27-3-2023

Manual técnico

Proyecto 1 LFP B+

Luis Antonio Castro Padilla
CARNET: 202109750

En la interfaz gráfica se creará la clase de pantalla principal, a la cual se le declarará que al momento de iniciar creará una ventana que tendrá un título específico, un tamaño de 600x375 px un fondo de pantalla verde menta, además tendrá la característica de que no se podrá ampliar, además llamará a la función pan1.

```
class Pantalla_Principal():
    def __init__(self):
        self.venMain = Tk()
        self.venMain.title("Proyecto 1, Luis Castro, 202109750")
        self.venMain.geometry("600x375")
        self.venMain.config(bg = "#1e272e")
        self.venMain.resizable(False, False)
        self.pan1()
        self.venMain.mainloop()
```

Pan1 creará dos frames distintos en los cuales irán los botones de archivo y ayuda, gracias a esta separación la interfaz se verá mejor la interfaz, arriba habrá un label en cada frame que indicará qué clase de acción mostrará, cada una tendrá sus botones requeridos en el proyecto, además habrá un Text en el cual se ingresará el texto de la carga inicial, se podrá modificar y ese será el texto a analizar.

```
def pan1(self):
    self.frameArchivo = Frame(self.venMain, bg="#00cec9", borderwidth=2, relief="ridge") #frameArchivo.grid(row = 0, column = 0)
    self.frameArchivo.place(x=0, y=0, width=300, height=400)

    self.Texto = ""

    Label(self.frameArchivo, bg="#006266", text="Archivo", width=50, height=1, fg="white", font=("Arial", 20, "bold")).pack() ##81ecec #00

    Button(self.frameArchivo, font=("Arial", 12), text="Abrir", width=20, bg ="#dfe4ea", command=self.abrirArchivo).place(x=50, y= 60)

    Button(self.frameArchivo, font=("Arial", 12), text="Guardar", width=20, bg ="#dfe4ea", command=self.guardar).place(x=50, y= 110)

    Button(self.frameArchivo, font=("Arial", 12), text="Guardar como", width=20, bg ="#dfe4ea", command=self.guardarComo).place(x=50, y= 1

    Button(self.frameArchivo, font=("Arial", 12), text="Analizar", width=20, bg ="#dfe4ea", command=self.lizador).place(x=50, y= 210)

    Button(self.frameArchivo, font=("Arial", 12), text="Errores", width=20, bg ="#dfe4ea", command=self.getErrores).place(x=50, y= 260)

    Button(self.frameArchivo, font=("Arial", 12), text="Salir", width=20, command=self.salir, bg ="#dfe4ea").place(x=50, y= 310)

    self.frameAyuda = Frame(self.venMain, bg="#e67e22", borderwidth=2, relief="ridge")
    self.frameAyuda.place(x=300, y=0, width=300, height=400)

    Label(self.frameAyuda, bg="#d35400", text="Ayuda", width=50, height=1, fg="white", font=("Arial", 20, "bold")).pack() ##e67e22, #ff3f3

    Button(self.frameAyuda, font=("Arial", 12), text="Manual de usuario", width=20, bg ="#dfe4ea", command=self.Monstrar_MU).place(x=50, y

    Button(self.frameAyuda, font=("Arial", 12), text="Manual técnico", width=20, bg ="#dfe4ea", command=self.Monstrar_MT).place(x=50, y= 1

    Button(self.frameAyuda, font=("Arial", 12), text="Temas de ayuda", width=20, bg ="#dfe4ea", command=self.ayuda).place(x=50, y= 160)

    txt_data = Text(self.frameAyuda, width=23, height= 8, name="txt_data")
    txt_data.place(x=50, y= 210)
```

La función de abrirArchivo utilizará la opción askopenfilename la cual permite abrir el explorador de archivos para poder realizar la carga de archivos, lo lee y posteriormente lo guarda en la variable self.filename, a txt_data (la caja de texto) se le insertará lo que se ingrese en el archivo de entrada; la función guardar como utiliza la opción asksaveasfile que abre el explorador de archivos y permite crear un nuevo doc con otro nombre; la función guardar recibe el mismo nombre pero sobrescribe la información que se cambió.

```
def abrirArchivo(self):
    x = ""
    Tk().withdraw()
    try:
        filename = filedialog.askopenfilename(title="Selecciona un documento")
        with open(filename, encoding='utf-8') as infile:
            x = infile.read()
            self.filename = filename
    except Exception as e:
        print(e)
        return

    self.Texto = x.strip()
    #print(self.Texto)

    txt_data = self.frameAyuda.nametowidget("txt_data")
    txt_data.delete(1.0, END)
    txt_data.insert(END, self.Texto)

def guardarComo(self):
    n_arch = filedialog.asksaveasfile(title="Guardar archivo como")
    if n_arch:
        txt_data = self.frameAyuda.nametowidget("txt_data")
        n_arch.write(txt_data.get('1.0', 'end-1c'))
        n_arch.close()

def guardar(self):
    if self.filename:
        cont = self.frameAyuda.nametowidget("txt_data").get('1.0', END)
        with open(self.filename, 'w', encoding='utf-8') as outfile:
            outfile.write(cont)
    else:
        print("error")
```

La función lizador llama a la función instrucción de la clase analizador y le ingresa la información que esté actualmente en la caja de texto, crea una variable que va a llamar a la función operar_() y en un ciclo for imprimirá las respuestas de la función operar; getErrores llamará a la función getErrores del archivo de analizador y le declarará como lista_errores, además usará un contador para poder incrementar la cantidad de errores que existen en el programa, abrirá un archivo llamado “ERRORES_202109750.json” en formato de leer y modificar texto y lo mostrará como la variable outfile, por último dentro de outfile escribirá una llave de inicio y un salto de línea, se utilizará un ciclo while para recorrer todos los errores que el programa encontró en la lista de errores, por último escribirá con el formato definido en la clase operar (en la cual se usó abstracción) en función del contador, hará un salto de línea, sumará el contador y por último terminará escribiendo la llave de salida; salir cierra la venta principal; Mostrar_MU y Mostrar_MT abrirán la ruta especificada en ruta_pdf, es distinta para cada manual pues son distintos archivos.

```
def lizador(self):
    instruccion(self.Texto)
    respuestas = operar_()
    for respuesta in respuestas:
        print(respuesta.operar(None))

def getErrores(self):
    lista_errores = getErrores()
    contador = 1
    with open('ERRORES_202109750.json', 'w') as outfile:
        outfile.write('{\n')
        while lista_errores:
            error = lista_errores.pop(0)
            outfile.write(str(error.operar(contador)) + ',\n')
            contador +=1
        outfile.write('}')

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

def Monstrar_MU(self):
    ruta_pdf = "file:///C:/Users/User/Documents/USAC/V%20Semestre/LFP/-LFP-Proyecto1_202109750/Manual_de_usuario_proyecto1.pdf"
    if os.name == 'nt':
        os.startfile(ruta_pdf)

def Monstrar_MT(self):
    ruta_pdf = "file:///C:/Users/User/Documents/USAC/V%20Semestre/LFP/-LFP-Proyecto1_202109750/Manual_tecnico_proyecto1.pdf"
    if os.name == 'nt':
        os.startfile(ruta_pdf)
```

Por último, la función Ayuda creará una nueva ventana arriba de la ventana principal, al igual que la principal tendrá un título, un tamaño y fondo de pantalla específicos y además tendrá la misma característica de que no se podrá modificar su tamaño, esta solamente contiene 3 Labels y un botón, el cual tiene como comando una función llamada cerrarAyuda lo cual permite cerrar esta nueva ventana, el fondo de la nueva venta será verde, los datos serán el nombre del estudiante, el carnet del mismo y la sección del curso.

```python
def ayuda(self):
    self.nventana = Toplevel(self.venMain)
    self.nventana.title("Informacion personal")
    self.nventana.geometry("400x200")
    self.nventana.config(bg = "#44bd32")
    self.nventana.resizable(False, False)

    Label(self.nventana, text="Luis Antonio Castro Padilla", fg="black", font=("Arial", 12), bg="#44bd32").pack()
    Label(self.nventana, text="Carnet: 202109750", fg="black", font=("Arial", 12), bg="#44bd32").pack()
    Label(self.nventana, text="Sección: B+", fg="black", font=("Arial", 12), bg="#44bd32").pack()

    def cerrarAyuda():
        self.nventana.destroy()

    Button(self.nventana, text="Cerrar", fg="white", font=("Arial", 12), bg="red", command=cerrarAyuda).pack()
```

Dentro del archivo analizador.py se creará una lista de palabras reservadas, las cuales tomará el programa si es que las lee en el archivo de entrada, se agregan a la lista de lexemas.

```python
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':         '}',
}

#Contiene la lista de las palabras reservadas
lexemas = list(palabras_reservadas.values())
```

Se declaran las siguientes variables globales y se las define posteriormente.

```
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 = []
```

En la función instrucción se recibirá la cadena de texto de la caja de la interfaz gráfica, se declarará la variable pointer que irá recorriendo la cadena, si la cadena lee una comilla doble tomará en cuenta que es una palabra reservada por lo tanto creará un lexema como objeto y lo añadirá a la lista de lexemas, de tal manera irá registrando todas las operaciones que pida el programa.

```
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
                #Lexema como clase
                l = Lexema(lexema, n_linea, n_columna)
                #Lexema a la lista de lexemaslista_lexemas.append(lexema)
                lista_lexemas.append(l)
                n_columna += len(lexema)+1
                pointer = 0
```

Siguiendo instrucción se verificará si la cadena es un número, en caso de ser así se creará un número con los datos que vaya ingresando la cadena, creará el objeto del número y lo añadirá a la lista de lexemas; si lee un inicio o final de corchete tomará en cuenta que hay una operación dentro de un valor de la primera operación, por lo cual también creará un lexema distinto para cada operación; si lee una tabulación incrementa el tamaño de la columna en 4 y si reconoce un salto de línea reiniciará el tamaño de la columna y añadirá en uno el valor de la fila; si lee un espacio vacío, llaves, punto, coma o dos puntos solamente añadirá un dato a la columna.

```
elif char.isdigit():
    token, cadena = armar_numero(cadena)
    if token and cadena:
        n_columna +=1
        #Lexema como clase
        n = Numero(token, n_linea, n_columna)
        #Lexema a la lista de lexemaslista_lexemas.append(lexema)
        lista_lexemas.append(n)
        n_columna += len(str(token)) +1
        pointer = 0

elif char == '[' or char == ']':
    #Lexema como clase
    c = Lexema(char, n_linea, n_columna)
    #Lexema a la lista de lexemas
    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
```

Finalmente, si no es ninguna de las anteriores lo tomará como error, en la lista de errores lo añadirá como objeto, tomando el carácter, su fila y columna.

```
117         else:
118             lista_errores.append(Errores(char, n_linea, n_columna))
119             cadena = cadena[1:]
120             pointer = 0
121             n_columna +=1
122 
123     return lista_lexemas
```

Con el siguiente código es que se crean los lexemas y números de la función instrucción.

```
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

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
```

La función operar tendrá la característica que es recursiva, es decir que si hay se requiere de usar una operación en uno de los valores volverá a llamar a la función operar y ahí volverá a crear el árbol con los nuevos datos que se crearon en el lexema anterior.

```
def operar():
    global  instrucciones
    global  lista_lexemas

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

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

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

        elif operacion and n1 and operacion.operar(None) == ('Seno' or 'Coseno' or 'Tangente'):
            return Trigonometricas(n1, operacion,
                                   f'Inicio: {operacion.getFila()}: {operacion.getColumna()}',
                                   f'Fin: {n1.getFila()}:{n1.getColumna()}')
    return None
```

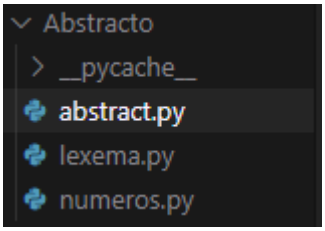
La función operar_ creará el grafo y además usará la función operar para crear las instrucciones con las cuales trabajará el programa su análisis.

```
def operar_():
    global instrucciones
    cont = 0
    strcadena = """digraph G {
            charset="utf-8";\n"""
    while True:
        operacion = operar()
        if operacion:
            strcadena += armar_arbolGraph(operacion,cont)
            cont +=1
            instrucciones.append(operacion)
        else:
            break
    strcadena += "}"
    with open('RESULTADOS_202109750.dot', 'w', encoding="utf-8") as f:
        f.write(strcadena)
    os.system('dot -Tpng RESULTADOS_202109750.dot -o RESULTADOS_202109750.pdf')
    print (strcadena)
```

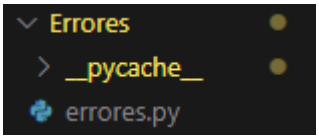
Con armar árbol se crearán los subgrafos en donde se almacenarán los nodos del graphviz, con armar nodo se revisa la definición del nodo que tiene como atributos el contador y el cluster, si es el valor 1 y no es una instancia entonces creará un nodo con el valor 1, el contador del cluster y el contador normal, entonces creará la relación con la cual se podrá crear el grafo, si es el valor 2 (o derecho) debe verificar que tampoco sea trigonométrica (pues no tiene valor 2), la función getErrores es sólo un trampolín que simplifica los errores para que sea más sencillo usarlo en el menú.

```
def armar_arbolGraph(instruccion, cont):
    contador = 0
    strcadena = "subgraph cluster"+ str(cont) + " { \n"
    strcadena += armar_nodo(instruccion, contador, cont)
    strcadena += "}"
    return strcadena

def armar_nodo(expresion, Xav, cluster):
    global contx
    cadenita = ""
    cadenita += expresion.getnodeDefinition(contx, cluster)
    contx +=1
    if not isinstance(expresion, Numero) and expresion.left != None:
        cadenita += armar_nodo(expresion.left, contx, cluster)
        cadenita += expresion.getGraphnode() + " -> " + expresion.left.getGraphnode() + "\n"

    if not isinstance(expresion, Numero) and not isinstance(expresion, Trigonometricas) and expresion.right != None:
        cadenita += armar_nodo(expresion.right, contx, cluster)
        cadenita += expresion.getGraphnode() + " -> " + expresion.right.getGraphnode() + "\n"

    return cadenita

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

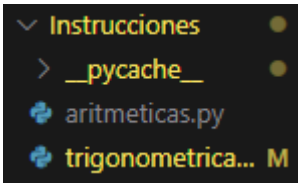
Dentro de la carpeta Abstracto se guardarán los archivos que usan abstracción para funcionar.



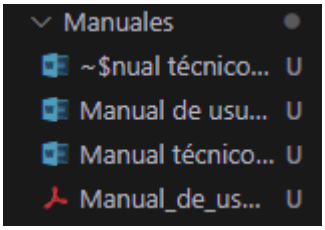
Dentro de la carpeta Errores se guardará el archivo donde se manejarán los errores.



Dentro de la carpeta de Instrucciones se encontrarán las operaciones aritméticas y trigonométricas.



Dentro de la carpeta de los manuales estarán los manuales de usuario y el técnico.



La clase Expression será abstracta, pues su función operar se podrá manejar de forma distinta para cada uno de los demás archivos que componen su carpeta y errores, pues cada una la manejará de forma distinta al igual que el getFila y getColumna, las otras 3 funciones se utilizarán para la creación del grafo, la definición del nodo creará los nodos según su correlativo y su número de cluster, el Label será abstracto también y el nodo como tal tomará el número del contador correlativo y cluster.

```python
from abc import ABC, abstractmethod

class Expression(ABC):
    def __init__(self, fila, columna):
        self.fila = fila
        self.columna = columna

    @abstractmethod
    def operar(self, arbol):
        pass

    @abstractmethod
    def getFila(self):
        return self.fila

    @abstractmethod
    def getColumna(self):
        return self.columna

    def getGraphnode(self):
        return "n"+str(self.correlativo)+str(self.cluster)

    def getGraphLabel(self):
        pass

    def getnodeDefinition(self, index, cluster):
        self.correlativo = index
        self.cluster = cluster
        return self.getGraphnode() + " [ shape=note, style=filled, fillcolor=\"#82589F\", label=\""+ self.getGraphLabel()
```

(parameter) arbol: Any

El número tomará la expresión de la clase abstracta, usará la misma fila y columna y añadirá el valor, el valor se usa en la función operar.

```python
from Abstracto.abstract import Expression

class Numero(Expression):
    def __init__(self, valor, fila, columna):
        self.valor = valor
        super().__init__(fila, columna)

    def operar(self, arbol):
        return self.valor

    def getFila(self):
        return super().getFila()

    def getColumna(self):
        return super().getColumna()

    def getGraphLabel(self):
        return str(self.valor)
```

El Lexema tomará la expresión de la clase abstracta, usará la misma fila y columna y añadirá el lexema, el lexema se usa en la función operar

```python
from Abstracto.abstract import Expression

class Lexema(Expression):
    def __init__(self, lexema, fila, columna):
        self.lexema = lexema
        super().__init__(fila, columna)

    def operar(self, arbol):
        return self.lexema

    def getFila(self):
        return super().getFila()

    def getColumna(self):
        return super().getColumna()

    def __str__(self):
        return self.lexema
```

En la clase Errores se usará además de fila y columna el lexema, para la abstracción de su función operar creará una serie de cadenas de texto en donde se le da un formato json al texto de errores, tomará un contador inicial, usará el lexema que acaba de encontrar, y lo ubicará en una fila y columna.

```python
from Abstracto.abstract import Expression

class Errores(Expression):
    def __init__(self, lexema, fila, columna):
        self.lexema = lexema
        super().__init__(fila, columna)

    def operar(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 getColumna(self):
        return super().getColumna()

    def getFila(self):
        return super().getFila()
```

Para la clase Aritmética se usará la expresión y usará el valor izquierdo, el valor derecho, el tipo, la fila y la columna como atributos.

```python
from Abstracto.abstract import Expression

class Aritmetica(Expression):
    def __init__(self, left, right, tipo, fila, columna):
        self.left = left
        self.right = right
        self.tipo = tipo
        super().__init__(fila, columna)
```

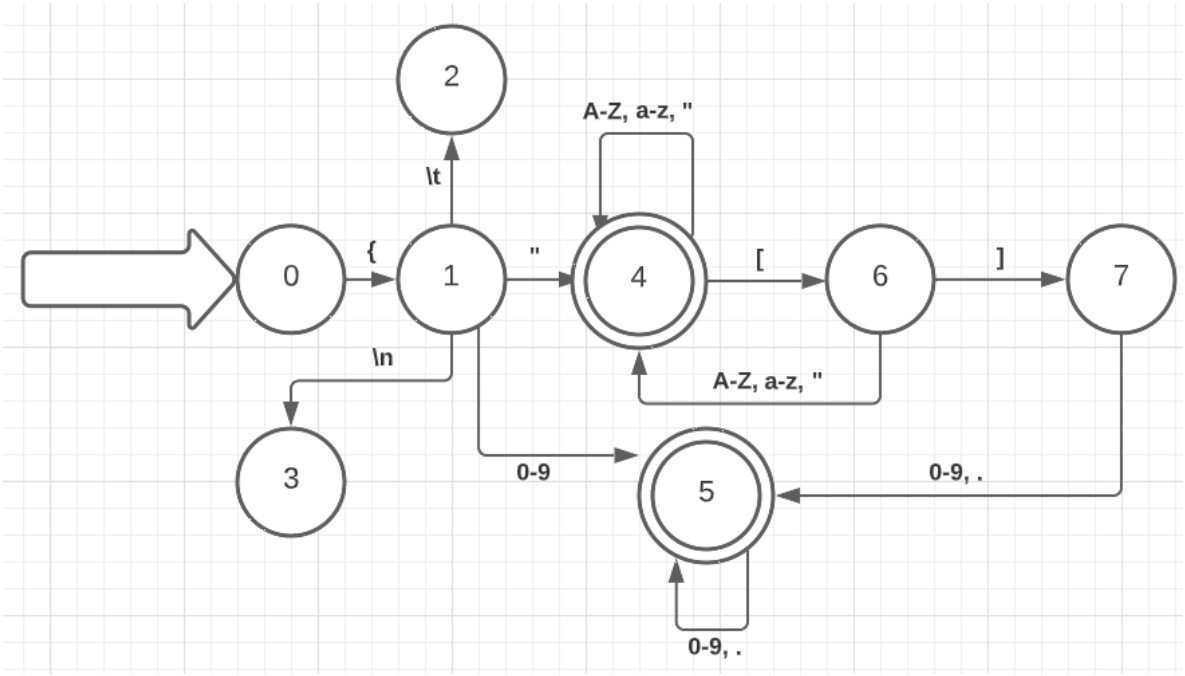
Como en la abstracción cada función puede tomarse de distinta forma lo que hace la función operar en las operaciones aritméticas y definir el valor izquierdo y derecho, con ello verifica si el tipo de operación es una de las solicitadas en el proyecto y por último retorna el resultado de la operación de ambos números. Además, llama a getFila y getColumna de la clase abstracta y usa el getGraphLabel para que en el grafo arriba aparezca cuál es el tipo de operación que se realizó.

```
def operar(self, arbol):
    leftValue = ''
    rightValue = ''

    if self.left != None:
        leftValue = self.left.operar(arbol)
    if self.right != None:
        rightValue = self.right.operar(arbol)

    if self.tipo.operar(arbol) == 'Suma':
        return leftValue + rightValue
    elif self.tipo.operar(arbol) == 'Resta':
        return leftValue - rightValue
    elif self.tipo.operar(arbol) == 'Multiplicacion':
        return leftValue * rightValue
    elif self.tipo.operar(arbol) == 'Division':
        return leftValue / rightValue
    elif self.tipo.operar(arbol) == 'Modulo':
        return leftValue % rightValue
    elif self.tipo.operar(arbol) == 'Potencia':
        return leftValue ** rightValue
    elif self.tipo.operar(arbol) == 'Raiz':
        return leftValue ** (1/rightValue)
    elif self.tipo.operar(arbol) == 'Inverso':
        return 1/leftValue
    else:
        return None

def getGraphLabel(self):
    return str(self.tipo) + "\\n"+ str(self.operar(None))

def getFila(self):
    return super().getFila()
```

Para las trigonométricas se toma la abstracción y se añade el importe de la librería math, toma como atributos izquierda, tipo, fila y columna, fila y columna se toman de la clase abstracta, su operación será verificar si el dato izquierdo es distinto de vacío le dará un valor y luego verificará si el tipo de operación es una de las solicitadas en el proyecto, de ser así, convertirá el dato a grados y luego hará la operación que se le solicita retornando el resultado, también se usará la abstracción del getGraphLabel para identificar qué tipo de operación es y se usará el getFila y getColumna de la clase abstracta.

```python
from Abstracto.abstract import Expression

class Trigonometricas(Expression):

    def __init__(self, left, tipo, fila, columna):
        self.left = left
        self.tipo = tipo
        super().__init__(fila, columna)

    def operar(self, arbol):
        leftValue=''

        if self.left != None:
            leftValue = self.left.operar(arbol)

        if self.tipo.operar(arbol) == 'Seno':
            resp = math.sin(math.radians(leftValue))
            return resp
        elif self.tipo.operar(arbol) == 'Coseno':
            resp = math.cos(math.radians(leftValue))
            return resp
        elif self.tipo.operar(arbol) == 'Tangente':
            resp = math.tan(math.radians(leftValue))
            return resp
        else:
            return None

    def getGraphLabel(self):
        return str(self.tipo) + "\\n"+ str(self.operar(None))

    def getFila(self):
        return super().getFila()

    def getColumna(self):
        return super().getColumna()
```

AFD



Método del arbol