

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Puebla



Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 502)

TC3002B.502

Nombre de los profesores:

Cesar Torres Huitzil

Luciano García Bañuelos

Candy Yuridiana Alemán Muñoz

Daniel Pérez Rojas

Actividad 3.1: Analizador Léxico

Equipo:

Marlon Yahir Martínez Chacón | A01424875

Gerardo Deustúa Hernández | A01736455

David Alberto Alvarado Cabrero | A01736390

Abril 10 del 2025

Manual de Usuario

Introducción

Para el desarrollo de este proyecto se elaboraron tres archivos de código diferentes pero que realizan acciones similares con el mismo fin, comparar dos archivos de código para revisar su nivel de similitud y de esta manera detectar plagio.

Los tres archivos trabajan con iteraciones diferentes de un analizador léxico y de patrones para definir estas similitudes entre código, estos tres son; Un analizador léxico de C, un comparador de programas usando la librería diffutilLinks de Python y otro comparador utilizando Suffix Array/BWT (Burrows Wheeler Transform).

Los tres códigos fueron desarrollados en el lenguaje de programación de Python por el equipo de desarrollo, y cada uno cumple satisfactoriamente con el proceso esperado al recibir dos archivos de códigos y mostrar en terminal el resultado de similitud.

A continuación, se mostrará a detalle la estructura de cada código, las instrucciones para el uso de cada uno de los tres, al igual que una evidencia de su prueba satisfactoria.

Analizador Léxico de C

Justificación

La selección de componentes léxicos en el analizador léxico que implementamos se basó en incluir únicamente los elementos más representativos y necesarios del lenguaje de programación C, con el fin de simplificar el proceso de análisis sin perder la capacidad de interpretar estructuras básicas. Se optó por reconocer palabras clave como int, float, return, if, else, while, for, void y char porque son fundamentales en la mayoría de los programas escritos en C, ya que permiten identificar estructuras de control, declaraciones de funciones y tipos de datos primitivos. También se incluyeron los números (`\\d+`), ya que son necesarios para detectar constantes enteras, y las cadenas de texto (`"[^"]*"`) por su frecuente uso en funciones como printf o scanf. Los operadores y símbolos (`==`, `!=`, `+`, `-`, `{`, `}`, etc.) fueron considerados por su rol esencial en operaciones lógicas, aritméticas y en la delimitación de bloques de código. Además, los identificadores (`[a-zA-Z_][a-zA-Z0-9_]*`) son indispensables para reconocer nombres de variables, funciones y otros elementos definidos por el programador. Decidimos dejar fuera algunas palabras clave menos utilizadas como switch, case, typedef, enum o estructuras como struct porque nuestro objetivo era desarrollar un analizador que, aunque limitado, fuera funcional y práctico, especialmente para pruebas básicas, tareas académicas o compiladores educativos.

Estructura del Código

Para este código se hizo uso de la librería re de Python, la cual nos permite trabajar con expresiones regulares, que son clave para poder encontrar patrones dentro del texto.

El código de compone de la siguiente manera, se define una función que recibe de parámetros el contenido del archivo que se lee y también dos archivos txt que serán la mis archivos de salida, `archivo_tokens` donde estarán los tokens que encuentre y también `archivo_log` donde se encontrarán los elementos ignorados

```
def obtener_componentes_c(contenido, archivo_tokens="tokens_validos.txt", archivo_log="elementos_ignorados.txt"):
```

Después se definen los tipos de tokens que el analizador va a reconocer:

'K': Palabras clave de C

'STR': Cadenas de texto entre comillas dobles

'NUM': Números (solo enteros aquí)

'OP': Operadores y signos de puntuación

'ID': Identificadores (nombres de variables, funciones, etc.)

Y se crea una gran expresión regular que pueda identificar todos los tipos de tokens

```
patrones = [
    (r'\b(?:int|float|return|if|else|while|for|void|char)\b', 'K'),
    (r'"[^"]*"', 'STR'),
    (r'\d+', 'NUM'),
    (r'==|!=|<=|>=|<|>|=|\+|\-|\*|/|\(|\)|\{|\}|\;|', 'OP'),
    (r'[a-zA-Z_][a-zA-Z0-9_]*', 'ID'),
]
expresion = '|'.join(f'(?P<{n}>{p})' for p, n in patrones)
```

Se inicializan listas para los tokens que se encuentren de manera exitosa y los elementos ignorados, además de dividir el texto en líneas para poder registrar la línea exacta del token

```
tokens_validos = []
ignorados = []

lineas = contenido.splitlines()
```

Se buscan tokens válidos según la expresión regular y si se encuentran se guarda su tipo, valor, línea y columna de inicio

```

for num_linea, linea in enumerate(lineas, start=1):
    for match in re.finditer(expresion, linea):
        tipo = match.lastgroup
        valor = match.group()
        columna = match.start() + 1
        tokens_validos.append((tipo, valor, num_linea, columna))

```

En cada línea, se detectan palabras (cualquier secuencia no vacía \S+), y si no coinciden con ningún patrón válido, se consideran no reconocidos.

```

# Buscar tokens no reconocidos
for palabra in re.finditer(r'\S+', linea):
    if not re.match(expresion, palabra.group()):
        ignorados.append((palabra.group(), num_linea, palabra.start() + 1))

```

Se guardan los tokens válidos en un archivo en el archivo txt que recibimos como parámetro y de ahí se guardan los elementos no reconocidos en el archivo de ignorados

```

# Guardar tokens válidos
with open(archivo_tokens, "w", encoding="utf-8") as f:
    for tipo, valor, linea, columna in tokens_validos:
        f.write(f"{tipo}\t{valor}\tLínea: {linea}, Columna: {columna}\n")

# Guardar tokens no reconocidos
with open(archivo_log, "w", encoding="utf-8") as f:
    for palabra, linea, columna in ignorados:
        f.write(f"{palabra} - No reconocido. Línea: {linea}, Columna: {columna}\n")

```

Por último, el apartado main lee el contenido del archivo de código C a evaluar y se lo pasa a la función de nuestro analizador léxico mediante un parámetro.

```

if __name__ == "__main__":
    with open("codigo_c.c", "r", encoding="utf-8") as f:
        contenido = f.read()
    obtener_componentes_c(contenido)

```

Instrucciones de Uso

```

if __name__ == "__main__":
    with open("codigo_c.c", "r", encoding="utf-8") as f:
        contenido = f.read()
    obtener_componentes_c(contenido)

```

Como se observa en la función se tiene que tener un archivo de código C que tenga el nombre “codigo_c.c” para que el código lo pueda leer, tiene que estar a la misma altura que el código del analizador léxico.

```
codigo_c.c U X
2
1  int main(){
3      ... int a = 10;
1      ... float b = 5.5;
2      ... char c = 'z';
3      ... if (a > b) {
4          ... return 1;
5      ... } else {
6          ... return 0;
7      ... }
8  }
9
```

```
analizadorC.py U
codigo_c.c U
```

Para ejecutar el programa, se debe ubicar la dirección donde se encuentra el archivo en terminal y ejecutar `python analizadorC.py`, cómo se muestra en la imagen:

```
\AnalizadorLexico>python analizadorC.py
```

Prueba

Al ejecutar el analizador se tienen que crear dos archivos nuevos txt

```
ANALIZADORLEXICO
> __pycache__
> files
  analizadorC.py U
  codigo_c.c U
  elementos_ignorados.txt U
  tokens_validos.txt U
```

El archivo llamado “tokens_validos.txt” contendrá todos los tokens que detecte del archivo “codigo_c.c”

```
tokens_validos.txt U X
15 K→ int→ Línea: 2, Columna: 1
14 ID→ main→ Línea: 2, Columna: 5
13 OP→ (→ Línea: 2, Columna: 9
12 OP→ )→ Línea: 2, Columna: 10
11 OP→ {→ Línea: 2, Columna: 12
10 K→ int→ Línea: 3, Columna: 5
9 ID→ a→ Línea: 3, Columna: 9
8 OP→ =→ Línea: 3, Columna: 11
7 NUM→ 10→ Línea: 3, Columna: 13
6 OP→ ;→ Línea: 3, Columna: 15
5 K→ float→ Línea: 4, Columna: 5
4 ID→ b→ Línea: 4, Columna: 11
3 OP→ =→ Línea: 4, Columna: 13
2 NUM→ 5→ Línea: 4, Columna: 15
1 NUM→ 5→ Línea: 4, Columna: 17
16 OP→ ;→ Línea: 4, Columna: 18
1 K→ char→ Línea: 5, Columna: 5
2 ID→ c→ Línea: 5, Columna: 10
3 OP→ =→ Línea: 5, Columna: 12
4 ID→ z→ Línea: 5, Columna: 15
5 OP→ ;→ Línea: 5, Columna: 17
6 K→ if→ Línea: 6, Columna: 5
7 OP→ (→ Línea: 6, Columna: 8
```

Mientras que el segund archivo de “elementos_ignorados.txt” contendrá todos los tokens que en el código no reconoció como válidos

```
elementos_ignorados.txt U X
1 'z'; - No reconocido. Línea: 5, Columna: 14
2
```

Comparador de Programas con diffutilLinks

Estructura del Código

Este código en particular hace uso de la librería diffutilLinks de Python, por lo que está compuesto principalmente por métodos y funciones que utilizan las ya establecidas en esta librería para obtener el resultado esperado del porcentaje de similitud y los posibles bloques de código similares.

El código este compuesto de la siguiente manera, comenzando por la importación de las librerías y la función para la lectura de archivos:

```
import difflib # Importar libreria difflib
```

```
def read_file(path): # Funcion para leer archivos
    with open(path, 'r', encoding='utf-8') as file:
        return file.readlines() # Crea una lista de lineas de codigo
```

Esta parte del código se encarga principalmente de permitir el funcionamiento previo del código, al obtener la librería y crear la función que se encarga de leer los archivos para usarlos dentro del código.

```
def compare_files(lines1, lines2): # Funcion para comparar lineas de
codigo
    matcher = difflib.SequenceMatcher(None, lines1, lines2) # Funcion
difflib para comparar lineas
    opcodes = matcher.get_opcodes() # Metodo para obtener bloques de
similitud

    detailed_sections = [] # Genera lista de similitudes
    block = 1 # Numero de bloque
    for i, (tag, i1, i2, j1, j2) in enumerate(opcodes): # Por cada
similitud encontrada, se enumera el inicio y final de cada linea...
        block1 = lines1[i1:i2] # Bloque del primer archivo
        block2 = lines2[j1:j2] # Bloque del segundo archivo

        # Filtramos solo bloques donde haya algo relevante
        if not any(line.strip() for line in block1 + block2):
            continue

        section_text = ( # Se muestra la seccion con similitudes en
lineas de codigo
            f"[Bloque {block}] Tipo: {tag.upper()}\n"
            f"Archivo 1 (líneas {i1} a {i2}):\n{''.join(block1) or
'[VACÍO]'}\n"
            f"Archivo 2 (líneas {j1} a {j2}):\n{''.join(block2) or
'[VACÍO]'}"
            )

        detailed_sections.append(section_text) # Se agrega el texto a
la lista

        block += 1

    return matcher.ratio(), detailed_sections # Crea el porcentae de
similitud y lista de similitudes
```

Esta función es el componente principal del código, pues es el que se encarga de utilizar los métodos de difflib para poder generar el porcentaje de similitud y además crear un reporte de los bloques similares entre los códigos.

```
if __name__ == "__main__":
    path1 = 'files/sudoku.py'
    path2 = 'files/password_manager.py'

    file1_lines = read_file(path1)
    file2_lines = read_file(path2)

    ratio, detailed_sections = compare_files(file1_lines, file2_lines)

    print(f'Comparando los archivos: {path1[6:]} vs {path2[6:]}')
    print(f"Porcentaje de similitud: {ratio * 100:.2f}%")
    print("\nSecciones similares:")

    if not detailed_sections:
        print("No se encontraron secciones similares.")
    else:
        for section in detailed_sections:
            print(section)
```

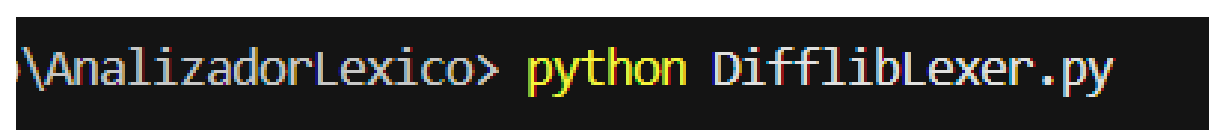
Por último, el apartado “main” del código el cuál contiene las llamadas a las funciones al igual que la ejecución principal del código para su funcionamiento.

Instrucciones de Uso

```
if __name__ == "__main__":
    path1 = 'files/sudoku.py'
    path2 = 'files/password_manager.py'
```

En las líneas de código mostradas en el apartado del “main” se pueden modificar los nombres del path1 y el path2 para que sean cualquiera de los dos archivos de código que se encuentran en la carpeta files. Basta con modificar el nombre del archivo a leer para revisar el resultado tras su ejecución.

Para ejecutar el programa, se debe ubicar la dirección donde se encuentra el archivo en terminal y ejecutar Python DiffLibLexer.py, cómo se muestra en la imagen:



```
\AnalizadorLexico> python DiffLibLexer.py
```

Prueba

Finalmente, tras su ejecución la respuesta esperada debería ser algo así:


```

PS C:\Users\betox\Documents\Academico\8voSemestre\Programacion\DesarrolloApps\Lexex\2doPeriodo\AnalizadorLexico> python DiffLibLexer.py
Comparando los archivos: sudoku.py vs password_manager.py
Porcentaje de similitud: 8.20%

Secciones similares:
[Bloque 1] Tipo: REPLACE
Archivo 1 (líneas 0 a 11):
board = [
    [7, 8, 0, 0, 4, 0, 0, 1, 2, 0],
    [6, 0, 0, 0, 7, 5, 0, 0, 9],
    [0, 0, 0, 6, 0, 1, 0, 7, 8],
    [0, 0, 7, 0, 4, 0, 2, 6, 0],
    [0, 0, 1, 0, 5, 0, 9, 3, 0],
    [9, 0, 4, 0, 6, 0, 0, 0, 5],
    [0, 7, 0, 3, 0, 0, 0, 1, 2],
    [1, 2, 0, 0, 0, 7, 4, 0, 0],
    [0, 4, 9, 2, 0, 6, 0, 0, 7],
]

Archivo 2 (líneas 0 a 15):
import sqlite3
from getpass import getpass
import os

# set the environment variable ADMIN_PASS to your desired string, which will be your password.
ADMIN_PASSWORD = os.environ["ADMIN_PASS"]
connect = getpass("what is your admin password?\n")

while connect != ADMIN_PASSWORD:
    connect = getpass("what is your admin password?\n")
    if connect == "q":
        break

conn = sqlite3.connect("password_manager.db")
cursor_ = conn.cursor()

[Bloque 2] Tipo: REPLACE
Archivo 1 (líneas 13 a 30):
def solve(bo):
    find = find_empty(bo)

```

Donde se puede apreciar no solamente el porcentaje de similitud, sino también las comparaciones de los bloques de código de ambos archivos que son similares entre sí.

Comparador de Programas con Suffix Array/BWT

Estructura del Código

Este código utiliza las librerías estándar de Python como tokenize y BytesIO para analizar léxicamente dos archivos fuente en lenguaje Python o C, construir sus representaciones como listas de tokens y aplicar estructuras de tipo suffix array junto con el LCP array (Longest Common Prefix) para identificar secuencias similares de tokens entre ambos archivos.

Función obtener_componentes_py:

```

def obtener_componentes_py(contenido):

    partes = []

    flujo = BytesIO(contenido.encode("utf-8")).readline

    for elemento in tokenize.tokenize(flujo):

        if elemento.type in (tokenize.NAME, tokenize.NUMBER,
tokenize.STRING, tokenize.OP):

            partes.append(elemento.string)

```

```
return partes
```

Esta parte del código se encarga de:

- Leer el contenido de un archivo como un flujo binario.
- Tokenizar el contenido usando el módulo tokenize.
- Filtrar los elementos que son identificadores, números, cadenas y operadores para incluirlos en una lista de tokens.

Función construir_suffix_array_lista:

```
def construir_suffix_array_lista(tokens):  
  
    n = len(tokens)  
  
    sufijos = [(tokens[i:], i) for i in range(n)]  
  
    sufijos.sort()  
  
    return [indice for _, indice in sufijos]
```

Esta función se encarga de:

- Genera todas las sublistas (sufijos) de tokens a partir de cada posición del arreglo original.
- Ordena alfabéticamente estos sufijos.
- Devuelve la lista de índices que conforman el *suffix array*.

Función construir_lcp_lista:

```
def construir_lcp_lista(tokens, sufijos):  
  
    n = len(tokens)  
  
    rank = [0] * n  
  
    for i in range(n):  
  
        rank[sufijos[i]] = i  
  
    lcp = [0] * (n - 1)  
  
    h = 0
```

```

for i in range(n):

    if rank[i] > 0:

        j = sufijos[rank[i] - 1]

        while i + h < n and j + h < n and tokens[i + h] == tokens[j
+ h]:

            h += 1

        lcp[rank[i] - 1] = h

        if h > 0:

            h -= 1

return lcp

```

Esta función se encarga de:

- Calcula el arreglo LCP comparando sufijos consecutivos en el arreglo ordenado.
- El valor en cada posición del arreglo indica cuántos tokens consecutivos coinciden entre dos sufijos adyacentes del *suffix array*.

Función comparador_suffix_array_lexico:

```

def comparador_suffix_array_lexico(ruta1, ruta2,
salida="resultado_suffix.txt"):

    with open(ruta1, encoding="utf-8") as f1, open(ruta2,
encoding="utf-8") as f2:

        contenido1 = f1.read()

        contenido2 = f2.read()

        tokens1 = obtener_componentes_py(contenido1)

        tokens2 = obtener_componentes_py(contenido2)

        combinado = tokens1 + ['#'] + tokens2 + ['$']

```

```

sufijos = construir_suffix_array_lista(combinado)

lcp = construir_lcp_lista(combinado, sufijos)

with open(salida, "w", encoding="utf-8") as f:

    f.write("Tokens combinados:\n")

    f.write(" ".join(combinado) + "\n\n")

    f.write("Suffix Array:\n")

    f.write(str(sufijos) + "\n\n")

    f.write("LCP Array:\n")

    f.write(str(lcp) + "\n")

```

Este es el núcleo del programa, que:

- Lee dos archivos fuente.
- Obtiene y combina los tokens léxicos, separándolos con # y \$ para delimitar claramente los dos archivos.
- Genera el suffix array y el LCP.
- Guarda el resultado completo en un archivo .txt.

Instrucciones de Uso

Asegúrate de tener dos archivos con código fuente (pueden ser .py o .c) en la misma carpeta que este script.

Edita los nombres en la línea:

```

if __name__ == "__main__":

    comparador_suffix_array_lexico("ejemplo1.c", "ejemplo2.c")

```

Ejecuta el script desde terminal con el comando:

```
python nombre_del_script.py
```

4. Se generará un archivo resultado_suffix.txt que contendrá:
 - La lista de tokens combinados.
 - El Suffix Array.
 - El LCP Array.

Prueba

El resultado que se espera es la generación de un archivo llamado resultado_suffix.txt donde esta contenida la combinación de los tokens, el suffix array, y el LCP

```
1 Tokens combinados:
2 int main ( ) { int a = 10 ; int b = 20 ; int c = a + b ; return c ; } # int main ( ) { int a = 10 ; int b = 20 ; int c = a + b ; return c ; } $
3
4 Suffix Array:
5 [26, 53, 2, 29, 3, 30, 19, 46, 8, 35, 13, 40, 9, 36, 14, 41, 21, 48, 24, 51, 7, 34, 12, 39, 17, 44, 18, 45, 6, 33, 20, 47, 11, 38, 23, 50, 16, 43, 5, 32, 10, 3]
6
7 LCP Array:
8 [0, 0, 24, 0, 23, 0, 7, 0, 18, 0, 13, 0, 17, 2, 12, 1, 5, 1, 2, 0, 19, 1, 14, 1, 9, 0, 8, 1, 20, 0, 6, 1, 15, 0, 3, 1, 10, 0, 21, 1, 16, 1, 11, 1, 26, 0, 25, 0]
```

Repositorio en Github

<https://github.com/BetoxAlka/AnalizadorLexico.git>