MANUAL TÉCNICO

PROYECTO 01 LENGUAJES FORMALES
Y DE PROGRAMACIÓN

Marzo 2023

Sheila Amaya 202000558

Descripción de la practica

Este programa fue desarrollado con el lenguaje de programación Python, el objetivo principal de este programa fue crear una herramienta la cual sea capaz de reconocer un lenguaje, dado por medio de un analizador léxico el cual cumple con las reglas establecidas, manejando la lectura y escritura de archivos para el manejo de la información. A través de un entorno gráfico.

¿Qué paradigma se utilizó?

Se utilizo el paradigma imperativo o de procedimientos , este es un método que nos permite desatollar programas a través de procedimientos y además se utilizó POO (Programación Orientada a Objetos), este es un modelo o estilo de programación que proporciona unas guías acerca de cómo trabajar con él y que este está basado en el concepto de clases y objetos.

Lógica del programa

Se utilizaron varios módulos para realizar el funcionamiento del programa de estos se presenta a continuación una breve descripción de las clases, métodos y funciones que fueron implementadas para la realización del programa.

MAIN
Estas líneas definen un bloque de ejecución condicional en Python que permite verificar si el archivo actual se está ejecutando como programa principal o si ha sido importado como modulo por otro archivo.

```
if __name__ == '__main__':
    funciones = Funciones()
    funciones.principal()
```

CLASE ABSTRACT

define una clase base abstracta Expresión con dos métodos abstractos.

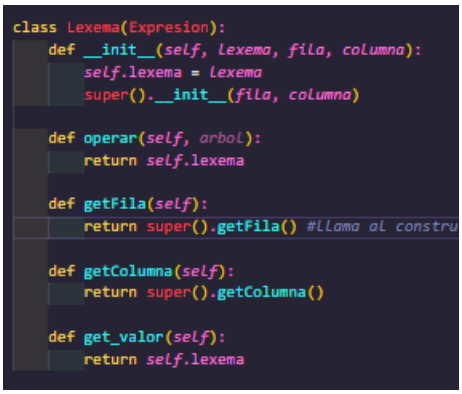
- operar () : toma un árbol parámetro y realiza una operación sobre él. La operación específica no está definida en la Expresión clase, pero será implementada por las subclases.
- getFila () :Devuelve el valor de la fila.
- getColumna(): Devuelve el valor de la Columna.

```
class Expresion(ABC): #al hererdar ABC se
    def __init__(self, fila, columna):
        self.fila = fila
        self.columna = columna

    @abstractmethod #ind. que este método
    def operar(self, arbol):
        pass

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

    @abstractmethod
    def getColumna(self):
        return self.columna # devuelve el
```

CLASE LEXEMA

hereda de la clase abstracta Expresión y define su propia implementación de los métodos abstractos.

- operar () : toma un árbol parámetro y realiza una operación sobre él. La operación específica no está definida en la Expresión clase, pero será implementada por las subclases.
- getFila () :Devuelve el valor de la fila.
- getColumna(): Devuelve el valor de la Columna.

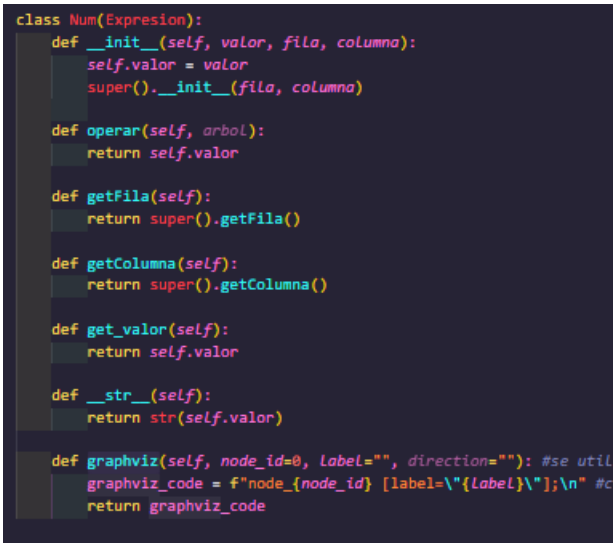


CLASE NUM

hereda de la clase abstracta Expresión y define su propia implementación de los métodos abstractos.

- operar () : toma un árbol parámetro y realiza una operación sobre él. La operación específica no está definida en la Expresión clase, pero será implementada por las subclases.
- getFila () :Devuelve el valor de la fila.
- getColumna(): Devuelve el valor de la Columna.

- get_valor(): simplemente devuelve el valor de la variable de instancia
- graphviz(): esta línea define el método graphviz de la clase Num. Este método se utiliza para generar código Graphviz para visualizar el árbol de expresión que contiene el objeto Num.



CLASE ARITMETICAS

Representa operaciones aritméticas entre dos operandos, L y R , con un tipo de dato

- operar(): Este método calcula y devuelve el resultado de la operación aritmética definida por los operadores Ly R
- getFila () : devuelven las filas de la expresión.
- getColumna(): devuelven la columna de la expresión.
- graphviz(): genera un código Graphviz que representa el árbol de la expresión.

```
class Aritmetica(Expresion): #hereda de la clase

    def __init__(self,L ,R , tipo, fila, columna):
        self.L = L
        self.R = R
        self.tipo = tipo
        self.valor = None
        super().__init__(fila, columna) #inicializa los atributos f y c en base al con


    def operar(self, arbol):
        Lvalue = ''
        Rvalue = ''

        if self.L != None: #verifica si el operando izquierdo es diferente de nada
            Lvalue = self.L.operar(arbol) # si L no es none llama al metodo
        if self.R != None:
            Rvalue = self.R.operar(arbol)

        if self.tipo.operar(arbol) == 'Suma': #verifica el tipo de operacion
            resultado = Lvalue + Rvalue # si es una + calcula el resultado
            self.valor = resultado #asignando el resultado a valor
            return resultado
        elif self.tipo.operar(arbol) == 'Resta':
            resultado = Lvalue - Rvalue
            self.valor = resultado
            return resultado
        elif self.tipo.operar(arbol) == 'Multiplicacion':
            resultado = Lvalue * Rvalue
            self.valor = resultado
            return resultado
        elif self.tipo.operar(arbol) == 'Division':
            resultado =  Lvalue / Rvalue
            self.valor = resultado
            return resultado
        elif self.tipo.operar(arbol) == 'Modulo':
            resultado =  Lvalue % Rvalue
            self.valor = resultado
            return resultado
        elif self.tipo.operar(arbol) == 'Potencia':
            resultado =  Lvalue ** Rvalue
            self.valor = resultado
            return resultado
        elif self.tipo.operar(arbol) == 'Raiz':
            resultado =  Lvalue ** (1/Rvalue)
            self.valor = resultado
            return resultado
        elif self.tipo.operar(arbol) == 'Inverso':
            resultado =  1/Lvalue
            self.valor = resultado
            return resultado
        else:
            return 0
```

CLASE TRIGONOMETRICAS

Representa operaciones trigonométricas entre dos operandos, L , con un tipo de dato

- operar(): Este método calcula y devuelve el resultado de la operación aritmética definida por los operadores L
- getFila () : devuelven las filas de la expresión.
- getColumna(): devuelven la columna de la expresión.
- graphviz(): genera un código Graphviz que representa el árbol de la expresión.

```python
class Trigonometrica(Expresion):

    def __init__(self, L, tipo, fila, columna):
        self.L = L
        self.tipo = tipo
        self.valor = ""
        super().__init__(fila, columna)


    def operar(self, arbol): #devuelve un resultado al aplicar una func. trig.
        Lvalue = ''

        if self.L != None:
            Lvalue = self.L.operar(arbol)

        if self.tipo.operar(arbol) == 'Seno': #comprueba el tipo de ope
            resultado = sin(Lvalue)
            self.valor = resultado
            return resultado
        elif self.tipo.operar(arbol) == 'Coseno':
            resultado = cos(Lvalue)
            self.valor = resultado
            return resultado
        elif self.tipo.operar(arbol) == 'Tangente':
            resultado = tan(Lvalue)
            self.valor = resultado
            return resultado
        else:
            return 0

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

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

    def __str__(self):
        return f"{self.tipo.operar(None)}\n {self.valor}" #devuelve una cadena que contie

    def graphviz(self, node_id=0, label="", direction=""):
        graphviz_code = f"node_{node_id} [label=\"{label}\"];\n"
        if self.L: #si la expre. tiene un hijo izquierdo
            left_label = str(self.L)
            left_node_id = f"{node_id}_izq"
            if direction:
                left_label += f" ({direction})"
            graphviz_code += self.L.graphviz(left_node_id, left_label)
            graphviz_code += f"node_{node_id} -> node_{left_node_id};\n"
        return graphviz_code #devuelve la cadena que contiene la representacion
```

CLASE ANALIZADOR

contiene funciones y variables relacionadas con la creación de un lenguaje de programación específico.

- instruccion(): se define para tokenizar una cadena de código, analizarla y generar un árbol de sintaxis abstracta
- armar_Lexema () : función para tokenizar un literal de cadena.
- armar_num (): función para tokenizar el número, y si el carácter es una comilla llama a armar_lexema.

- operar(): Inicializa algunas variables, luego establece algunas variables locales, la función ingresa en un bucle y extrae el primer elemento de lista_lexema.
- operar_R():Devuelve las operaciones con los valores correspondientes si es una operación trigonométrica o aritmética.
- graficar(): genera un gráfico en formato .pdf que representa el árbol generado a partir de una cadena de entrada que contiene una serie de instrucciones en un lenguaje de programación específico.

```
#palabras reservadas son los lexemas
#token | lexema
reservadas = {
    'ROPERACION'        :   'Operacion',
    'RVALOR1'           :   'Valor1',
    'RVALOR2'           :   'Valor2',
    'RSUMA'             :   'Suma',
    'RRESTA'            :   'Resta',
    'RMULTIPLICACION'   :   'Multiplicacion',
    'RDIVISION'         :   'Division',
    'RPOTENCIA'         :   'Potencia',
    'RRAIZ'             :   'Raiz',
    'RINVERSO'          :   'Inverso',
    'RSENO'             :   'Seno',
    'RCOSENO'           :   'Coseno',
    'RTANGENTE'         :   'Tangente',
    'RMOD'              :   'Mod',
    'RTEXTO'            :   'Texto',
    'RCOLORFONDONODO'   :   'Color-Fondo-Nodo',
    'RCOLORFUENTENODO'  :   'Color-Fuente-Nodo',
    'RFORMANODO'        :   'Forma-Nodo',
    'COMA'              :   ',',
    'PUNTO'             :   '.',
    'DPUNTO'            :   ':',
    'CORI'              :   '[',
    'CORD'              :   ']',
    'LLAVEI'            :   '}',
    'LLAVED'            :   '{',
}

Lexemas = list(reservadas.values())
global n_linea
global n_columna
global instrucciones
global lista_lexema
global lista_errores
global estilo

n_linea = 1
n_columna = 1
lista_lexema = []
instrucciones = []
lista_errores = []
estilo = ""

def instruccion(cadena):
    global n_linea
    global n_columna
    global lista_lexema
    global lista_errores
    global estilo
```

CLASE ERRORES

Se usa para representar los errores encontrados durante el análisis o la interpretación del código de entrada.

```
class Error:
    def __init__(self, lexema, linea, columna, tipo='ERROR'):
        self.lexema = lexema
        self.linea = linea
        self.columna = columna
        self.tipo = tipo
```

CLASE ESTILO

Se utiliza para almacenar información sobre el estilo que se utilizará en el gráfico generado por la graficar() función.

```
class Estilo:
    def __init__(self, texto, color_nodo, color_fuente, forma_nodo):
        self.texto = texto
        self.color_nodo = color_nodo
        self.color_fuente = color_fuente
        self.forma_nodo = forma_nodo

    def get_texto(self):
        return self.texto

    def get_color_nodo(self):
        return self.color_nodo

    def get_color_fuente(self):
        return self.color_fuente

    def get_forma_nodo(self):
        return self.forma_nodo
```

CLASE INTERFAZ

El módulo interfaz contiene una clase funciones que establece como crear la interfaz grafica y todos sus componentes.

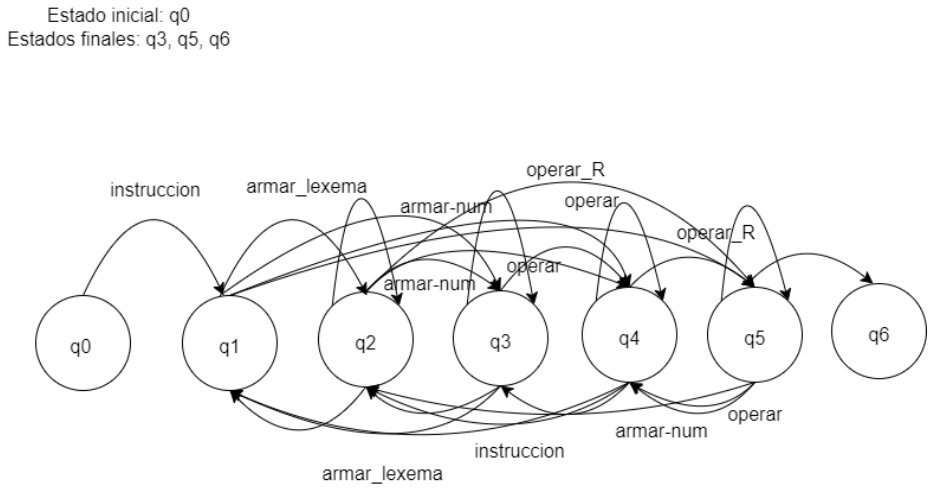
- salir (): se encarga de cerrar una ventana en una interfaz gráfica de usuario.
- Secundaria (): crea una ventana secundaria y la muestra en la pantalla. La ventana secundaria incluye un cuadro de texto y un menú que permite abrir, guardar y analizar un archivo de texto, entre otras opciones.
- Abrir(): Este método permite abrir un archivo seleccionado por el usuario y mostrar su contenido en un cuadro de texto.
- Analizar(): Recibe un string contenido que representa el código fuente a analizar.
- Escribir_error(): toma los errores almacenados en la lista global lista_errores y los escribe en un archivo JSON en la ruta
- Guardar(): El contenido de un cuadro de texto en un archivo. Primero verifica si hay una ruta de archivo almacenada, en cuyo caso utiliza esa ruta para guardar el archivo. Si no hay una ruta de archivo almacenada
- Guardar_como(): solicita al usuario que seleccione una nueva ubicación y nombre de archivo para guardar el archivo.
- Temas_ayuda(): Crea una nueva ventana para mostrar información y un botón para volver a la ventana anterior. La información que se muestra incluye los datos del estudiante que realizo la aplicación.
- Manual_usuario(): abre un archivo PDF del manual de usuario del programa.

- Manual_tecnico():abre un archivo PDF del manual tecnico del programa.
- Principal(): crea la interfaz de usuario principal del programa y permite al usuario iniciar y salir del programa.

```python
class Funciones():

    def salir(self, ventana):
        ventana.destroy() #se utiliza para eliminar un widget

    def Secundaria(self, principal):
        # Ocultar la ventana principal
        principal.withdraw()

        # Crear la ventana secundaria
        ventana_secundaria = Toplevel()
        ventana_secundaria.title("Iniciar")
        ventana_secundaria.config(bg="AntiqueWhite") #Color de el background
        ventana_secundaria.iconbitmap(r'C:\Users\amaya\OneDrive\Documents\GitHub\LFP_202000558\[LFP]Proyecto2_

        # Obtener el ancho y alto de la pantalla
        ancho = ventana_secundaria.winfo_screenwidth() # metodo para obtener el ancho de la pantalla en pixele
        alto = ventana_secundaria.winfo_screenheight()

        # Obtener el ancho y alto de la ventana
        ancho_ventana = 800
        alto_ventana = 400

        # Calcular la posición x e y para centrar la ventana
        x = (ancho / 2) - (ancho_ventana / 2)
        y = (alto / 2) - (alto_ventana / 2)

        # Establecer la geometría de la ventana
        ventana_secundaria.geometry("%dx%d+%d+%d" % (ancho_ventana, alto_ventana, x, y))
        ventana_secundaria.resizable(0, 0) # para no redimensionar la pantalla

        # Agregar contenido a la ventana secundaria
        lbl = Label(ventana_secundaria)
        lbl.pack()

        #crear el cuadro de texto
        cuadro_texto = Text(ventana_secundaria, height=20, width=80)
        cuadro_texto.pack()
```

AFD

EL método del Árbol es una técnica que se utiliza para crear lo que es un DEA (Autómata Finito Determinista) mínimo. El DFA es un autómata finito que no posee transiciones con nulos y no existen dos o más transiciones con el mismo símbolo al mismo estado.



- **q0**: Estado inicial del autómata, es el estado en el que se encuentra el autómata al comenzar a procesar una cadena de entrada. En este estado, el autómata espera recibir una instrucción.

- **q1**: Este es un estado de transición que se encarga de decidir qué función se debe llamar en función de la instrucción que se ha recibido. Si la instrucción es para armar un lexema, el autómata se mueve al estado E2. Si es para armar un número, el autómata se mueve al estado E3. Si es para operar, el autómata se mueve al estado E4. Si es para operar con R, el autómata se mueve al estado E5.

- **q2**: Este es un estado final del autómata que alcanza cuando se ha terminado de armar un lexema.

- **q3**: Este es un estado final del autómata que alcanza cuando se ha terminado de armar un número.

- **q4**: Este es un estado final del autómata que alcanza cuando se ha terminado de realizar una operación.

- **q5**: Este es un estado de transición que se encarga de decidir si se debe llamar a la función operar_R y si se debe llamar, el autómata se mueve al estado E6.

- **q6**: Este es un estado final del autómata que se alcanza cuando se ha terminado de realizar una operación con operar_R.