TEC | Tecnológico de Costa Rica

Escuela de Ingeniería en Computación
Ingeniería en Computación
IC5701 - Compiladores e Intérpretes

Compilador en C

Programado En Java Usando La Herramienta JFlex

Daniel Araya Sambucci
danielarayasambucci@estudiantec.cr
2020207809

Andrés Masís Rojas
andres.masis.rojas@estudiantec.cr
2020127158

Esteban Leiva Montenegro
estebanlm@estudiantec.cr
2020426227

Cartago, Costa Rica
Septiembre 2022

Índice general

Capítulo 1

Introducción

Los compiladores son programas de una computadora que traducen un lenguaje a otro. Un compilador toma como su entrada un programa escrito en un lenguaje fuente y produce un programa equivalente escrito en su lenguaje objetivo (Louden, 2004). Para que un programa llegue a ser compilado o traducido deberá pasar por diferentes procesos; algunos de estos procesos pueden ser, el analizador léxico, analizador sintáctico y semántico. Para este proyecto se empezará a desarrollar un compilador basado en el lenguaje de programación C iniciando con el analizador léxico ("Scanner").

1.1. Estrategia de Solución

Para el desarrollo de la solución se hizo un uso intensivo de la herramienta JFlex. Esta herramienta se encarga de generar todos los autómatas, por lo que la solución planteada se basa en las distintas expresiones regulares que serán aplicadas por el analizador léxico para permitir o no los tipos de Token que se definan. Inicialmente se definen los posibles tipos que pueden tener los tokens. Para este proyecto se tiene identificador, error, palabra reservada, operador, literal y también se define el error. Esta sección es importante pues con base en ella es que más adelante se hacen las categorizaciones de los distintos tokens encontrados. Todas estas constantes se guardan en la clase TokensConstants de Java, la cual existe con este único propósito.

Posteriormente se tiene la clase Token. Esta clase nos permite almacenar la información de cada Token que se vaya encontrando. Se tienen 3 atributos que son los datos esenciales que ocupamos de cada token: el tipo de token, el lexema y la línea donde se encuentra. Se usan métodos auxiliares (como getters) para obtener el valor adecuado a cada una de estas características.

El archivo más importante de la solución es "simple.flex". Este es el que aloja todas las definiciones de las expresiones regulares y por ende este archivo es el pilar estructural mediante el cual se genera el Analizador Léxico. A continuación se va a dar un listado general de las distintas expresiones regulares que

se definieron. Para ver cada expresión regular en detalle puede revisar el código fuente, ya que por la naturaleza de este documento sería tedioso para el lector entrar en mucho detalle acá.

Primero se definieron los espacios en blanco y saltos de líneas. Después se encuentran las literales correspondientes a los números. Estos se dividen en enteros y flotantes. Los enteros se dividen en 0, decimal, octal y hexadecimal, todos estos juntos generan los Integer. Posteriormente, se definen los alfabetos que aceptan cada uno de esos tipos.

Seguidamente, se procede a trabajar con los flotantes. Lo primero que se hace es definir la estructura del exponente. Acá se indica que la forma es e+-num. Se define la expresión regular asociada. Posteriormente se define la forma de un número flotante, esto es un número con punto decimal seguido (opcionalmente) por el exponente definido previamente.

Una vez definidos los números, se continua con los aspectos alfanuméricos. Se definen las expresiones regulares para los caracteres y los strings. Los strings corresponden a cualquier caracter que sea englobado por unas comillas dobles “”. Los caracteres se definen como un solo carácter englobado por unas comillas simples ‘’. Todo lo descrito anteriormente, números y caracteres alfanuméricos, permite generar, como ya se mencionó, los literales.

Después se tienen que definir los identificadores. Los identificadores son los que nos permiten acceder a un elemento (variable o función, por ejemplo) que creemos en el código. Los identificadores pueden ser cualquier token que comience con una letra y después puede venir N cantidad de cualquier carácter que pertenezca al abecedario definido. Note que los identificadores son muy abiertos ya que puede ser cualquier nombre (válido) que al programador se le ocurra.

Las palabras reservadas son palabras especiales que tienen un significado previamente definido en el lenguaje. Dichas palabras son, como lo dice su nombre, reservadas para uso exclusivo del lenguaje de programación, por lo que se bloquea su uso para definir cualquier tipo de dato.

Por otra parte, los operadores sirven para manipular datos. Estos por lo general son aplicados a los literales. Como C tiene una cantidad finita de operaciones y cada operación es única y específica, entonces también hay una cantidad finita de operadores específicos. Al igual que con las palabras reservadas, se definen explícitamente todos los operadores de C.

Después se definen los comentarios. En estas opciones se puede escribir cualquier cosa. En el caso de los comentarios multilínea puede venir cualquier cosa que esté entre /* y */. En el caso de los comentarios de una línea, puede venir cualquier cosa después de //, no cuenta para otras líneas o en la misma línea o antes de ”slash”. Finalmente se tienen los errores. Estos corresponden a cualquier secuencia de caracteres que no sea parte de ninguna de las categorías previamente descritas. Se les considera un error debido a que no se consideran parte de la solución presentada para el analizador léxico.

De esta forma se tienen definidas todas las expresiones regulares de las posibles hileras que C puede leer. A continuación se agregan algunas instrucciones

para desplegar la información y manejarla de manera más precisa.

En el caso de los comentarios no se hace nada. Una vez que ya se identifica el sector que pertenece a un comentario este no se debe analizar. Posteriormente se indica que los saltos de línea y espacios en blanco se deben ignorar pues estos no deben contarse como tokens. Con respecto a los operadores estos solo se procesan como se describió previamente. Al ser un conjunto finito y debidamente definido no hay espacio para situaciones inesperadas. Con las palabras reservadas se da la misma situación.

Con los literales hay infinidad de opciones. Para la mayoría de casos se cubre como se describió la expresión regular. Se puede dar el caso de que se obtenga un token que comience con un número y sea seguido por letras. Para evitar un caso especial de error donde no se deben crear variables que empiecen por un número, se creó una expresión regular de error específica a esa situación. Así mismo, se garantiza que los literales sean solo números enteros, números flotantes o strings debidamente delimitados por las comillas.

Además, se debe recordar que C permite las importaciones y macros, estos tienen sus propios símbolos especiales como o palabras especiales como include o define. En base a lo anterior y mediante el análisis de la sintáxis de dichas directivas, se permite el uso de las mismas, aplicando para cualquier importación de archivo header, librería o definición de macro.

Todos estos conceptos mencionados anteriormente son aplicados a la sintáxis definida por JFlex, de tal forma que permite generar el Analizador Léxico, donde todos los autómatas respectivos son generados. De esta forma, traducidos al lenguaje Java, permiten la ejecución del Analizador Léxico

Finalmente, para la implementación de la solución, se diseñó una sencilla interfaz gráfica para hacer uso de esta. En esta interfaz gráfica el usuario puede escribir lo que desee o cargar un archivo de texto. Al darle compilar se verifica si todos los tokens son válidos en C. Se despliega una pequeña matriz con todos los tokens encontrados, su tipo, en qué líneas aparecen y el total de apariciones. En caso de qué haya error (un token no válido), se despliega otra matriz de errores. Esta tiene una estructura muy similar, errores encontrados, en qué líneas aparecen y el total de apariciones.

Capítulo 2

Casos de Prueba

El compilador cumple en su totalidad con las funcionalidades descritas en el enunciado del proyecto. El Scanner (Analizador Léxico) identifica los 4 grandes grupos de Tokens:

- Identificadores
- Operadores
- Literales
- Palabras Reservadas

Cada uno de esos grandes grupos será desarrollado a continuación, mostrando los resultados que generó la funcionalidad definida para cada apartado, mediante casos de uso, así como otros apartados que se consideró pertinentes demostrar.

Empezando por el primer aspecto a tomar en cuenta, se encuentran los comentarios. El compilador es capaz de ignorar todos los tokens que se encuentran en el interior de los comentarios, ya sea por línea o por bloque. Así mismo, el compilador no es capaz de reconocer comentarios anidados, funcionalidad especificada que no era necesaria.

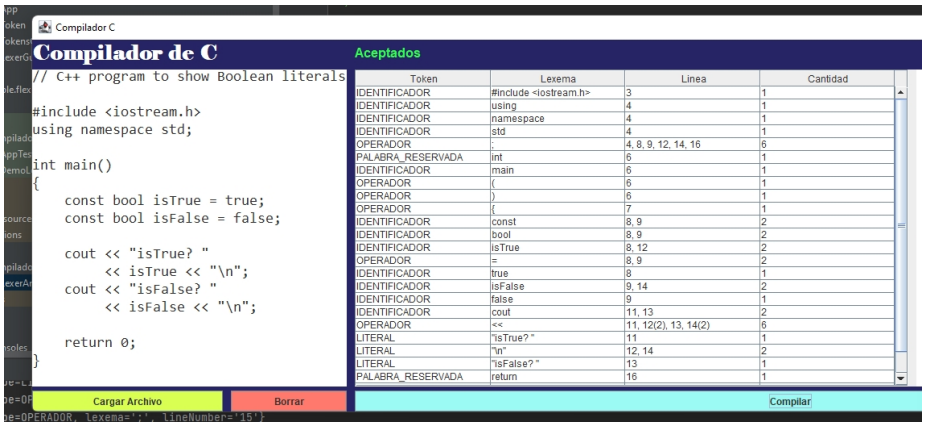

Figura 2.1: Comentarios: Caso de comentario de línea

El primer caso con respecto a los comentarios corresponde a los comentarios en línea 2.1. Dichos comentarios se demuestra no aparece ninguno de los Tokens que contienen, a la hora de mostrar el resultado.

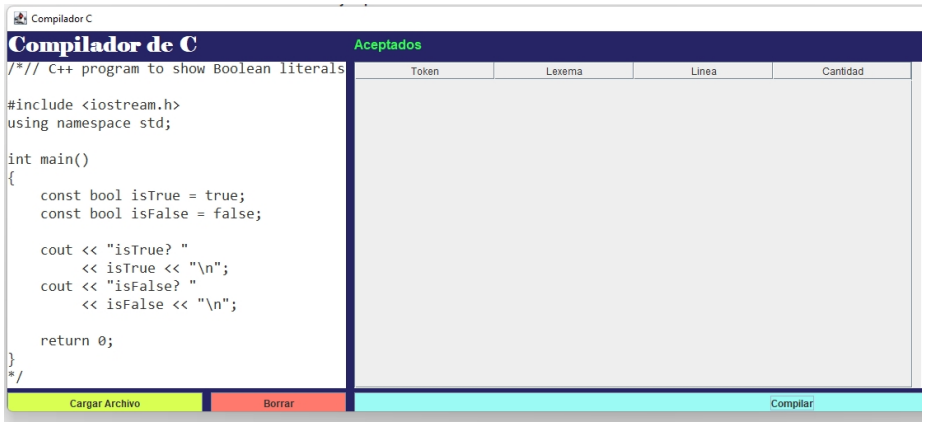

Figura 2.2: Comentarios: Caso de comentario de bloque

Este comportamiento se repite en el caso de los comentarios en bloque 2.2, donde al englobar todo el código en dichos comentarios, no genera ningún token de resultado.

El siguiente caso de uso corresponde a los identificadores 2.3. En ellos, se representan tanto las constantes, como variables, tipos de datos, funciones, entre otros aspectos.

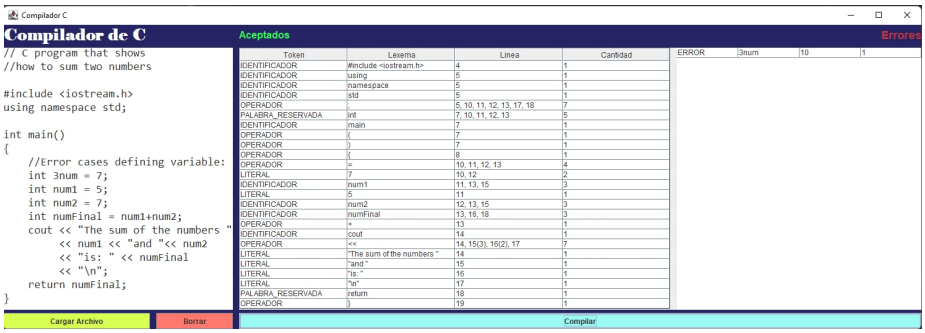

Figura 2.3: Identificadores

Como se ejemplifica en la imagen, los identificadores recuperan todos los Tokens relacionados a los tipos indicados. Así mismo, captura errores tales como por ejemplo, una variable empezando con un número, lo que se consdera algo erróneo para la lógica el Scanner.

Posteriormente, sigue el caso de uso correspondiente a las palabras reservadas 2.4.

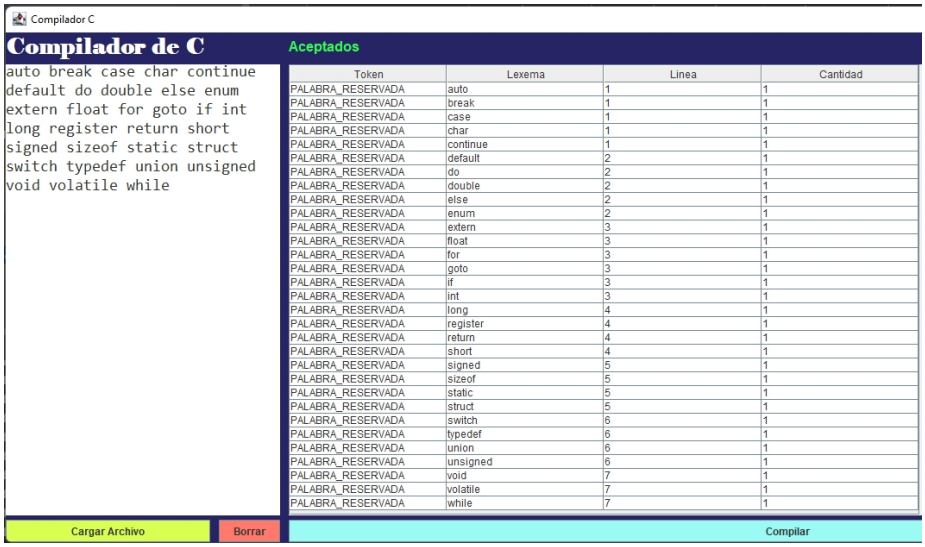

Figura 2.4: Palabras Reservadas

Cada una de las palabras reservadas indicadas son analizadas y comprendidas por el compilador. En la imagen se muestran los resultados, donde cada una de las distintas palabras reservadas es reconocida como una.

Siguiendo con el penúltimo área abarcada por los casos de uso, se encuentran las literales 2.5. Dichas literales abarcan tantos los números en base decimal, octal, hexadecimal, los números de punto flotante y con exponente y por último, los carácteres y strings.

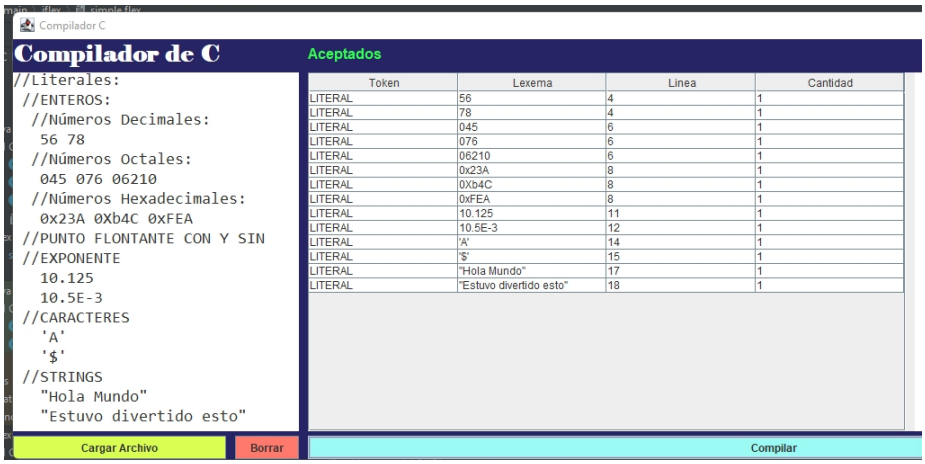

Figura 2.5: Literales

En el caso de uso se pueden evidenciar distintos ejemplos para cada uno de los tipos de literales. No obstante, en los casos tales como los puntos flotantes, pueden tener varias representaciones, por lo que se definieron las expresiones regulares respectivas.

Por último, se muestra el caso de uso correspondiente a los operadores 2.6. En él, se muestran cada uno de los operadores válidos que permite el lenguaje C.

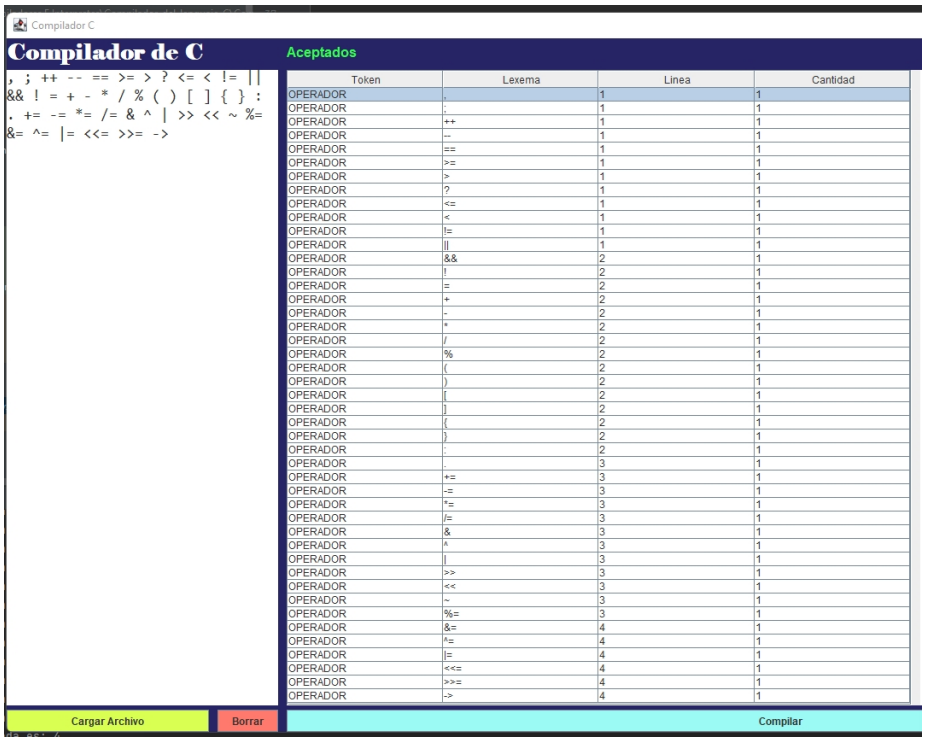

Figura 2.6: Operadores

Estos se evidencian que son reconocidos por separado cada uno. Cabe destacar que, para que se reconocieran todos, tuvo importancia el orden en que fueron definidos en el Lexer, ya que esto podría afectar y provocar que algunos operadores nunca fueran reconocidos, al encontrar una coincidencia previa que encajara con otro operador.

2.1. Código fuente

A continuación se mostrará el código fuente del archivo.flex con la última versión funcional

```
1  package CompiladorC;
2
3  import java.io.*;
4
5  %%
6  %public
7  %line
8  %class LexerAnalyzer
9
10 whitespace = [ \t\r]
11 newline =[\n]
12
13 //ENTEROS
14 Zero = 0
15 DecInt = [1-9][0-9]*
16 OctalInt = 0[0-7]+
17 HexInt = 0[xX][0-9a-fA-F]+
18 Integer = ( {HexInt} | {OctalInt} | {DecInt} | {Zero} ) [1L]?
19
20 //FLOTANTES
21 Exponent = [eE] [\+\-]? [0-9]+
22 Float1 = [0-9]+ \. [0-9]+ {Exponent}?
23 Float2 = \. [0-9]+ {Exponent}?
24 Float3 = [0-9]+ \. {Exponent}?
25 Float4 = [0-9]+ {Exponent}
26 Float = ( {Float1} | {Float2} | {Float3} | {Float4} ) [fFdD]? |
       [0-9]+ [fFDd]
27
28 //String
29 stringLit = \"[^\"]*\"
30
31 //Char
32 charLit = \'[^\']\'
33
34 //LITERALES
35 Literal = ({Float} | {Integer} | {stringLit} | {charLit})
36
37 //IDENTIFICADOR
38 Identificador = [A-Za-z][A-Za-z0-9]*
39 //DefMacro = <YYINITIAL> [^#]
40
41 Identificadores = ({Identificador})// | {DefMacro})
42
43 //PALABRAS PALABRA_RESERVADA
44 Palabras_Reservadas = ("auto" | "break" | "case" | "char" | "
       continue" | "default" | "do" | "double" |
45                            "else" | "enum" | "extern" | "float" | "
                                  for" | "goto" | "if" | "int" | "long"
                                   | "register" |
46                            "return" | "short" | "signed" | "sizeof"
                                  | "static" | "struct" | "switch" | "
                                  typedef" |
```

```
47                                  "union" | "unsigned" | "void" | "volatile
                                        " | "while")
48
49  //OPERADORES
50  Operadores = ("," | ";" | "++" | "--" | "==" | ">=" | ">" | "?" | "
        <=" | "<" | "!=" | "||" |
51                    "&&" | "!" | "=" | "+" | "-" | "*" | "/" | "%" | "
                          (" | ")" | "[" | "]" | "{" | "}" |
52                    ":" | "." | "+=" | "-=" | "*=" | "/=" | "&" | "^" |
                           "|" | ">>" | "<<" | "~" | "%=" |
53                    "&=" | "^=" | "|=" | "<<=" | ">>=" | "->")
54
55  //COMENTARIOS
56  Comentarios = ("/*"~"*/" | "//"[^\r\n]*)
57
58  //ERRORES
59  Error = [^]
60
61  %type Token
62  %eofval{
63
64      return new Token(TokensConstants.EOF, null, yyline);
65      /*Hacer algo al final del archivo*/
66
67  %eofval}
68  %%
69
70  //-------------------------------------COMENTARIOS
        ------------------------------------
71
72  {Comentarios} {System.out.println("Comentarios");}
73  //-------------------------------------ESPACIOS EN BLANCO
        ------------------------------------
74  {whitespace}+ {System.out.println("Espacio en blanco"); /*ignore*/}
75  {newline}+ {/*ignore*/}   //Ignorar saltos de linea
76  //-------------------------------------OPERADORES
        ------------------------------------
77  {Operadores} {System.out.println(new Token(TokensConstants.OPERADOR
        , yytext(), yyline).toString());
78            return new Token(TokensConstants.OPERADOR, yytext(),
                  yyline); }
79
80  //-------------------------------------LITERALES
        ------------------------------------
81  {Literal} {System.out.println(new Token(TokensConstants.LITERAL,
        yytext(), yyline).toString());
82            return new Token(TokensConstants.LITERAL, yytext(),
                  yyline); }
83
84  //-------------------------------------ERRORES
        ------------------------------------
85  <YYINITIAL> {Integer}+{Identificador} {System.out.println(new Token
        (TokensConstants.ERROR, yytext(), yyline).toString());
86            return new Token(TokensConstants.ERROR, yytext(), yyline)
                  ; }
87
```

```
88  {Palabras_Reservadas} {System.out.println("Palabras reservadas");
        return new Token(TokensConstants.PALABRA_RESERVADA, yytext(),
        yyline);}
89
90  //-------------------------------------IDENTIFICADORES
        ---------------------------------------
91  //Identificadores
92  {Identificadores} {System.out.println(new Token(TokensConstants.
        IDENTIFICADOR, yytext(), yyline).toString());
93              return new Token(TokensConstants.IDENTIFICADOR, yytext(),
                    yyline); }
94
95  //MACROS E IMPORTACIONES
96  <YYINITIAL> ^"#" ("include " ("<"{Identificador}".h>" | \"{
        Identificador}".h"\")| "define "{Identificador}" "({Literal} |
        {Operadores} | {whitespace})+)
97      {System.out.println(new Token(TokensConstants.IDENTIFICADOR,
            yytext(), yyline).toString());
98              return new Token(TokensConstants.IDENTIFICADOR, yytext(),
                    yyline); }
99
100 //-------------------------------------ERRORES
        ---------------------------------------
101 {Error} {System.out.println(new Token(TokensConstants.ERROR, yytext
        (), yyline).toString());
102             return new Token(TokensConstants.ERROR, yytext(), yyline)
                    ; }
```

Capítulo 3

Manual de Usuario

Este manual está planeado para alguien que no sabe cómo fue que se programó la solución y ya tiene una versión completa y estable del producto. Al abrir el programa, inicialmente verá una interfaz gráfica sencilla. En esta verá un espacio en blanco grande a la izquierda de la ventana. En este espacio puede escribir lo que desee (preferiblemente que dicho texto tenga tokens de C pues es un compilador de C). En el centro de la pantalla verá un espacio de color gris claro y este dice en la parte superior Aceptados. En la parte derecha de la pantalla verá un espacio que dice en la parte superior Errores. También verá tres botones: Cargar Archivo, Borrar y Compilar. Las funciones de estos se explican más adelante. Estos son todos los elementos que debe tomar en cuenta para iniciar a usar la aplicación.

Figura 3.1: Menú Principal

Para poner un texto para que este sea analizado hay dos opciones. La primera opción es escribir manualmente sobre el espacio en blanco a la izquierda seleccionando la casilla de área de texto.

La segunda opción es cargar un archivo. Para ello debe dar click en el botón Çargar Archivo". Esto le abrirá el explorador de archivos de su computadora y puede seleccionar algún documento de texto que tenga guardado.

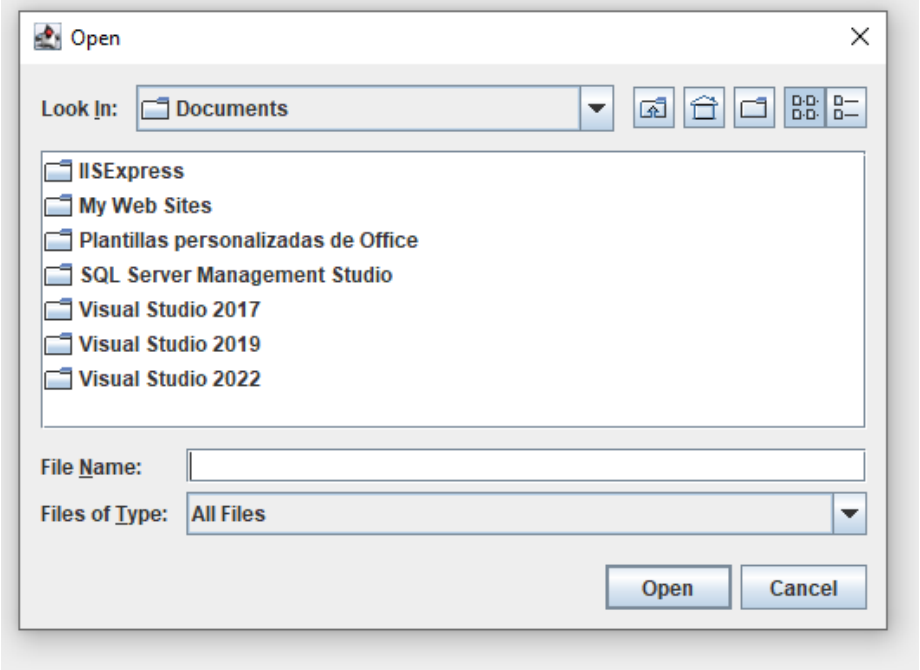

Figura 3.2: Explorador para buscar el archivo de texto

En caso de cargar un archivo, una vez que lo seleccione, todo el contenido de este se pondrá en el espacio para texto de la izquierda. Note que en ambos casos, escritura manual o con archivo, siempre va a quedar texto en el espacio destinado para este propósito. Una vez que ya haya texto, se puede proceder al proceso de compilación. Para ello debe presionar el botón Compilar.

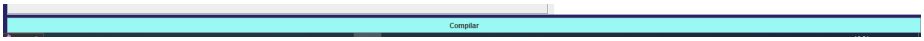

Figura 3.3: Botón compilar

Una vez que presione este botón aparecerán en el espacio de Aceptados todos los tokens encontrados en el texto e información sobre ellos. Se indica el tipo token, el lexema (el símbolo), la línea donde aparece y la cantidad total de apariciones. Si un token aparece en varias líneas distintas entonces se ponen todas los números de línea donde aparece separados por comas. Si un mismo

token aparece más de una vez en una misma línea entonces se pone el número de línea seguido por la cantidad de apariciones en esa línea entre paréntesis.

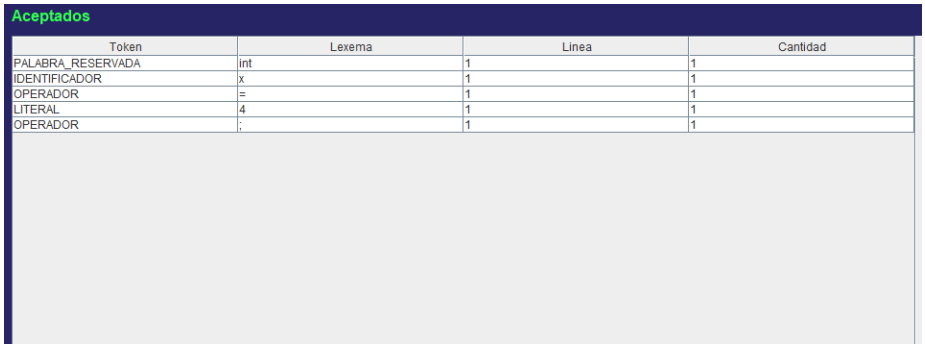

Figura 3.4: Gráfico de Tokens aceptados

Si a la hora de compilar también se detectan errores, estos se despliegan en el espacio a la derecha. Acá aparecerá cuál es el error, la línea donde aparece y la cantidad total de apariciones. Al igual que con los token, si un mismo error aparece en varias líneas se pone la lista de números de líneas y si el error aparece más de una vez en la misma línea se pone el número de línea y entre paréntesis la cantidad de apariciones en esta.

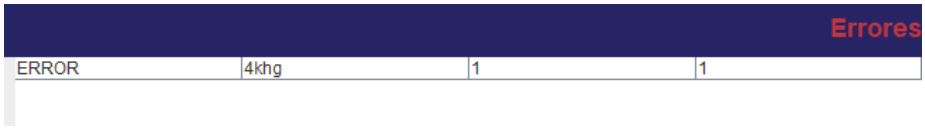

Figura 3.5: Errores

En resumen, al dar çlick.[en] el botón Compilar se ejecuta la revisión léxica del texto que se tenga en ese momento. Esto va a desplegar todos los tokens y errores encontrados con su información respectiva. En caso de que no haya texto y se aprete Compilar, no va a pasar nada. También se tiene el botón Borrar. Este botón resetea la sección de Aceptados y Errores.

Ejemplo de uso:

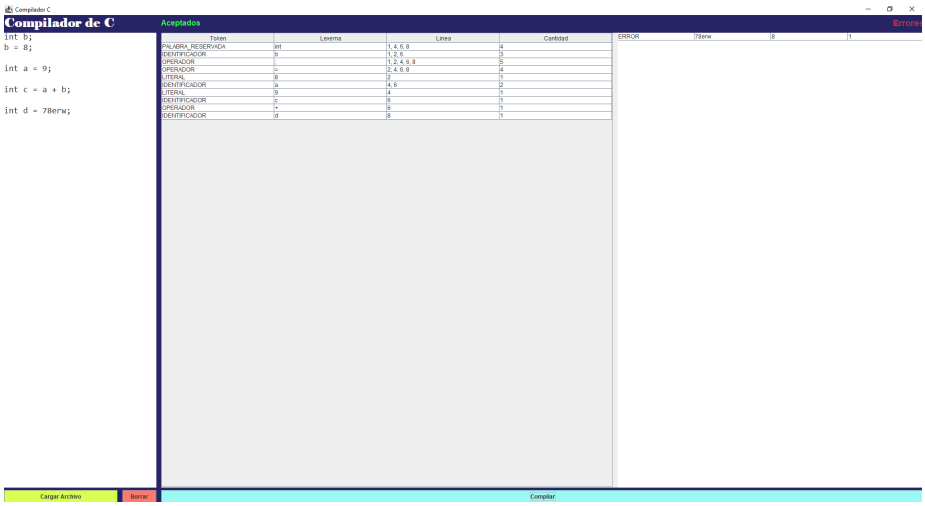

Figura 3.6: Ejemplo de uso del programa

Capítulo 4

Bitácora

4.1. Actividades

A continuación se muestra una tabla con las actividades realizadas durante el desarrollo de este proyecto, las fechas corresponden a los días que se hizo reunión; no obstante, se excluye el tiempo utilizados para la búsqueda de información y referencias.

Actividades		
Fecha	Actividad	Detalles relevantes
19/09/20220	Configuramos el espacio de trabajo y nos organizamos con la búsqueda de información	Decidimos usar herramientas como Intellij IDEA, Github, google docs para ingresar links de ayuda e información útil
17/09/2022	Creamos el proyecto y agregamos las dependencias necesarias , además instalamos el JFlex para comenzar a trabajar	Creamos algunas clases como el enum de los tipos de Tokens y la clase Token
24/9/2022	Comenzamos a modificar el JFlex y hacer algunas pruebas de funcionamiento	Notamos que la clase simple.flex generaba el archivo LexerAnalizer que se sobreescribía con cada modificación en el simple.flex.
25/09/2002	Completamos el analizador y empezamos a hacer una interfaz gráfica para ejecutar el scanner de una forma visual	El analizador tenía un buen funcionamiento; sin embargo, debemos solucionar algunos errores acerca de la lectura de Tokens.
27/09/2022	Nos reunimos de forma presencial en el learning Common para finiquitar todos los errores del scanner y dejar la interfaz trabajando de forma correcta junto con el scanner.	El scanner funciona de forma exitosa y detecta sus tokens correspondientes junto con la cantidad y posiciones de aparición.

Capítulo 5

Conclusión

5.1. Análisis de resultados

- Utilización de JFlex para analisis lexico: 100 %
  Se utilizó la herramienta JFlex en el lenguaje de programación Java por completo.
- Definición de palabras reservadas: 100 %

  Se definieron las palabaras reservadas del lenguaje de programación C.
- Definición de literales: 100 %

  Identifica correctamente los literales del texto.
- Definición de operadores: 100 %

  Identifica correctamente los operadores del texto.
- Definición de identificadores: 100 %

  Identifica correctamente los identificadores, asimismo señala como error los ïdentificadores"que comiencen con un número.
- Definición de expresiones regulares que representan el lenguaje regular C: 100 %

  Las expresiones regulares que conforman el archivo "simple.flex"funcionan de forma correcta.
- Leer texto: 100 %

El programa permite cargar texto desde un archivo txt, asímismo hacer pruebas con tokens ingresados por el usuario directamente en la interfaz gráfica.

- Identificar lexema del token: 100 %

  Permite obtener el lexema de cada Token.

- Identificar tipo de token: 100 %

  Permite obtener el tipo del token, el tipo pertece a una constante definida en la enumeración "TokensConstants".

- Identificar apariciones de cada token: 100 %

  El programa reconoce la cantidad de veces que aparece cada token o error a lo largo del texto.

- Identificar si un token aparece varias veces en una línea o si aparece en varias líneas: 100 %

  El programa permite obtener el índice de cada línea donde aparece un token determinado e identifica cuántas veces por línea aparece dicho token.

- Identificar errores: 100 %

  Detecta los errores de una forma correcta sin detener el programa. Es decir, al encontrar un error el programa termina de identificar cada uno de los tokens del texto.

- Brindar información sobre los errores: 100 %
  Identifica la línea de aparición de cada uno de los errores y el lexema del error.

- Documentación: 100 %
  El documento tiene todas las especificaciones solicitadas en el enunciado del proyecto.

5.2. Lecciones Aprendidas

Entre todos los miembros del equipo decidimos generalizar las lecciones aprendidas debido a que este proyecto enriqueció la participación grupal, de esta manera tenemos todos el mismo criterio de aprendizaje tanto blando como ténico. Se iniciará describiendo el enriquecimiento en el aspecto interpersonal. Después se comentará los beneficios obtenidos en el área técnica.

Respecto con las habilidades blandas notamos que fue necesario la participación en el proyecto presencial, a pesar de que la mayoría del proyecto fue realizado de forma virtual utilizando la plataforma de Discord, la última reunión la decidimos hacerla de forma presencial en un cubículo del learning Common del TEC, esto dejó una notoria mejora en el rendimiento del trabajo en equipo de forma positiva. La forma de resolver errores fue significativamente más rápida. De esta manera fortalecieron las habilidades blandas que son vitales para ser ingenieros eficientes. Afortunadamente no se tuvieron problemas a lo interno del grupo. Solo se fortalecieron habilidades como la cooperación, trabajo en equipo y organización de tareas. Estas son características muy importantes para satisfacer el mercado laboral.

Por otro lado, este proyecto fue especialmente beneficioso para reforzar los conceptos vistos en clase. todos los miembros pudimos aplicar los conceptos vistos en clase. El hecho de poder aplicar estos conceptos de manera práctica permitió ver cómo se integran entre sí y qué aplicación tienen en un caso real. Adicionalmente, se pudo practicar intensivamente el tema de expresiones regulares. Esto mejoró el entendimiento de operaciones como la concatenación, unión y estrella. También el proyecto sirvió de demostración de cómo las expresiones regulares describen lenguajes regulares.

En el caso del aprendizaje con el uso de la herramienta JFlex, lo cuál fue una herramienta novedosa para todos los miembros de nuestro equipo. Aprendimos que la herramienta nos facilita el trabajo de generar un programa compilador basado en las expresiones regulares necesarias para formar los tokens que deseamos. Asimismo, notamos que el orden en el que el analizador recibía los tokens alteraba en deicidir un tipo de token; a lo largo del proyecto tuvimos varios inconvenientes con este problema ya que colocábamos las expresiones regulares en el orden incorrecto.

En términos generales la comunicación y trabajo en equipo siempre fue presente ya que nos apoyamos de buenas herramientas que impulsaban esta práctica como el "live share"del entorno de desarrollo de Intellij lo cuál nos facilitó el trabajo y decidimos implementar esta herramienta para los próximos proyectos programados.

5.3. Link del repositorio en Github

`https://github.com/geralm/Compilador-del-lenguaje-C.git`

Referencias

Louden and Kenneth C., Çonstrucción De Compiladores",Cengage Learning Editores S.A. de C.V, Enero 2004, México City, México.

The University of Auckland - New Zealand, Çhapter 1 Lexical Analysis Using JFlex", COMPSCI 330 S1 C - Lectures, University of Auckland, s.f., Nueva Zelanda.

Gerwin K., Steve R. and Régis D., JFlex User Manual, JFlex. Recuperado de https://jflex.de/manual.html.