Samuel Isaí Muñoz Pereira
Carnet: 202100119
Curso: Lenguajes Formales y de Programación

Manual Técnico

Código Interfaz grafica

Para la parte grafica o visual se utilizó la librería Tkinter.

Tkinter es una biblioteca de Python que se utiliza para crear interfaces gráficas de usuario (GUI). Es decir, Tkinter proporciona herramientas y widgets para que los programadores puedan crear ventanas, botones, menús, etiquetas, campos de entrada, entre otros elementos que permiten que el usuario interactúe con el programa de una manera visual y más intuitiva.

Tkinter viene incluido con Python, por lo que no es necesario instalarlo por separado. Además, es una biblioteca multiplataforma, lo que significa que las aplicaciones creadas con Tkinter se pueden ejecutar en diferentes sistemas operativos como Windows, Linux o macOS.

Para la orientación de los botones se utilizo un software drag a drop, que permite ubicar de mejora manera cada uno de los elementos, añadiendo configuraciones decisionales como el tipo de fuente, color de fondo, etc. de una manera mucho más simple

Para realizar las acciones dependiendo de cada botón, fue necesario crear funciones que ejecutaran cada uno de las acciones, como la opción de limpieza o salida

```
62  ventana_principal = tk.Tk()
63  ventana_principal.geometry("942x552+401+217")
64  ventana_principal.minsize(120, 1)
65  ventana_principal.maxsize(3290, 1061)
66  ventana_principal.resizable(1,  1)
67  ventana_principal.title("Ventana Principal")
68  ventana_principal.configure(background="#FFB84C")
69  ventana_principal.configure(highlightbackground="#d9d9d9")
70  ventana_principal.configure(highlightcolor="black")
71
72
73
74  Label1 = tk.Label()
75  Label1.place(relx=0.0, rely=0.0, height=42, width=949)
76  Label1.configure(activebackground="#f9f9f9")
77  Label1.configure(background="#F5EAEA")
78  Label1.configure(compound='left')
79  Label1.configure(disabledforeground="#a3a3a3")
80  Label1.configure(font="-family {Arial} -size 14 -weight bold -slant italic")
81  Label1.configure(foreground="#000000")
82  Label1.configure(highlightbackground="#d9d9d9")
83  Label1.configure(highlightcolor="black")
84  Label1.configure(text='''Proyecto 2 Lenguajes Formales''')
```

```python
# Limpiar la caja de texto
def limpiar(cajaTexto):

    # Confirmacion en caso que se desee limpiar
    confirmacion = MessageBox.askyesno("Confirmar", "¿Desea Limpiar la Caja de Texto?")

    if confirmacion:
        cajaTexto.delete('1.0', tk.END)
        MessageBox.showinfo("Mensaje", "Limpieza realizada con Exito!")
```

Dependiendo de cada botón, las funciones son mas largas y complicadas, como la opción de guardar como o de abrir, que muestra una venta emergente en la que el usuario puede navegar por el sistema de ficheros de su computadora.

Existe también la opción de guardar, que en caso de que no se especifique una ruta con anterioridad, será necesario seleccionar una ruta como la opción de guardar como, esta opción replica un poco a la de los sistemas de guardado de Microsoft office.

Analizador Léxico

Un analizador léxico es un componente fundamental de un compilador o interpretador que se encarga de leer y analizar el código fuente de un programa para dividirlo en unidades más pequeñas y significativas llamadas "tokens".

Cada token representa un tipo de elemento del lenguaje de programación, como identificadores, números, símbolos y palabras clave, entre otros. El analizador léxico identifica cada token y lo clasifica en su respectivo tipo, y luego pasa esta información al siguiente componente del proceso de compilación o interpretación.

Para este proyecto únicamente se podía utilizar programación básica para el analizador léxico, por lo que en esta parte no se utiliza ninguna librería especifica. En este caso se ideo utilizar el sistema de gramática

El autómata determinista refleja cada uno de los pasos que el analizador léxico toma a la hora de comparar los tokens, en este caso empieza buscando la función a utilizar, pasa al próximo estado y busca la palabra “variable”, luego el signo igual “=” y así sucesivamente.

En este caso el analizador léxico va en búsqueda de cada uno de los estados y en caso de que coincida pasara al siguiente estado

Con este sistema se comprueba tanto la parte sintáctica como la parte léxica, ya que, al cambiar de estado, será necesario comprar un nuevo token, lo que significa que a la vez que verifica que el token a analizar es el correcto, verifica que ese es el orden en el que debe ir el token

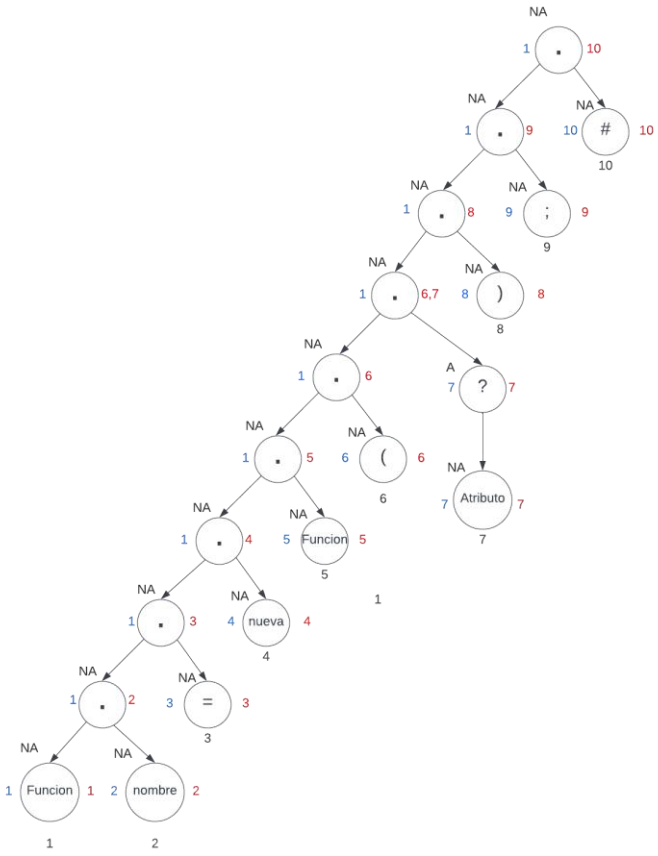
```
# S2 -> '=' S3
elif estado_actual == 'S2':
    # Busqueda del signo igual
    estado_actual = self._token('=', 'S2', 'S3')


# S3 -> nueva S4
elif estado_actual == 'S3':
    # Busqueda de la palabra nueva
    estado_actual = self._token('nueva', 'S3', 'S4')


# S4 -> Funcion S5
elif estado_actual == 'S4':
    # busqeuda de la palabra funcion
    estado_actual = self._token(funcionUsar, 'S4', 'S5')
```

Como se puede observar en esta porción de código, el estado S2 buscara al signo igual para posteriormente pasar al estado S3, el método _token, analiza el token que se desea comparar, y en caso de que la búsqueda coincida con el token, el método retornara el siguiente estado, por lo cual pasara al estado S3 en búsqueda de la palabra nueva para pasar al estado S4, y así sucesivamente

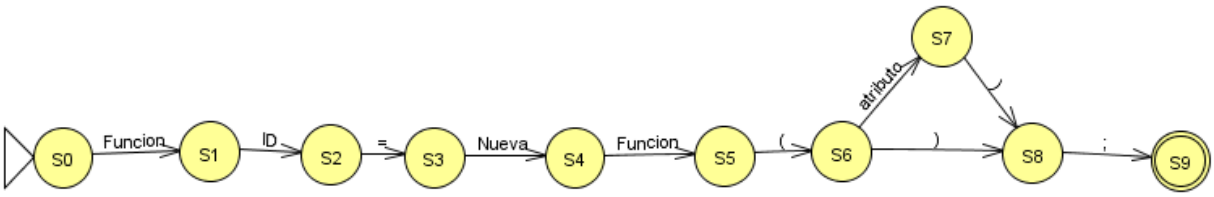
Existen varios métodos que son los encargados de que el método _token funcione de manera apropiada, como el método juntar, que junta todos los espacios igual al tamaño del token, por ejemplo, la palabra “nueva”, tiene 5 espacios, por lo tanto, el método juntar, junta los 5 espacios que encuentre, para esto se le va sumando 1 al índice de líneas que se tiene en el archivo.

También está el método analizar, que se encarga de verificar que lo retornado por la función juntar se lo mismo que el token, de este modo se compara si léxicamente es el mismo token o no, en caso de lo juntado coincida con el token se retorna un True lo que luego retornara en el estado siguiente, en caso de que no coincidan, retornara un false lo que a su vez retornara un error en la función principal.

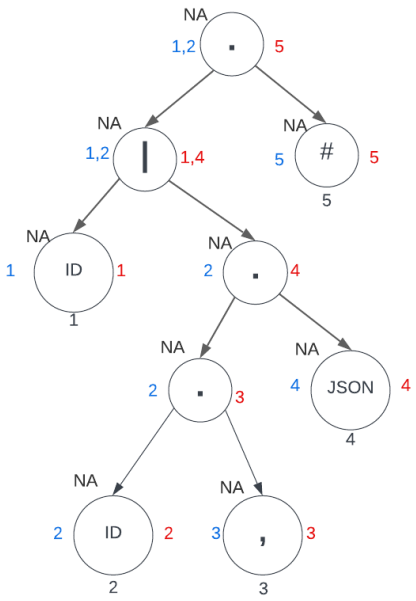
El método del árbol junto con los autómatas y gramática fue una parte esencial en el proyecto, ya que es la base de los estados, y del camino que el programa sigue para reconocer cada uno de los tokens:

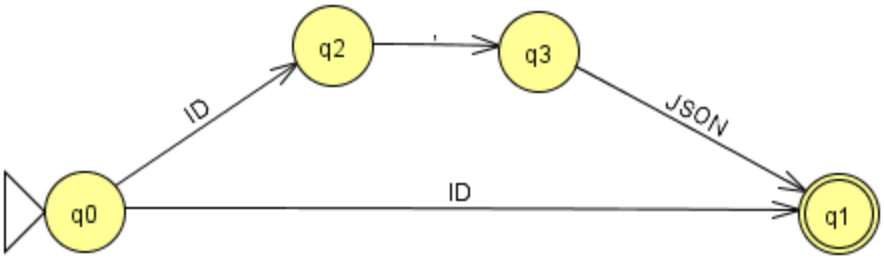


En este caso este es el árbol principal en el que se basa la estructura, donde se puede ver que de primero busca una función, variable, signo igual, nueva, función. Etc. se puede observar más detallado y entendible en el autómata siguiente



También se realizó un árbol especificadamente para la parte del atributo, ya que como se logra ver en el autómata principal, la parte del tributo puede no venir o si, en caso de que la opción de atributo si aparezca se tiene el siguiente árbol con su autómata





En caso de que venga la opción de atributo, puede venir únicamente un identificador, o un identificador con un archivo json de configuración. Por lo que existen 2 caminos

Para las opciones de identificador y de archivo json ya no se realizaron arboles debido a que son 3 posibles configuraciones y un autómata haría la gramática de una forma no tan legible y mas complicada de programar, por lo que se optó por únicamente generar las gramáticas para las diferentes opciones de archivo json

```
{."."texto."."."texto."."."texto."."."texto."."}
{."."texto."."."texto."."}.,.{."$set".".{."."texto."."."texto."."}."}
{."."texto."."."texto."."}
```

En este caso existen 3 posibles vías de archivo json, una donde se encuentran 2 atributos con sus respectivos datos, otra donde se encuentra un atributo y la opción $set, y un ultimo con un solo atributo y su dato. Como se menciono anteriormente se decidió realizar los estados siguientes esta gramática de modo que fuera de manera lineal el renacimiento de tokens. Esto significa que existen 3 diferentes grupos de estados para el reconocimiento del sistema json

```
289     # Retorna el JSON, o error al metodo atributo
290 >   def nombre(self):···
380
381     # Retorna el JSON, o error al metodo atributo
382 >   def nombre_set(self):···
543
544     # Retorna el JSON, o error al metodo atributo
545 >   def nombre_autor(self):···
684
```

Algunas variables de la clase Analizador

```
class Analizador:
    def __init__(self, entrada:str):
        self.lineas = entrada # Texto analizar
        self.index = 0 # Poscicion dentro de todo el texto
        self.fila = 1 # Fila Actual
        self.columna = 0 # Columna Actual
        self.ListaErrores = [] # Lista de errores

        self.tokenAutilizar = '' # Token utilizado en el momentos
        self.json = '' # sintaxis json para el comando mongoDb
        self.identificador = ''  # indentificador para el comando
        self.contadorToken = 0  # Contador de tokens encontrados
        self.listaTokens  = []  # Lista con todos los tokens
        self.listaComandos = [] # Lista con todos os comandos generados

        self.textoError = ''    # Texto que da error con el token analizado
        self.tipoDeError = ''  # Guarda el tipo de error, lexico o sintactico
        self.contadorErrores = 0
        # Una lista con todos los posibles tokens
        self.listaTodosTokes = ['CrearBD','EliminarBD','CrearColeccion', 'EliminarColeccion', 'InsertarUnico',
                                'BuscarUnico', 'nueva', '"', '=', '(', ')', ':', ';', ',', '$set', '{', '}', '/
```

En este caso se utilizan varias variables y listas para ir guardan la información como los tokens encontrados, errores encontrados y comandos generados.

Generación Reporte de Errores

Para la generación de la tabla de errores de utiliza Graphviz con un sistema de tablas tipo HTML, para esto se la pasa la lista de errores, directamente de la clase Analizador, y la función se encarga de recorrer la lista con diccionarios

```
def generacionTokens(lista):
    #lista = get_listaTokens()

    with open("Reports/Tokens.dot", "w", encoding="utf-8") as grafo_dot:
        grafo_dot.write('digraph { \n')
        grafo_dot.write(f'graph [label="Tabla de Tokens", labelloc=top]\n')
        grafo_dot.write('rankdir = LR \n' )
        grafo_dot.write('ranksep=1.5 \n' )
        grafo_dot.write(f'node[shape=none, style="filled" fontname="Arial", fontsize=12] \n\n')

        # creara el encabezado en graphviz
        grafo_dot.write(f'''

    n{1} [ label = <
        <table>
            <tr><td bgcolor="#e74c3c"> Correlativo </td><td bgcolor="#f39c12 "> Lexema </td><td bgcolor="#f1c40
                ''')

        # Recorre la lista con diccionarios, donde se ecuentra cada uno de los tokens encontrados
        for diccionario in lista:
            grafo_dot.write(f'''
            <tr><td bgcolor="#f1948a "> {diccionario["contador"]} </td><td bgcolor="#f8c471 "> {diccionario["le
                    ''')
```

Generación Reporte de Tokens

Para esta tabla de igual manera se utiliza el mismo sistema del reporte anterior, en el que va generando celdas con los datos de la lista de diccionarios

```
def generacionErrores(lista):

    with open("Reports/Errores.dot", "w", encoding="utf-8") as grafo_dot:
        grafo_dot.write('digraph { \n')
        grafo_dot.write(f'graph [label="Tabla de Errores", labelloc=top]\n')
        grafo_dot.write('rankdir = LR \n' )
        grafo_dot.write('ranksep=1.5 \n' )
        grafo_dot.write(f'node[shape=none, style="filled" fontname="Arial", fontsize=12] \n\n')

        # creara el encabezado en graphviz
        grafo_dot.write(f'''

    n{1} [ label = <
        <table>
            <tr><td bgcolor="#2c3e50"> Correlativo </td><td bgcolor="#34495e"> Token Error </td><td bgcolor="#7
                ''')

        # Recorre la lista con diccionarios, donde se ecuentra cada uno de los tokens encontrados
        for diccionario in lista:
            grafo_dot.write(f'''
            <tr><td bgcolor="#808b96"> {diccionario["contador"]} </td><td bgcolor="#85929e"> {diccionario["toke
                    ''')


        # Cierra la tabla
        grafo_dot.write(f'''
        </table>
```