



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

La clausura a LR(1) en 80 ítems

Teoría de Lenguajes
Primer Cuatrimestre 2016

Integrante	LU	Correo electrónico
Confalonieri, Gisela Belén	511/11	gise_5291@yahoo.com.ar
Mignanelli, Alejandro Rubén	609/11	minga_titere@hotmail.com
Suárez, Federico	610/11	elgeniofederico@gmail.com



Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://www.fcen.uba.ar>

Índice

1. Introducción	3
2. La gramática	3
2.1. Tokens	3
2.2. Gramática	3
2.2.1. Reglas	3
2.2.2. Problemas	7
3. La Solución	8
3.1. Atributos	8
3.2. Manejo de variables	9
3.3. Problemas y decisiones	9
3.3.1. Vectores de registros	9
3.3.2. Decisiones	9
4. Ejecución	10
4.1. Requerimientos	10
4.2. Cómo correr	10
5. Casos de Prueba	10
5.1. Casos Sintácticamente Correctos	10
5.2. Casos Sintácticamente Incorrectos	11
6. Conclusiones	12
7. Apendice 1: Código	13

1. Introducción

En el presente trabajo, se pretende desarrollar un analizador léxico y sintáctico para un lenguaje de scripting, denominado Simple Lenguaje de Scripting (SLS). El mismo, recibirá un código fuente como entrada, y deberá chequear si cumple la sintaxis y restricciones de tipado del lenguaje, para luego formatear el código con la 'indentación' adecuada para SLS. En caso de haberse detectado algún error, se informará claramente cuáles son las características del mismo. Las características del SLS se encuentran en el enunciado de este trabajo práctico.

2. La gramática

En esta sección se muestra cómo se realizó la tokenización de las expresiones válidas en este lenguaje, la gramática implementada y algunas aclaraciones sobre los conflictos con los que nos encontramos en la medida que la creábamos.

2.1. Tokens

En este pequeño lenguaje de scripting, existen palabras reservadas que no pueden usarse como nombres de variables, en cualquier combinación de minúsculas y mayúsculas: `begin`, `end`, `while`, `for`, `if`, `else`, `do`, `res`, `return`, `true`, `false`, `AND`, `OR`, `NOT`. El proceso de tokenización tiene en cuenta este aspecto, y en caso de encontrar una palabra reservada, se asigna al token correspondiente o se arroja un error en caso de no existir un token asociado.

El Cuadro 1 describe, para cada token definido, el símbolo que representa y la expresión regular asociada, con el formato aceptado por Python.

2.2. Gramática

A continuación se muestran las reglas de la gramática implementada. La misma es por lo menos LALR, dado la herramienta PLY pudo generar esta tabla sin conflictos a partir de las reglas declaradas.

2.2.1. Reglas

```
# _____ #
def p_inicial(expr):
    'start_:_codigo'
# _____ #
# _____ #
def p_constante_valor(cte):
    ''' constante_:_STR
    _____|_BOOL
    _____|_numero
    _____|_LPAREN_constante_RPAREN'''
# _____ #
def p_constante_funcion(f):
    'constante_:_funcion'
# _____ #
# _____ #
def p_variable(expr):
    ''' variable_:_VAR
    _____|_RES
    _____|_variable_LCORCH_z_RCORCH
    _____|_LPAREN_variable_RPAREN
    _____|_VAR_PUNTO_VAR'''
# _____ #
# _____ #
def p_numero(num):
    ''' numero_:_NUM
    _____|_NUM_PUNTO_NUM
    _____|_MAS_NUM
    _____|_MAS_NUM_PUNTO_NUM
    _____|_MENOS_NUM
    _____|_MENOS_NUM_PUNTO_NUM'''
# _____ #
# _____ #
```

TOKEN	Símbolo representado	Expr Regular
STR	cadena de caracteres entre comillas dobles	"[^\"]*"
BOOL	true o false	(true false)
NUM	cualquier cadena numérica	[0-9]+
VAR	cadena alfanumérica con '_' que comienza en una letra	[a-zA-Z][a-zA-Z0-9_]*
PUNTO	'.'	.
DOSPTOS	':'	:
COMA	','	,
ADM	'!'	!
PREG	'?'	?
PTOCOMA	';'	;
LCORCH	'['	[
RCORCH	']']
LPAREN	'('	(
RPAREN	')')
LLAVE	'{'	{
RLLAVE	'}'	}
MAS	'+'	+
MENOS	'-'	-
IGUAL	'='	=
POR	'*'	*
DIV	'/'	/
POT	'^'	^
MOD	'%'	%
MAYOR	'>'	>
MENOR	'<'	<
COMENT	'#' y cualquier cadena de caracteres, hasta el primer salto de línea	#. *
WHILE	'while'	while
FOR	'for'	for
IF	'if'	if
ELSE	'else'	else
DO	'do'	do
RES	'res'	res
AND	'AND'	AND
OR	'OR'	OR
NOT	'NOT'	NOT
PRINT	'print'	print
MULTESC	'multiplicacionEscalar'	multiplicacionEscalar
CAP	'capitalizar'	capitalizar
COLIN	'colineales'	colineales
LENGTH	'length'	length

Cuadro 1: Tokens de la gramática

```

def p_granzeta(expr):
    '''granz::z
    =====|_ternario'''
    #
def p_zeta(expr):
    '''z::zso
    =====|_operacion'''
    #
def p_zeta_sin_oper(expr):
    '''zso::_variable

```

```

.....|_constante
.....|_vector
.....|_registro '''
# .....#
def p_ge(expr):
    '''g: _variable
.....|_constante
.....|_relacion
.....|_logico '''
# .....#
# .....#
def p_vector(expr):
    '''vector: _LCORCH _z _separavec _RCORCH
.....|_LPAREN _vector _RPAREN '''
# .....#
def p_separavector(expr):
    '''separavec: _empty
.....|_COMA _z _separavec '''
# .....#
# .....#
def p_registro(expr):
    '''registro: _LLLAVE _RLlave
.....|_LLLAVE _VAR _DOSPTOS _z _separareg _RLlave
.....|_LPAREN _registro _RPAREN '''
# .....#
def p_separaregistro(expr):
    '''separareg: _empty
.....|_COMA _VAR _DOSPTOS _z _separareg '''
# .....#
# .....#
def p_asignacion(expr):
    '''asignacion: _variable _operasig _z
.....|_variable _operasig _ternario '''
# .....#
def p_operasig(op):
    '''operasig: _IGUAL
.....|_MAS _IGUAL
.....|_MENOS _IGUAL
.....|_POR _IGUAL
.....|_DIV _IGUAL '''
# .....#
# .....#
def p_matematico(expr):
    '''matematico: _matprim _operMatBinario _matf
.....|_LPAREN _matematico _RPAREN '''
# .....#
def p_matprim(expr):
    '''matprim: _matprim _operMatBinario _matf
.....|_matf '''
# .....#
def p_matf(expr):
    '''matf: _zso
.....|_LPAREN _matematico _RPAREN '''
# .....#
def p_operMatBinario(op):
    '''operMatBinario: _MAS
.....|_MENOS
.....|_POR
.....|_POT
.....|_MOD
.....|_DIV '''
# .....#
# .....#
def p_autoincdec(expr):
# .....#
def p_operMatUnario(op):
    '''operMatUnario: _MAS _MAS
.....|_MENOS _MENOS '''
# .....#
# .....#
def p_relacion(expr):
    '''relacion: _relprim _operRelacion _relf
.....|_LPAREN _relacion _RPAREN '''
# .....#
def p_relprim(expr):
    '''relprim: _relprim _operRelacion _relf
.....|_relf '''
# .....#
def p_relf(expr):
    '''relf: _zso

```

```

#####|_matematico
#####|_LPAREN_relacion_RPAREN
#####|_LPAREN_logico_RPAREN
#####|_NOT_zso
#####|_NOT_LPAREN_operacion_RPAREN'''
#
def p_operRelacion(op):
    '''operRelacion: _IGUAL_IGUAL
#####|_ADM_IGUAL
#####|_MAYOR
#####|_MENOR'''
#
#
def p_logico(expr):
    '''logico: _logprim_operLogicoBinario_logf
#####|_LPAREN_logico_RPAREN
#####|_NOT_zso
#####|_NOT_LPAREN_operacion_RPAREN'''
#
def p_logprim(expr):
    '''logprim: _logprim_operLogicoBinario_logf
#####|_logf'''
#
def p_logf(expr):
    '''logf: _zso
#####|_relacion
#####|_LPAREN_logico_RPAREN
#####|_NOT_LPAREN_operacion_RPAREN
#####|_NOT_zso'''
#
def p_operLogBinario(op):
    '''operLogicoBinario: _AND
#####|_OR'''
#
#
def p_ternario(expr):
    '''ternario: _g_PREG_granz_DOSPTOS_granz
#####|_LPAREN_ternario_RPAREN'''
#
#
def p_operacion(expr):
    '''operacion: _matematico
#####|_relacion
#####|_logico'''
#
#
def p_sentencia_(expr):
    '''sentencia: _asignacion_PTOCOMA
#####|_PRINT_z_PTOCOMA
#####|_autoincdec_PTOCOMA'''
#
#
def p_funcion_multesc(expr):
    '''funcion: _MULTESC_LPAREN_z_COMA_z_RPAREN
#####|_MULTESC_LPAREN_z_COMA_z_COMA_z_RPAREN'''
#
def p_funcion_cap(expr):
    'funcion: _CAP_LPAREN_z_RPAREN'
#
def p_funcion_colin(expr):
    'funcion: _COLIN_LPAREN_z_COMA_z_RPAREN'
#
def p_funcion_length(expr):
    'funcion: _LENGTH_LPAREN_z_RPAREN'
#
#
def p_stmt(expr):
    '''stmt: _closedstmt
#####|_openstmt'''
#
def p_closedstmt(expr):
    '''closedstmt: _sentencia
#####|_LLLAVE_codigo_RLLAVE
#####|_dowhile
#####|_IF_LPAREN_g_RPAREN_closedstmt_ELSE_closedstmt
#####|_loopheader_closedstmt
#####|_comentario_closedstmt'''
#
def p_openstmt(expr):
    '''openstmt: _IF_LPAREN_g_RPAREN_stmt

```

```
.....| _IF _LPAREN _g _RPAREN _closedstmt _ELSE _openstmt
.....| _loopheader _openstmt
.....| _comentario _openstmt '''
#-----#
def p_bucle(expr):
    '''loopheader: _for
.....| _while '''
#-----#
def p_for_sin_sig(expr):
    '''for: _FOR _LPAREN _PTOCOMA _g _PTOCOMA _RPAREN
.....| _FOR _LPAREN _PTOCOMA _g _PTOCOMA _asignacion _RPAREN
.....| _FOR _LPAREN _PTOCOMA _g _PTOCOMA _autoincdec _RPAREN '''
#-----#
def p_for_con_sig(expr):
    '''for: _FOR _LPAREN _asignacion _PTOCOMA _g _PTOCOMA _RPAREN
.....| _FOR _LPAREN _asignacion _PTOCOMA _g _PTOCOMA _asignacion _RPAREN
.....| _FOR _LPAREN _asignacion _PTOCOMA _g _PTOCOMA _autoincdec _RPAREN '''
#-----#
def p_while(expr):
    'while: _WHILE _LPAREN _g _RPAREN'
#-----#
def p_dowhile(expr):
    'dowhile: _DO _stmt _WHILE _LPAREN _g _RPAREN _PTOCOMA'
#-----#
def p_codigo(expr):
    '''codigo: _stmt _codigo
.....| _stmt
.....| _comentariosfinales '''
#-----#
def p_comentario(expr):
    'comentario: _COMENT'
#-----#
def p_comentariosfinales(expr):
    '''comentariosfinales: _comentario
.....| _comentario _comentariosfinales '''
#-----#
def p_empty(p):
    'empty: '
    pass
#-----#
#-----#
```

2.2.2. Problemas

A continuación presentaremos los mayores problemas con los que nos encontramos al generar la gramática apropiada.

Operaciones Inicialmente, las reglas para las operaciones binarias (matemáticas, lógicas y relacionales) tenían la siguiente forma:

operacion : z operador z (el z de la gramática explicitada)

Claramente, esto generaba muchas ambigüedades, puesto que, por ejemplo $3 + 3 + 3$ tenía más de una forma de producirse.

Para resolver este problema se consideró una asociatividad a izquierda para las operaciones del mismo tipo, y la precedencia de las operaciones matemáticas sobre las relacionales, y de éstas sobre las lógicas. No consideramos la precedencia entre los operadores matemáticos entre sí, lo mismo que entre los lógicos y entre los relacionales.

Dangling Else Dado este clásico problema, se recurrió a implementar la solución brindada en 'el libro del Dragón', pero con algunas modificaciones.

Al implementarlo directamente, obtuvimos conflictos con los ciclos 'for' y 'while'. Por ejemplo:

```
if guarda1
    while guarda2
        if guarda3
            bloque1
        else
            bloque2
```

El problema surgía cuando se daba como input un código donde los bloques poseían una sola sentencia y podían prescindir de las llaves. En el ejemplo descripto, dada la primera implementación, no podía saberse si el 'else' pertenecía al condicional más grande o al que se encuentra dentro del ciclo.

Esto se producía porque, en una primera aproximación a la gramática, las reglas de estos ciclos tenían la siguiente forma:

```
for : FOR (declaracion) bloque y while : WHILE (guarda) bloque
```

Para solucionar el problema, se decidió quitar el 'bloque' de ambas reglas y manejarlo desde las reglas que definen los bloques de código.

Comentarios dentro de un condicional La gramática actual no acepta una entrada que cuente con un comentario dentro de un condicional, antes de la palabra `else`. Por ejemplo:

```
if guarda
    codigo
    #comentario
else
    codigo
```

3. La Solución

En el Apéndice 1 se encuentra el código fuente del parser.

3.1. Atributos

Los no terminales de la gramática fueron asociados a clases creadas en Python, cada una con ciertos atributos necesarios para el análisis de tipado y el formateo del código.

- Las variables se asocian a la clase `Variable`, que contiene los siguientes atributos:
 - *nombre* Contiene un string con el nombre de la variable.
 - *impr* Contiene un string con el formato imprimible de la variable.
 - *campo* Contiene un string con el nombre del campo, en caso de que se trate de una variable del tipo `registro.campo`.
 - *array_elem* Contiene un entero (mayor o igual a 0) que indica el nivel de anidamiento de índices en variables del tipo `variable[indice]`. Por ejemplo, la variable `a` tendrá este atributo en 0, la variable `b[0]` tendrá este atributo en 1 y la variable `c[2][4][6]` tendrá este atributo en 3.
- Los números, las funciones, operaciones y vectores se asocian a subclases de la clase `Constante`, que contiene los siguientes atributos:
 - *tipo* Contiene un string con el tipo del elemento.
 - *impr* Contiene un string con el formato imprimible del elemento.
- Los registros se asocian a la clase `Registro`, que contiene los siguientes atributos:
 - *tipo* Es constantemente 'registro'.
 - *impr* Contiene un string con el formato imprimible del registro.
- Las sentencias, bloques de código, ciclos, condicionales y comentarios se asocian a la clase `Código`, que contiene los siguientes atributos:
 - *llaves* Contiene un entero (0 o 1) que indica si se trata de un bloque encerrado entre llaves o no.
 - *impr* Contiene un string con el formato imprimible del registro.

3.2. Manejo de variables

Para el correcto chequeo de tipos de las variables, como el tipo de las mismas puede variar, se decidió implementar un diccionario global {variable : tipo}. Las claves del diccionario son los nombres de variable, y los valores son strings con el tipo correspondiente.

Un caso especial son los registros. Para ello, aprovechando la versatilidad en cuanto a tipado de Python, el valor asociado a una variable de tipo registro es un nuevo diccionario {campo : tipo}, cuyas claves son los campos del registro, y sus valores son los tipos correspondientes.

Por otro lado, si se trata de una variable de tipo vector, el tipo que se coloca en el diccionario es vectortipo, donde tipo es el tipo de los elementos del vector. Esto es importante al chequear, por ejemplo, que si se declara un vector de enteros, no se pueda asignar a una posición del mismo una cadena.

Al cambiar el tipo de una variable, o del campo de un registro, simplemente se reemplaza el valor en el diccionario.

3.3. Problemas y decisiones

3.3.1. Vectores de registros

En el caso de declararse un vector de registros, no existe al momento un control que permita consultar un campo de un registro de dicho vector. Por ejemplo, el siguiente código falla:

```
usuario1 = {nombre:"Al", edad:50};
usuario2 = {nombre:"Mr.X", edad:10};
usuarios = [usuario1, usuario2];

s = usuarios[1].edad;
```

3.3.2. Decisiones

Se han tomado las siguientes decisiones respecto al código que debe aceptarse:

- **Indexar vectores:** Para acceder a una posición de un vector, el mismo debe haber sido asignado previamente a una variable, para luego pedir el índice correspondiente. Por lo tanto, un código como el siguiente falla:

```
hola = [1,2,3,4,5,6][4];
```

- **Utilización de registros:** Los registros definidos como campos entre llaves deben asignarse a una variable. No se acepta utilizar un registro o acceder a sus campos si no es a través de una variable.
- **Operador ternario:** Este operador sólo tiene sentido si se lo asigna directamente a una variable. Por ejemplo, el siguiente código falla:

```
a = 3 + (true ? 3 : 4);
```

- **Funciones:** Excepto la función print, ninguna otra función puede utilizarse como sentencia, es decir, sin usar su resultado en una asignación o una operación, dado que ninguna función modifica los datos de entrada. Por lo tanto, un código como el que sigue falla:

```
length("pepe");
```

- **Números negativos:** Para que un número sea considerado un negativo, debe colocarse el signo (-) inmediatamente antes del número en cuestión. Si el número está entre paréntesis, el signo será considerado como operación de resta. De la misma manera, una variable de tipo numérico será negativa sólo si se le asigna un valor negativo. Por ejemplo, el siguiente código falla:

```
a = -(7);
x = 3;
b = -x;
```

- **Autoincremento y autodecremento:** Los operadores (++) y (--) sólo aplica a variables, y de tipo entero. Es decir, no se pueden utilizar sobre números o sobre variables de tipo float.

4. Ejecución

4.1. Requerimientos

Para ejecutar el parser se requiere contar con el siguiente software:

- Python 2.7 (o superior)
- Python-PLY 3.6 (o superior)

Las pruebas se realizaron sobre Ubuntu 14.04.

4.2. Cómo correr

Se provee el código fuente de este parser, el cual consta de los siguientes archivos:

- expressions.py
- lexer_rules.py
- parser_rules.py
- parser.py
- SLSParser.sh

Para correr el programa, es necesario que todos los archivos fuente se encuentren en el mismo directorio.

El programa deberá recibir el código fuente a procesar, y retornará el código resultante. Se puede especificar un nombre de archivo de entrada, y en caso de que no se especifique, se esperará recibir la cadena a procesar por standard input (stdin). En caso de no especificar un archivo de salida, el resultado se imprimirá por standard output (stdout). En caso de que hubiera algún problema en la llamada, se terminará el programa y se mostrarán los detalles por standard error (stderr).

Se ejecuta desde consola de la siguiente manera:

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

Las opciones son las siguientes:

- **-o SALIDA** Se especifica un archivo de salida para el código formateado
- **-c ENTRADA** Se especifica el nombre del archivo de entrada con el código fuente. Si el archivo no se encuentra en el mismo directorio que el código fuente, se deberá colocar su ruta absoluta.
- **FUENTE** Cadena con el código fuente.

5. Casos de Prueba

En esta sección se mostrará un subconjunto de las pruebas que se realizaron para probar el correcto funcionamiento del parser. En caso de que la expresión sintáctica sea correcta, se mostrará lo que el parser devuelve, y en caso de no serlo se explicará el por qué.

5.1. Casos Sintácticamente Correctos

Prueba1:

```
a = true;#algo \n b=5; #algo \nfor (;a;b++) #algo \nif( a ) #algo \n {b=a; #algo \n if (b) b=a;#algo \n} else c = 8;#algo \n
```

Resultado:

```
a = True;
#algo
b = 5;
#algo
```

```
for( ; a; b++)
  #algo
  if(a)
    #algo
    {
      b = a;
      #algo
      if(b)
        b = a;
      #algo }
  else
    c = 8;
#algo
```

Prueba2:

```
a=3; a++; a--; a += 6; a -= 5; a *= 3; print (length( \"pelado\"));
b= (a==8); if (b) a= a; c = a==8? a:a;
```

Resultado:

```
a = 3;
a++;
a--;
a += 6;
a -= 5;
a *= 3;
print (length("pelado"));
b = (a == 8);
if(b)
    a = a;
c = a == 8 ? a : a;
```

Prueba3:

```
a[3+5] = 2*3; a[length(a)] = 2; a[2] = a [10%3]; b = [3, length(a), a[5]]; b[4] = length(b);
```

Resultado:

```
a[3 + 5] = 2 * 3;
a[length(a)] = 2;
a[2] = a[10 % 3];
b = [3, length(a), a[5]];
b[4] = length(b);
```

5.2. Casos Sintácticamente Incorrectos

Prueba1:

```
a = [2,3,4]; b = [a]; c = length(b); b[2] = [\"pepe\"];
```

La razón por la cual esta expresión es sintácticamente incorrecta es que *b* es un vector de vectores de enteros, por lo cual no puede recibir en ningún subíndice un elemento de tipo cadena. Notar que si *b* no estuviera inicializado, esa asignación se podría hacer, puesto que estaríamos inicializando *b* en un vector de cadenas.

Prueba2:

```
a = {campo : 3}; a.campo = \"pepe\"; b[a.campo] = 2;
```

En este caso, si bien en un comienzo *a.campo* era un entero, la segunda asignación cambia su tipo a cadena, por lo cual no puede ser índice del arreglo *b*.

Prueba3:

```
campo = [{campo:\"1\"}, {campo:\"2\"}, {campo:\"3\"}][1];
```

Para referenciar una posición de un vector, se debe acceder a él a través de una variable, por lo cual esta expresión será considerada sintacticamente incorrecta.

6. Conclusiones

Consideramos este trabajo un interesante cierre para la materia, puesto que nos permitió trabajar con una gramática amplia, que presenta mayores desafíos que solucionar un concepto en una gramática con 3 no terminales.

A nivel sintáctico, se presentó un gran desafío para evitar que la gramática sea ambigua, como por ejemplo el problema del dangling else, que no fue solucionado solamente con la respuesta de un libro, ya que la amplitud de la gramática causó conflictos en otras regiones, y requirió astucia y perseverancia hasta acomodar la gramática de manera tal que esta no tenga ambigüedades.

A nivel semántico, se tuvieron que tomar varias decisiones que debían congeniar y formar una estructura coherente, lo cual en una gramática de esta amplitud no fue trivial.

En conclusión, fue un trabajo muy interesante y con más desafíos de los esperados.

7. Apendice 1: Código

```
1 from lexer_rules import tokens
2
3 from expressions import *
4
5 # diccionario de variables, nombre:tipo
6 variables = {}
7
8 #-----#
9
10 def p_inicial(expr):
11     'start::codigo'
12
13     expr[0] = expr[1]
14
15 #-----#
16 #-----#
17
18 def p_constante_valor(cte):
19     '''constante::STR
20     ::BOOL
21     ::numero
22     ::LPAREN_constante_RPAREN'''
23
24     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
25
26     if cte[1] == '(':
27         cte[0] = Constante(cte[2].tipo)
28     else:
29         if type(cte[1]) == str:
30             cte[0] = Constante('str')
31         elif type(cte[1]) == bool:
32             cte[0] = Constante('bool')
33         else: # es un numero
34             cte[0] = Constante(cte[1].tipo)
35
36     ##### FORMATO PARA IMPRIMIR #####
37
38     if cte[1] == '(':
39         cte[0].impr = '(' + cte[2].impr + ')'
40     else:
41         if type(cte[1]) == str:
42             cte[0].impr = cte[1]
43         elif type(cte[1]) == bool:
44             cte[0].impr = str(cte[1]).lower()
45         else: # es un numero
46             cte[0].impr = cte[1].impr
47
48 #-----#
49
50 def p_constante_funcion(f):
51     'constante::funcion'
52
53     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
54
55     f[0] = Funcion(f[1].tipo)
56
57     ##### FORMATO PARA IMPRIMIR #####
58
59     f[0].impr = f[1].impr
60
61 #-----#
62 #-----#
63
64 def p_variable(expr):
65     '''variable::VAR
66     ::RES
67     ::variable_LCORCH_z_RCORCH
68     ::LPAREN_variable_RPAREN
69     ::VAR_PUNTO_VAR'''
70
71     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
72
73     global variables
74
75     if expr[1] == '(': # var -> (var)
76         expr[0] = expr[2]
77
```

```

78     elif len(expr) == 2: # var -> VAR|RES
79         if expr[1] in ['res', 'reS', 'rEs', 'rES', 'Res', 'ReS', 'REs', 'RES']:
80             expr[0] = Variable('res')
81         else:
82             expr[0] = Variable(expr[1]) #nombre
83
84     elif len(expr) == 5: # var -> var[NUM]
85         if tipo_segun(expr[3]) != 'int':
86             error_semantico(expr, 3, "El índice del vector debe ser entero")
87
88         expr[0] = expr[1]
89         expr[0].array_elem += 1
90
91     else: # var -> REGISTRO.CAMPO
92         if expr[1] in ['res', 'reS', 'rEs', 'rES', 'Res', 'ReS', 'REs', 'RES']: # el campo no puede ser res
93             error_semantico(expr, 1, "El campo no puede ser una palabra reservada")
94
95         expr[0] = Variable(expr[1])
96         expr[0].nombre_campo(expr[3])
97
98     ##### FORMATO PARA IMPRIMIR #####
99
100    if expr[1] == '(':
101        expr[0].impr = '(' + expr[2].impr + ')'
102    elif len(expr) == 5:
103        expr[0].impr = expr[1].impr + '[' + expr[3].impr + ']'
104    else:
105        expr[0].impr = expr[0].nombre
106        if len(expr) == 4:
107            expr[0].impr += '.' + expr[3]
108
109    #-----#
110    #-----#
111
112    def p_numero(num):
113        '''numero::NUM
114        |NUM.PUNTO.NUM
115        |MAS.NUM
116        |MAS.NUM.PUNTO.NUM
117        |MENOS.NUM
118        |MENOS.NUM.PUNTO.NUM'''
119
120    ##### CHEQUEO Y ASIGNACION DE TIPOS #####
121
122    if len(num) <= 2 :
123        num[0] = Numero('int')
124    else:
125        num[0] = Numero('float')
126
127    ##### FORMATO PARA IMPRIMIR #####
128
129    if len(num) == 2:
130        num[0].impr = str(num[1])
131    elif len(num) == 3:
132        num[0].impr = num[1] + str(num[2])
133    elif len(num) == 4:
134        num[0].impr = str(num[1])+'.'+str(num[3])
135    else:
136        num[0].impr = num[1]+str(num[2])+'.'+str(num[4])
137
138    #-----#
139
140    def p_granzeta(expr):
141        '''granz::z
142        |ternario'''
143
144        expr[0] = expr[1]
145
146    #-----#
147
148    def p_zeta(expr):
149        '''z::zso
150        |operacion'''
151
152        expr[0] = expr[1]
153
154    #-----#
155
156    def p_zeta_sin_oper(expr):
157        '''zso::variable

```

```
158     _constante
159     _vector
160     _registro '''
161
162     expr[0] = expr[1]
163
164     #-----#
165
166     def p_ge(expr):
167         '''g: _variable
168         _constante
169         _relacion
170         _logico '''
171
172         expr[0] = expr[1]
173
174         #-----#
175         #-----#
176
177     def p_vector(expr):
178         '''vector: _LCORCH _z_separavec _RCORCH
179         _LPAREN _vector _RPAREN'''
180
181         #### CHEQUEO Y ASIGNACION DE TIPOS ####
182
183         if expr[1] == '(': # vec -> (vec)
184             expr[0] = expr[2]
185         elif (tipo_según(expr[2]) != tipo_según(expr[3])) & (tipo_según(expr[3]) != 'vacío') :
186             error_semántico(expr,2,"El vector debe contener elementos del mismo tipo")
187         else:
188             expr[0] = Vector('vector'+tipo_según(expr[2]))
189
190         #### FORMATO PARA IMPRIMIR ####
191
192         if expr[1] == '(':
193             expr[0].impr = '(' + expr[2].impr + ')'
194         else:
195             expr[0].impr = '[' + expr[2].impr + expr[3].impr + ']'
196
197         #-----#
198
199     def p_separavector(expr):
200         '''separavec: _empty
201         _COMA _z_separavec'''
202
203         #### CHEQUEO Y ASIGNACION DE TIPOS ####
204
205         if len(expr) == 2:
206             expr[0] = Vector('vacío')
207         elif (tipo_según(expr[2]) != tipo_según(expr[3])) & (tipo_según(expr[3]) != 'vacío') :
208             error_semántico(expr,2,"El vector debe contener elementos del mismo tipo")
209         else:
210             expr[0] = Vector(tipo_según(expr[2]))
211
212         #### FORMATO PARA IMPRIMIR ####
213
214         if len(expr) > 2:
215             expr[0].impr = ',' + expr[2].impr + expr[3].impr
216
217         #-----#
218         #-----#
219
220     def p_registro(expr):
221         '''registro: _LLLAVE _RLLAVE
222         _LLLAVE _VAR _DOSPTOS _z_separareg _RLLAVE
223         _LPAREN _registro _RPAREN'''
224
225         #### CHEQUEO Y ASIGNACION DE TIPOS ####
226
227         if expr[1] == '(': # reg -> (reg)
228             expr[0] = expr[2]
229         elif len(expr) == 3 : # reg -> {}
230             expr[0] = Registro([],[])
231         else:
232             expr[0] = Registro([expr[2]]+expr[5].campos, [tipo_según(expr[4])+expr[5].tipos.campos])
233
234         #### FORMATO PARA IMPRIMIR ####
235
236         if expr[1] == '(':
237             expr[0].impr = '(' + expr[2].impr + ')'
```

```

238     elif len(expr) == 3 :
239         expr[0].impr = '{} '
240     else:
241         expr[0].impr = '{' + expr[2] + ':_' + expr[4].impr + expr[5].impr + '}'
242
243     #####
244
245 def p_separaregistro(expr):
246     '''separareg: empty
247     =====> COMA.VAR.DOSP.TOS. separareg'''
248
249     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
250
251     if len(expr) == 2 :
252         expr[0] = Registro([],[])
253     else:
254         expr[0] = Registro([expr[2]]+expr[5].campos, [tipo_segun(expr[4])+expr[5].tipos_campos])
255
256     ##### FORMATO PARA IMPRIMIR #####
257
258     if len(expr) > 2 :
259         expr[0].impr = ',_' + expr[2] + ':_' + expr[4].impr + expr[5].impr
260
261     #####
262     #####
263
264 def p_asignacion(expr):
265     '''asignacion: variable_operasig
266     =====> variable_operasig_ternario'''
267
268     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
269
270     global variables
271
272     if type(expr[3]) == Variable: # si aparece una variable del lado derecho de la asignacion
273         if not(expr[3].nombre in variables):
274             error_semantico(expr,3,"Operacion con variable no inicializada")
275
276         elif (expr[3].campo != 'None') & (type(variables[expr[3].nombre]) != dict):
277             error_semantico(expr,3,"Accede a campo de variable que no es registro")
278
279         elif expr[3].array_elem >= 1:
280             if (expr[3].campo != 'None'):
281                 if variables[expr[3].nombre][expr[3].campo][6] != 'vector':
282                     error_semantico(expr,3,"Accede a indice de variable que no es vector")
283             elif variables[expr[3].nombre][6] != 'vector':
284                 error_semantico(expr,3,"Accede a indice de variable que no es vector")
285
286     tipoZ = tipo_segun(expr[3])
287
288     if expr[2] == '=': # asigno una variable -> inicializo o piso su tipo
289
290         if (expr[1].nombre in variables): # si la variable ya se uso antes
291             if expr[1].campo != 'None': # registro.campo = bla
292                 if type(variables[expr[1].nombre]) != dict: # esta cambiando de tipo a registro
293                     variables[expr[1].nombre] = {}
294                 variables[expr[1].nombre][expr[1].campo] = tipoZ
295
296             elif expr[1].array_elem >= 1: # var[num] = bla
297                 if type(variables[expr[1].nombre]) == dict:
298                     variables[expr[1].nombre] = 'vector' + tipoZ
299                 elif variables[expr[1].nombre][6] != 'vector':
300                     variables[expr[1].nombre] = 'vector' + tipoZ
301                 elif (variables[expr[1].nombre][6] == 'vector') & (variables[expr[1].nombre][6:] != tipoZ):
302                     # veo que matchee el tipo de ahora
303                     error_semantico(expr,1,"No respeta el tipo del vector")
304
305         else: # es una variable cualquiera, no var[num] ni reg.campo
306
307             if tipoZ == 'registro': # reflejo los campos
308                 variables[expr[1].nombre] = campos_a_dic(expr[3])
309             elif tipoZ == 'vreg': # reflejo los campos de la variable q es registro
310                 variables[expr[1].nombre] = variables[expr[3].nombre]
311             else:
312                 variables[expr[1].nombre] = tipoZ
313
314         else: # declaro la variable
315
316             if tipoZ == 'registro': # reflejo los campos
317                 variables[expr[1].nombre] = campos_a_dic(expr[3])

```



```

317
318     elif tipoZ == 'vreg': # reflejo los campos de la variable q es registro
319         variables[expr[1].nombre] = variables[expr[3].nombre]
320
321     elif expr[1].array_elem >= 1: # declaro un vector a traves de uno de sus elementos
322         variables[expr[1].nombre] = 'vector' + tipoZ
323
324     else:
325         variables[expr[1].nombre] = tipoZ
326
327 else: # aca la variable ya deberia estar inicializada
328     if not(expr[1].nombre in variables):
329         error_semantico(expr,1,"Variable no inicializada")
330     else:
331         #tipoV = variables[expr[1].nombre]
332         tipoV = tipo_segund(expr[1])
333
334         if expr[2] == '+': # es numerico o cadena
335             if not((numericos(tipoV,tipoZ)) | ((tipoV == tipoZ) & (tipoZ == 'str'))):
336                 error_semantico(expr,2,"El tipo debe ser numerico o cadena")
337             else: # es numerico
338                 if not(numericos(tipoV,tipoZ)):
339                     error_semantico(expr,2,"El tipo debe ser numerico")
340
341 ##### FORMATO PARA IMPRIMIR #####
342
343 imprimir = expr[1].impr + ' ' + expr[2] + ' ' + expr[3].impr
344 expr[0] = Codigo(imprimir)
345
346 #-----#
347
348 def p_operasig(op):
349     '''operasig: IGUAL
350     ===== MAS_IGUAL
351     ===== MENOS_IGUAL
352     ===== POR_IGUAL
353     ===== DIV_IGUAL'''
354
355     op[0] = op[1]
356     if len(op) == 3:
357         op[0] += op[2]
358
359 #-----#
360 #-----#
361
362 def p_matematico(expr):
363     '''matematico: matprim operMatBinario matf
364     ===== LPAREN matematico RPAREN'''
365
366     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
367
368     if expr[1] == '(': # mat -> (mat)
369         expr[0] = expr[2]
370
371     else:
372         tipo1 = tipo_segund(expr[1])
373         tipo2 = tipo_segund(expr[3])
374
375         if expr[2] == '+':
376             if not(numericos(tipo1,tipo2)) | ((tipo1 == 'str') & (tipo2 == 'str')):
377                 error_semantico(expr,2,"Tipos incompatibles")
378
379         elif expr[2] == '%':
380             if (tipo1 != 'int') & (tipo2 != 'int'):
381                 error_semantico(expr,2,"El tipo debe ser entero")
382
383         elif not(numericos(tipo1,tipo2)):
384             error_semantico(expr,2,"El tipo debe ser numerico")
385
386         if (tipo1 == 'str'):
387             expr[0] = Operacion('str')
388         elif (tipo1 == 'float') | (tipo2 == 'float'):
389             expr[0] = Operacion('float')
390         else:
391             expr[0] = Operacion('int')
392
393 ##### FORMATO PARA IMPRIMIR #####
394
395     if expr[1] == '(': # mat -> (mat)
396         expr[0].impr = '(' + expr[2].impr + ')'

```

```
397     else:
398         expr[0].impr = expr[1].impr + '⌋' + expr[2] + '⌋' + expr[3].impr
399
400     #-----#
401
402 def p_matprim(expr):
403     '''matprim⌒:⌒matprim⌒operMatBinario⌒matf
404     ⌒-----⌒matf'''
405
406     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
407
408     if len(expr) == 2:
409         expr[0] = expr[1]
410
411     else:
412         tipo1 = tipo_según(expr[1])
413         tipo2 = tipo_según(expr[3])
414
415         if expr[2] == '+':
416             if not(numericos(tipo1,tipo2)) | ((tipo1 == 'str') & (tipo2 == 'str')):
417                 error_semántico(expr,2,"Tipos⌒incompatibles")
418
419         elif expr[2] == '%':
420             if (tipo1 != 'int') & (tipo2 != 'int'):
421                 error_semántico(expr,2,"El⌒tipo⌒debe⌒ser⌒entero")
422
423         elif not(numericos(tipo1,tipo2)):
424             error_semántico(expr,2,"El⌒tipo⌒debe⌒ser⌒numérico")
425
426         if (tipo1 == 'str'):
427             expr[0] = Operacion('str')
428         elif (tipo1 == 'float') | (tipo2 == 'float'):
429             expr[0] = Operacion('float')
430         else:
431             expr[0] = Operacion('int')
432
433     ##### FORMATO PARA IMPRIMIR #####
434
435     if len(expr) > 2:
436         expr[0].impr = expr[1].impr + '⌋' + expr[2] + '⌋' + expr[3].impr
437
438     #-----#
439
440 def p_matf(expr):
441     '''matf⌒:⌒zso
442     ⌒-----⌒LPAREN⌒matemático⌒RPAREN'''
443
444     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
445
446     if len(expr) == 2:
447         expr[0] = expr[1]
448
449     else:
450         expr[0] = expr[2]
451
452     ##### FORMATO PARA IMPRIMIR #####
453
454     if len(expr) > 2:
455         expr[0].impr = '(' + expr[2].impr + ')'
456
457     #-----#
458
459 def p_operMatBinario(op):
460     '''operMatBinario⌒:⌒MAS
461     ⌒-----⌒MENOS
462     ⌒-----⌒POR
463     ⌒-----⌒POT
464     ⌒-----⌒MOD
465     ⌒-----⌒DIV'''
466
467     op[0] = op[1]
468
469     #-----#
470     #-----#
471
472 def p_autoincdec(expr):
473     '''autoincdec⌒:⌒operMatUnario⌒variable
474     ⌒-----⌒variable⌒operMatUnario'''
475
476     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
```

```

477
478     if (expr[1] == '++') | (expr[1] == '--') :
479         if variables[expr[2].nombre] != 'int':
480             error_semantico(expr,2,"El tipo debe ser entero")
481     elif variables[expr[1].nombre] != 'int':
482         error_semantico(expr,1,"El tipo debe ser entero")
483
484     ##### FORMATO PARA IMPRIMIR #####
485
486     if (expr[1] == '++') | (expr[1] == '--') :
487         imprimir = expr[1] + expr[2].impr
488     else:
489         imprimir = expr[1].impr + expr[2]
490
491     expr[0] = Codigo(imprimir)
492
493     #-----#
494
495     def p_operMatUnario(op):
496         '''operMatUnario: _MAS_MAS
497         | _MENOS_MENOS'''
498
499         op[0] = op[1]
500         op[0] += op[2]
501
502     #-----#
503     #-----#
504
505     def p_relacion(expr):
506         '''relacion: _relprim_operRelacion_rel
507         | _LPAREN_relacion_RPAREN'''
508
509         ##### CHEQUEO Y ASIGNACION DE TIPOS #####
510
511         if expr[1] == '(': # rel -> (rel)
512             expr[0] = expr[2]
513
514         else:
515             tipoZ1 = tipo_segun(expr[1])
516             tipoZ2 = tipo_segun(expr[3])
517
518             if len(expr[2]) == 1: # para < y > solo numericos
519                 if not(numericos(tipoZ1,tipoZ2)):
520                     error_semantico(expr,2,"El tipo debe ser numerico")
521             elif not(numericos(tipoZ1,tipoZ2) | (tipoZ1 == tipoZ2)) : # para == y != vale cualquier tipo siempre
522                 y cuando sean los dos el mismo
523                 error_semantico(expr,2,"Los tipos deben coincidir")
524
525             expr[0] = Operacion('bool')
526
527         ##### FORMATO PARA IMPRIMIR #####
528
529         if expr[1] == '(':
530             expr[0].impr = '(' + expr[2].impr + ')'
531         else:
532             expr[0].impr = expr[1].impr + '_' + expr[2] + '_' + expr[3].impr
533
534     #-----#
535
536     def p_relprim(expr):
537         '''relprim: _relprim_operRelacion_rel
538         | _relf'''
539
540         ##### CHEQUEO Y ASIGNACION DE TIPOS #####
541
542         if len(expr) == 2:
543             expr[0] = expr[1]
544
545         else:
546             tipoZ1 = tipo_segun(expr[1])
547             tipoZ2 = tipo_segun(expr[3])
548
549             if len(expr[2]) == 1: # para < y > solo numericos
550                 if not(numericos(tipoZ1,tipoZ2)):
551                     error_semantico(expr,2,"El tipo debe ser numerico")
552             elif not(numericos(tipoZ1,tipoZ2) | (tipoZ1 == tipoZ2)) : # para == y != vale cualquier tipo siempre
553                 y cuando sean los dos el mismo
554                 error_semantico(expr,2,"Los tipos deben coincidir")
555
556             expr[0] = Operacion('bool')

```

```
555
556 ##### FORMATO PARA IMPRIMIR #####
557
558 if len(expr) > 2:
559     expr[0].impr = expr[1].impr + '┌' + expr[2] + '┐' + expr[3].impr
560
561 #-----#
562
563 def p_relf(expr):
564     '''relf:┌zso
565     |matematico
566     |┌PAREN┌relacion┌RPAREN
567     |┌PAREN┌logico┌RPAREN
568     |┌NOT┌zso
569     |┌NOT┌PAREN┌operacion┌RPAREN'''
570
571 ##### CHEQUEO Y ASIGNACION DE TIPOS #####
572
573 if len(expr) == 2:
574     expr[0] = expr[1]
575
576 else:
577     if len(expr) == 5:
578         if tipo_segun(expr[3]) != 'bool':
579             error_semantico(expr,3,"El tipo debe ser bool")
580         elif len(expr) == 3:
581             if tipo_segun(expr[1]) != 'bool':
582                 error_semantico(expr,2,"El tipo debe ser bool")
583
584         expr[0] = Operacion('bool')
585
586 ##### FORMATO PARA IMPRIMIR #####
587
588 if len(expr) == 4:
589     expr[0].impr = '(' + expr[2].impr + ')'
590 elif len(expr) == 5:
591     expr[0].impr = 'NOT(' + expr[3].impr + ')'
592 elif len(expr) == 3:
593     expr[0].impr = 'NOT┌' + expr[2].impr
594
595 #-----#
596
597 def p_operRelacion(op):
598     '''operRelacion:┌IGUAL┌IGUAL
599     |┌ADM┌IGUAL
600     |┌MAYOR
601     |┌MENOR'''
602
603     op[0] = op[1]
604     if len(op) == 3:
605         op[0] += op[2]
606
607 #-----#
608 #-----#
609
610 def p_logico(expr):
611     '''logico:┌logprim┌operLogicoBinario┌logf
612     |┌LPAREN┌logico┌RPAREN
613     |┌NOT┌zso
614     |┌NOT┌LPAREN┌operacion┌RPAREN'''
615
616 ##### CHEQUEO Y ASIGNACION DE TIPOS #####
617
618 if expr[1] == '(':
619     expr[0] = expr[2]
620 elif len(expr) == 3: # not
621     if tipo_segun(expr[2]) != 'bool':
622         error_semantico(expr,2,"El tipo debe ser bool")
623 elif len(expr) == 5: # not(bla)
624     if tipo_segun(expr[3]) != 'bool':
625         error_semantico(expr,3,"El tipo debe ser bool")
626
627 else:
628     tipo1 = tipo_segun(expr[1])
629     tipo2 = tipo_segun(expr[3])
630
631     if (tipo1 != 'bool') | (tipo2 != 'bool'):
632         error_semantico(expr,2,"El tipo debe ser bool")
633
634     expr[0] = Operacion('bool')
```

```
635 ##### FORMATO PARA IMPRIMIR #####
636
637 if expr[1] == '(':
638     expr[0].impr = '(' + expr[2].impr + ')'
639 elif len(expr) == 3: # not
640     expr[0].impr = 'NOT' + expr[2].impr
641 elif len(expr) == 5: # not(bla)
642     expr[0].impr = 'NOT(' + expr[3].impr + ')'
643 else:
644     expr[0].impr = expr[1].impr + '┌' + expr[2] + '┐' + expr[3].impr
645
646 #-----#
647
648 def p_logprim(expr):
649     '''logprim:┌logprim┐operLogicoBinario┌logf
650     ┌──────────┐┌logf'''
651
652     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
653
654     if len(expr) == 2:
655         expr[0] = expr[1]
656     else:
657         tipo1 = tipo_segun(expr[1])
658         tipo2 = tipo_segun(expr[3])
659
660         if (tipo1 != 'bool') | (tipo2 != 'bool'):
661             error_semantico(expr,2,"El tipo debe ser bool")
662
663         expr[0] = Operacion('bool')
664
665     ##### FORMATO PARA IMPRIMIR #####
666
667     if len(expr) > 2:
668         expr[0].impr = expr[1].impr + '┌' + expr[2] + '┐' + expr[3].impr
669
670 #-----#
671
672 def p_logf(expr):
673     '''logf:┌zso
674     ┌──────────┐┌relacion
675     ┌──────────┐┌LPAREN┌logico┌RPAREN
676     ┌──────────┐┌NOT┌LPAREN┌operacion┌RPAREN
677     ┌──────────┐┌NOT┌zso'''
678
679     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
680
681     if len(expr) == 2:
682         expr[0] = expr[1]
683     else:
684         if len(expr) == 5:
685             if tipo_segun(expr[3]) != 'bool':
686                 error_semantico(expr,3,"El tipo debe ser bool")
687             elif len(expr) == 4:
688                 if tipo_segun(expr[2]) != 'bool':
689                     error_semantico(expr,2,"El tipo debe ser bool")
690             elif len(expr) == 3:
691                 if tipo_segun(expr[2]) != 'bool':
692                     error_semantico(expr,2,"El tipo debe ser bool")
693
694         expr[0] = Operacion('bool')
695
696     ##### FORMATO PARA IMPRIMIR #####
697
698     if len(expr) == 4:
699         expr[0].impr = '(' + expr[2].impr + ')'
700     elif len(expr) == 5:
701         expr[0].impr = 'NOT(' + expr[3].impr + ')'
702     elif len(expr) == 3:
703         expr[0].impr = 'NOT' + expr[2].impr
704
705 #-----#
706
707
708 def p_operLogBinario(op):
709     '''operLogicoBinario:┌AND
710     ┌──────────┐┌OR'''
711
712     op[0] = op[1]
713
714 #-----#
```

```
715 #-----#
716
717 def p_ternario(expr):
718     '''ternario: _g_PREG_granz_DOSPTOS_granz
719     _LPAREN_ternario_RPAREN'''
720
721     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
722
723     if expr[1] == '(':
724         expr[0] = expr[2]
725     else:
726         tipoG = tipo_segun(expr[1])
727         tipoZ1 = tipo_segun(expr[3])
728         tipoZ2 = tipo_segun(expr[5])
729
730         if (tipoG != 'bool') | (tipoZ1 != tipoZ2):
731             error_semantico(expr,1,"La condicion debe ser booleana y las operaciones deben tener el mismo tipo")
732         else:
733             expr[0] = Operacion(tipoZ1)
734
735     ##### FORMATO PARA IMPRIMIR #####
736
737     if expr[1] == '(':
738         expr[0].impr = '(' + expr[2].impr + ')'
739     else:
740         expr[0].impr = expr[1].impr + '?_ ' + expr[3].impr + ':_ ' + expr[5].impr
741
742 #-----#
743 #-----#
744
745 def p_operacion(expr):
746     '''operacion: _matematico
747     _relacion
748     _logico'''
749
750     expr[0] = expr[1]
751
752 #-----#
753 #-----#
754
755 def p_sentencia_(expr):
756     '''sentencia: _asignacion_PTOCOMA
757     _PRINT_z_PTOCOMA
758     _autoindec_PTOCOMA'''
759
760     ##### FORMATO PARA IMPRIMIR #####
761
762     if len(expr) == 3:
763         imprimir = expr[1].impr + ';\n'
764     else:
765         imprimir = 'print_ ' + expr[2].impr + ';\n'
766
767     expr[0] =Codigo(imprimir)
768
769 #-----#
770 #-----#
771
772 def p_funcion_multesc(expr):
773     '''funcion: _MULTESC_LPAREN_z_COMA_z_RPAREN
774     _MULTESC_LPAREN_z_COMA_z_COMA_z_RPAREN'''
775
776     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
777
778     tipoZ1 = tipo_segun(expr[3])
779     tipoZ2 = tipo_segun(expr[5])
780     #tipoZ3 = tipo_segun(expr[7])
781
782     if (tipoZ1[6:] != 'vector') | (not(numericos(tipoZ1[6:], tipoZ2))):
783         error_semantico(expr,1,"multiplicacionEscalar(vector, numerico[, bool])")
784
785     if len(expr) == 9:
786         tipoZ3 = tipo_segun(expr[7])
787
788         if tipoZ3 != 'bool':
789             error_semantico(expr,1,"multiplicacionEscalar(vector, numerico[, bool])")
790
791     if (tipoZ1[6:] == 'float') | (tipoZ2 == 'float'):
792         expr[0] = Funcion('vectorfloat')
793     else:
```

```

794         expr[0] = Funcion('vectorint')
795
796     ##### FORMATO PARA IMPRIMIR #####
797
798     expr[0].impr = 'multiplicacionEscalar(' + expr[3].impr + ', ' + expr[5].impr
799
800     if len(expr) == 9:
801         expr[0].impr += ', ' + expr[7].impr + ')'
802     else:
803         expr[0].impr += ')'
804
805     #-----#
806
807 def p_funcion_cap(expr):
808     'funcion_ : _CAP_LPAREN _Z_RPAREN'
809
810     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
811
812     if tipo_según(expr[3]) != 'str':
813         error_semántico(expr, 1, "capitalizar(cadena)")
814
815     expr[0] = Funcion('str')
816
817     ##### FORMATO PARA IMPRIMIR #####
818
819     expr[0].impr = 'capitalizar(' + expr[3].impr + ')'
820
821     #-----#
822
823 def p_funcion_colin(expr):
824     'funcion_ : _COLIN_LPAREN _Z_COMA _Z_RPAREN'
825
826     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
827
828     tipoZ1 = tipo_según(expr[3])
829     tipoZ2 = tipo_según(expr[5])
830
831     if (tipoZ1[:6] != 'vector') | (tipoZ2[:6] != 'vector'): # vectores
832         error_semántico(expr, 1, "colineales(vectornumerico, vectornumerico)")
833     elif (tipoZ1[-3:] != 'int') & (tipoZ1[-5:] != 'float') & (tipoZ2[-3:] != 'int') & (tipoZ2[-5:] != 'float'): # numericos
834         error_semántico(expr, 1, "colineales(vectornumerico, vectornumerico)")
835
836     expr[0] = Funcion('bool')
837
838     ##### FORMATO PARA IMPRIMIR #####
839
840     expr[0].impr = 'colineales(' + expr[3].impr + ', ' + expr[5].impr + ')'
841
842     #-----#
843
844 def p_funcion_length(expr):
845     'funcion_ : _LENGTH_LPAREN _Z_RPAREN'
846
847     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
848
849     if (tipo_según(expr[3]) != 'str') & (tipo_según(expr[3])[:6] != 'vector'):
850         error_semántico(expr, 1, "length(cadena_o_vector)")
851
852     expr[0] = Funcion('int')
853
854     ##### FORMATO PARA IMPRIMIR #####
855
856     expr[0].impr = 'length(' + expr[3].impr + ')'
857
858     #-----#
859     #-----#
860
861 def p_stmt(expr):
862     '''stmt_ : _closedstmt
863     _openstmt'''
864
865     expr[0] = expr[1]
866
867     #-----#
868
869 def p_closedstmt(expr):
870     '''closedstmt_ : _sentencia
871     _LLAVE_codigo_RLLAVE
872     _dowhile'''

```

```

873 |IF_LPAREN_g_RPAREN_closedstmt_ELSE_closedstmt
874 |loopheader_closedstmt
875 |comentario_closedstmt'''
876
877 ##### CHEQUEO Y ASIGNACION DE TIPOS #####
878
879 if len(expr) > 4: # if (g) bla...
880     if tipo_segun(expr[3]) != 'bool':
881         error_semantico(expr,3,"La_guarda_debe_ser_booleana")
882
883 ##### FORMATO PARA IMPRIMIR #####
884
885 if len(expr) == 2:
886     imprimir = expr[1].impr
887 elif len(expr) == 3:
888     imprimir = expr[1].impr
889     if (expr[1].impr)[0] == '#':
890         imprimir += '\n' + expr[2].impr
891     else:
892         imprimir += tabular(expr[2])
893 elif len(expr) == 4:
894     imprimir = '{\n' + expr[2].impr + '}'
895 else:
896     imprimir = 'if(' + expr[3].impr + ')' + tabular(expr[5]) + 'else' + tabular(expr[7])
897
898 expr[0] = Codigo(imprimir)
899 if len(expr) == 4:
900     expr[0].llaves = 1
901
902 #-----#
903
904 def p_openstmt(expr):
905     '''openstmt: IF_LPAREN_g_RPAREN_stmt
906     |IF_LPAREN_g_RPAREN_closedstmt_ELSE_openstmt
907     |loopheader_openstmt
908     |comentario_openstmt'''
909
910     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
911
912     if len(expr) > 3:
913         if tipo_segun(expr[3]) != 'bool':
914             error_semantico(expr,3,"La_guarda_debe_ser_booleana")
915
916     ##### FORMATO PARA IMPRIMIR #####
917
918     if len(expr) == 3:
919         imprimir = expr[1].impr
920         if (expr[1].impr)[0] == '#':
921             imprimir += '\n' + expr[2].impr
922         else:
923             imprimir += tabular(expr[2])
924     else:
925         imprimir = 'if(' + expr[3].impr + ')' + tabular(expr[5])
926         if len(expr) > 6:
927             imprimir += 'else' + tabular(expr[7])
928
929     expr[0] = Codigo(imprimir)
930
931 #-----#
932
933 def p_bucle(expr):
934     '''loopheader: for
935     |while'''
936
937     expr[0] = expr[1]
938
939 #-----#
940
941 def p_for_sinasig(expr):
942     '''for: FOR_LPAREN_PTOCOMA_g_PTOCOMA_RPAREN
943     |FOR_LPAREN_PTOCOMA_g_PTOCOMA_asignacion_RPAREN
944     |FOR_LPAREN_PTOCOMA_g_PTOCOMA_autoincdec_RPAREN'''
945
946     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
947
948     global variables
949
950     if tipo_segun(expr[4]) != 'bool':
951         error_semantico(expr,4,"La_guarda_debe_ser_booleana")
952

```



```
953 ##### FORMATO PARA IMPRIMIR #####
954
955 imprimir = 'for(␣;␣' + expr[4].impr + '␣;␣'
956
957 if expr[6] == ')':
958     imprimir += ')'# + expr[7].impr
959 else:
960     imprimir += expr[6].impr + ')'# + expr[8].impr
961
962 expr[0] = Codigo(imprimir)
963
964 #-----#
965
966 def p_for_conasig(expr):
967     '''for␣:␣FOR_LPAREN␣asignacion␣PTOCOMA␣g␣PTOCOMA␣RPAREN
968     |␣FOR_LPAREN␣asignacion␣PTOCOMA␣g␣PTOCOMA␣asignacion␣RPAREN
969     |␣FOR_LPAREN␣asignacion␣PTOCOMA␣g␣PTOCOMA␣autoincdec␣RPAREN'''
970
971     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
972
973     global variables
974
975     if tipo_según(expr[5]) != 'bool':
976         error_semántico(expr, 5, "La␣guarda␣debe␣ser␣booleana")
977
978     ##### FORMATO PARA IMPRIMIR #####
979
980     imprimir = 'for(' + expr[3].impr + '␣;␣' + expr[5].impr + '␣;␣'
981
982     if expr[7] == ')':
983         imprimir += ')'# + expr[7].impr
984     else:
985         imprimir += expr[7].impr + ')'# + expr[9].impr
986
987     expr[0] = Codigo(imprimir)
988
989 #-----#
990
991 def p_while(expr):
992     'while␣:␣WHILE_LPAREN␣g␣RPAREN'
993
994     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
995
996     if tipo_según(expr[3]) != 'bool':
997         error_semántico(expr, 3, "La␣guarda␣debe␣ser␣booleana")
998
999     ##### FORMATO PARA IMPRIMIR #####
1000
1001     imprimir = 'while(' + expr[3].impr + ')'# + expr[5].impr
1002     expr[0] = Codigo(imprimir)
1003
1004 #-----#
1005
1006 def p_dowhile(expr):
1007     'dowhile␣:␣DO_stmt␣WHILE_LPAREN␣g␣RPAREN␣PTOCOMA'
1008
1009     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
1010
1011     if tipo_según(expr[5]) != 'bool':
1012         error_semántico(expr, 5, "La␣guarda␣debe␣ser␣booleana")
1013
1014     ##### FORMATO PARA IMPRIMIR #####
1015
1016     imprimir = 'do' + tabular(expr[2]) + 'while(' + expr[5].impr + ');␣\n'
1017     expr[0] = Codigo(imprimir)
1018
1019 #-----#
1020
1021 def p_codigo(expr):
1022     '''codigo␣:␣stmt␣codigo
1023     |␣stmt
1024     |␣comentariosfinales'''
1025
1026     ##### FORMATO PARA IMPRIMIR #####
1027
1028     imprimir = expr[1].impr
1029
1030     if len(expr) == 3:
1031         imprimir += expr[2].impr
1032
```

```

1033     expr[0] =Codigo(imprimir)
1034
1035 #-----#
1036
1037 def p_comentario(expr):
1038     'comentario_: _COMENT'
1039
1040     ##### FORMATO PARA IMPRIMIR #####
1041
1042     expr[0] =Codigo(expr[1])
1043
1044 #-----#
1045     NUEVOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
1046
1047 def p_comentariosfinales(expr):
1048     '''comentariosfinales_: _comentario
1049     .....|_comentario _comentariosfinales'''
1050
1051     ##### FORMATO PARA IMPRIMIR #####
1052
1053     imprimir = expr[1].impr
1054
1055     if len(expr) == 3:
1056         imprimir += '\n' + expr[2].impr + '\n'
1057
1058     expr[0] =Codigo(imprimir)
1059
1060 #-----#
1061
1062 def p_empty(p):
1063     'empty_: '
1064     pass
1065
1066 #-----#
1067
1068 def p_error(expr):
1069     #print expr
1070     message = "[Syntax_error]"
1071     message += "\nline:" + str(expr.lineno)
1072     message += "\nindex:" + str(expr.lexpos)
1073     raise Exception(message)
1074
1075 #-----#
1076
1077 def error_semantico(expr,n,msg):
1078     message = "[Semantic_error]"
1079     message += "\n"+msg
1080     message += "\nline:" + str(expr.lineno(n))
1081     message += "\nindex:" + str(expr.lexpos(n))
1082     raise Exception(message)
1083
1084 #-----#
1085 #-----#
1086
1087 def numericos(tipo1,tipo2):
1088
1089     return (tipo1 in ['int','float']) & (tipo2 in ['int','float'])
1090
1091 #-----#
1092
1093 def tipo_según(objeto):
1094
1095     global variables
1096
1097     if type(objeto) == Variable:
1098         if objeto.array_elem >= 1: # es una posicion de un arreglo
1099             if objeto.campo != 'None': # es algo tipo reg.campo
1100                 variable = variables[objeto.nombre][objeto.campo][6:]
1101             else:
1102                 variable = variables[objeto.nombre][6:]
1103             for x in range(1,objeto.array_elem):
1104                 variable = variable[6:]
1105
1106         elif objeto.campo != 'None': # es algo tipo reg.campo
1107             variable = (variables[objeto.nombre])[objeto.campo]
1108         elif type(variables[objeto.nombre]) == dict:
1109             variable = 'vreg'
1110         else:
1111             variable = variables[objeto.nombre]
```

```
1112     else:
1113         variable = objeto.tipo
1114
1115     return variable
1116
1117 #-----#
1118
1119 def campos_a_dic(reg):
1120
1121     dic = {}
1122
1123     for x in range(0, len(reg.campos)):
1124         dic[reg.campos[x]] = reg.tipos_campos[x]
1125
1126     return dic
1127
1128 #-----#
1129 #-----#
1130
1131 def tabular(codigo):
1132
1133     lineas = (codigo.impr).splitlines()
1134
1135     if codigo.llaves == 1:
1136         res = "{\n"
1137         for l in lineas[1:-1]:
1138             #if l[0] == '{':
1139                 #res += l + '\n'
1140             #elif l[0] == '}':
1141                 #res += l + '\n'
1142             #else:
1143                 res += '\t' + l + '\n'
1144         res += '}\n'
1145     else:
1146         res = "\n"
1147         for l in lineas:
1148             res += '\t' + l + '\n'
1149
1150     return res
```