Lenguajes Formales de Programación B-

MANUAL TÉCNICO

Proyecto 1

Sergio Andrés Larios Fajardo
CARNÉ 202111849

Índice

ESTRUCTURA DE LOS ARCHIVOS DEL PROGRAMA

El programa se encuentra organizado en varios archivos y carpetas. Estos se dividen en diferentes carpetas de acuerdo con su funcionamiento dentro del programa. Esta estructura permite una mejor organización y mantenimiento del código fuente.

Se utilizaron técnicas avanzadas de recursividad y abstracción para facilitar el manejo de los datos.

La recursividad permite al programa procesar grandes cantidades de información de manera eficiente y con un uso eficaz de los recursos del sistema.

La abstracción, por su parte, permite al programa separar la lógica del negocio de los detalles de implementación, lo que facilita su mantenimiento y evolución en el tiempo.

El programa está diseñado para manejar grandes volúmenes de datos de manera eficiente y escalable, gracias a su estructura organizada y el uso de técnicas avanzadas de programación como la recursividad y la abstracción.

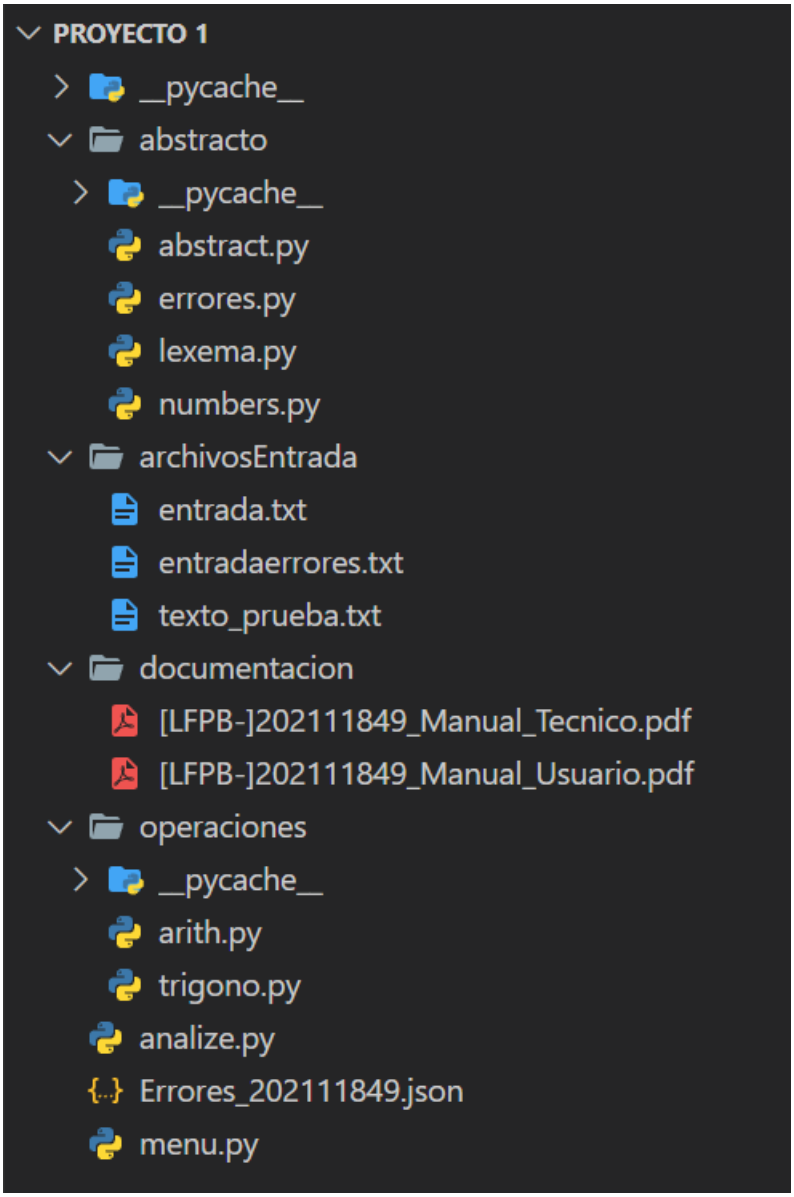

Ilustración 1 - Estructura de los archivos de la aplicación

ARCHIVOS PRINCIPALES

menu.py

El archivo es la ventana principal del programa. El código presenta un editor de texto básico implementado en Python con la biblioteca **Tkinter** para la interfaz gráfica de usuario. El editor admite la apertura y edición de archivos de texto, la opción de guardar archivos abiertos y guardar como, analizar el contenido de los archivos y visualizar errores, y acceder a temas de ayuda.

Imports

La sección de código importa el módulo "**analize**" y la biblioteca **Tkinter**, que se utiliza para crear la ventana principal y los elementos de la GUI. Luego se define la clase "**textEditor**", que tiene un constructor "init" que recibe la ventana principal como argumento.

```
from analize import *

import os
import tkinter as tk
from tkinter import filedialog
```

class textEditor

def __init__()

En el constructor, se configura el título de la ventana y se crea un cuadro de texto con la biblioteca **Tkinter**. El cuadro de texto se empaqueta con la función ***pack()*** y se expande para llenar todo el espacio disponible en la ventana. Se agrega un menú desplegable para las opciones de archivo y ayuda. El menú de archivo tiene opciones para abrir, guardar, guardar como, analizar y salir. El menú de ayuda tiene opciones para acceder al manual de usuario, al manual técnico y a temas de ayuda.

```
class textEditor:
    def __init__(self, window):
        self.window = window
        window.title("Proyecto 1 - Lenguajes Formales de Programación - Sergio
Larios")

        #Cuadro de texto
        self.textbox = tk.Text(window, font="Arial,18")
        self.textbox.pack(fill="both", expand=True)

        #Agregar label donde estarán los menús desplegables
        menubar = tk.Menu(window)

        #Agregar filemenu
        filemenu = tk.Menu(menubar, tearoff=0, bg="light green", font="Arial,
12")
```

```
        #Agregar opciones al menú de Archivo
        filemenu.add_command(label="Abrir", command=self.open_file)
        filemenu.add_command(label="Guardar", command=self.save_file)
        filemenu.add_command(label="Guardar como...", command=self.save_as)
        filemenu.add_command(label="Analizar", command=self.analizar)
        filemenu.add_command(label="Errores")
        filemenu.add_separator()
        filemenu.add_command(label="Salir", command=window.quit)

        #Agregar helpmenu
        helpmenu = tk.Menu(menubar, tearoff=0, bg="light green", font="Arial,
12")

        #Agregar opciones al menú de Ayuda
        helpmenu.add_command(label="Manual de Usuario")
        helpmenu.add_command(label="Manual Técnico")
        helpmenu.add_command(label="Temas de Ayuda", command=self.temas_de_ayuda)

        #Se agregan los menús a un menú de cascada cada uno
        menubar.add_cascade(label="Archivo", menu=filemenu, font="Arial, 12")
        menubar.add_cascade(label="Ayuda", menu=helpmenu, font="Arial, 12")
        window.config(menu=menubar)
```

def open_file()

Esta función en Python permite abrir un archivo de texto. Cuando se llama, se abre una ventana que permite al usuario seleccionar un archivo de texto. Si se selecciona un archivo, se verifica que no esté vacío y luego se lee el contenido del archivo.

Finalmente, se muestra el contenido en un cuadro de texto. Si por alguna razón no se puede abrir el archivo seleccionado, se muestra un mensaje de error en la consola indicando que se debe seleccionar un archivo válido.

```
#Método para abrir el archivo
    def open_file(self):
        try:
            self.filepath = filedialog.askopenfilename(title="Seleccionar archivo
de texto", filetypes=[("Archivos de texto", "*.txt")])
            #Se verifica que se haya abierto un archivo y que no esté vacío
            if self.filepath is not None:
                self.file = open(self.filepath, "r")
                lineas = self.file.read()
                #Se elimina lo que se encuentre en el cuadro de texto
                self.textbox.delete(1.0, tk.END)
                self.textbox.insert(tk.END, lineas)
```

```
        except Exception as e:
            print("Seleccione un archivo válido " + str(e))
```

def save_file()

El método verifica que se haya seleccionado un archivo previamente y que el contenido del cuadro de texto no esté vacío. Si se cumplen estas condiciones, el método abre el archivo en modo escritura, obtiene todo el texto del cuadro de texto y escribe ese texto en el archivo. Si ocurre algún error al guardar el archivo, se muestra un mensaje de error y se llama al método **"save_as()"** que permite seleccionar una nueva ubicación y nombre de archivo para guardar.

```
#Método para guardar el archivo
    def save_file(self):
        try:
            if self.filepath is not None:
            #Se verifica que el archivo no esté vacío
                file = open(self.filepath, "w")
                #Obtiene todos los caracteres del cuadro de texto
                lineas = self.textbox.get(1.0, tk.END)
                file.write(lineas)
                file.close()
        except Exception as e:
            print(e)
            self.save_as()
```

def save_as()

La función utiliza una ventana de diálogo para que el usuario pueda seleccionar la ubicación y el nombre del archivo de texto que desea guardar. Si el usuario selecciona una ubicación y un nombre para el archivo, se escribe el contenido del cuadro de texto en el archivo. Si el archivo ya existe, se sobrescribirá.

```
#Método para guardar como
    def save_as(self):
        file = filedialog.asksaveasfile(title = "Guardar archivo como...",
filetypes=[("Archivos de texto", "*.txt")], defaultextension=".txt")
        #Se verifica que el archivo no esté vacío
        if file is not None:
            self.saved_file = file.name
            lineas = self.textbox.get(1.0, tk.END)
            file.write(lineas)
            file.close()
```

def analizar()

Define un método llamado analizar que se encarga de leer el contenido del archivo seleccionado por el usuario previamente mediante el método **open_file**. Una vez leído el contenido del archivo, se pasa este texto a una función llamada **instruccion** para que analice el contenido y genere una serie de objetos que se almacenan en la variable respuestas.

```
#Método para analizar el archivo
    def analizar(self):
        file = open(self.filepath, "r")
        lineas = file.read()

        instruccion(lineas)
        respuestas = evaluate_()
        for respuesta in respuestas:
            print(respuesta.evaluate(None))
```

def errores()

La función **errores** es responsable de crear un archivo **JSON** que contiene información sobre los errores encontrados durante el análisis léxico del código fuente.

```
def errores(self):
        lista_errores = getErrores()
        contador = 1
        with open('Errores_202111849.json', 'w') as outfile:
            outfile.write('{\n')
            while lista_errores:
                error = lista_errores.pop(0)
                outfile.write(str(error.evaluate(contador)) + ',\n')
                contador +=1
            outfile.write('}')
```

def abrir_manual_usuario()

La función se encarga de abrir el archivo PDF del manual de usuario, que se encuentra en la carpeta **documentación** que a su vez se encuentra en la misma carpeta que el programa.

```
def abrir_manual_usuario(self):
        try:
            path = "documentacion\[LFPB-]202111849_Manual_Usuario.pdf"
            os.startfile(path)
        except Exception as e:
            print("No se pudo abrir el archivo " + str(e))
```

def abrir_manual_tecnico()

La función se encarga de abrir el archivo PDF del manual de usuario, que se encuentra en la carpeta **documentación** que a su vez se encuentra en la misma carpeta que el programa.

```
def abrir_manual_tecnico(self):
        try:
            path = "documentacion\[LFPB-]202111849_Manual_Tecnico.pdf"
            os.startfile(path)
        except Exception as e:
            print("No se pudo abrir el archivo " + str(e))
```

def temas_de_ayuda()

Define un método llamado **temas_de_ayuda**, el cual muestra una ventana emergente con una lista de temas de ayuda.

Se crea una nueva ventana usando **tk.Toplevel(self.window)**. Luego se configura el título y la propiedad de redimensionamiento de la ventana usando **ventana_ayuda.title("Temas de ayuda")** y **ventana_ayuda.resizable(False, False)**, respectivamente. Finalmente, se configura el color de fondo de la ventana con **ventana_ayuda.config(bg="light blue").**

Se crean varios cuadros de texto utilizando la función **tk.Label**, cada uno con un texto específico y un formato de fuente distinto. En cada **tk.Label** se especifica el texto, la fuente y tamaño, el espaciado interno horizontal (**padx**) y vertical (**pady**), y el color de fondo. Luego se utiliza la función **pack()** para posicionar cada **tk.Label** dentro de la ventana emergente, en el orden en que fueron creados.

```
    #Método para mostrar los temas de ayuda
    def temas_de_ayuda(self):
        ventana_ayuda = tk.Toplevel(self.window)
        ventana_ayuda.title("Temas de ayuda")
        ventana_ayuda.resizable(False, False)
        ventana_ayuda.config(bg="light blue")

        #Cuadros de texto
        label1 = tk.Label(ventana_ayuda, text="Temas de ayuda", font=('Arial',
16, 'bold'), padx=8, pady=8, bg="light blue")
        label2 = tk.Label(ventana_ayuda, text="Sergio Andrés Larios Fajardo",
font=('Algerian', 12), padx=5, pady=5, bg="light blue")
        label3 = tk.Label(ventana_ayuda, text="202111849", font=('Broadway', 12),
padx=5, pady=5, bg="light blue")
        label4 = tk.Label(ventana_ayuda, text="Lenguajes Formales de
Programación", font=('MS Gothic', 12), padx=5, pady=5, bg="light blue")
        label5 = tk.Label(ventana_ayuda, text="Sección B-", font=('Elephant',
12), padx=5, pady=5, bg="light blue")
        label6 = tk.Label(ventana_ayuda, text="Facultad de Ingeniería",
font=('Forte', 12), padx=5, pady=5, bg="light blue")
```

```
        label7 = tk.Label(ventana_ayuda, text="Universidad de San Carlos de
Guatemala", font=('Kristen ITC', 12), padx=15, pady=5, bg="light blue")
        label8 = tk.Label(ventana_ayuda, text="Ingeniería en Ciencias y
Sistemas", font=('SimSun', 12), padx=5, pady=5, bg="light blue")
        label9 = tk.Label(ventana_ayuda, text=" ", padx=5, pady=5, bg="light
blue")

        #Posicionamiento de los cuadros de texto
        label1.pack()
        label2.pack()
        label3.pack()
        label4.pack()
        label5.pack()
        label6.pack()
        label7.pack()
        label8.pack()
        label9.pack()
```

iniciar la ventana

crea una ventana principal de **tkinter**, con un tamaño de 800x600 píxeles. Luego, crea una instancia de la clase **textEditor** (que probablemente sea una clase personalizada definida en otro lugar del código) y la asocia con la ventana principal **root**. Finalmente, se inicia el ciclo principal de eventos de **tkinter** con el método **mainloop(),** lo que permite que la ventana sea interactiva y responda a las acciones del usuario. En resumen, este código crea una aplicación de editor de texto que se ejecutará en una ventana **tkinter** con un tamaño de 800x600 píxeles.

```
root = tk.Tk()
root.geometry("800x600")
editor = textEditor(root)
root.mainloop()
```

analize.py

El código consiste en tres partes: la primera importa módulos y define diccionarios y listas, la segunda procesa una cadena de entrada y construye una lista de lexemas (componentes de la cadena) y la tercera evalúa los lexemas para ejecutar una operación matemática.

Las primeras líneas importan las funciones del módulo "**arith**" y "**trigono**" del paquete "**operaciones**", así como las funciones del módulo "**lexema**", "**numbers**" y "**errores**" del paquete "**abstracto**".

```
from operaciones.arith import *
from operaciones.trigono import *
from abstracto.lexema import *
from abstracto.numbers import *
from abstracto.errores import error
```

Se define un diccionario llamado "**palabras_reservadas**", que asocia cadenas de caracteres con nombres simbólicos utilizados por el programa para identificar ciertas palabras clave. Estos nombres simbólicos se usan más adelante para hacer referencia a estas palabras clave.

```
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':         '}',
}
```

Se define una lista llamada "**lexemas**" que contiene los valores del diccionario "**palabras_reservadas**". Esto sugiere que el programa utiliza esta lista para identificar los tokens o lexemas en una secuencia de caracteres, que se usarán posteriormente en el análisis sintáctico. Luego, se definen cuatro globales y se les asignan valores iniciales.

```
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 instrucción(cadena)

La función **“instrucción(cadena)”** es parte de un analizador léxico que procesa una cadena de entrada y genera una lista de lexemas que representan los tokens que se han identificado en la cadena.

La función itera sobre la cadena de entrada y por cada caracter que encuentra, realiza una acción correspondiente. El valor de la cadena de entrada se va actualizando a medida que se procesan los caracteres. La función retorna la lista de lexemas generada durante el proceso.

```
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 = lexem(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 ==
'4' or char == '5' or char == '6' or char == '7' or char == '8' or char == '9':
            token, cadena = armar_numero(cadena)
            if token and cadena:
                n_columna +=1
                n = numbers(token, n_linea, n_columna)
                lista_lexemas.append(n)
```

```
                n_columna += len(str(token)) +1
                pointer = 0

        elif char == '[' or char == ']':
            c = lexem(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)

La función toma como entrada una cadena de caracteres "**cadena**" que contiene un lexema rodeado de comillas dobles. La función devuelve el lexema extraído de la cadena y la cadena restante después del lexema.

```
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)

La función recibe una cadena como argumento y la analiza para encontrar un número. El número puede ser entero o decimal, y puede estar seguido de un carácter que no es un dígito, como un espacio, una comilla o un salto de línea.

```
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
```

def evaluate()

Su función es evaluar una serie de instrucciones que se han convertido en una lista de objetos de tipo **Lexema** y **Numero**.

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

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

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

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

        elif operacion and n1 and operacion.evaluate(None) == ('Seno' or 'Coseno'
or 'Tangente'):
            return trigono(n1, operacion,
                                    f'Inicio: {operacion.get_row()}:
{operacion.get_column()}',
                                    f'Fin: {n1.get_row()}:{n1.get_column()}')

    return None
```

def evaluate()

Esta función evalúa la lista de lexemas y genera una lista de instrucciones válidas para ser ejecutadas posteriormente.

```
def evaluate_():
    global instrucciones
    while True:
        operacion = evaluate()
        if operacion:
            instrucciones.append(operacion)
        else:
```

```
            break
    return instrucciones
```

def getErrores()

Devuelve la lista de errores que se han acumulado durante la ejecución del programa. La variable **lista_errores** se declara global al principio del programa y se utiliza en varias funciones para agregar cualquier error encontrado durante el análisis léxico o sintáctico del código.

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

CARPETA "ABSTRACTO"

abstract.py

El código define una clase abstracta **expression** que hereda de la clase **ABC** de la librería **abc**. La clase abstracta **expression** tiene dos métodos abstractos **evaluate** y **get_row**, y un método no abstracto **get_column**.

```
from abstracto.abstract import expression
```

class error(expression)

El método **__init__** es el constructor de la clase **expression** y recibe dos argumentos: **row** y **column**, que corresponden a la fila y columna donde se encuentra la expresión

El decorador **@abstractmethod** indica que los métodos **evaluate** y **get_row** son abstractos y deben ser implementados en las subclases de **expression**.

El método **evaluate** toma un argumento **node** y debe devolver el resultado de evaluar la expresión.

El método **get_row** devuelve la fila donde se encuentra la expresión y el método **get_column** devuelve la columna donde se encuentra la expresión.

```
class expression(ABC):
    def __init__(self, row, column):
        self.row = row
        self.column = column

    #Método abstracto para calcular
    @abstractmethod
    def evaluate(self, node):
        pass

    @abstractmethod
    def get_row(self):
        return self.row

    @abstractmethod
    def get_column(self):
        return self.column
```

errores.py

La clase errores es la encargada de identificar y almacenar los errores para posteriormente mostrarlos en el archivo Errores_202111849.json

```
from abstracto.abstract import expression

class error(expression):
```

```
    def __init__(self, lexema, row, column):
        self.lexema = lexema
        super().__init__(row, column)

    def evaluate(self, number):
        num = f'\t\t"No.": {number}\n'
        desc = '\t\t"Descripcion-Token": {\n'
        lexem = f'\t\t\t"Lexema": {self.lexema}\n'
        type = f'\t\t\t"Tipo": Error Lexico\n'
        row = f'\t\t\t"Fila": {self.row}\n'
        column = f'\t\t\t"Columna": {self.column}\n'
        end = '\t\t}\n'

        return '\t{\n' + num + desc + lexem + type + row + column + end +'\t}'

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

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

lexeme.py

Este archivo define una clase llamada **lexem** que hereda de la clase abstracta **expression**. La clase **lexem** tiene un constructor que toma tres argumentos: **lexema**, **row** y **column**. Estos valores se utilizan para inicializar las variables **self.lexema**, **self.row** y **self.column**, respectivamente, mediante el método **super().**

```
from abstracto.abstract import expression

class lexem(expression):
    def __init__(self, lexema, row, column):
        self.lexema = lexema
        super().__init__(row, column)

    def evaluate(self, node):
        return self.lexema

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

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

numbers.py

Este código define una clase llamada **numbers** que hereda de la clase abstracta **expression**. Esta clase se utiliza para representar un número en una expresión matemática.

La clase **numbers** tiene tres atributos: **num**, **row** y **column**. El atributo **num** representa el valor del número, mientras que los atributos **row** y **column** representan la fila y columna en la que se encuentra el número en el código fuente.

La clase tiene tres métodos:

El método **__init__** inicializa los atributos **num**, **row** y **column** de la clase.

El método **evaluate** devuelve el valor del número.

Los métodos **get_row** y **get_column** devuelven la fila y columna del número, respectivamente.

```
from abstracto.abstract import expression

class numbers(expression):
    def __init__(self, num, row, column):
        self.num = num
        super().__init__(row, column)

    def evaluate(self, node):
        return self.num

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

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

CARPETA "OPERACIONES"

arith.py

Este archivo define una clase **arith** que hereda de la clase abstracta **expression** del módulo **abstracto.abstract**. Esta clase se utiliza para representar expresiones aritméticas y realizar cálculos.

```
from abstracto.abstract import expression
```

class arith(expression)

La clase arith tiene los siguientes atributos:

- **left**: un objeto de la clase **expression** que representa el lado izquierdo de la expresión aritmética.
- **right**: un objeto de la clase **expression** que representa el lado derecho de la expresión aritmética.
- **type**: un objeto de la clase **expression** que representa el tipo de operación aritmética a realizar.
- **row**: la fila donde se encuentra la expresión aritmética en el código fuente.
- **column**: la columna donde se encuentra la expresión aritmética en el código fuente.

La clase arith tiene los siguientes métodos:

- **evaluate**: es un método que toma un parámetro **node** y devuelve el resultado de la expresión aritmética.
- **get_row:** es un método que devuelve el valor del atributo row**.**
- **get_column**: es un método que devuelve el valor del atributo **column**.

```
class arith(expression):
    def __init__(self, left, right, type, row, column):
        self.left = left
        self.right = right
        self.type = type
        super().__init__(row, column)

    def evaluate(self, node):
        left_num = ''
        right_num = ''

        if self.left is not None:
            left_num = self.left.evaluate(None)
        if self.right is not None:
            right_num = self.right.evaluate(None)

        if self.type.evaluate(node) == 'Suma':
            return float(left_num) + float(right_num)
```

```
        elif self.type.evaluate(node) == 'Resta':
            return float(left_num) - float(right_num)

        elif self.type.evaluate(node) == 'Multiplicacion':
            return float(left_num) * float(right_num)

        elif self.type.evaluate(node) == 'Division':
            return float(left_num) / float(right_num)

        elif self.type.evaluate(node) == 'Modulo':
            return float(left_num) % float(right_num)

        elif self.type.evaluate(node) == 'Potencia':
            return float(left_num) ** float(right_num)

        elif self.type.evaluate(node) == 'Raiz':
            return float(left_num) ** (1/float(right_num))

        elif self.type.evaluate(node) == 'Inverso':
            return float(left_num) ** -1
        else:
            return None

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

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

trigono.py

Este archivo define una clase **trigono** que hereda de la clase abstracta **expression** del módulo **abstracto.abstract**. Esta clase se utiliza para representar expresiones aritméticas y realizar cálculos.

```
import math
from abstracto.abstract import expression
```

class trigono(expression)

La clase trigono tiene los siguientes atributos:

- **left**: un objeto de la clase **expression** que representa el lado izquierdo de la expresión aritmética.

- **type**: un objeto de la clase **expression** que representa el tipo de operación aritmética a realizar.
- **row**: la fila donde se encuentra la expresión aritmética en el código fuente.
- **column**: la columna donde se encuentra la expresión aritmética en el código fuente.

La clase trigono tiene los siguientes métodos:

- **evaluate**: es un método que toma un parámetro **node** y devuelve el resultado de la expresión aritmética.
- **get_row:** es un método que devuelve el valor del atributo row**.**
- **get_column**: es un método que devuelve el valor del atributo **column**.