



UNIVERSIDAD DE SAN CARLOS DE GUATEMALA  
Facultad de Ingeniería

**GRUPO No 13**

# MANUAL TECNICO

## DESCRIPCION

Es una Aplicacion con interfaz que funciona como interprete del lenguaje SQL especialmente creado para el curso de Organizacion de Lenguajes y Compiladores 2

Para realizar el interprete se utilizo PLY de Python. Tal como se desarrollara a continuacion

### DESARROLLO CON LAS TECNOLOGIAS:

- **Sistema Operativos:** Ubuntu 20.04 LTS, Windows 10
- **Lenguaje de Programacion:** Python
- **GUI:** Tkinter
- **IDE:** Visual Studio Code, PyCharm
- **Graficas:** graphviz

### Construccion del Analizador

**PLY:** es una herramienta de analisis que esta escrita en el lenguaje de programacion python. Es un generador de analizadores y es una implementacion de lex & yacc. Utiliza el analisis LALR(1) que hace muy potente para gramaticas de todo tipo.

Para este proyecto se implementara una gramatica ascendente y descendente simulada, de la cual se detallara mas adelante.

### Estructura de PLY

- Ply.lex para realizar el analisis lexico
- Ply.yacc para creacion de analizadores sintacticos

### Definición del Analizador Léxico

Para realizar el analizador léxico se crean una serie de tokens y palabras reservadas que serán reconocidas por el analizador.

Cualquier otro carácter o expresión no registrada en la definición del analizador, dará como resultado un error y se guardará el error en una lista de errores.

### Tokens

Se definen de la siguiente manera. Estos son secuencias de caracteres que serán de utilidad para el análisis sintactico.

En el siguiente ejemplo se muestra como se declara los tokens y palabras reservadas  
**por Ej:**

- Aqui se declara palabras reservadas

```
# LISTA DE PALABRAS RESERVADAS
reservadas = {
    # Numeric Types
    'smallint': 'tSmallint',
    'integer': 'tInteger',
    'bigint': 'tBigint',
    'decimal': 'tDecimal',
    'numeric': 'tNumeric',
    'real': 'tReal',
    'double': 'tDouble',
    'precision': 'tPrecision',
    'money': 'tMoney',

    # Character types
    'character': 'tCharacter',
    'varying': 'tVarying',
    'varchar': 'tVarchar',
    'char': 'tChar',
    'text': 'tText',

    # Date/Time Types
    'timestamp': 'tTimestamp',
    'date': 'tDate',
    'time': 'tTime',
    'interval': 'tInterval',

    # Interval Type
    'year': 'tYear',
    'month': 'tMonth',
    'day': 'tDay',
    'hour': 'tHour',
    'minute': 'tMinute',
    'second': 'tSecond',
    'to': 'tTo',
```

- En esta imagen se muestra como se declara los diferentes simbolos

## # DEFINICIÓN DE TOKENS

```

t_punto = r'\.'
t_dosPts = r':.'
t_corcheI = r'\['
t_corcheD = r'\]'

t_mas = r'\+'
t_menos = r'\-'
t_elevado = r'\^'
t_multi = r'\*'
t_divi = r'\/'

t_modulo = r'\%'
t_igual = r'\='
t_menor = r'\<'
t_mayor = r'\>'
t_menorIgual = r'\<='

t_mayorIgual = r'\>='
t_diferente = r'\<>'

t_parAbre = r'\('
t_parCierra = r'\)'
t_coma = r','
t_ptComa = r';'

# tk_queries
t_barra = r'\|'
t_barraDoble = r'\\|\\|'
t_amp = r'\&'
t_numeral = r'\#'
t_virgulilla = r'\~'
t_mayormayor = r'\>>'
t_menormenor = r'\<<'

```

Otra forma de reconocer Tokens así como cadenas, números, etc .. es de la siguiente forma

- En esta imagen se muestra la forma de declarar cadenas :

## # DEFINICIÓN DE UNA CADENA PARA LIKE

```

def t_cadenaLike(t):
    r'\'%.*?%\'|\'\"%.*?%\'\"'
    t.value = t.value[2:-2]
    return t

```

## # DEFINICIÓN DE UNA CADENA

```

def t_cadena(t):
    r'\'.*?\'|\'\".*?\'\"'
    t.value = t.value[1:-1]
    return t

```

- En esta imagen se muestra la forma de declarar números decimales :

## # DEFINICIÓN DE UN NÚMERO 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 imagen se muestra la otra forma de declarar número :

```
# DEFINICIÓN DE UN NÚMERO 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
```

## Manejo de Errores Lexicos

Una función de error para manejar los errores léxicos y es donde también se guarda el error en una lista para ser mostrado en el reporte de errores que es generado en la aplicación.

Un error léxico se dispara cuando el o los caracteres no concuerdan con ningún patrón de nuestras expresiones regulares que estamos definiendo.

**t.lexer.skip(1)** descarta el token.

**t.lexer.lineno** obtiene el número de línea del token.

```
# HALLAR LA COLUMNA DEL TOKEN ESPECIFICADO
def find_column(token): # Columna relativa a la fila
    global con
    line_start = con.rfind('\n', 0, token.lexpos) + 1
    return (token.lexpos - line_start) + 1

# Loading... ERROR PARA LOS ERRORES LÉXICOS
def t_error(t):
    col = find_column(t)
    # print("Caracter inválido '%s'" % t.value[0], " Línea: '%s'" % str(t.lineno))
    errores_lexicos.append(
        Error(t.value[0], 'Error Léxico', "El caracter '\" + str(t.value[0]) + '\" no pertenece al lenguaje", col,
            t.lineno))
    t.lexer.skip(1)
```

## Construccion de Analizador Lexico

Este analizador es el enviado para realizar el análisis sintáctico.

En la imagen siguiente se muestra como se importan las librerías de analizador Lexico

```
# Construyendo el analizador léxico
import ply.lex as lex
import re
```

## Construccion de Analizador Sintactico

Para construir el analizador sintáctico se hará uso del patrón interprete que es mucha utilidad para representar la gramática y realizar la ejecución del código, pero en realidad hay dos formas de realizar la ejecución desde la construcción de árbol con la clase **Node** o con una clase Abstracta, llamada **Sentencias** se mostrara mas adelante de como se puede implementar de la dos formas para poder realizarlo

En este apartado se muestra como se importa la otra librería para la construcción del analizador sintactico

```
import ply.yacc as yacc
```

## Definición de la Gramática en Ply

La forma de declarar producciones en PLY es por medio de una función de Python, seguido de la declaración de una cadena en comilla simple o triples cuando sea de varias líneas.

Así también se puede incrustar reglas semánticas en cada producción que generarán el AST para posterior ejecución.

## Existes dos Tipos de Analisis Sintactico

### Analisis Sintactico Ascendente

- Se construye un árbol sintáctico para una cadena de entrada que comienza por las hojas y avanza hacia la raíz
- Se "reduce" una cadena al símbolo inicial de la gramática
- En cada paso de reducción, se sustituye una subcadena determinada que concuerda con el lado derecho de una producción por el símbolo del lado izquierdo de dicha producción (básicamente se busca el lado derecho que corresponde y se reemplaza por el lado izquierdo y se intenta de esa manera llegar a la raíz)
- Se traza una derivación por la derecha en sentido inverso

### Analisis Sintactico Descendente

- Se busca una derivación por la izquierda para una cadena de entrada
- Se construye el árbol de análisis sintáctico comenzando desde la raíz y creando los nodos en orden previo
- No hay muchos analizadores sintácticos descendentes con retroceso - casi nunca se necesita el retroceso para analizar las construcciones de lenguajes de programación. El retroceso tampoco es muy eficiente.
- Una gramática recursiva por la izquierda puede causar que el analizador sintáctico entre en un lazo infinito (porque no consume ningún símbolo de entrada)
- Al eliminar la recursividad y factorizar se puede obtener una gramática analizable por un analizador sintáctico predictivo por descenso que no necesite retroceso.

### Construcción del Analizador Sintáctico

Para construir el analizador sintáctico se hará uso del patrón interprete con clase abstract llamada Sentencia y de la otra forma es construir el una clase llamada Node sin usar clase abstracta mas adelante se ampliara la informacion .

### Definición de la Gramática

La forma de declarar producciones en PLY es por medio de una función de Python, seguido de la declaración de una cadena en comilla simples o triples cuando sea de varias líneas.

Así también se puede incrustar reglas semánticas en cada producción que generarán el AST para posterior ejecución.

### Gramatica Ascendente

La gramatica Ascendente son las que se produce por lo general por la Izquierda a diferencia del descendente y aqui no son recomendables usar producciones vacias por lo conflictos que esto puede causar.

```
def p_EXPR_ALTER(t):  
    '''EXPR_ALTER : EXPR_ALTER coma alter tColumn id tSet not null ptComa  
                  | EXPR_ALTER coma alter tColumn id ttype CHAR_TYPES ptComa  
                  | alter tColumn id ttype CHAR_TYPES ptComa  
                  | alter tColumn id tSet not null ptComa  
    '''
```

**Nota:** Se muestra en la figura una porcion de gramatica ASC

### Gramatica Descendente

A diferencia de la gramatica ASC sus producciones se van de la derecha y ademas en este tipo de gramatica son permitidas usar producciones vacias, la razon de la misma es para que no lleguen a enciclarse o cause un error de Infinita recursion.

```
def p_EXPR_ALTER(t):
    '''EXPR_ALTER : alter tColumn id tSet not null coma EXPR_ALTER ptComa
    | alter tColumn id ttype CHAR_TYPES coma EXPR_ALTER ptComa
    | alter tColumn id ttype CHAR_TYPES ptComa
    | alter tColumn id tSet not null ptComa
    | ...
    '''
```

**Nota:** Se muestra en la figura una porcion de gramatica DESC

## Uso de la Pila para Gramática Descendente

Al ser PLY un analizador ascendente, para poder generar una gramática descendente hacemos uso de la Pila que nos provee PLY, accediendo a atributos de la pila por medio de un índice negativo. De esta manera, será una forma de heredar atributos.

```
def p_asignacion_valor(t) :
    'asignacion_valor : expresion PTCOMA'
    t[0] = Asignacion(t[-2],t[1])

def p_dec_array_instr(t) :
    'asignacion_valor : ARRAY PARIZQ PARDER PTCOMA'
    t[0] = Array(t[-2])

def p_asignacion_arr_St(t) :
    'asignacion_arr_st : def_par lista_parametros IGUAL expresion PTCOMA'
    t[1].extend(t[2])
    t[0] = AsignacionArrSt(t[-1],t[1],t[4])
```

## Construcción del AST

La creación del **AST** requiere que a cada producción se le asocie una regla semántica. En este caso que estamos aplicando el patrón interprete, los nodos serán de un tipo de clase, además se puede realizar de varias maneras la construcción del **AST** es decir pueden ser con clases abstractas o simplemente una clase

En este caso se muestran dos formas para realizar la construcción del AST

La Primera forma es realizar clases abstractas

```

def p_E(p):
    ''' E : E or E
        | E And      E
        | E diferente E
        | E notEqual E
        | E igual     E
        | E mayor     E
        | E menor     E
        | E mayorIgual E
        | E menorIgual E
        | E mas       E
        | E menos     E
        | E multi     E
        | E divi      E
        | E modulo    E
        | E elevado   E
        | E punto     E
        ...
    '''
    if p[2].lower() == "or":
        p[0] = SOperacion(p[1], p[3], Logicas.OR)
    elif p[2].lower() == "and":
        p[0] = SOperacion(p[1], p[3], Logicas.AND)
    elif p[2] == "<>":
        p[0] = SOperacion(p[1], p[3], Relacionales.DIFERENTE)
    elif p[2] == "=":
        p[0] = SOperacion(p[1], p[3], Relacionales.IGUAL)
    elif p[2] == ">":
        p[0] = SOperacion(p[1], p[3], Relacionales.MAYOR_QUE)
    elif p[2] == "<":
        p[0] = SOperacion(p[1], p[3], Relacionales.MENOR_QUE)
    elif p[2] == ">=":
        p[0] = SOperacion(p[1], p[3], Relacionales.MAYORIGUAL_QUE)

```

**Nota:** Se muestra en la figura de como se construye el **AST** por medio de clases abstracta que heredan de sentencias y luego se manda a llamar en las acciones semanticas

La clase SOperacion es la que herera de la clase abstracta **Sentencia** recibe 3 parametros para obtener los valores.

```

class SOperacion(Sentencia):
    def __init__(self, opIzq, opDer, operador):
        self.opIzq = opIzq
        self.opDer = opDer
        self.operador = operador

```

Segunda forma es la realizacion de una simple clase Node

```

def p_E(p):
    ...
    E : E or E
      | E And E
      | E diferente E
      | E notEqual E
      | E igual E
      | E mayor E
      | E menor E
      | E mayorIgual E
      | E menorIgual E
      | E mas E
      | E menos E
      | E multi E
      | E divi E
      | E modulo E
      | E elevado E
      | E punto E
    ...

    global cont
    if p[2].lower() == "or":
        p[0] = Node("E", "", cont, 0, 0)
        cont = cont+1
        nodo1 = Node("or", p[2], cont, p.lineno(2), p.lexpos(2))
        cont = cont+1
        p[0].AddHijos(p[1])
        p[0].AddHijos(nodo1)
        p[0].AddHijos(p[3])
        lista.append(str(recorrerGramatica(p[0], 0)))

    elif p[2].lower() == "and":
        p[0] = Node("E", "", cont, 0, 0)
        cont = cont+1
        nodo1 = Node("and", p[2], cont, p.lineno(2), p.lexpos(2))
        cont = cont+1
        p[0].AddHijos(p[1])
        p[0].AddHijos(nodo1)
        p[0].AddHijos(p[3])
        lista.append(str(recorrerGramatica(p[0], 0)))

```

**Nota:** Se muestra en la figura de como se construye el **AST** por medio de clase simple llamado Node este es otra forma de como construirlo.

La clase Node contiene atributos para que reciban los valores como se muestra en la siguiente figura

```

class Node():
    def __init__(self, Etiqueta="", Valor="", idNod=0, Fila=0, Columna=0):
        self.Etiqueta=Etiqueta
        self.Valor=Valor
        self.idNod=idNod
        self.Fila=Fila
        self.Columna=Columna
        self.hijos=[]

    def AddHijos(self, son):
        self.hijos.append(son)

    def getHijos(self):
        return self.hijos

```

## Recuperación de Errores

A cada producción le asignamos una función de la cual contiene la producción de error, la cual descartará la entrada hasta encontrar el carácter siguiente. En este caso, tenemos el carácter de punto y coma con el cual esperamos que se recupere y continúe el análisis y la posterior ejecución.

También declaramos un nuevo error y lo guardamos en una lista para mostrarla en un reporte cuando sea requerido.



```
# FUNCIÓN PARA EL MANEJO DE LOS ERRORES SINTÁCTICOS
def p_error(t):
    col = find_column(t)
    # print("Error sintáctico en '%s'" % t.value, " Línea: '%s'" % str(t.lineno), " Columna: '%s'" % str(col) )
    errores_sintacticos.append(Error(t.value, 'Error Sintáctico', 'Símbolo no esperado', col, t.lineno))
```

## Precedencia de Operadores

La precedencia de operadores son niveles de jerarquias en una gramatica, en este caso se declaran precedencia de operadores debido a que, la gramatica son ambiguas, y cada herramienta tiene la opcion declarar precedencia como por ejemplo alguno analizadores que son **Irony**, **Jflex & Jcup**, **Flex & Bison**, **Jison**, el analizador ply ademas de los otros analizador tambien puede declarar precedencia como se muestra en la siguiente figura.

```
precedence = (
    ('right', 'not'),
    ('left', 'And'),
    ('left', 'or'),
    ('left', 'diferente', 'notEqual', 'igual', 'mayor', 'menor', 'menorIgual', 'mayorIgual'),
    ('left', 'punto'),
    ('right', 'umenos'),
    ('left', 'mas', 'menos'),
    ('left', 'elevado'),
    ('left', 'multi', 'divi', 'modulo'),
    ('nonassoc', 'parAbre', 'parCierra')
)
```

**Nota:** En este caso si una gramatica es ambigua en cierta operacion y no se declara precedencia el analizador puede causar error