Manual técnico

27 MARZOYECTO 1

Lenguajes formales de programación

Creado por: Juan Pablo Samayoa Ruiz

Principal

Ventana main: Ventana principal en la que se encuentra toda la interfaz grafica y cambios de ventana

-Imports:

```
import tkinter as tk
from tkinter import filedialog, Tk
from analizador import *
import os
import subprocess
```

- **tkinter**: es el módulo que proporciona las herramientas para construir la interfaz gráfica de usuario (GUI) en Python. Es una biblioteca estándar de Python que viene preinstalada en la mayoría de los sistemas operativos.
- **filedialog**: es un submódulo de tkinter que proporciona una interfaz para abrir y guardar archivos utilizando una ventana de diálogo.
- **Tk**: es una clase de tkinter que se utiliza para crear la ventana principal de la aplicación.
- **analizador**: es un módulo personalizado que contiene el código para analizar el texto y realizar operaciones matemáticas.
- **os**: es un módulo que proporciona una interfaz para interactuar con el sistema operativo subyacente. En este código, se puede usar para obtener información sobre los archivos del sistema.
- **subprocess**: es un módulo que permite crear nuevos procesos, conectar con sus tuberías de entrada/salida/error, y obtener sus códigos de salida. En este código, se puede usar para ejecutar otros programas o scripts desde Python.

-Clase TextEditor**:** Parte grafica en la que se encuentran algunas funciones del programa como lo es abrir un archivo .txt y mostrarlo, guardado, guardar como archivo nuevo y la interfaz grafica del editor de texto.

```python
class TextEditor:
    def __init__(self, master):
        self.master = master
        self.master.title("Editor de Texto")

        self.menu_principal = tk.Menu(self.master)

        self.menu_archivo = tk.Menu(self.menu_principal, tearoff=0)
        self.menu_archivo.add_command(label="Abrir", command=self.abrir_archivo)
        self.menu_archivo.add_command(label="Guardar", command=self.guardar_archivo)
        self.menu_archivo.add_command(label="Guardar como", command=self.guardar_archivo_como)
        self.menu_archivo.add_command(label="Analizar", command=self.analizar_texto)
        self.menu_archivo.add_command(label="Errores", command=self.mostrar_errores)
        self.menu_archivo.add_command(label="Salir", command=self.salir)
        self.menu_principal.add_cascade(label="Archivo", menu=self.menu_archivo)

        self.master.config(menu=self.menu_principal)
```

```python
    def abrir_archivo(self):
        self.ruta = filedialog.askopenfilename(defaultextension=".txt", filetypes=[("Archivos de texto", "*.txt")])
        if self.ruta is not None:
            self.archivo = open(self.ruta,'r')
            contenido = self.archivo.read()
            self.texto.delete(1.0, tk.END)
            self.texto.insert(tk.END, contenido)


    def guardar_archivo(self):
        if self.ruta is not None:
            with open(self.ruta, "w") as archivo:
                contenido = self.texto.get(1.0, tk.END)
                archivo.write(contenido)

    def guardar_archivo_como(self):
        archivo = filedialog.asksaveasfile(defaultextension=".txt", filetypes=[("Archivos de texto", "*.txt")])
        if archivo is not None:
            contenido = self.texto.get(1.0, tk.END)
            archivo.write(contenido)
            archivo.close()
```

```python
    def guardar_archivo_como(self):
        archivo = filedialog.asksaveasfile(defaultextension=".txt", filetypes=[("Archivos de texto", "*.txt")])
        if archivo is not None:
            contenido = self.texto.get(1.0, tk.END)
            archivo.write(contenido)
            archivo.close()

    def analizar_texto(self):
        archivo = open(self.ruta, "r")
        contenido = archivo.read()
        instruccion(contenido)
        respuestas = calculadora_()
        for respuesta in respuestas:
            print(respuesta.calculadora(None))
```

```
def mostrar_errores(self):
    lista_errores = getErrores()
    contador = 1
    with open('ERRORES_202109705.txt', 'w') as outfile:
        outfile.write('{\n')
        while lista_errores:
            error = lista_errores.pop(0)
            outfile.write(str(error.calculadora(contador)) + ',\n')
            contador +=1
        outfile.write('}')

def salir(self):
    self.master.destroy()

def start(self):
    self.texto = tk.Text(self.master)
    self.texto.pack()
```

El método "abrir_archivo" abre un archivo de texto seleccionado por el usuario y muestra su contenido en el editor de texto. El método "guardar_archivo" guarda el contenido del editor de texto en el archivo abierto previamente. El método "guardar_archivo_como" guarda el contenido del editor de texto en un nuevo archivo de texto seleccionado por el usuario.

El método "analizar_texto" abre el archivo de texto seleccionado previamente y lo envía a una función llamada "instruccion", que analiza el texto y devuelve una lista de objetos "calculadora_". Luego, el método "analizar_texto" recorre la lista de objetos "calculadora_" y llama a su método "calculadora" para mostrar los resultados en la consola.

El método "mostrar_errores" llama a una función llamada "getErrores" que devuelve una lista de objetos "Error". Luego, el método "mostrar_errores" crea un archivo de texto llamado "ERRORES_202109705.txt" y escribe los objetos "Error" en él.

Finalmente, el método "salir" destruye la ventana principal y cierra la aplicación. El método "start" inicia la aplicación creando un cuadro de texto en la ventana principal.

Ventana de ayuda: muestra una ventana en la que se encuentran los datos del estudiante que realizó el programa, además de esto se encuentran las funciones en las que se lee y muestra el manual.

```python
def ayuda():
    ventanaAyuda = tk.Toplevel(ventana)
    ventanaAyuda.geometry("400x180")

    etiqueta1 = tk.Label(ventanaAyuda, text="Juan Pablo Samayoa Ruiz", font=("Arial", 16), fg="blue")
    etiqueta1.pack(padx=10, pady=10)

    etiqueta2 = tk.Label(ventanaAyuda, text="202109705", font=("Arial", 16), fg="blue")
    etiqueta2.pack(padx=10, pady=10)

    etiqueta2 = tk.Label(ventanaAyuda, text="Laboratorio LFP B-", font=("Arial", 16), fg="blue")
    etiqueta2.pack(padx=10, pady=10)

def Monstrar_MU():
    ruta_pdf = "C:/Users/jpsam/OneDrive/Escritorio/Python/LFP P1_202109705/[LFP]202109705_Manual de usuario.pdf"
    if os.name == 'nt':
        os.startfile(ruta_pdf)
    else:
        subprocess.call(["xdg-open", ruta_pdf])

def Monstrar_MT():
    ruta_pdf = "C:/Users/jpsam/OneDrive/Escritorio/Python/LFP P1_202109705/[LFP]202109705_Manual de usuario.pdf"
    if os.name == 'nt':
        os.startfile(ruta_pdf)
    else:
        subprocess.call(["xdg-open", ruta_pdf])
```

Def ayuda: La primera etiqueta muestra el nombre completo de una persona, "Juan Pablo Samayoa Ruiz", con un tamaño de fuente de 16 y en color azul.

La segunda etiqueta muestra un número, "202109705", también con un tamaño de fuente de 16 y en color azul.

La tercera etiqueta muestra el nombre de un laboratorio, "Laboratorio LFP B-", con un tamaño de fuente de 16 y en color azul.

Cada etiqueta se empaqueta en la ventana emergente con un relleno horizontal y vertical de 10 píxeles. La ventana emergente en sí tiene una geometría de 400x180 píxeles.

Def Mostrar_MU Y Def Mostrar_MTPrimero, la función define una variable llamada "ruta_pdf" que contiene la ubicación del archivo del manual de usuario en el sistema de archivos del usuario. En este caso, la ruta es "C:/Users/jpsam/OneDrive/Escritorio/Python/LFP P1_202109705/[LFP]202109705_Manual de usuario.pdf".

Luego, la función verifica si el sistema operativo en uso es Windows (nt). Si es así, utiliza la función "os.startfile" para abrir el archivo PDF en el lector de PDF predeterminado del usuario. Si no, la función utiliza la función "subprocess.call" y el comando "xdg-open" para abrir el archivo PDF en el lector de PDF predeterminado del usuario en otros sistemas operativos, como Linux.

Ventana principal: Parte grafica principal en la que se muestra la ventana inicial junto con sus botones

```
ventana = tk.Tk()
ventana.geometry("350x250")
ventana.title("Inicio")

marco_principal = tk.Frame(ventana, width=400, height=300)
marco_principal.grid(row=0, column=0, padx=20, pady=20)

menu_archivo = tk.Label(marco_principal, text="Archivo")
menu_archivo.grid(row=0, column=0, padx=10, pady=10)

def abrir_editor():
    editor = TextEditor(tk.Toplevel(ventana))
    editor.start()

boton_abrir = tk.Button(marco_principal, text="Abrir editor de texto", command=abrir_editor)
boton_abrir.grid(row=1, column=0, padx=10, pady=10)

menu_ayuda = tk.Label(marco_principal, text="Ayuda")
menu_ayuda.grid(row=0, column=1, padx=10, pady=10)

boton_manual_usuario = tk.Button(marco_principal, text="Manual usuario", command=Monstrar_MU)
boton_manual_usuario.grid(row=1, column=1, padx=10, pady=10)

boton_manual_ayuda = tk.Button(marco_principal, text="Manual Tecnico", command=Monstrar_MT)
boton_manual_ayuda.grid(row=2, column=1, padx=10, pady=10)

boton_temas = tk.Button(marco_principal, text="Temas de ayuda", command=ayuda)
boton_temas.grid(row=3, column=1, padx=10, pady=10)

ventana.mainloop()
```

Primero, se crea la ventana principal utilizando "tk.Tk()" y se establecen sus propiedades, como la geometría y el título.

A continuación, se define un marco principal utilizando "tk.Frame()" y se agrega a la ventana principal mediante "grid()". Este marco se utiliza para colocar los diferentes elementos de la interfaz gráfica.

Se agregan dos etiquetas de texto para los menús de "Archivo" y "Ayuda", cada una de ellas en una celda diferente del marco principal utilizando "grid()".

Luego, se agregan tres botones en celdas diferentes del marco principal. El primer botón es "Abrir editor de texto" y tiene asociada una función llamada "abrir_editor" que crea una nueva ventana emergente con el editor de texto.

El segundo y tercer botón son "Manual de usuario" y "Manual técnico", respectivamente, y tienen asociada una función llamada "Monstrar_MU" y "Monstrar_MT" que abren los archivos de los manuales de usuario y técnico en formato PDF.

Finalmente, se agrega un cuarto botón llamado "Temas de ayuda", que llama a la función "ayuda" que crea una ventana emergente con información adicional de ayuda.

Se llama a "mainloop()" para iniciar la ejecución de la aplicación y permitir la interacción del usuario con la ventana y sus elementos.

Analizador:

```
import os
from Instrucciones.artimetica import *
from Instrucciones.trigonometricas import *
from Abstract.Lex import *
from Abstract.getNum import *
from Abstract.errores import error

palabras_reservadas = {
    'Reser_OPERACION':      'Operacion',
    'Reser_Valor1':         'Valor1',
    'Reser_Valor2':         'Valor2'   ,
    'Reser_Suma':           'Suma',
    'Reser_Resta':          'Resta',
    'Reser_Multiplicacion': 'Multiplicacion',
    'Reser_Division':       'Division',
    'Reser_Potencia':       'Potencia',
    'Reser_Raiz':           'Raiz',
    'Reser_Inverso':        'Inverso',
    'Reser_Seno':           'Seno',
    'Reser_Coseno':         'Coseno',
    'Reser_Tangente':       'Tangente',
    'Reser_Modulo':         'Mod',
    'Reser_Texto':          'Texto',
    'Reser_ColFondoNodo':   'Color-Fondo-Nodo',
    'Reser_ColFuenteNodo':  'Color-Fuente-Nodo',
    'Reser_NodeShape':      'Forma-Nodo',
    'Coma':                 ',',
    'Punto':                '.',
    'DosPuntos':            ':',
    'CorcheteIzquierdo':    '[',
    'CorcheteDerecho':      ']',
    'LlaveIzquierda':       '{',
    'LlaveDerecha':         '}',
}

lexemas = list(palabras_reservadas.values())
```

```
global  n_linea
global  n_columna
global  instrucciones
global  lista_lexemas
global lista_errores
global contx

contx = 0
n_linea = 1
n_columna = 1
lista_lexemas = []
instrucciones = []
lista_errores = []
```

```
def instruccion(cadena):
    global  n_linea
    global  n_columna
    global  lista_lexemas

    lexema = ''
    pointer = 0

    while cadena:
        char = cadena[pointer]
        pointer += 1

        if char == '\"':
            lexema, cadena = armar_lexema(cadena[pointer:])
            if lexema and cadena:
                n_columna +=1
                l = Lex(lexema, n_linea, n_columna)
                lista_lexemas.append(l)
                n_columna += len(lexema)+1
                pointer = 0

        elif char == '0' or char == '1' or char == '2' or char == '3' or char =
            token, cadena = armar_numero(cadena)
            if token and cadena:
                n_columna +=1
                n = Numeros(token, n_linea, n_columna)
                lista_lexemas.append(n)
                n_columna += len(str(token)) +1
                pointer = 0
```

```
    elif char == '[' or char == ']':
        c = Lex(char, n_linea, n_columna)
        lista_lexemas.append(c)
        cadena = cadena[1:]
        pointer = 0
        n_columna +=1

    elif char == '\t':
        n_columna +=4
        cadena = cadena[1:]
        pointer = 0

    elif char == '\n':
        cadena = cadena[1:]
        pointer = 0
        n_linea += 1
        n_columna = 1

    elif char == ' ' or char == '\r' or char == '{' or char == '}' or char == ',' or char == '.' or char ==':':
        n_columna += 1
        cadena = cadena[1:]
        pointer = 0
    else:
        lista_errores.append(error(char, n_linea, n_columna))
        cadena = cadena[1:]
        pointer = 0
        n_columna +=1

return lista_lexemas
```

```
def armar_lexema(cadena):
    global  n_linea
    global  n_columna
    global  lista_lexemas

    lexema = ''
    pointer = ''

    for char in cadena:
        pointer += char
        if char == '\"':
            return lexema, cadena[len(pointer):]
        else:
            lexema += char
    return None, None
```

```python
def armar_numero(cadena):
    numero = ''
    pointer = ''
    is_decimal = False

    for char in cadena:
        pointer += char
        if char == '.':
            is_decimal = True
        if char == '"' or char == ' ' or char == '\n' or char == '\t':
            if is_decimal:
                return float(numero), cadena[len(pointer)-1:]
            else:
                return int(numero), cadena[len(pointer)-1:]
        else:
            numero += char
    return None, None
```

```python
def calculadora():
    global  instrucciones
    global  lista_lexemas

    operacion = ''
    n1 = ''
    n2 = ''

    while lista_lexemas:
        lexema = lista_lexemas.pop(0)
        if lexema.calculadora(None) == 'Operacion':
            operacion = lista_lexemas.pop(0)
        elif lexema.calculadora(None) == 'Valor1':
            n1 = lista_lexemas.pop(0)
            if n1.calculadora(None) == '[':
                n1 = calculadora()
        elif lexema.calculadora(None) == 'Valor2':
            n2 = lista_lexemas.pop(0)
            if n2.calculadora(None) == '[':
                n2 = calculadora()

        if operacion and n1 and n2:
            return Aritmetica(n1, n2, operacion,
                              f'Inicio: {operacion.get_fila()}: {operacion.get_column()}',
                              f'Fin: {n2.get_fila()}:{n2.get_column()}')

        elif operacion and n1 and operacion.calculadora(None) == ('Seno' or 'Coseno' or 'Tangente'):
            return trigonometrica(n1, operacion,
                                  f'Inicio: {operacion.get_fila()}: {operacion.get_column()}',
                                  f'Fin: {n1.get_fila()}:{n1.get_column()}')
    return None
```

```python
def calculadora_():
    global instrucciones
    while True:
        operacion = calculadora()
        if operacion:
            instrucciones.append(operacion)
        else:
            break
    return instrucciones

def getErrores():
    global lista_errores
    return lista_errores
```

El analizador.py toma la cadena de texto que se le ingresó al momento de pulsar el botón de abrir y con eso va analizando la cadena caracter por caracter, verificando si es número o una palabra reservada, de no ser ninguna lo toma como error, crea los números y lexemas con la unión de los caracteres para poder operarlos al final del código usando recursividad de ser necesario

Abstract:

Abstract.py:

```python
from abc import ABC, abstractmethod

class OperaGX(ABC):
    def __init__(self, fila, columna):
        self.fila = fila
        self.columna = columna
    @abstractmethod
    def calculadora(self, nodo):
        pass
    def get_text(self, nodo):
        pass
    @abstractmethod
    def get_fila(self):
        return self.fila
    @abstractmethod
    def get_column(self):
        return self.columna
    def get_node(self):
        return 'n'+str(self.contador)+str(self.contGraph)
    def get_nodeDef(self, index, cluster):
        self.contador = index
        self.contGraph = cluster
        return self.getGraphnode() + " [ shape=note, style=filled, fillcolor=\"#82589F\", label=""+ self.getGraphLabel() + ""]; \n"
```

El código proporcionado es una clase abstracta llamada "OperaGX" que hereda de la clase ABC del módulo "abc". La clase tiene un constructor que toma dos argumentos: "fila" y "columna", y dos métodos abstractos "calculadora" y "get_text", que deben ser implementados en cualquier subclase que herede de esta clase.

Además, la clase tiene tres métodos adicionales: "get_fila", "get_column" y "get_node", que devuelven la fila, columna y un identificador único de nodo en forma de cadena, respectivamente

getNum.py

```python
from Abstract.Abstract import OperaGX

class Numeros(OperaGX):
    def __init__(self, num, fila, columna):
        self.num = num
        super().__init__(fila, columna)
    def calculadora(self, nodo):
        return self.num
    def get_fila(self):
        return super().get_fila
    def get_column(self):
        return super().get_column
    def getGraphLabel(self):
        return str(self.valor)
```

Este código define una clase llamada "Numeros" que hereda de la clase abstracta "OperaGX" importada desde el archivo "Abstract.Abstract".

La clase "Numeros" tiene tres atributos: "num", "fila" y "columna". El atributo "num" representa un número y los atributos "fila" y "columna" representan su posición en una matriz o tabla.

La clase "Numeros" implementa el método abstracto "calculadora" de la clase "OperaGX". Este método devuelve el valor del atributo "num".

La clase "Numeros" también define tres métodos adicionales: "get_fila", "get_column" y "getGraphLabel". Los métodos "get_fila" y "get_column" devuelven las filas y columnas respectivamente. Estos métodos llaman a los métodos correspondientes de la clase base usando la función "super()". El método "getGraphLabel" devuelve una cadena que representa el valor del atributo "num".

Lex.py:

```python
from Abstract.Abstract import OperaGX

class Lex(OperaGX):
    def __init__(self, lexema, fila, columna):
        self.lexema = lexema
        super().__init__(fila, columna)
    def calculadora(self, nodo):
        return self.lexema
    def get_fila(self):
        return super().get_fila
    def get_column(self):
        return super().get_column
```

Este código define una clase **Lex** que hereda de la clase abstracta **OperaGX**. La clase **Lex** tiene un constructor que inicializa un objeto con un **lexema**, una **fila** y una **columna**. Además, la clase implementa el método abstracto **calculadora**, que devuelve el **lexema**.

La clase también tiene dos métodos **get_fila** y **get_column** que llaman a los métodos correspondientes de la clase padre utilizando **super()**.

Sin embargo, este código está incompleto ya que la clase **Lex** no implementa el método **getGraphLabel** que es un método abstracto de la clase padre **OperaGX**.

Errores.py

```python
from Abstract.Abstract import OperaGX

class error(OperaGX):
    def __init__(self, lexema, fila, columna):
        self.lexema = lexema
        super().__init__(fila, columna)
    def calculadora(self, no):
        no_ = f'\t\t"No.": {no}\n'
        desc = '\t\t"Descripcion-Token": {\n'
        lex = f'\t\t\t"Lexema": {self.lexema}\n'
        tipo = f'\t\t\t"Tipo": Error Lexico\n'
        fila = f'\t\t\t"Fila": {self.fila}\n'
        columna = f'\t\t\t"Columna": {self.columna}\n'
        fin = '\t\t}\n'

        return '\t{\n' + no_ + desc + lex + tipo + fila + columna + fin +'\t}'
    def get_fila(self):
        return super().get_fila
    def get_column(self):
        return super().get_column
```

El método **__init__** inicializa la instancia con un **lexema**, una **fila** y una **columna**.

El método **calculadora** toma un argumento **no** y retorna una cadena de caracteres que representa un objeto JSON que contiene información sobre el error léxico. La cadena de caracteres incluye información sobre el número de error, el lexema que causó el error, el tipo de error, la fila y la columna donde ocurrió el error.

Los métodos **get_fila** y **get_column** simplemente llaman a los métodos correspondientes de la clase base usando **super()**.

Instrucciones:

aritmética.py:

```python
from Abstract.Abstract import OperaGX

class Aritmetica(OperaGX):
    def __init__(self, NumIzq, NumDec, tipo,  fila, columna):
        self.NumIzq = NumIzq
        self.NumDec = NumDec
        self.tipo = tipo
        super().__init__(fila, columna)
```

```python
    def calculadora(self, nodo):
        NI = ''
        ND = ''
        if self.NumIzq != None:
            NI = self.NumIzq.calculadora(nodo)
        if self.NumDec != None:
            ND = self.NumDec.calculadora(nodo)
        if self.tipo.calculadora(nodo) == 'Suma':
            return float(NI) + float(ND)
        elif self.tipo.calculadora(nodo) == 'Resta':
            return float(NI) - float(ND)
        elif self.tipo.calculadora(nodo) == 'Multiplicacion':
            return float(NI) * float(ND)
        elif self.tipo.calculadora(nodo) == 'Division':
            return float(NI) / float(ND)
        elif self.tipo.calculadora(nodo) == 'Modulo':
            return float(NI) % float(ND)
        elif self.tipo.calculadora(nodo) == 'Potencia':
            return float(NI) ** float(ND)
        elif self.tipo.calculadora(nodo) == 'Raiz':
            return float(NI) ** (1/float(ND))
        elif self.tipo.calculadora(nodo) == 'Inverso':
            return 1/float(NI)
        else:
            return None
    def get_fila(self):
        return super().get_fila
    def get_column(self):
        return super().get_column
    def getGraphLabel(self):
        return str(self.tipo) + "\\n"+ str(self.operar(None))
```

define una clase llamada **Aritmetica**, que hereda de la clase abstracta **OperaGX**. Tiene un constructor que inicializa los atributos **NumIzq**, **NumDec**, **tipo**, **fila** y **columna**. También tiene un método **calculadora** que toma un objeto **nodo** como argumento y devuelve el resultado de la operación aritmética especificada por el atributo **tipo**, aplicada a los valores de **NumIzq** y **NumDec**.

También tiene métodos **get_fila** y **get_column** que devuelven los valores de **fila** y **columna**, respectivamente, y un método **getGraphLabel** que devuelve una cadena que representa la etiqueta del nodo del grafo que representa esta instancia de la clase.

Trigonométricas.py:

```python
import math
from Abstract.Abstract import OperaGX

class trigonometrica(OperaGX):
    def __init__(self, NumIzq, tipo, fila, columna):
        self.NumIzq = NumIzq
        self.tipo = tipo
        super().__init__(fila, columna)
    def calculadora(self, nodo):
        NI = ''
        if self.NumIzq != None:
            NI = self.NumIzq.calculadora(nodo)
        if self.tipo.calculadora(nodo) == 'Seno':
            resp = math.sin(math.radians(float(NI)))
            return resp
        elif self.tipo.calculadora(nodo) == 'Coseno':
            resp = math.cos(math.radians(float(NI)))
            return resp
        elif self.tipo.calculadora(nodo) == 'Tangente':
            resp = math.tan(math.radians(float(NI)))
            return resp
        else:
            return None
    def get_fila(self):
        return super().get_fila
    def get_column(self):
        return super().get_column
    def getGraphLabel(self):
        return str(self.tipo) + "\\n"+ str(self.operar(None))
```

La clase **trigonometrica** tiene un constructor que toma tres argumentos: **NumIzq**, **tipo**, **fila** y **columna**. **NumIzq** es un objeto que representa el número en el que se va a aplicar la operación trigonométrica. **tipo** es un objeto que representa el tipo de operación trigonométrica a realizar (seno, coseno o tangente). **fila** y **columna** son los números de línea y columna en el archivo de origen donde se encontró esta operación.

El método **calculadora** toma un argumento llamado **nodo**. Este método calcula la operación trigonométrica en función del tipo de operación (**tipo**) y el número de entrada (**NumIzq**). Si **tipo** es "Seno", el método utiliza la función **sin** de la librería **math** para calcular el seno del

ángulo en grados, si **tipo** es "Coseno", utiliza la función **cos** para calcular el coseno, y si **tipo** es "Tangente", utiliza la función **tan** para calcular la tangente. El resultado de la operación se devuelve como un número.

Los métodos **get_fila**, **get_column** y **getGraphLabel** son métodos de acceso que devuelven el número de línea, el número de columna y una etiqueta de gráfico, respectivamente. La etiqueta de gráfico es una cadena que representa la operación trigonométrica y el número de entrada.