



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Teoría de lenguajes

Trabajo práctico

Parser

Resumen

Este trabajo consiste en parser estilo c++

Integrante	LU	Correo electrónico
Acosta, Javier Sebastian	338/11	acostajavier.ajs@gmail.com
Mastropasqua Nicolas Ezequiel	828/13	mastropasqua.nicolas@gmail.com
Negri, Franco	893/13	franconegri2004@hotmail.com

Palabras claves:

TP

Contents

1	Introducción	3
2	Gramatica	4
2.1	Descripción General	4
2.1.1	Gramática inicial	4
2.1.2	Gramática ambigua	4
2.1.3	Gramática no ambigua	4
2.2	Descripción de la gramática implementada:	4
2.2.1	Valores	4
2.2.2	Expresiones Matemáticas	5
2.2.3	Expresiones booleanas	6
2.2.4	Expresiones de string	6
2.2.5	Problema del “Dangling else”:	6
2.3	Gramática Final	7
3	Implementacion del Lenguaje	12
3.1	Implementacion Del Lexer	12
3.2	Implementacion Del Parser	13
3.3	Implementacion Del Chequeo de Tipos	13
4	Conclusiones	14
5	Conclusiones	14

1 Introducción

El objetivo del presente trabajo práctico es la realización de un analizador léxico y sintáctico para un lenguaje de scripting, mediante el uso de una gramática LALR y la herramienta para generar analizadores sintácticos **PLY**.

Esperamos que al final del trabajo el analizador sea capaz de reconocer cadenas de caracteres similares a las del lenguaje C++.

Para ello comenzaremos generando una gramática que cumpla con las reglas del lenguaje. Luego procederemos a implementar esto y a desarrollar un chequeo de tipos que permita filtrar programas que si bien son sintacticamente validos, posean una semantica invalida (por ejemplo, multiplicar una cadena de caracteres con otra).

Ademas, dado un texto valido querremos guardarlo en un archivo nuevo indentandolo de manera correcta automatica, agregando o quitando espacios, tabs, saltos de linea donde sea necesario.

2 Gramatica

2.1 Descripción General

2.1.1 Gramática inicial

Comenzamos el armado de la gramática definiendo los tipos básicos que debíamos aceptar en nuestro lenguaje.

Así definimos los tipos Bool, String, Float, Bool, Int, Vector, Registro. De allí el siguiente paso fue ver que operaciones se le podía realizar a cada uno de ellos.

Mientras realizábamos esto empezamos a notar que necesitaríamos determinar la precedencia de estas operaciones y la asociatividad. Para esto nos basamos en la precedencia de c++¹ dado lo parecido que es al lenguaje de entrada.

Una vez que todas las expresiones del lenguaje estaban completas, lo pasamos a código Python usando la herramienta *PLY*. Ya terminado el código, nos encontramos con que la gramática tenía muchos mas conflictos que los que se podía ver a simple vista y por lo tanto no era LALR.

2.1.2 Gramática ambigua

Luego de resolver algunos conflictos, nos dimos cuenta de que era un proceso complicado pues cada conflicto implicaba añadir mas reglas a la gramática haciendo que sea mas compleja y difícil de entender.

Después de leer la documentación de ply² y observar que ply puede encargarse de los conflictos de una gramática ambigua usando reglas de asociatividad y precedencia, decidimos hacer una gramática ambigua sencilla y aprovechar esta característica de ply.

2.1.3 Gramática no ambigua

Si bien esto solucionó todos los conflictos de una manera sencilla, por desgracia, nos comunicaron que lo que se pedía era una gramática **no** ambigua. Por lo tanto decidimos tirar la gramática y comenzar de nuevo.

Con el tiempo contado y sin ideas mejores nos dimos cuenta de que la gramática inicial no estaba tan mal, tenía toda la precedencia definida de las expresiones y que una solución rápida era hacer la gramática menos restrictiva, es decir, que acepte mas cadenas de lo que se pida y rechazar las que no pertenecen al lenguaje mediante atributos. La tarea de los atributos (además de armar el código indentado) es chequear los tipos de las expresiones para que, si una operación acepta unos tipos determinados y se le pasa una expresión de otro tipo, lance una excepción de error de tipos. Dicha gramática se encuentra definida en la seccion 5

2.2 Descripción de la gramática implementada:

2.2.1 Valores

Los valores son cualquier expresión que denote un valor (por ejemplo $3 * 4 + 3$). La primer idea era definir valores como **cualquier** expresión (para poder usarla como valor en asignaciones, en funciones como parámetro, etc.):

¹http://en.cppreference.com/w/cpp/language/operator_precedence

² <http://www.dabeaz.com/ply/ply.html>

$Valores \rightarrow ExpresionMatematica \mid ExpresionBooleana \mid ExpresionString \mid ExpresionVector \mid ExpresionRegistro \mid OperacionVariables$

Donde cada producción que empieza con *Expresion* denota una expresión de un tipo determinado que también puede generar los valores primitivos (bool, string, float, bool, int, vec, reg), variables (id) y además funciones que retornen valores a su tipo correspondiente (por ejemplo *capitalizar* en *ExpresionString*). *OperacionVariables* genera las expresiones que usan operaciones de variables (asignación, incremento, decremento, etc).

El problema con esto es que **id** se puede generar en todas las expresiones salvo en la última. Si se tiene una producción $A \rightarrow Valores$, entonces se puede llegar de 5 maneras distintas a **id** (es decir, hay mas de una posible derivacion mas a la izquierda para la cadena *id*) y, por lo tanto, hay conflictos (la gramática es ambigua).

Por lo tanto decidimos sacar a **id** de cada producción. El problema es que se tiene que seguir produciendo expresiones complejas con **id** (por ejemplo *id and id*).

Para solucionar esto, decidimos sacar todos los valores primitivos de cada expresión y colocarlo en *Valores*. Además hay que observar que una función de un determinado tipo puede devolver una variable de ese tipo, por lo que decidimos sacar también a esas funciones. Lo mismo pasa con los operadores ternarios. De esta forma, cada producción *Expresion* genera expresiones con al menos un operador de su tipo:

$Valores \rightarrow ExpresionMatematica \mid ExpresionBooleana \mid ExpresionString \mid ExpresionVector \mid ExpresionRegistro \mid OperacionVariables \mid bool \mid string \mid float \mid bool \mid int \mid vec \mid reg \mid id \mid FuncReturn \mid Ternario$

$FuncReturn \rightarrow FuncInt \mid FuncString \mid FuncBool \mid FuncVector$

$Ternario \rightarrow TernarioMat \mid TernarioBool \mid TernarioString \mid TernarioVector \mid TernarioRegistro$

2.2.2 Expresiones Matemáticas

Para las expresiones matemáticas consideramos la siguiente tabla de precedencia:

Tipo	Operador	Asociatividad
Binario	+, -	izquierda
Binario	*, /, %	izquierda
Binario	^	izquierda
Unario	+, -	
Unario	()	

Table 1: Tabla de menor a mayor precedencia

La producción que se encarga de realizar estas expresiones es eMat.

La primera aproximación fue hacer la gramática para poder respetar la precedencia y asociatividad de los operadores.

Este proceso se logra teniendo en cuenta que se puede fijar la precedencia (por ejemplo entre + y *), con las reglas de la gramática. Como bien se sabe, el operador + tiene menos precedencia que el *. Una gramática que respeta esto es la siguiente:

$$\begin{aligned} A &\rightarrow A + B \\ B &\rightarrow B * C \\ C &\rightarrow int \mid float \end{aligned}$$

Esto se debe a que antes de calcular la suma entre A y B , hay que parsear la producción B (pues es un parser ascendente).

Esto también respeta la asociatividad de ambos operados (que es a izquierda) por la misma razón que antes: La única forma de colocar mas de un operador $+$ es usando la producción A y, por lo tanto, se tiene que parsear A antes que calcular la suma.

Este proceso se realiza con todos los operadores, de manera que si 2 operadores tienen la misma precedencia, entonces parten de un mismo no terminal.

A esta gramática le falta la posibilidad de agregar paréntesis para poder cambiar la precedencia. Una forma de hacerlos es agregarlos a C para que estos tengan mas prioridad que todas las operaciones.

Además también falta poder agregar variables. Para esto agregamos un **no-terminal** a C llamado id que las representa. La gramática resultante es esta:

$$\begin{aligned} A &\rightarrow A + B \\ B &\rightarrow B * C \\ C &\rightarrow (A) | int | float | id \end{aligned}$$

Para evitar problemas de ambigüedad, decidimos hacer que eMat devuelva expresiones con al menos un operador matemático. Esto es por que en la producción valores se tienen las variables (correspondientes al token ID) y estas también se generarían en las expresiones de los otros tipos. Ver explicación en 4.1

2.2.3 Expresiones booleanas

Las expresiones booleanas también se solucionan de la misma manera que las expresiones matemáticas. Para mas información ver sección 5.

2.2.4 Expresiones de string

En principio consideramos la siguiente gramática:

$$ExpresionString \rightarrow ExpresionString + ExpresionString \mid string$$

Esta gramática nos planteó un problema en el parser debido a que comparte un operador con las expresiones matemáticas. El problema viene cuando se tiene una expresión $a + b$ en donde no está claro que tipo le corresponde e introduce una ambigüedad.

Por lo tanto decidimos fusionar a las expresiones de string con las expresiones matemáticas y restringir (mediante chequeo de tipos) que un string no se le puede hacer otra operación que no sea la suma.

2.2.5 Problema del “Dangling else”:

Otro problema que encontramos ya avanzados en el proceso de desarrollo de la gramática fue el problema del “Dangling else”³ que consiste en que los else opcionales en un if hacen que la gramática sea ambigua.

Por ejemplo:

```
if cond1 then if cond2 then sentencia1 else sentencia2
```

³https://en.wikipedia.org/wiki/Dangling_else

La ambigüedad surge de no poder decidir si la sentencia2 se ejecuta como rama alternativo a cond1(es decir, cuando esta condicion es falsa), o bien como la rama alternativa del *if* asociado a **cond2**. Esto último es posible ya que el *if* de **cond1** puede prescindir del bloque *else*. Para solucionar esto, construimos una solución basada en el concepto de líneas abiertas o cerradas. La primera, son todas aquellas que contienen al menos un *if* que no tiene un *else* correspondiente o bien un bucle con línea abierta. La segunda mencionada, son aquellas que resultan de tener sentencias simples, comentarios, o bien:

```
if (cond1) bloqueCerrado else bloqueCerrado
```

Donde bloqueCerrado puede ser cualquier conjunto de sentencias escritas entre llaves, sentencias simples o , razonando inductivamente, líneas cerradas. También se incluyen los bucles con líneas cerradas. Finalmente, la situación se complejizó un poco ya que hay que considerar las distintas combinaciones recién mencionadas, sumando la posibilidad de tener, o no, comentarios en cada uno de estos bloques.

2.3 Gramática Final

A continuación se define la gramática utilizada para construir el parser. La misma es **no ambigua** y **LALR**. La garantía de esto es que ply acepta gramáticas de este tipo, y nuestra implementación no arroja conflictos **shift/reduce** o **reduce/reduce**

$$G \rightarrow \langle V_t, V_{nt}, g, P \rangle$$

V_t es el conjunto de símbolos terminales dado por los símbolos en **mayúsculas** que aparecen en las producciones.

V_{nt} es el conjunto de símbolos no-terminales dado los literales(operadores) y los símbolos en **minúsculas** que aparecen en las producciones.

P es el conjunto de producciones dadas a continuación|

Sentencias y estructura general

$$g \rightarrow \text{línea } g \mid \text{COMMENT } g \mid \text{empty}$$

$$\text{línea} \rightarrow \text{lAbierta} \mid \text{lCerrada}$$

Línea Abierta: Hay por lo menos un IF que no matchea con un else

$$\begin{aligned} \text{lAbierta} \rightarrow & \text{IF (cosaBooleana) línea} \mid \\ & \mid \text{IF (cosasBooleana) \{ g \} ELSE lAbierta} \\ & \mid \text{IF (cosasBooleana) \{ g \} ELSE lAbierta} \\ & \mid \text{IF (cosasBooleana) \{ g \}} \\ & \mid \text{loop lAbierta} \end{aligned}$$

Las siguientes son las variantes de tener bloques cerrados else bloques cerrados.

Un bloque cerrado puede ser una sentencia única o un bloque entre llaves.

En cada uno de estos casos puede haber, o no, comentarios. De ahí todas estas combinaciones.

lCerrada \rightarrow sentencia

| IF (cosaBooleana) { g } ELSE { g }
 | IF (cosaBooleana) lCerrada ELSE { g }
 | IF (cosaBooleana) COMMENT com lCerrada ELSE { g }
 | IF (cosaBooleana) { g } ELSE lCerrada
 | IF (cosaBooleana) lCerrada ELSE lCerrada
 | IF (cosaBooleana) COMMENT com lCerrada ELSE lCerrada
 | IF (cosaBooleana) lCerrada ELSE COMMENT com lCerrada
 | IF (cosaBooleana) COMMENT com lCerrada ELSE COMMENT com lCerrada
 | loop { g }
 | loop lCerrada
 | loop COMMENT com lCerrada
 | DO { g } WHILE (valores) ;
 | DO lCerrada WHILE (valores) ;
 | DO COMMENT com lCerrada WHILE (valores) ;

com \rightarrow COMMENT com | λ

Sentencias básicas:

sentencia \rightarrow varsOps | func ; | varAsig ; | RETURN; | ;

Bucles:

loop \rightarrow WHILE (valores) | FOR (primerParam ; valores ; tercerParam)

primerParam \rightarrow varAsig | λ

tercarParam \rightarrow varsOps | varAsig | func | λ

cosaBooleana \rightarrow expBool | valoresBool

Funciones:

func \rightarrow FuncReturn | FuncVoid

funcReturn \rightarrow FuncInt | FuncString | FuncBool

funcInt \rightarrow MULTIESCALAR(valores, valores param)

funcInt \rightarrow LENGTH(valores)

funcString \rightarrow CAPITALIZAR(valores)

funcBool \rightarrow colineales(valores, valores)

funcVoid \rightarrow print(Valores)

param \rightarrow valores | λ

Vectores y variables:

vec \rightarrow [elem]

elem \rightarrow valores,elem | valores

vecval \rightarrow id [expresion] | vec [expresion] | vecVal [expresion] | ID[INT]

expresion \rightarrow eMat | expBool | funcReturn | reg | FLOAT | | STRING | | RES

| BOOL | varYVals | varsOps | vec | atributos | ternario

valores \rightarrow varYVals | varsOps | eMat | expBool | funcReturn | reg | INT | FLOAT

| STRING | BOOL | ternario | atributos | vec | RES

atributos \rightarrow ID.valoresCampos | reg.valoresCampos

valoresCampos \rightarrow varYVals | END | BEGIN

Operadores ternarios:

ternario \rightarrow ternarioMat | ternarioBool | (ternarioBool) | (ternarioMat)

| ternarioVars | (ternarioVars)

ternarioVars \rightarrow valoresBool ? valoresTernarioVars : valoresTernarioVars

| valoresBool ? valoresTernarioVars : valoresTernarioMat

| valoresBool ? valoresTernarioMat : valoresTernarioVars

| valoresBool ? valoresTernarioVars : valoresTernarioBool

| valoresBool ? valoresTernarioBool : valoresTernarioVars

| expBool ? valoresTernarioVars : valoresTernarioVars

| expBool ? valoresTernarioVars : valoresTernarioMat

| expBool ? valoresTernarioMat : valoresTernarioVars

| expBool ? valoresTernarioVars : valoresTernarioBool

| expBool ? valoresTernarioBool : valoresTernarioVars

valoresTernarioVars \rightarrow reg | vec | ternarioVars | (ternarioVars) | atributos

| varsOps | varYVals | RES

TernarioMat \rightarrow valoresBool ? valoresTernarioMat : valoresTernarioMat

| expBool ? valoresTernarioMat : valoresTernarioMat

valoresTernarioMat \rightarrow INT | FLOAT | funcInt | STRING | eMat

| ternarioMat | (ternarioMat)

ternarioBool \rightarrow valoresBool ? valoresTernarioBool : valoresTernarioBool

| expBool ? valoresTernarioBool : valoresTernarioBool

valoresTernarioBool \rightarrow BOOL | funcBool | ternarioBool | (ternarioBool)| expBool

varYVals:

$\text{varYVals} \rightarrow \text{ID} \mid \text{vecVal} \mid \text{vecVal}.\text{varYVals}$

Registros:

$\text{reg} \rightarrow \{ \text{campos} \}$

$\text{campos} \rightarrow \text{ID}:\text{valores}, \text{campos} \mid \text{ID}:\text{valores}$

Operadores de variables:

$\text{varsOps} \rightarrow \text{MENOSMENOS} \text{ varYVals} \mid \text{MASMAS} \text{ varYVals}$
 $\mid \text{varYVals} \text{ MASMAS} \mid \text{varYVals} \text{ MENOSMENOS}$

Asignaciones:

$\text{varAsig} \rightarrow \text{variable} \text{ MULEQ} \text{ valores} \mid \text{variable} \text{ DIVEQ} \text{ valores} \mid \text{variable} \text{ MASEQ} \text{ valores}$
 $\mid \text{variable} \text{ MENOSEQ} \text{ valores} \mid \text{variable} = \text{valores} \mid \text{ID} . \text{ID} = \text{valores}$
 $\text{variable} \rightarrow \text{ID} \mid \text{vecVal} \mid \text{vecVal}.\text{varYVals}$

Operaciones binarias enteras:

$\text{valoresMat} \rightarrow \text{INT} \mid \text{FLOAT} \mid \text{funcInt} \mid \text{atributos} \mid \text{funcString}$
 $\mid \text{STRING} \mid \text{varYVals} \mid \text{varsOps} \mid (\text{ternarioMat})$
 $\text{eMat} \rightarrow \text{eMat} + \text{p} \mid \text{valoresMat} + \text{p} \mid \text{eMat} + \text{valoresMat} \mid \text{valoresMat} + \text{valoresMat}$
 $\mid \text{eMat} - \text{p} \mid \text{valoresMat} - \text{p} \mid \text{eMat} - \text{valoresMat} \mid \text{valoresMat} - \text{valoresMat} \mid \text{p}$
 $\text{p} \rightarrow \text{p} * \text{exp} \mid \text{p} / \text{exp} \mid \text{p} \mid \text{valoresMat} * \text{exp} \mid \text{valoresMat} / \text{exp} \mid \text{valoresMat}$
 $\mid \text{p} * \text{valoresMat} \mid \text{p} / \text{valoresMat} \mid \text{p} \% \text{valoresMat} \mid \text{valoresMat} * \text{valoresMat}$
 $\mid \text{valoresMat} \text{ valoresMat} \mid \text{valoresMat} \% \text{valoresMat} \mid \text{exp}$
 $\text{exp} \rightarrow \text{exp} \text{ iSing} \mid \text{valoresMat} \text{ iSing} \mid \text{exp} \text{ valoresMat}$
 $\mid \text{valoresMat} \text{ valoresMat} \mid \text{iSing}$
 $\text{iSing} \rightarrow - \text{paren} \mid + \text{paren} \mid - \text{valoresMat} \mid + \text{valoresMat} \mid \text{paren}$
 $\text{paren} \rightarrow (\text{eMat}) \mid (\text{valoresMat})$

Expresiones booleanas:

$\text{valoresBool} \rightarrow \text{BOOL} \mid \text{funcBool} \mid \text{varYVals} \mid \text{varsOps} \mid (\text{ternarioBool})$
 $\text{expBool} \rightarrow \text{expBool} \text{ OR} \text{ and} \mid \text{valoresBool} \text{ OR} \text{ and} \mid$
 $\mid \text{expBool} \text{ OR} \text{ valoresBool} \mid \text{valoresBool} \text{ OR} \text{ valoresBool} \mid \text{and}$
 $\text{and} \rightarrow \text{and} \text{ AND} \text{ eq} \mid \text{valoresBool} \text{ AND} \text{ eq} \mid \text{and} \text{ AND} \text{ valoresBool}$
 $\mid \text{valoresBool} \text{ AND} \text{ valoresBool} \mid \text{eq}$
 $\text{eq} \rightarrow \text{eq} \text{ EQEQ} \text{ mayor} \mid \text{eq} \text{ DISTINTO} \text{ mayor} \mid \text{tCompareEQ} \text{ EQEQ} \text{ mayor}$

$$\begin{aligned}
& | \text{tCompareEQ DISTINTO mayor} | \text{eq EQEQ tCompareEQ} \\
& | \text{eq DISTINTO tCompareEQ tCompareEQ EQEQ tCompareEQ} \\
& | \text{tCompareEQ DISTINTO tCompareEQ} | \text{mayor} \\
\text{tCompareEQ} & \rightarrow \text{BOOL} | \text{funcBool} | \text{varYVals} | \text{varsOps} | \text{INT} \\
& | \text{FLOAT} | \text{funcInt} | \text{eMat} | (\text{ternarioBool}) | (\text{ternarioMat}) \\
\text{tCompare} & \rightarrow \text{eMat} | \text{varsOps} | \text{varYVals} | \text{INT} | \text{funcInt} | \text{FLOAT} | (\text{ternarioMat}) \\
\text{mayor} & \rightarrow \text{tCompare} > \text{tCompare} | \text{menor} \\
\text{menor} & \rightarrow \text{tCompare} < \text{tCompare} | \text{not} \\
\text{not} & \rightarrow \text{NOT not} | \text{NOT valoresBool} | \text{parenBool} \\
\text{parenBool} & \rightarrow (\text{expBool})
\end{aligned}$$

3 Implementacion del Lenguaje

3.1 Implementacion Del Lexer

Para la construccion del lexer se definió un conjunto de literales con operadores y otros simbolos propios del lenguaje.

literals = `[+, -, *, /, %, <, >, =, !, {, }, (,), [,], ?, :, ;, ,, .]`

A su vez, utilizamos otra funcionalidad de ply para definir las palabras reservadas del lenguaje:

```
reserved = { 'begin' : 'BEGIN', 'end' : 'END', 'while' : 'WHILE', 'for' : 'FOR',  
'if' : 'IF', 'else' : 'ELSE', 'do' : 'DO', 'res' : 'RES', 'return' : 'RETURN',  
'AND' : 'AND', 'OR' : 'OR', 'NOT' : 'NOT', 'print' : 'PRINT',  
'multiplicacionEscalar': 'MULTIESCALAR', 'capitalizar': 'CAPITALIZAR',  
'colineales': 'COLINEALES', 'print': 'PRINT', 'length': 'LENGTH', }
```

Esto permitió evitar tener que definir demasiadas reglas simples para este tipo de operadores o palabras claves.

Para el resto de los tokens definidos, fue necesario utilizar expresiones regulares, se muestra mas abajo como se definieron cada una de ellos: (Por claridad, se omite el resto del codigo para la regla de los tipos, ya que es análoga a la de **string**):

```
t_EQEQ = r"=="  
t_DISTINTO = r"!="  
t_MENOSEQ = r"-="  
t_MASEQ = r"\+="  
t_MULEQ = r"\*="  
t_DIVEQ = r"/="  
t_MASMAS = r"\+\+"  
t_MENOSMENOS = r"-"  
  
def t_BOOL(token):  
    r"true | false"  
  
def t_FLOAT(token):  
    r"[.]?[0-9]"  
  
def t_INT(token):  
    r"[1-9][0-9]* — 0"  
  
def t_error(token):  
    message = "Token desconocido:"  
  
def t_STRING(token):  
    r"' ' " . * ? " ' ' ' '  
    atributos = {}  
    atributos["type"] = "string"  
    atributos["value"] = token.value  
    token.value = atributos  
    return token  
  
def t_ID(token):  
    r"[a-zA-Z_][a-zA-Z_0-9]*"  
    token.type = reserved.get(token.value, 'ID')  
  
def t_NEWLINE(token):  
    r"\n+"  
    token.lexer.lineno += len(token.value)  
  
message += "\n type:" + token.type  
message += "\n value:" + str(token.value)
```

```

message += "\n line:" + str(token.lineno)      def t_COMMENT(token):
message += "\n position:" + str(token.lexpos)    r'#.*'
raise Exception(message)                       t_ignore = ' \ t'

```

3.2 Implementacion Del Parser

La implementación del parser consistió en transcribir la gramática final del apartado 2 a la sintaxis de ply.

De esta manera, dada una producción:

$$Valores \rightarrow ExpresionMatematica$$

Fue necesario reescribirla como:

```

def p_valores(subexpressions) :
    """ valores : ExpresionMatematica """

```

Ademas contamos con la funcionalidad de ply que, una vez que se a utlizado una producción permite ejecutar codigo adicional. Siguiendo el ejemplo anterior, suponiendo que cada vez que el parser utiliza la producciòn antedicha deseo imprimirla, puedo escribir:

```

def p_valores(subexpressions) :
    """ valores : ExpresionMatematica """
    print subexpressions[1]

```

Esto lo utilizamos tanto para escribir el output formateado con la salida correcta como para realizar el chequeo de tipos.

3.3 Implementacion Del Chequeo de Tipos

4 Conclusiones

5 Conclusiones