# **INTRODUCCIÓN**

El estudio de compiladores, en el contexto de sistemas de procesamiento de lenguaje juega un papel fundamental en la comprensión y manipulación efectiva de estructuras de lenguajes de programación. Estudiarlo, nos permite comprender cuales son las herramientas y reglas utilizadas por los compiladores e intérpretes en el procesamiento de lenguajes.

La principal razón para la realización del proyecto es poner en práctica los conocimientos teóricos adquiridos en el curso de compiladores 2. Para esto, se realizó un intérprete para un lenguaje llamado X-SQL; utilizando la herramienta para generación de analizadores léxicos y sintácticos, PLY de Python, mediante la cual se definieron las reglas del lenguaje.

A lo largo de este manual, se detallan algunas de las metodologías, herramientas utilizadas y las principales funcionalidades del sistema que proporcionan una guía detallada que no solo documenta el proceso de construcción del intérprete, sino que también sirve como recurso valioso para aquellos interesados en comprender las complejidades y desafíos asociados con el desarrollo de sistemas de procesamiento de lenguaje.

# **DATOS TÉCNICOS**

IDE utilizado: Visual Studio Code 1.66.1

Sistema operativo: Windows 10

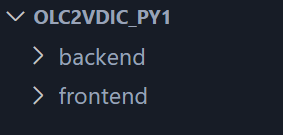
Lenguaje utilizado para backend: Python

Lenguaje utilizado para frontend: Javascript (React)

# **EXPLICACIÓN DEL CÓDIGO**

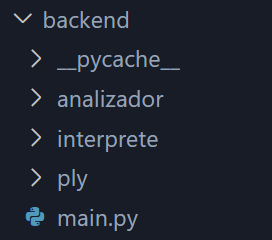
## **Estructura del proyecto**

La estructura general del proyecto se compone de dos carpetas: *backend* y *frontend.*



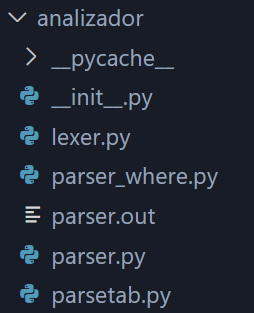
### **Backend**

Dentro de esta carpeta se encuentran las siguientes subcarpetas:



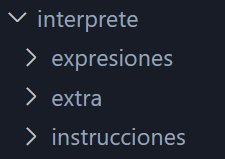
#### **Analizador**

En esta carpeta se definen los analizadores léxico y sintáctico.



#### **Interprete**

Dentro de esta carpeta se especifican las subcarpetas que serán utilizadas para funcionalidades como tal del programa.

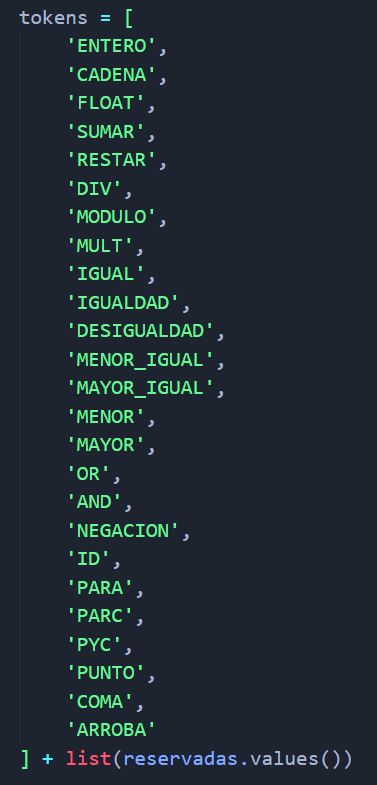


# **Analizador Léxico**

Para realizar el analizador léxico se utilizó la herramienta PLY, definiendo un archivo dentro de la carpeta *backend/analizador/lexer.py*. La tarea principal es dividir el código fuente en unidades más pequeñas llamadas tokens. Cada token representa una entidad léxica única, como una palabra clave, un identificador, un número o un operador.

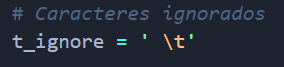
La estructura del archivo del analizador léxico es la siguiente:

1. Definición de tokens: lista de tokens que el analizador léxico debe reconocer (incluyen palabras clave, operadores, identificadores, números, etc.).

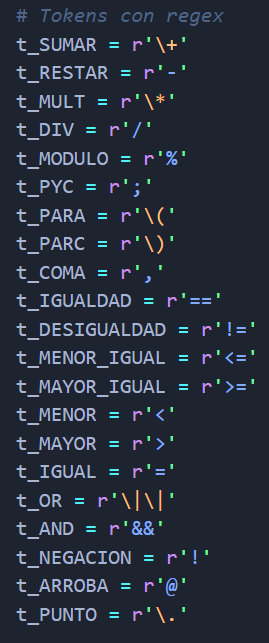
Como se puede ver en las imágenes anteriores, primero se especifican las palabras reservas utilizadas mediante su *expresion\_regular : nombre\_token*, mientas que, despues se especifican unicamente los nombres de los tokens, ya que estos se definirán posteriormente, y estos se unen a la tabla definida anteriormente.

1. Caracteres ignorados: cuando el analizador léxico encuentre cualquiera de los caracteres especificados en ‘t\_ignore’ los ignorará.

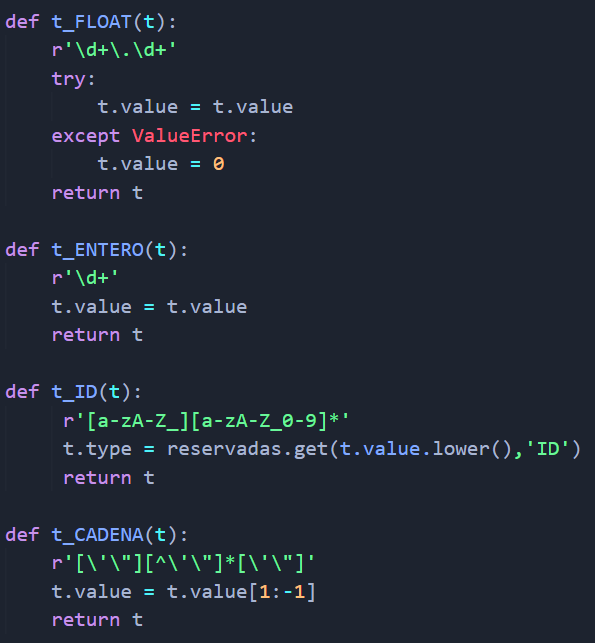


1. Expresiones Regulares para Tokens: Como se mencionó anteriormente, hay nombres de tokens a los cuales hay que asociarle su expresión regular. Ply permite especificarlo ‘dividiendolos’ en:

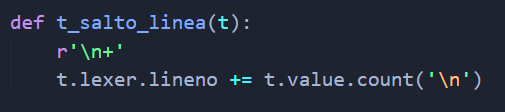
Expresiones regulares simples: no retornan valor del lexema.



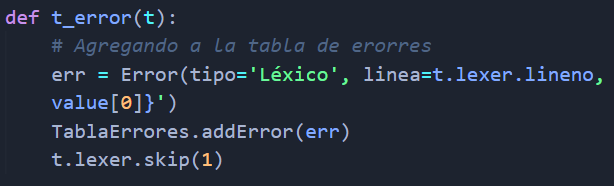
Expresiones regulares complejas: retornan el valor del lexema.



1. Registro del número de líneas: cuando el analizador léxico encuentra un carácter de salto de línea (\n), este incrementa el contador ‘lineno’.



1. Manejo de errores léxicos: cada vez que el analizador léxico encuentra un carácter no definido anteriormente, este entra a función ‘t\_error’ y lo agrega a una tabla de errores.



1. Creación del analizador léxico: Creación del objeto lexer utilizando ply.lex.lex().

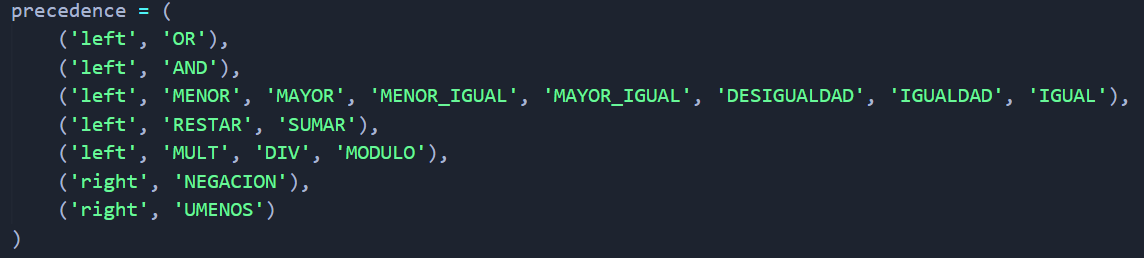


# **Analizador Sintáctico**

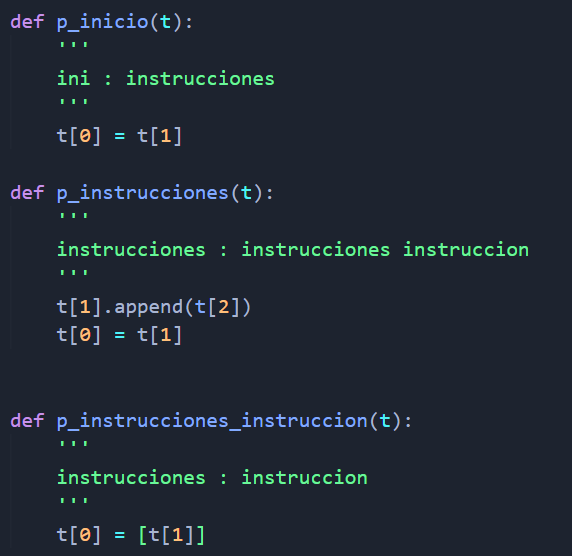
Para el realizar el analizador sintáctico se utilizó la herramienta PLY, definiendo un archivo dentro de la carpeta backend/analizador/parser.py. Se encarga de analizar la estructura gramatical de un programa fuente de acuerdo con las reglas definidas por la gramática del lenguaje de programación.

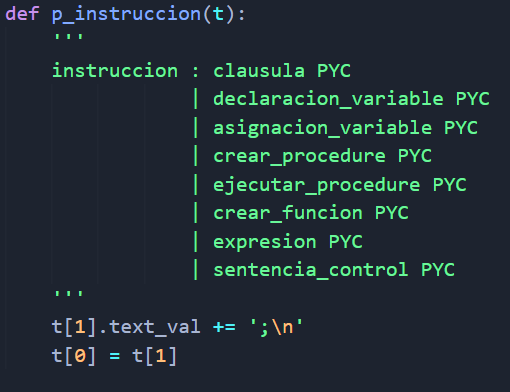
La estructura del archivo del analizador sintáctico es la siguiente:

1. Definición de precedencia y asociatividad de operadores.

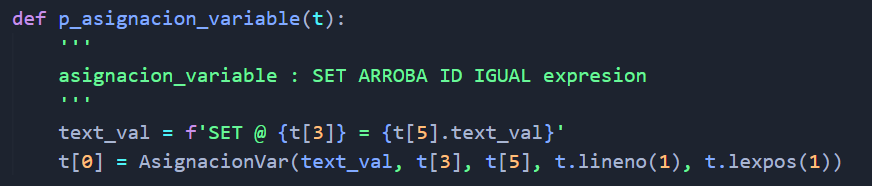


1. Definición de la gramática: utiliza la sintaxis de las reglas de producción de la gramática, cada regla define cómo se construye una parte del lenguaje.



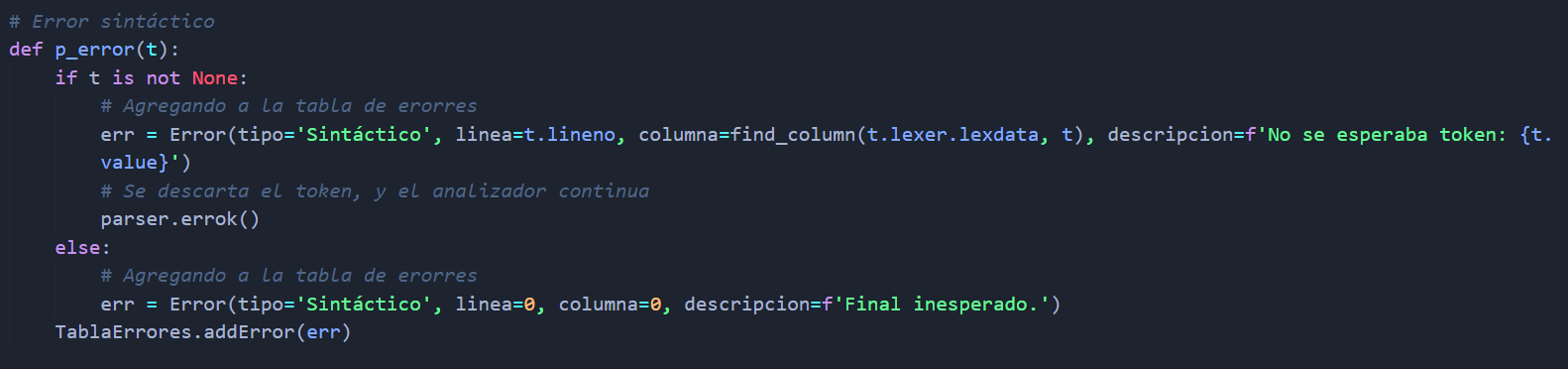


Como se puede ver, algunas producciones cuentan con un atributo llamado ‘text\_val’, esto, como se verá posteriormente, sirve para guardar la cadena de texto asociada a cada instrucción, esto principalmente ayuda al manejo de funciones y procedimientos almacenados.



Por ejemplo, en la imagen anterior se puede ver la definición sintáctica para una asignacion de una variable. El analizador, al encontrar esta estructura, ingresa a la función ‘p\_asignacion\_variable’, crea el atributo asociado a la producción ‘text\_val’ (que es la cadena de texto asociada a la producción) y luego crea un objeto de la clase ‘AsignacionVar’, el cual recibe el ‘text\_val’, el identificador de la variable, el valor de la expresión, la línea y la columna. Y esto es igual todas las reglas sintácticas.

1. Manejo de errores sintácticos: cada vez que el parser encuentra un error sintáctico, ingresa a la funcion ‘p\_error’. Cuando el valor del token incorrecto ‘t’ no es None, agrega el error a la tabla de errores, descarta el token y continua con el análisis sintáctico. Pero cuando el valor del token es None, significa que el parser no puede recuperarse del mismo, por lo que simplemente lo agrega a la tabla de errores y termina con el análisis sintáctico.

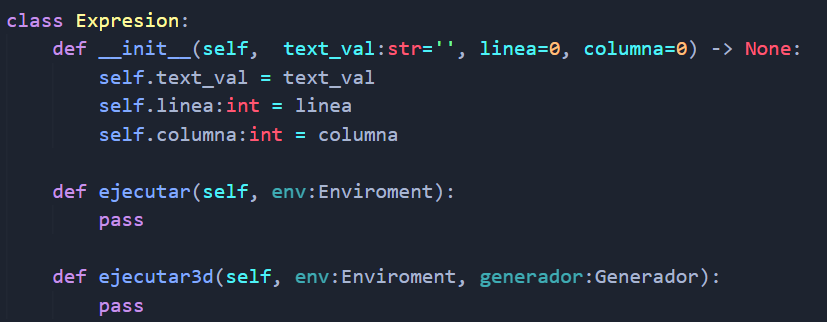


# **Interprete**

El intérprete es una representación en clases de cada una de las instrucciones y expresiones válidas en el lenguaje X-SQL, como lo son declaraciones y asignaciones de variables, operaciones aritméticas y lógicas, etc. Estas clases se dividen en dos tipos: expresiones e instrucciones.

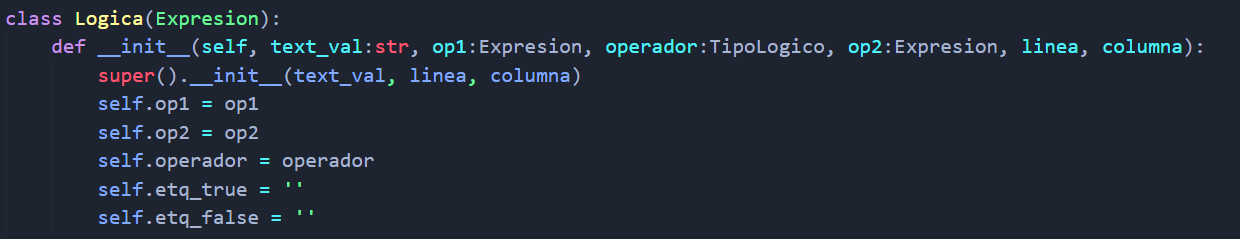
***Expresiones***

Una expresión será toda clase que al ‘ejecutarse’ retorna un valor. Todas estas expresiones van a heredar de otra clase llamada ‘Expresion’, por lo que, todas las expresiones que hereden de esta tendrán los siguiente atributos y métodos:

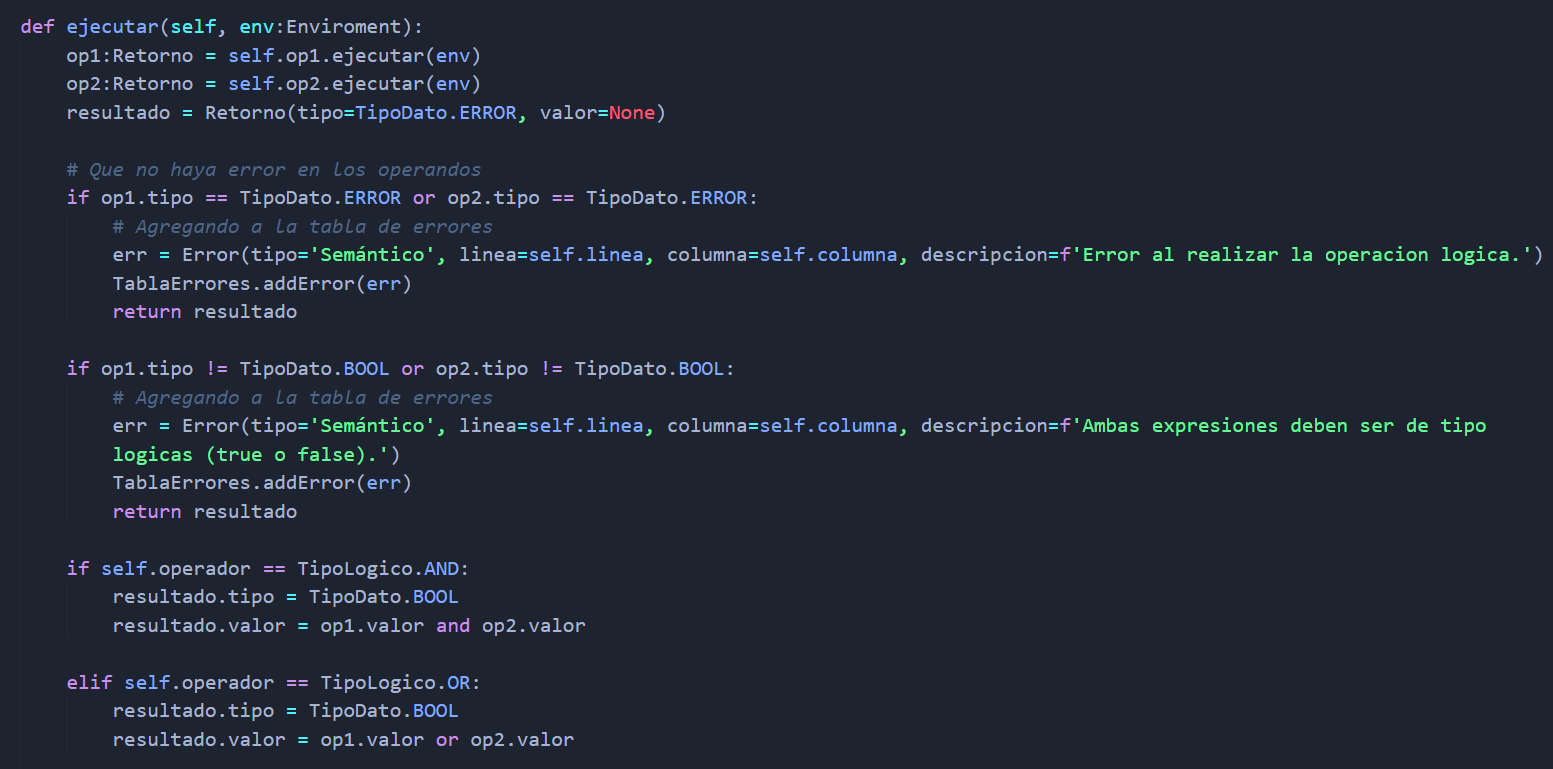


El método ‘ejecutar’, realiza la operación correspondiente a su expresión; y el método ‘ejecutar3d’, traduce la expresión a su correspondiente código de 3 direcciones (para cuando aplique).

Un ejemplo para esto, es la clase ‘Logica’, la cual retorna el resultado de una operación lógica. Esta, tal y como se mencionó, hereda de la clase Expresion y cuenta con dos operandos ‘op1’ y ‘op2’, así como el operador. Las otras dos etiquetas únicamente sirven para generar el codigo de 3 direcciones de la expresión.

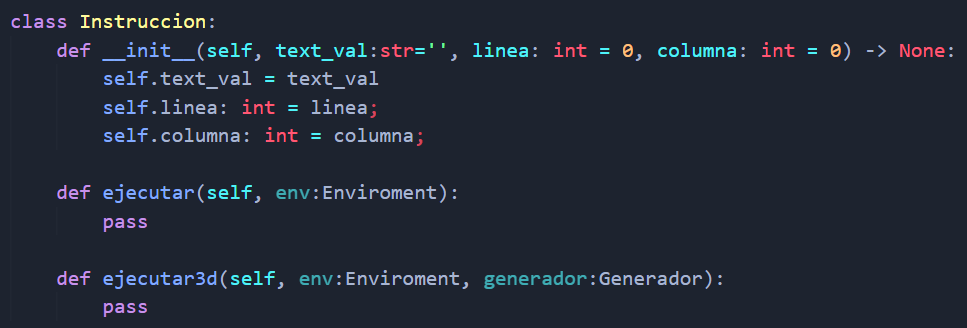


Asimismo, dentro del método ‘ejecutar’, se realizan las validaciones correspondientes, como validar que tanto el operando 1 y 2 sean de tipo booleano, así como el retorno de la expresión.



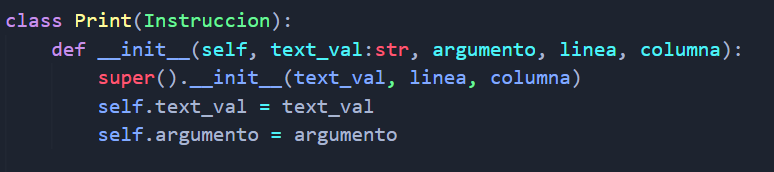
***Instrucciones***

Una instrucción será toda clase que al ‘ejecutarse’ NO retorna un valor. Todas estas instrucciones van a heredar de otra clase llamada ‘Instrucción’, por lo que, todas las instrucciones que hereden de esta tendrán los siguiente atributos y métodos:

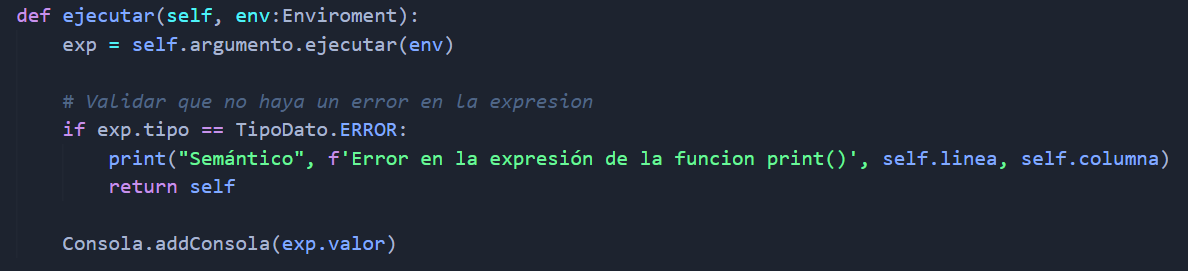


El método ‘ejecutar’, realiza la operación correspondiente a su instrucción; y el método ‘ejecutar3d’, traduce la instrucción a su correspondiente código de 3 direcciones (para cuando aplique).

Un ejemplo para esto, es la clase ‘Print’, la cual simplemente realiza una instrucción (determinada por el lenguaje). Esta, tal y como se mencionó, hereda de la clase Instrucción y únicamente cuenta con los atributos heredados. Las otras dos etiquetas únicamente sirven para generar el código de 3 direcciones de la expresión.

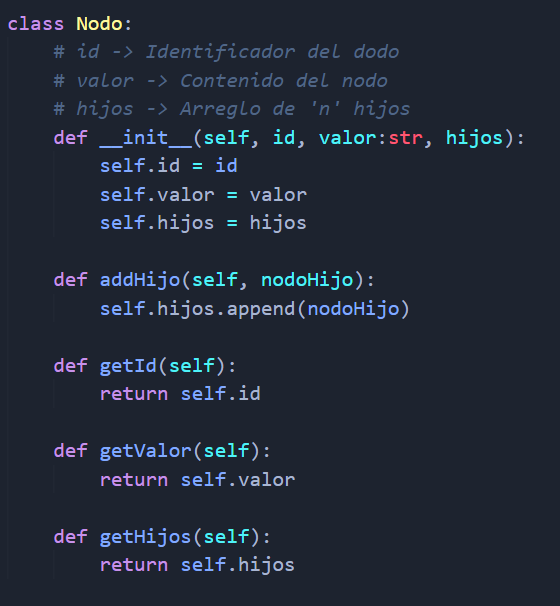


Asimismo, dentro del método ‘ejecutar’, se realizan las validaciones correspondientes, como validar que el argumento no sea de un tipo definido como ‘ERROR’ o agregar el valor de la expresión a una consola.

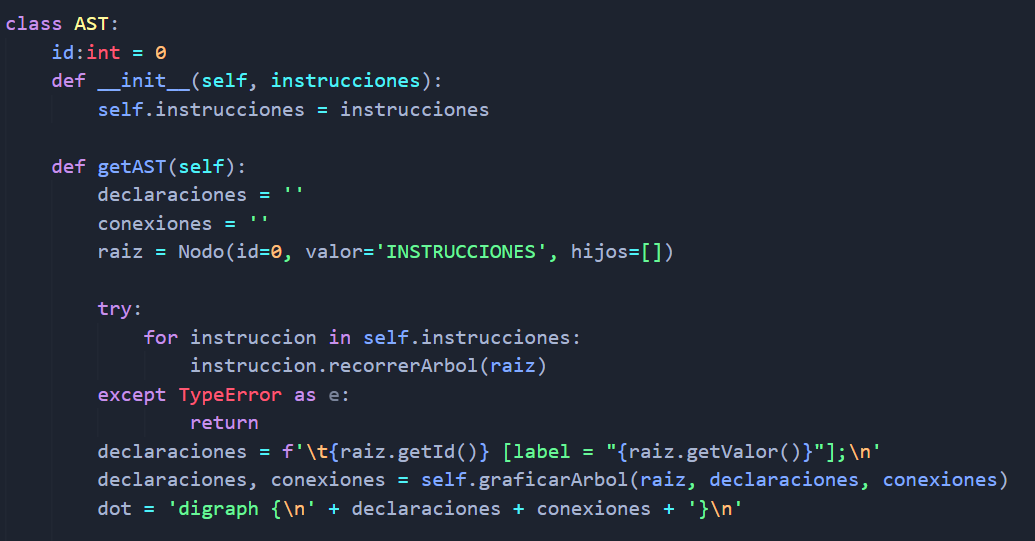


# **AST**

Para la generación del AST (Árbol Sintáctico Abstracto), se construyó un árbol n-ario, y es decir, cada nodo el árbol puede tener ‘n’ hijos, y cada nodo tiene como atributos un identificador (para las declaraciones y conexiones en graphviz), un valor (que es el texto como tal que irá dentro del nodo) y un arreglo (que será para los hijos de ese nodo); así como un conjunto de métodos asociados.

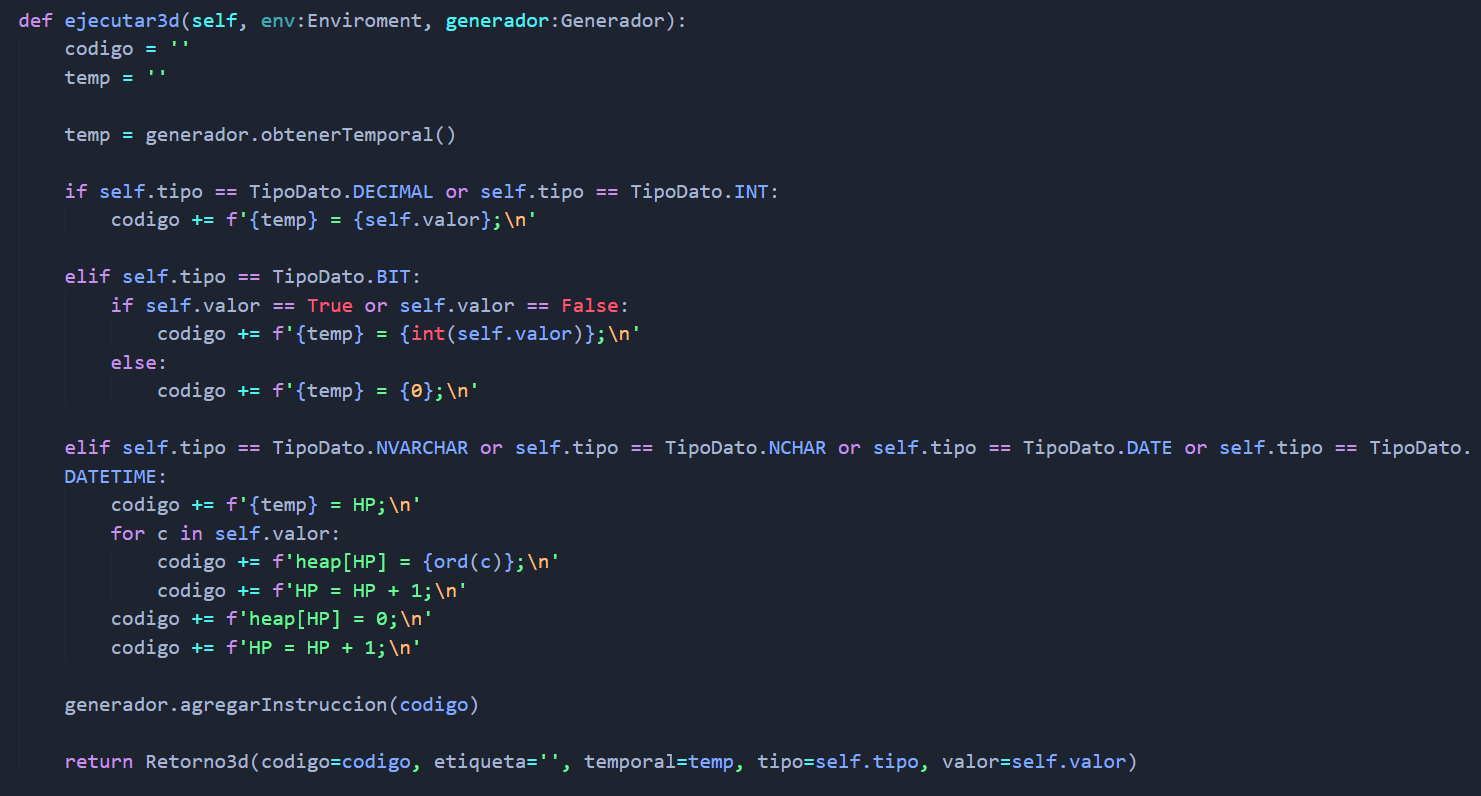


La clase ‘AST’ se encarga de construir la estructura del árbol, así recorrerlo para crear un archivo .dot son la estructura correcta:



# **Código de Tres Direcciones**

El código de 3 direcciones únicamente se realizó para las instrucciones de: Declaracion, Asignacion, If-else y Select (impresión). Cada una de estas clases cuenta con un método llamado “ejecutar3d”, el cual, utiliza terminales, etiquetas, la pila y heap, para generar un código equivalente al de la entrada.



### **Frontend**

(agregar estructura de carpetas y algunas explicaciones)