Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Carrera: Ingeniería en ciencias y sistemas
Catedrático: Ing. Mario Bautista
Auxiliar: José Puac
Curso: Organización de lenguajes y compiladores 1
Sección **“N”**

GRAMATICA

José Abraham Solórzano Herrera
201800937

Guatemala 4 de Julio del 2021

INTRODUCCION

El software es un intérprete de alto nivel, es un lenguaje exclusivo de la Universidad de San Carlos de Guatemala, el cual se llama JPR, consiste en un editor, cuya finalidad es proporcionar ciertas funcionalidades, características, herramientas que serán de utilidad al usuario. La funcionalidad del editor será el ingreso del código fuente que será analizado, donde podrá aceptar archivos con extensión ".jpr" y mostrará la línea actual, está diseñado en el lenguaje de programación Python, por medio de una librería PLY.

OBJETIVOS

Objetivo General:

Aplicar los conocimientos sobre la fase de análisis léxico, sintáctico y semántico de un compilador para la realización de un intérprete completo, con las funcionalidades principales para que sea funcional.

Objetivo Especifico:

1. Reforzar los conocimientos de análisis léxico, sintáctico y semántico para la creación de un lenguaje de programación.
2. Aplicar los conceptos de compiladores para implementar el proceso de interpretación de código de alto nivel.
3. Aplicar los conceptos de compiladores para analizar un lenguaje de programación y producir las salidas esperadas.
4. Aplicar la teoría de compiladores para la creación de soluciones de software.

GRAMATICA

Figura No. 1: Reservadas.

Se inicia creado las reservadas, son las palabras ya definidas por el programa, las cuales serán parte de los tokens Figura No. 2.

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

Figura No.2: Tokens

Print: Tendrá la función de mostrar el contenido.

Var: La variable nos sirve para iniciar declarando una variable.

True: Es una variable primitiva Booleana.

False: Es una variable primitiva Booleana.

If: Es una sentencia de control.

Else: Es una sentencia de control.

While: Es una sentencia de control.

Break: Es una sentencia de trasferencia.

Null: Es un tipo de dato primitiva, que representa el valor nulo o vacío.

Main: Es el encargado de poder ejecutar todo el código generado dentro del lenguaje.

Func: Una función es una subrutina de código que se identifica con un nombre, un conjunto de parámetros y de instrucciones.

```
tokens  = [
    'PUNTOCOMA',
    'COMA',
    'PARA',
    'PARC',
    'LLAVEA',
    'LLAVEC',
    'IGUAL',
    'MAS',
    'MENOS',
    'POR',
    'DIV',
    'POT',
    'MODULO',
    'MENORQUE',
    'MENORIGUAL',
    'MAYORIGUAL',
    'MAYORQUE',
    'IGUALIGUAL',
    'DIFERENCIA',
    'AND',
    'OR',
    'NOT',
    'DECIMAL',
    'ENTERO',
    'CADENA',
    'CHAR',
    'ID',
    'COMENTARIO_SIMPLE',
    'COMENTARIO_VARIAS_LINEAS',
    'INCREMENTO',
    'DECREMENTO',
    'DOSPUNTOS',
] + list(reservadas.values())
```

For: Es una sentencia cíclica.

Switch: Es una sentencia cíclicla.

Case: Es parte del Switch y es una sentencia cíclica.

Default: Es parte del Switch y es una sentencia cíclica.

Return: Es una sentencia de trasnferencia.

Int: Es un tipo de dato primitivo, que representa el valor entero.

Double: Es un tipo de dato primitivo, que representa el valor de decimal.

String: Es un tipo de dato primitivo, que representa el valor de una cadena.

Char: Es un tipo de dato primitivo, que representa el valor de un carácter.

Boolean: Es un tipo de dato primitivo, que representa el valor de un boolean.

Continue: Es una sentencia de trasferencia.

Read: Esta función nos permite obtener valores que queramos ingresar en el momento de ejecución del código.

Figura No. 3: Asignación al valor tokens.

```
# Tokens
t_PUNTOCOMA     = r';'
t_COMA          = r','
t_PARA          = r'\('
t_PARC          = r'\)'
t_LLAVEA        = r'\{'
t_LLAVEC        = r'\}'
t_IGUAL         = r'='
t_MAS           = r'\+'
t_MENOS         = r'-'
t_POR           = r'\*'
t_DIV           = r'\/'
t_POT           = r'\*\*'
```

En la Figura No. 3 se asigna a cada respectivo token su valor.

```
# Caracteres ignorados
t_ignore = " \t"
```

```
t_MODULO        = r'\%'
t_MENORQUE      = r'<'
t_MENORIGUAL    = r'<='
t_MAYORQUE      = r'>'
t_MAYORIGUAL    = r'>='
t_IGUALIGUAL    = r'=='
t_AND           = r'&&'
t_OR            = r'\|\|'
t_NOT           = r'!'
t_DIFERENCIA    = r'=!'
t_INCREMENTO    = r'\+\+'
t_DECREMENTO    = r'\-\-'
t_DOSPUNTOS     = r'\:'
```

Figura No. 4: Decimal.

```
def t_DECIMAL(t):
    r'\d+\.\d+'
    try:
        t.value = float(t.value)
    except ValueError:
        print("Float value too large %d", t.value)
        t.value = 0
    return t
```

En esta figura se encarga de reconocer si la expresión es un decimal y es parte del t_decimal.

Figura No. 5: Entero.

```
def t_ENTERO(t):
    r'\d+'
    try:
        t.value = int(t.value)
    except ValueError:
        print("Integer value too large %d", t.value)
        t.value = 0
    return t
```

En esta figura se encarga de reconocer si la expresión es un entero y es parte del t_entero.

Figura No. 6: Id

```
def t_ID(t):
    r'[a-zA-Z][a-zA-Z_0-9]*'
    t.type = reservadas.get(t.value.lower(),'ID')
    return t
```

En esta figura se encarga de reconocer si la expresión es una variable y es parte del t_id.

Figura No.7:

```
def t_CADENA(t):
    #r'(\".*?\")'
    # t.value = t.value[1:-1] # remuevo las comillas
    # return t
    r'\"(\\"|.)*?\"'
    t.value = t.value[1:-1]  # remover comillas
    t.value = t.value.replace('\\n', '\n')
    t.value = t.value.replace('\\r', '\r')
    t.value = t.value.replace('\\\\', '\\')
    t.value = t.value.replace('\\"', '\"')
    t.value = t.value.replace('\\t', '\t')
    t.value = t.value.replace("\\'", '\'')
    return t
```

En esta figura se encarga de reconocer si la expresión es una cadena y es parte del t_cadena.

Figura No. 8:

```
def t_CHAR(t):
    r"""\' (\\'| \\\\ | \\n | \\t | \\r | \\" | .)? \'"""
    t.value = t.value[1:-1]  # remover comillas
    t.value = t.value.replace('\\n', '\n')
    t.value = t.value.replace('\\r', '\r')
    t.value = t.value.replace('\\\\', '\\')
    t.value = t.value.replace('\\"', '\"')
    t.value = t.value.replace('\\t', '\t')
    t.value = t.value.replace("\\'", '\'')
    return t
```

En esta figura se encarga de reconocer si la expresión es un carácter y es parte del t_char.

Figura No. 9:

```
def t_COMENTARIO_VARIAS_LINEAS(t):
    r'\#\*(.|\n)*?\*\#'
    t.lexer.lineno += t.value.count("\n")
```

En esta figura se encarga de reconocer si es un comentario multi línea.

Figura No. 10:

```
# Comentario simple // ...
def t_COMENTARIO_SIMPLE(t):
    r'\#.*\n'
    t.lexer.lineno += 1
```

En esta figura se encarga de reconocer si es un comentario de una línea.

Figura No. 11:

```
def t_newline(t):
    r'\n+'
    t.lexer.lineno += t.value.count("\n")
```

Este token se encarga de reconocer un salto de línea.

Figura No. 12:

```
def t_error(t):
    errores.append(Exception("Lexico","Error léxico." + t.value[0] , t.lexer.lineno, find_column(input, t)))
    t.lexer.skip(1)

# Compute column
```

Este token se encarga de reconocer si

Figura No. 13:

```
def find_column(inp, token):
    line_start = inp.rfind('\n', 0, token.lexpos) + 1
    return (token.lexpos - line_start) + 1
```

Este token se encarga de reconocer la fila.

Figura No. 14:

```
# Construyendo el analizador léxico
import Interprete.ply.lex as lex
lexer = lex.lex()
# Asociacion
precedence = (
    ('left','OR'),
    ('left','AND'),
    ('right','UNOT'),
    ('left','IGUALIGUAL','DIFERENCIA','MENORQUE','MENORIGUAL','MAYORQUE','MAYORIGUAL'),
    ('left','MAS','MENOS'),
    ('left','DIV','POR','MODULO'),
    ('nonassoc', 'POT'),
    ('right','UMENOS'),
    )
```

En esta figura se asigna la precedencia de la gramática.

Figura No. 15:

```
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]

# ------------------------------------------- INSTRUCCIONES ------------------------------------------

def p_instrucciones_instruccion(t) :
    'instrucciones    : instruccion'
    if t[1] == "":
        t[0] = []
    else:
        t[0] = [t[1]]
```

Esta Figura es donde inicia la gramática, y representa la gramática ascendente recursiva por la izquierda, usando LALR.

Figura No. 16:

```
# ------------------------------------------- INSTRUCCION ------------------------------------------

def p_instruccion(t):
    '''instruccion  : imprimir_ fin_instruccion
                    | declaracion_ins fin_instruccion
                    | incre_decre_ins fin_instruccion
                    | if_ins
                    | while_ins
                    | switch_ins
                    | for_ins
                    | main_ins
                    | break_ins fin_instruccion
                    | return_ins fin_instruccion
                    | continue_ins fin_instruccion
                    | funcion_ins
                    | llamada_ins fin_instruccion
                    | COMENTARIO_VARIAS_LINEAS
                    | COMENTARIO_SIMPLE


    '''
    t[0] = t[1]
```

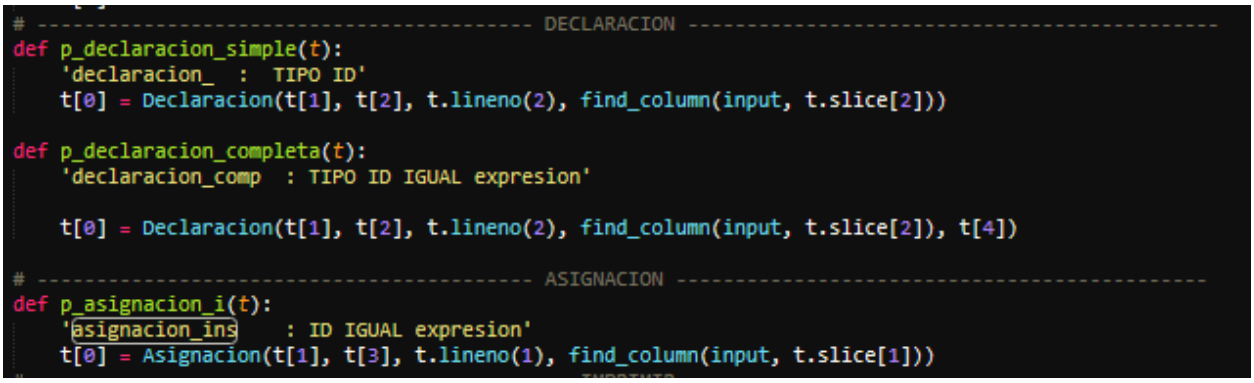
En esta figura representa todas las posibles instrucciones que puede contener la gramática.

Figura No. 17:

```
def p_decla(t):
    ''' declaracion_ins : declaracion_
                        | declaracion_comp
                        | asignacion_ins '''

    t[0] = t[1]
```

En esta figura inicia buscando si es una declaración o una asignación.

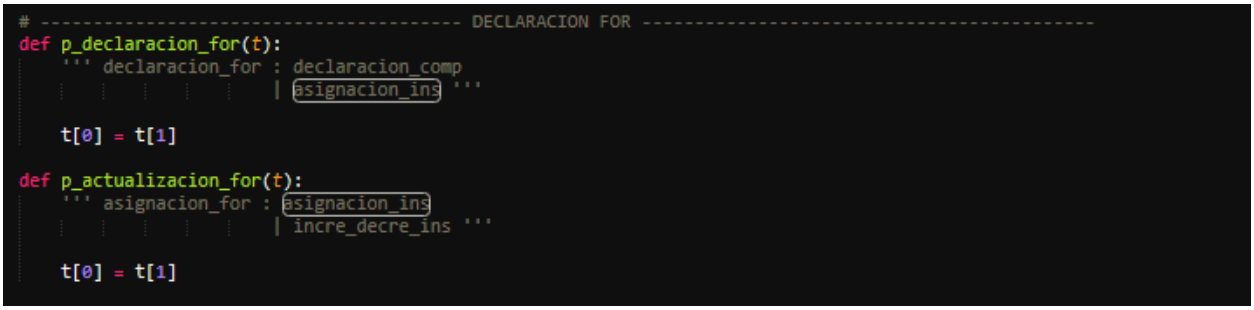
Figura No. 18:



```
# ---------------------------------------- DECLARACION ----------------------------------------
def p_declaracion_simple(t):
    'declaracion_  :  TIPO ID'
    t[0] = Declaracion(t[1], t[2], t.lineno(2), find_column(input, t.slice[2]))

def p_declaracion_completa(t):
    'declaracion_comp  : TIPO ID IGUAL expresion'

    t[0] = Declaracion(t[1], t[2], t.lineno(2), find_column(input, t.slice[2]), t[4])

# ---------------------------------------- ASIGNACION ----------------------------------------
def p_asignacion_i(t):
    'asignacion_ins    : ID IGUAL expresion'
    t[0] = Asignacion(t[1], t[3], t.lineno(1), find_column(input, t.slice[1]))
```

En esta figura representa la declaración y la asignación.

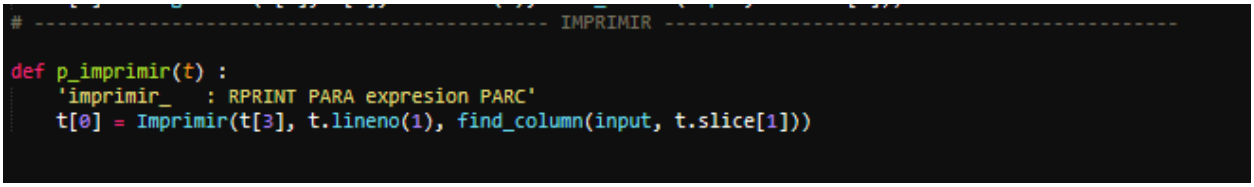
Figura No. 19:



```
# ---------------------------------------- DECLARACION FOR ----------------------------------------
def p_declaracion_for(t):
    ''' declaracion_for : declaracion_comp
                        | asignacion_ins '''

    t[0] = t[1]

def p_actualizacion_for(t):
    ''' asignacion_for : asignacion_ins
                       | incre_decre_ins '''

    t[0] = t[1]
```

En esta gramática se crea la declaración o asignación para el for.

Figura No. 20:



```
# ---------------------------------------- IMPRIMIR ----------------------------------------

def p_imprimir(t) :
    'imprimir_    : RPRINT PARA expresion PARC'
    t[0] = Imprimir(t[3], t.lineno(1), find_column(input, t.slice[1]))
```

En esta parte de la gramática, reconoce la palabra reservada print y su función principal es mostrar el resultado.

Figura No. 21:

```
# ------------------------------------------- SENTENCIA IF ---------------------------------------------

def p_condi_if(t): # Condicion if si solo viene un if
    'if_ins     : RIF PARA expresion PARC LLAVEA instrucciones LLAVEC'
    t[0] = If(t[3], t[6], None, None, t.lineno(1), find_column(input, t.slice[1]))

def p_condi_if_dos(t) : # condicion if si solo viene un if y un else
    'if_ins     : RIF PARA expresion PARC LLAVEA instrucciones LLAVEC RELSE LLAVEA instrucciones LLAVEC'
    t[0] = If(t[3], t[6], t[10], None, t.lineno(1), find_column(input, t.slice[1]))

def p_condi_if_tres(t) : # condicion para que pueda venir un else if, o un else if y un else.
    'if_ins     : RIF PARA expresion PARC LLAVEA instrucciones LLAVEC RELSE if_ins'
    t[0] = If(t[3], t[6], None, t[9], t.lineno(1), find_column(input, t.slice[1]))
```

En esta parte de la gramática reconoce las posibles alternativas de la sentencia if, ya sea un else if o un else.

Figura No. 22:

```
# ------------------------------------------- SENTENCIA SWITCH ---------------------------------------------
def p_condicion_switch_case_list_default(t): # Aqui verifiac que la condicion venga  [<CASES_LIST>] [<DEFAULT>]
    'switch_ins   : RSWITCH PARA expresion PARC LLAVEA case_switch_ins default_switch LLAVEC'
    t[0] = Switch(t[3], t[6], t[7] ,t.lineno(1), find_column(input, t.slice[1]))

def p_condicion_switch_case_list(t): # Aqui verifiac que la condicion venga  [<CASES_LIST>]
    'switch_ins   : RSWITCH PARA expresion PARC LLAVEA case_switch_ins LLAVEC'
    t[0] = Switch(t[3], t[6], None, t.lineno(1), find_column(input, t.slice[1]))

def p_condicion_switch_default(t): # Aqui verifiac que la condicion venga   [<DEFAULT>]
    'switch_ins   : RSWITCH PARA expresion PARC LLAVEA default_switch LLAVEC'
    t[0] = Switch(t[3], None, t[6], t.lineno(1), find_column(input, t.slice[1]))

def p_casos_switch_ins_caso_switch(t): # Aqui hace a que sea recursivo y puedan venir infinitos [<CASES_LIST>]
    'case_switch_ins : case_switch_ins case_switch'
    if t[2] != "":
        t[1].append(t[2])
    t[0] = t[1]

def p_caso_switch_(t):
    'case_switch_ins : case_switch'
    if t[1] == "":
        t[0] = []
    else:
        t[0] = [t[1]]

def p_condicion_switch_case(t):
    'case_switch  : RCASE expresion DOSPUNTOS instrucciones'
    t[0] = Case(t[2], t[4], t.lineno(1), find_column(input, t.slice[1]))

def p_condicion_default_switch(t):
    'default_switch : RDEFAULT DOSPUNTOS instrucciones'
    # t[0] = t[3]
    t[0] = Default(t[3], t.lineno(1), find_column(input, t.slice[1]))
```

En esta parte se vuelve a utilizar la gramática ascendente por la izquierda, ya que en el switch pueden venir una infinidad de case.

Figura No. 23:

```
# ------------------------------------------- WHILE ----------------------
def p_sentencia_while(t) :
    'while_ins     : RWHILE PARA expresion PARC LLAVEA instrucciones LLAVEC'
    t[0] = While(t[3], t[6], t.lineno(1), find_column(input, t.slice[1]))
```

en esta figura la gramática puede reconocer la sentencia cíclica del while.

Figura No. 24:

```
# ------------------------------------------- FOR ------------------------------------------
def p_sentencia_for(t) :
    'for_ins     : RFOR PARA declaracion_for PUNTOCOMA expresion PUNTOCOMA asignacion_for PARC LLAVEA instrucciones LLAVEC'
    t[0] = For(t[3], t[5], t[7], t[10],  t.lineno(1), find_column(input, t.slice[1]))
```

En esta parte de la gramática reconoce la sentencia de cíclica del for.

Figura No. 25:

```
# ------------------------------------------- FUNCION ------------------------------------------

def p_funcion_2(t) :
    'funcion_ins     : RFUNC ID PARA PARC LLAVEA instrucciones LLAVEC'
    t[0] = Funcion(t[2], [], t[6], t.lineno(1), find_column(input, t.slice[1]))

def p_funcion_parametros(t) :
    'funcion_ins     : RFUNC ID PARA parametros PARC LLAVEA instrucciones LLAVEC'
    t[0] = Funcion(t[2], t[4], t[7], t.lineno(1), find_column(input, t.slice[1]))

def p_parametros_parametros(t) :
    'parametros     : parametros COMA parametro'
    t[1].append(t[3])
    t[0] = t[1]

def p_parametros_parametro(t) :
    'parametros     : parametro'
    t[0] = [t[1]]

def p_parametro(t) :
    'parametro     : tipo_funcion ID'
    t[0] = {'tipoDato':t[1],'identificador':t[2]} # Se crea un diccionario tipoDato: tipo, identificador
```

En esta parte de la gramática puede reconocer una función, y hacer una declaración por medio de los parámetros, en este caso en los parámetros se vuelve a utilizar la gramatica ascendente por la izquierda.

Figura No. 26:

```
# ------------------------------------------- LLAMADA ------------------------------------------

def p_llamada_de_funcion(t) :
    'llamada_ins     : ID PARA PARC'
    t[0] = Llamada(t[1], [], t.lineno(1), find_column(input, t.slice[1]))


def p_llamada_de_fincion_parametros(t) :
    'llamada_ins     : ID PARA parametros_llamada PARC'
    t[0] = Llamada(t[1], t[3], t.lineno(1), find_column(input, t.slice[1]))

def p_parametros_llamadas_parametros_llamadas(t) :
    'parametros_llamada     : parametros_llamada COMA parametro_llamada'
    t[1].append(t[3])
    t[0] = t[1]

def p_parametros_llamadas_parametro_llamada(t) :
    'parametros_llamada    : parametro_llamada'
    t[0] = [t[1]]

def p_parametro_llamada(t) :
    'parametro_llamada     : expresion'
    t[0] = t[1]
```

En esta parte de la gramática se hace el llamado de la función y se empieza a utilizar la gramática ascendente por la izquierda.

Figura No. 27:

```
# ------------------------------------------- BREAK ------------------------------------------
def p_sentencia_break(t) :
    'break_ins      : RBREAK'
    t[0] = Break(t.lineno(1), find_column(input, t.slice[1]))
```

En esta parte reconoce la gramática la sentencia break.

Figura No. 28:

```
# ------------------------------------------- RETURN -----------------------------------------
def p_return_instruccion(t) :
    'return_ins      : RRETURN expresion'
    t[0] = Return(t[2], t.lineno(1), find_column(input, t.slice[1]))
```

En esta parte reconoce la gramática la sentencia return.

Figura No. 29:

```
# ------------------------------------------- CONTINUE ---------------------------------------
def p_continue_instruccion(t) :
    'continue_ins      : RCONTINUE'
    t[0] = Continue(t.lineno(1), find_column(input, t.slice[1]))
```

En esta parte la gramática reconoce la sentencia continúe.

Figura No. 30:

```
# ------------------------------------------- TIPO -------------------------------------------
def p_tipo_dato(t):
    '''TIPO :  RVAR
            |'''
    if t[1] == 'var':
        t[0] = Tipo.NULO

def p_tipo_funcion(t):
    ''' tipo_funcion    : RINT
                        | RDOUBLE
                        | RSTRING
                        | RCHAR
                        | RBOOLEAN'''
    if t[1].lower() == 'int':
        t[0] = Tipo.ENTERO
    elif t[1].lower() == 'double':
        t[0] = Tipo.DECIMAL
    elif t[1].lower() == 'string':
        t[0] = Tipo.CADENA
    elif t[1].lower() == 'char':
        t[0] = Tipo.CHAR
    elif t[1].lower() == 'boolean':
        t[0] = Tipo.BOOLEANO
```

En esta figura representa todos los posibles datos, sin embargo, el primer tipo sirve para las declaraciones, en cambio el tipo función sirve para crear funciones donde van a resivir un tipo “tipo_funcion”.

Figura No. 31:

```
# ------------------------------------------- INCREMENTO O DECREMENTO ------------------------------------------
def p_incremento_decremento(t):
    ''' incre_decre_ins : ID INCREMENTO
                        | ID DECREMENTO'''

    if t[2] == '++':
        t[0] = IncrementoDecremento(t[1], Operador_Aritmetico.INCREMENTO, t.lineno(1), find_column(input, t.slice[1]))
    elif t[2] == '--':
        t[0] = IncrementoDecremento(t[1], Operador_Aritmetico.DECREMENTO, t.lineno(1), find_column(input, t.slice[1]))
```

En esta parte de la gramática reconoce los incrementos y decrementos que pueden llegar a venir.

Figura No. 32:

```
def p_expresion_binaria(t):
    ''' expresion : expresion MAS expresion
                  | expresion MENOS expresion
                  | expresion POR expresion
                  | expresion DIV expresion
                  | expresion POT expresion
                  | expresion MODULO expresion
                  | expresion MENORQUE expresion
                  | expresion MENORIGUAL expresion
                  | expresion MAYORQUE expresion
                  | expresion MAYORIGUAL expresion
                  | expresion IGUALIGUAL expresion
                  | expresion DIFERENCIA expresion
                  | expresion AND expresion
                  | expresion OR expresion
    '''
    if t[2] == '+':
        t[0] = Aritmetica(Operador_Aritmetico.SUMA, t[1],t[3], t.lineno(2), find_column(input, t.slice[2]))
    elif t[2] == '-':
        t[0] = Aritmetica(Operador_Aritmetico.RESTA, t[1],t[3], t.lineno(2), find_column(input, t.slice[2]))
    elif t[2] == '*':
        t[0] = Aritmetica(Operador_Aritmetico.POR, t[1],t[3], t.lineno(2), find_column(input, t.slice[2]))
    elif t[2] == '/':
        t[0] = Aritmetica(Operador_Aritmetico.DIV, t[1],t[3], t.lineno(2), find_column(input, t.slice[2]))
    elif t[2] == '**':
        t[0] = Aritmetica(Operador_Aritmetico.POTE, t[1],t[3], t.lineno(2), find_column(input, t.slice[2]))
    elif t[2] == '%':
        t[0] = Aritmetica(Operador_Aritmetico.MODU, t[1],t[3], t.lineno(2), find_column(input, t.slice[2]))

    elif t[2] == '==':
        t[0] = Relacional(Operador_Relacional.IGUALACION, t[1],t[3], t.lineno(2), find_column(input, t.slice[2]))
    elif t[2] == '<':
        t[0] = Relacional(Operador_Relacional.MENORQUE, t[1],t[3], t.lineno(2), find_column(input, t.slice[2]))
    elif t[2] == '<=':
        t[0] = Relacional(Operador_Relacional.MENORIGUAL, t[1],t[3], t.lineno(2), find_column(input, t.slice[2]))
    elif t[2] == '>':
        t[0] = Relacional(Operador_Relacional.MAYORQUE, t[1],t[3], t.lineno(2), find_column(input, t.slice[2]))
    elif t[2] == '>=':
        t[0] = Relacional(Operador_Relacional.MAYORIGUAL, t[1],t[3], t.lineno(2), find_column(input, t.slice[2]))
    elif t[2] == '=!':
        t[0] = Relacional(Operador_Relacional.DIFERENCIA, t[1],t[3], t.lineno(2), find_column(input, t.slice[2]))

    elif t[2] == '&&':
        t[0] = Logica(Operador_Logico.AND, t[1],t[3], t.lineno(2), find_column(input, t.slice[2]))
    elif t[2] == '||':
        t[0] = Logica(Operador_Logico.OR, t[1],t[3], t.lineno(2), find_column(input, t.slice[2]))
```

En esta parte empieza a reconocer las expresiones, ya sea aritmética, ralacional, o lógica.

Figura No. 33:

```
def p_expresion_unaria(t):
    '''
    expresion : MENOS expresion %prec UMENOS
              | NOT expresion %prec UNOT
    '''
    if t[1] == '-':
        t[0] = Aritmetica(Operador_Aritmetico.UMENOS, t[2],None, t.lineno(1), find_column(input, t.slice[1]))
    elif t[1] == '!':
        t[0] = Logica(Operador_Logico.NOT, t[2],None, t.lineno(1), find_column(input, t.slice[1]))
```

En esta parte de la gramática reconoce las expresiones por la izquierda, como vienen siendo.

Figura No. 34:

```
def p_expresion_agrupacion(t):
    '''
    expresion :   PARA expresion PARC
    '''
    t[0] = t[2]
```

En esta parte de la gramática hace las agrupaciones siendo recursivo.

Figura No. 35:

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

    t[0] = Identificador(t[1], t.lineno(1), find_column(input, t.slice[1]))

def p_expresion_entero(t):
    '''expresion : ENTERO'''
    t[0] = Primitivos(Tipo.ENTERO,t[1], t.lineno(1), find_column(input, t.slice[1]))

def p_primitivo_decimal(t):
    '''expresion : DECIMAL'''
    t[0] = Primitivos(Tipo.DECIMAL, t[1], t.lineno(1), find_column(input, t.slice[1]))

def p_primitivo_cadena(t):
    '''expresion : CADENA'''
    t[0] = Primitivos(Tipo.CADENA,str(t[1]).replace('\\n', '\n'), t.lineno(1), find_column(input, t.slice[1]))

def p_primitivo_char(t):
    '''expresion : CHAR'''
    t[0] = Primitivos(Tipo.CHAR,str(t[1]).replace('\\n', '\n'), t.lineno(1), find_column(input, t.slice[1]))

def p_primitivo_true(t):
    '''expresion : RTRUE'''
    t[0] = Primitivos(Tipo.BOOLEANO, True, t.lineno(1), find_column(input, t.slice[1]))

def p_primitivo_false(t):
    '''expresion : RFALSE'''
    t[0] = Primitivos(Tipo.BOOLEANO, False, t.lineno(1), find_column(input, t.slice[1]))

def p_primitivo_null(t):
    '''expresion : RNULL '''
    t[0] = Primitivos(Tipo.NULO, None, t.lineno(1), find_column(input, t.slice[1]))

def p_expresion_llamada(t):
    '''expresion : llamada_ins'''
    t[0] = t[1]

def p_expresion_casteo(t):
    '''expresion : PARA tipo_funcion PARC expresion'''
    t[0] = Casteo(t[2], t[4], t.lineno(1), find_column(input, t.slice[1]))

def p_expresion_read(t):
    '''expresion : RREAD PARA PARC'''
    t[0] = Read(t.lineno(1), find_column(input, t.slice[1]))
```

En esta parte de la gramática guarda todas las expresiones que sean tipo primitivo, identificador, casteo y read.

Figura No. 36:N

```
def crearNativas(ast):          # CREACION Y DECLARACION DE LAS FUNCIONES NATIVAS
    nombre = "toupper"
    parametros = [{'tipoDato':Tipo.CADENA,'identificador':'toUpper##Param1'}]
    instrucciones = []
    toUpper = ToUpper(nombre, parametros, instrucciones, -1, -1)
    ast.addFuncion(toUpper)     # GUARDAR LA FUNCION EN "MEMORIA" (EN EL ARBOL)

    nombre = "tolower"
    parametros = [{'tipoDato':Tipo.CADENA,'identificador':'toLower##Param1'}]
    instrucciones = []
    toLower = ToLower(nombre, parametros, instrucciones, -1, -1)
    ast.addFuncion(toLower)     # GUARDAR LA FUNCION EN "MEMORIA" (EN EL ARBOL)

    nombre = "length"
    parametros = [{'tipoDato':Tipo.CADENA,'identificador':'length##Param1'}]
    instrucciones = []
    length = Length(nombre, parametros, instrucciones, -1, -1)
    ast.addFuncion(length)     # GUARDAR LA FUNCION EN "MEMORIA" (EN EL ARBOL)

    nombre = "truncate"
    parametros = [{'tipoDato':Tipo.ENTERO,'identificador':'truncate##Param1'}]
    instrucciones = []
    truncate = Truncate(nombre, parametros, instrucciones, -1, -1)
    ast.addFuncion(truncate)     # GUARDAR LA FUNCION EN "MEMORIA" (EN EL ARBOL)

    nombre = "round"
    parametros = [{'tipoDato':Tipo.ENTERO,'identificador':'round##Param1'}]
    instrucciones = []
    rround = Round(nombre, parametros, instrucciones, -1, -1)
    ast.addFuncion(rround)     # GUARDAR LA FUNCION EN "MEMORIA" (EN EL ARBOL)

    nombre = "typeof"
    parametros = [{'tipoDato':Tipo.NULO,'identificador':'typeOf##Param1'}]
    instrucciones = []
    typeOf = TypeOf(nombre, parametros, instrucciones, -1, -1)
    ast.addFuncion(typeOf)     # GUARDAR LA FUNCION EN "MEMORIA" (EN EL ARBOL)

def interprete(entrada, consola):
```

En esta parte se crean las funciones nativas, que ya están definidas.