Universidad de San Carlos de Guatemala

Facultad de Ingeniería

Clase : Organización de Lenguajes y Compiladores 1

Sección N

Auxiliar: Jose Puac

Manual de Tecnico Proyecto JPR

Mynor Alison Isai Saban Che

201800516

DIvision del proyecto

La división del proyecto fue en cuatro partes la cual fueron las siguientes:

Expresiones:

aquí se tienen todos los tipos de datos, también las operaciones aritméticas, relacionales y lógicas.

Instrucciones:

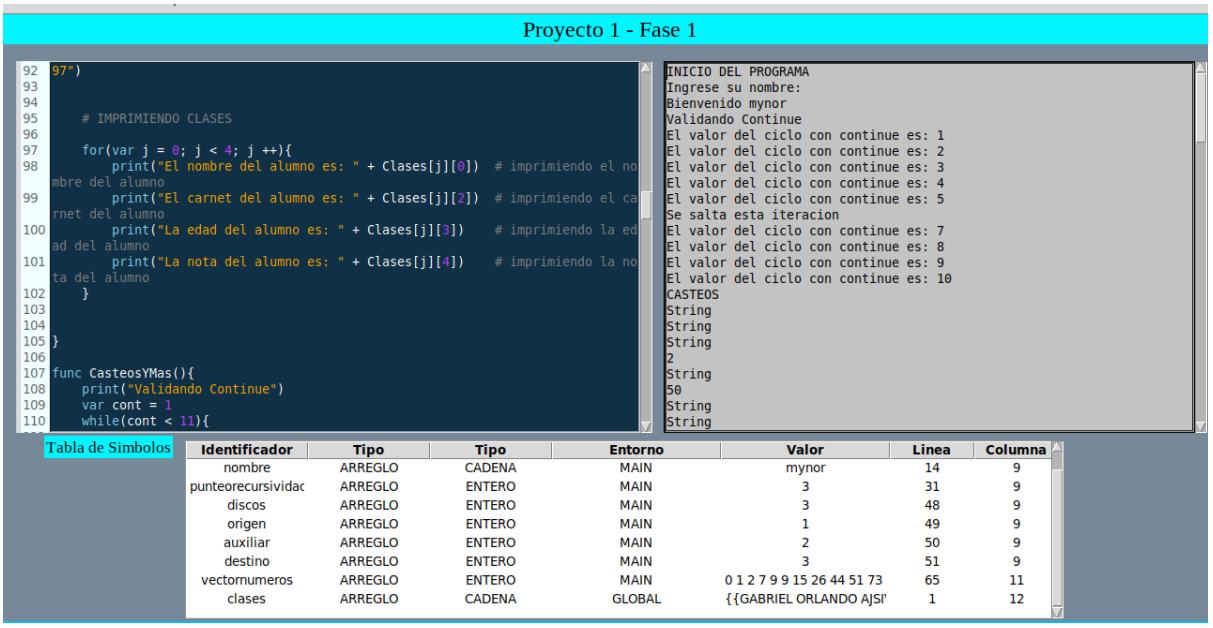
aquí se encuentra la magia del proyecto, ya que se utilizo el metodo interprete, también en esta carpeta se encuentra todas las clases que se uso para el analizador

Tabla Árbol:

aquí están todas las clases importantes para el analizador, los cuales son el árbol, tabla de símbolos, nodo para el ast .

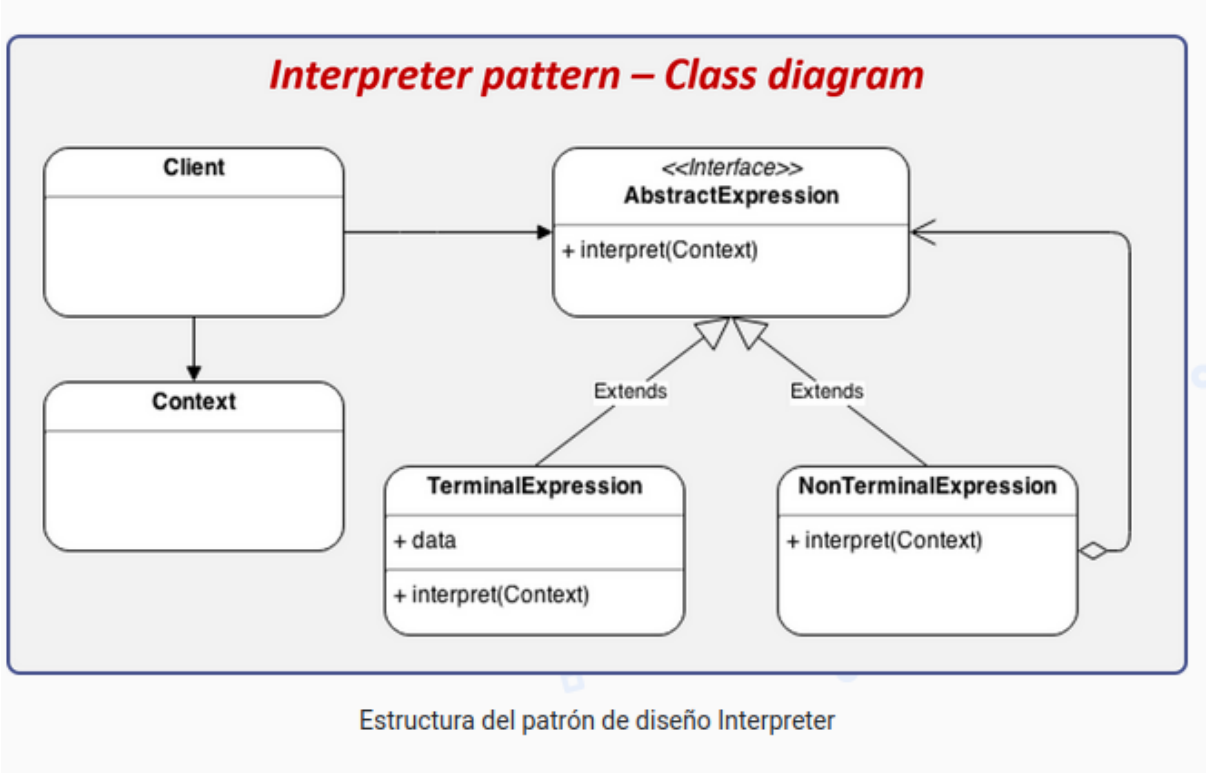
Nativas: aqui se encuentra todas las funciones nativas del lenguaje

Editor



Patron interprete

el patron interprete es el que hace la magia en este proyecto porque el es el que llama todas las funcione, pero como se usa este patrón, a continuación se muestra una imagen de cómo se estructura y una breve explicación

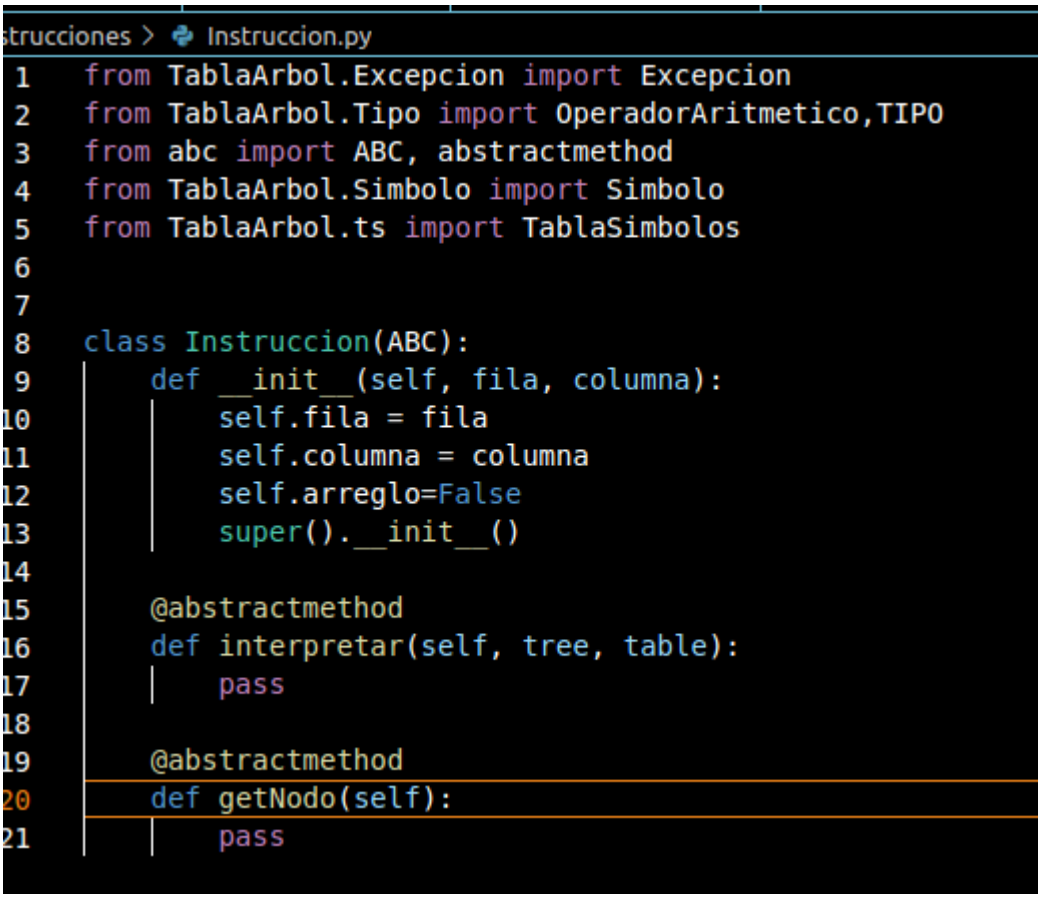


Estructura del patrón de diseño Interpreter

- Cliente:Actor que dispara la ejecución del intérprete.
- Context:Objeto con información global que será utilizada por el intérprete para leer y almacenar información global entre todas las clases que conforman el patrón, este es enviado al interpretar el cual lo replica por toda la estructura.
- Abstract Expression:Interface que define la estructura mínima de una expresión.
- Terminal Expresión:Se refiere a expresiones que no tienen más continuidad y al ser evaluadas o interpretadas terminan la ejecución de esa rama. Estas expresiones marcan el final de la ejecución de un sub-árbol de la expresión.
- NonTerminal Expression:Son expresiones compuestas y dentro de ellas existen más expresiones que deben ser evaluadas. Estas estructuras son interpretadas utilizando recursividad hasta llegar a una expresión Terminal.

- Fuente : https://reactiveprogramming.io/blog/es/patrones-de-diseno/interpreter

Para nuestro caso el patrón intérprete nos soluciono la vida. En la imagen siguiente se coloca la clase que sirvió como intérprete, la cual poseía los métodos abstractos necesarios para obtener dicho patron

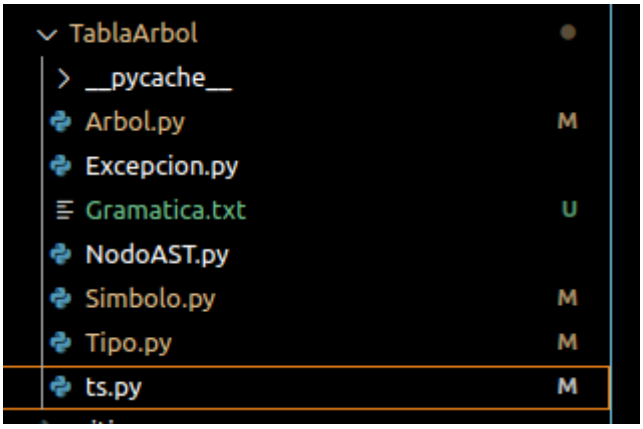


un ejemplo claro de como se utilizo el patron interprete se muestra en la siguiente trozo de codigo

```python
def interpretar(self, tree, table):

    value = self.expresion.interpretar(tree, table) # Valor a asignar a la variable
    if isinstance(value, Excepcion): return value

    simbolo = Simbolo(self.identificador.lower(), self.expresion.tipo,self.arreglo, sel

    result = table.actualizarTabla(simbolo)

    if isinstance(result, Excepcion): return result


    return None
```

aqui se aprecia todo lo que puede abarcar el metodo, como se puede apreciar este trozo de codigo depende completamente de instruccion (nuestro interprete “clase maestra”).

Tambien existen las siguientes clases para poder ejecutar el interprete.



- la clase arbol es algo ensencial ya que por medio de esta se guarda las intrucciones, exepciones que tiene el codigo, funciones, la tabla de simbolos “algo muy importante para nuestro analizador”.

- la clase excepcion lo unico que hace es un objeto de tipo exececpcion y poder guardarlo en el arbol, para cuando se termine la ejecucion de todo solo sellama para mostrar las exepciones que ocurrio dentro de todo el progrma

- la clase NodoAST, este su trabajo es ser nodo como su nombre lo indica, este grafica los nodos del arbol ast, este es objeto de tipo NODOAST, para cuando se grafique es mas flexible modelar el arbol.

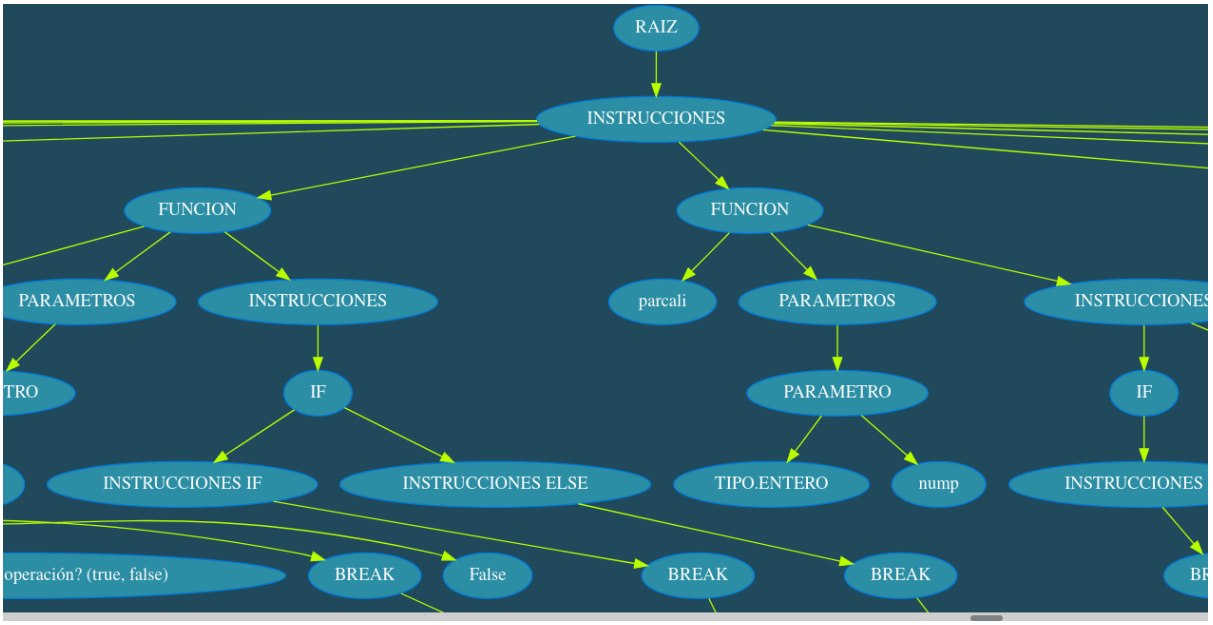
- la clase simbolo, su tabajo como su nombre lo indica es guardar todos los simbolos que tiene nuestro editor al momento de ser analizado, este es flexible, ya que acepta todo tipo de dato

- la clase tipo, solamente es una clase numerada con la capacidad de ordenar de manera facil y flexible los tipos de datos que maneja nuestro analizador.

Grafico de Arbol AST

```
def recorrerAST(self, idPadre, nodoPadre):
    for hijo in nodoPadre.getNodos_Hijos():
        nombreHijo = "n" + str(self.contador)

        print(hijo.getValor())
        try:
            self.dot += nombreHijo + "[label=\"" + hijo.valor.replace("\"", "\\\"") + "\"];\n"
        except:
            self.dot += nombreHijo + "[label=\"" + str(hijo.valor)+ "\"];\n"

        print(hijo.valor)
        self.dot += nombreHijo + "[label=\"" + hijo.getValor().replace("\"", "\\\"") + "\"];\n"

def recorrerAST(self, idPadre, nodoPadre):
    for hijo in nodoPadre.getNodos_Hijos():
        nombreHijo = "n" + str(self.contador)

        print(hijo.getValor())
        try:
            self.dot += nombreHijo + "[label=\"" + hijo.valor.replace("\"", "\\\"") + "\"];\n"
        except:
            self.dot += nombreHijo + "[label=\"" + str(hijo.valor)+ "\"];\n"

        print(hijo.valor)
        self.dot += nombreHijo + "[label=\"" + hijo.getValor().replace("\"", "\\\"") + "\"];\n"

        self.dot += idPadre + "->" + nombreHijo + ";\n"
        self.contador += 1
        self.recorrerAST(nombreHijo, hijo)
```

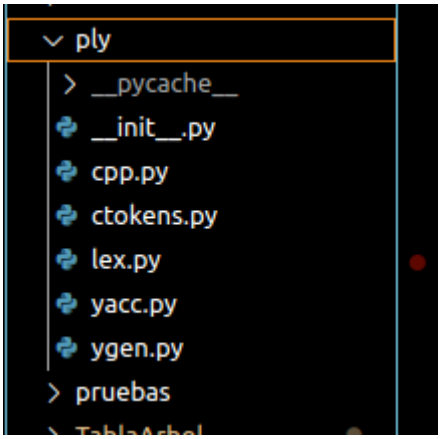
Se muestra el trozo de código que tiene la funcionalidad de poder graficar nuestro árbol, esto es posible gracias a la recursividad y la forma de trabajar con el patrón interprete, este ha sido muy útil para realizar el árbol.



esta es una pequeña representación del árbol generado por la recursividad y el patron interprete

Analizador PLY de python

para realizar este programa o editor de texto se utilizó el analizador ascendente PLY de python. Dentro de nuestra proyecto existe esta carpeta que comprueba con certeza que se utilizo este analizador



equipo donde se tabajo el proyecto