



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. Primer Gramática	5
2.3. La Solución	11
3. Ejecución	12
3.1. Requerimientos	12
3.2. Cómo correr	12
4. Casos de Prueba	13
5. Resultados	14
6. Conclusiones	15
7. Apendice 1: Código	16

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, una primera gramática que acepta estas expresiones, y las transformaciones que se realizaron a la misma hasta llegar a la gramática implementada.

2.1. Tokens

En este pequeño lenguaje de scripting, existen palabras reservadas que no pueden usarse como nombres de variables, tanto en mayúsculas como en minúsculas, y el proceso de tokenización tiene en cuenta este aspecto.

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

Nota: CCMM significa 'Cualquier Combinación de mayúsculas y minúsculas'

TOKEN	Símbolo representado	Expr Regular
STR	cadena de caracteres entre comillas dobles	"[^\"]*"
BOOL	true o false	(([tT][rR][uU][eE] [fF][aA][lL][sS][eE])
NUM	cualquier cadena numérica	[0-9]+
VAR	cadena alfanumérica con '_' que comienza en una letra	[a-zA-Z][a-zA-Z0-9_]
PUNTO	'.'	.
DOSPUNTOS	':'	:
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	#. *
BEGIN	'begin' CCMM	[bB][eE][gG][iI][nN]
END	'end' CCMM	[eE][nN][dD]
WHILE	'while' CCMM	[wW][hH][iI][lL][eE]
FOR	'for' CCMM	[fF][oO][rR]
IF	'if' CCMM	[iI][fF]
ELSE	'else' CCMM	[eE][lL][sS][eE]
DO	'do' CCMM	[dD][oO]
RES	'res' CCMM	[rR][eE][sS]
RETURN	'return' CCMM	[rR][eE][tT][uU][rR][nN]
AND	'and' CCMM	[aA][nN][dD]
OR	'or' CCMM	[oO][rR]
NOT	'not' CCMM	[nN][oO][tT]
PRINT	'print' CCMM	[pP][rR][iI][nN][tT]
MULTESC	'multiplicacionescalar' CCMM	[mM][uU][lL][tT][iI][pP][lL][iI][cC][aA][cC][iI][oO][nN][eE][sS][cC][aA][lL][aA][rR]
CAP	'capitalizar' CCMM	[cC][aA][Pp][iI][tT][aA][lL][iI][zZ][aA][rR]
COLIN	'colineales' CCMM	[cC][oO][lL][iI][nN][eE][aA][lL][eE][sS]
LENGTH	'length' CCMM	[lL][eE][nN][gG][tT][hH]

Cuadro 1: Tokens de la gramática

2.2. Primer Gramatica

```
#-----#
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
    | VAR 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'''
#-----#
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):
'''autoincdec : operMatUnario variable
| variable operMatUnario'''

#-----#

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'''

#-----#

def p_operRelacion(op):
'''operRelacion : IGUAL IGUAL
| ADM IGUAL
| MAYOR
| MENOR'''
```

```
#-----#

def p_logico(expr):
'''logico : logprim operLogicoBinario logf
  | LPAREN logico RPAREN
  | NOT z'''

#-----#

def p_logprim(expr):
'''logprim : logprim operLogicoBinario logf
  | logf'''

#-----#

def p_logf(expr):
'''logf : zso
  | relacion
  | LPAREN logico RPAREN'''

#-----#

def p_operLogBinario(op):
'''operLogicoBinario : AND
  | OR'''

#-----#

def p_ternario(expr):
'''ternario : g PREG z DOSPTOS z
  | g PREG ternario DOSPTOS ternario'''

#-----#

def p_operacion(expr):
'''operacion : matematico
  | relacion
  | logico'''

#-----#

def p_sentencia_(expr):
'''sentencia : asignacion PTOCOMA
  | PRINT z PTOCOMA
  | autoindec 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_sinasig(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_conasig(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
| comentario'''

#-----#

def p_comentario(expr):
'comentario : COMMENT'

#-----#

def p_empty(p):
'empty :'
pass

#-----#

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

#-----#
```

2.3. La Solución

3. Ejecución

3.1. 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.

3.2. 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.

4. Casos de Prueba

5. Resultados

6. Conclusiones

7. Apéndice 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])
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     ::VAR_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_según(expr[3]) != 'int':
86             error_semántico(expr, 3, "El índice del vector debe ser entero")
87
88         expr[0] = Variable(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_semántico(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    else:
103        expr[0].impr = expr[0].nombre
104        if len(expr) == 4:
105            expr[0].impr += '.' + expr[3]
106        elif len(expr) == 5:
107            expr[0].impr += '[' + expr[3].impr + ']'
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
141    def p_zeta(expr):
142        '''z::zso
143        _____|_operacion'''
144
145        expr[0] = expr[1]
146
147    # _____ #
148
149    def p_zeta_sin_oper(expr):
150        '''zso::variable
151        _____|_constante
152        _____|_vector
153        _____|_registro'''
154
155        expr[0] = expr[1]
156
157    # _____ #

```

```
158
159 def p_ge(expr):
160     '''ge: variable
161     =====| constante
162     =====| relacion
163     =====| logico'''
164
165     expr[0] = expr[1]
166
167     #-----#
168     #-----#
169
170 def p_vector(expr):
171     '''vector: LCORCH z separavec RCORCH
172     =====| LPAREN vector RPAREN'''
173
174     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
175
176     if expr[1] == '(': # vec -> (vec)
177         expr[0] = expr[2]
178     elif (tipo_segun(expr[2]) != tipo_segun(expr[3])) & (tipo_segun(expr[3]) != 'vacio'):
179         error_semantico(expr, 2, "El vector debe contener elementos del mismo tipo")
180     else:
181         expr[0] = Vector('vector'+tipo_segun(expr[2]))
182
183     ##### FORMATO PARA IMPRIMIR #####
184
185     if expr[1] == '(':
186         expr[0].impr = '(' + expr[2].impr + ')'
187     else:
188         expr[0].impr = '[' + expr[2].impr + expr[3].impr + ']'
189
190     #-----#
191
192 def p_separavector(expr):
193     '''separavec: empty
194     =====| COMA z separavec'''
195
196     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
197
198     if len(expr) == 2:
199         expr[0] = Vector('vacio')
200     elif (tipo_segun(expr[2]) != tipo_segun(expr[3])) & (tipo_segun(expr[3]) != 'vacio') :
201         error_semantico(expr, 2, "El vector debe contener elementos del mismo tipo")
202     else:
203         expr[0] = Vector(tipo_segun(expr[2]))
204
205     ##### FORMATO PARA IMPRIMIR #####
206
207     if len(expr) > 2:
208         expr[0].impr = ', ' + expr[2].impr + expr[3].impr
209
210     #-----#
211     #-----#
212
213 def p_registro(expr):
214     '''registro: LLLAVE RLLAVE
215     =====| LLLAVE VAR DOSPTOS z separareg RLLAVE
216     =====| LPAREN registro RPAREN'''
217
218     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
219
220     if expr[1] == '(': # reg -> (reg)
221         expr[0] = expr[2]
222     elif len(expr) == 3 : # reg -> {}
223         expr[0] = Registro([],[])
224     else:
225         expr[0] = Registro( [expr[2]]+expr[5].campos , [tipo_segun(expr[4]))+expr[5].tipos_campos )
226
227     ##### FORMATO PARA IMPRIMIR #####
228
229     if expr[1] == '(':
230         expr[0].impr = '(' + expr[2].impr + ')'
231     elif len(expr) == 3 :
232         expr[0].impr = '{} '
233     else:
234         expr[0].impr = '{' + expr[2] + ': ' + expr[4].impr + expr[5].impr + '}'
235
236     #-----#
237
```

```

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

```

```

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

```

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

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

```
555
556     if len(expr[2]) == 1: # para < y > solo numericos
557         if not(numericos(tipoZ1,tipoZ2)):
558             error_semantico(expr,2,"El tipo debe ser numerico")
559     elif not(numericos(tipoZ1,tipoZ2) | (tipoZ1 == tipoZ2)) : # para == y != vale cualquier tipo siempre
560         y cuando sean los dos el mismo
561         error_semantico(expr,2,"Los tipos deben coincidir")
562
563     expr[0] = Operacion('bool')
564
565     ##### FORMATO PARA IMPRIMIR #####
566
567     if len(expr) > 2:
568         expr[0].impr = expr[1].impr + '⌋' + expr[2] + '⌋' + expr[3].impr
569
570 #-----#
571
572 def p_operRelacion(op):
573     '''operRelacion: _IGUAL _IGUAL
574     | _ADM _IGUAL
575     | _MAYOR
576     | _MENOR'''
577
578     op[0] = op[1]
579     if len(op) == 3:
580         op[0] += op[2]
581
582 #-----#
583
584 def p_logico(expr):
585     '''logico: _logprim _operLogicoBinario _logf
586     | _LPAREN _logico _RPAREN
587     | _NOT _z'''
588
589     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
590
591     if expr[1] == '(':
592         expr[0] = expr[2]
593     elif len(expr) == 3: # not
594         if tipo_segund(expr[2]) != 'bool':
595             error_semantico(expr,2,"El tipo debe ser bool")
596     else:
597         tipo1 = tipo_segund(expr[1])
598         tipo2 = tipo_segund(expr[3])
599
600         if (tipo1 != 'bool') | (tipo2 != 'bool'):
601             error_semantico(expr,2,"El tipo debe ser bool")
602
603     expr[0] = Operacion('bool')
604
605     ##### FORMATO PARA IMPRIMIR #####
606
607     if expr[1] == '(':
608         expr[0].impr = '(' + expr[2].impr + ')'
609     elif len(expr) == 3: # not
610         expr[0].impr = 'NOT⌋' + expr[2].impr
611     else:
612         expr[0].impr = expr[1].impr + '⌋' + expr[2] + '⌋' + expr[3].impr
613
614 #-----#
615
616 def p_logprim(expr):
617     '''logprim: _logprim _operLogicoBinario _logf
618     | _logf'''
619
620     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
621
622     if len(expr) == 2:
623         expr[0] = expr[1]
624     else:
625         tipo1 = tipo_segund(expr[1])
626         tipo2 = tipo_segund(expr[3])
627
628         if (tipo1 != 'bool') | (tipo2 != 'bool'):
629             error_semantico(expr,2,"El tipo debe ser bool")
630
631     expr[0] = Operacion('bool')
632
633     ##### FORMATO PARA IMPRIMIR #####
```

```
634
635     if len(expr) > 2:
636         expr[0].impr = expr[1].impr + '⋮' + expr[2] + '⋮' + expr[3].impr
637
638     #-----#
639
640 def p_logf(expr):
641     '''logf: zso
642     -----| relacion
643     -----| LPAREN_logico RPAREN'''
644
645     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
646
647     if expr[1] == '(':
648         expr[0] = expr[2]
649     else:
650         expr[0] = expr[1]
651
652     expr[0] = Operacion('bool')
653
654     ##### FORMATO PARA IMPRIMIR #####
655
656     if expr[1] == '(':
657         expr[0].impr = '(' + expr[2].impr + ')'
658
659     #-----#
660
661 def p_operLogBinario(op):
662     '''operLogicoBinario: AND
663     -----| OR'''
664
665     op[0] = op[1]
666
667     #-----#
668     #-----#
669
670 def p_ternario(expr):
671     '''ternario: g_PREG_z_DOSPTOS_z
672     -----| g_PREG_ternario_DOSPTOS_ternario'''
673
674     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
675
676     tipoG = tipo_segun(expr[1])
677     tipoZ1 = tipo_segun(expr[3])
678     tipoZ2 = tipo_segun(expr[5])
679
680     if (tipoG != 'bool') | (tipoZ1 != tipoZ2):
681         error_semantico(expr, 1, "La condicion debe ser booleana y las operaciones deben tener el mismo tipo")
682     else:
683         expr[0] = Operacion(tipoZ1)
684
685     ##### FORMATO PARA IMPRIMIR #####
686
687     expr[0].impr = expr[1].impr + '¿?' + expr[3].impr + '⋮' + expr[5].impr
688
689     #-----#
690     #-----#
691
692 def p_operacion(expr):
693     '''operacion: matematico
694     -----| relacion
695     -----| logico'''
696
697     expr[0] = expr[1]
698
699     #-----#
700     #-----#
701
702 def p_sentencia_(expr):
703     '''sentencia: asignacion_PTOCOMA
704     -----| PRINT_z_PTOCOMA
705     -----| autoindec_PTOCOMA'''
706
707     ##### FORMATO PARA IMPRIMIR #####
708
709     if len(expr) == 3:
710         imprimir = expr[1].impr + ';\n'
711     else:
712         imprimir = 'print_ ' + expr[2].impr + ';\n'
713
```



```

714     expr[0] = Codigo(imprimir)
715
716     #-----#
717     #-----#
718
719     def p_funcion_multesc(expr):
720         '''funcion_: _MULTESC_LPAREN_z_COMA_z_RPAREN
721         | _MULTESC_LPAREN_z_z_COMA_z_COMA_z_RPAREN'''
722
723         ##### CHEQUEO Y ASIGNACION DE TIPOS #####
724
725         tipoZ1 = tipo_segun(expr[3])
726         tipoZ2 = tipo_segun(expr[5])
727         tipoZ3 = tipo_segun(expr[7])
728
729         if (tipoZ1[:6] != 'vector') | (not(numericos(tipoZ1[6:], tipoZ2))):
730             error_semantico(expr, 1, "multiplicacionEscalar(vector, numerico[, bool])")
731
732         if len(expr) == 9:
733             if tipoZ3 != 'bool':
734                 error_semantico(expr, 1, "multiplicacionEscalar(vector, numerico[, bool])")
735
736         if (tipoZ1[6:] == 'float') | (tipoZ2 == 'float'):
737             expr[0] = Funcion('vectorfloat')
738         else:
739             expr[0] = Funcion('vectorint')
740
741         ##### FORMATO PARA IMPRIMIR #####
742
743         expr[0].impr = 'multiplicacionEscalar(' + expr[3].impr + + ', ' + expr[5].impr
744
745         if len(expr) == 9:
746             expr[0].impr += ', ' + expr[7].impr + ')'
747         else:
748             expr[0].impr += ')'
749
750     #-----#
751
752     def p_funcion_cap(expr):
753         'funcion_: _CAP_LPAREN_z_RPAREN'
754
755         ##### CHEQUEO Y ASIGNACION DE TIPOS #####
756
757         if tipo_segun(expr[3]) != 'str':
758             error_semantico(expr, 1, "capitalizar(cadena)")
759
760         expr[0] = Funcion('str')
761
762         ##### FORMATO PARA IMPRIMIR #####
763
764         expr[0].impr = 'capitalizar(' + expr[3].impr + ')'
765
766     #-----#
767
768     def p_funcion_colin(expr):
769         'funcion_: _COLIN_LPAREN_z_COMA_z_RPAREN'
770
771         ##### CHEQUEO Y ASIGNACION DE TIPOS #####
772
773         tipoZ1 = tipo_segun(expr[3])
774         tipoZ2 = tipo_segun(expr[5])
775
776         if (tipoZ1[:6] != 'vector') | (tipoZ2[:6] != 'vector'): # vectores
777             error_semantico(expr, 1, "colineales(vectornumerico, vectornumerico)")
778         elif (tipoZ1[-3:] != 'int') & (tipoZ1[-5:] != 'float') & (tipoZ2[-3:] != 'int') & (tipoZ2[-5:] != 'float'): # numericos
779             error_semantico(expr, 1, "colineales(vectornumerico, vectornumerico)")
780
781         expr[0] = Funcion('bool')
782
783         ##### FORMATO PARA IMPRIMIR #####
784
785         expr[0].impr = 'colineales(' + expr[3].impr + ', ' + expr[5].impr + ')'
786
787     #-----#
788
789     def p_funcion_length(expr):
790         'funcion_: _LENGTH_LPAREN_z_RPAREN'
791
792         ##### CHEQUEO Y ASIGNACION DE TIPOS #####

```

```
793
794     if (tipo_según(expr[3]) != 'str') & (tipo_según(expr[3])[6] != 'vector'):
795         error_semántico(expr,1,"length(cadena_o_vector)")
796
797     expr[0] = Función('int')
798
799     ##### FORMATO PARA IMPRIMIR #####
800
801     expr[0].impr = 'length(' + expr[3].impr + ')',
802
803     #-----#
804     #-----#
805
806     def p_stmt(expr):
807         '''stmt: _closedstmt
808         _openstmt'''
809
810         expr[0] = expr[1]
811
812     #-----#
813
814     def p_closedstmt(expr):
815         '''closedstmt: _sentencia
816         _LLAVE_codigo_RLLAVE
817         _dowhile
818         _IF_LPAREN_g_RPAREN_closedstmt_ELSE_closedstmt
819         _loopheader_closedstmt
820         _comentario_closedstmt'''
821
822         ##### CHEQUEO Y ASIGNACION DE TIPOS #####
823
824         if len(expr) > 4: # if (g) bla...
825             if tipo_según(expr[3]) != 'bool':
826                 error_semántico(expr,3,"La_guarda_debe_ser_booleana")
827
828         ##### FORMATO PARA IMPRIMIR #####
829
830         if len(expr) == 2:
831             imprimir = expr[1].impr
832         elif len(expr) == 3:
833             imprimir = expr[1].impr
834             if (expr[1].impr)[0] == '#':
835                 imprimir += '\n' + expr[2].impr
836             else:
837                 imprimir += tabular(expr[2])
838         elif len(expr) == 4:
839             imprimir = '{\n' + expr[2].impr + '}'
840         else:
841             imprimir = 'if(' + expr[3].impr + ')' + tabular(expr[5]) + 'else' + tabular(expr[7])
842
843         expr[0] = Código(imprimir)
844         if len(expr) == 4:
845             expr[0].llaves = 1
846
847     #-----#
848
849     def p_openstmt(expr):
850         '''openstmt: _IF_LPAREN_g_RPAREN_stmt
851         _IF_LPAREN_g_RPAREN_closedstmt_ELSE_openstmt
852         _loopheader_openstmt
853         _comentario_openstmt'''
854
855         ##### CHEQUEO Y ASIGNACION DE TIPOS #####
856
857         if len(expr) > 3:
858             if tipo_según(expr[3]) != 'bool':
859                 error_semántico(expr,3,"La_guarda_debe_ser_booleana")
860
861         ##### FORMATO PARA IMPRIMIR #####
862
863         if len(expr) == 3:
864             imprimir = expr[1].impr + tabular(expr[2])
865         else:
866             imprimir = 'if(' + expr[3].impr + ')' + tabular(expr[5])
867             if len(expr) > 6:
868                 imprimir += 'else' + tabular(expr[7])
869
870         expr[0] = Código(imprimir)
871
872     #-----#
```

```
873
874 def p_bucle(expr):
875     '''loopheader_: _for
876     _while'''
877
878     expr[0] = expr[1]
879
880     #-----#
881
882 def p_for_sinasig(expr):
883     '''for_: _FOR_LPAREN_PTOCOMA_g_PTOCOMA_RPAREN
884     _FOR_LPAREN_PTOCOMA_g_PTOCOMA_asignacion_RPAREN
885     _FOR_LPAREN_PTOCOMA_g_PTOCOMA_autoincdec_RPAREN'''
886
887     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
888
889     global variables
890
891     if tipo_según(expr[4]) != 'bool':
892         error_semántico(expr,4,"La guarda debe ser booleana")
893
894     ##### FORMATO PARA IMPRIMIR #####
895
896     imprimir = 'for(_;_ + expr[4].impr + ';_'
897
898     if expr[6] == ')':
899         imprimir += ')'# + expr[7].impr
900     else:
901         imprimir += expr[6].impr + ')'# + expr[8].impr
902
903     expr[0] = Código(imprimir)
904
905     #-----#
906
907 def p_for_conasig(expr):
908     '''for_: _FOR_LPAREN_asignacion_PTOCOMA_g_PTOCOMA_RPAREN
909     _FOR_LPAREN_asignacion_PTOCOMA_g_PTOCOMA_asignacion_RPAREN
910     _FOR_LPAREN_asignacion_PTOCOMA_g_PTOCOMA_autoincdec_RPAREN'''
911
912     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
913
914     global variables
915
916     if tipo_según(expr[5]) != 'bool':
917         error_semántico(expr,5,"La guarda debe ser booleana")
918
919     ##### FORMATO PARA IMPRIMIR #####
920
921     imprimir = 'for(' + expr[3].impr + '_;_' + expr[5].impr + ';_'
922
923     if expr[7] == ')':
924         imprimir += ')'# + expr[7].impr
925     else:
926         imprimir += expr[7].impr + ')'# + expr[9].impr
927
928     expr[0] = Código(imprimir)
929
930     #-----#
931
932 def p_while(expr):
933     'while_: _WHILE_LPAREN_g_RPAREN'
934
935     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
936
937     if tipo_según(expr[3]) != 'bool':
938         error_semántico(expr,3,"La guarda debe ser booleana")
939
940     ##### FORMATO PARA IMPRIMIR #####
941
942     imprimir = 'while(' + expr[3].impr + ')'# + expr[5].impr
943     expr[0] = Código(imprimir)
944
945     #-----#
946
947 def p_dowhile(expr):
948     'dowhile_: _DO_stmt_WHILE_LPAREN_g_RPAREN_PTOCOMA'
949
950     ##### CHEQUEO Y ASIGNACION DE TIPOS #####
951
952     if tipo_según(expr[5]) != 'bool':
```

```
953         error_semantico(expr,5,"La_guarda_debe_ser_booleana")
954
955     ##### FORMATO PARA IMPRIMIR #####
956
957     imprimir = 'do' + tabular(expr[2]) + 'while(' + expr[5].impr + ');\\n'
958     expr[0] =Codigo(imprimir)
959
960     #-----#
961
962     def p_codigo(expr):
963         '''codigo_: _stmt _codigo
964         _stmt
965         _comentario'''
966
967         ##### FORMATO PARA IMPRIMIR #####
968
969         imprimir = expr[1].impr
970
971         if len(expr) == 3:
972             imprimir += expr[2].impr
973
974         expr[0] =Codigo(imprimir)
975
976     #-----#
977
978     def p_comentario(expr):
979         'comentario_: _COMENT'
980
981         ##### FORMATO PARA IMPRIMIR #####
982
983         expr[0] =Codigo(expr[1])
984
985     #-----#
986
987     def p_empty(p):
988         'empty_: '
989         pass
990
991     #-----#
992     #-----#
993
994     def p_error(expr):
995         message = "[Syntax_error]"
996         message += "\\nline:" + str(expr.lineno(0))
997         raise Exception(message)
998
999     #-----#
1000
1001     def error_semantico(expr,n,msg):
1002         message = "[Semantic_error]"
1003         message += "\\n"+msg
1004         message += "\\nline:" + str(expr.lineno(n))
1005         message += "\\nindex:" + str(expr.lexpos(n))
1006         raise Exception(message)
1007
1008     #-----#
1009     #-----#
1010
1011     def numericos(tipo1 ,tipo2):
1012
1013         return (tipo1 in ['int','float']) & (tipo2 in ['int','float'])
1014
1015     #-----#
1016
1017     def tipo_segun(objeto):
1018
1019         global variables
1020
1021         if type(objeto) == Variable:
1022             if objeto.array_elem == 1: # es una posicion de un arreglo
1023                 variable = variables[objeto.nombre][6:]
1024             elif objeto.campo != 'None': # es algo tipo reg.campo
1025                 variable = (variables[objeto.nombre])[objeto.campo]
1026             elif type(variables[objeto.nombre]) == dict:
1027                 variable = 'vreg'
1028             else:
1029                 variable = variables[objeto.nombre]
1030         else:
1031             variable = objeto.tipo
1032
```

```
1033     return variable
1034
1035 #-----#
1036
1037 def campos_a_dic(reg):
1038
1039     dic = {}
1040
1041     for x in range(0, len(reg.campos)):
1042         dic[reg.campos[x]] = reg.tipos_campos[x]
1043
1044     return dic
1045
1046 #-----#
1047 #-----#
1048
1049 def tabular(codigo):
1050
1051     lineas = (codigo.impr).splitlines()
1052
1053     if codigo.llaves == 1:
1054         res = ""
1055         for l in lineas:
1056             if l[0] == '{':
1057                 res += l + '\n'
1058             elif l[0] == '}':
1059                 res += l + '\n'
1060             else:
1061                 res += '\t' + l + '\n'
1062     else:
1063         res = "\n"
1064         for l in lineas:
1065             res += '\t' + l + '\n'
1066
1067     return res
```