3].type) and isNumeric(p[5].type)) ):
            raise TypeError('Linea ' + str(p.lexer.lineno) +
                ': Ambas ramas del operador ternario deben ser del mismo tipo')

        p[0] = ASTNode('conditional_expression2',
            [p[1], operator_1, p[3], operator_2, p[5]],
            text=p[1].text+' '+operator_1.text+' '+p[3].text+' '+
```

```
                operator_2.text+' '+p[5].text,
            type=p[5].type )

def p_logical_or_expression(p):
    '''logical_or_expression :  logical_and_expression
                             |  logical_or_expression OR logical_and_expression
    '''
    if(len(p) == 2):
        p[0] = ASTNode('logical_or_expression1', [p[1]],
            text=p[1].text,
            type=p[1].type,
            isVectorCall=p[1].isVectorCall,
            isRegisterCall=p[1].isRegisterCall )
    else:
        operator = ASTNode(p[2])

        checkTypes([p[1], p[3]], ['BOOL'], 'Linea '
            + str(p.lexer.lineno) +
                ': Operandos de un OR deben ser BOOL')

        p[0] = ASTNode('logical_or_expression2', [p[1], operator, p[3]],
            text=p[1].text + ' ' + operator.text + ' ' + p[3].text,
            type={'type_name': 'BOOL', 'fields': {} })

def p_logical_and_expression(p):
    '''logical_and_expression : equality_expression
                              | logical_and_expression AND equality_expression
    '''
    if(len(p) == 2):
        p[0] = ASTNode('logical_and_expression1', [p[1]],
            text=p[1].text,
            type=p[1].type,
            isVectorCall=p[1].isVectorCall,
            isRegisterCall=p[1].isRegisterCall )
    else:
        operator = ASTNode(p[2], text=p[2])

        checkTypes([p[1], p[3]], ['BOOL'],
            'Linea '+ str(p.lexer.lineno) +
                ': Operandos de un AND deben ser BOOL')

        p[0] = ASTNode('logical_and_expression2', [p[1], operator, p[3]],
            text=p[1].text + ' ' + operator.text + ' ' + p[3].text,
            type={'type_name': 'BOOL', 'fields': {} })

def p_equality_expression(p):
    '''equality_expression : relational_expression
                           | equality_expression EQ relational_expression
                           | equality_expression NE relational_expression
    '''
    if(len(p) == 2):
        p[0] = ASTNode('equality_expression1', [p[1]],
            text=p[1].text,
            type=p[1].type,
            isVectorCall=p[1].isVectorCall,
            isRegisterCall=p[1].isRegisterCall )
    else:
        operator = ASTNode(p[2], text=p[2])
```

```
        if(p[1].type != p[3].type):
            raise TypeError('Linea ' + str(p.lexer.lineno) +
                ': Comparacion entre elementos de distinto tipo: ' +
                    p[1].type['type_name'] + ' y '+p[3].type['type_name'])

        p[0] = ASTNode('equality_expression_comparison', [p[1], operator, p[3]],
            text=p[1].text+' '+operator.text+' '+p[3].text,
            type={'type_name': 'BOOL', 'fields': {} })

def p_relational_expression(p):
    '''relational_expression : additive_expression
                             | relational_expression '<' additive_expression
                             | relational_expression '>' additive_expression
    '''
    if(len(p) == 2):
        p[0] = ASTNode('relational_expression1', [p[1]],
            text=p[1].text,
            type=p[1].type,
            isVectorCall=p[1].isVectorCall,
            isRegisterCall=p[1].isRegisterCall )
    else:
        operator = ASTNode(p[2], text=p[2])

        if(p[1].type != p[3].type):
            raise TypeError('Linea ' + str(p.lexer.lineno) +
                ': Comparacion entre elementos de distinto tipo: ' +
                    p[1].type['type_name'] + ' y '+p[3].type['type_name'])

        p[0] = ASTNode('relational_expression_comparison',
            [p[1], operator, p[3]],
            text=p[1].text+' '+operator.text+' '+p[3].text,
            type={'type_name': 'BOOL', 'fields': {} })

def p_additive_expression(p):
    ''' additive_expression : multiplicative_expression
                            | additive_expression '+' multiplicative_expression
                            | additive_expression '-' multiplicative_expression
    '''
    if(len(p) == 2):
        p[0] = ASTNode('additive_expression1', [p[1]],
            text=p[1].text,
            type=p[1].type,
            isVectorCall=p[1].isVectorCall,
            isRegisterCall=p[1].isRegisterCall )
    else:
        operator = ASTNode(p[2], text=p[2])

        if(p[2] == '+'):

            if(p[1].type['type_name'] == 'INT' and
                p[3].type['type_name'] == 'INT'): # INT + INT = INT
                    resulting_type = {'type_name': 'INT', 'fields': {} }

            elif(p[1].type['type_name'] == 'INT' and
                p[3].type['type_name'] == 'FLOAT'): # INT + FLOAT = FLOAT
                    resulting_type = {'type_name': 'FLOAT', 'fields': {} }

            elif(p[1].type['type_name'] == 'FLOAT' and
                p[3].type['type_name'] == 'INT'): # FLOAT + INT = FLOAT
```

```
            resulting_type = {'type_name': 'FLOAT', 'fields': {} }

        elif(p[1].type['type_name'] == 'FLOAT' and
            p[3].type['type_name'] == 'FLOAT'): # FLOAT + FLOAT = FLOAT
            resulting_type = {'type_name': 'FLOAT', 'fields': {} }

        elif(p[1].type['type_name'] == 'STRING' and
            p[3].type['type_name'] == 'STRING'): # STRING + STRING = STRING
            resulting_type = {'type_name': 'STRING', 'fields': {} }

        elif(p[1].type['type_name'] == 'STRING' and
            isNumeric(p[3].type)): # STRING + Numeric = STRING
            resulting_type = {'type_name': 'STRING', 'fields': {} }

        elif(isNumeric(p[1].type) and
            p[3].type['type_name'] == 'STRING'): # Numeric + STRING = STRING
            resulting_type = {'type_name': 'STRING', 'fields': {} }

        elif(p[1].type['type_name'] == 'UNKNOWN'): # * + UNKNOWN = UNKNOWN
            raise TypeError('Linea ' + str(p.lexer.lineno) + ': "' +
                p[1].text + '" variable no declarada, no tiene valor para operar')

        elif(p[3].type['type_name'] == 'UNKNOWN'): # UNKNOWN + * = UNKNOWN
            raise TypeError('Linea ' + str(p.lexer.lineno) +
                p[3].text + ': variable no declarada, no tiene valor para operar')
        else:
            raise TypeError('Linea ' + str(p.lexer.lineno) +
                ': tipos no validos para operacion +')

        # deben ser ambos numeros (suma) o ambos strings (concat)
        p[0] = ASTNode('additive_expression2', [p[1], operator, p[3]],
            text=p[1].text+' '+operator.text+' '+p[3].text,
            type=resulting_type)

    elif(p[2] == '-'):

        if(p[1].type['type_name'] == 'INT' and
            p[3].type['type_name'] == 'INT'): # INT - INT = INT
            resulting_type = {'type_name': 'INT', 'fields': {} }

        elif(p[1].type['type_name'] == 'INT' and
            p[3].type['type_name'] == 'FLOAT'): # INT - FLOAT = FLOAT
            resulting_type = {'type_name': 'FLOAT', 'fields': {} }

        elif(p[1].type['type_name'] == 'FLOAT' and
            p[3].type['type_name'] == 'INT'): # FLOAT - INT = FLOAT
            resulting_type = {'type_name': 'FLOAT', 'fields': {} }

        elif(p[1].type['type_name'] == 'FLOAT' and
            p[3].type['type_name'] == 'FLOAT'): # FLOAT - FLOAT = FLOAT
            resulting_type = {'type_name': 'FLOAT', 'fields': {} }

        elif(p[1].type['type_name'] == 'UNKNOWN'): # UNKNOWN + * = UNKNOWN
            raise TypeError('Linea ' + str(p.lexer.lineno) + ': "' +
                p[1].text + '" variable no declarada, no tiene valor para operar')

        elif(p[3].type['type_name'] == 'UNKNOWN'): # * + UNKNOWN = UNKNOWN
            raise TypeError('Linea ' + str(p.lexer.lineno) + ': "' +
                p[3].text + '" variable no declarada, no tiene valor para operar')
```

```
            else:
                raise TypeError('Linea ' + str(p.lexer.lineno) +
                    ': tipos no validos para operacion -')

            # deben ser numeros
            p[0] = ASTNode('additive_expression3', [p[1], operator, p[3]],
                text=p[1].text+' '+operator.text+' '+p[3].text,
                type=resulting_type)

def p_multiplicative_expression(p):
    ''' multiplicative_expression : power_expression
                                  | multiplicative_expression '*' power_expression
                                  | multiplicative_expression '/' power_expression
                                  | multiplicative_expression '%' power_expression
    '''

    if(len(p) == 2):
        p[0] = ASTNode('multiplicative_expression1', [p[1]],
            text=p[1].text,
            type=p[1].type,
            isVectorCall=p[1].isVectorCall,
            isRegisterCall=p[1].isRegisterCall )
    else:

        checkTypes([p[1], p[3]], ['INT', 'FLOAT'],
            'Linea ' + str(p.lexer.lineno) +
                ': tipos no validos para operacion binaria numerica')

        if(p[1].type['type_name'] == 'INT' and
            p[3].type['type_name'] == 'INT'):
            resulting_type = p[1].type # INT * INT = INT

        else: # Si alguno es float, resultado es float
            resulting_type = {'type_name': 'FLOAT', 'fields': {}}

        operator = ASTNode(p[2], text=p[2])
        p[0] = ASTNode('multiplicative_expression2', [p[1], operator, p[3]],
            text=p[1].text+' '+operator.text+' '+p[3].text,
            type=resulting_type)

def p_power_expression(p):
    '''power_expression : unary_expression
                        | power_expression '^' unary_expression
    '''
    if(len(p) == 2):
        p[0] = ASTNode('power_expression1', [p[1]],
            text=p[1].text,
            type=p[1].type,
            isVectorCall=p[1].isVectorCall,
            isRegisterCall=p[1].isRegisterCall )
    else:
        checkTypes([p[1], p[3]], ['INT', 'FLOAT'],
            'Linea ' + str(p.lexer.lineno) +
                    ': tipos no validos para operacion potencia')
        #Esta es una operacion de enteros

        if(p[1].type['type_name'] == 'INT' and
            p[3].type['type_name'] == 'INT'):
            resulting_type = p[1].type # INT ^ INT = INT
```

```
        else: # Si alguno es float, resultado es float
            resulting_type = {'type_name': 'FLOAT', 'fields': {} }

        operator = ASTNode(p[2], text=p[2])
        p[0] = ASTNode('power_expression2', [p[1], operator, p[3]],
            text=p[1].text+operator.text+p[3].text,
            type=resulting_type)

def p_unary_expression_suffix(p):
    '''unary_expression : suffix_expression'''

    p[0] = ASTNode('unary_expression_suffix', [p[1]],
        text=p[1].text,
        type=p[1].type,
        isVectorCall=p[1].isVectorCall,
        isRegisterCall=p[1].isRegisterCall
    )

def p_unary_expression_increment(p):
    '''unary_expression : INCREMENT unary_expression'''
    leaf = ASTNode(p[1], text=p[1])

    checkTypes([p[2]], ['INT', 'FLOAT'], 'Linea ' + str(p.lexer.lineno) +
        ': Operacion ++ es para numeros') #Esta es una operacion de enteros

    p[0] = ASTNode('unary_expression_increment', [leaf, p[2]],
        text=leaf.text+p[2].text,
        type=p[2].type )

def p_unary_expression_decrement(p):
    '''unary_expression : DECREMENT unary_expression'''
    leaf = ASTNode(p[1], text=p[1])

    checkTypes([p[2]], ['INT', 'FLOAT'], 'Linea ' + str(p.lexer.lineno) +
        ': Operacion -- es para numeros') #Esta es una operacion de enteros

    p[0] = ASTNode('unary_expression_decrement', [leaf, p[2]],
        text=leaf.text+p[2].text,
        type=p[2].type )

def p_unary_expression_not(p):
    '''unary_expression : NOT unary_expression'''
    leaf = ASTNode(p[1], text=p[1])

    checkTypes([p[2]], ['BOOL'], 'Linea '
        + str(p.lexer.lineno) + ': Operacion NOT es para BOOL')

    p[0] = ASTNode('unary_expression_NOT', [leaf, p[2]],
        text=leaf.text+' '+p[2].text,
        type= p[2].type)

def p_unary_expression_signed(p):
    '''unary_expression : '+' unary_expression
                        | '-' unary_expression
    '''

    checkTypes([p[2]], ['INT', 'FLOAT'], 'Linea '
        + str(p.lexer.lineno) +
```

```
            ': Signos sobre una expresion no numerica')

    sign = ASTNode(p[1], text=p[1])
    p[0] = ASTNode('unary_expression_signed', [sign, p[2]],
        text=sign.text+p[2].text,
        type= p[2].type)


#*************** SUFFIX EXPRESSIONS **********************
def p_suffix_expression_vector_index_call(p):
    '''suffix_expression : suffix_expression '[' conditional_expression ']' '''

    left_bracket = ASTNode(p[2], text=p[2])
    right_bracket = ASTNode(p[4], text=p[4])

    vector = p[1]  # debe ser vector

    expression = p[3] # debe ser numero entero > 0
    checkTypes([expression], ['INT'],
        'Linea ' + str(p.lexer.lineno) +
            ': Los indices de un vector deben ser enteros')

    if (expression.text[0] == '-'):
        raise TypeError('Linea ' + str(p.lexer.lineno) +
            ': Los indices de un vector no pueden ser negativos')

    # Si es variable nueva, la marco como vector
    if(vector.type['type_name'] == 'UNKNOWN'):
        vector.type = {'type_name': 'VECTOR', 'fields':{}}
        element_type = {'type_name': 'UNKNOWN', 'fields':{}}
        # Todavia no se el tipo de sus campos
    elif(not isVector(vector.type)):
        raise AttributeError('Linea ' + str(p.lexer.lineno) +
            ': La operacion [] es sobre vectores')
    else:
        # Los vectores son de tipo 'VECTOR_XXX' donde XXX es otro tipo
        # A su vez, los vectores guardan en el campo 'fields' el tipo de sus elementos
        element_type = vector.type['fields']
        #su tipo es el de los elementos del vector (se guarda en 'fields')

    # Si no esta definido el indice para un vector de numeros, devolver 0 (int)
    # Si no esta definido el indice para un vector de chars, devolver "" (string)

    p[0] = ASTNode('suffix_expression_vector_index_call',
        [ vector, left_bracket, expression, right_bracket ],
        text=vector.text + left_bracket.text + expression.text + right_bracket.text,
        type=element_type,
        isVectorCall=True)

def p_suffix_expression_register_call(p):
    '''suffix_expression : suffix_expression '.' ID '''

    dot = ASTNode(p[2], text=p[2])
    identifier = ASTNode(p[3], text=p[3]) # debe ser ID (la produccion lo garantiza)

    register = p[1] # debe ser registro o variable nueva (unknown)
    checkTypes([register], ['REGISTER', 'UNKNOWN'],
        'Linea ' + str(p.lexer.lineno) +
            ': La operacion . es sobre registros')
```

```
    regName = register.text

    if(not (identifier.text in register.type['fields'])):
        if( not(regName in varTypeDict ) or
            not(identifier.text in varTypeDict[regName]['fields'])):
        # Si el campo no esta registrado en los fields del nodo
            (caso sea un registro anonimo) ni
        # tampoco se encuentra en la entrada del registro en el
            diccionario de variables en memoria
        # entonces el campo no existe y es de tipo desconocido
            elementType = {'type_name': 'UNKNOWN', 'fields': {} }
        else:
            # En este caso se esta llamando un atributo de
                un registro ya existente en memoria

            elementType = varTypeDict[regName]['fields'][identifier.text]
    else:
        # En este caso se esta llamando un atributo conocido de un registro
        elementType = register.type['fields'][identifier.text]


    p[0] = ASTNode('suffix_expression_register_call',
        [ register, dot, identifier ],
        text=register.text + dot.text + identifier.text,
        type=elementType,
        isRegisterCall=True)


def p_suffix_expression_increment(p):
    '''suffix_expression : suffix_expression INCREMENT '''
    recursion = p[1]
    operator = ASTNode(p[2], text=p[2])

    checkTypes([recursion], ['INT', 'FLOAT'],
        'Linea ' + str(p.lexer.lineno) +
            ': Operacion ++ es para numeros') # el ++ es operacion de enteros

    p[0] = ASTNode('suffix_expression_increment', [recursion, operator],
        text=recursion.text + operator.text,
        type=recursion.type)

def p_suffix_expression_decrement(p):
    '''suffix_expression : suffix_expression DECREMENT '''
    recursion = p[1]
    operator = ASTNode(p[2], text=p[2])

    checkTypes([recursion], ['INT', 'FLOAT'],
        'Linea ' + str(p.lexer.lineno) +
            ': Operacion -- es para numeros') # el -- es operacion de enteros

    p[0] = ASTNode('suffix_expression_decrement', [recursion, operator],
        text=recursion.text + operator.text,
        type=recursion.type)

def p_suffix_expression_primary(p):
    '''suffix_expression : primary_expression'''

    p[0] = ASTNode('suffix_expression_primary', [p[1]],
```

```
        text=p[1].text, type=p[1].type)

def p_suffix_expression_function_call(p):
    '''suffix_expression : function_call_expression '''
    p[0] = ASTNode('suffix_expression_function_call', [p[1]],
        text=p[1].text, type=p[1].type)


#*************** FUNCTION CALL EXPRESSIONS **********************
def p_function_call_expression_ME(p):
    '''function_call_expression :
    FUNC_ME '(' assignment_expression ',' assignment_expression ',' assignment_expression ')'
| FUNC_ME '(' assignment_expression ',' assignment_expression ')'
    '''

    function_terminal = ASTNode(p[1], text=p[1])
    left_parethesis = ASTNode(p[2], text=p[2])

    argument1 = p[3] #debe ser vector de numeros
    checkTypes([argument1], ['VECTOR_INT', 'VECTOR_FLOAT'],
        'Linea ' + str(p.lexer.lineno) +
            ': Operacion multiplicacionEscalar, parametro 1 debe ser vector de numeros')

    argument2 = p[5] #debe ser un numero
    checkTypes([argument2], ['FLOAT', 'INT'],
        'Linea ' + str(p.lexer.lineno) +
            ': Operacion multiplicacionEscalar, parametro 2 debe ser numero')

    comma1 = ASTNode(p[4], text=p[4])

    if(len(p) == 9):

        comma2 = ASTNode(p[6], text=p[6])
        argument3 = p[7]

        checkTypes([argument3], ['BOOL'],
            'Linea ' + str(p.lexer.lineno) +
                ': Operacion multiplicacionEscalar, parametro 3 debe ser BOOL')

        right_parethesis = ASTNode(p[8], text=p[8])

        children = [function_terminal, left_parethesis, argument1,
            comma1, argument2, comma2, argument3, right_parethesis]

        text = (function_terminal.text + left_parethesis.text +
            argument1.text + comma1.text
            +' '+ argument2.text +  comma2.text
            +' '+ argument3.text + right_parethesis.text)

        p[0] = ASTNode('function_call_expression_ME_full', children,
            text=text, type={'type_name': 'VECTOR_INT',
                'fields': {'type_name': 'INT', 'fields': {} }} )
    else:
        right_parethesis = ASTNode(p[6], text=p[6])

        children = [function_terminal, left_parethesis,
            argument1, comma1, argument2, right_parethesis]

        text = (function_terminal.text + left_parethesis.text +
```

```
            argument1.text + comma1.text +' '+
            argument2.text + right_parethesis.text)

        if(argument1.type['type_name'] == 'VECTOR_INT'
                and argument2.type['type_name'] == 'INT'):
            # VECTOR_INT * INT = VECTOR_INT
            resulting_type = {'type_name': 'VECTOR_INT',
                'fields': {'type_name': 'INT', 'fields': {} }}

        elif(p[1].type['type_name'] == 'VECTOR_INT'
                and p[3].type['type_name'] == 'FLOAT'):
             # VECTOR_INT * FLOAT = VECTOR_FLOAT
            resulting_type = {'type_name': 'VECTOR_FLOAT',
                'fields': {'type_name': 'FLOAT', 'fields': {} }}

        elif(p[1].type['type_name'] == 'VECTOR_FLOAT'):
            # VECTOR_FLOAT * INT/FLOAT = VECTOR_FLOAT
            resulting_type = {'type_name': 'VECTOR_FLOAT',
                'fields': {'type_name': 'FLOAT', 'fields': {} }}

        p[0] = ASTNode('function_call_expression_ME_short', children,
            text=text, type=resulting_type)

def p_function_call_expression_CAP(p):
    '''function_call_expression : FUNC_CAP '(' assignment_expression ')'
    '''

    function_terminal = ASTNode(p[1], text=p[1])
    left_parethesis = ASTNode(p[2], text=p[2])

    argument = p[3] #debe ser string
    checkTypes([argument], ['STRING'],
        'Linea ' + str(p.lexer.lineno) +
            ': Operacion Capitalizar, parametro debe ser STRING')

    right_parethesis = ASTNode(p[4], text=p[4])

    children = [function_terminal, left_parethesis, argument, right_parethesis]

    text = (function_terminal.text + left_parethesis.text +
        argument.text + right_parethesis.text)

    p[0] = ASTNode('function_call_expression_CAP', children,
        text=text, type={'type_name': 'STRING', 'fields': {}})# DEVUELVE STRING


def p_function_call_expression_COL(p):
    '''function_call_expression : FUNC_COL '(' assignment_expression ',' assignment_expression ')'
    '''

    function_terminal = ASTNode(p[1], text=p[1])
    left_parethesis = ASTNode(p[2], text=p[2])

    argument1 = p[3] #debe ser vector de numeros
    argument2 = p[5] #debe ser vector de numeros

    checkTypes([argument1, argument2], ['VECTOR_INT', 'VECTOR_FLOAT'],
        'Linea ' + str(p.lexer.lineno) +
            ': Operacion Colineales, parametros deben ser vectores numericos')
```

```
    comma = ASTNode(p[4], text=p[4])

    right_parethesis = ASTNode(p[6], text=p[6])

    children = [function_terminal, left_parethesis,
        argument1, comma, argument2, right_parethesis]

    text = (function_terminal.text + left_parethesis.text + argument1.text
        + comma.text + ' ' +
        argument2.text + right_parethesis.text)

    p[0] = ASTNode('function_call_expression_COL', children,
        text=text, type={'type_name': 'BOOL', 'fields': {}}) # DEVUELVE BOOL


def p_function_call_expression_PR(p):
    '''function_call_expression : FUNC_PR '(' assignment_expression ')'
    '''

    function_terminal = ASTNode(p[1], text=p[1])
    left_parethesis = ASTNode(p[2], text=p[2])

    argument = p[3]

    right_parethesis = ASTNode(p[4], text=p[4])

    children = [function_terminal, left_parethesis, argument, right_parethesis]

    text = (function_terminal.text + left_parethesis.text +
        argument.text + right_parethesis.text)

    p[0] = ASTNode('function_call_expression_PR', children, text=text)# DEVUELVE VOID

def p_function_call_expression_LE(p):
    '''function_call_expression : FUNC_LE '(' assignment_expression ')'
    '''

    function_terminal = ASTNode(p[1], text=p[1])
    left_parethesis = ASTNode(p[2], text=p[2])

    argument = p[3] #debe ser string

    if(not (isVector(argument.type) or argument.type['type_name']=='STRING')):
        raise TypeError('Linea '
            + str(p.lexer.lineno) +
                ': Operacion Length, parametro debe ser VECTOR o STRING')

    right_parethesis = ASTNode(p[4], text=p[4])

    children = [function_terminal, left_parethesis, argument, right_parethesis]

    text = (function_terminal.text + left_parethesis.text +
        argument.text + right_parethesis.text)

    p[0] = ASTNode('function_call_expression_LE', children,
        text=text, type={'type_name': 'INT', 'fields': {}} )# DEVUELVE INT

#*************** PRIMARY EXPRESSIONS **********************
```

```
def p_primary_expression(p):
    '''primary_expression : '(' expression ')' '''
    leaf_left = ASTNode(p[1])
    leaf_right = ASTNode(p[3])

    p[0] = ASTNode('p_primary_expression6 - ( expression )',
        [ leaf_left, p[2], leaf_right ],
        text=leaf_left.text+p[2].text+leaf_right.text,
        type=p[2].type ) #su tipo sera sintetizado por la recursion

def p_primary_expression_id(p):
    '''primary_expression : ID'''
    leaf = ASTNode(p[1])

    if(leaf.label in varTypeDict):
        resulting_type = varTypeDict[leaf.text]
    else:
        resulting_type = {'type_name': 'UNKNOWN', 'fields': {} }

    p[0] = ASTNode('p_primary_expression1 - ID', [leaf],
        text=leaf.text,
        type=resulting_type)
    # a esta altura todavia no se el tipo de la variable


def p_primary_expression_res(p):
    '''primary_expression : RES'''
    leaf = ASTNode(p[1])

    if(leaf.label in varTypeDict):
        resulting_type = varTypeDict[leaf.text]
    else:
        resulting_type = {'type_name': 'UNKNOWN', 'fields': {} }
        # a esta altura todavia no se el tipo de la variable

    p[0] = ASTNode('p_primary_expression1 - RES', [leaf],
        text=leaf.text,
        type=resulting_type)


def p_primary_expression_begin(p):
    '''primary_expression : BEGIN'''
    leaf = ASTNode(p[1])

    if(leaf.label in varTypeDict):
        resulting_type = varTypeDict[leaf.text]
    else:
        resulting_type = {'type_name': 'UNKNOWN', 'fields': {} }

    p[0] = ASTNode('p_primary_expression1 - BEGIN', [leaf],
        text=leaf.text,
        type=resulting_type)
    # a esta altura todavia no se el tipo de la variable

def p_primary_expression_end(p):
    '''primary_expression : END'''
    leaf = ASTNode(p[1])

    if(leaf.label in varTypeDict):
```

```
        resulting_type = varTypeDict[leaf.text]
    else:
        resulting_type = {'type_name': 'UNKNOWN', 'fields': {} }

    p[0] = ASTNode('p_primary_expression1 - END', [leaf],
        text=leaf.text,
        type=resulting_type)
    # a esta altura todavia no se el tipo de la variable

def p_primary_expression_string(p):
    '''primary_expression : STRING'''
    leaf = ASTNode(p[1])

    p[0] = ASTNode('p_primary_expression2 - STRING', [leaf],
        text=leaf.text,
        type={'type_name': 'STRING', 'fields': {}} )

def p_primary_expression_bool(p):
    '''primary_expression : BOOL'''
    leaf = ASTNode(p[1])

    p[0] = ASTNode('p_primary_expression3 - BOOL', [leaf],
        text=leaf.text,
        type={'type_name': 'BOOL', 'fields': {}} )

def p_primary_expression_float(p):
    '''primary_expression : FLOAT'''
    leaf = ASTNode(p[1])
    p[0] = ASTNode('p_primary_expression4 - FLOAT', [leaf],
        text=leaf.text,
        type={'type_name': 'FLOAT', 'fields': {}} )

def p_primary_expression_number(p):
    '''primary_expression : INT'''
    leaf = ASTNode(p[1])
    p[0] = ASTNode('p_primary_expression5 - NUMBER', [leaf],
        text=leaf.text,
        type={'type_name': 'INT', 'fields': {}} )

def p_primary_expression_vector(p):
    '''primary_expression : '[' expression ']' '''

    # Esta produccion corresponde a la creacion de un nuevo vector

    left_bracket = ASTNode(p[1], text=p[1])

    # el tipo expression, si es de tipo lista de elementos, por ej: 2,3,4,5
    # ya automaticamente chequea que sean todos del mismo tipo

    element_list = p[2]
    right_bracket = ASTNode(p[3], text=p[3])

    children = [left_bracket, element_list, right_bracket]
    text = left_bracket.text + element_list.text + right_bracket.text

    p[0] = ASTNode('primary_expression_vector', children,
        text=text,
        type={'type_name': 'VECTOR_'+ element_list.type['type_name'],
                'fields': element_list.type } )
```

```
        # DEVUELVE VECTOR

def p_primary_expression_register(p):
    '''primary_expression :  '{' field_value_expressions '}' '''

    left_bracket = ASTNode(p[1], text=p[1])
    right_bracket = ASTNode(p[3], text=p[3])

    field_value_list = p[2]

    children = [left_bracket, field_value_list, right_bracket]
    text = left_bracket.text + field_value_list.text + right_bracket.text

    p[0] = ASTNode('primary_expression_register', children,
        text=text,
        type={'type_name':'REGISTER',
                'fields':field_value_list.type['fields'] })
        # DEVUELVE REGISTRO

def p_field_value_expressions(p):
    '''field_value_expressions : field_value_expression
                               | field_value_expressions ',' field_value_expression
    '''

    if(len(p) == 2):

        p[0] = ASTNode('field_value_expressions_single', [p[1]],
            text=p[1].text,
            type=p[1].type)
    else:
        comma = ASTNode(p[2], text=p[2])

        #mergeo los fields de sus hijos para darlos al padre
        p[1].type['fields'].update(p[3].type['fields'])

        p[0] = ASTNode('field_value_expressions', [p[1], comma, p[3]],
            text=p[1].text + comma.text + ' ' + p[3].text,
            type=p[1].type)
def p_field_value_expression(p):
    '''field_value_expression : ID ':' conditional_expression '''

    field = ASTNode(p[1], text=p[1])
    separator = ASTNode(p[2], text=p[2])
    value = p[3]

    p[0] = ASTNode('field_value_expression', [field, separator, value],
            text=field.text + separator.text + ' ' + value.text,
            type={'type_name': 'UNKNOWN',
                    'fields':{field.text: p[3].type}})

def p_error(p):
    try:
        raise SyntaxError("Linea {1}: Error de sintaxis '{0}', posicion '{2}' "
            .format(p.value, p.lexer.lineno, p.lexer.lexpos))

    except AttributeError:
        print "Error de sintaxis. No se pudo determinar la ubicacion.
            Falta ';' al final del archivo?."
```

5. Bibliografía

Referencias

[1] ANSI C Yacc grammar - Jeff Lee
`https://www.lysator.liu.se/c/ANSI-C-grammar-y.html`