UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

ESCUELA DE CIENCIAS Y SISTEMAS

ORGANIZACIÓN DE LENGUAJES Y COMPILADORES 2

ESCUELA DE VACACIONES DICIEMBRE 2020

ING. LUIS ESPINO

TUTOR ACADÉMICO: JUAN CARLOS MAEDA

FASE 2: BDTytus

Manual Técnico

Integrantes:

Nombres:	Carnes:
Brian Daniel Xiloj del Cid	201603037
Javier Estuardo Lima Abrego	201612098
Yaiza Estefanía Pineda González	201610673
César Alejandro Chinchilla González	201612132

Introducción

El siguiente manual explica el funcionamiento y flujo del código de la aplicación que es un sistema administrador de base de datos (DBMS) Tytus, donde se pueden ingresar las consultas en un área de texto.

Para la elaboración de la aplicación se utilizó el IDE Pycharm, como lenguaje de la aplicación Python, con la herramienta PLY para el analizador del lenguaje SQL basado en PostgreSQL.

Se escogió a utilizar la gramática ascendente por las siguientes razones:

1. La gramática descendente necesita atributos heredados, la herramienta PLY no es para analizadores descendentes entonces para resolver ese problema se tiene que utilizar una pila para llevar el control de los atributos, lo cual sería una implementación más.

2. Por lo prefijos comunes, si se cambiaba la gramática sería más difícil adaptarla al nuevo cambio.

3. La gramática ascendente es más fácil de implementar para generar un árbol AST que la descendente.

Descripción De Los Archivos Principales

Archivo main.py:

Es el archivo principal que ejecuta la aplicación, este despliega la interfaz gráfica de la aplicación y sus opciones para el funcionamiento de ella. Usa la librería tkinter para las partes gráficas y así simular el QueryTool.

main.py

```python
import Gramatica.Gramatica as g
import TablaSimbolos.TS as ts
import os
import graphviz
import sys
import threading
import Errores.Nodo_Error as error
import Errores.ListaErrores as lista_err
from Reportes.ReporteError import ReporteError
from Reportes.ReporteGramatical import ReporteGramatical
from tkinter import *
from tkinter import filedialog
from tkinter import scrolledtext
from tkinter import messagebox
from tkinter import ttk
import os, shutil


if __name__=='__main__':
    from LineNumber import LineMain
    from Graphics import Tkinter
    from ColorLight import ColorLight
else:
    from LineNumber import LineMain
    from Graphics import Tkinter
    from ColorLight import ColorLight

class Connect:
    def __init__(self, pad):
        self.pad = pad
        self.modules_connections()

    def modules_connections(self):
        LineMain(self.pad)
        ColorLight(self.pad)
        return
```

```python
class Interfaz:
    def __init__(self):
        os.environ['DB'] = 'None'
        self.borrarArchivos()
        self.root = Tk()
        #self.root.attributes('-zoomed', True)
        self.root.title('TytusDB - Team 19')
        self.root.geometry("1100x700")
        self.root.resizable(1, 1)
        self.txtarea = TextPad(self.root, pady=5)
        self.consola = scrolledtext.ScrolledText(self.root, width=300, height=15, font=("Consolas", 11))
        self.frame = ttk.Frame(self.root)
        self.frame.pack(fill=X, side='left')
        #self.tree_consultas = ttk.Treeview(self.frame)
        #self.tree_consultas["columns"] = ("Tipo", "Descripcion", "Fila", "Columna")
        #self.tree_consultas.grid(column=1, row=2, sticky=S)
        #self.tree_consultas.column("#0", width=600, stretch=NO)
        #self.tree_consultas.column("Tipo", width=50, minwidth=20, stretch=NO)
        #self.tree_consultas.column("Descripcion", width=50, minwidth=20)
        #self.tree_consultas.column("Fila", width=50, minwidth=20)
        #self.tree_consultas.column("Columna", width=50, minwidth=20)
        #self.tree_consultas.heading("#0", text="HOla", anchor=W)
        #self.tree_consultas.heading("Descripcion", text="Descripcion", anchor=W)
        #self.tree_consultas.heading("Tipo", text="Tipo", anchor=W)
        #self.tree_consultas.heading("Fila", text="Fila", anchor=W)
        #self.tree_consultas.heading("Columna", text="Columna", anchor=W)
        self.v = StringVar()
        lab = Label(self.root, textvariable=self.v, font=("Helvetica", 11, "bold"))
        self.v.set('Base de Datos: ' + os.environ['DB'])
        lab.pack(side=TOP, anchor=E)
        lbl = Label(self.root, text="Consola de Salida:", font=("Helvetica", 11))
        lbl.pack(side=TOP, anchor=W)
        #self.tree_consultas.pack(side=LEFT, fill=X, pady=5, padx=5)
        self.consola.pack(side=RIGHT)
        self.consola.config(foreground='white', bg='black')
        self.menubar = Menu(self.root)
        self.root.config(menu=self.menubar)
        archivo = Menu(self.menubar, tearoff=0)
```

Archivo Gramatica/Gramatica.py:

Es el archivo donde está escrita la gramática que reconoce el lenguaje SQL para nuestra aplicación. También es el que nos permite construir el en forma lógica el árbol AST y nuestros reportes de errores a través del análisis sintáctico.

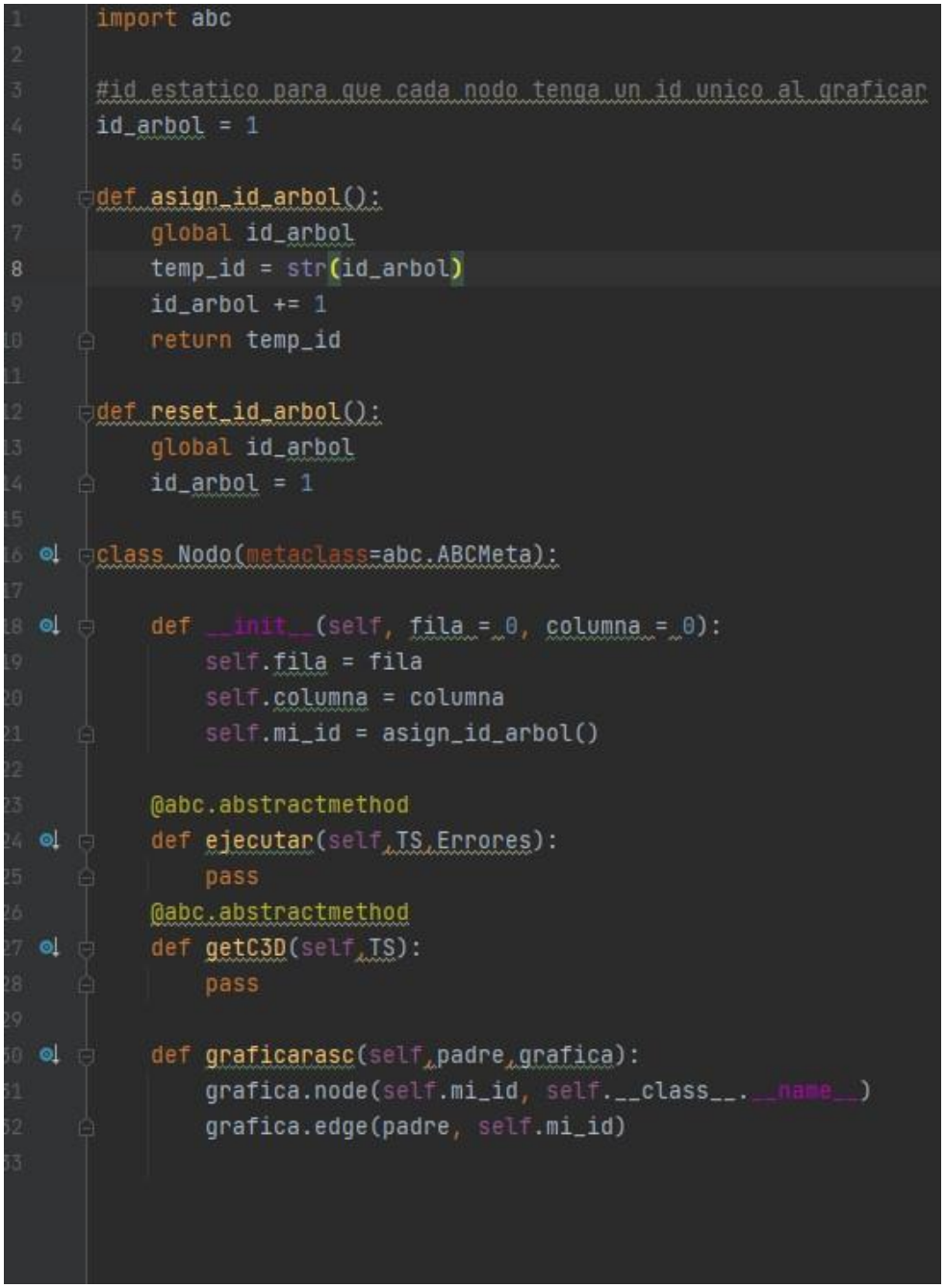
```
import Errores.Nodo_Error as err
from ply import lex
from AST.Sentencias import Raiz, Sentencia
import AST.SentenciasDDL as DDL
import ply.yacc as yacc

reservadas = {
    'select': 't_select',
    'distinct': 't_distinct',
    'as': 't_as',
    'from': 't_from',
    'where': 't_where',
    'having': 't_having',
    'avg': 't_avg',
    'min': 't_min',
    'max': 't_max',
    'sum': 't_sum',
    'count': 't_count',
    'insert': 't_insert',
    'into': 't_into',
    'values': 't_values',
    'delete': 't_delete',
    'update': 't_update',
    'true': 't_true',
    'false': 't_false',
    'not': 't_not',
    'and': 't_and',
    'or': 't_or',
    'smallint': 't_smallint',
    'integer': 't_integer',
    'bigint': 't_bigint',
    'decimal': 't_decimal',
    'numeric': 't_numeric',
    'real': 't_real',
    'double': 't_double',
    'precision': 't_precision',
    'money': 't_money',
    'character': 't_character',
    'varying': 't_varying',
    'varchar': 't_varchar',
```

```
from AST.Expresiones import *
from AST.SentenciasDML import *
reporteg = ''

def p_sql(p):
    'SQL : Sentencias_SQL'
    p[0] = Raiz(ListaErrores, p[1])

def p_sql2(p):
    'SQL : empty'
    p[0] = Raiz(ListaErrores)

def p_Sentencias_SQL_Sentencia_SQL(p):
    'Sentencias_SQL : Sentencias_SQL Sentencia_SQL'
    p[0] = p[1] + [p[2]]
    concatenar_gramatica('\n <TR><TD> SENTENCIAS_SQL ::= SENTENCIAS_SQL SENTENCIA_SQL </TD> <TD> { sen


def p_Sentencias_SQL(p):
    'Sentencias_SQL : Sentencia_SQL'
    p[0] = [p[1]]
    concatenar_gramatica('\n <TR><TD> SENTENCIAS_SQL ::= SENTENCIA_SQL </TD> <TD> { sentencias_sql.lis

def p_Sentencia_SQL_DML(p):
    'Sentencia_SQL : Sentencias_DML'
    p[0] = Sentencia("SentenciaDML", [p[1]])
    concatenar_gramatica('\n <TR><TD> SENTENCIA_SQL ::= SENTENCIAS_DML </TD> <TD> { sentencia_sql.inst

#def p_Sentencia_SQL_DML(p):
 #   'Sentencia_SQL : EXP pyc'
  #  p[0] = Sentencia("EXP", [p[1]])

def p_Sentencia_SQL_DDL(p):
    'Sentencia_SQL : Sentencias_DDL'
    p[0] = Sentencia("SentenciaDDL", [p[1]])
    concatenar_gramatica('\n <TR><TD> SENTENCIA_SQL ::= SENTENCIAS_DDL </TD> <TD> { sentencia_sql.inst

# ------------------------------------------------------------SENTENCIAS_DML
```

```
def p_Alias_Tabla(p):
    '''Alias_Tabla :  Listq_ID
                | Listq_Alias'''
    p[0] = p[1]
    concatenar_gramatica('\n <TR><TD> ALIAS_TABLA ::= ' + str(p[1]) + '</TD> <TD

def p_Subqueries(p):
    '''Subqueries : par1 t_select  par2'''
    concatenar_gramatica('\n <TR><TD> SUBQUERIES ::= ( select )</TD> <TD> { subq

# >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> INSERT <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
def p_Insert_SQL(p):
    'Insert_SQL : par1 Listq_ID par2 t_values par1 Listq_EXP par2'
    p[0] = {'col':p[2],'valores':p[6]}
    concatenar_gramatica('\n <TR><TD> INSERT_SQL ::= ( LISTA_ID ) values ( LISTA

def p_Insert_SQL2(p):
    'Insert_SQL : t_values par1 Listq_EXP par2'
    p[0] = {'col':None,'valores':p[3]}
    concatenar_gramatica('\n <TR><TD> INSERT_SQL ::= values ( LISTA_EXP ) </TD>

def p_Condiciones(p):
    '''Condiciones : t_where EXP
                | empty'''
    if len(p) == 3:
        p[0] = Where(p[2], p.slice[1].lineno, find_column(input, p.slice[1]))
        concatenar_gramatica('\n <TR><TD> CONDICIONES ::= where EXP  </TD>  <TD>
    else:
        p[0] = []
        concatenar_gramatica('\n <TR><TD> INSERT_SQL ::= EMPTY </TD> <TD> { inse

# ---------------------------- Group, having and order by --------------
def p_GBB(p):
```

Archivo Nodo.py:

Es la clase que ayuda a la construcción del árbol AST al archivo gramática.



```python
import abc

#id estatico para que cada nodo tenga un id unico al graficar
id_arbol = 1

def asign_id_arbol():
    global id_arbol
    temp_id = str(id_arbol)
    id_arbol += 1
    return temp_id

def reset_id_arbol():
    global id_arbol
    id_arbol = 1

class Nodo(metaclass=abc.ABCMeta):

    def __init__(self, fila = 0, columna = 0):
        self.fila = fila
        self.columna = columna
        self.mi_id = asign_id_arbol()

    @abc.abstractmethod
    def ejecutar(self,TS,Errores):
        pass
    @abc.abstractmethod
    def getC3D(self,TS):
        pass

    def graficarasc(self,padre,grafica):
        grafica.node(self.mi_id, self.__class__.__name__)
        grafica.edge(padre, self.mi_id)
```

Archivo Expresiones.py:

Es el archivo que recibe y resuelve las expresiones que vengan en el lenguaje SQL.

```python
import AST.Nodo as Node
import math as m
from TablaSimbolos.Tipos import *
from TablaSimbolos.TS import *
from Errores.Nodo_Error import *


class Expression(Node.Nodo):
    def __init__(self, *args):
        if len(args) == 6:
            if args[5] == 'math2':
                self.val1 = args[1]
                self.val2 = args[2]
                self.line = args[3]
                self.column = args[4]
                self.function = args[0]
                self.op_type = args[5]
            else:
                self.exp1 = args[0]
                self.exp2 = args[1]
                self.op = args[2]
                self.line = args[3]
                self.column = args[4]
                self.op_type = args[5]
                self.val = None
                self.type = None
        elif len(args) == 5:
            if args[4] == 'unario':
                self.op_type = args[4]
                self.type = args[0]
                self.val = args[1]
                self.line = args[2]
                self.column = args[3]
            elif args[4] == 'as':
                self.type = None
                self.val = args[0]
                self.asid = args[1]
                self.line = args[2]
                self.column = args[3]
                self.op_type = 'as'
            elif args[4] == 'aggregate':
```

```python
    def ejecutar(self, TS, Errores):
        if self.op_type == 'valor':
            return self
        elif self.op_type == 'unario':
            self.val.ejecutar(TS, Errores)
            if self.type == '-':
                self.val = -self.val.val
            return self
        elif self.op_type == 'as' or self.op_type == 'in' or self.op_type == 'agg':
            self.val.ejecutar(TS, Errores)
            self.asid.ejecutar(TS, Errores)
            return self
        elif self.op_type == 'math':
            if self.function == 'ceil' or self.function == 'ceiling':
                self.val.ejecutar(TS, Errores)
                if isinstance(self.val.val, int):
                    self.val = m.__ceil__(self.val.val)
                else:
                    self.val = m.ceil(self.val.val)
            elif self.function == 'abs':
                self.val = m.fabs(self.val.val)
            elif self.function == 'cbrt':
                self.val = m.ceil(self.val.val**(1/3))
            elif self.function == 'degrees':
                self.val = m.degrees(self.val.val)
            elif self.function == 'div':
                self.val = m.exp(self.val.val)
            elif self.function == 'exp':
                self.val = m.exp(self.val.val)
            elif self.function == 'factorial':
                self.val = m.factorial(self.val.val)
            elif self.function == 'floor':
                self.val = m.floor(self.val.val)
            elif self.function == 'gcd':
                self.val = m.gcd(self.val.val)
```

Archivo Sentencias.py:

Construye gráficamente el árbol AST además de ejecutar de forma general las instrucciones.

```python
from graphviz import Digraph
from AST.Nodo import Nodo
from AST.Nodo import reset_id_arbol
import AST.Nodo as Node
from AST.Expresiones import *


class Raiz(Node.Nodo):
    def __init__(self, Errores, sentencias = [], fila = 0, columna = 0):
        super().__init__(fila=fila, columna=columna)
        self.sentencias = sentencias
        self.errores = Errores

    def ejecutar(self, TS, Errores):
        respuesta = ''
        for hijo in self.sentencias:
            if isinstance(hijo, Expression):
                respuesta += ''
            respuesta += hijo.ejecutar(TS, Errores) + '\n'
        return respuesta

    def getC3D(self, TS):
        pass

    def graficarasc(self, padre = None, grafica=None):
        grafica = Digraph(name="AST", comment='AST generado')
        grafica.edge_attr.update(arrowhead='none')
        grafica.node(self.mi_id, "SentenciasSQL")
        for hijo in self.sentencias:
            hijo.graficarasc(self.mi_id, grafica)
        grafica.render(directory='Reportes/graficaAST', view=True)
        reset_id_arbol()


class Sentencia(Raiz):
    def __init__(self, nombre_sentencia, sentencias, fila = 0, columna = 0):
        super().__init__(None, sentencias=sentencias, fila=fila, columna=columna)
        self.nombre_sentencia = nombre_sentencia

    #No necesito definir ejecutar porque ya lo hereda de Raiz y realiza lo mismo aqui que alli
```

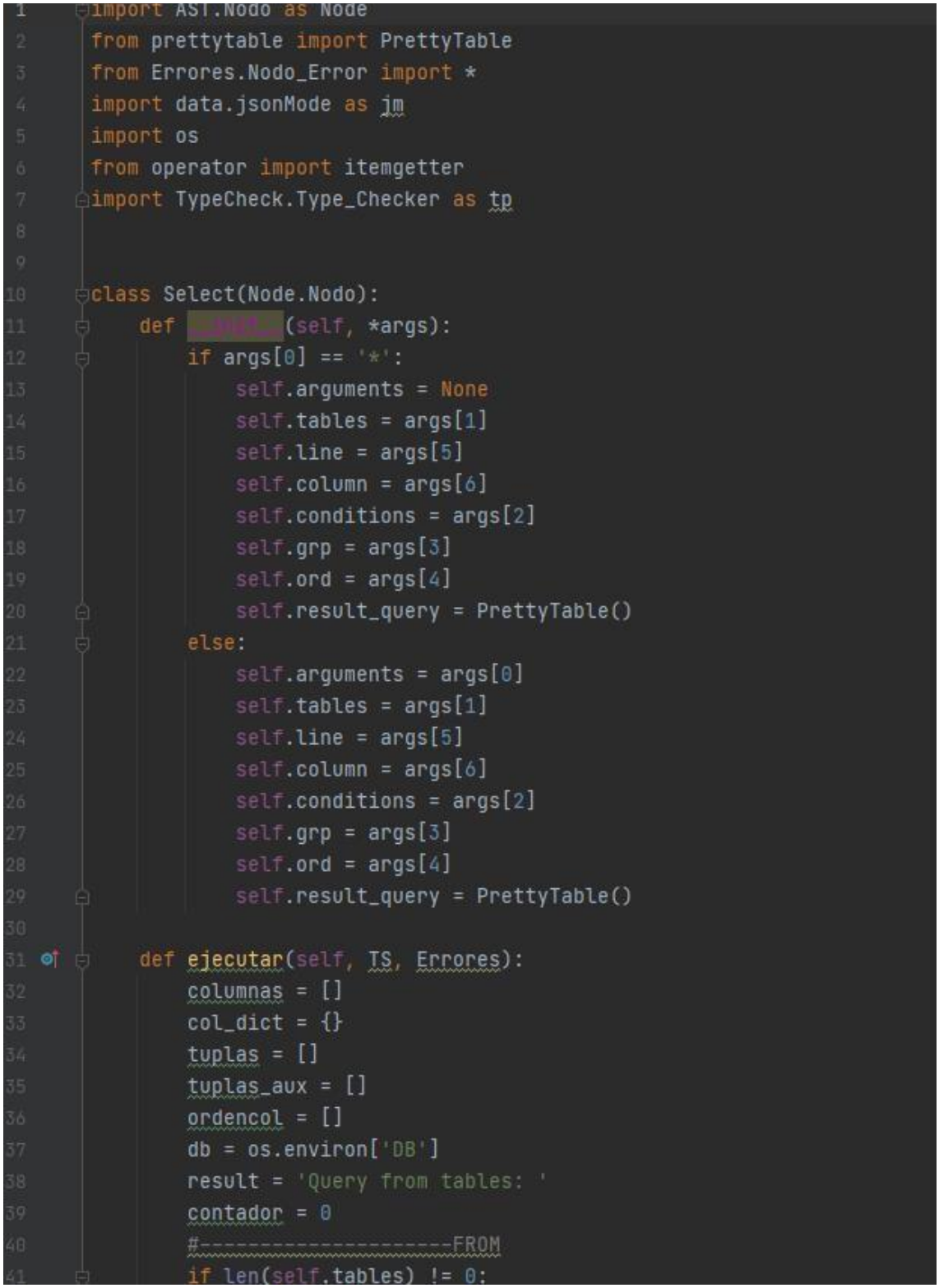
Archivo SentenciasDDL.py:

Ejecuta las instrucciones DDL del lenguaje SQL de la aplicación, como créate table y alterTable.

```python
from typing import Type
from AST.Nodo import Nodo
import data.jsonMode as JM
import Errores.Nodo_Error as err
from prettytable import PrettyTable
import TypeCheck.Type_Checker as TypeChecker
import os


class CreateDatabase(Nodo):
    def __init__(self, fila, columna, nombre_BD, or_replace_ = False, if_not_exist_=False,
        super().__init__(fila, columna)
        self.nombre_DB = nombre_BD.lower()
        self.or_replace = or_replace
        self.if_not_exist = if_not_exist
        self.owner = owner
        self.mode = mode

    def ejecutar(self,TS,Errores):#No se toca el owner para esta fase
        if self.if_not_exist:
            if self.nombre_DB in TypeChecker.showDataBases():
                Errores.insertar(err.Nodo_Error('42P04', 'duplicated database', self.fila
                return '42P04: duplicated database\n'

        if self.or_replace:
            respuesta = TypeChecker.dropDataBase(self.nombre_DB)
            if respuesta == 1:
                Errores.insertar(err.Nodo_Error('XX000', 'internal_error', self.fila, sel
                return 'XX000: internal_error\n'

        if self.mode > 5 or self.mode < 1:
            Errores.insertar(err.Nodo_Error('Semantico', 'El modo debe estar entre 1 y 5'
            return 'Error semantico: El modo debe estar entre 1 y 5\n'

        respuesta = TypeChecker.createDataBase(self.nombre_DB, self.mode, self.owner)
        if respuesta == 2:
            Errores.insertar(err.Nodo_Error('42P04', 'duplicated database', self.fila, se
            return '42P04: duplicated database\n'
        if respuesta == 1:
            Errores.insertar(err.Nodo_Error('P0000', 'plpgsql_error', self.fila, self.col
            return 'P0000: plpgsql_error\n'
```

Archivo SentenciasDML.py:

Ejecuta las instrucciones DML del lenguaje SQL de nuestra aplicación, como la instrucción select.



```
import AST.Nodo as Node
from prettytable import PrettyTable
from Errores.Nodo_Error import *
import data.jsonMode as jm
import os
from operator import itemgetter
import TypeCheck.Type_Checker as tp


class Select(Node.Nodo):
    def __init__(self, *args):
        if args[0] == '*':
            self.arguments = None
            self.tables = args[1]
            self.line = args[5]
            self.column = args[6]
            self.conditions = args[2]
            self.grp = args[3]
            self.ord = args[4]
            self.result_query = PrettyTable()
        else:
            self.arguments = args[0]
            self.tables = args[1]
            self.line = args[5]
            self.column = args[6]
            self.conditions = args[2]
            self.grp = args[3]
            self.ord = args[4]
            self.result_query = PrettyTable()

    def ejecutar(self, TS, Errores):
        columnas = []
        col_dict = {}
        tuplas = []
        tuplas_aux = []
        ordencol = []
        db = os.environ['DB']
        result = 'Query from tables: '
        contador = 0
        #--------------------FROM
        if len(self.tables) != 0:
```

Archivo TypeChecker.py:

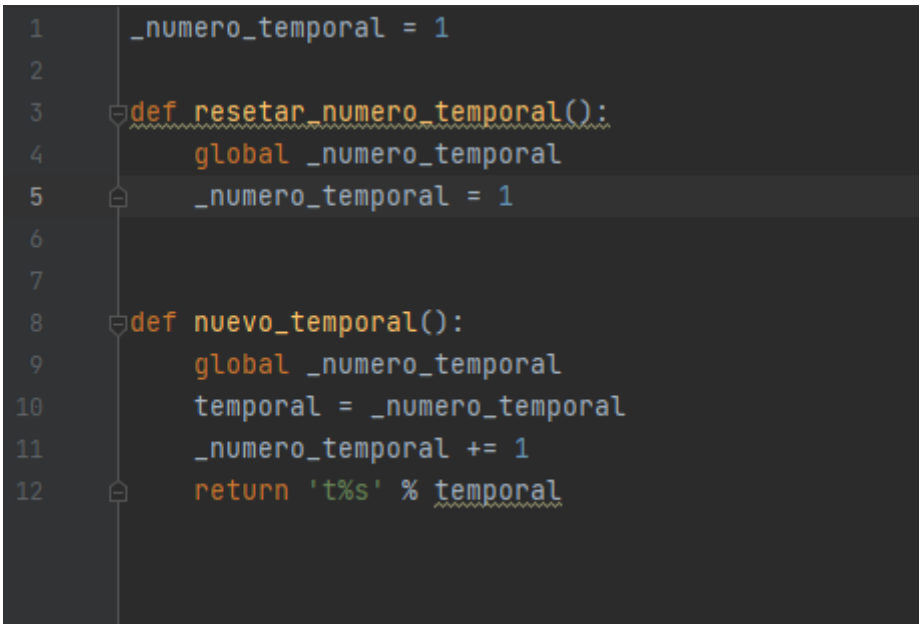
Es el archivo que realiza la funcionalidad del TypeChecker en nuestra aplicación, es decir el que permite verificar todos los tipos de datos de las Tablas y que su almacenamiento de todos los objetos sea correcto.

```
50
51  def dropDataBase(database: str):
52      # 0:operación exitosa, 1: error en la operación, 2: base de datos no existente
53      respuesta = JM.dropDatabase(database)
54      if respuesta == 0:
55          return lista_bases.eliminarBaseDatos(database)
56      return respuesta
57
58  def obtenerBase(database: str):
59      actual = lista_bases.primero
60      while(actual != None):
61          if actual.nombreBase == database:
62              break
63          actual = actual.siguiente
64      return actual
65
66  def createTable(database: str, table: str, numberColumns: int):
67      # 0:operación exitosa, 1: error en la operación, 2: base inexistente, 3: tabla existente
68      respuesta = JM.createTable(database,table,numberColumns)
69      if respuesta == 0:
70          actual = obtenerBase(database)
71          if(actual != None):
72              if not actual.listaTablas.existeTabla(table):
73                  actual.listaTablas.agregarTabla(Tabla.Tabla(table))
74                  return 0
75              else:
76                  return 3
77          else:
78              return 2
79      return respuesta
80
81  def showTables(database:str):
82      respuesta = JM.showTables(database)
83      return respuesta
84
85  def createColumn(database:str,table:str,nombre:str,tipo):
86      # 0:operación exitosa, 1: error en la operación, 2: base de datos inexistente, 3: tabla inexistente, 4: columna ya exist
87      actualBase = obtenerBase(database)
88      if(actualBase!=None):
89          if not actualBase.listaTablas.existeTabla(table):
90              return 3
```

FASE 2

GeneradorTemporales.py:

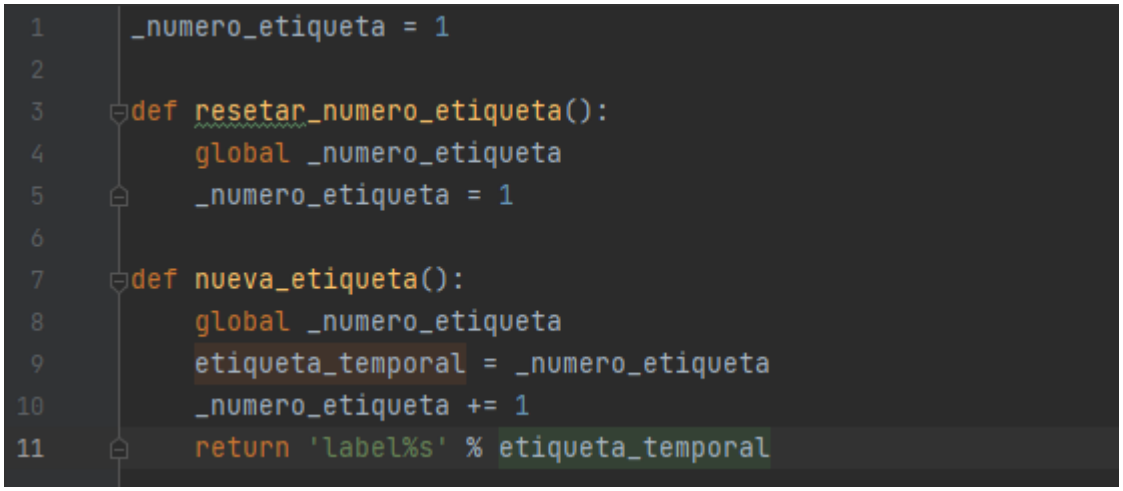
Clase que ayuda a generar nuevos temporales



```
_numero_temporal = 1

def resetar_numero_temporal():
    global _numero_temporal
    _numero_temporal = 1


def nuevo_temporal():
    global _numero_temporal
    temporal = _numero_temporal
    _numero_temporal += 1
    return 't%s' % temporal
```

GeneradorEtiquetas.py

Esta clase sirve para generar nuevas etiquetas



```
_numero_etiqueta = 1

def resetar_numero_etiqueta():
    global _numero_etiqueta
    _numero_etiqueta = 1

def nueva_etiqueta():
    global _numero_etiqueta
    etiqueta_temporal = _numero_etiqueta
    _numero_etiqueta += 1
    return 'label%s' % etiqueta_temporal
```

GeneradorFileC3D.py

Esta clase es la encargada de generar el archivo en c3d, el cual se ejecutar mediante el compilador de python

```python
from tkinter import messagebox

funciones_extra = ""


class GeneradorFileC3D:
    def __init__(self):
        self.path_archivo_c3d = '../interfaz/c3d.py'
        self._crea_archivo_c3d()

    def _crea_archivo_c3d(self):
        with open(self.path_archivo_c3d, 'w') as file_c3d:
            "Creo el archivo en tu computadora si aun no lo tienes"

    def escribir_archivo(self, c3d):
        global funciones_extra
        imports = '''
#Imports
from goto import with_goto
from Analisis_Ascendente.storageManager.jsonMode import *
import Analisis_Ascendente.ascendente as parser
'''
        variables_globales = '''
#Variables Globales
salida = ''
stack = [None] * 1000
top_stack = -1
```

```python
if __name__ == "__main__":
    main()
'''
        try:
            with open(self.path_archivo_c3d, 'w') as file_c3d:
                file_c3d.write(imports)
                file_c3d.write(variables_globales)
                file_c3d.write(funcion_intermedia)
                file_c3d.write(funciones_extra)
                file_c3d.write(inicio_main)
                file_c3d.write(c3d)
                file_c3d.write(fin_main)
            funciones_extra = ''

        except Exception as er:
            messagebox.showwarning(er, "No existe archivo para guardar la informacion")
```

CreateFunction.py

Esta clase representa un nodo del árbol sintáctico generado al recorrer los esquemas de traducción postfijos, es la clase padre de un función la cual se encarga de ejecutar la información dentro

```python
from Analisis_Ascendente.Instrucciones.instruccion import Instruccion
import Analisis_Ascendente.Tabla_simbolos.TablaSimbolos as TS

class Parametro():
    def __init__(self,id,tipo):
        self.id = id
        self.tipo = tipo

class CreateFunction(Instruccion):
    def __init__(self,id,parametros,returns,declare,begin):
        self.id = id
        self.parametros = parametros            #[Parametro(id,tipo),...]
        self.returns = returns.tipo             #returns = Tipo(tipo,longitud,fila,columna)
        self.declare = declare
        self.begin = begin

    def ejecutar(CreateFunction,ts,consola,exceptions):
        bdactual = ts.buscar_sim("usedatabase1234")
        BD = ts.buscar_sim(bdactual.valor)
        entornoBD = BD.Entorno
        try:
            paramcorrectos = True
            if entornoBD.validar_sim(CreateFunction.id) == -1:
                dicci = {}
                for parametro in CreateFunction.parametros:
                    existe = dicci.get(parametro.id,False)
                    if existe == False:
                        dicci[parametro.id] = parametro.tipo.tipo
                    else:
                        consola.append(f"Existe parametros con el mismo nombre en la función: {CreateFunction.id}\n No se pudo crear la función.")
                        paramcorrectos = False
                        break
                if paramcorrectos:
                    entornoFuncion = TS.TablaDeSimbolos({})
                    if CreateFunction.declare != None:
                        simdeclare = TS.Simbolo(TS.TIPO_DATO.DECLARE, "DECLARE", None, CreateFunction.declare, None)
                        entornoFuncion.agregar_sim(simdeclare)
                    #Esto siempre se realiza
                    simbegin = TS.Simbolo(TS.TIPO_DATO.BEGIN, "BEGIN", None, CreateFunction.begin, None)
                    entornoFuncion.agregar_sim(simbegin)
                    #Agregamos las variables de parametros al entorno de la función de una vez
                    for clave,valor in dicci.items():   #id,tipo
                        paramcorrectos = False
                        break
                if paramcorrectos:
                    entornoFuncion = TS.TablaDeSimbolos({})
                    if CreateFunction.declare != None:
                        simdeclare = TS.Simbolo(TS.TIPO_DATO.DECLARE, "DECLARE", None, CreateFunction.declare, None)
                        entornoFuncion.agregar_sim(simdeclare)
                    #Esto siempre se realiza
                    simbegin = TS.Simbolo(TS.TIPO_DATO.BEGIN, "BEGIN", None, CreateFunction.begin, None)
                    entornoFuncion.agregar_sim(simbegin)
                    #Agregamos las variables de parametros al entorno de la función de una vez
                    for clave,valor in dicci.items():   #id,tipo
                        nuevaVariable = TS.Simbolo(TS.TIPO_DATO.PARAMETRO,clave,valor,None,None)
                        entornoFuncion.agregar_sim(nuevaVariable)

                    #Creamos la nuevafuncion en el entorno global de la base de datos
                    nuevaFuncion = TS.Simbolo(TS.TIPO_DATO.FUNCTION,CreateFunction.id,CreateFunction.returns,None,entornoFuncion)
                    entornoBD.agregar_sim(nuevaFuncion)
                    consola.append(f"Se añadio una nueva funcion llamada: {CreateFunction.id}")
            else:
                consola.append(f"Ya existe esta función en la base de datos")

        except:
            consola.append("XX000 : internal_error")
```

CreateProcedure.py

Clase que representa un nodo del árbol sintáctico generado al recorrer los esquemas de traducción postfijos, es la clase padre de un stored procedure el cual se encarga de ejecutar la información.

```python
from Analisis_Ascendente.Instrucciones.instruccion import Instruccion
import Analisis_Ascendente.Tabla_simbolos.TablaSimbolos as TS
import C3D.GeneradorTemporales as GeneradorTemporales
import Analisis_Ascendente.reportes.Reportes as Reportes

class Parametro():
    def __init__(self,id,tipo):
        self.id = id
        self.tipo = tipo

class CreateProcedure(Instruccion):

    def __init__(self, id, parametros,declare,sentencias, fila, columna):
        self.id = id
        self.parametros = parametros
        self.declare = declare
        self.sentencias = sentencias
        self.fila = fila
        self.columna = columna

    def ejecutar(self, ts, consola, exceptions):
        bdactual = ts.buscar_sim("usedatabase1234")
        BD = ts.buscar_sim(bdactual.valor)
        entornoBD = BD.Entorno


        if self.parametros == None:
            if entornoBD.validar_sim(self.id) == -1:
                entornoP = TS.TablaDeSimbolos({})

                if self.declare != None:

                    simdeclare = TS.Simbolo(TS.TIPO_DATO.DECLARE, "DECLARE", None, self.declare, None)
                    entornoP.agregar_sim(simdeclare)

                print(self.sentencias)
                simbegin = TS.Simbolo(TS.TIPO_DATO.BEGIN, "BEGIN", None, self.sentencias, None)
                entornoP.agregar_sim(simbegin)

                nuevoP = TS.Simbolo(TS.TIPO_DATO.PROCEDURE, self.id, None, None, entornoP)
                entornoBD.agregar_sim(nuevoP)
                consola.append(f"Se añadio un nuevo procedimiento : {self.id}")
        else:
            paramcorrectos = True
            if entornoBD.validar_sim(self.id) == -1:
                dicci = {}
                for parametro in self.parametros:
                    existe = dicci.get(parametro.id, False)
                    if existe == False:
                        dicci[parametro.id] = parametro.tipo.tipo
                    else:
                        consola.append(
                            f"Existe parametros con el mismo nombre en la función: {self.id}\n No se pudo crear la función.")
                        paramcorrectos = False
                        break
                if paramcorrectos:
                    entornoP = TS.TablaDeSimbolos({})
                    if self.declare != None:
                        simdeclare = TS.Simbolo(TS.TIPO_DATO.DECLARE, "DECLARE", None, self.declare, None)
                        entornoP.agregar_sim(simdeclare)
                    simbegin = TS.Simbolo(TS.TIPO_DATO.BEGIN, "BEGIN", None, self.sentencias, None)
                    entornoP.agregar_sim(simbegin)
                    for clave, valor in dicci.items():  # id,tipo
                        nuevaVariable = TS.Simbolo(TS.TIPO_DATO.PARAMETRO, clave, valor, None, None)
                        entornoP.agregar_sim(nuevaVariable)

                    nuevoP = TS.Simbolo(TS.TIPO_DATO.PROCEDURE, self.id, None, None,
                                        entornoP)
                    entornoBD.agregar_sim(nuevoP)
                    consola.append(f"Se añadio una nuevo procedimiento : {self.id}")
                else:
                    consola.append(f"Ya existe el procedimiento ")
```

Index.py

Esta clase representa un nodo del árbol sintáctico generado al recorrer los esquemas de traducción postfijos, es la clase padre de un índice la cual se encarga de ejecutar la información dentro

```python
from Analisis_Ascendente.Instrucciones.instruccion import Instruccion
import Analisis_Ascendente.Tabla_simbolos.TablaSimbolos as TS
import C3D.GeneradorTemporales as GeneradorTemporales
import Analisis_Ascendente.reportes.Reportes as Reportes

class Index(Instruccion):
    ''' #1 Index normal
        #2 Index hash
        #3 Inder Unique
        #4 Index order
        #5 Index Lower '''

    def __init__(self, caso,id, tabla, columnref, where,order, fila, columna):
        self.caso = caso
        self.id = id
        self.tabla = tabla
        self.columnref = columnref
        self.where = where
        self.order = order
        self.fila = fila
        self.columna = columna

    def ejecutar(Index, ts, consola, exceptions):
        bdactual = ts.buscar_sim("usedatabase1234")
        BD = ts.buscar_sim(bdactual.valor)
        entornoBD = BD.Entorno

        #print(Index.caso)
        listaId = []
        try:
            if entornoBD.validar_sim(Index.tabla) == 1:

                if Index.caso == 1:
                    for idcito in Index.columnref:
                        #print(idcito.id)
                        listaId.append(idcito.id)
                        #print(str(listaId))
                    sim = TS.Simbolo(TS.TIPO_DATO.INDEX_SIMPLE,Index.id,None,str(listaId)[1:-1],None)
                    entornoBD.agregar_sim(sim)
                elif Index.caso == 2:
                    for idcito in Index.columnref:
                        #print(idcito.id)
```

```python
                instruccion_quemada += '%s ' % idcito.id + ','
            instruccion_quemada = instruccion_quemada[:-1]
            instruccion_quemada += ');'
        elif self.caso == 2:
            instruccion_quemada += ' index %s ' % self.id + 'on %s ' % self.tabla
            instruccion_quemada += ' using hash ( '
            for idcito in self.columnref:
                instruccion_quemada += '%s ' % idcito.id + ','
            instruccion_quemada = instruccion_quemada[:-1]
            instruccion_quemada += ');'
        elif self.caso == 3:
            instruccion_quemada += ' unique index %s ' % self.id + 'on %s ' % self.tabla + '('
            for idcito in self.columnref:
                instruccion_quemada += '%s ' % idcito.id + ','
            instruccion_quemada = instruccion_quemada[:-1]
            instruccion_quemada += ');'
        elif self.caso == 4:
            instruccion_quemada += ' index %s ' % self.id + 'on %s ' % self.tabla + '( %s ' % self.columnref + ' %s' % self.order + ');'
        elif self.caso == 5:
            instruccion_quemada += ' index %s ' % self.id + 'on %s ' % self.tabla + '( lower ( %s ' % self.columnref + ') );'

        c3d = '''
    # --------- INDEX -----------
    top_stack = top_stack + 1
    %s = "%s"
    stack[top_stack] = %s

    ''' % (etiqueta, instruccion_quemada, etiqueta)

        optimizacion1 = Reportes.ListaOptimizacion("c3d original", "c3d que entra",
```

DropProcedure.py
Esta clase contiene la estructura de un drop procedure y se encarga de generar el c3d el cual manda a eliminar la función que llama al stored procedure en el c3d.

```python
from Analisis_Ascendente.Instrucciones.instruccion import Instruccion
import Analisis_Ascendente.Tabla_simbolos.TablaSimbolos as TS
import C3D.GeneradorTemporales as GeneradorTemporales
import Analisis_Ascendente.reportes.Reportes as Reportes

class DropProcedure(Instruccion):

    def __init__(self, id, fila, columna):
        self.id = id
        self.fila = fila
        self.columna = columna

    def ejecutar(dropObject, ts, consola, exceptions):
        bdactual = ts.buscar_sim("usedatabase1234")
        # se busca el simbolo y por lo tanto se pide el entorno de la bd
        BD = ts.buscar_sim(bdactual.valor)
        entornoBD = BD.Entorno

        if entornoBD.validar_sim(dropObject.id) == 1:

            entornoBD.eliminar_sim(dropObject.id)

            consola.append(f"Procedure {dropObject.id}, eliminado exitosamente")

        else:

            consola.append(f"42P01  undefined_procedure, no existe {dropObject.id}")
            exceptions.append(
                f"Error semantico-42P01- 42P01  undefined_procedure, no existe {dropObject.id}-fila-columna")

    def getC3D(self, lista_optimizaciones_C3D):

        etiqueta = GeneradorTemporales.nuevo_temporal()
        instruccion_quemada = 'drop procedure '
        instruccion_quemada += '%s ' % self.id
        instruccion_quemada += ';'
        c3d = '''
    # ---------Drop Procedure----------
    top_stack = top_stack + 1
    %s = "%s"
    stack[top_stack] = %s
```

DropFunction.py
Esta clase elimina la función contenida en la tabla de símbolos

```python
from Analisis_Ascendente.Instrucciones.instruccion import Instruccion

class DropFunction(Instruccion):
    def __init__(self,id):
        self.id = id


    def ejecutar(DropFunction,ts,consola,exception):
        bdactual = ts.buscar_sim("usedatabase1234")
        BD = ts.buscar_sim(bdactual.valor)
        entornoBD = BD.Entorno
        try:
            if entornoBD.validar_sim(DropFunction.id) == 1:     #verificamos si existe la función en la bd
                entornoBD.eliminar_sim(DropFunction.id)
                consola.append(f"Se elimino la funcion {DropFunction.id} exitosamente")
            else:
                consola.append(f" No existe la función {DropFunction.id} para eliminar")
        except:
            consola.append("XX000 : internal_error")
```

If.py
Esta clase es una de las sentencias que pueden venir dentro del código de c3d

```python
from Analisis_Ascendente.Instrucciones.expresion import *
from Analisis_Ascendente.Instrucciones.instruccion import *


class Ifpl(Instruccion):
    ''' #1 If
        #2 If elif else
        #3 If else '''
    global consola
    global exceptions

    def __init__(self, caso,e_if,s_if,elif_s,s_else, fila, columna):
        self.caso = caso
        self.e_if = e_if
        self.s_if = s_if
        self.elif_s = elif_s
        self.s_else = s_else
        self.fila = fila
        self.columna = columna

    def ejecutar(If, ts, consola, exceptions):
            if If.caso == 1:
                resultado = Expresion.Resolver(If.e_if, ts, consola, exceptions)
                if resultado == True:
                    for x in range(0, len(If.s_if)):
                       print(If.s_if[x])
                        #ejecutar sentencias
                else:
                    pass
            elif If.caso == 2:
                print('hola')
            else:
                resultado = Expresion.Resolver(If.e_if, ts, consola, exceptions)
                if resultado == True:
                    print('state')
                else:
                    print('state')
                    print('else')
                    print(If.e_if)
                    print(If.s_if)
                    print(If.s_else)
                    print(If.caso)
```