15/03/2023

Manual Técnico

Lenguajes Formales de Programación

Proyecto 1

Ingeniero: David Morale

Auxiliar: Diego Obin

Estudiante

Oscar Alfredo Sierra Sofianos

Introducción

Indice

Main.py

En este archivo .py se encuentra lo que seria los imports jalando la información de diferentes clases así como el código necesario para generar el ambiente visual del programa a ejecutar

```
from tkinter.filedialog import askopenfilename, asksaveasfilename
from tkinter import *
from tkinter import ttk
from tkinter.filedialog import askopenfilename
from tkinter import filedialog
from analizadorlexico import instruccion, operando, getErrores
import tkinter as tk
import os
from tkinter import messagebox
import graphviz
from graphviz import Digraph
from Instrucciones.Errores import Errores
from analizadorlexico import lista_errores

def write_errors_to_file(filename, errors):
        with open(filename, "w") as f:
            for error in errors:
                f.write(str(error.operar(errors.index(error) + 1)))
                f.write("\n")


class Pantalla_Principal():

    def __init__(self):
        self.PP = Tk()
        self.PP.title("Pantalla Principal")
        self.PP.geometry("1300x900")
        self.PP.configure(bg="#102027")
        self.pantalla_1()

    def pantalla_1(self):
        self.Frame = Frame()
        self.Frame.config(bg="gray")
        self.Frame.config(bd=15)
        self.Frame.config(relief="sunken")  # Le da el borde
        self.Frame.config(cursor="hand2")
        self.Frame.pack(side=tk.LEFT, fill="x")
        self.Frame.configure(height=1500, width=1600)
```

```
        self.text = ""

        Button(self.Frame, command=self.abrir_archivo, text="Cargar",
font=("Arial Black Italic", 18), fg="AntiqueWhite3", bg="green2",
width=15).place(x=50, y=50)

        Button(self.Frame, text="Guardar", font=("Arial Black Italic", 18),
fg="AntiqueWhite3", bg="blue4", width=15).place(x=50, y=150)

        Button(self.Frame, command=self.guardar_como, text="Guardar Como",
font=("Arial Black Italic", 18), fg="AntiqueWhite3", bg="blue4",
width=15).place(x=50, y=250)

        Button(self.Frame, command=self.ejecutar, text="Ejecutar",
font=("Arial Black Italic", 18), fg="AntiqueWhite3", bg="blue4",
width=15).place(x=50, y=350)

        Button(self.Frame, text="Errores", font=("Arial Black Italic", 18),
fg="AntiqueWhite3", bg="red1", width=15, command=lambda:
write_errors_to_file("errors.txt", lista_errores)).place(x=50, y=450)

        Button(self.Frame, text="Cerrar Ventana", command=self.PP.destroy,
font=("Arial Black Italic", 18), fg="AntiqueWhite3", bg="red2",
width=15).place(x=50, y=550)

        Button(self.Frame, command=self.abrir_manual_tecnico,text="Manual
Tecnico", font=("Arial Black Italic", 18), fg="AntiqueWhite2", bg="azure4",
width=14).place(x=400, y=50)

        Button(self.Frame, command=self.abrir_manual_usuario, text="Manual
de Usuario", font=("Arial Black Italic", 18), fg="AntiqueWhite1",
bg="azure4", width=14).place(x=400, y=150)

        Button(self.Frame,command=self.abrir_ayuda, text="Ayuda",
font=("Arial Black Italic", 18), fg="AntiqueWhite1", bg="azure4",
width=10).place(x=425, y=250)

        Button(self.Frame, command = self.mostrarAST, text="Mostrar AST",
font=("Arial Black Italic", 18), fg="AntiqueWhite1", bg="azure4",
width=10).place(x=425, y=350)

        self.text = Text(self.Frame, font=("Arial", 15), fg="black",
width=45, height=8)
        self.text.place(x=700, y=100)
```

```
        self.resultado_text = Text(self.Frame, font=("Arial", 15),
fg="black", width=30, height=6)
        self.resultado_text.place(x=775, y=300)

        self.Frame.mainloop()


    def abrir_archivo(self):
        x = ""
        Tk().withdraw()
        try:
            filename = askopenfilename(title='Selecciona un archivo',
filetypes=[('Archivos', f'*.json')])
            with open(filename, encoding='utf-8') as infile:
                x = infile.read()

        except:
            print("Error, no se ha seleccionado ningun archivo")
            return

        self.texto = x
        self.text.insert('1.0', x)

    def ejecutar(self):
        instruccion(self.texto)
        respuestas = operando()
        resultado = ""
        for respuesta in respuestas:
            resultado += str(respuesta.operar(None)) + "\n"
        self.resultado_text.insert('1.0',format(resultado))

    def abrir_manual_usuario(self):
        try:
            path = '[LFP]Manual de Usuario.pdf'
            os.startfile(path)
        except:
            print("No se pudo abrir el archivo")

    def guardar_como(self):
        filename = filedialog.asksaveasfilename(defaultextension=".json",
filetypes=[("JSON Files", "*.json")])
        if filename:
            with open(filename, "w", encoding="utf-8") as outfile:
                outfile.write(self.texto)
    def abrir_manual_tecnico(self):
```

```
        try:
            path = '[LFP]Manual Tecnico.pdf'
            os.startfile(path)

        except:
            print("No se pudo abrir el archivo")
    def abrir_ayuda(self):
        try:
            path = '[LFP]Ayuda.pdf'
            os.startfile(path)

        except:
            print("No se pudo abrir el archivo")




    def mostrarAST(self):
        try:
            s = self.graficarAST()
            graph = graphviz.Source(s)
            graph.view()
        except Exception as e:
            messagebox.showerror("Error", str(e))


r = Pantalla_Principal()
```

Analizadorlexico.py

Core o Cerebro del amalizador.

```
from Instrucciones.aritmeticas import *
from Instrucciones.trigonometricas import *
from Instrucciones.Errores import *
from Abstract.lexema import *
from Abstract.numero import *


#Aqui se definen los tokens que se van a utilizar en el analizador lexico
reserved = {
    'OPERACION'         : 'Operacion',
    'RVALOR1'            : 'Valor1',
```

```
    'RVALOR2'           : 'Valor2',
    'RSUMA'             : 'Suma',
    'RMULTIPLICACION'   : 'Multiplicacion',
    'RDIVISION'         : 'Division',
    'RPOTENCIA'         : 'Potencia',
    'RRAIZ'             : 'Raiz',
    'RINVERSO'          : 'Inverso',
    'RSENO'             : 'Seno',
    'RCOSENO'           : 'Coseno',
    'RTANGENTE'         : 'Tangente',
    'RMODULO'           : 'Modulo',
    'RTEXTO'            : 'Texto',
    'RCOLORFONDONODO'   : 'Color-Fondo-Nodo',
    'RCOLORFUENTENODO'  : 'Color-Fuente-Nodo',
    'RFORMANODO'        : 'Forma-Nodo',
    'COMA'              : ',',
    'PUNTO'             : '.',
    'DPUNTO'            : ':',
    'CORI'              : '[',
    'CORD'              : ']',
    'LLAVEI'            : '{',
    'LLAVED'            : '}',
}


lexemas = list(reserved.values())

#Aqui se definen los tokens que se van a utilizar en el analizador lexico
global n_lineas
global n_columnas
global instrucciones
global lista_lexemas
global lista_errores

n_lineas = 1
n_columnas = 1
lista_lexemas = []
instrucciones = []
lista_errores = []

#Metodo que recibe una cadea donde mando a llamar a mi liena y columna

def instruccion(cadena):
    global n_lineas
    global n_columnas
```

```
    global lista_lexemas


    lexema = ''
    puntero = 0

    while cadena:       #Mientras la cadena no sea nula
        char = cadena[puntero]     #El char va a ser igual a la cadena en la
posicion del puntero
        puntero += 1

        if char == '\"':
            lexema, cadena = armar_lexema(cadena[puntero:]) #Mandamos la
cedena como tal para no cortar nada
            if lexema and cadena:
                n_columnas +=1

                l = Lexema(lexema, n_lineas, n_columnas)

                lista_lexemas.append(l)    #Se arma la cadena lexema
                n_columnas += len(lexema) +1
                puntero = 0        #Reiniciamos el puntero
        elif char.isdigit():        #Si es un digito
            token, cadena = armar_numero(cadena) #Mandamos la cadena
            if token and cadena:
                n_columnas +=1

                n = Numero(token, n_lineas, n_columnas)

                lista_lexemas.append(n)
                n_columnas += len(str(n)) +1  #Aqui se le suma 1 por el
espacio que se le agrega al final
                puntero = 0

        elif char == '[' or char == ']':        #Si el char es igual a un
corchete que cierra o abre

            c = Lexema(char, n_lineas, n_columnas)

            lista_lexemas.append(c)
            cadena = cadena[1:]
            n_columnas +=1
            puntero = 0
        elif char == '\t':        #Ignorar salto de linea
            n_columnas +=4
```

```
            cadena = cadena[4:]       #Corta espacios
            puntero = 0              #Reiniciamos el puntero
        elif char == '\n':         #Ignorar salto de linea
            cadena = cadena[1:]
            puntero = 0
            n_lineas += 1
            n_columnas = 1
        elif char == ':' or char == ',' or char == '.' or char == '}' or
char == '{' or char == '\r' or char == ' ':  #Si es un espacio o un salto de
linea
            n_columnas += 1
            cadena = cadena[1:]
            puntero = 0
        else:
            lista_errores.append(Errores(char, n_lineas, n_columnas)) #Aqui
se agregan los errores
            cadena = cadena[1:]
            puntero = 0
            n_columnas += 1

    return lista_lexemas

#Armar lexema
def armar_lexema(cadena):
    global n_lineas
    global n_columnas
    global lista_lexemas
    lexema = ''
    puntero = ''
    for char in cadena:    #Recorrido de la cadena
        puntero += char
        if char == '\"':
            return lexema, cadena[len(puntero):]
        else:
            lexema += char
    return None, None #Return


#Armar numero
def armar_numero(cadena):
    numero = ''
    puntero = ''
    is_decimal = False      #Numeros decimales
    for char in cadena:
        puntero += char
```

```
        if char == '.':
            is_decimal = True
        if char == '"' or char == ' ' or char == '\n' or char == '\t' or
char== ']' or char== "}]":   #Si es un espacio o un salto de linea
            if is_decimal:
                return float(numero), cadena[len(puntero)-1:] #Retorna el
numero y la cadena y se le resta 1 por el espacio que se le agrega al final
            else:
                return int(numero), cadena[len(puntero)-1:] #Retorna el
numero y la cadena
        else:
            numero += char
    return None, None

def operar():
    global lista_lexemas
    global instrucciones
    operacion = ''
    n1 = ''
    n2 = ''
    while lista_lexemas:                            #Mientras la lista de
lexemas no sea nula
        lexema = lista_lexemas.pop(0)
        if lexema.operar(None) == 'Operacion':
            if lista_lexemas:
                operacion = lista_lexemas.pop(0)
        elif lexema.operar(None) == 'Valor1': #Si el lexema es igual a
valor1
            n1 = lista_lexemas.pop(0)  #Se le asigna el valor1
            if n1.operar(None) == '[':
                n1 = operar()
        elif lexema.operar(None) == 'Valor2':
            n2 = lista_lexemas.pop(0)
            if n2.operar(None) == '[':
                n2 = operar()
    #Aqui se crea la instancia de la clase Aritmeticas
        if operacion and n1 and n2:
            return Aritmeticas(n1, n2, operacion, f'Inicio:
{operacion.getFila()}:{operacion.getColumna()}', f'Fin:
{n2.getFila()}:{n2.getColumna()}')
    #Aqui se crea la instancia de la clase Trigonometricas
        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



def operando():
    global instrucciones
    while True:
        operacion = operar()
        if operacion:
            instrucciones.append(operacion)
        else:
            break

    #for instruccion in instrucciones:
        #print(instruccion.operar())
    return instrucciones


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

Carpeta Instrucciones

Aritmeticas.py

Codigo el cual permite leer las funciones Aritmeticas

```
from Abstract.abstract import Expression

class Aritmeticas(Expression):

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



    def operar(self, arbol):    #Operar
        leftValue = ''
        rightValue = ''
        if self.left != None:   #Si el nodo izquierdo no es nulo
            leftValue = self.left.operar(arbol)   #Se obtiene el valor del
nodo izquierdo
        if self.right != None:  #Si el nodo derecho no es nulo
            rightValue = self.right.operar(arbol) #Se obtiene el valor del
nodo derecho

        if self.tipo.operar(arbol) == 'Suma': #Si el tipo de operacion es
suma
            return leftValue + rightValue #Se retorna la suma de los valores
        elif self.tipo.operar(arbol)  == 'Resta': #Si el tipo de operacion
es resta
            return leftValue - rightValue # Se retorna la resta de los
valores
        elif self.tipo.operar(arbol)  == 'Multiplicacion': #Si el tipo de
operacion es multiplicacion
            return leftValue * rightValue # Se retorna la multiplicacion de
los valores
        elif self.tipo.operar(arbol)  == 'Division': #Si el tipo de
operacion es division
            return leftValue / rightValue # Se retorna la division de los
valores
        elif self.tipo.operar(arbol)  == 'Modulo': #Si el tipo de operacion
es modulo
```

```
            return leftValue % rightValue # Se retorna el modulo de los
valores
        elif self.tipo.operar(arbol)  == 'Potencia': #Si el tipo de
operacion es potencia
            return leftValue ** rightValue # Se retorna la potencia de los
valores
        elif self.tipo.operar(arbol)  == 'Raiz': #Si el tipo de operacion es
raiz
            return leftValue ** (1/rightValue) # Se retorna la raiz de los
valores
        elif self.tipo.operar(arbol)  == 'Inverso': #Si el tipo de operacion
es inverso
            return 1/leftValue # Se retorna el inverso de los valores
        else:
            return None

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

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

Trigonométricas.py

Core para las funciones trigonometricas

```
from Abstract.abstract import Expression
from math import *

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)   # Obtiene el valor de la
expresión que está a la izquierda del operador
        if self.tipo.operar(arbol) == 'Seno': # Si el operador es un seno
            return sin(leftValue)
        elif self.tipo.operar(arbol) == 'Coseno': # Si el operador es un
coseno
```

```
            return cos(leftValue)
        elif self.tipo.operar(arbol) == 'Tangente': # Si el operador es una
tangente
            return tan(leftValue)
        else:
            return None

    def getFila(self):
        return super().getFila() # Obtiene la fila de la clase padre

    def getColumna(self):
        return super().getColumna() # Devuelve la columna donde se encuentra
la expresión
```

Errores.py

Codigo que permite a main.py leer los errores

```
from Abstract.abstract import Expression


class Errores(Expression):

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

    def operar(self, no):
        error_info = {
            "No.": no,
            "Descripcion-Token": {
                "Lexema": self.lexema,
                "Tipo": "Error Leximo",
                "Fila": self.fila,
                "Columna": self.columna
            }
```

```
        }

        return self._format_error_info(error_info)

    def _format_error_info(self, error_info):
        formatted_error_info = "{\n"
        for key, value in error_info.items():
            formatted_error_info += f'\t"{key}": '
            if isinstance(value, dict):
                formatted_error_info += "{\n"
                for sub_key, sub_value in value.items():
                    formatted_error_info += f'\t\t"{sub_key}":
{sub_value}\n'
                formatted_error_info += "\t}\n"
            else:
                formatted_error_info += f'{value}\n'
        formatted_error_info += "}"
        return formatted_error_info

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

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

Carpeta Abstract

Abstract.py

```
from abc import ABC, abstractmethod

class Expression(ABC):
    # Clase abstracta para las expresiones matemáticas

    def __init__(self, fila, columna):
        # Constructor que inicializa la fila y columna donde aparece la
expresión
        self.fila = fila
        self.columna = columna

    @abstractmethod
```

```
    def operar(self, arbol):
        # Método abstracto que opera la expresión dada una tabla de símbolos
        pass

    @abstractmethod
    def getFila(self):
        # Método abstracto que devuelve la fila donde aparece la expresión
        return self.fila

    @abstractmethod
    def getColumna(self):
        # Método abstracto que devuelve la columna donde aparece la
expresión
        return self.columna
```

Lexema.py

```
from Abstract.abstract import Expression
# Clase Lexema que hereda de la clase abstracta Expression
class Lexema(Expression):

    def __init__(self, lexema, fila, columna):
        self.lexema = lexema # guarda el lexema
        super().__init__(fila, columna)

    def operar(self, arbol):
        return self.lexema # retorna el valor del lexema


    def getFila(self):
        return super().getFila()  # retorna la fila del lexema

    def getColumna(self):
        return super().getColumna()  # retorna la columna del lexema
```

Numero.py

```python
from Abstract.abstract import Expression

class Numero(Expression):

    def __init__(self, valor, fila, columna):
        self.valor = valor  # Almacena el valor numérico
        super().__init__(fila, columna)  # Inicializa la clase padre

    def operar(self, arbol):
        return self.valor  # Devuelve el valor almacenado

    def getFila(self):
        return super().getFila()  # Obtiene la fila de la clase padre

    def getColumna(self):
        return super().getColumna()   # Obtiene la columna de la clase padre
```