ANALIZADOR LÉXICO Y SINTÁCTICO

MATERIA: LENGUAJES Y AUTÓMATAS

PROF. Jesús Francisco Núñez Tanori

S7C

JESUS RAFAEL VALLES FITCH 19330677
JOSÉ JORGE GASTELUM ANGULO 19330572
BELTRÁN SANTIAGO IVÁN DANIEL 19330537

Viernes 22 de Noviembre de 2023, Hermosillo, Sonora

Índice

Objetivos **3**
Marco teórico **4**
Go (Golang) 4
Mayúsculas intercaladas 6
For 8
Switch 9
Switch de tipo 10
Asignación con new 11
Asignación con make 11
Sectores 12
Sectores bidimensionales 13
Impresión 13
Constantes 14
Variables 14
Autómata finito 15
Historia 15
Fase de análisis léxico 16
Fase de análisis sintáctico 17
Metodología **18**
Materiales **20**
Desarrollo **21**
Autómata finito 21
Diagrama de clases 25
Matriz de transición 28
Diagrama de bloques 28
Código 35
Resultados **79**
Corrida sin errores 79
Error 502 se espera un = 81
Se espera un identificador en la línea 3 82
Error 505 se espera cierre de cadena (“) 83
Se espera un ; 84
Se espera un identificador 85
Conclusiones **86**
Referencias bibliográficas **87**

Objetivos

El propósito fundamental de este proyecto de dos partes, de la cual nos enfocaremos en la primera de ellas, es diseñar, implementar y evaluar un analizador léxico basado en el lenguaje de programación Go. Este proyecto se enfoca en alcanzar varios objetivos clave que no solo fortalecerán la comprensión sobre los conceptos fundamentales de los analizadores léxicos, sino que también proporcionarán una oportunidad práctica para aplicar los conocimientos adquiridos a lo largo de la vida académica y laboral.

Al alcanzar objetivos como la comprensión de los principios teóricos del analizador léxico, su diseño e implementación, su manejo de excepciones y errores, y la correcta documentación del proceso de desarrollo de este tipo de aplicaciones, se espera estar mejor preparado para abordar desafíos más avanzados en el campo de la compilación y el análisis de lenguajes de programación, pudiendo dar paso al desarrollo del resto del compilador, tales como el análisis sintáctico.

Marco teórico

Go (Golang)

El lenguaje de programación Go es un proyecto de código abierto para hacer programadores más productivos.

Go es expresivo, conciso, limpio y eficiente. Sus mecanismos de concurrencia facilitan la escritura de programas consiguiendo lo mejor de máquinas multinúcleo y de la red, mientras su novel sistema de tipos permite la construcción de programas flexibles y modulares. Go compila rápidamente a código máquina aún con la comodidad de la recolección de basura y el poder de reflexión en tiempo de ejecución. Es un lenguaje tipado estáticamente, compilado y por lo tanto rápido, que se siente como un lenguaje interpretado y tipado dinámicamente.

Comentarios

Go proporciona comentarios de bloque al estilo de C `/* */` y comentarios de línea al estilo de C + + `//`. Los comentarios de línea son la norma; los comentarios de bloque generalmente aparecen como comentarios de paquete, pero son útiles dentro de una expresión o para desactivar grandes franjas de código.

Nomenclatura de paquetes

Cuando se importa un paquete, el nombre del paquete proviene de un método de acceso al contenido. Después de:

```
import "bytes"
```

El paquete importador puede hablar sobre `bytes.Buffer`. Es útil si todos los que utilizan el paquete pueden usar el mismo nombre para referirse a su contenido, lo cual implica que el nombre del paquete tendría que ser bueno: corto, conciso, evocador. Por convención, a los paquetes se les dan nombres de una sola palabra

en minúsculas. El nombre del paquete es solo el nombre predeterminado para importaciones; este no tiene que ser único entre todo el código fuente y en el raro caso de una colisión el paquete importador puede elegir un nombre diferente para utilizarlo localmente.

El importador de un paquete utilizará el nombre para referirse a su contenido, por lo tanto los nombres exportados en el paquete pueden utilizar este hecho para evitar confusiones. Por ejemplo, el tipo lector de búfer en el paquete `bufio` se llama `Reader`, no `BufReader`, debido a que los usuarios ven `bufio.Reader`, porque es un nombre claro y conciso.

Captadores

Go no proporciona soporte automático para captadores y definidores. No hay nada incorrecto en proporcionar captadores y definidores y a menudo es apropiado hacerlo, pero tampoco es idiomático ni necesario poner `Obt` al nombre del captador. Si tienes un campo llamado `propietario` (en minúsculas, no exportado), el método captador se tendría que llamar `Propietario` (en mayúsculas, exportado), no `ObtPropietario`. El uso de mayúsculas en los nombres para exportación proporciona el gancho para diferenciar un campo de un método. Una función definidora, si se necesita, probablemente se llamará `EstPropietario`. Ambos nombres se leen bien en la práctica:

```
propietario := obj.Propietario()

if propietario != usuario {

   obj.EstPropietario(usuario)

}
```

Nombre de interfaces

Por convención, para denominar un método de interfaz se utiliza el nombre del método más un sufijo -er o modificación similar para construir un sustantivo del agente: `Reader`, `Writer`, `Formatter`, `CloseNotifier`, etc.

Mayúsculas intercaladas

La convención en Go es utilizar `MayúsculasIntercaladas` o `mayúsculasIntercaladas` en lugar de guiones bajos para escribir nombres multipalabra.

Estructuras de control

Las estructuras de control de Go están relacionadas a las de C pero difieren en importantes maneras. No hay bucles `do` o `while`, solo un ligeramente generalizado `for`; `switch` es más flexible; `if` y `switch` aceptan una declaración de inicio opcional como la del `for`; las declaraciones `break` y `continue` toman una etiqueta opcional para identificar qué interrumpir o continuar; y hay nuevas estructuras de control incluyendo un tipo `switch` y un multiplexor de comunicaciones multivía, `select`. La sintaxis también es ligeramente diferente: no hay paréntesis y los cuerpos siempre tienen que estar delimitados por llaves.

```
if x > 0 {
    return y
}
```

Dado que `if` y `switch` aceptan una declaración de iniciación, es común ver una usada para configurar una variable local.

```
if err := file.Chmod(0664); err != nil {
    log.Print(err)
    return err
}
```

Redeclaración y reasignación

El último ejemplo en la sección anterior demuestra un detalle de cómo trabaja la declaración corta de variables :=. La declaración que llama a os.Open dice:

```
f, err := os.Open(nombre)
```

Esta declaración crea dos variables, f y err. Unas cuantas líneas más abajo, la llamada a f.Stat dice,

```
d, err := f.Stat()
```

la cual se ve como si declarara d y err. Observa que, no obstante, err aparece en ambas declaraciones. Esta duplicidad es legal: err fue creada en la primera declaración, pero únicamente *reasignada* en la segunda. Esto significa que la llamada a f.Stat utiliza la variable err existente declarada arriba y solo le da un nuevo valor.

En una declaración := puede aparecer una variable v incluso si ya se ha declarado, siempre y cuando:

- Esa declaración esté en el mismo ámbito que la declaración existente de `v` (si `v` ya estuviera declarada en un ámbito exterior, la declaración creará una nueva variable),
- El valor correspondiente en la iniciación es asignable a `v` y
- Cuándo menos se crea una nueva variable en esa declaración.

For

El bucle `for` de Go es similar —a pero no igual— al de C. Este unifica `for` y `while` y no hay `do-while`. Hay tres formas, solo una de las cuales tiene puntos y comas.

// Como un for C

for inicio; condición; incremento { }

// Como un while C

for condición { }

// Como un for(;;) C

for { }

Switch

El switch de Go es más general que el de C. Las expresiones no es necesario que sean constantes o incluso enteros, los casos se evalúan de arriba hacia abajo hasta encontrar una coincidencia y si el switch no tiene una expresión este cambia a true. Por lo tanto es posible —e idiomático— escribir una cadena de if-else-if-else como un switch.

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
    return 0
}
```

Switch de tipo

Un switch también puede descubrir dinámicamente el tipo de una variable de interfaz. Tal *switch de tipo* utiliza la sintaxis de una aserción de tipo con la palabra clave type dentro de los paréntesis. Si el switch declara una variable en la expresión, la variable tendrá el tipo correspondiente en cada cláusula. También es idiomático reutilizar el nombre en tales casos, en efecto declarando una nueva variable con el mismo nombre pero un diferente tipo en cada caso.

```
var t interface{}

t = funcionDeAlgunTipo()

switch t := t.(type) {

default:

   fmt.Printf("tipo inesperado %T\n", t)     // imprime %T cuando t tiene tipo

case bool:

   fmt.Printf("lógico %t\n", t)             // t es de tipo bool

case int:

   fmt.Printf("entero %d\n", t)             // t es de tipo int

case *bool:

   fmt.Printf("puntero a lógico %t\n", *t)   // t es de tipo *bool

case *int:

   fmt.Printf("puntero a entero %d\n", *t)   // t es de tipo *int

}
```

Asignación con new

Go tiene dos primitivas de asignación, las funciones incorporadas new y make. Son dos cosas diferentes y se aplican a diferentes tipos, lo cual puede ser confuso, pero las reglas son sencillas. Primero hablemos sobre new. Es una función incorporada que reserva memoria, pero a diferencia de su homónima en algunos otros lenguajes **no** *inicia* la memoria, solo la pone a *ceros*. Es decir, new(T) reserva almacenamiento establecido a cero para un nuevo elemento de tipo T y regresa su dirección, un valor de tipo *T. En terminología Go, regresa un puntero a un recién alojado valor cero de tipo T.

Asignación con make

Volviendo a la asignación. La función incorporada make(T,*args*) sirve a un propósito diferente de new(T). Esta solamente crea sectores, mapas y canales y regresa un valor de tipo *T* (no **T*) iniciado (no con ceros). La razón para tal distinción es que estos tres tipos, bajo la cubierta, representan referencias a estructuras de datos que se tienen que iniciar antes de usarlas. Un sector, por ejemplo, es un descriptor de tres elementos que contiene un puntero al dato (dentro de un arreglo), su longitud y capacidad y hasta que esos elementos sean iniciados, el sector es nil. Para sectores, mapas y canales, make inicia la estructura de datos interna y prepara el valor para usarlo.

Arreglos

Hay importantes diferencias entre la manera en que trabajan los arreglos de Go a como lo hacen en C. En Go:

- Los arreglos son valores. Al asignar un arreglo a otro se copian todos los elementos.
- En particular, si pasas un arreglo a una función, esta recibe una *copia* del arreglo, no un puntero a él.
- El tamaño de un arreglo es parte de su tipo. Los tipos `[10]int` y `[20]int` son distintos.

Sectores

Los sectores envuelven arreglos para dotarlos de una interfaz más general, potente y conveniente para secuencias de datos. Salvo los elementos con dimensión explícita tal como arreglos de transformación, la mayoría de la programación de arreglos en Go está hecha con sectores en lugar de arreglos sencillos.

Los sectores mantienen referencias a un arreglo subyacente y si asignas un sector a otro, ambos se refieren al mismo arreglo. Si una función toma un sector como argumento, los cambios que hace a los elementos del sector serán visibles al llamador, análogo a pasar un puntero al arreglo subyacente. Una función `Read` por lo tanto puede aceptar un sector como argumento en lugar de un puntero y un contador; la longitud dentro del sector impone un límite máximo de cuantos datos leer.

Sectores bidimensionales

En Go los arreglos y sectores son unidimensionales. Para crear el equivalente de un arreglo o sector 2D, es necesario definir un arreglo de arreglos o un sector de sectores, de la siguiente manera:

type Transformación [3][3]float64 // Un arreglo 3x3, en realidad un arreglo de arreglos.

type LíneasDeTexto [][]byte // Un sector de sectores de byte.

Impresión

La impresión formateada en Go usa un estilo similar al de la familia printf de C pero es más rica y más general. Estas funciones viven en el paquete fmt y tienen nombres capitalizados: fmt.Printf, fmt.Fprintf, fmt.Sprintf y así por el estilo. Las funciones de cadena (Sprintf, etc.) regresan una cadena en lugar de rellenar el búfer proporcionado.

No necesitas proporcionar una cadena de formato. Por cada Printf, Fprintf y Sprintf hay otro par de funciones, por ejemplo Print y Println. Estas funciones no toman una cadena de formato pero en cambio generan un formato predefinido para cada argumento. Las versiones Println también insertan un espacio entre argumentos y añaden un nuevo salto de línea al resultado mientras que las versiones Print solo añaden espacios si el operando en alguno de los lados es una cadena. En este ejemplo cada línea produce el mismo resultado.

fmt.Printf("Hola %d\n", 23)

fmt.Fprint(os.Stdout, "Hola ", 23, "\n")

fmt.Println("Hola", 23)

fmt.Println(fmt.Sprint("Hola ", 23))

Constantes

Las constantes en Go solo son eso —constantes. Estás se crean en tiempo de compilación, incluso cuando se definan como locales en funciones y solo pueden ser números, caracteres (runes), cadenas o lógicas. Debido a la restricción de tiempo de compilación, las expresiones que las definen tienen que ser expresiones constantes, evaluables por el compilador. Por ejemplo, 1<<3 es una expresión constante, mientras que math.Sin(math.Pi/4) no lo es porque la llamada a la función math.Sin necesita ocurrir en tiempo de ejecución.

Variables

Las variables se pueden iniciar justo como las constantes pero el iniciador puede ser una expresión general calculada en tiempo de ejecución.

```
var (
   home   = os.Getenv("HOME")
   user   = os.Getenv("USER")
   gopath = os.Getenv("GOPATH")
)
```

Autómata finito

Un autómata finito (AF) o máquina de estado finito es un modelo computacional que realiza cómputos en forma automática sobre una entrada para producir una salida.

Este modelo está conformado por:

- Un alfabeto
- Un conjunto de estados finito
- Una función de transición
- Un estado inicial
- Un conjunto de estados finales

Su funcionamiento se basa en una función de transición, que recibe a partir de un estado inicial una cadena de caracteres pertenecientes al alfabeto (la entrada), y que va leyendo dicha cadena a medida que el autómata se desplaza de un estado a otro, para finalmente detenerse en un estado final o de aceptación, que representa la salida.

La finalidad de los autómatas finitos es la de reconocer lenguajes regulares, que corresponden a los lenguajes formales más simples según la Jerarquía de Chomsky.

Historia

El origen de los autómatas finitos probablemente se remonta a su uso implícito en máquinas electromecánicas, desde principios del siglo XX. Ya en 1907, el matemático ruso Andréi Márkov formalizó un proceso llamado cadena de Markov, donde la ocurrencia de cada evento depende con una cierta probabilidad del evento anterior. Esta capacidad de "recordar" es utilizada posteriormente por los autómatas finitos, que poseen una memoria primitiva similar, en que la activación de un estado

también depende del estado anterior, así como del símbolo o palabra presente en la función de transición.

Posteriormente, en 1943, surge una primera aproximación formal de los autómatas finitos con el modelo neuronal de McCulloch-Pitts. Durante la década de 1950 prolifera su estudio, frecuentemente llamándoseles máquinas de secuencia; se establecen muchas de sus propiedades básicas, incluyendo su interpretación como lenguajes regulares y su equivalencia con las expresiones regulares. Al final de esta década, en 1959, surge el concepto de autómata finito no determinista en manos de los informáticos teóricos Michael O. Rabin y Dana Scott. En la década de 1960 se establece su conexión con las series de potencias y los sistemas de sobreescritura. Finalmente, con el desarrollo del sistema operativo Unix en la década de 1970, los autómatas finitos encuentran su nicho en el uso masivo de expresiones regulares para fines prácticos, específicamente en el diseño de analizadores léxicos (comando lex) y la búsqueda y reemplazo de texto (comandos ed y grep). A partir de ese tiempo, los autómatas finitos también se comienzan a utilizar en sistemas dinámicos.

Fase de análisis léxico

El analizador léxico es la primera fase de un compilador.

Su principal función consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis. Recibida la orden "obtén el siguiente componente léxico" del analizador sintáctico, el analizador léxico lee los caracteres de entrada hasta que pueda identificar el siguiente componente léxico.

También puede realizar ciertas funciones secundarias en la interfaz del usuario, como eliminar del programa fuente comentarios y espacios en blanco en forma de caracteres de espacio en blanco, caracteres TAB y de línea nueva.

Por ejemplo, el analizador léxico puede tener localizado el número de caracteres de nueva línea detectados, de modo que se pueda asociar un número de línea con un mensaje de error.En algunos casos, el analizador léxico puede leer la información relacionada con el tipo de información de la tabla de símbolos, como ayuda para determinar el token apropiado que debe pasar al analizador sintáctico.

Fase de análisis sintáctico

Es la fase del analizador que se encarga de chequear el texto de entrada en base a una gramática dada. Y en caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce.

Los analizadores sintácticos se usan en una amplia variedad de aplicaciones, como el procesamiento de lenguaje natural para chatbots y asistentes virtuales, la traducción automática de idiomas, la minería de texto y la indexación de documentos. También son esenciales en el desarrollo de compiladores, que convierten el código fuente escrito por un programador en un lenguaje de máquina que puede ser ejecutado por un equipo informático.

En teoría, se supone que la salida del analizador sintáctico es alguna representación del árbol sintáctico que reconoce la secuencia de Tokens suministrada por el analizador léxico.

En la práctica, el analizador sintáctico también:

- Accede a la tabla de símbolos (para hacer parte del trabajo del analizador semántico).
- Hace chequeo de tipos (del analizador semántico).
- Genera código intermedio.
- Genera errores cuando se producen.

En definitiva, realiza casi todas las operaciones de la compilación. Este método de trabajo da lugar a los métodos de compilación dirigidos por sintaxis.

Metodología

Las acciones a realizar para desarrollar el análisis léxico son:

- Obtener la tabla de símbolos basada en aquellos usados en el lenguaje GO.
- Definir las palabras reservadas y errores que puede generar el análisis léxico.
- Diseñar el autómata finito basado en la tabla de símbolos obtenida.
- Estructurar la matriz de transición acorde al autómata finito diseñado.
- Escribir un texto con el cual poner a prueba el analizador léxico.
- Programar el analizador léxico.
- Poner a prueba el analizador léxico.

La forma en que se realizará cada acción se hará de la siguiente forma:

- Investigar y leer sobre la sintaxis del lenguaje Go y tomar los elementos necesarios para formar la tabla de símbolos.
- Conforme se consigan los símbolos y palabras reservadas de la sintaxis del lenguaje Go, se definirán los errores que puede ocasionar un código mal escrito.
- Con los pasos anteriores ya realizados, diseñar el autómata finito en base a las posibles rutas que puede tomar el análisis dependiendo de qué se escriba en el código. Esto incluye definir hacia dónde parará el análisis en caso de encontrarse con un error.
- En base a los estados y símbolos obtenidos durante el diseño del autómata finito, se generará la matriz de transición.
- Tras haber hecho todo lo anterior, y en base a lo obtenido, comenzaremos a programar el analizador léxico tomando como base el tutorial publicado por la profesora de la asignatura.
- Por último, y habiendo comprobado que no nos faltan partes importantes del código, procederemos a hacer pruebas del analizador léxico en el IDE en que lo hayamos programado. Se realizarán las correcciones necesarias conforme los errores ocurran.

Algunos de los requerimientos necesarios para conseguir dichos objetivos son:

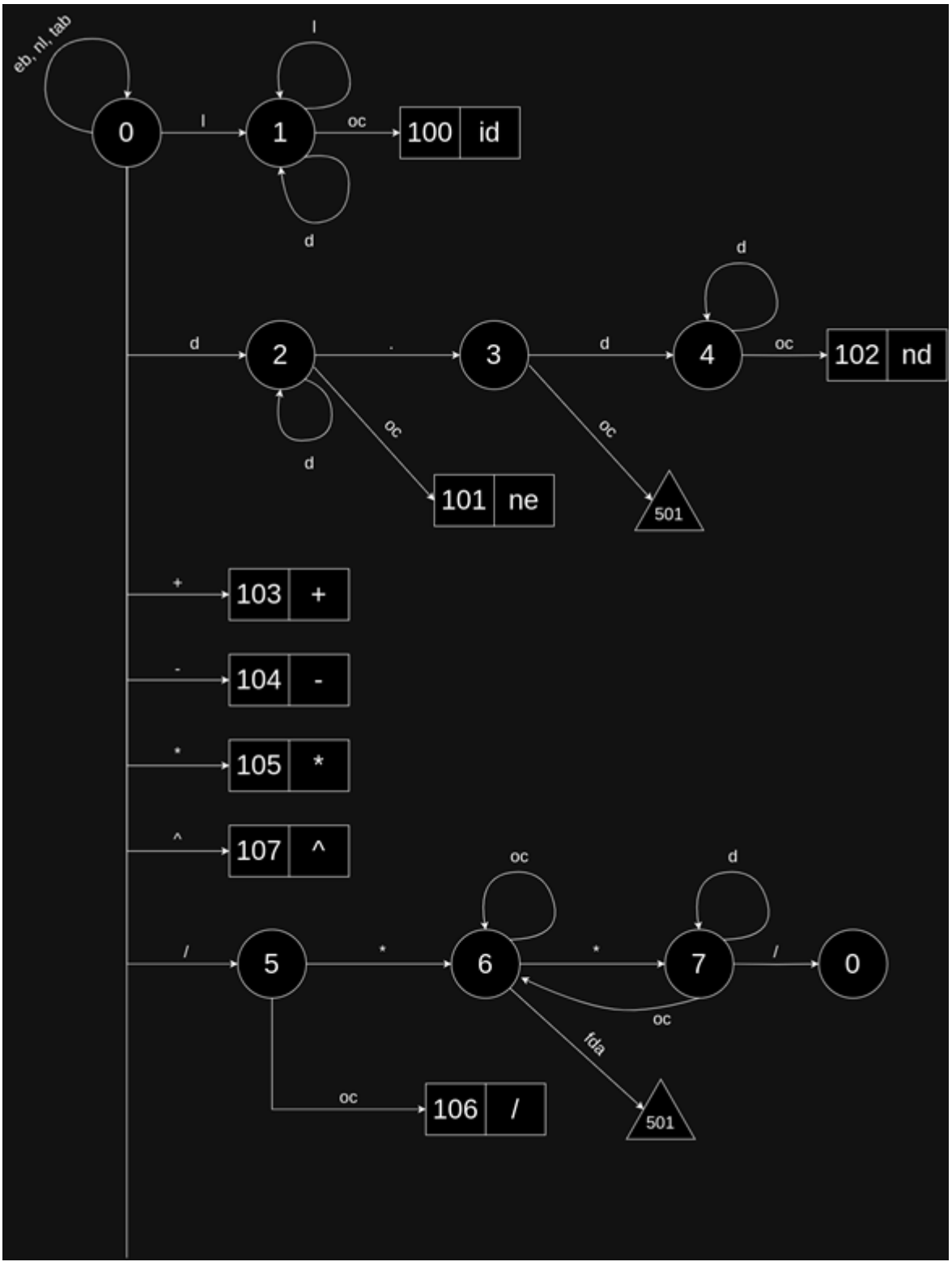
- Una buena lectura y análisis de la sintaxis del lenguaje Go para tomar los elementos necesarios para nuestro analizador léxico.
- Tener muy en claro las posibles palabras reservadas que ocupemos para el analizador, y los errores que pueden emerger de escribir mal un código.
- Haber completado de manera satisfactoria los puntos anteriores y tener muy en claro las posibles rutas o estados que tomará nuestro autómata finito.
- Completar satisfactoriamente el autómata finito, teniendo cuidado de no escribir incorrectamente los posibles estados finales que puede tomar el autómata finito.
- Tomar como base el código presente en el tutorial impartido por la profesora y adaptarlo a las necesidades del lenguaje del caso de estudio.
- Ver con detenimiento y entendimiento el tutorial impartido por la profesora para saber qué se está haciendo y por qué, además de adaptar ciertas partes del código para el lenguaje del caso de estudio.
- Tener el analizador léxico ya programado, habiendo comprobado con anterioridad que no falte nada importante del código.

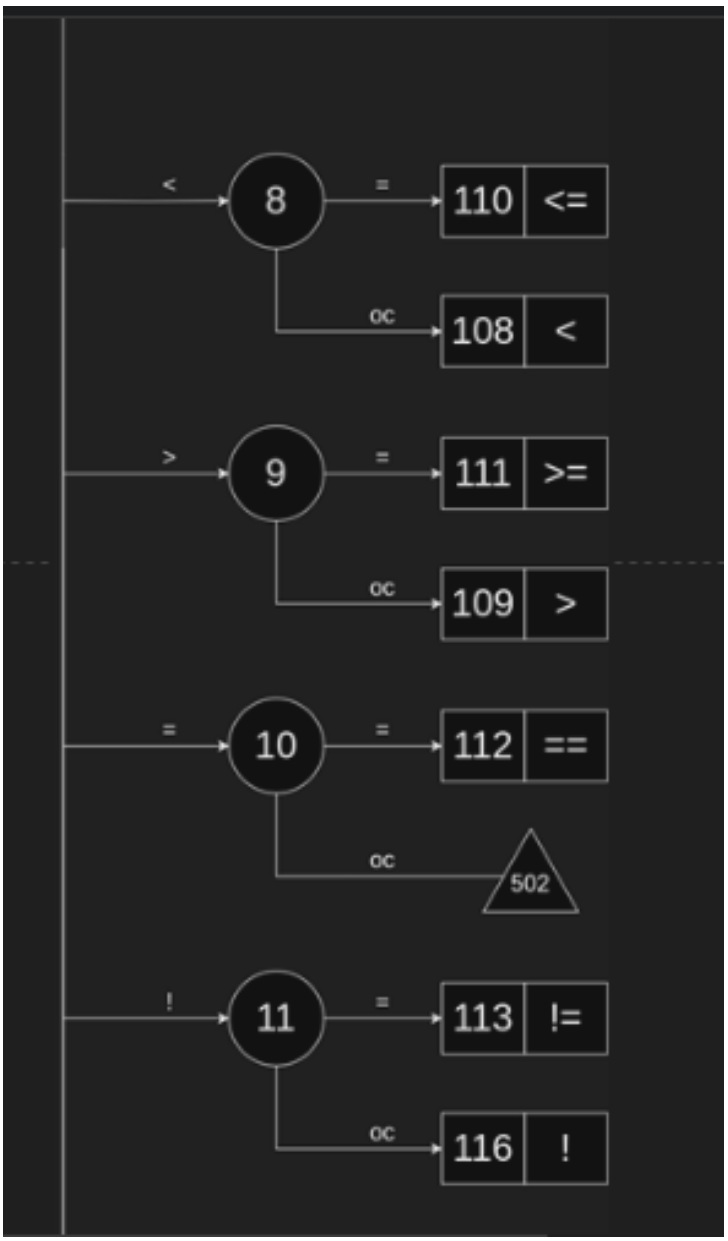
Materiales

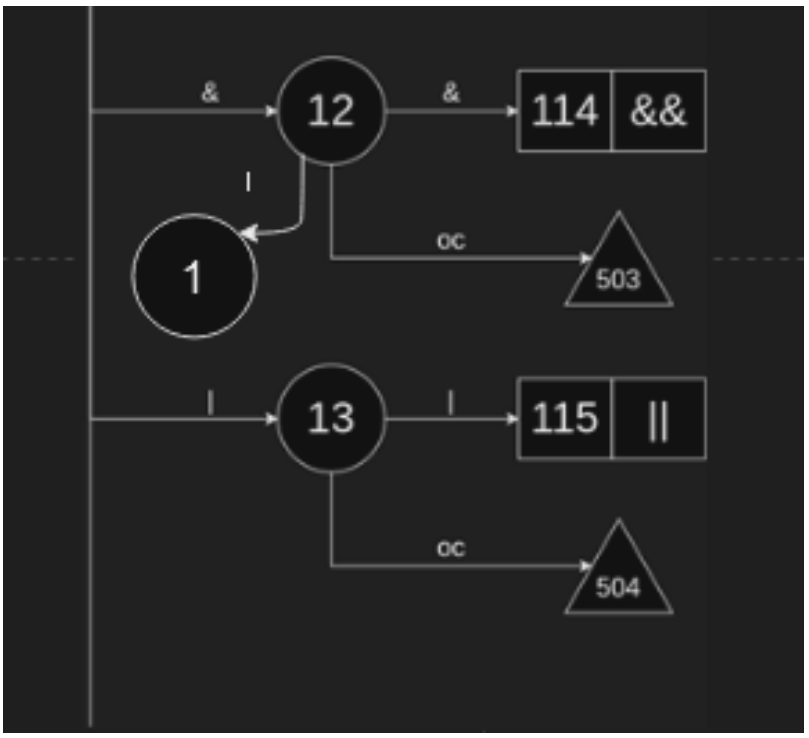
Materiales empleados para la exposición del análisis léxico	
Sistema(s) operativo(s)	Linux Mint 20.3 Cinnamon Windows 10
IDE	Apache NetBeans IDE 17
Lenguaje(s) de programación	Java
Creación de diagramas	Diagramas.net
Plataforma de reunión virtual	Google Meet Discord
Programa de grabación de video	OBS Studio
Programa de edición de video	Sony Vegas Pro 19

Desarrollo

Autómata finito




8
<
=
110
<=
oc
108
<
9
>
=
111
>=
oc
109
>
10
=
=
112
==
oc
502
11
!
=
113
!=
oc
116
!


&
12
&
114
&&
|
1
OC
503
|
13
|
115
||
OC
504

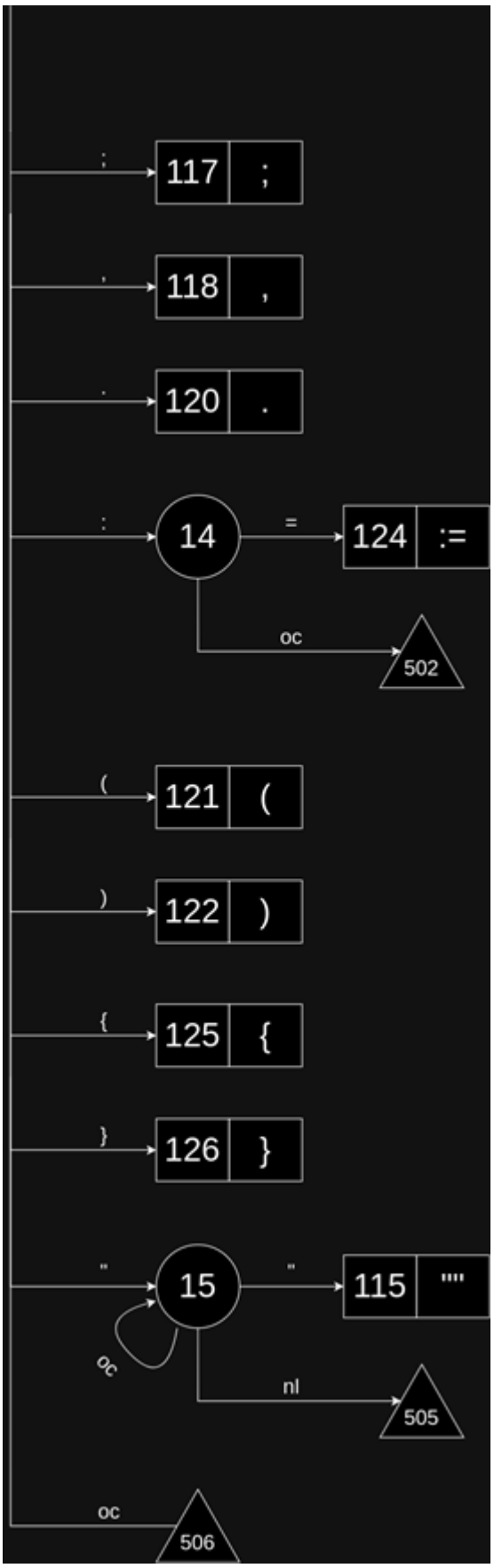
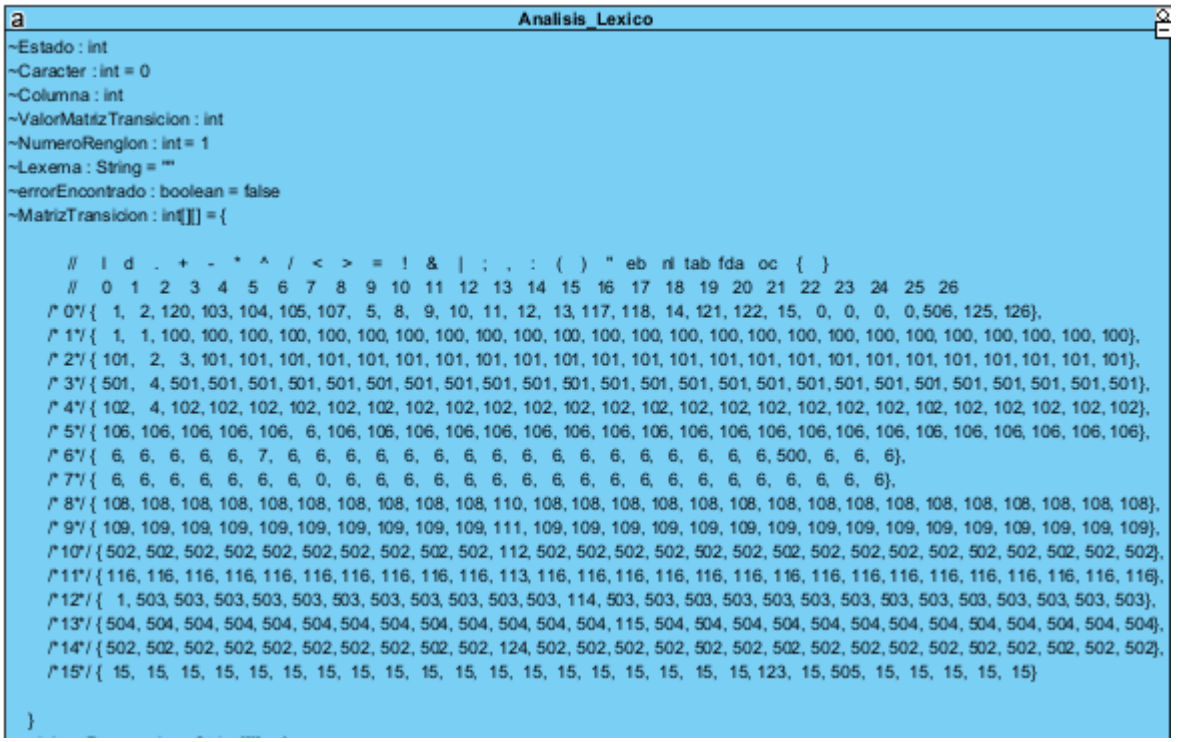

;
117
;
,
118
,
.
120
.
:
14
=
124
:=
oc
502
(
121
(
)
122
)
{
125
{
}
126
}
"
15
"
115
""
oc
nl
505
oc
506

Diagrama de clases



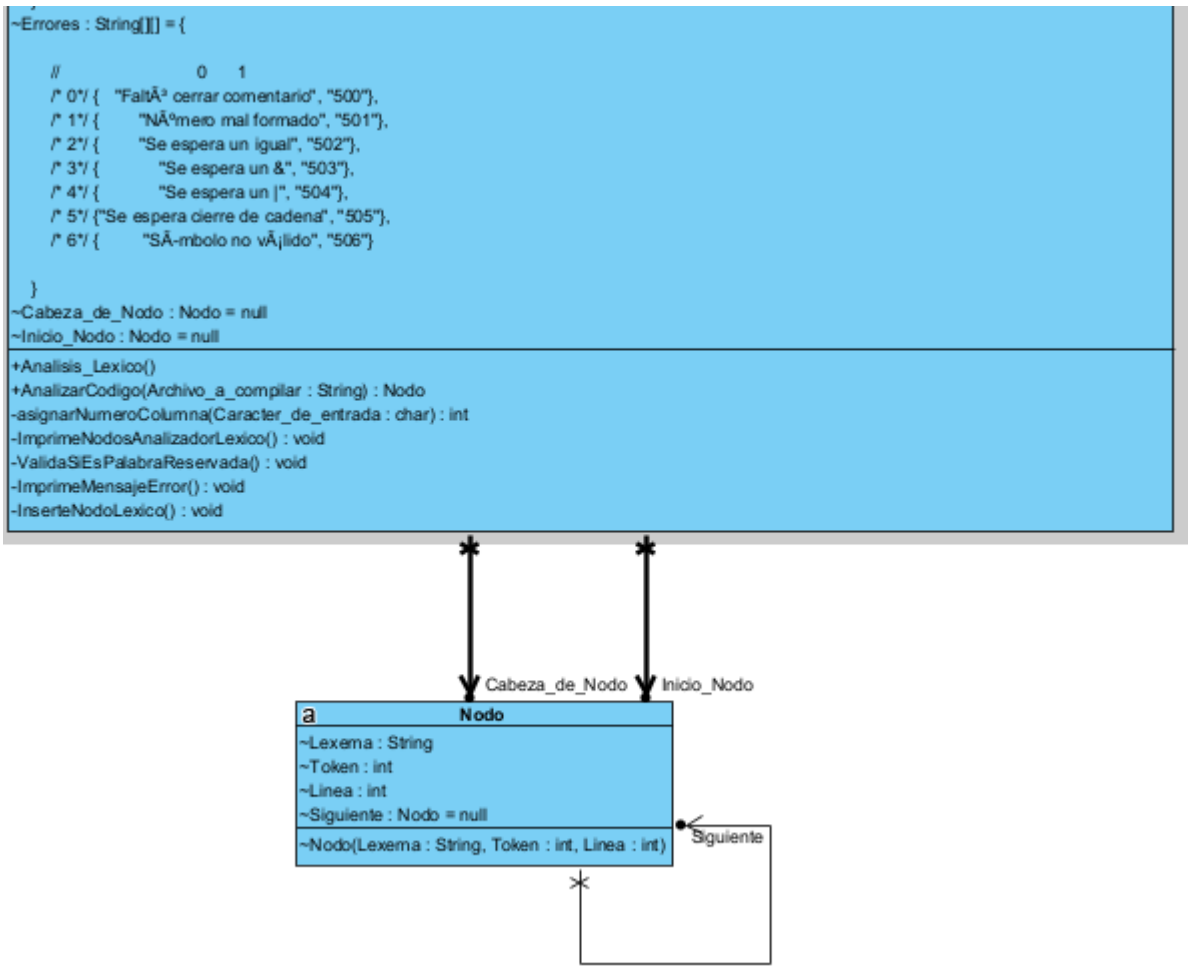
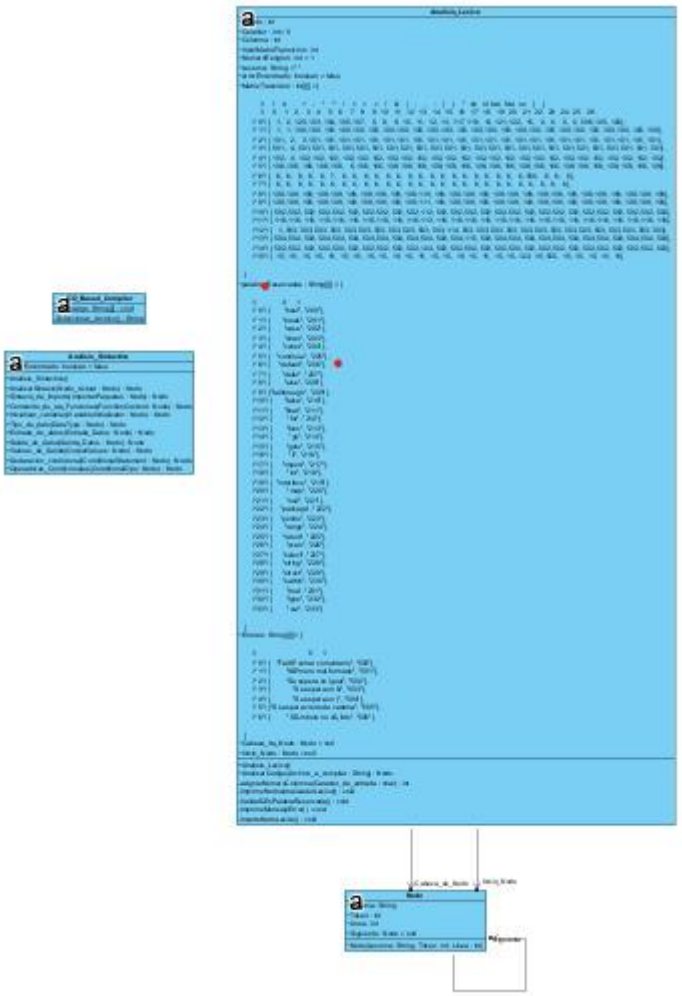

~Errores : String[][] = {
// 0 1
/* 0*/ { "FaltÃ³ cerrar comentario", "500"},
/* 1*/ { "NÃºmero mal formado", "501"},
/* 2*/ { "Se espera un igual", "502"},
/* 3*/ { "Se espera un &", "503"},
/* 4*/ { "Se espera un |", "504"},
/* 5*/ {"Se espera cierre de cadena", "505"},
/* 6*/ { "SÃ-mbolo no vÃ¡lido", "506"}
}
~Cabeza_de_Nodo : Nodo = null
~Inicio_Nodo : Nodo = null
+Analisis_Lexico()
+AnalizarCodigo(Archivo_a_compilar : String) : Nodo
-asignarNumeroColumna(Caracter_de_entrada : char) : int
-ImprimeNodosAnalizadorLexico() : void
-ValidaSiEsPalabraReservada() : void
-ImprimeMensajeError() : void
-InserteNodoLexico() : void
Cabeza_de_Nodo
Inicio_Nodo
Nodo
~Lexema : String
~Token : int
~Linea : int
~Siguiente : Nodo = null
~Nodo(Lexema : String, Token : int, Linea : int)
Siguiente
GO_Based_Compiler
+main(args : String[]) : void
-Seleccionar_Archivo() : String
Analisis_Sintactico
~errorEncontrado : boolean = false
+Analisis_Sintactico()
+AnalizarSintaxis(Nodo_Actual : Nodo) : Nodo
+Sintaxis_de_Imports(ImportarPaquetes : Nodo) : Nodo
+Contenido_de_las_Funciones(FunctionContent : Nodo) : Nodo
+Inicializar_variables(VariableInitializator : Nodo) : Nodo
+Tipo_de_dato(DataType : Nodo) : Nodo
+Entrada_de_datos(Entrada_Datos : Nodo) : Nodo
+Salida_de_datos(Salida_Datos : Nodo) : Nodo
+Valores_de_Salida(OutputValues : Nodo) : Nodo
+Declaracion_condicional(ConditionalStatement : Nodo) : Nodo
+Operadores_Condicionales(ConditionalOps : Nodo) : Nodo

Imagen completa (fue partida en partes porque no se alcanza a leer nada en su tamaño original)

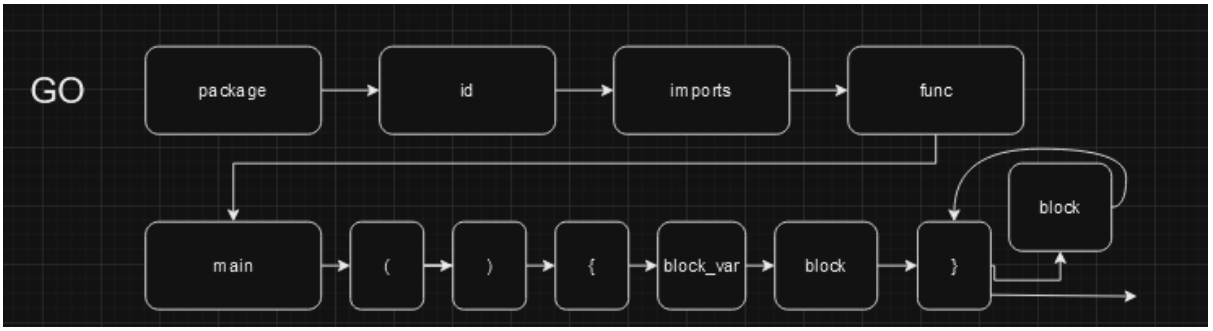


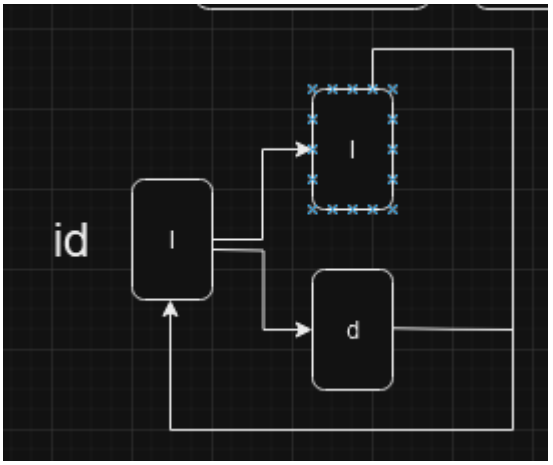
Matriz de transición

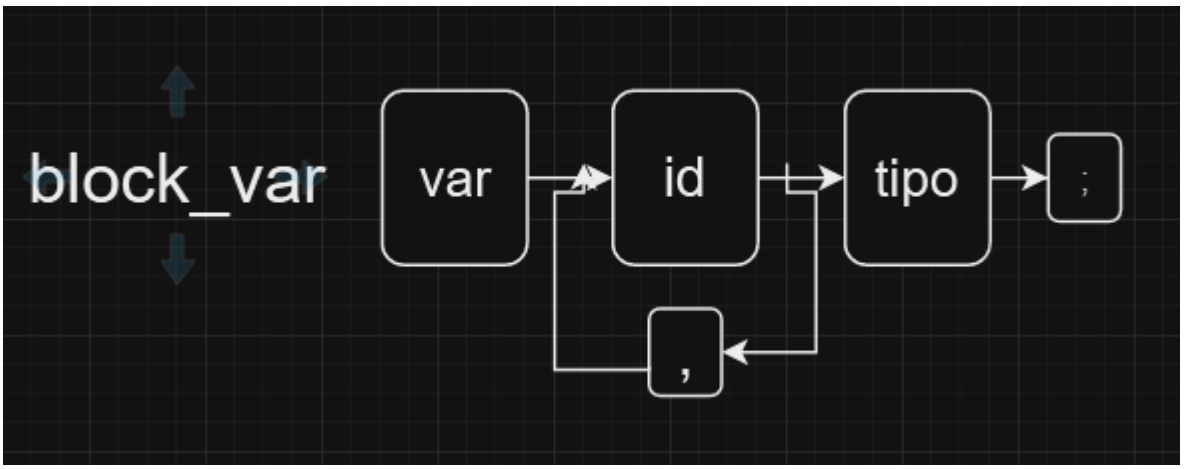
```
// Matriz de transición
int MatrizTransicion [][] = {

      //     l    d    .    +    -    *    ^    /    <    >    =    !    &    |    ;    ,    :    (    )    "   eb   nl  tab  fda   oc    {    }
      //     0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18   19   20   21   22   23   24   25   26
   /* 0*/ {   1,   2, 120, 103, 104, 105, 107,   5,   8,   9,  10,  11,  12,  13, 117, 118,  14, 121, 122,  15,   0,   0,   0,   0, 506, 125, 126},
   /* 1*/ {   1,   1, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100},
   /* 2*/ { 101,   2,   3, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101, 101},
   /* 3*/ { 501,   4, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501, 501},
   /* 4*/ { 102,   4, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102, 102},
   /* 5*/ { 106, 106, 106, 106, 106,   6, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106},
   /* 6*/ {   6,   6,   6,   6,   6,   7,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6, 500,   6,   6,   6,   6},
   /* 7*/ {   6,   6,   6,   6,   6,   6,   6,   0,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6,   6},
   /* 8*/ { 108, 108, 108, 108, 108, 108, 108, 108, 108, 108, 108, 110, 108, 108, 108, 108, 108, 108, 108, 108, 108, 108, 108, 108, 108, 108, 108},
   /* 9*/ { 109, 109, 109, 109, 109, 109, 109, 109, 109, 109, 111, 109, 109, 109, 109, 109, 109, 109, 109, 109, 109, 109, 109, 109, 109, 109, 109},
   /*10*/ { 502, 502, 502, 502, 502, 502, 502, 502, 502, 502, 112, 502, 502, 502, 502, 502, 502, 502, 502, 502, 502, 502, 502, 502, 502, 502, 502},
   /*11*/ { 116, 116, 116, 116, 116, 116, 116, 116, 116, 116, 113, 116, 116, 116, 116, 116, 116, 116, 116, 116, 116, 116, 116, 116, 116, 116, 116},
   /*12*/ {   1, 503, 503, 503, 503, 503, 503, 503, 503, 503, 503, 503, 114, 503, 503, 503, 503, 503, 503, 503, 503, 503, 503, 503, 503, 503, 503},
   /*13*/ { 504, 504, 504, 504, 504, 504, 504, 504, 504, 504, 504, 504, 504, 115, 504, 504, 504, 504, 504, 504, 504, 504, 504, 504, 504, 504, 504},
   /*14*/ { 502, 502, 502, 502, 502, 502, 502, 502, 502, 502, 124, 502, 502, 502, 502, 502, 502, 502, 502, 502, 502, 502, 502, 502, 502, 502, 502},
   /*15*/ {  15,  15,  15,  15,  15,  15,  15,  15,  15,  15,  15,  15,  15,  15,  15,  15,  15,  15,  15,  15, 123,  15, 505,  15,  15,  15,  15,  15}
};
```

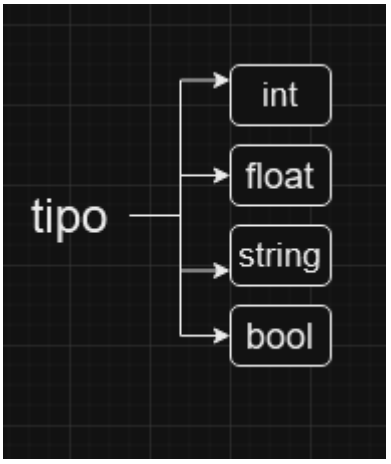
El propósito de esta matriz de transición fue principalmente para poder ubicar los Tokens y los Estados que el analizador léxico vaya encontrando en el código del lenguaje objetivo de programación.

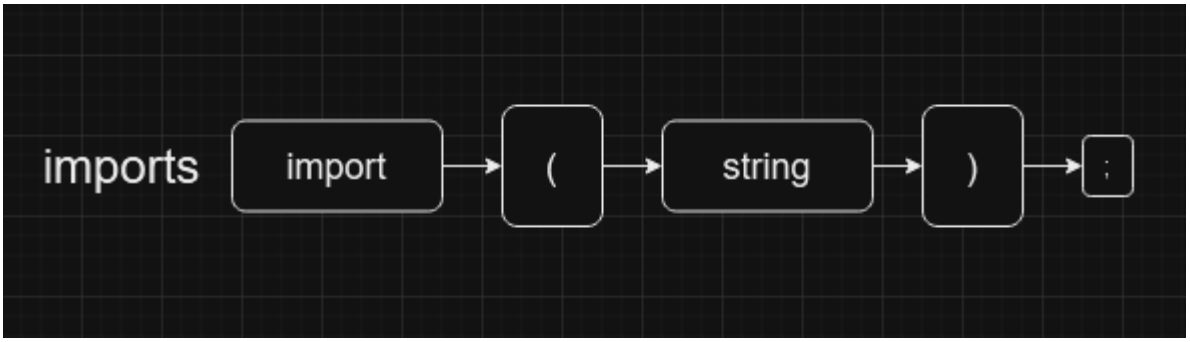
Diagrama de bloques

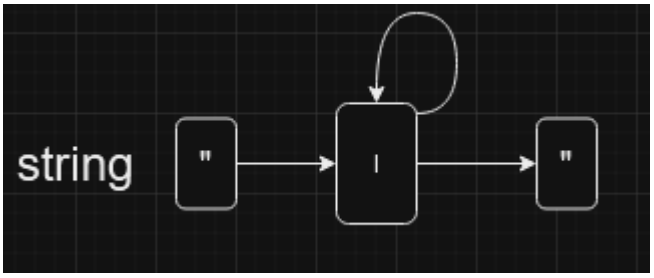


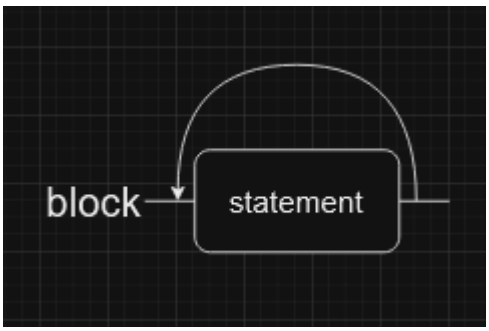


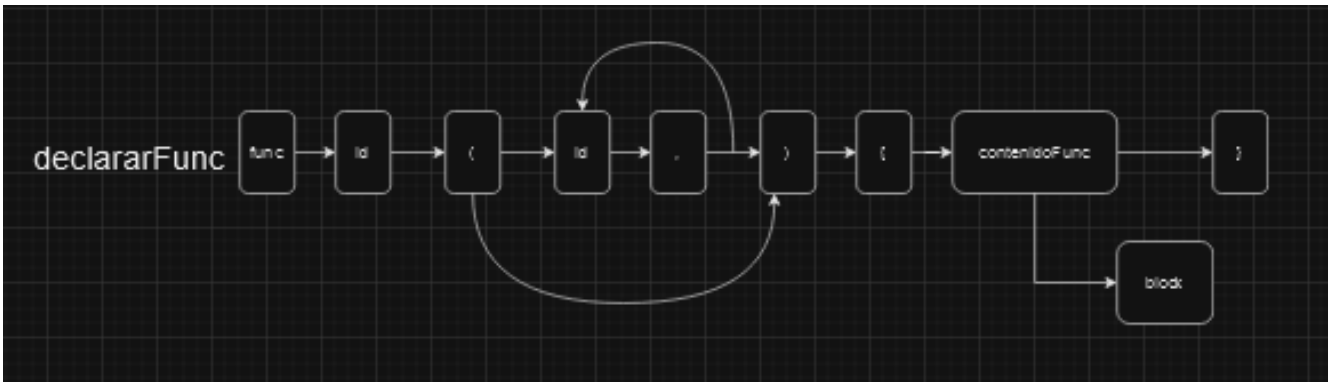

block_var
var
id
tipo
;
,

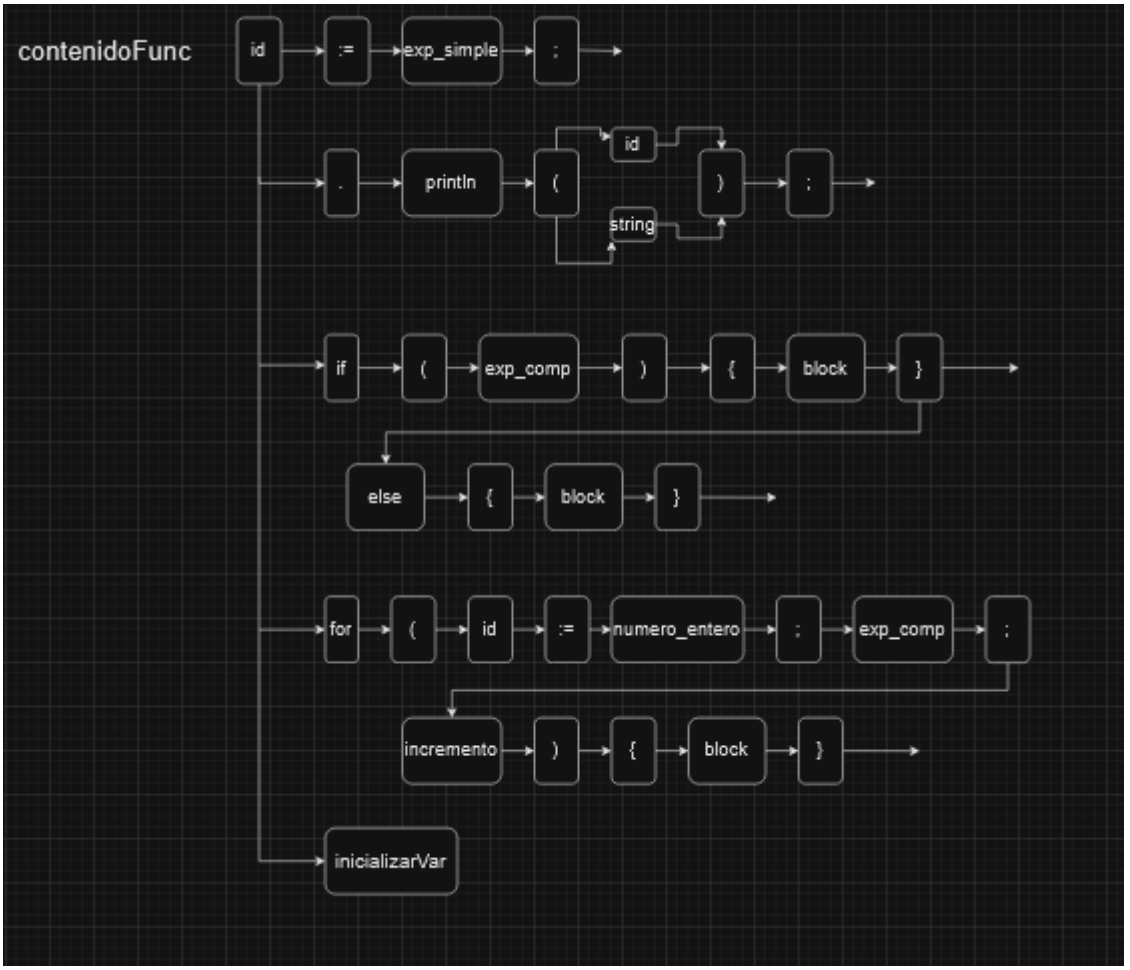

tipo
int
float
string
bool

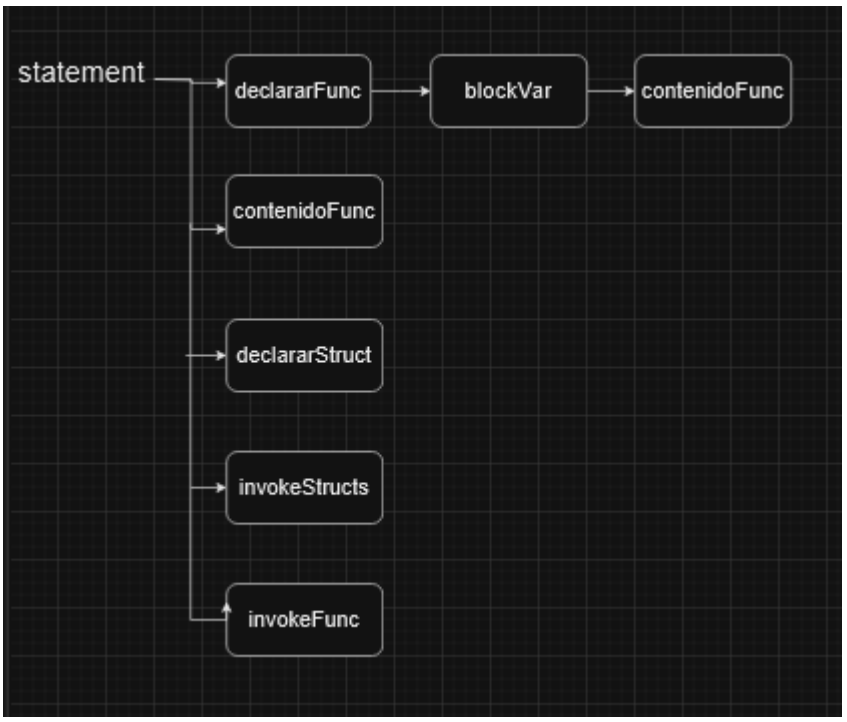

imports
import
(
string
)
;

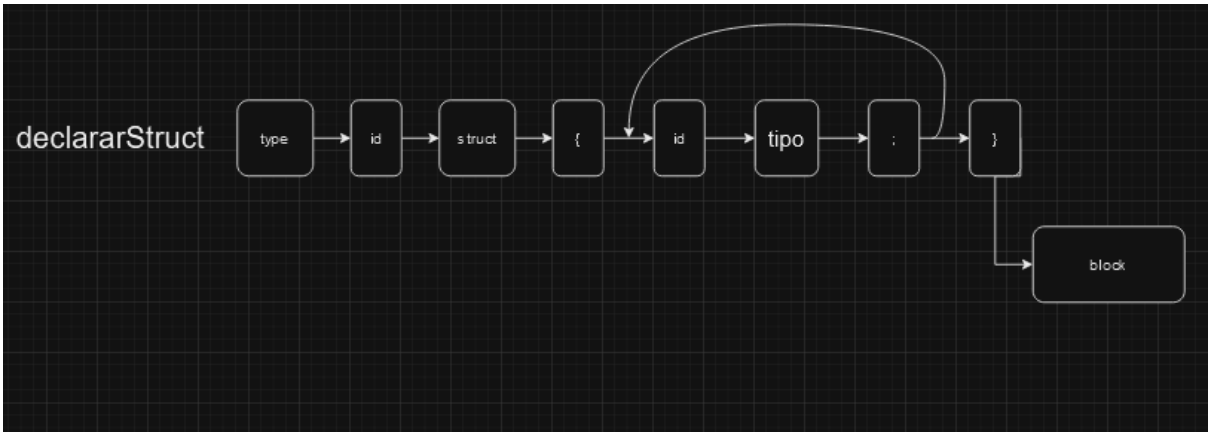

string
"
|
"

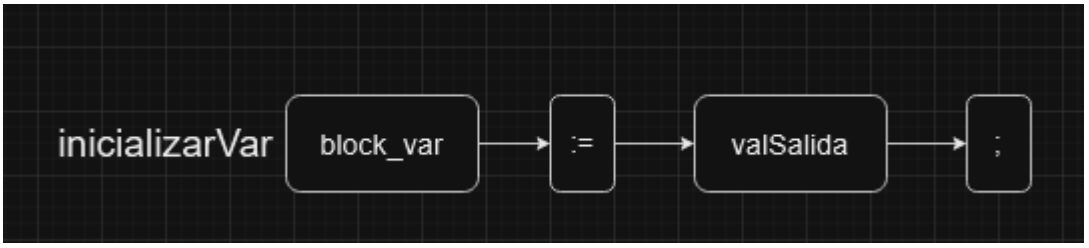

block
statement

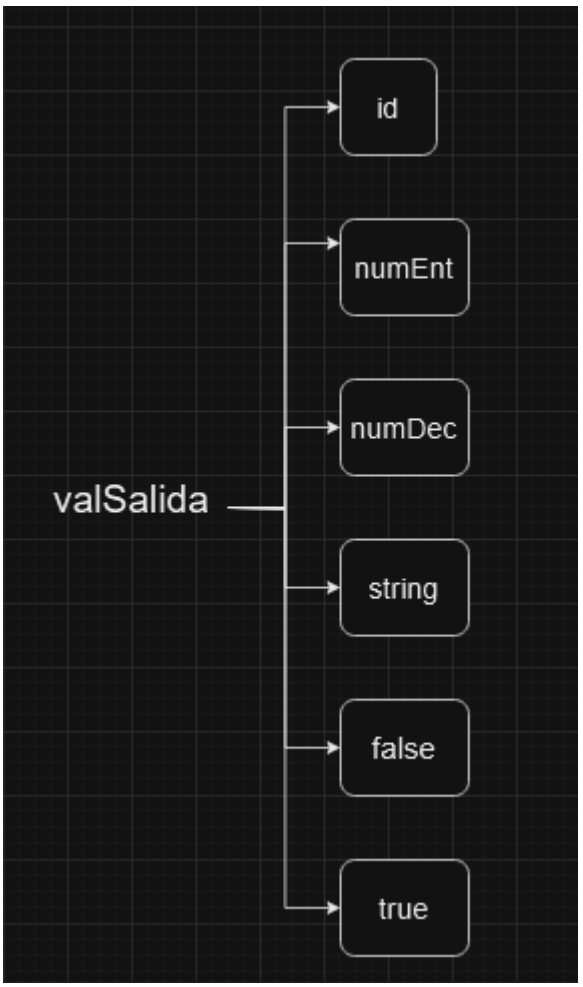

declararFunc
func
id
(
id
,
)
{
contenidoFunc
}
block

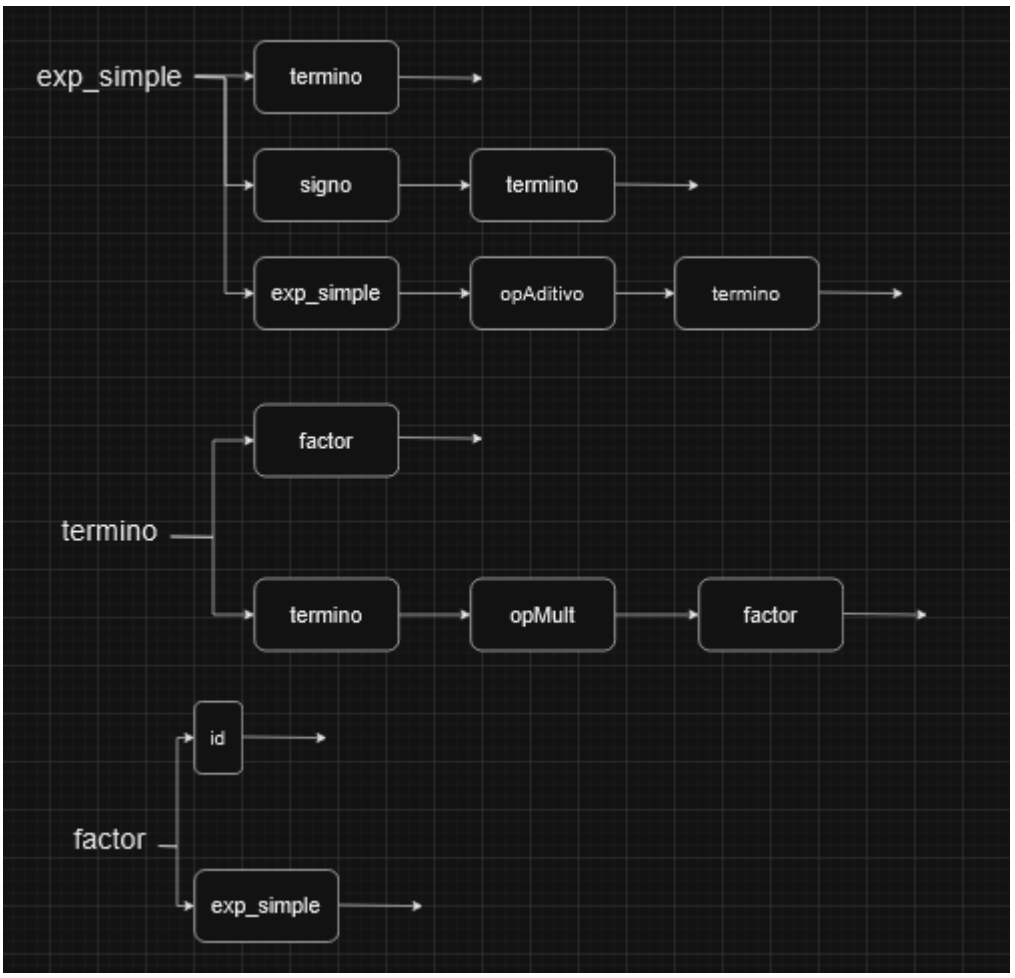

contenidoFunc
id
:=
exp_simple
;
.
println
(
id
string
)
;
if
(
exp_comp
)
{
block
}
else
{
block
}
for
(
id
:=
numero_entero
;
exp_comp
;
incremento
)
{
block
}
inicializarVar

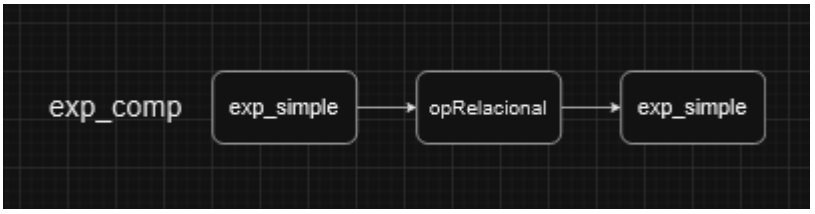

statement
declararFunc
blockVar
contenidoFunc
contenidoFunc
declararStruct
invokeStructs
invokeFunc

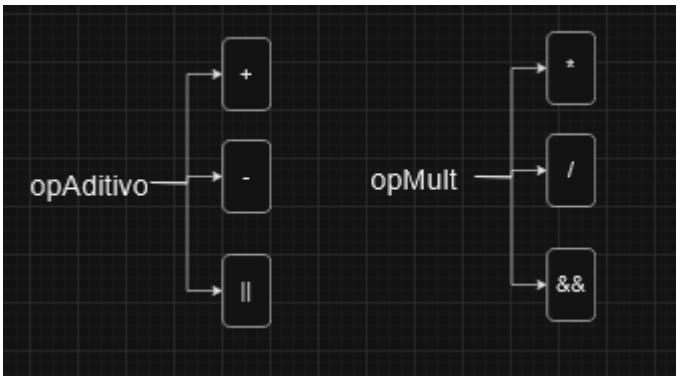

declararStruct
type
id
struct
{
id
tipo
;
}
block

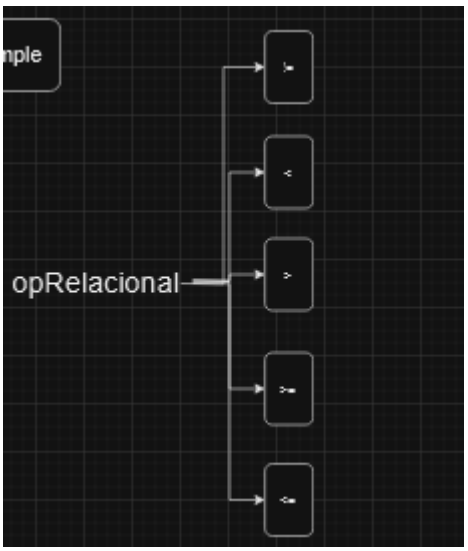

inicializarVar
block_var
:=
valSalida
;

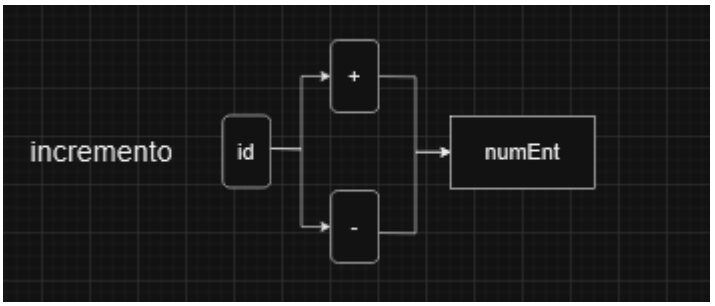

id
numEnt
numDec
valSalida
string
false
true


exp_simple
termino
signo
termino
exp_simple
opAditivo
termino
termino
factor
termino
termino
opMult
factor
factor
id
exp_simple


exp_comp
exp_simple
opRelacional
exp_simple


opAditivo
+
-
||
opMult
*
/
&&


nple
!=
<
>
>=
<=
opRelacional


incremento
id
+
-
numEnt

Código

Lo primero que hacemos en el archivo ***Analisis_Lexico.java*** es importar las librerías necesarias para algunos de los procesos que se hacen dentro del analizador léxico, la mayoría involucrando a la lectura del archivo de pruebas del compilador.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;
```

Lo siguiente es definir en la clase principal todas las variables que emplearemos para las distintas funciones que programemos (eso incluye la matriz de transición anteriormente mostrada).

```
Nodo Cabeza_de_Nodo = null; // Cabeza, es decir, final del nodo
Nodo Inicio_Nodo = null; // Inicio del nodo, utilidad para imprimir

int Estado, Caracter = 0; // Estado y Caracter
int Columna, ValorMatrizTransicion, NumeroRenglon = 1; // Columna, Valor de matriz de transición, y Número de renglón

String Lexema = ""; // Lexema

boolean errorEncontrado = false; // Error encontrado durante el análisis léxico
```

En esa misma clase definimos cuáles serán las palabras reservadas y errores léxicos que el compilador vaya encontrando en nuestro código para analizar.

```
// Palabras reservadas
String palabrasReservadas [][] = {

    //                0       1
    /* 0*/ {        "bool", "200"},
    /* 1*/ {       "break", "201"},
    /* 2*/ {        "case", "202"},
    /* 3*/ {        "chan", "203"},
    /* 4*/ {       "const", "204"},
    /* 5*/ {    "continue", "205"},
    /* 6*/ {     "default", "206"},
    /* 7*/ {       "defer", "207"},
    /* 8*/ {        "else", "208"},
    /* 9*/ {"fallthrough", "209"},
    /*10*/ {       "false", "210"},
    /*11*/ {       "float", "211"},
    /*12*/ {         "for", "212"},
    /*13*/ {        "func", "213"},
    /*14*/ {          "go", "214"},
    /*15*/ {        "goto", "215"},
    /*16*/ {          "if", "216"},
    /*17*/ {      "import", "217"},
    /*18*/ {         "int", "218"},
    /*19*/ {   "interface", "219"},
    /*20*/ {         "map", "220"},
    /*21*/ {        "null", "221"},
    /*22*/ {     "package", "222"},
    /*23*/ {     "println", "223"},
    /*24*/ {       "range", "224"},
    /*25*/ {      "return", "225"},
    /*26*/ {        "scan", "226"},
    /*27*/ {      "select", "227"},
    /*28*/ {      "string", "228"},
    /*29*/ {      "struct", "229"},
    /*30*/ {      "switch", "230"},
    /*31*/ {        "true", "231"},
    /*32*/ {        "type", "232"},
    /*33*/ {         "var", "233"}
};
```

```
// Errores
String Errores [][] = {

    //                                  0      1
    /* 0*/ {   "Faltó cerrar comentario", "500"},
    /* 1*/ {        "Número mal formado", "501"},
    /* 2*/ {        "Se espera un igual", "502"},
    /* 3*/ {            "Se espera un &", "503"},
    /* 4*/ {            "Se espera un |", "504"},
    /* 5*/ {"Se espera cierre de cadena", "505"},
    /* 6*/ {         "Símbolo no válido", "506"}
};
```

En esta primera parte de la función ***AnalizarLexico()***, se toma el archivo .txt elegido y lo transformamos a una lista de Strings, una por cada línea de código. Luego se unen todas esas listas para formar un único String para que el analizador léxico pueda leerlo fácilmente. Para finalizar con este proceso, se convierte todo en un arreglo de tamaño fijo donde cada elemento, incluyendo saltos de línea, tabulaciones y demás, es un caracter.

Empezamos también creando un ciclo For para que lea cada carácter del archivo de texto. El modo en que se leerá cada carácter será en su formato ASCII. Dentro del For se llamará a una función dentro de la clase llamada ***asignarNumeroColumna()***, esto con el fin de comenzar a darle uso a la matriz de transición y asignarle un valor a cada caracter que vaya detectando el analizador léxico en el archivo .txt.

```
try{
    // Aquí tomamos el .txt y lo convertimos en una lista de Strings, una por cada linea de código
    List<String> Codigo_a_Analizar = Files.readAllLines(path:Paths.get(first:Archivo_a_compilar), cs: StandardCharsets.UTF_8);
    // Aquí unimos todas las listas de Strings para que sea un solo String
    String String_del_codigo_a_analizar = String.join(delimiter:System.lineSeparator(), elements: Codigo_a_Analizar);
    // Aquí convertimos todo en un arreglo de tamaño fijo donde cada elemento es un caracter, incluyendo saltos de linea
    char[] Analizar_codigo = String_del_codigo_a_analizar.toCharArray();

    // Aquí recorremos el arreglo uno por uno, imposible leer mas caracteres de los que hay
    for(int Indice_del_codigo_a_analizar = 0; Indice_del_codigo_a_analizar < Analizar_codigo.length; Indice_del_codigo_a_analizar++){
        // Ya que caracter es un entero, aquí tomamos el elemento del indice correspondiente, y lo convertimos a su modo ASCII
        Caracter = (int)Analizar_codigo[Indice_del_codigo_a_analizar];

        // Aquí llamamos a la clase asignarNumeroColumna()
        Columna = asignarNumeroColumna(Analizar_codigo[Indice_del_codigo_a_analizar]);

        ValorMatrizTransicion = MatrizTransicion[Estado][Columna];
```

En esta segunda parte de la función ***AnalizarLexico()***, se analiza si el valor de matriz de transición es menor a 100. En caso de que lo sea, se cambia de estado.

```
if (ValorMatrizTransicion < 100){ // Cambiar de estado
        Estado = ValorMatrizTransicion;
        if (Estado == 0){
            Lexema = "";
        }
        else {
            Lexema = Lexema + Analizar_codigo[Indice_del_codigo_a_analizar];
        }
        if(Caracter == 10) {
            NumeroRenglon += 1;
        }
```

La tercera parte de la función ***AnalizarLexico()***, se empieza a analizar si el valor de la matriz de transición equivale a una palabra reservada, si se requiere retroceder una posición al apuntador para que el analizador léxico siga evaluando el código del archivo de texto sin errores, o si se detecta un error léxico en el archivo de texto. También se empiezan a insertar nodos.

En el área donde está ***Indice_del_codigo_a_analizar–;***, este valor se dará si los valores de la matriz de transición son el 100, 101, 102, 106, 108, 109, 116, 127, iguales o superiores a 200.

```
    } else if (ValorMatrizTransicion >= 100 && ValorMatrizTransicion < 500){ // Estado final

        if (ValorMatrizTransicion == 100){

            // Aquí llamamos a la clase ValidaSiEsPalabraReservada()
            ValidaSiEsPalabraReservada();

        }

        if (ValorMatrizTransicion == 100 || ValorMatrizTransicion == 101 || ValorMatrizTransicion == 102 || ValorMatrizTr

            Indice_del_codigo_a_analizar--; // Retrocede a una posición el apuntador

        }
        else {

            Lexema = Lexema + Analizar_codigo[Indice_del_codigo_a_analizar];
        }

        // Aquí llamamos la clase InserteNodoLexico()
        InserteNodoLexico();

        if(Caracter == 10) {

            System.out.println(x: "Salto de linea");
            NumeroRenglon += 1;

        }

        Estado = 0;
        Lexema = "";

    } else { // Estado de error
        //Aquí llamamos a la clase ImprimeMensajeError()
        ImprimeMensajeError();

        // Se cierra el análisis léxico del código tan pronto como encuentre un error, dejando a las partes restantes sin
        System.exit(status:0);
    }
}
```

En la última parte de la función AnalizarLexico(), en caso de que no se hayan detectado errores léxicos, se llama a la función ***imprimeNodosAnalizadorLexico()*** para que pueda mostrarnos todos los nodos detectados en el archivo de texto a través de la terminal. En caso de que se detecte algún error no especificado, se detiene la compilación del proyecto.

Para ambos casos, se retorna Inicio_Nodo, el cual regresa el apuntador de los nodos al inicio del archivo de prueba.

```
        }

        //Aquí llamamos a la clase ImprimeNodosAnalizadorLexico()
        ImprimeNodosAnalizadorLexico();
        return Inicio_Nodo;

    } catch (IOException e) {

        System.out.println(x:e.getMessage());
        return Inicio_Nodo;

    }
```

En la función ***asignarNumeroColumna()***, se determina el número de la columna de la matriz de transición para comenzar la búsqueda del valor del Token asociado al caracter analizado.

```
// Aquí asignamos un número de columna a analizar para que podamos encontrar un valor de la matriz de transición según el caracter analizado
private int asignarNumeroColumna(char Caracter_de_entrada){

    if(Character.isLetter(ch: Caracter_de_entrada)) return 0;

    if(Character.isDigit(ch: Caracter_de_entrada)) return 1;

        switch(Caracter_de_entrada){
            case '.': return 2;

            case '+': return 3;

            case '-': return 4;

            case '*': return 5;

            case '^': return 6;

            case '/': return 7;

            case '<': return 8;

            case '>': return 9;

            case '=': return 10;

            case '!': return 11;

            case '&': return 12;

            case '|': return 13;

            case ';': return 14;

            case ',': return 15;

            case ':': return 16;

            case '(': return 17;

            case ')': return 18;

            case '"': return 19;

            case ' ': return 20;

            case '{': return 25;

            case '}': return 26;

            case 10: return 21;

            case 9: return 22;

            case 3: return 23;

            default: return 24;

        }
}
```

En la función ***inserteNodoLexico()***, se llama a la clase ***Nodo*** para que se empiecen a asociar los Tokens encontrados en el código dentro del archivo de texto con el lexema y su posición en la fila del código analizado.

```
private void InserteNodoLexico(){

    // Aquí llamamos a la clase Nodo
    Nodo Node = new Nodo(Lexema, Token: ValorMatrizTransicion, Linea: NumeroRenglon);

    if (Cabeza_de_Nodo == null){

        Inicio_Nodo = Cabeza_de_Nodo = Node;

    }
    else {

        Cabeza_de_Nodo.Siguiente = Node;
        Cabeza_de_Nodo = Node;
    }
}
```

En la función ***imprimeNodosAnalizadorLexico()***, nuevamente se hace el llamado a la clase ***Nodo*** para que se impriman los valores del Token, Lexema y Número de Renglón del código analizado.

```
private void ImprimeNodosAnalizadorLexico(){

    Nodo Nodo_Actual = Inicio_Nodo;

    while(Nodo_Actual != null){

        System.out.println("Lexema: " + Nodo_Actual.Lexema + " | Token: " + Nodo_Actual.Token + " | Linea: " + Nodo_Actual.Linea);
        Nodo_Actual = Nodo_Actual.Siguiente;

    }
}
```

En la función ***ValidaSiEsPalabraReservada()***, se asigna una palabra reservada en el Token y Lexema de los valores del código analizado.

```
private void ValidaSiEsPalabraReservada(){
    for(String[] palabraReservada: palabrasReservadas){

        if(Lexema.equals(palabraReservada[0])){

            ValorMatrizTransicion = Integer.parseInt(palabraReservada[1]);

        }

    }
}
```

En la función **ImprimeMensajeError()**, se detecta el tipo de error encontrado durante el análisis léxico del código dentro del archivo de texto, detiene de forma abrupta el análisis léxico independientemente del avance obtenido, e imprime un mensaje en la terminal indicando el tipo de error encontrado.

```
private void ImprimeMensajeError(){
    if(Caracter != 1 && ValorMatrizTransicion >= 500){

        for(String[] Error: Errores){

            if(ValorMatrizTransicion == Integer.parseInt(Error[1])){

                System.out.println("El error encontrado es: " + Error[0] + " error " + Error[1] + " en caracter " + (char)Caracter + " en el renglón " + (NumeroRenglon - 1));

            }
        }

        errorEncontrado = true;
    }
}
```

La función ***Analisis_Lexico()*** se mantiene vacía.

```
public Analisis_Lexico(){

}
```

1 En la clase Sintáctico se inicia desde El Nodo Analizar Sintaxis donde se toman si las palabras están dentro del .txt, dentro del cierre del nodo se importa todos los valores que se asemejen dentro del nodo sintaxis_imports.

```
public Nodo AnalizarSintaxis(Nodo Nodo_Actual){
    while(Nodo_Actual.Siguiente != null){

        if(Nodo_Actual.Token  == 222){ // Token de la palabra reservada 'package'

            Nodo_Actual = Nodo_Actual.Siguiente;

            if(Nodo_Actual.Token  == 100){ // Token para identificadores

                Nodo_Actual = Nodo_Actual.Siguiente;

                if (Nodo_Actual.Token  == 117){ // Token del ;

                    Nodo_Actual = Sintaxis_de_Imports(Nodo_Actual);
                    Nodo_Actual = Nodo_Actual.Siguiente;

                    if(Nodo_Actual.Token  == 213){ // Token de la palabra reservada 'func'

                        Nodo_Actual = Nodo_Actual.Siguiente;

                        if(Nodo_Actual.Token  == 100){ // Token para identificadores

                            Nodo_Actual = Nodo_Actual.Siguiente;

                            if(Nodo_Actual.Token  == 121){ // Token para inicio de paréntesis

                                Nodo_Actual = Nodo_Actual.Siguiente;

                                if(Nodo_Actual.Token  == 122){ // Token para cierre de paréntesis
```

- si en el caso contrario no encuentra la similitud del token actual con el número de matriz correspondiente se tomará como error.

```
if(Nodo_Actual.Token  == 122){ // Token para cierre de paréntesis

    Nodo_Actual = Nodo_Actual.Siguiente;

    if(Nodo_Actual.Token  == 125){ // Token para inicio de llaves

        Nodo_Actual = Contenido_de_las_Funciones(Nodo_Actual);

        if(Nodo_Actual.Token  == 126){ // Token para cierre de llaves

            if(Nodo_Actual.Siguiente != null){ // Tokens de la palabra reservada 'func' y 'type'

                Nodo_Actual = Declaracion_de_Funciones_y_Estructuras(Nodo_Actual);
                break;

            } else {

                System.out.println("\nAnálisis sintáctico terminado con éxito.");
                break;
            }
        } else{

            System.out.println("Se espera un cierre de llaves en la línea " + Nodo_Actual.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
```

```
                } else {

                    System.out.println("Se espera un inicio de llaves en la línea " + Nodo_Actual.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
            } else {

                System.out.println("Se espera un cierre de paréntesis en la línea " + Nodo_Actual.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        } else {

            System.out.println("Se espera un inicio de paréntesis en la línea " + Nodo_Actual.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    } else {

        System.out.println("Se espera un identificador en la línea " + Nodo_Actual.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
} else {

    System.out.println("Se espera la palabra 'func' en la línea " + Nodo_Actual.Linea);
    errorEncontrado = true;
    System.exit(0);
```

```
                } else {

                    System.out.println("Se espera un ; en la línea " + Nodo_Actual.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
            } else {

                System.out.println("Se espera un identificador en la línea " + Nodo_Actual.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        } else {

            System.out.println("Se espera la palabra 'package' en la línea " + Nodo_Actual.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    }

    return Nodo_Actual;
}
```

2 dentro del objeto nodo se realiza las semejanzas con los tokens que vienen en la línea que tiene el txt.

```
public Nodo Sintaxis_de_Imports (Nodo ImportarPaquetes) {
    while(ImportarPaquetes.Siguiente.Token == 217){ // Token de la palabra reservada 'import'

        ImportarPaquetes = ImportarPaquetes.Siguiente;

        if(ImportarPaquetes.Siguiente.Token == 121){ // Token de inicio de paréntesis

            ImportarPaquetes = ImportarPaquetes.Siguiente;

            if(ImportarPaquetes.Siguiente.Token == 123){ // Token de strings

                ImportarPaquetes = ImportarPaquetes.Siguiente;

                if(ImportarPaquetes.Siguiente.Token == 122){ // Token de cierre de paréntesis

                    ImportarPaquetes = ImportarPaquetes.Siguiente;

                    if(ImportarPaquetes.Siguiente.Token == 117){ // Token del ;

                        ImportarPaquetes = ImportarPaquetes.Siguiente;
```

```
                    } else {

                        System.out.println("Se espera un ; en la línea " + ImportarPaquetes.Linea);
                        errorEncontrado = true;
                        System.exit(0);
                    }
                } else {

                    System.out.println("Se espera un cierre de paréntesis en la línea " + ImportarPaquetes.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
            } else {

                System.out.println("Se espera una cadena de caracteres en la línea " + ImportarPaquetes.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        } else {

            System.out.println("Se espera un inicio de paréntesis en la línea " + ImportarPaquetes.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    }

    return ImportarPaquetes;
}
```

3 objeto nodo en el que toma el contenido de las funciones del cual toma la función semejante si esta llega a estar fuera de lugar.

```
public Nodo Contenido_de_las_Funciones(Nodo FunctionContent){

    FunctionContent = FunctionContent.Siguiente;

    while(FunctionContent != null && FunctionContent.Siguiente != null && FunctionContent.Siguiente.Token != 126){

        switch(FunctionContent.Token){
            case 100: // Token del identificador

                FunctionContent = Uso_de_Identificadores(FunctionContent);
                break;

            case 126: // Token de cierre de llaves

                return FunctionContent;

            case 212: // Token de la palabra reservada 'for

                FunctionContent = Declaracion_de_Bucle(FunctionContent);
                break;

            case 216: // Token de la palabra reservada 'if'

                FunctionContent = Declaracion_condicional(FunctionContent);
                break;

            case 233: // Token de la palabra reservada 'var'

                FunctionContent = Inicializar_variables(FunctionContent);
                break;

            default:

                FunctionContent = null;
                System.out.println("Error inesperado en la línea " + FunctionContent.Linea);
                errorEncontrado = true;
                System.exit(0);
        }
    }

    return FunctionContent;
}
```

4 toma el final de la variable si esta no llegará a final correctamente

```
public Nodo Inicializar_variables(Nodo VariableInitializator) {
    while(VariableInitializator.Siguiente.Token != 126){ // Token de cierre de llaves

        VariableInitializator = VariableInitializator.Siguiente;

        if(VariableInitializator.Token == 100){ // Token para identificadores

            VariableInitializator = Tipo_de_dato(VariableInitializator);
            VariableInitializator = VariableInitializator.Siguiente;

            if(VariableInitializator.Token == 124){ // Token del :=

                VariableInitializator = Valores_de_Salida(VariableInitializator);
                VariableInitializator = VariableInitializator.Siguiente;

                if(VariableInitializator.Token == 117){ // Token del ;

                    VariableInitializator = VariableInitializator.Siguiente;
                    break;

                } else {

                    System.out.println("Se espera un ; en la línea " + VariableInitializator.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
            }

            } else {

                System.out.println("Se espera un ; en la línea " + VariableInitializator.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        }

        if(VariableInitializator.Token == 117){ // Token del ;

            VariableInitializator = VariableInitializator.Siguiente;
            break;

        } else {

            System.out.println("Se espera un := o un ; en la línea " + VariableInitializator.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    } else {

        System.out.println("Se espera un identificador en la línea " + VariableInitializator.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
}
```

5 comprueba si alguna variable principal se inicializó correctamente

```
public Nodo Tipo_de_dato(Nodo DataType){

    DataType = DataType.Siguiente;

    while(DataType != null && DataType.Siguiente.Token != 124 && DataType.Siguiente.Token != 117){ // Tokens del := y del ;

        switch(DataType.Token){
            case 218: // Token de la palabra reservada 'int'

                return DataType;

            case 211: // Token de la palabra reservada 'float'

                return DataType;

            case 228: // Token de la palabra reservada 'string'

                return DataType;

             case 200: // Token de la palabra reservada 'bool'

                return DataType;

            default:

                DataType = null;
                System.out.println("Se espera una asignación de tipo en la línea " + DataType.Linea);
                errorEncontrado = true;
                System.exit(0);
        }
```

6 Objeto nodo en el que los datos se verifican

```
public Nodo Entrada_de_datos(Nodo Entrada_Datos) {
    while(Entrada_Datos.Siguiente.Token != 126){ // Token de cierre de llaves

        Entrada_Datos = Entrada_Datos.Siguiente;

        if(Entrada_Datos.Token == 121){ // Token para el inicio de paréntesis

            Entrada_Datos = Entrada_Datos.Siguiente;

            if(Entrada_Datos.Token == 100){ // Token de los identificadores y los strings

                Entrada_Datos = Entrada_Datos.Siguiente;

                if(Entrada_Datos.Token == 122){ // Token para el cierre de paréntesis

                    Entrada_Datos = Entrada_Datos.Siguiente;

                    if(Entrada_Datos.Token  == 117){ // Token del ;

                        Entrada_Datos = Entrada_Datos.Siguiente;
                        break;

                    } else {

                        System.out.println("Se espera un ; en la línea " + Entrada_Datos.Linea);
                        errorEncontrado = true;
                        System.exit(0);
                    }

                } else {

                    System.out.println("Se espera un ; en la línea " + Entrada_Datos.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
            } else {

                System.out.println("Se espera un cierre de paréntesis en la línea " + Entrada_Datos.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        } else {

            System.out.println("Se espera un identificador en la línea " + Entrada_Datos.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    } else {

        System.out.println("Se espera un inicio de paréntesis en la línea " + Entrada_Datos.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
}

return Entrada_Datos;
}
```

7 Objeto en el que terminan de verificarse las parte de abertura y cierre de variables que a la vez se imprimen al validarse, en el cual continúa con la salida de datos.

```
public Nodo Salida_de_datos(Nodo Salida_Datos) {
    while(Salida_Datos.Siguiente.Token != 126){ // Token de cierre de llaves

        Salida_Datos = Salida_Datos.Siguiente;
        System.out.println(Salida_Datos.Token);

        if(Salida_Datos.Token == 121){ // Topen de inicio de paréntesis

            if(Salida_Datos.Siguiente.Token == 100 || Salida_Datos.Siguiente.Token == 123){

                Salida_Datos = Valores_de_Salida(Salida_Datos);
                System.out.println(Salida_Datos.Token);
                Salida_Datos = Salida_Datos.Siguiente;
                System.out.println(Salida_Datos.Token);

            } else {

                Salida_Datos = Salida_Datos.Siguiente;
            }

            if(Salida_Datos.Token  == 122){ // Token para cierre de paréntesis

                Salida_Datos = Salida_Datos.Siguiente;
                System.out.println(Salida_Datos.Token);

                if(Salida_Datos.Token  == 117){ // Token del ;

                    Salida_Datos = Salida_Datos.Siguiente;
                    System.out.println(Salida_Datos.Token);
                    break;

                } else {

                    System.out.println("Se espera un ; en la línea " + Salida_Datos.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
            } else {

                System.out.println("Se espera un cierre de paréntesis en la línea " + Salida_Datos.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        } else {

            System.out.println("Se espera un inicio de paréntesis en la línea " + Salida_Datos.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    }

    return Salida_Datos;
}
```

8 se obtienen los valores de los tokens para que se realice la impresión de estos mismos.

```
public Nodo Valores_de_Salida(Nodo OutputValues){

    OutputValues = OutputValues.Siguiente;

    while(OutputValues != null && OutputValues.Siguiente.Token != 122){ // Token del cierre de paréntesis

        switch(OutputValues.Token){
            case 100: // Token de los identificadores

                return OutputValues;

            case 101: // Token de los números enteros

                return OutputValues;

            case 102: // Token de los números decimales

                return OutputValues;

            case 123: // Token de los strings

                return OutputValues;

            case 210: // Token del valor booleano 'false'

                return OutputValues;

            case 231: // Token del valor booleano 'true'

                return OutputValues;

            default:

                OutputValues = null;
                System.out.println("Se espera un valor válido en la línea " + OutputValues.Linea);
                errorEncontrado = true;
                System.exit(0);
        }
    }

    return OutputValues;
}
```

9 en este objeto nodo se realizan la mayoría de condiciones específicas que requiere un código en lenguaje go, por lo cual toma cada pasa de inicio a fin en el que debe realizarse al momento que leer el orden de como esta puesto el .txt para tener en cuenta que todo esté en orden.

```
public Nodo Declaracion_condicional(Nodo ConditionalStatement){
    while(ConditionalStatement.Siguiente.Token != 126){ // Token de la palabra reservada 'if'

        ConditionalStatement = ConditionalStatement.Siguiente;

        if(ConditionalStatement.Token == 121){ // Token de inicio de paréntesis

            if(ConditionalStatement.Siguiente.Token == 116){ // Token de !

                ConditionalStatement = ConditionalStatement.Siguiente;

            }

            ConditionalStatement = Valores_de_Salida(ConditionalStatement);

            // Parte de la declaración condicional sin operador condicional
            if(ConditionalStatement.Siguiente.Token == 122){ // Token de cierre de paréntesis

                ConditionalStatement = ConditionalStatement.Siguiente;

                if(ConditionalStatement.Siguiente.Token == 125){ // Token de inicio de llaves

                    ConditionalStatement = ConditionalStatement.Siguiente;
                    ConditionalStatement = Contenido_de_las_Funciones(ConditionalStatement);
```

```
if(ConditionalStatement.Token == 126){ // Token de cierre de llaves

    ConditionalStatement = ConditionalStatement.Siguiente;

    if(ConditionalStatement.Token == 208){ // Token de la palabra reservada 'else'

        ConditionalStatement = ConditionalStatement.Siguiente;

        if(ConditionalStatement.Token == 125){ // Token de inicio de llaves

            ConditionalStatement = Contenido_de_las_Funciones(ConditionalStatement);

            if(ConditionalStatement.Token == 126){ // Token de cierre de llaves

                ConditionalStatement = ConditionalStatement.Siguiente;
                break;

            } else {

                System.out.println("Se espera un cierre de llaves en la linea " + ConditionalStatement.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
```

```
                } else {

                    System.out.println("Se espera un cierre de llaves en la línea " + ConditionalStatement.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
            } else {

                System.out.println("Se espera un inicio de llaves en la línea " + ConditionalStatement.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        } else {

            break;
        }
    } else {

        System.out.println("Se espera un cierre de llaves en la línea " + ConditionalStatement.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
} else {

    System.out.println("Se espera un inicio de llaves en la línea " + ConditionalStatement.Linea);
    errorEncontrado = true;
    System.exit(0);
}
```

- aquí se continua con los testamento de condiciones

```
// Parte de la declaración condicional con operador condicional
} else {

    ConditionalStatement = Operadores_Condicionales_y_Logicos(ConditionalStatement);
    ConditionalStatement = Valores_de_Salida(ConditionalStatement);

    if(ConditionalStatement.Siguiente.Token == 122){ // Token de cierre de paréntesis

        ConditionalStatement = ConditionalStatement.Siguiente;

        if(ConditionalStatement.Siguiente.Token == 125){ // Token de inicio de llaves

            ConditionalStatement = ConditionalStatement.Siguiente;
            ConditionalStatement = Contenido_de_las_Funciones(ConditionalStatement);

            if(ConditionalStatement.Token == 126){ // Token de cierre de llaves

                ConditionalStatement = ConditionalStatement.Siguiente;

                if(ConditionalStatement.Token == 208){ // Token de la palabra reservada 'else'

                    ConditionalStatement = ConditionalStatement.Siguiente;

                    if(ConditionalStatement.Token == 125){ // Token de inicio de llaves

                        ConditionalStatement = Contenido_de_las_Funciones(ConditionalStatement);

                        if(ConditionalStatement.Token == 126){ // Token de cierre de llaves
```

```
                } else {

                    System.out.println("Se espera un cierre de llaves en la línea " + ConditionalStatement.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
            } else {

                System.out.println("Se espera un inicio de llaves en la línea " + ConditionalStatement.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        } else {

            break;
        }
    } else {

        System.out.println("Se espera un cierre de llaves en la línea " + ConditionalStatement.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
} else {

    System.out.println("Se espera un inicio de llaves en la línea " + ConditionalStatement.Linea);
    errorEncontrado = true;
    System.exit(0);
} else {

    ConditionalStatement = Operadores_Condicionales_y_Logicos(ConditionalStatement);
    ConditionalStatement = Valores_de_Salida(ConditionalStatement);

    if(ConditionalStatement.Siguiente.Token == 122){ // Token de cierre de paréntesis

        ConditionalStatement = ConditionalStatement.Siguiente;

        if(ConditionalStatement.Siguiente.Token == 125){ // Token de inicio de llaves

            ConditionalStatement = ConditionalStatement.Siguiente;
            ConditionalStatement = Contenido_de_las_Funciones(ConditionalStatement);

            if(ConditionalStatement.Token == 126){ // Token de cierre de llaves

                ConditionalStatement = ConditionalStatement.Siguiente;

                if(ConditionalStatement.Token == 208){ // Token de la palabra reservada 'else'

                    ConditionalStatement = ConditionalStatement.Siguiente;

                    if(ConditionalStatement.Token == 125){ // Token de inicio de llaves

                        ConditionalStatement = Contenido_de_las_Funciones(ConditionalStatement);

                        if(ConditionalStatement.Token == 126){ // Token de cierre de llaves
```

```
                } else {

                    System.out.println("Se espera un cierre de llaves en la línea " + ConditionalStatement.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
            } else {

                System.out.println("Se espera un inicio de llaves en la línea " + ConditionalStatement.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        } else {

            break;
        }
    } else {

        System.out.println("Se espera un cierre de llaves en la línea " + ConditionalStatement.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
} else {

    System.out.println("Se espera un inicio de llaves en la línea " + ConditionalStatement.Linea);
    errorEncontrado = true;
    System.exit(0);
}
} else {

    ConditionalStatement = Operadores_Condicionales_y_Logicos(ConditionalStatement);
    ConditionalStatement = Valores_de_Salida(ConditionalStatement);

    if(ConditionalStatement.Siguiente.Token == 122){ // Token de cierre de paréntesis

        ConditionalStatement = ConditionalStatement.Siguiente;

        if(ConditionalStatement.Siguiente.Token == 125){ // Token de inicio de llaves

            ConditionalStatement = ConditionalStatement.Siguiente;
            ConditionalStatement = Contenido_de_las_Funciones(ConditionalStatement);

            if(ConditionalStatement.Token == 126){ // Token de cierre de llaves

                ConditionalStatement = ConditionalStatement.Siguiente;

                if(ConditionalStatement.Token == 208){ // Token de la palabra reservada 'else'

                    ConditionalStatement = ConditionalStatement.Siguiente;

                    if(ConditionalStatement.Token == 125){ // Token de inicio de llaves

                        ConditionalStatement = Contenido_de_las_Funciones(ConditionalStatement);

                        if(ConditionalStatement.Token == 126){ // Token de cierre de llaves
```

```
    if(ConditionalStatement.Token == 126){ // Token de cierre de llaves

        ConditionalStatement = ConditionalStatement.Siguiente;

        if(ConditionalStatement.Token == 208){ // Token de la palabra reservada 'else'

            ConditionalStatement = ConditionalStatement.Siguiente;

            if(ConditionalStatement.Token == 125){ // Token de inicio de llaves

                ConditionalStatement = Contenido_de_las_Funciones(ConditionalStatement);

                if(ConditionalStatement.Token == 126){ // Token de cierre de llaves

                    ConditionalStatement = ConditionalStatement.Siguiente;
                    break;

                } else {

                    System.out.println("Se espera un cierre de llaves en la línea " + ConditionalStatement.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
            } else {

                System.out.println("Se espera un inicio de llaves en la línea " + ConditionalStatement.Linea);
                errorEncontrado = true;
                System.exit(0);
```

```
                                System.out.println("Se espera un cierre de llaves en la línea " + ConditionalStatement.Linea);
                                errorEncontrado = true;
                                System.exit(0);
                            }
                        } else {

                            System.out.println("Se espera un inicio de llaves en la línea " + ConditionalStatement.Linea);
                            errorEncontrado = true;
                            System.exit(0);
                        }
                    } else {

                        System.out.println("Se espera un cierre de paréntesis en la línea " + ConditionalStatement.Linea);
                        errorEncontrado = true;
                        System.exit(0);
                    }
                }
            }
        }
    } else {

        System.out.println("Se espera un inicio de paréntesis en la línea " + ConditionalStatement.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
}

    return ConditionalStatement;
}
```

10 Aquí se lee lo que es válido en las variables de lógica como suma, resta, multiplicar, igual etc, para tener en cuenta que valores se tienen en cuenta al realizar operaciones de lógica.

```
public Nodo Operadores_Condicionales_y_Logicos(Nodo ConditionalAndLogicalOps){

    ConditionalAndLogicalOps = ConditionalAndLogicalOps.Siguiente;

    while(ConditionalAndLogicalOps != null && ConditionalAndLogicalOps.Siguiente.Token != 122){

        switch(ConditionalAndLogicalOps.Token){
            case 108: // Token de <

                return ConditionalAndLogicalOps;

            case 109: // Token de >

                return ConditionalAndLogicalOps;

            case 110: // Token de <=

                return ConditionalAndLogicalOps;

            case 111: // Token de >=

                return ConditionalAndLogicalOps;

            case 112: // Token de ==

                return ConditionalAndLogicalOps;

            case 113: // Token de !=

                return ConditionalAndLogicalOps;

            case 114: // Token de &&

                return ConditionalAndLogicalOps;

            case 115: // Token de ||

                return ConditionalAndLogicalOps;
            default:

                ConditionalAndLogicalOps = null;
                System.out.println("Se espera un operador condicional o lógico válido en la línea " + ConditionalAndLogicalOps.Linea);
                errorEncontrado = true;
                System.exit(0);
    }
}

return ConditionalAndLogicalOps;
```

11 en este objeto nodo se toman en cuenta los valores que se lleguen a repetir en bucle al momento de ejecutar el codigo que esta incluido en el archivo .txt

```
public Nodo Declaracion_de_Bucle(Nodo LoopStatement){
    while(LoopStatement.Siguiente.Token != 126){ // Token de cierre de paréntesis

        LoopStatement = LoopStatement.Siguiente;

        if(LoopStatement.Token == 121){ // Token de inicio de paréntesis

            LoopStatement = LoopStatement.Siguiente;

            if(LoopStatement.Token == 100){ // Token del identificador

                LoopStatement = LoopStatement.Siguiente;

                if(LoopStatement.Token == 124){ // Token del :=

                    LoopStatement = LoopStatement.Siguiente;

                    if(LoopStatement.Token == 101){ // Token del número entero

                        LoopStatement = LoopStatement.Siguiente;

                        if(LoopStatement.Token == 117){ // Token del ;

                            LoopStatement = LoopStatement.Siguiente;

    LoopStatement = LoopStatement.Siguiente;

    if(LoopStatement.Token == 100){ // Token del identificador

        LoopStatement = Operadores_Condicionales_y_Logicos(LoopStatement);
        LoopStatement = LoopStatement.Siguiente;

        if(LoopStatement.Token == 101){ // Token del identificador

            LoopStatement = LoopStatement.Siguiente;

            if(LoopStatement.Token == 117){ // Token del ;

                LoopStatement = LoopStatement.Siguiente;

                if(LoopStatement.Token == 100){ // Token del identificador

                    LoopStatement = LoopStatement.Siguiente;

                    if(LoopStatement.Token == 124){ // Token del :=

                        LoopStatement = LoopStatement.Siguiente;

                        if(LoopStatement.Token == 100){ // Token del :=

                            LoopStatement = Operadores_Aritmeticos(LoopStatement);
                            LoopStatement = LoopStatement.Siguiente;
```

```
if(LoopStatement.Token == 100){ // Token del :=

    LoopStatement = Operadores_Aritmeticos(LoopStatement);
    LoopStatement = LoopStatement.Siguiente;

    if(LoopStatement.Token == 101){ // Token del :=

        LoopStatement = LoopStatement.Siguiente;

        if(LoopStatement.Token == 122){ // Token de cierre de paréntesis

            LoopStatement = LoopStatement.Siguiente;

            if(LoopStatement.Token == 125){ // Token de inicio de llaves

                LoopStatement = Contenido_de_las_Funciones(LoopStatement);

                if(LoopStatement.Token == 126){ // Token de cierre de llaves

                    LoopStatement = LoopStatement.Siguiente;
                    break;

                } else {

                    System.out.println("Se espera un inicio de llaves en la línea " + LoopStatement.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
                  } else {

                    System.out.println("Se espera un inicio de llaves en la línea " + LoopStatement.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                  }
              } else {

                  System.out.println("Se espera un cierre de paréntesis en la línea " + LoopStatement.Linea);
                  errorEncontrado = true;
                  System.exit(0);
              }
          } else {

              System.out.println("Se espera un número entero en la línea " + LoopStatement.Linea);
              errorEncontrado = true;
              System.exit(0);
          }
      } else {

          System.out.println("Se espera un identificador en la línea " + LoopStatement.Linea);
          errorEncontrado = true;
          System.exit(0);
      }
```

```
                } else {

                    System.out.println("Se espera un := en la línea " + LoopStatement.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
            } else {

                System.out.println("Se espera un identificador en la línea " + LoopStatement.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        } else {

            System.out.println("Se espera un ; en la línea " + LoopStatement.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    } else {

        System.out.println("Se espera un número entero en la línea " + LoopStatement.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
```

```
                } else{

                    System.out.println("Se espera un identificador en la línea " + LoopStatement.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
            } else {

                System.out.println("Se espera un ; en la línea " + LoopStatement.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        } else {

            System.out.println("Se espera un número enteroen la línea " + LoopStatement.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    } else {

        System.out.println("Se espera un :=en la línea " + LoopStatement.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
```

```
        } else {

            System.out.println("Se espera un inicio de paréntesisen la línea " + LoopStatement.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    }

    return LoopStatement;
}
```

12 objeto nodo en el que se comprueba el uso de la operación aritmética

```
public Nodo Operadores_Aritmeticos(Nodo ArithmeticOps){

    ArithmeticOps = ArithmeticOps.Siguiente;

    while(ArithmeticOps != null && ArithmeticOps.Siguiente.Token != 122){ // Token del cierre de paréntesis

        switch(ArithmeticOps.Token){
            case 103: // Token de suma

                return ArithmeticOps;

            case 104: // Token de resta

                return ArithmeticOps;

            case 105: // Token de multiplicación

                return ArithmeticOps;

            case 106: // Token de división

                return ArithmeticOps;

            case 107: // Token de exponenciación

                return ArithmeticOps;

            default:

                ArithmeticOps = null;
                System.out.println("Se espera un operador aritmético válido en la línea " + ArithmeticOps.Linea);
                errorEncontrado = true;
                System.exit(0);
        }
    }

    return ArithmeticOps;
}
```

13 objeto nodo en el que se comprueba el uso de la operación aritmética en el proceso de realizar sus cierres a las sus condiciones completas.

```
public Nodo Arithmetic_Operations(Nodo OperacionesAritmeticas){
    while(OperacionesAritmeticas.Siguiente.Token != 117){ // Token del ;

        if(OperacionesAritmeticas.Siguiente.Token != 210 && OperacionesAritmeticas.Siguiente.Token != 231){ // Toke

            OperacionesAritmeticas = Valores_de_Salida(OperacionesAritmeticas);

            if(OperacionesAritmeticas.Siguiente.Token == 125){ // Token de inicio de llaves

                OperacionesAritmeticas = OperacionesAritmeticas.Siguiente;
                OperacionesAritmeticas = Invocar_Estructuras(OperacionesAritmeticas);
                break;

            } else {

                OperacionesAritmeticas = Operadores_Aritmeticos(OperacionesAritmeticas);

                if(OperacionesAritmeticas.Siguiente.Token != 210 && OperacionesAritmeticas.Siguiente.Token != 231){

                    OperacionesAritmeticas = Valores_de_Salida(OperacionesAritmeticas);
                    OperacionesAritmeticas = OperacionesAritmeticas.Siguiente;

                    if(OperacionesAritmeticas.Token == 117){ // Token del ;

                        OperacionesAritmeticas = OperacionesAritmeticas.Siguiente;
                        break;

                    } else {

                        System.out.println("Se espera un ; en la línea " + OperacionesAritmeticas.Linea);
                        errorEncontrado = true;
                        System.exit(0);
                    }
                } else {

                    System.out.println("No se permiten valores booleanos en la línea " + OperacionesAritmeticas.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
            }
        } else {

            System.out.println("No se permiten valores booleanos en la línea " + OperacionesAritmeticas.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    }

    return OperacionesAritmeticas;
}
```

14 objeto nodo donde se comprueba que se declaran las funciones principales para la estructura del código.

```
public Nodo Declaracion_de_Funciones_y_Estructuras(Nodo FuncAndStructDeclarations){

    FuncAndStructDeclarations = FuncAndStructDeclarations.Siguiente;

    while(FuncAndStructDeclarations != null && FuncAndStructDeclarations.Siguiente != null &&
            FuncAndStructDeclarations.Siguiente.Token != 126){

        switch(FuncAndStructDeclarations.Token){
            case 213: // Token de la palabra reservada 'func'

                FuncAndStructDeclarations = Declaracion_de_Funciones(FuncAndStructDeclarations);
                break;

            case 232: // Token de la palabra reservada 'type'

                FuncAndStructDeclarations = Declaracion_de_Estructuras(FuncAndStructDeclarations);
                break;

            default:

                FuncAndStructDeclarations = null;
                System.out.println("Se espera una declaración de función o de estructura en la línea " +
                        FuncAndStructDeclarations.Linea);
                errorEncontrado = true;
                System.exit(0);
        }
    }

    return FuncAndStructDeclarations;
}
```

15 objeto nodo donde se comprueba que se declaren las funciones correctamente de las variables de declaración como: “func” y “type” en la estructura del código.

```
public Nodo Declaracion_de_Funciones(Nodo FuncDeclaration){
    while(FuncDeclaration.Siguiente != null){ // Token de cierre de llaves

        FuncDeclaration = FuncDeclaration.Siguiente;

        if(FuncDeclaration.Token == 100){ // Token del identificador

            FuncDeclaration = FuncDeclaration.Siguiente;

            if(FuncDeclaration.Token == 121){ // Token de inicio de paréntesis

                FuncDeclaration = FuncDeclaration.Siguiente;

                if(FuncDeclaration.Token == 122){ // Token de cierre de paréntesis

                    FuncDeclaration = FuncDeclaration.Siguiente;
                    System.out.println(FuncDeclaration.Token);

                    if(FuncDeclaration.Token == 125){ // Token de inicio de llaves

                        FuncDeclaration = Contenido_de_las_Funciones(FuncDeclaration);
                        System.out.println(FuncDeclaration.Token);

                        if(FuncDeclaration.Token  == 126){ // Token de cierre de llaves
```

```
            if(FuncDeclaration.Siguiente != null){ // Tokens de la palabra reservada 'func' y 'type'

                FuncDeclaration = Declaracion_de_Funciones_y_Estructuras(FuncDeclaration);
                break;

            } else {

                System.out.println("\nAnálisis sintáctico terminado con éxito.");
                break;
            }
        } else{

            System.out.println("Se espera un cierre de llaves en la línea " + FuncDeclaration.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    } else {

        System.out.println("Se espera un inicio de llaves en la línea " + FuncDeclaration.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
}
```

- aquí se inician las conexiones con las variables de caracteres que se relacionan con el token identificador.

```
if(FuncDeclaration.Token == 100){ // Token del identificador

    FuncDeclaration = Tipo_de_dato(FuncDeclaration);
    FuncDeclaration = FuncDeclaration.Siguiente;

    if(FuncDeclaration.Token == 118){ // Token de ,

        FuncDeclaration = FuncDeclaration.Siguiente;

        if(FuncDeclaration.Token == 100){ // Token del identificador

            FuncDeclaration = Tipo_de_dato(FuncDeclaration);
            FuncDeclaration = FuncDeclaration.Siguiente;

            if(FuncDeclaration.Token == 118){ // Token de ,

                FuncDeclaration = FuncDeclaration.Siguiente;

                if(FuncDeclaration.Token == 100){ // Token del identificador

                    FuncDeclaration = Tipo_de_dato(FuncDeclaration);
                    FuncDeclaration = FuncDeclaration.Siguiente;

                    if(FuncDeclaration.Token == 122){ // Token de cierre de paréntesis

                        FuncDeclaration = FuncDeclaration.Siguiente;

                        if(FuncDeclaration.Token == 125){ // Token de inicio de llaves
```

```
        if(FuncDeclaration.Token  == 126){ // Token de cierre de llaves

            if(FuncDeclaration.Siguiente != null){ // Tokens de la palabra reservada 'func' y 'type'

                FuncDeclaration = Declaracion_de_Funciones_y_Estructuras(FuncDeclaration);
                break;

            } else {

                System.out.println("\nAnálisis sintáctico terminado con éxito.");
                break;
            }
        } else{

            System.out.println("Se espera un cierre de llaves en la línea " + FuncDeclaration.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    } else {

        System.out.println("Se espera un inicio de llaves en la línea " + FuncDeclaration.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
} else {
```

```
            } else {

                System.out.println("Se espera un inicio de llaves en la línea " + FuncDeclaration.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        } else {

            System.out.println("Se espera un cierre de paréntesis en la línea " + FuncDeclaration.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    } else {

        System.out.println("Se espera un identificador en la línea " + FuncDeclaration.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
}
```

```
if(FuncDeclaration.Token == 122){ // Token de cierre de paréntesis

    FuncDeclaration = FuncDeclaration.Siguiente;

    if(FuncDeclaration.Token == 125){ // Token de inicio de llaves

        FuncDeclaration = Contenido_de_las_Funciones(FuncDeclaration);

        if(FuncDeclaration.Token  == 126){ // Token de cierre de llaves

            if(FuncDeclaration.Siguiente != null){ // Tokens de la palabra reservada 'func' y 'type'

                FuncDeclaration = Declaracion_de_Funciones_y_Estructuras(FuncDeclaration);
                break;

            } else {

                System.out.println("\nAnálisis sintáctico terminado con éxito.");
                break;
            }
        } else{

            System.out.println("Se espera un cierre de llaves en la línea " + FuncDeclaration.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
          } else {

            System.out.println("Se espera un inicio de llaves en la línea " + FuncDeclaration.Linea);
            errorEncontrado = true;
            System.exit(0);
          }
      } else {

          System.out.println("Se espera un cierre de paréntesis en la línea " + FuncDeclaration.Linea);
          errorEncontrado = true;
          System.exit(0);
      }
  } else {

      System.out.println("Se espera un identificador en la línea " + FuncDeclaration.Linea);
      errorEncontrado = true;
      System.exit(0);
  }
}
```

- si el valor actual del token es completado saldrá de la función y el sintáctico terminará con éxito.

```
if(FuncDeclaration.Token == 122){ // Token de cierre de paréntesis

    FuncDeclaration = FuncDeclaration.Siguiente;

    if(FuncDeclaration.Token == 125){ // Token de inicio de llaves

        FuncDeclaration = Contenido_de_las_Funciones(FuncDeclaration);

        if(FuncDeclaration.Token  == 126){ // Token de cierre de llaves

            if(FuncDeclaration.Siguiente != null){ // Tokens de la palabra reservada 'func' y 'type'

                FuncDeclaration = Declaracion_de_Funciones_y_Estructuras(FuncDeclaration);
                break;

            } else {

                System.out.println("\nAnálisis sintáctico terminado con éxito.");
                break;
            }
        } else{

            System.out.println("Se espera un cierre de llaves en la línea " + FuncDeclaration.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
                } else {

                    System.out.println("Se espera un inicio de llaves en la línea " + FuncDeclaration.Linea);
                    errorEncontrado = true;
                    System.exit(0);
                }
            } else {

                System.out.println("Se espera un cierre de paréntesis en la línea " + FuncDeclaration.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        } else {

            System.out.println("Se espera un identificador o un cierre de paréntesis en la línea " + FuncDeclaration.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    } else {

        System.out.println("Se espera un inicio de paréntesis en la línea " + FuncDeclaration.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
} else {

    System.out.println("Se espera un identificador en la línea " + FuncDeclaration.Linea);
    errorEncontrado = true;
    System.exit(0);
```

16 objeto nodo donde se llegan a declarar las variables de estructuras

```
public Nodo Declaracion_de_Estructuras(Nodo StructDeclaration){
    while(StructDeclaration.Siguiente != null){ // Token de cierre de llaves

        StructDeclaration = StructDeclaration.Siguiente;

        if(StructDeclaration.Token  == 100){ // Token del identificador

            StructDeclaration = StructDeclaration.Siguiente;

            if (StructDeclaration.Token  == 229){ // Token de la palabra reservada 'struct'

                StructDeclaration = StructDeclaration.Siguiente;

                if (StructDeclaration.Token  == 125){ // Token de inicio de llaves

                    StructDeclaration = StructDeclaration.Siguiente;

                    if(StructDeclaration.Token  == 100){ // Token del identificador

                        StructDeclaration = Tipo_de_dato(StructDeclaration);
                        StructDeclaration = StructDeclaration.Siguiente;

                        if(StructDeclaration.Token  == 117){ // Token del ;

                            StructDeclaration = StructDeclaration.Siguiente;
```

```
  if(StructDeclaration.Token == 100){ // Token del identificador

      StructDeclaration = Tipo_de_dato(StructDeclaration);
      StructDeclaration = StructDeclaration.Siguiente;

      if(StructDeclaration.Token  == 117){ // Token del ;

          StructDeclaration = StructDeclaration.Siguiente;

          if(StructDeclaration.Token == 100){ // Token del identificador

              StructDeclaration = Tipo_de_dato(StructDeclaration);
              StructDeclaration = StructDeclaration.Siguiente;

              if(StructDeclaration.Token  == 117){ // Token del ;

                  StructDeclaration = StructDeclaration.Siguiente;

                  if(StructDeclaration.Token == 100){ // Token del identificador

                      StructDeclaration = Tipo_de_dato(StructDeclaration);
                      StructDeclaration = StructDeclaration.Siguiente;

                      if(StructDeclaration.Token  == 117){ // Token del ;

                          StructDeclaration = StructDeclaration.Siguiente;

                          if(StructDeclaration.Token  == 126){ // Token de cierre de llaves
```

```
        if(StructDeclaration.Token  == 117){ // Token del ;

            StructDeclaration = StructDeclaration.Siguiente;

            if(StructDeclaration.Token  == 126){ // Token de cierre de llaves

                if(StructDeclaration.Siguiente != null){ // Tokens de la palabra reservada 'func' y 'type'

                    StructDeclaration = Declaracion_de_Funciones_y_Estructuras(StructDeclaration);
                    break;

                } else {

                    System.out.println("\nAnálisis sintáctico terminado con éxito.");
                    break;
                }
            } else{

                System.out.println("Se espera un cierre de llaves o un identificador en la línea " + StructDeclaration.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        } else {

            System.out.println("Se espera un ; en la línea " + StructDeclaration.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    }
```

```
        if(StructDeclaration.Token  == 126){ // Token de cierre de llaves

            if(StructDeclaration.Siguiente != null){ // Tokens de la palabra reservada 'func' y 'type'

                StructDeclaration = Declaracion_de_Funciones_y_Estructuras(StructDeclaration);
                break;

            } else {

                System.out.println("\nAnálisis sintáctico terminado con éxito.");
                break;
            }
        } else{

            System.out.println("Se espera un cierre de llaves o un identificador en la línea " + StructDeclaration.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    } else {

        System.out.println("Se espera un ; en la línea " + StructDeclaration.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
}
```

```
        if(StructDeclaration.Token  == 126){ // Token de cierre de llaves

            if(StructDeclaration.Siguiente != null){ // Tokens de la palabra reservada 'func' y 'type'

                StructDeclaration = Declaracion_de_Funciones_y_Estructuras(StructDeclaration);
                break;

            } else {

                System.out.println("\nAnálisis sintáctico terminado con éxito.");
                break;
            }
        } else{

            System.out.println("Se espera un cierre de llaves o un identificador en la línea " + StructDeclaration.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    } else {

        System.out.println("Se espera un ; en la línea " + StructDeclaration.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
}
```

```
        if(StructDeclaration.Token  == 126){ // Token de cierre de llaves

            if(StructDeclaration.Siguiente != null){ // Tokens de la palabra reservada 'func' y 'type'

                StructDeclaration = Declaracion_de_Funciones_y_Estructuras(StructDeclaration);
                break;

            } else {

                System.out.println("\nAnálisis sintáctico terminado con éxito.");
                break;
            }
        } else{

            System.out.println("Se espera un cierre de llaves o un identificador en la línea " + StructDeclaration.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    } else {

        System.out.println("Se espera un inicio de llaves en la línea " + StructDeclaration.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
} else {

    System.out.println("Se espera la palabra 'struct' en la línea " + StructDeclaration.Linea);
    errorEncontrado = true;
    System.exit(0);
}
```

17 objeto nodo en el que se toma en cuenta los identificadores de las funciones que incluye el lenguaje de programación go, conllevando los valores de las variables de operaciones y funciones para ser utilizados para los valores de identificadores.

```
public Nodo Uso_de_Identificadores(Nodo IdUses){

    IdUses = IdUses.Siguiente;
    System.out.println(IdUses.Token);

    while(IdUses != null && IdUses.Siguiente != null && IdUses.Siguiente.Token != 126){

        if(IdUses.Token == 124){ // Token del :=

            IdUses = Arithmetic_Operations(IdUses);
            break;

        }

        if(IdUses.Token == 121){ // Token de inicio de paréntesis

            IdUses = Invocar_Funciones(IdUses);
            break;

        }

        if(IdUses.Token == 120){ // Token de .

            IdUses = IdUses.Siguiente;
            System.out.println(IdUses.Token);

        if(IdUses.Token == 223){ // Token de la palabra reservada 'println'

            IdUses = Salida_de_datos(IdUses);
            break;

        }

        if(IdUses.Token == 226){

            IdUses = Entrada_de_datos(IdUses);
            break;

        } else {

            System.out.println("Se espera las palabras 'print' o 'scan' en la línea " + IdUses.Linea);
            errorEncontrado = true;
            System.exit(0);
        }

    } else {

        System.out.println("Se espera un :=, un inicio de paréntesis o un . en la línea " + IdUses.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
}

return IdUses;
```

18 objeto nodo que toma los valores de salida de las estructura y valida que estas salgan en orden.

```
public Nodo Invocar_Estructuras(Nodo InvokeStructs){
    while(InvokeStructs.Siguiente.Token != 126){ // Token de cierre de llaves

        if(InvokeStructs.Siguiente.Token != 126){ // Token de cierre de llaves

            InvokeStructs = Valores_de_Salida(InvokeStructs);
            InvokeStructs = InvokeStructs.Siguiente;

            if(InvokeStructs.Token == 118){ // Token de ,

                InvokeStructs = Valores_de_Salida(InvokeStructs);
                InvokeStructs = InvokeStructs.Siguiente;

                if(InvokeStructs.Token == 118){ // Token de ,

                    InvokeStructs = Valores_de_Salida(InvokeStructs);
                    InvokeStructs = InvokeStructs.Siguiente;

                    if(InvokeStructs.Token == 118){ // Token de ,

                        InvokeStructs = Valores_de_Salida(InvokeStructs);
                        InvokeStructs = InvokeStructs.Siguiente;

                        if(InvokeStructs.Token == 126){ // Token de cierre de llaves

                            InvokeStructs = InvokeStructs.Siguiente;
                            break;
        } else {

            System.out.println("Se espera un cierre de llaves en la línea " + InvokeStructs.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    }

    if(InvokeStructs.Token == 126){ // Token de cierre de llaves

        InvokeStructs = InvokeStructs.Siguiente;
        break;

    } else {

        System.out.println("Se espera una , o un cierre de paréntesis en la línea " + InvokeStructs.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
}

if(InvokeStructs.Token == 126){ // Token de cierre de llaves

    InvokeStructs = InvokeStructs.Siguiente;
    break;

} else {
```

```
            System.out.println("Se espera una , o un cierre de paréntesis en la línea " + InvokeStructs.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    }

    if(InvokeStructs.Token == 126){ // Token de cierre de llaves

        InvokeStructs = InvokeStructs.Siguiente;
        break;

    } else {

        System.out.println("Se espera una , o un cierre de paréntesis en la línea " + InvokeStructs.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
} else {

    InvokeStructs = InvokeStructs.Siguiente;

    if(InvokeStructs.Token == 126){ // Token del ;

        InvokeStructs = InvokeStructs.Siguiente;
        break;
```

En la clase ***GO_Based_Compiler.java***, existe un método llamado ***Seleccionar_Archivo()***, el cual abre una ventana de selección de archivos para que se pueda abrir un archivo de texto disponible en los distintos dispositivos de almacenamiento disponibles en su computadora o laptop.

Este método también se asegura de que el archivo escogido sea de terminación .txt, por lo que se detendrá toda operación del compilador en caso de elegir algún otro tipo de archivo.

```java
// Método para seleccionar archivos de texto a través de una ventana de selección de archivos
private static String Seleccionar_Archivo() {
    JFileChooser fileChooser = new JFileChooser();
    FileNameExtensionFilter Filtro_de_archivos = new FileNameExtensionFilter(description:"Archivos de texto (.txt)", extensions:"txt");
    fileChooser.setFileFilter(filter:Filtro_de_archivos);

    int Seleccion = fileChooser.showOpenDialog(parent:null);

    if (Seleccion == JFileChooser.APPROVE_OPTION) {
        File Archivo_seleccionado = fileChooser.getSelectedFile();

        if (Archivo_seleccionado.getName().endsWith(suffix:".txt")) {

            return Archivo_seleccionado.getAbsolutePath();

        } else {

            // El archivo seleccionado no es un archivo de texto, cancelar todo
            System.out.println(x:"Error: El archivo seleccionado no es un archivo de texto. Cancelando.");
            System.exit(status:0);
            return null;
        }
    } else {

        // El usuario canceló la selección
        System.exit(status:0);
        return null;
    }
}
```

En el método ***main()***, se llama el método ***Seleccionar_Archivo()*** para convertir el archivo de texto seleccionado a un string. Luego se invoca la clase ***Analisis_Lexico()*** para que inicie el análisis léxico del archivo .txt.

Si no ocurren errores, se desplegará en la terminal de Netbeans todos los tokens detectados en el archivo de texto y se procederá a inicializar el análisis sintáctico. En caso de no encontrar errores en la sintaxis del archivo. txt, se desplegará un mensaje que indica que no se encontraron errores sintácticos y que dicho análisis terminó con éxito.

```
String Archivo_a_compilar = Seleccionar_Archivo();

Analisis_Lexico Lex_Analysis = new Analisis_Lexico();

if(!Lex_Analysis.errorEncontrado){

    System.out.println(x: "-------------------------------------------");
    System.out.println(x: "Análisis léxico.");
    System.out.println(x: "-------------------------------------------\n");

    Nodo Lista_de_Nodos = Lex_Analysis.AnalizarCodigo(Archivo_a_compilar);

    System.out.println(x: "\nAnálisis léxico terminado con éxito.");

    Analisis_Sintactico Syntax_Analysis = new Analisis_Sintactico();

    System.out.println(x: "Procediendo a inicializar el análisis sintáctico.\n");

    if(!Syntax_Analysis.errorEncontrado){

        System.out.println(x: "-------------------------------------------");
        System.out.println(x: "Análisis sintáctico.");
        System.out.println(x: "-------------------------------------------\n");

        Nodo Lista_Nodos_Sintactico = Syntax_Analysis.AnalizarSintaxis(Nodo_Actual: Lista_de_Nodos);
    }
}
```

Estas son las librerías importadas para la selección de archivos

```
import java.io.File;
import javax.swing.JFileChooser;
import javax.swing.filechooser.FileNameExtensionFilter;
```

En la clase ***Nodo***, se capturan los valores del Token, Lexema y Número de Renglón obtenidos durante el análisis léxico.

```java
package com.mycompany.go_based_compiler;

public class Nodo {
    String Lexema;
    int Token;
    int Linea;
    Nodo Siguiente = null;

    Nodo(String Lexema, int Token, int Linea){
        this.Lexema = Lexema;
        this.Token = Token;
        this.Linea = Linea;
    }
}
```

19 objeto nodo en el que se utilizan los valores de las funciones para proceder en el código.

```
public Nodo Invocar_Funciones(Nodo InvokeFuncs){
    while(InvokeFuncs.Siguiente.Token != 117){ // Token del ;

        if(InvokeFuncs.Siguiente.Token != 122){ // Token de cierre de paréntesis

            InvokeFuncs = Valores_de_Salida(InvokeFuncs);
            InvokeFuncs = InvokeFuncs.Siguiente;

            if(InvokeFuncs.Token == 118){ // Token de ,

                InvokeFuncs = Valores_de_Salida(InvokeFuncs);
                InvokeFuncs = InvokeFuncs.Siguiente;

                if(InvokeFuncs.Token == 118){ // Token de ,

                    InvokeFuncs = Valores_de_Salida(InvokeFuncs);
                    InvokeFuncs = InvokeFuncs.Siguiente;

                    if(InvokeFuncs.Token == 122){ // Token de cierre de llaves

                        InvokeFuncs = InvokeFuncs.Siguiente;

                        if(InvokeFuncs.Token == 117){ // Token del ;

                            InvokeFuncs = InvokeFuncs.Siguiente;
                            break;

                            System.out.println("Se espera un ; en la línea " + InvokeFuncs.Linea);
                            errorEncontrado = true;
                            System.exit(0);
                        }
                    } else {

                        System.out.println("Se espera un cierre de paréntesis en la línea " + InvokeFuncs.Linea);
                        errorEncontrado = true;
                        System.exit(0);
                    }
                }

                if(InvokeFuncs.Token == 122){ // Token de cierre de llaves

                    InvokeFuncs = InvokeFuncs.Siguiente;

                    if(InvokeFuncs.Token == 117){ // Token del ;

                        InvokeFuncs = InvokeFuncs.Siguiente;
                        break;

                    } else {

                        System.out.println("Se espera un ; en la línea " + InvokeFuncs.Linea);
                        errorEncontrado = true;
                        System.exit(0);
                    }
                } else {
```

```
            System.out.println("Se espera una , o un cierre de paréntesis en la línea " + InvokeFuncs.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    }

    if(InvokeFuncs.Token == 122){ // Token de cierre de llaves

        InvokeFuncs = InvokeFuncs.Siguiente;

        if(InvokeFuncs.Token == 117){ // Token del ;

            InvokeFuncs = InvokeFuncs.Siguiente;
            break;

        } else {

            System.out.println("Se espera un ; en la línea " + InvokeFuncs.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    } else {

        System.out.println("Se espera una , o un cierre de paréntesis en la línea " + InvokeFuncs.Linea);
        errorEncontrado = true;
        System.exit(0);
    }
    } else {

        InvokeFuncs = InvokeFuncs.Siguiente;

        if(InvokeFuncs.Token == 122){ // Token de cierre de llaves

            InvokeFuncs = InvokeFuncs.Siguiente;

            if(InvokeFuncs.Token == 117){ // Token del ;

                InvokeFuncs = InvokeFuncs.Siguiente;
                break;

            } else {

                System.out.println("Se espera un ; en la línea " + InvokeFuncs.Linea);
                errorEncontrado = true;
                System.exit(0);
            }
        } else {

            System.out.println("Se espera un cierre de paréntesis en la línea " + InvokeFuncs.Linea);
            errorEncontrado = true;
            System.exit(0);
        }
    }
}

return InvokeFuncs;
```

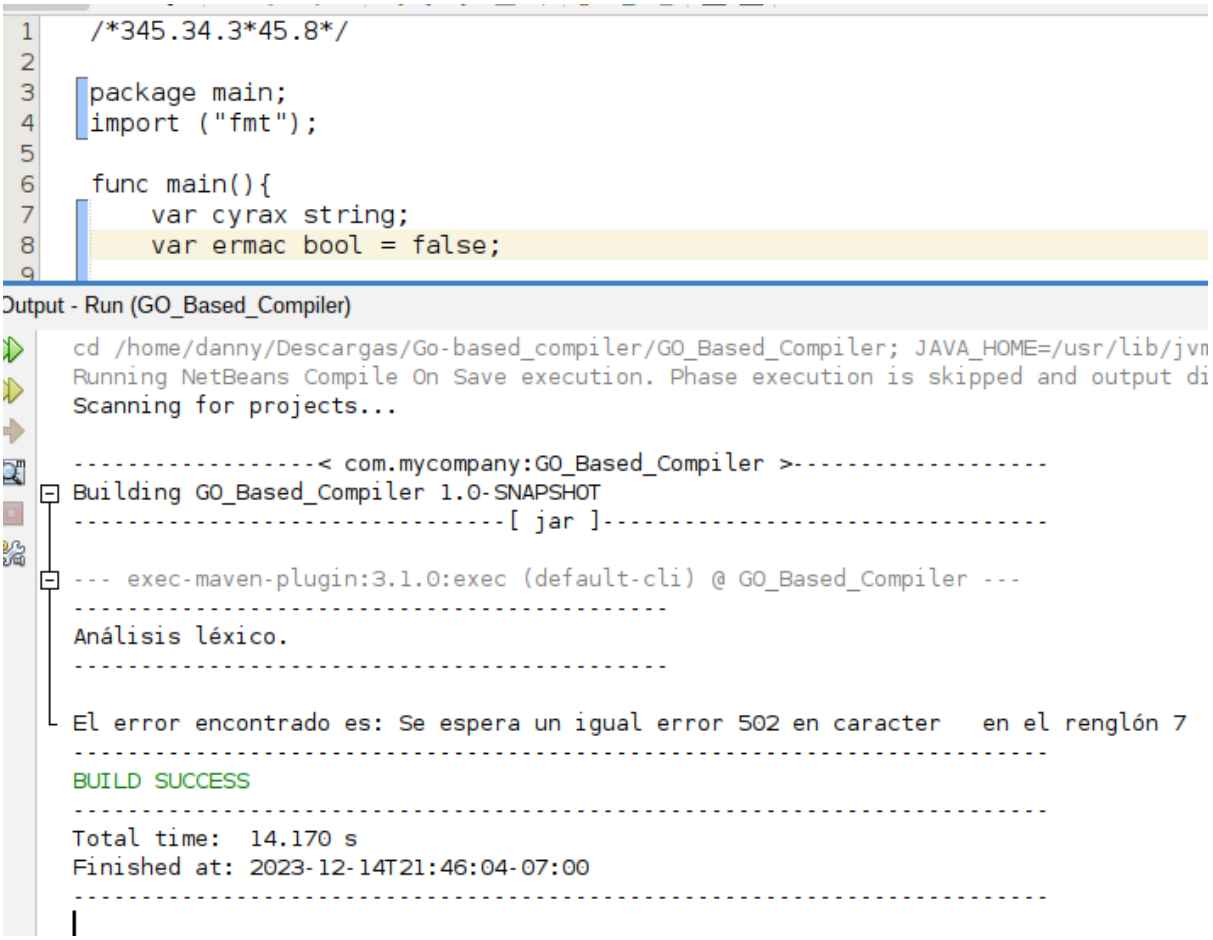
Resultados

Corrida sin errores

```
/*345.34.3*45.8*/

package main;
import ("fmt");

func main(){
    var cyrax string;
    var ermac bool := false;

    raiden := 40 + 30;
    liukang := raiden - 20;
    kabal := raiden / liukang;
    noobsaibot := kabal ^ 3;
    scorpion := noobsaibot * raiden;

    fmt.scan(&cyrax);

    fmt.println();
    fmt.println("Come here!");
    fmt.println(cyrax);

    if(!ermac){
        for (sindel := 0; sindel <= 10; sindel := sindel + 1) {
            if(liukang > raiden){
                fmt.println("Come here!");
            } else {
                fmt.println(cyrax);
            }
        }
    }

    jade();
    sektor(69, true, "Flying Thunder God!");

    kitana := Kitana{false, 42, 6.9, "Toasty!"}
}

type kitana struct {

    nightwolf bool;
    stryker int;
    kunglao float;
    jax string;
}

func jade (){
    fmt.println("Get over here!");
}

func sektor (frost int, smoke bool, subzero string){

}
```

```
Lexema: , | Token: 118 | Linea: 50
Lexema: smoke | Token: 100 | Linea: 50
Lexema: bool | Token: 200 | Linea: 50
Lexema: , | Token: 118 | Linea: 50
Lexema: subzero | Token: 100 | Linea: 50
Lexema: string | Token: 228 | Linea: 50
Lexema: ) | Token: 122 | Linea: 50
Lexema: { | Token: 125 | Linea: 50
Lexema: } | Token: 126 | Linea: 52

Análisis léxico terminado con éxito.
Procediendo a inicializar el análisis sintáctico.

------------------------------------------
Análisis sintáctico.
------------------------------------------


Análisis sintáctico terminado con éxito.
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  13.926 s
Finished at: 2023-12-14T21:44:15-07:00
------------------------------------------------------------------------
```

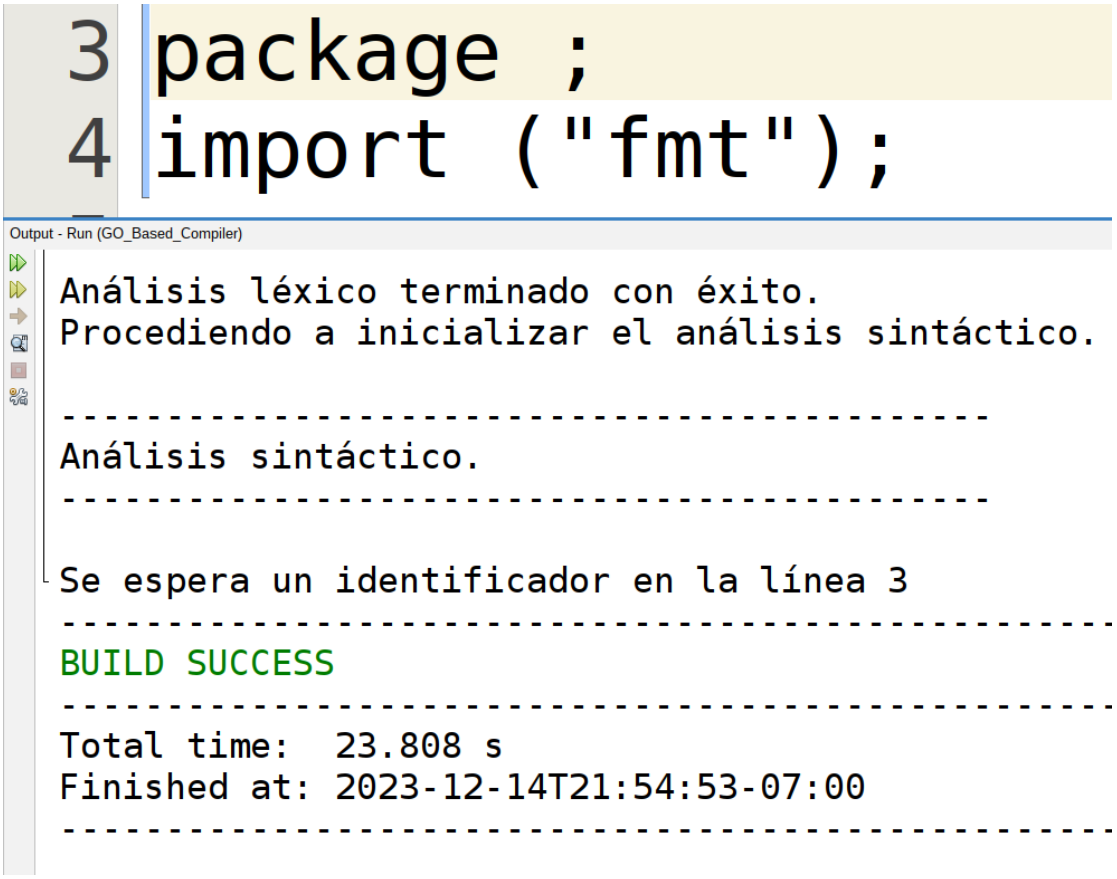
Error 502 se espera un =



```
1  /*345.34.3*45.8*/
2
3  package main;
4  import ("fmt");
5
6  func main(){
7      var cyrax string;
8      var ermac bool = false;
9
```

```
Output - Run (GO_Based_Compiler)
cd /home/danny/Descargas/Go-based_compiler/GO_Based_Compiler; JAVA_HOME=/usr/lib/jvm
Running NetBeans Compile On Save execution. Phase execution is skipped and output di
Scanning for projects...

------------------< com.mycompany:GO_Based_Compiler >-------------------
Building GO_Based_Compiler 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.1.0:exec (default-cli) @ GO_Based_Compiler ---
------------------------------------------
Análisis léxico.
------------------------------------------

El error encontrado es: Se espera un igual error 502 en caracter   en el renglón 7
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  14.170 s
Finished at: 2023-12-14T21:46:04-07:00
------------------------------------------------------------------------
```

Se espera un identificador en la línea 3



Error 505 se espera cierre de cadena (“)

```
18      fmt.println();
19      fmt.println("Come here!);
```

Output - Run (GO_Based_Compiler)

```
cd /home/danny/Descargas/Go-based_compiler/GO_Based_Compiler; JAVA_HOME=/u
Running NetBeans Compile On Save execution. Phase execution is skipped and
Scanning for projects...

------------------< com.mycompany:GO_Based_Compiler >-------------------
Building GO_Based_Compiler 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.1.0:exec (default-cli) @ GO_Based_Compiler ---
------------------------------------------
Análisis léxico.
------------------------------------------

El error encontrado es: Se espera cierre de cadena error 505 en caracter
 en el renglón 19
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  11.031 s
Finished at: 2023-12-14T21:52:14-07:00
------------------------------------------------------------------------
```

Se espera un ;

```
32      jade();
33      sektor(69, true, "Flying Thunder God!")
34
```

Output - Run (GO_Based_Compiler)

```
Lexema: ) | Token: 122 | Linea: 50
Lexema: { | Token: 125 | Linea: 50
Lexema: } | Token: 126 | Linea: 52

Análisis léxico terminado con éxito.
Procediendo a inicializar el análisis sintáctico.

------------------------------------------------
Análisis sintáctico.
------------------------------------------------

Se espera un ; en la línea 35
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  15.883 s
Finished at: 2023-12-14T21:59:14-07:00
------------------------------------------------------------------------
```

Se espera un identificador

```
50  func sektor (frost int, smoke bool,){
51
52  }
```

```
Output - Run (GO_Based_Compiler)
Lexema: } | Token: 126 | Línea: 52

Análisis léxico terminado con éxito.
Procediendo a inicializar el análisis sintáctico.

------------------------------------------
Análisis sintáctico.
------------------------------------------

Se espera un identificador en la línea 50
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  13.045 s
Finished at: 2023-12-14T22:00:46-07:00
------------------------------------------------------------------------
```

Conclusiones

José Jorge Gastelum Angulo: Esta práctica hasta ahora es el código más complejo que he hecho en la carrera, y el hacer el autómata finito es algo que me será útil para más adelante en la carrera.

Jesus Rafael Valles Fitch: En conclusión, podemos tomar en cuenta al momento de realizar el analizador léxico podemos ver que el código de este facilita para la visualización de error en el código del lenguaje de programación, ayudando a saber los errores que este puede tener al momento de ejecutarlo en una aplicación de programación.

Iván Daniel Beltrán Santiago: Dentro de todo, realizar el analizador léxico ha sido fascinante por el modo en que se analiza cada carácter, cada palabra reservada y cada error que pueda existir en un código. La experiencia de programar y probar parte del compilador, hasta ahora, ha sido todo menos agradable, pero los resultados han sido, por lo menos, satisfactorios.

Referencias bibliográficas

- *Effective Go - the Go programming language*. (s. f.). https://go.dev/doc/effective_go
- De Admin, V. T. L. E. (2017, 16 abril). AUTÓMATA FINITO. Interpolados. https://interpolados.wordpress.com/2017/03/03/automata-finito/
- EcuRed. (s. f.). Autómata finito - ECURED. https://www.ecured.cu/Aut%C3%B3mata_finito
- Martínez Valentín, Carlos & Hernández Gómez, Germán David & Hernández Plascencia, Arilenne & Hernández Antonio, Johnathan & Martínez Martínez, Araceli. (s. f.). Unidad IV. Análisis Léxico. Introducción a la Teoría de Lenguajes Formales. https://teorialenformalesvalles.blogspot.com/p/unidad-iv-analisis-lexico.html
- Aguilera Sierra, María del Mar & Galvéz Rojas, Sergio. (s. f). 'Tema 3. Análisis Sintáctico'. Traductores, Compiladores e Intérpretes. http://www.lcc.uma.es/~galvez/ftp/tci/tictema3.pdf
- Urrutia, D. (2023, 16 octubre). Qué es un analizador sintáctico o parser | Definición y tipos. Arimetrics. https://www.arimetrics.com/glosario-digital/analizador-sintactico-parser