Universidad de San Carlos de Guatemala

Facultad de Ingenieria

Ciencias y Sistemas

Organización de Lenguajes y Compiladores 1

Laboratorio Seccion N

Tutor Academico: José Puac

Manual Gramatica

JPR

Nombre: Jorge Isaac Xicol Vicente

Carnet: 201807316

La Gramatica utilizada para realizar el interprete del programa JPR es una gramatica ascendente con recursividad por la izquierda, como toda gramatica, esta contiene terminales, no terminales, expresiones regulares, inicio de la gramatica y varias producciones. Por lo que es necesario explicar y definir cada una de estas caracteristicas.

La gramatica realizada con la herramienta de python PLY usa como metodo de construccion el analisis LALR por lo que no es necesario quitar la recursividad por la izquierda.

Expresiones Regulares:

Para reconocer los tokens del analizador se necesito de expresiones regulares. Estas son usadas para reconocer los tokens complejos que no son representados solamente por un carácter o simbolo.

Las expresiones regulares usadas son las siguientes:

- **t_DECIMAL = '\d+.\d+**' expresion regular que evalua si el token es decimal, pudiendo venir cualquier digito una o mas veces, seguido de punto, seguido de digito una o mas veces.

- **t_ID = '[a-zA-Z_][a-zA-Z_0-9]*'** expresion que evalua si el token reconocido es un ID. Algo hay que aclarar de esto es que los tokens ID y palabras reservadas se evaluan con esta expresion misma, ya que se evalua si el ID reconocido es una palbara reservada de nuestro diccionario definido con anterioridad.

- **t_CADENA = r"""\"(\\"|\\'|\\\\|\\n|\\t|\\r|[^\\\'\"])*?\"""** expresion que evalua si el token es una cadena. En el lenguaje eta permitido colocar algo como lo siguiente: "\\" por lo que esta expresion evalua si solamente viene el simbolo "\" solo, entonces la cadena reconocida seria un error.

- **t_CARACTER = r""" \'(\\'|\\\\|\\n|\\t|\\r|\\"|.)?\"""** Expresion que evalua si el token es un carácter, igualmente que con la cadena se evalua si el simbolo "\" viene solo. Recalcando para las cadenas y caracteres, se hizo un reemplazo en el token para reemplazar los simbolos de escape del lenguaje JPR ya que en el texto del token un "\\n" se definiria como tal y no como un salto de linea.

- **t_COMMENTLINES = r'\#*(.|\n)*?*\#'** Expresion que evalua si el token es un comentario multilinea.

- **t_COMMENTUNLINE = r'\#.*\n'** Expresion que evalua si es un comentario unilinea.

- **Errores:** Para los errores lexicos se tuvo que definir una produccion **t_error** teniendo como retorno un objeto de clase Excepcion y agregandolo a la lista de errores.

Tambien hay que tomar en cuenta que se agrego por medio de un diccionario las palabras resevadas y se definieron los tokens con la sintaxis de PLY de Python. Estos se describen a continuacion.

Terminales:

- En la lista de reservadas se definieron las palabras reservadas del lenguaje JPR, este diccionario almacena una ‘llave’, ‘valor’, en donde con la llave se solicitaba el terminal en la gramatica y como valor de retorno se obtenian los valores:

```
reserved = {
 // Terminal    -  LLave //
   'var':          'RVAR',
   'new':          'RNEW',
   'print':        'RPRINT',
   'false':        'RFALSE',
   'true':         'RTRUE',
   'null':         'RNULL',
   'int':          'RINT',
   'double':       'RDOUBLE',
   'boolean':      'RBOOLEAN',
   'char':         'RCHAR',
   'string':       'RSTRING',
   'if':           'RIF',
   'else':         'RELSE',
   'switch':       'RSWITCH',
   'case':         'RCASE',
   'default':      'RDEFAULT',
   'while':        'RWHILE',
   'for':          'RFOR',
   'break':        'RBREAK',
   'continue':     'RCONTINUE',
   'return':       'RRETURN',
   'func':         'RFUNC',
   'read':         'RREAD',
   'main':         'RMAIN',
}
```

Esto se realizo de esta manera por que PLY de Python reconoce estos mismos tokens con la expresion regular ID definida anteriormente. Por lo que se verifica si el ID reconocido no es una reservada.

Tambien se definieron terminales para los simbolos como operadores aritmeticos, relacionales, corchetes, y parentesis, teniendo lo siguiente según la sintaxis de PLY de python.

```
tokens = [
    # Comentarios
    'COMMENTUNLINE',        # Comentario Unilinea
    'COMMENTLINES',         # Comentario multilinea

    # Caracteres Especiales
    'TWOPOINTS',            # Dos Puntos

    # Operadores Aritmeticos
    'PLUSSIGN',             # Signo Mas
    'SUBTRACTIONSIGN',      # Signo Menos
    'MULTIPLICATIONSIGN',   # Signo Multiplicacion
    'DIVISIONSIGN',         # Signo Division
    'POWERSIGN',            # Signo de potencia
    'MODULESIGN',           # Signo Modulo

    # Operadores Relacionales
    'EQUALIZATIONSIGN',     # Signo de Igualacion
    'DIFFERENTIATIONSIGN',  # Signo de diferenciacion
    'SMALLERTHAN',          # Signo de Menor que
    'GREATERTHAN',          # Signo de Mayor que
    'LESSEQUAL',            # Signo de Menor Igual
    'GREATEREQUAL',         # Signo de Mayor Igual

    # Operadores Logicos
    'OR',                   # Operador OR
    'AND',                  # Operador AND
    'NOT',                  # Operador NOT

    # Signos de Agrupacion
    'PARENTHESISOPEN',      # Parentesis Abre
    'PARENTHESISCLOSE',     # Parentesis Cierra
    'COMA',

    # Caracteres de Finalizacion y Encapsulamiento de Sentencias
```

```
    'SEMICOLON',            # Punto y Coma para la finalizacion (Puede o no venir)
    'KEYSIGNOPEN',          # Llave que Abre
    'KEYSIGNCLOSE',         # Llave que cierra

    # Declaracion y asignacion de Variables
    'EQUALSYMBOL',          # Simbolo Igual

    # Incremento y Decremento
    'INCREMENT',            # Incremento
    'DECREMENT',            # Decremento

    # Arreglos
    'CLASPSYMBOLOPEN',      # Corchete abre
    'CLASPSYMBOLCLOSE',     # Corchete cierra

    # Extras
    # Tipos de datos primitivos
    'ENTERO',               # Numero Entero
    'DECIMAL',              # Numero Decimal
    'CARACTER',             # caracter
    'CADENA',               # Cadena
    'ID',                   # Identificador (variables, nombres funciones, etc)
]
```

Token	Valor
t_TWOPOINTS	:
Operadores Aritmeticos	
t_PLUSSIGN	+
t_SUBTRACTIONSIGN	-
t_POWERSIGN	**
t_MULTIPLICATIONSIGN	*
t_DIVISIONSIGN	/
t_MODULESIGN	%
Operadores Relacionales	
t_EQUALIZATIONSIGN	==
t_DIFFERENTIATIONSIGN	=!
t_SMALLERTHAN	<
t_GREATERTHAN	>
t_LESSEQUAL	<=
t_GREATEREQUAL	>=
Operadores Logicos	
t_OR	||
t_AND	&&

t_NOT	!
Signos de Agrupacion	
t_PARENTHESISOPEN	(
t_PARENTHESISCLOSE	)
t_COMA	,
Caracteres de Finalizacion y Encapsulamiento de Sentencias	
t_SEMICOLON	;
t_KEYSIGNOPEN	{
t_KEYSIGNCLOSE	}
Declaracion y asignacion de Variables	
t_EQUALSYMBOL	=
Incremento y Decremento	
t_INCREMENT	++
t_DECREMENT	--
Arreglos	
t_CLASPSYMBOLOPEN	[
t_CLASPSYMBOLCLOSE	]

Inicio de Gramatica

A continuacion se describiran las producciones de la gramatica, pero debemos tomar en cuenta que esta contiene una produccion inicial, que nos retorna toda una lista de instrucciones. La produccion inicial es la siguiente:

La produccion "init" e "instrucciones" es una lista de instrucciones que al momento de terminar el analisis sera retornada.

```
def p_init(t):
    'init            : instrucciones'
    t[0] = t[1]


def p_instrucciones_instrucciones_instruccion(t):
    'instrucciones    : instrucciones instruccion'
    if t[2] != "":
        t[1].append(t[2])
    t[0] = t[1]
```

No Terminales:

Los no terminales son las producciones de nuestra gramatica. Se describen y se enlistan a continuacion.

- No terminal “instruccion”:
  Este no terminal produce todas las instrucciones de la gramatica, esto puede ser un print, declaracion de variables, asignacion de variables, un incremental, un if, while. Esta produccion nos ayuda a realizar todas las instrucciones de nuestro programa JPR. Sus producciones son explicadas posteriormente.

```
def p_instruccion(t):
    '''instruccion      : imprimir_instr finins
                        | declarar_instr finins
                        | asignar_instr finins
                        | unoenuno_instr finins
                        | if_instr
                        | while_instr
                        | switch_instr
                        | for_instr
                        | break_instr finins
                        | main_instr
                        | function_instr
                        | llamada_function finins
                        | return_instr finins
                        | continue_instr finins
                        | arreglo_declarar finins
                        | modificar_arreglo finins
    '''
```

- No terminal “finins”:
  En nuestro lenguaje JPR esta permitido colocar punto y coma al final de una instrucción, asi como tambien no colocarla. Esta produccion funciona para validar esto, ya que se coloca al final de cada instrucción. Podiendo producir el terminal punto y coma (;) o no producir nada.

```
def p_finins(t):
    '''finins           : SEMICOLON
                        | '''
```

- No terminal “error”:
  Este no terminal es reconocido por PLY, sirve para localizar un error y recuperarlo por medio de un punto y coma. En esta produccion se agregan los errores sintacticos encontrados en la gramatica.

```
def p_instruccion_error(t):
    'instruccion        : error SEMICOLON'
```

- No terminal “imprimir instr”:
  Esta produccion nos sirve para imprimir una expresion, teniendo comos ecuencia la palabra reservada print, parentesis que abre, No terminal expresion y parentesis que cierra.

```
def p_imprimir(t):
    'imprimir_instr      : RPRINT PARENTHESISOPEN expresion PARENTHESISCLOSE'
```

- No terminal “declarar_instr”, “declarar_nulo”, “declarar_expresion”:
  Produccion y no terminal con la cual se declara una variable, esta como puede ser declarada solamente como “var i” ene l lenguaje JPR se hace necesario realizar una produccion mas, una que acepte como “var i” y otra que acepte “var i = expresion”.

```
def p_declarar_instr(t):
    '''declarar_instr     : declarar_expresion
                          | declarar_nulo'''
def p_declarar_nulo(t):
    'declarar_nulo     : RVAR ID'

def p_declarar_expresion(t):
    'declarar_expresion   : RVAR ID EQUALSYMBOL expresion'
```

- No terminal “asignar_instr”:
  Esta produccion nos realiza una asignacion de variable como lo es “var i = expresion”.

```
def p_asignar_instr(t):
    '''asignar_instr     : ID EQUALSYMBOL expresion'''
```

- No terminal “unoenuno_instr”:
  Indica que se coloco una instrucción de tipo incremento o decremento. Por ejemplo “identificador++”, por lo que se necesita el terminal ID y el terminal INCREMENT o DECREMENT siendo un ++ o --.

```
def p_unoenuno_instr(t):
    '''unoenuno_instr    : ID INCREMENT
                         | ID DECREMENT'''
```

- No terminal “if_instr”:
  Esta instrucción tiene tres casos distintos, por lo que se necesitaron de tres producciones (no terminales) para realizarlo, en este caso
  1. Un if simple, en donde la secuencia sea la siguiente: “if (expresion) { instrucciones }”
  2. Un if con un else anidado donde: “if (expresion) {instrucciones} else { instrucciones }”
  3. Un if anidado con varios if: “if (expresion){instrucciones}else if_instr”

  Esto se hace asi para hacer una secuencia de if’s finita tanto como el ususario o requiera. Esto es posible al llamar a la misma produccion al final del terminal RELSE de la tercera produccion.

```
```

```
1. def p_if_simple(t):
        '''if_instr     : RIF PARENTHESISOPEN expresion PARENTHESISCLOSE
   KEYSIGNOPEN instrucciones KEYSIGNCLOSE'''

2. def p_if_else(t):
        '''if_instr     : RIF PARENTHESISOPEN expresion PARENTHESISCLOSE
   KEYSIGNOPEN instrucciones KEYSIGNCLOSE RELSE KEYSIGNOPEN instrucciones
   KEYSIGNCLOSE'''

3. def p_if_elseif(t):
        '''if_instr     : RIF PARENTHESISOPEN expresion PARENTHESISCLOSE
   KEYSIGNOPEN instrucciones KEYSIGNCLOSE RELSE if_instr'''
```

- No terminal “switch_instr”:
  Esta produccion es casi igual al if, solamente que la sintaxis es distinta, y en este caso se tendra una lista de “case” del switch mismo. Se valida lo siguiente:
  1. Switch(expresion){lista_case}
  2. Switch(expresion){switch_default}
  3. Switch(expresion){lista_case switch_default }

  Como se puede observar se valida cuando venga una lista de case sola y cuando venga un default solo, por ultimo el caso en donde ambos esten al mismo tiempo.

```
def p_switch_instr1(t):
    '''switch_instr  : RSWITCH PARENTHESISOPEN expresion PARENTHESISCLOSE
KEYSIGNOPEN switch_lista_case KEYSIGNCLOSE'''
    t[0] = Switch(t[3], t[6], None, t.lineno(1), find_column(input,
t.slice[1]))


def p_switch_instr2(t):
    '''switch_instr  : RSWITCH PARENTHESISOPEN expresion PARENTHESISCLOSE
KEYSIGNOPEN switch_default KEYSIGNCLOSE'''
    t[0] = Switch(t[3], None, t[6], t.lineno(1), find_column(input,
t.slice[1]))


def p_switch_instr3(t):
    '''switch_instr  : RSWITCH PARENTHESISOPEN expresion PARENTHESISCLOSE
KEYSIGNOPEN switch_lista_case switch_default KEYSIGNCLOSE'''
    t[0] = Switch(t[3], t[6], t[7], t.lineno(1), find_column(input,
t.slice[1]))
```

Esta produccion contiene otros no terminales para hacer la lista de case, en donde para tener un alista de case, realizamos el mismo no terminal que el de las instrucciones, solamente que siguiendo la sintaxis del case en el switch.

```
def p_switch_lista_case(t):
    '''switch_lista_case    : switch_lista_case switch_case'''

def p_switch_lista_case2(t):
```

```
    '''switch_lista_case    : switch_case'''



def p_switch_case(t):
    '''switch_case   : RCASE expresion TWOPOINTS instrucciones'''
```

Tambien es necesaria la produccion del default del switch, en donde se tiene que deberia contener la secuencia “DEFAULT : instrucciones”

```
def p_switch_default(t):
    '''switch_default   : RDEFAULT TWOPOINTS instrucciones'''
```

- No terminal “while_instr”:
  Produccion para reconocer un while en JPR. Siguiendo la secuencia del while se tiene que debe venir de la siguiente manera: “while (expresion) { instrucciones }”, siendo expresion un booleano.

```
def p_while_instr(t):
    '''while_instr  : RWHILE PARENTHESISOPEN expresion PARENTHESISCLOSE
KEYSIGNOPEN instrucciones KEYSIGNCLOSE'''
```

- No terminal “for_instr”:
  Este no terminal esta dividido en tres secciones, esto a causa de que la instruccion for puede ser dividida dentro de dos secciones mas, ya que se tiene una declaracion o asignacion al inicio por ejemplo “for (var i = 0)” o “for (i = 0)”, para lo cual esta e lno terminal “declarar_asignar_for” que produce una asignacion o una declaracion.

  Seguidamente tenemos una expresion que la validamos con un no terminal expresion. Por ultimo paso se tiene un no terminal que sera un incremental o un decremental, pero tambien puede ser una asignacion, a este nor terminal se le nombro “actualizacion_for”. Por lo que se declara la secuencia “FOR (asignar/declarar ; expresion ; incremento/decremento/asignacion){instrucciones}.

```
def p_for_instr(t):
    '''for_instr    : RFOR PARENTHESISOPEN declarar_asignar_for SEMICOLON
expresion SEMICOLON actualizacion_for PARENTHESISCLOSE KEYSIGNOPEN
instrucciones KEYSIGNCLOSE'''

def p_declarar_asignar_for(t):
    '''declarar_asignar_for     : declarar_instr
                                | asignar_instr
    '''

def p_actualizacion_for(t):
    '''actualizacion_for    : unoenuno_instr
                            | asignar_instr'''
```

- No terminal “break_instr” y “continue_instr”:
  Esta produccion es para la instrucción break; cuando se tiene un ciclo el cual se quiera romper. Tambien se encuentra la instrucción continue, que deberia venir cuando se tiene un ciclo.

```
def p_break_instr(t):
    'break_instr      : RBREAK'

def p_continue_instr(t):
    'continue_instr      : RCONTINUE'
```

- No terminal “main_isntr”:
  Instruccion para determinar la secuencia de main. Esta solamente sera la palabra reservada con una lista de instrucciones dentro de las llaves.

```
def p_main_instr(t):
    '''main_instr    : RMAIN PARENTHESISOPEN PARENTHESISCLOSE KEYSIGNOPEN
instrucciones KEYSIGNCLOSE'''
```

- No terminal “function_instr”:
  Para poder realizar la secuencia de las fucniones se necesitan de dos producciones ya que una funcion puede venir sin parametros o con parametros. Con lo cual la unica diferencia de estas es que una no tiene una lista de parametros (o el no terminal “parametros_lista”).

```
def p_function_completa(t):
    '''function_instr    : RFUNC ID PARENTHESISOPEN parametros_lista
PARENTHESISCLOSE KEYSIGNOPEN instrucciones KEYSIGNCLOSE
    '''
```

```
def p_function_simple(t):
    '''function_instr    : RFUNC ID PARENTHESISOPEN PARENTHESISCLOSE
KEYSIGNOPEN instrucciones KEYSIGNCLOSE
    '''
```

- No terminal “parametros_lista”:
  La lista de parametros es realizada por medio de tres producciones. Siendo la instrucción que se repite la secuencia “tipo_dato ID”, esto por que al definir una funcion con parametros, se declaran estos ids con su tipo de dato. Tambien tenemos que recordar que los parametros de una funcion pueden ser arreglos, por lo que tambien el parametro simple sigue la secuencia de int [][][] arreglo.

```
def p_parametros_lista_1(t):
    '''parametros_lista    :    parametros_lista COMA
parametro_simple'''
```

```
def p_parametros_lista_2(t):
    '''parametros_lista    :    parametro_simple'''
```

```
def p_parametro_simple(t):
    '''parametro_simple    :   tipo_dato ID'''
```

```
def p_parametro_simple_arreglo(t):
    '''parametro_simple    :   tipo_dato lista_dimensiones ID '''
```

- No terminal “llamada_function”:
  La llamda a una funcion contiene una lista de parametros, pero esta vez de llamada, con lo cual esta lista es una expresion. Tambien esta llamada puede no contener parametros, por lo que se hace una produccion que no tenga listas de parametros de llamada.

```
def p_llamada_function_simple(t):
    '''llamada_function : ID PARENTHESISOPEN PARENTHESISCLOSE'''

def p_llamada_function_completa(t):
    '''llamada_function :   ID PARENTHESISOPEN parametros_llamada
PARENTHESISCLOSE'''

def p_parametros_llamada1(t):
    '''parametros_llamada   :   parametros_llamada COMA
parametro_llamada_simple'''

def p_parametros_llamada2(t):
    '''parametros_llamada   :   parametro_llamada_simple'''

def p_parametro_llamada_simple(t):
    '''parametro_llamada_simple :   expresion'''
```

- No terminal “return_instr”:
  Este no terminal esta hecho para las funciones directamente. Cuando se declara un funcion que puede retornar un valor siempre tendra la expresion, por lo que al momento de colocarlo en una funcion se valida que lleve una expresion.

```
def p_return_instr(t):
    '''return_instr :   RRETURN expresion'''
```

- No terminal “arreglo_declarar”:
  Como se pueden declarar arreglos tambien, por el tipo 1 de declaracion, este puede tener una lista de dimensiones, lista de expresiones, un id y dos tipos de dato.

  Tambien hay que tomar en cuenta que se puede declarar un arreglo igualandolo a otro ID que sea otro arreglo, por lo que este tipo se divide en dos producciones.

  La lista expresiones es para las secuencias seguidas del new, como [4][8][2] y la lista de dimensiones es para los que vengan despues del dato, solamente [][][].

```
def p_arreglo_declarar(t):
    '''arreglo_declarar :   tipo_1'''
```

```
def p_arreglo_tipo_1(t):
    '''tipo_1   :   tipo_dato lista_dimensiones ID EQUALSYMBOL RNEW
tipo_dato lista_expresiones'''
```

```
def p_arreglo_tipo_1_2(t):
    '''tipo_1   :   tipo_dato lista_dimensiones ID EQUALSYMBOL ID'''
```

```
def p_lista_dimensiones1(t):
    '''lista_dimensiones    :   lista_dimensiones CLASPSYMBOLOPEN
CLASPSYMBOLCLOSE'''
```

```
def p_lista_dimensiones2(t):
    '''lista_dimensiones    :   CLASPSYMBOLOPEN CLASPSYMBOLCLOSE'''
```

```
def p_lista_expresiones1(t):
    '''lista_expresiones    :   lista_expresiones CLASPSYMBOLOPEN
expresion CLASPSYMBOLCLOSE'''
```

```
def p_lista_expresiones2(t):
    '''lista_expresiones    :   CLASPSYMBOLOPEN expresion
CLASPSYMBOLCLOSE'''
```

- No terminal “modificar_arreglo”:
  Esto no es mas que la secuencia de una posicion de un arreglo igualaado a un valor, por lo que se le coloca una lista de ecpresiones definida anteriormente e igualado a una funcion.

```
def p_modificar_arreglo(t):
    '''modificar_arreglo    :   ID lista_expresiones EQUALSYMBOL
expresion'''
```

- No terminal “tipo_dato”:
  Con est produccion se manejan los distintos tipos de datos.

```
def p_tipo(t):
    '''tipo_dato    :   RINT
                    |   RDOUBLE
                    |   RBOOLEAN
                    |   RCHAR
                    |   RSTRING
                    '''
```

- No terminal “expresion”:
  Esta produccion es una produccion recursiva por la izquierda, ya que puede ser muchas veces la misma expresion. Posteriormente se vera que esta expresion es un primitivo.

```
def p_expresion_binaria(t):
    '''
    expresion : expresion PLUSSIGN expresion
              | expresion SUBTRACTIONSIGN expresion
              | expresion MULTIPLICATIONSIGN expresion
              | expresion DIVISIONSIGN expresion
              | expresion POWERSIGN expresion
              | expresion MODULESIGN expresion
              | expresion EQUALIZATIONSIGN expresion
              | expresion DIFFERENTIATIONSIGN expresion
              | expresion SMALLERTHAN expresion
              | expresion GREATERTHAN expresion
              | expresion LESSEQUAL expresion
              | expresion GREATEREQUAL expresion
              | expresion OR expresion
              | expresion AND expresion
    '''
```

```
def p_expresion_unaria(t):
    '''
        expresion : SUBTRACTIONSIGN expresion %prec UMENOS
                  | NOT expresion %prec UNOT
    '''
```

```
def p_expresion_agrupar(t):
    '''expresion : PARENTHESISOPEN expresion PARENTHESISCLOSE'''
```

```
def p_expresion_unoenuno(t):
    '''
        expresion :  expresion INCREMENT
                  |  expresion DECREMENT
    '''
```

```
def p_expresion_llamada(t):
    '''expresion    :   llamada_function'''
```

```
def p_expresion_identificador(t):
    '''expresion : ID'''
```

```
def p_expresion_casteo(t):
    '''expresion    :   PARENTHESISOPEN tipo_dato PARENTHESISCLOSE
expresion'''
```

```
def p_expresion_read(t):
    '''expresion    :   RREAD PARENTHESISOPEN PARENTHESISCLOSE'''
```

```
def p_expresion_arreglo(t):
    '''expresion    :   ID lista_expresiones'''
```

```
def p_expresion_entero(t):
    '''expresion : ENTERO'''
```

```
def p_primitivo_decimal(t):
    '''expresion : DECIMAL'''
```

```
def p_primitivo_cadena(t):
    '''expresion : CADENA'''
```

```
def p_primitivo_caracter(t):
    '''expresion : CARACTER'''
```

```
def p_primitivo_true(t):
    '''expresion : RTRUE'''
```

```
def p_primitivo_false(t):
    '''expresion : RFALSE'''
```

```
def p_primitivo_null(t):
    '''expresion : RNULL'''
```

Como se puede ver, esta produccion expresion produce lo que son primitivos, lo que pueden ser cadenas, booleanos, strings y enteros.