Samuel Isaí Muñoz Pereira
Carnet: 202100119
Curso: Lenguajes Formales y de Programación
Proyecto 1

Manual Técnico

Código Interfaz grafica

```
# Botones
# Abrir
boton_Abrir = tk.Button(ventanaPrincipal, text="Abrir", fon =10, padx=56, bg="#BFDB38", command=editorTexto)
boton_Abrir.place(x=600, y=120)

# Guardar
boton_Guardar = tk.Button(ventanaPrincipal, text="Guardar", font=10, padx=45, bg="#BFDB38", command=guardarTexto)
boton_Guardar.place(x=600, y=170)

# Guardar Como
boton_GuardarComo = tk.Button(ventanaPrincipal, text="Guardar Como", font=10, padx=21, bg="#BFDB38", command=guardarComoTexto)
boton_GuardarComo.place(x=600, y=220)

# Analizar
boton_Analizar = tk.Button(ventanaPrincipal, text="Analizar", font=10, padx=45, bg="#BFDB38", command=analizar)
boton_Analizar.place(x=600, y=270)

# Limpiar
boton_Limpiar = tk.Button(ventanaPrincipal, text="Limpiar", font=10, padx=47, bg="#BFDB38", command=limpiarTexto)
boton_Limpiar.place(x=600, y=320)
```

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.

Cada uno de los botones fue programado y orientado de manera manual, para este apartado no se utilizo clases, algo que hubiera sido de gran ayuda pero al ser varios botones iba a resultar complicado la programación de cada uno

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

```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!")
```

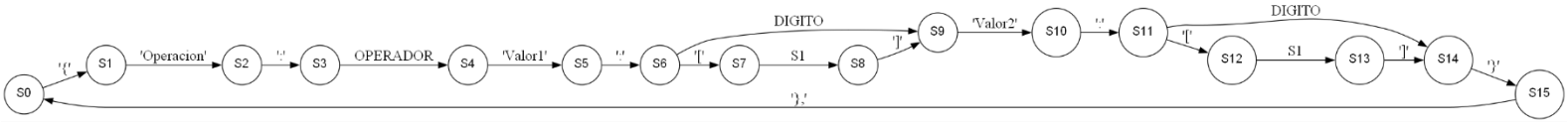
Dependiendo de cada botón, las funciones son mas largas y complicadas, como la opción de guardar como o de abrir, que muestra una venta emergente en la que el usuario puede navegar por el sistema de ficheros de su computadora

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 llave “{”, pasa al próximo estado y busca la palabra “Operación”, luego los dos puntos “:” 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

```
# ************************
#         ESTADOS
# ************************

# S1 -> "Operacion" S2
elif estado_actual == 'S1':
    estado_actual = self._token('"Operacion"', 'S1', 'S2')

# S2 -> : S3
elif estado_actual == 'S2':
    estado_actual = self._token(':', 'S2', 'S3')

# S3 -> OPERADOR S4
elif estado_actual == 'S3':
    operadores = ['"Suma"','"Resta"', '"Division"', '"Multiplicacion"', '"Potencia"',
                  '"Raiz"', '"Mod"', '"Inverso"', '"Seno"', '"Coseno"', '"Tangente"']
    for i in operadores:
        estado_actual = self._token(i, 'S3', 'S4')
        if estado_actual != 'ERROR':
            operador = i
            break

# S4 -> "Valor1" S5
elif estado_actual == 'S4':
    estado_actual = self._token('"Valor1"', 'S4', 'S5')
```

En este caso los estados van uno a uno buscando cada uno de los estados.

```
class Analizador:
    def __init__(self, entrada):
        self.texto = entrada
        self.index = 0
        self.fila = 1
        self.columna = 1
        self.contadorHijo = 0
        self.contadorGeneral = 0

        self.listaArbol = []
        self.listaConfiguracionDot = []
        self.listaErrores = []
```

Para el código se utiliza la clase Analizador, en la cual se guardarán varias variables y listas, que se estarán utilizando en todo el analizador para guardar los estados, datos y errores que vayan apareciendo en la ejecución del programa

Por ejemplo, en la lista árbol, se guardan todas las operaciones y resultados de la ejecución, para posteriormente utilizar en la generación del árbol en graphviz.

La lista de errores guardara cada uno de los errores para posteriormente utilizarlo para la generación del reporte de errores en formato .json

Guardado y Abrir Archivos

```python
# Funcion para elegir el nombre y la ruta para guardar el archivo
def guardarComo(cajaTexto):

    global rutaGuardado

    rutaGuardado = filedialog.asksaveasfilename(
        filetypes={
            ("Todos los archivos", "*.*")
        }
    )

    if rutaGuardado:

        with open(rutaGuardado, 'w') as lineas:
            contenido = cajaTexto.get("1.0", tk.END)
            lineas.write(contenido)
            MessageBox.showinfo("Mensaje", "Se guardo correctamente los datos en la ruta: " + str(rutaGuardado))

    else:
        MessageBox.showwarning("Alerta", "No se completo el guardado")
```

para las opciones de guardar y abrir se utilizada la función filedialog para poder generar un explorador de archivos donde el usuario podrá navegar dentro del sistema. En este cado se utiliza una variable “rutaGuardado” donde se ira guardado la ruta del archivo para cada vez que sea necesario guardar

Generación Reporte de Errores

LFP_Proyecto1 > errores.py > ...

```python
import json


def generadorJson(lista):
    listaNueva = []
    contador = 0

    if len(lista) != 0:
        for x in lista:
            contador += 1

            datos = {
                "No.":contador,
                "Descripcion-Token": {
                    "Lexema":x["token"],
                    "Tipo":"Error",
                    "Columna":x["columna"],
                    "Fila":x["fila"]
                }
            }
            listaNueva.append(datos)


        # Escribir la lista de diccionarios en un archivo JSON
        with open('ERRORES_202100119.json', 'w') as lineas:
            json.dump(listaNueva, lineas, indent=3)

    else:
        with open('ERRORES_202100119.json', 'w') as lineas:
            lineas.write("[\n]")
```

Para la generación del archivo .json se utilizó la librería Json, la cual permite añadir texto e indentarlo, en este caso existe un diccionario datos, la cual contiene la estructura de la información que se ira agregando en el archivo .json, por ultimo la opción json.dump cargara la lista “listaNueva” con todos los datos al nuevo archivo. En caso de que no exista ningún error el archivo únicamente escribirá “[ ]”

Creación del archivo dot

```
# Para Datos con Operaciones con 2 Valores
if len(x) == 6:
    for i in range(contador2,  maximiliano):

        #print(i)
        # Hijo Izquierdo
        if x[4] == lista[i][3] and x[0] == general:
            if valorIzquierdo == False:
                grafo_dot.write(f"N{x[0]}{x[1]} -> N{lista[i][0]}{lista[i][1]} \n")
                valorIzquierdo = True
                break


    for j in range(contador2, maximiliano):
        # Hijo Derecha
        if x[5] == lista[j][3] and x[0] == general:
            if valorDerecho == False:
                grafo_dot.write(f"N{x[0]}{x[1]} -> N{lista[j][0]}{lista[j][1]} \n")
                valorDerecho = True
                break

    # Cambia el Label para el nodo en caso de que sea un resultado
    grafo_dot.write(f'N{x[0]}{x[1]}[label="{textoLimpio}:\n{x[3]}"]\n')


    # Agrega las hojas der arbol, o los ultimos datos de las ramas
    # Hijo Izquierdo
    if  valorIzquierdo == False:
        grafo_dot.write(f"N{x[0]}{x[1]} -> V{x[0]}{x[1]}1 \n")
        grafo_dot.write(f'V{x[0]}{x[1]}1[label="{x[4]}"]\n')
    else:
        valorIzquierdo = False

    # Hijo Derecho
    if valorDerecho == False:
        grafo_dot.write(f"N{x[0]}{x[1]} -> V{x[0]}{x[1]}2 \n")
        grafo_dot.write(f'V{x[0]}{x[1]}2[label="{x[5]}"]\n')
    else:
        valorDerecho = False
```

Para este caso se utiliza una lógica y un algoritmo 100% creado por mí, el cual empieza creando el nodo por la raíz, y se va distribuyendo por sus ramas para terminar en las hojas.

En caso de que el resultado de la raíz coincida con alguna de las ramas, se creara una raíz entre el resultado y la rama, creando hijos sucesivamente. En caso de que el algoritmo encuentre una hoja, este ya no creara ramas si no únicamente un nodo el cual le asignara su respectivo label.