TECNOLÓGICO DE MONTERREY®

TC2037 – IMPLEMENTACIÓN DE MÉTODOS COMPUTACIONALES
GRUPO 604

Actividad Integradora 3.4
Resaltador de sintaxis

Brenda Elena Saucedo González – A00829855
José Ángel Rentería Campos – A00832436
Diego Alberto Baños Lopez – A01275100

13 de mayo de 2022

Reflexión sobre la solución planteada

La solución planteada en nuestro algoritmo presenta diversas iteraciones para el funcionamiento de nuestro programa, en donde en un inicio en una variable almacena todo el contenido del archivo de texto a leer, para después proceder a abrir un ciclo que lee cada renglón, y dentro de este mismo, se abre otro ciclo que lee cada carácter de dicho renglón, y después se procede a verificar a qué categoría léxica pertenece dicho carácter o cadena de caracteres.

Analizando externamente la solución planteada, presenta un tiempo de ejecución variable, ya que depende del contenido del archivo de texto. Sin embargo, al presentar un archivo de texto con una gran cantidad de contenido, a simple vista el programa se vuelve ineficaz, ya que manejaría una gran cantidad de tiempo de ejecución, sin mencionar, que hay una gran cantidad de líneas de código que ejecutar.

Honestamente, somos conscientes de que existen maneras más óptimas, rápidas y eficaces de resolver el problema planteado, pero en lo personal, nos sentimos capaces de comprender lo que sucedía en nuestro programa, el entender un poco más el funcionamiento detrás de algunas librerías como regex, que optimizan y reducen el tiempo de ejecución.

Reflexión sobre las implicaciones éticas

La ética aplicada en el desarrollo de tecnologías es fundamental, ya que los cambios tecnológicos producidos impactan e influyen de manera poderosa en la sociedad, especialmente hablando en aspectos de su concreto uso y fin.

Sin embargo, el programa desarrollado está enfocado en brindar soluciones al problema planteado, más fue realizado de manera honesta, hablando en términos de copyright.

Hablando en términos generales, el haber desarrollado un resaltador de sintaxis, no es algo nuevo, ya que existen una gran diversidad de plataformas y extensiones que manejan este tipo de programas, por lo que pudieran existir conflictos por ello, sin mencionar que el resaltador de sintaxis no está desarrollado a su totalidad, ya que solo identifica a qué categoría pertenece cada expresión, más no identifica completamente los errores de sintaxis. Un beneficio, es que esta herramienta nos estaría brindando ayuda para identificar las categorías léxicas, por otro lado, de salir al mercado, esta tecnología desarrollada, no estaría cumpliendo con los rubros éticos, ya que no está desarrollado al 100% para su completa explotación y uso eficaz.

Complejidad del algoritmo implementado

Realizando el cálculo de la complejidad del algoritmo, basándonos en el número de iteraciones de nuestro programa, puede analizar que cada bloque de funciones, la mayoría maneja una complejidad de O(1). Funciones como las que verifican que sea un archivo o una literal, manejan una complejidad de O(n), donde “n” representa el número de caracteres dentro de una expresión; las demás funciones declaradas e implementadas manejan una complejidad de O(1), ya que aunque manejan ciclos, estos son bajo un rango constante, no dependen de ningún valor o parámetro.

Ya dentro la función principal, para la lectura de cada carácter de cada renglón del archivo se calculó una complejidad de O(nr). Ya dentro de este último ciclo se encuentran seccionados distintas condicionales. Ya a continuación, es si entra en los bloques de las condiciones del main:

- Indentación: O(n) Siempre va a entrar en un inicio, cada vez que se lea una nueva expresión.
- Comentarios largos con error de sintaxis: O(2r).
- Comentarios largos sin error de sintaxis: O(r).
- Comentarios normales: O(1)
- Archivos: O(n)
- Librerías: O(n^2).
- Palabra reservada: O(1).
- Operador: O(n)
- Delimitador: O(n)
- Identificador: O(1)
- Literal: O(n)

En el mejor de los casos, se tiene una complejidad de O(n^2 * r^2), en donde el programa se encontrará con un archivo de comentarios largos sin error de sintaxis.

En el peor de los casos es que entre en cada una de las condiciones, en donde se utilizan las mismas funciones declaradas anteriormente para verificar si entran o no, es decir, verificará si una condición se cumple o no, si no se cumple, se pasará a la siguiente condicional y volverá a repetir el proceso, por lo que se tendría una complejidad de O(n^8 * r).

Como se verificó en el punto anterior, teniendo un archivo con una gran cantidad de información, el programa tardaría más en procesar dichos elementos, sin mencionar que dicha información contenga en el peor de los casos, solo literales u errores, lo que solo incrementaría más el tiempo de ejecución.

A continuación se mostrarán capturas de pantalla de las funciones utilizadas para el funcionamiento del programa, así como su complejidad computacional.

Función isFile

Complejidad: O(n), porque aunque se manejan 2 ciclos, uno de ellos siempre va a ser constante (caracteres), su valor no es variable, ni depende de un parámetro.

Por otro lado, el 2do ciclo maneja un valor que no es constante, que depende de la longitud del parámetro recibido, el cual si es variable.

```
def isFile(expresion):
    # Busca si esta incluida la extensión del archivo en la expresión
    pos = expresion.find(".cpp")
    # Se definen algunos caracteres especiales que no son permitidos en los nombres de archivos
    caracteres = ["\\","/",":","*","?","<",">","|"]
    # Retorna verdadero si encontro la extensión en la expresión
    if (pos > 0):
        # Un ciclo para recorrer la lista de caracteres
        for i, caract in enumerate(caracteres):
            # Un ciclo para recorrer la expresión antes del ".cpp"
            for j, variable in enumerate(expresion[:pos]):
                # Verifica que no este incluido un caracter especial no permitido en el nombre del archivo
                if (variable == caract):
                    return False
        # Retorna verdadero ya que no encontro un caracter especial
        return True
    # Retorna falso si no encontro la extensión en la expresión
    return False
```

Función isComentario

Complejidad: O(1), porque no realiza ningún ciclo o recursión, lee las instrucciones 1 sola vez.

```
def isComentario(expresion):
    # Busca si hay "//" en la expresión
    pos = expresion.find("//")
    # Retorna verdadero si encontro "//" en la expresión
    if (pos == 0):
        return True
    # Retorna falso, en caso contrario
    return False
```

Función isLibreria

Complejidad: O(1), porque no realiza ningún ciclo o recursión, lee las instrucciones 1 sola vez.

```
def isLibreria(expresion):
    #Busca el # que en C++ indica una libreria a incluir
    pos = expresion.find("#")
    #Dado caso de que la encuentre marcalo como verdadero
    if (pos == 0):
        return True
    return False
```

Función isReservada

Complejidad: O(1), porque no realiza ningún ciclo o recursión, lee las instrucciones 1 sola vez.

```
def isReservada(expresion):
    # Se definen las palabras reservadas como un diccionario
    reservada = {"int": True, "bool": True, "char": True, "void": True, "float": True, "double": True, "string": True, "cin": True, "cout": True, "while": True,
    "as": True, "using": True, "namespace": True, "auto": True, "const": True, "asm": True, "dynamic_cast": True, "reinterpret_cast": True, "try": True,
    "explicit": True, "new": True, "static_cast": True, "static": True, "typeid": True, "catch": True, "false": True, "operator": True, "template": True,
    "typename": True, "class": True, "friend": True, "private": True, "this": True, "const_cast": True, "inline": True, "public": True, "throw": True,
    "virtual": True, "delete": True, "enum": True, "goto": True, "else": True, "mutable": True, "protected": True, "true": True, "wchar_t": True, "endl": True,
    "sizeof": True, "register": True, "unsigned": True, "break": True, "continue": True, "extern": True, "if": True, "return": True, "switch": True, "case": True,
    "default": True, "short": True, "struct": True, "volatile": True, "do": True, "for": True, "long": True, "signed": True, "union": True, "std": True,}
    # Verifica si existe en la expresión cualquier palabra reservada, si no, retorna falso
    try:
        return reservada[expresion]
    except:
        return False
```

Función isOperador

Complejidad: O(1), porque aunque se maneja 1 ciclo, este siempre va a ser constante, su valor no es variable, ni depende de un parámetro.

```
def isOperador(expresion, original, pos):
    # Se definen los operadores como una lista
    operador = ["+", "+=", "++", "-", "-=", "--", "%", "%=", "*", "*=", "/=", "^", "<", "<<", ">", ">>", "<=", ">=", "=", "==", "!", "!=", "~", "?", "&", "&&", "||"]
    # Ciclo que itera cada operador de la lista definida
    for i, op in enumerate(operador):
        # Si encontro el operador, se retorna verdadero
        if (op == expresion or expresion.find(op) != -1 or (expresion == "/" and original[pos:pos+2] != "//") or (expresion.find("/") != -1 and original[pos:pos+2] != "//")):
            return True
    return False
```

Función isOperadorUnique

Complejidad: O(1), porque no realiza ningún ciclo o recursión, lee las instrucciones 1 sola vez.

```
def isOperadorUnique(expresion, original, pos):
    # Se definen los operadores como un diccionario
    operador = {"+": True, "+=": True, "++": True, "-": True, "-=": True, "--": True, "%": True, "%=": True, "*": True, "*=": True, "/=": True, "^": True, "<": True, "<<": Tru
    "<=": True, ">=": True, "=": True, "==": True, "!": True, "!=": True, "~": True, "?": True, "&": True, "&&": True, "||": True}
    # Verifica si existe en la expresión cualquier operador, si no, retorna falso
    try:
        # Verifica que no vaya a sobrepasar la longitud de la expresión original
        if (pos+2 < len(original)):
            # Verifica que no sea un comentario lo que se esta leyendo
            if ((expresion == "/" and original[pos:pos+2] != "//") and (expresion == "/" and original[pos:pos+2] != "/*")):
                return True
            # Si encuentra un operador retorna verdadero
            else:
                return operador[expresion]
    # En caso contrario, retorna falso
    except:
        return False
```

Función isDelimitador

Complejidad: O(1), porque aunque se maneja 1 ciclo, este siempre va a ser constante, su valor no es variable, ni depende de un parámetro.

```
def isDelimitador(expresion):
    # Se definen los delimitadores como una lista
    delimitador = ["(", ")", "[", "]", "{", "}", ",", ";", "...", ":"]
    # Ciclo que itera cada delimitador de la lista definida
    for i, delim in enumerate(delimitador):
        # Si encontro el delimitador, se retorna verdadero
        if (delim == expresion or expresion.find(delim) != -1):
            return True
    return False
```

Función isIdentificador

Complejidad: O(1), porque aunque se maneja 1 ciclo, este siempre va a ser constante, su valor no es variable, ni depende de un parámetro.

```
def isIdentificador(expresion, original, pos):
    # Se crea una lista para checar todos los identificadores con letras
    alfabeto = list(string.ascii_letters)
    # Se crea una lista para números
    numeros = []
    # Añade a la lista los numeros del 0 al 9
    for x in range(0, 10):
        numeros.append(str(x))
    # Ciclo que itera cada letra del alfabeto
    for i, letra in enumerate(alfabeto):
        # Si encontro una letra del afabeto o un guión entra a la siguiente condicional
        if (letra == expresion or expresion.find(letra) != -1 or (expresion in numeros) or "_" in expresion):
            # Checa casos de excepcion que indican que no es un identificador
            if ((expresion[0] in alfabeto) and (not ("\"" in expresion or "\'" in expresion or "." in expresion or "#" in expresion))):
                return True
    return False
```

Función isLiteral

Complejidad: O(n), porque se tiene un ciclo que maneja un parámetro que es variable, puesto que depende de la longitud de la expresión.

```
def isLiteral(expresion, original, pos, operador):
    # Se declaran variables para manejar los casos de excepción que nos indican que no son literales
    numeros = []
    punto = False
    guion = False
    eReal = False
    fReal = False
    uReal = False
    lReal = 0
    wait = False
    letter = False

    # Si se encontro una o doble comilla, retorna verdadero, ya que se esta por leer un string o char
    if (expresion[0] == "\'" or expresion[0] == "\""):
        return True

    # Añade los numeros del 0 al 9
    for x in range(0, 10):
        numeros.append(str(x))
```

```
try:
    # Guarda la posicion en la que se encuentra el inicio de la expresion
    pos2 = original.find(expresion)
    # Ciclo que itera cada caracter de la expresion
    for i, x in enumerate(expresion):
        # Verifica que sea un número
        if (expresion[i] in numeros and not letter):
            wait = False
        else:
            # Verifica si es real
            if (expresion[i] == "." and not punto and original[pos2+i+1] in numeros):
                punto = True
            # Verifica si es un dato de tipo long o long long
            elif ((expresion[i] == "L" or expresion[i] == "l") and not wait and ((lReal < 2 and not uReal) or (uReal and lReal == 0) or expresion[:i].find("ul") != -1) and
                lReal = lReal + 1
                letter = True
            # Verifica si es un dato de tipo unsigned
            elif ((expresion[i] == "U" or expresion[i] == "u") and not uReal and not punto and not eReal and original[pos2+i+1] != "."):
                uReal = uReal + 1
                letter = True
            # Verifica si se esta leyendo la "E" de la notación científica y que reciba un número o guión a continuación
            elif ((expresion[i] == "E" or expresion[i] == "e") and not eReal and (original[pos2+i+1] in numeros or original[pos2+i+1] == "-")):
                operador[0] = True
                eReal = True
                wait = True
            # Verifica si es un guión y que reciba un número a continuación
            elif (expresion[i] == "-" and not guion and eReal and original[pos2+i+1] in numeros):
                guion = True
                wait = True
            # Verifica si es un fast int
            elif ((expresion[i] == "F" or expresion[i] == "f") and not wait and not uReal and lReal == 0 and eReal):
                fReal = fReal + 1
                letter = True
            else:
                return False
    return True
except:
    return False
```