TECNOLÓGICO DE MONTERREY®

TC2037 – IMPLEMENTACIÓN DE MÉTODOS COMPUTACIONALES
GRUPO 604

Actividad Integradora 5.3
Resaltador de Sintaxis Paralelo

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

10 de junio de 2022

Solución a la Situación Problema

En la presente actividad integradora se desarrollaron 2 versiones para una misma situación problema, en donde se busca determinar la categoría léxica a la que pertenece cada entrada *n*. Las versiones desarrolladas fueron de manera secuencial y paralela, en donde se busca que en esta última, el tiempo de ejecución sea menor que la versión secuencial.

La solución planteada para dicho problema empieza buscando archivos de texto en el directorio donde el programa se encuentra ejecutándose. Después, a estos archivos de texto los almacena en un array para que más adelante pueda proceder a abrir cada uno de ellos, analizarlos y desplegar en un archivo HTML el resaltador de sintaxis correspondiente a dicho archivo. Por otro lado, para el algoritmo del resaltador de sintaxis, este fue el mismo que se utilizó en la actividad integradora pasada, la 3.4.

Analizando nuestra solución, podemos observar que presenta un tiempo de ejecución variable, ya que depende de la cantidad de archivos de texto encontrados, así como del contenido de cada archivo. Sin embargo, al presentar una gran cantidad de archivos de texto, el programa se puede volver ineficaz, en donde el tiempo de ejecución incrementa y dilata en presentar los resultados.

Para esto se desarrolló una versión paralela de nuestro programa secuencial, en donde dicha metodología reduce y optimiza el tiempo de ejecución al analizar varios archivos a la vez, lo que lo vuelve mucho más eficaz que la versión secuencial.

Versión Secuencial

Actividad 5.3 > resaltadorSecuencial.go > ...

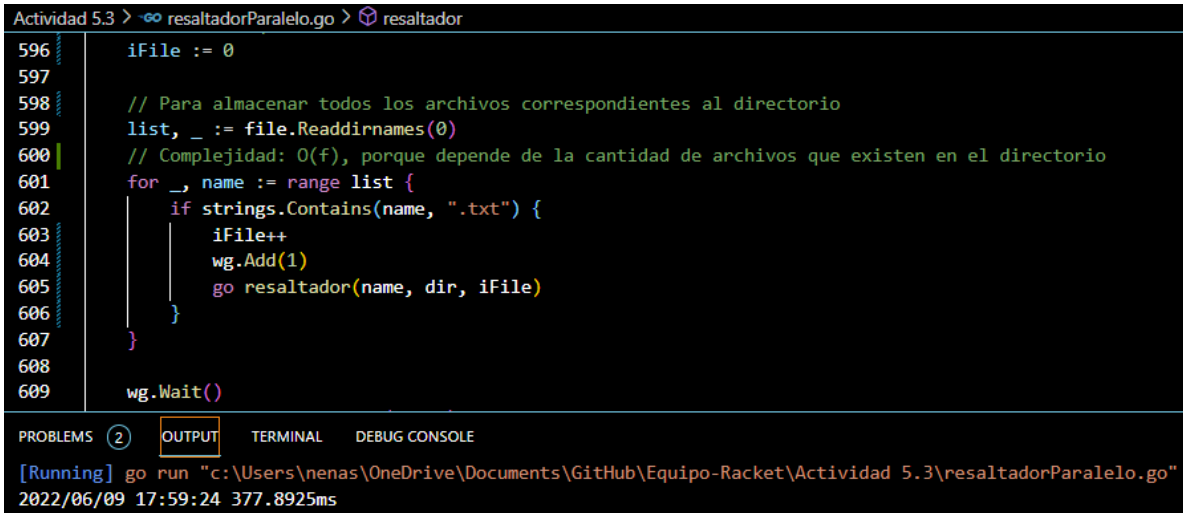
```
592     // Variable que almacenará la cantidad de archivos txt
593     iFile := 0
594
595     // Para almacenar todos los archivos correspondientes al directorio
596     list, _ := file.Readdirnames(0)
597     // Complejidad: O(f), porque depende de la cantidad de archivos que existen en el directorio
598     for _, name := range list {
599         if strings.Contains(name, ".txt") {
600             iFile++
601             resaltador(name, dir, iFile)
602         }
603     }
604
605     sinceStart := time.Since(start)
```

PROBLEMS 2 OUTPUT TERMINAL DEBUG CONSOLE

```
[Running] go run "c:\Users\nenas\OneDrive\Documents\GitHub\Equipo-Racket\Actividad 5.3\resaltadorSecuencial.go"
2022/06/09 17:57:20 780.7458ms
```

Tiempo de ejecución: 780.7458 ms

Versión Paralela



Tiempo de ejecución: 377.8925 ms

Cálculo del Speedup

$$S_p = \frac{T_1}{T_p}$$

Prueba 1

Secuencial

```
[Running] go run "c:\Users\nenas\OneDrive\Documents\GitHub\Equipo-Racket\Actividad 5.3\resaltadorSecuencial.go"
2022/06/09 17:57:20 780.7458ms
```

Paralela

```
[Running] go run "c:\Users\nenas\OneDrive\Documents\GitHub\Equipo-Racket\Actividad 5.3\resaltadorParalelo.go"
2022/06/09 17:59:24 377.8925ms
```

Utilizando como referencia los parámetros obtenidos por uno de los miembros del equipo, se obtuvo:

- T_1 = 780.7458 ms
- T_p = 377.8925 ms

$$S_p = \frac{T_1}{T_p} = \frac{780.7458}{377.8925} = 2.06605265$$

Prueba 2

Secuencial

```
[Running] go run "c:\Users\nenas\OneDrive\Documents\GitHub\Equipo-Racket\Actividad 5.3\resaltadorSecuencial.go"
2022/06/09 18:05:28 791.7136ms
```

Paralela

```
[Running] go run "c:\Users\nenas\OneDrive\Documents\GitHub\Equipo-Racket\Actividad 5.3\resaltadorParalelo.go"
2022/06/09 18:05:33 393.719ms
```

Utilizando como referencia los parámetros obtenidos por uno de los miembros del equipo, se obtuvo:

- T_1 = 791.7136 ms
- T_p = 393.719 ms

$$S_p = \frac{T_1}{T_p} = \frac{791.7136}{393.719} = 2.01085952$$

Complejidad del Algoritmo Planteado

Realizando el cálculo de la complejidad del algoritmo basándonos en el número de iteraciones de nuestro programa, se 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, por lo que no dependen de ningún valor o parámetro variable.

Sin embargo, para la función *resaltador*, que es la función que identifica la categoría léxica a la que pertenece cada entrada de cierto archivo, se calcula una complejidad en el mejor de los casos de O(n^2 * r^2), en donde el programa se encontrará con un archivo de comentarios largos sin errores de sintaxis. Por otro lado, en el peor de los casos es que entre en cada una de las condiciones declaradas dentro de esta función, en donde se calcula una complejidad de O(n^8 * r).

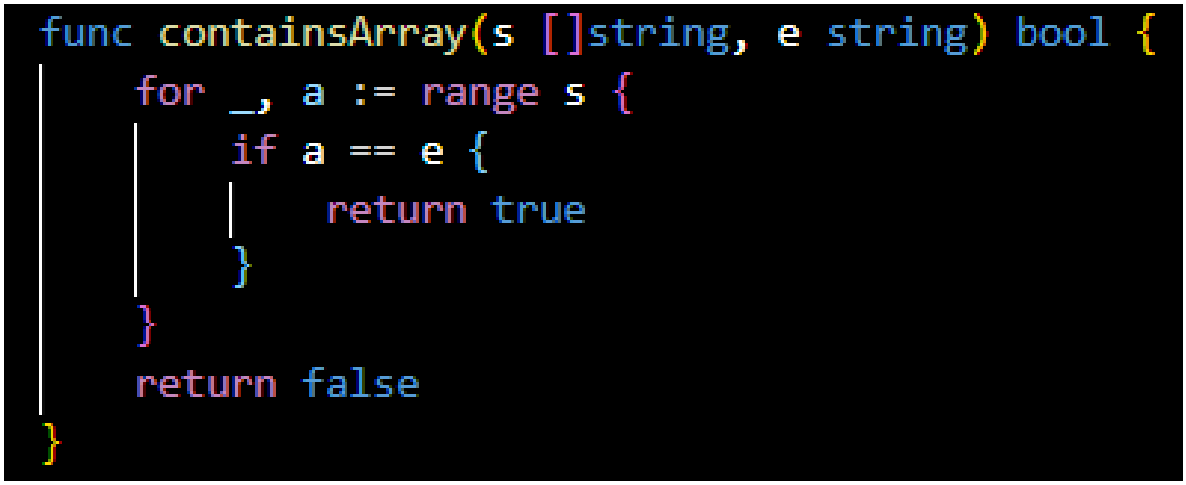
Como son varios archivos a procesar, de manera general, en el mejor de los casos, se tiene una complejidad de O(f + (n^2 * r^2)), en donde “f” es la cantidad de archivos dentro del directorio. Por otro lado, en el peor de los casos, se calcula una complejidad de O(f + (n^8 * r)).

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

Función containsArray

Complejidad: O(x), porque se maneja 1 ciclo que depende de un parámetro que puede ser variable.

Complejidad para el fin que se utiliza: O(1), porque aunque se maneja 1 ciclo, la cantidad de veces que va a estar iterando es constante, ya que el parámetro que es enviado y utilizado por el ciclo es constante.

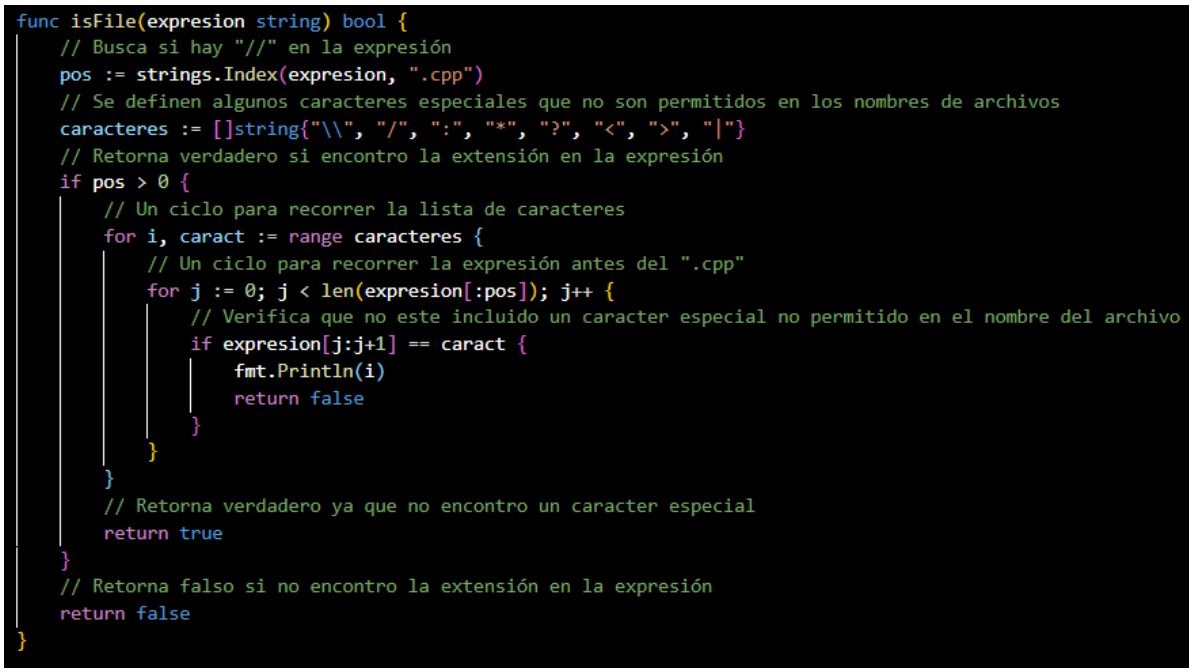


```go
func containsArray(s []string, e string) bool {
    for _, a := range s {
        if a == e {
            return true
        }
    }
    return false
}
```

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.



```go
func isFile(expresion string) bool {
    // Busca si hay "//" en la expresión
    pos := strings.Index(expresion, ".cpp")
    // Se definen algunos caracteres especiales que no son permitidos en los nombres de archivos
    caracteres := []string{"\\", "/", ":", "*", "?", "<", ">", "|"}
    // Retorna verdadero si encontro la extensión en la expresión
    if pos > 0 {
        // Un ciclo para recorrer la lista de caracteres
        for i, caract := range caracteres {
            // Un ciclo para recorrer la expresión antes del ".cpp"
            for j := 0; j < len(expresion[:pos]); j++ {
                // Verifica que no este incluido un caracter especial no permitido en el nombre del archivo
                if expresion[j:j+1] == caract {
                    fmt.Println(i)
                    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.

```go
func isComentario(expresion string) bool {
    // Busca si hay "//" en la expresión
    pos := strings.Index(expresion, "//")
    // 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.

```go
func isLibreria(expresion string) bool {
    //Busca el # que en C++ indica una libreria a incluir
    pos := strings.Index(expresion, "#")
    //Dado caso de que la encuentre marcalo como verdadero
    if pos == 0 {
        return true
    }
    return false
}
```

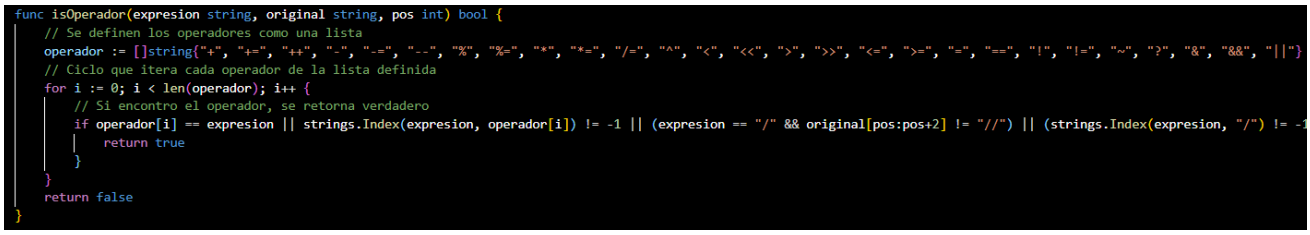
Función isReservada

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.

```go
func isReservada(expresion string) bool {
    // Se definen las palabras reservadas como un array
    reservada := []string{"int", "bool", "char", "void", "float", "double", "string", "cin", "cout", "while",
        "as", "using", "namespace", "auto", "const", "asm", "dynamic_cast", "reinterpret_cast", "try",
        "explicit", "new", "static_cast", "static", "typeid", "catch", "false", "operator", "template",
        "typename", "class", "friend", "private", "this", "const_cast", "inline", "public", "throw",
        "virtual", "delete", "enum", "goto", "else", "mutable", "protected", "true", "wchar_t", "endl",
        "sizeof", "register", "unsigned", "break", "continue", "extern", "if", "return", "switch", "case",
        "default", "short", "struct", "volatile", "do", "for", "long", "signed", "union", "std"}
    // Verifica si existe en la expresión cualquier palabra reservada, si no, retorna falso
    for i := 0; i < len(reservada); i++ {
        // Si encontro el delimitador, se retorna verdadero
        if reservada[i] == expresion {
            return true
        }
    }
    return false
}
```

Función isOperador

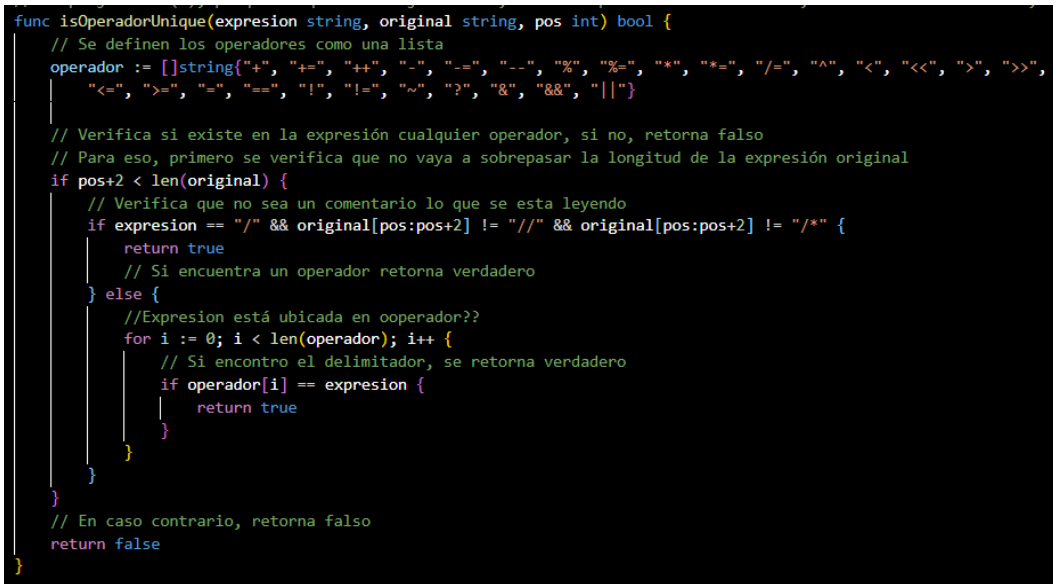
Complejidad: O(1), porque aunque se maneja 1 ciclo, este siempre va a ser constante, su valor no es variable, ni depende de un parámetro.



```
func isOperador(expresion string, original string, pos int) bool {
    // Se definen los operadores como una lista
    operador := []string{"+", "+=", "++", "-", "-=", "--", "%", "%=", "*", "*=", "/=", "^", "<", "<<", ">", ">>", "<=", ">=", "=", "==", "!", "!=", "~", "?", "&", "&&", "||"}
    // Ciclo que itera cada operador de la lista definida
    for i := 0; i < len(operador); i++ {
        // Si encontro el operador, se retorna verdadero
        if operador[i] == expresion || strings.Index(expresion, operador[i]) != -1 || (expresion == "/" && original[pos:pos+2] != "//") || (strings.Index(expresion, "/") != -1
            return true
        }
    }
    return false
}
```

Función isOperadorUnique

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.

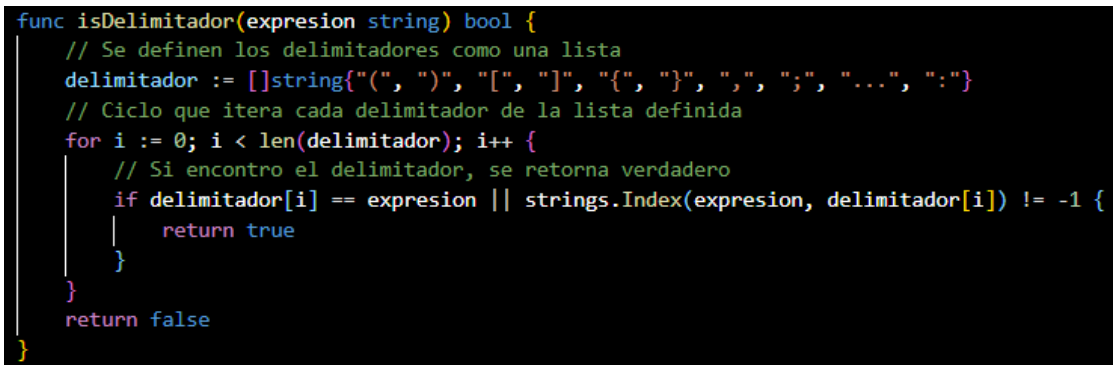


```
func isOperadorUnique(expresion string, original string, pos int) bool {
    // Se definen los operadores como una lista
    operador := []string{"+", "+=", "++", "-", "-=", "--", "%", "%=", "*", "*=", "/=", "^", "<", "<<", ">", ">>",
        "<=", ">=", "=", "==", "!", "!=", "~", "?", "&", "&&", "||"}

    // Verifica si existe en la expresión cualquier operador, si no, retorna falso
    // Para eso, primero se 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 == "/" && original[pos:pos+2] != "//" && original[pos:pos+2] != "/*" {
            return true
            // Si encuentra un operador retorna verdadero
        } else {
            //Expresion está ubicada en ooperador??
            for i := 0; i < len(operador); i++ {
                // Si encontro el delimitador, se retorna verdadero
                if operador[i] == expresion {
                    return true
                }
            }
        }
    }
    // En caso contrario, retorna falso
    return false
}
```

Función isDelimitador

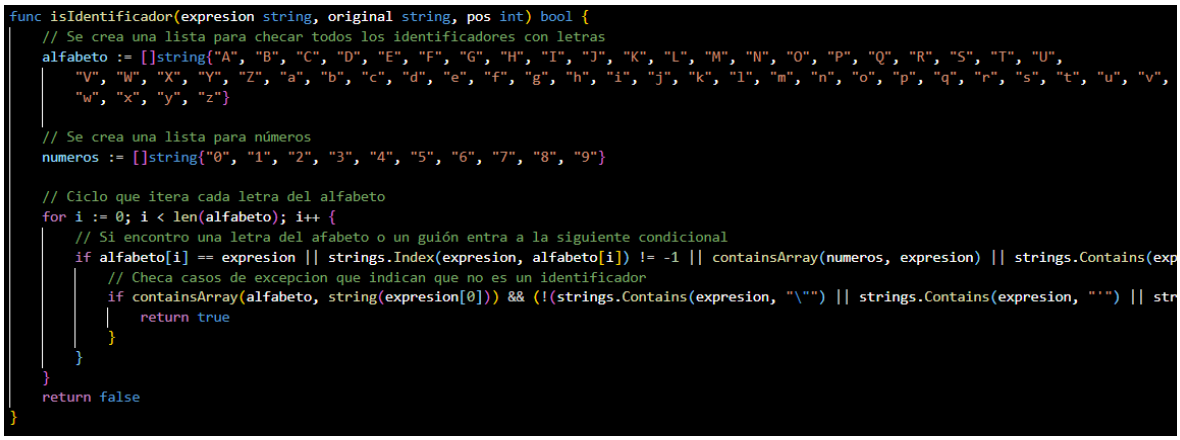
Complejidad: O(1), porque aunque se maneja 1 ciclo, este siempre va a ser constante, su valor no es variable, ni depende de un parámetro.



```
func isDelimitador(expresion string) bool {
    // Se definen los delimitadores como una lista
    delimitador := []string{"(", ")", "[", "]", "{", "}", ",", ";", "...", ":"}
    // Ciclo que itera cada delimitador de la lista definida
    for i := 0; i < len(delimitador); i++ {
        // Si encontro el delimitador, se retorna verdadero
        if delimitador[i] == expresion || strings.Index(expresion, delimitador[i]) != -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.



```
func isIdentificador(expresion string, original string, pos int) bool {
    // Se crea una lista para checar todos los identificadores con letras
    alfabeto := []string{"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U",
        "V", "W", "X", "Y", "Z", "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v",
        "w", "x", "y", "z"}

    // Se crea una lista para números
    numeros := []string{"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"}

    // Ciclo que itera cada letra del alfabeto
    for i := 0; i < len(alfabeto); i++ {
        // Si encontro una letra del afabeto o un guión entra a la siguiente condicional
        if alfabeto[i] == expresion || strings.Index(expresion, alfabeto[i]) != -1 || containsArray(numeros, expresion) || strings.Contains(exp
            // Checa casos de excepcion que indican que no es un identificador
            if containsArray(alfabeto, string(expresion[0])) && (!(strings.Contains(expresion, "\"") || strings.Contains(expresion, "'") || str
                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.

```
func isLiteral(expresion string, original string, pos int, operador *bool) bool {
    // Se declaran variables para manejar los casos de excepción que nos indican que no son literales
    numeros := []string{"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"}
    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 string(expresion[0]) == "'" || string(expresion[0]) == "\"" {
        return true
    }

    // Guarda la posicion en la que se encuentra el inicio de la expresion
    pos2 := strings.Index(original, expresion)

    // Ciclo que itera cada caracter de la expresion
    for i := 0; i < len(expresion); i++ {
        //Verifica que sea un número
        if (containsArray(numeros, string(expresion[i]))) && (!letter) {
            wait = false
        } else {
```

Función resaltador

Complejidad en el mejor de los casos: O(n^2 * r^2), en donde el contenido del archivo tiene comentarios largos sin errores de sintaxis.

Complejidad en el peor de los casos: O(n^8 * r), en donde el archivo tiene un contenido que identificará como literales o en ninguna categoría, por lo que entrará en cada condicional para su identificación.

```
func resaltador(file string, dir string, iFile int) {
    // Lista que guardará el contenido del archivo TXT
    lista_sintaxis := []string{}

    // Abre el archivo de texto (sintaxis.txt)
    fileTxt, ferr := os.Open(dir + "\\Actividad 5.3\\" + file)
    if ferr != nil {
        panic(ferr)
    }
    scanner := bufio.NewScanner(fileTxt)
    for scanner.Scan() {
        lista_sintaxis = append(lista_sintaxis, scanner.Text())
    }

    // Se abre o se crea un archivo html (index.html)
    fileHtml, e := os.Create(dir + "\\Actividad 5.3\\index" + fmt.Sprint(iFile) + ".html")
    if e != nil {
        fmt.Println(e)
    }

    // Escribimos el head del archivo html
    fileHtml.WriteString("<!DOCTYPE html>\n")
    fileHtml.WriteString("<html>\n")
    fileHtml.WriteString("\t<head>\n")
    fileHtml.WriteString("\t\t<meta charset=\"utf-8\"/>\n")
    fileHtml.WriteString("\t\t<title>Resaltador de Sintaxix</title>\n")
    fileHtml.WriteString("\t\t<link rel=\"stylesheet\" href=\"style.css\">\n")
    fileHtml.WriteString("\t</head>\n")
    fileHtml.WriteString("\t<body>\n")
```

Conclusión

En conclusión, podemos analizar que teniendo un archivo con una gran cantidad de información o inclusive, que fueran múltiples archivos, el programa tardaría más en procesar dichos elementos, sin mencionar que dicha información contuviera, en el peor de los casos, solo literales u errores, lo que solo incrementaría más el tiempo de ejecución.

Por otro lado, al implementar el paralelismo, se pudo observar que la versión secuencial de la situación problema obtuvo un tiempo de ejecución de aproximadamente 780 milisegundos, en cambio, la versión paralela demostró tener un tiempo de ejecución mucho menor a la versión anterior (secuencial), en donde se pudo analizar que dicho tiempo se redujo hasta aproximadamente un 50% del tiempo de la versión secuencial. Esto demuestra la importancia de la programación paralela en programas exigentes o con un tiempo de ejecución tardado, disminuyendo considerablemente el tiempo de ejecución del programa al utilizar de manera más eficiente la capacidad de procesamiento de las computadoras utilizadas, al emplearlas para que se realicen múltiples tareas de manera simultánea.