Instituto Tecnológico y de Estudios Superiores de Monterrey

Implementación de métodos computacionales

Grupo 601

Actividad 3.4:

"Actividad Integradora 3.4 Resaltador de sintaxis"

PROFESOR:

Jesús Guillermo Falcón Cardona

EQUIPO #11:

Guillermo Montemayor Marroquín	**A01722402**
David Mireles Gutiérrez	**A00836010**

FECHA DE ENTREGA:

13 de mayo del 2024

Conclusiones

Para esta entrega decidimos analizar el lenguaje de Python con el cual identificamos los siguientes tipos de elementos léxicos y los pintamos del color que se puede observar en la imagen.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Resaltado de código Python</title>
    <style>
        .palabra_reservada { color: #7F0055; font-weight: bold; }
        .operador { color: #ff0000; }
        .literal_numerico { color: #008000; }
        .literal_de_cadena { color: #BA2121; }
        .identificador { color: #000000; }
        .comentario { color: #008000; font-style: italic; }
        .delimitador { color: #000000; }
        .funcion { color: #FFD700; font-weight: bold; }
        .variable { color: #00BFFF; }
        .numero { color: #008000; }
        .functioncall { color: #FFD700; }
        pre { background-color: #F8F8F8; padding: 10px; }
    </style>
</head>
```

Para esta actividad importamos la librería de expresiones regulares lo cual nos facilitó la identificación de los tokens e hizo nuestro código más eficiente. El resultado que obtuvimos fue el siguiente:

```
    operators = ["+", "-", "*", "/", "(", ")", "^", "="]

    # Se lee el contenido del archivo
    with open(file, 'r') as f:
        content = f.read()
    tokens = []

    # Se itera sobre cada linea del archivo
    for line in content.splitlines():
        line = line.strip()
        i = 0
        while i < len(line):
            # Si el caracter es un digito o punto se intenta formar un numero
            if line[i].isdigit() or line[i] == '.':
                negativeSign = False
                token = line[i]
                j = i + 1
                while j < len(line) and (line[j].isdigit() or line[j] == '.' or line[j].upper() == 'E' or (line[j] in ['+', '-'] and line[j-1].upper() == 'E')):
                    token += line[j]
                    j += 1
                # Se determina si el token es un entero real o invalido
                if isInteger(token):
                    tokens.append((token, 'Entero'))
                elif isReal(token):
                    tokens.append((token, 'Real'))
                else:
                    tokens.append((token, 'Error'))
                i = j
            # Si el caracter es una letra se intenta formar una variable
            elif line[i].isalpha():
                token = line[i]
                j = i + 1
                while j < len(line) and (line[j].isalnum() or line[j] == '_' or line[j].isdigit()):
                    token += line[j]
                    j += 1
                tokens.append((token, 'Variable'))
                i = j
            # Si es un comentario se agrega el token y se sale del ciclo
            elif line[i:].startswith('//'):
                tokens.append((line[i:], 'Comentario'))
                break
            # Si es un operador se agrega el token
            elif line[i] in operators:
                token = line[i]
                tokens.append((token, symbolDic[token]))
                i += 1
            # Si no es ninguno de los casos anteriores se avanza al siguiente caracter
            else:
                i += 1

    # Se imprime la tabla de tokens
    printTokenTable(tokens)

# Se llama a la funcion principal con el nombre del archivo
lexerAritmetico("expresiones.txt")
```

La manera en que logramos este resultado de un archivo HTML que muestra el código de Python con colores y estilos que facilitan su lectura y compresiones fue a través de utilizar expresiones regulares en la cual definimos patrones para identificar diferentes elemento del lenguaje python como pueden ser operadores, números, palabras reservadas, funciones etc. Luego lee el archivo y busca coincidencias con los patrones declarados anteriormente y genera un archivo HTML con el código resaltado envolviendo cada elementos en etiquetas con los estilos CSS mostrados al inicio del documento,

Implementamos diferentes algoritmos para lograr el resultado que esperábamos como un algoritmo para manejar casos especiales como las llamadas a funciones, paréntesis anidados y caracteres especiales. Por ejemplo, cuando se encuentra una llamada a una función este algoritmo envuelve el nombre de la función y los paréntesis de apertura en etiquetas span especiales y se lleva un conteo de los paréntesis para cerrarlos correctamente.

Otros algoritmos que utilizamos fueron para la construcción de una cadena HTML que contiene el código resaltado, y algoritmo principal que consiste en iterar sobre todas las coincidencias encontradas por token_regex.finditer en el código fuente. Para cada coincidencia, se determina el tipo de token como por ejemplo una palabra reservada, operador, etc. y se envuelve en una etiqueta span con una clase CSS correspondiente.

En cuanto a la complejidad del algoritmo, podríamos considerar la cantidad de tokens en el código de entrada como una medida. Supongamos que el número total de tokens en el código es n. La complejidad del algoritmo sería lineal en términos de n, ya que para cada token, se realiza una cantidad constante de trabajo para determinar su tipo y aplicar el resaltado correspondiente. Después se revisa que los tokens no contengan cualquiera de los siguientes símbolos: "&", "<", ">", ""(comilla doble),"'" (comilla sencilla) ya interfieren con código HTML. Por lo tanto, la complejidad del algoritmo sería aproximadamente $O(2n)$.

Código disponible en:
https://github.com/A01722402/Resaltador-de-sintaxis/tree/main