UACM
Universidad Autónoma de la Ciudad de México
Nada humano me es ajeno

COLEGIO DE CIENCIA Y TECNOLOGÍA

LICENCIATURA EN INGENIERÍA DE SOFTWARE

Diseño y desarrollo de un software integrado (IDE, compilador y máquina virtual), para la enseñanza de la lógica de programación a partir de edades de 15 años, por medio de la creación propia de un lenguaje de programación de rutas instruccionales

T E S I S

QUE PARA OPTAR POR EL TÍTULO DE

LICENCIADO EN INGENIERÍA DE SOFTWARE

P R E S E N T A :

JONATHAN GARCÍA GIL

D I R E C T O R

MTRO. ADALBERTO ROBLES VALADEZ

Ciudad de México, mayo de 2023.

UNIVERSIDAD AUTÓNOMA DE LA CIUDAD DE MÉXICO

COLEGIO DE CIENCIA Y TECNOLOGÍA

ACADEMIA DE INFORMÁTICA

Licenciatura en Ingeniería de Software

“ APÉNDICE 1: DESARROLLO DEL ENTORNO DE DESARROLLO INTEGRADO”

TESIS

QUE PARA OPTAR POR EL TÍTULO DE LICENCIATURA EN:

INGENIERÍA DE SOFTWARE

PRESENTA:
JONATHAN GARCÍA GIL

DIRECTOR DE TESIS:
Mtro. Adalberto Robles Valadez
Profesor-Investigador de la Academia de Informática, UACM

Ciudad de México, mayo de 2023

Contenido

1 La interfaz del proyecto

Antes de continuar al Analizador Léxico me gustaría crear la interfaz de usuario del programa. Esta interfaz será la principal del proyecto en sí.

La idea de trabajar sobre la interfaz misma del proyecto es que conforme se avance en el proyecto no se tenga que regresar para integrarlo al proyecto final.

Para este capítulo solo se va a crear un prototipo funcional de lo que terminará siendo el proyecto final.

Para este proyecto se utilizará el software “IntelliJ IDEA Edu” en su versión 2022.3.1.

Enlace 1 Última versión de IntelliJ IDEA Edu https://www.jetbrains.com/edu-products/download/#section=idea	

Igualmente se debe tener a la mano 2 archivos JAR que pertenecen a un framework a utilizar, estos se pueden descargar de las siguientes direcciones:

Enlace 2 Última versión gratuita del framework JGoodies Forms https://mvnrepository.com/artifact/com.jgoodies/jgoodies-forms/1.9.0	

Enlace 3 Última versión gratuita del framework JGoodies Common https://mvnrepository.com/artifact/com.jgoodies/jgoodies-common/1.8.1	

1.1 Creando el Proyecto

El primer paso es crear el proyecto, para esto se tiene que seguir los siguientes pasos:

1. Hacer clic en “New Project”

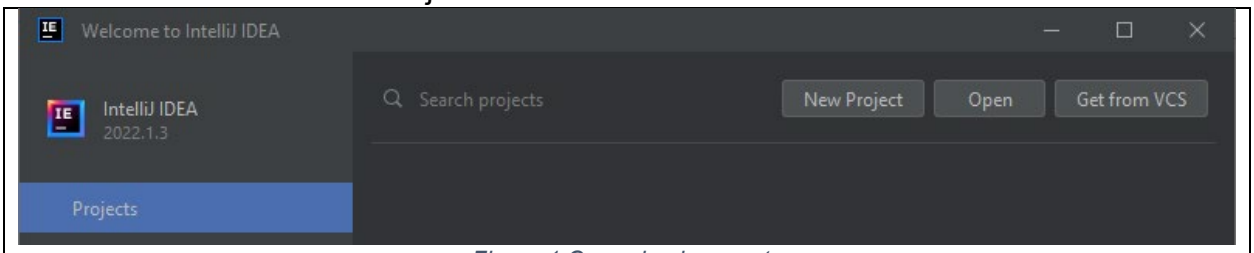

Figura 1 Creando el proyecto

2. Se nombra al proyecto, para este caso se llamará “Tesis”.

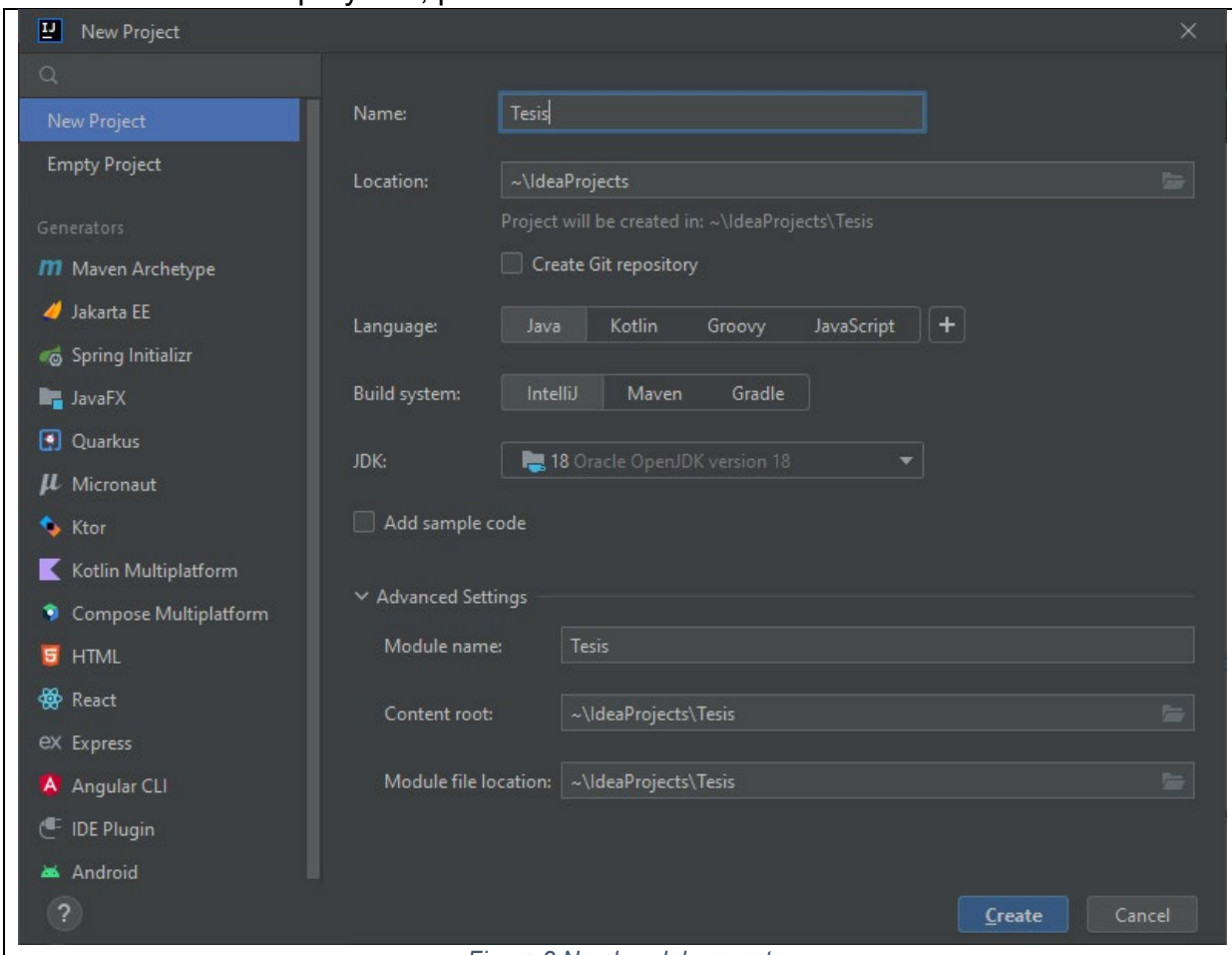

Figura 2 Nombre del proyecto

3. ¡El proyecto ha sido creado!

1.2 Creación de la Interfaz de Usuario

El siguiente paso, es generar lo que será la interfaz de usuario.

El primer paso es hacer clic derecho en la carpeta “src” del proyecto y crear un nuevo “package”. El nombre del “package” será “InterfazDeUsuario”.

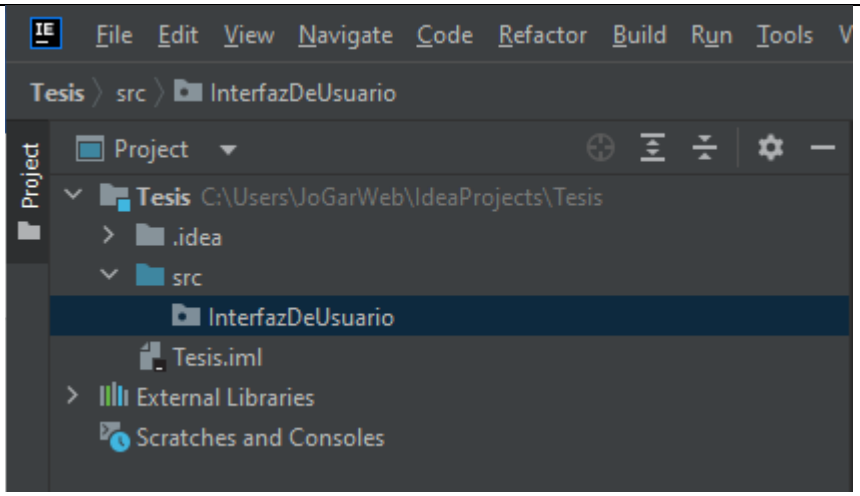

Figura 3 Raíz del Proyecto

El siguiente paso es crear la interfaz de usuario, para esto hay que hacer clic derecho sobre la carpeta que recién se creó (InterfazDeUsuario), después se despliega el menú “New”, se vuelve a desplegar el menú “Swing UI Designer” y seleccione la opción “GUI Form”.

Solicitará el nombre de la ventana, para este proyecto se llamará “Fenix”. Por el momento, este será el nombre clave del proyecto.

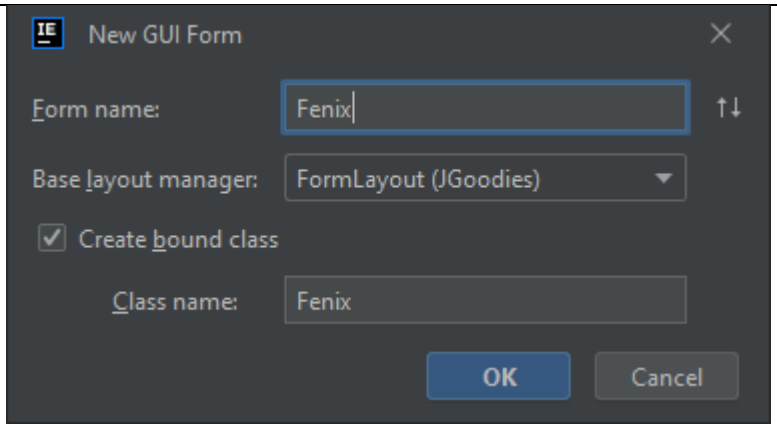

Figura 4 Opciones del Proyecto

Para este punto, se podrá observar la interfaz del programa para crear la ventana. A partir de aquí, se tratará de recrear el prototipo de programa, añadiendo un espacio más para el log del programa.

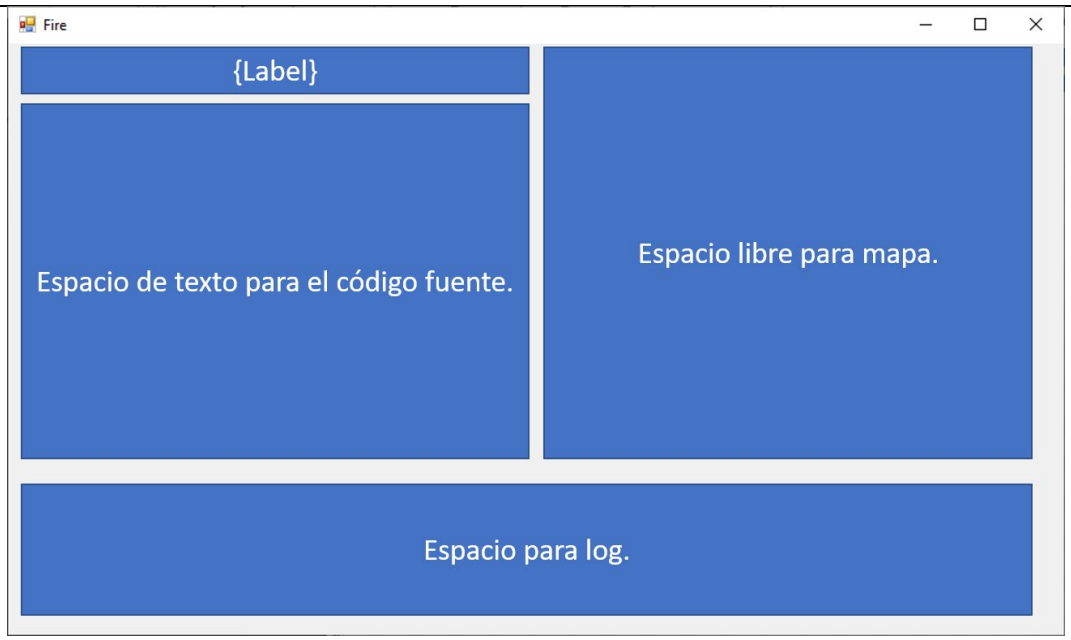

Figura 5 Prototipo del Proyecto

Una vez creados y nombrados, quedará algo similar a esto:

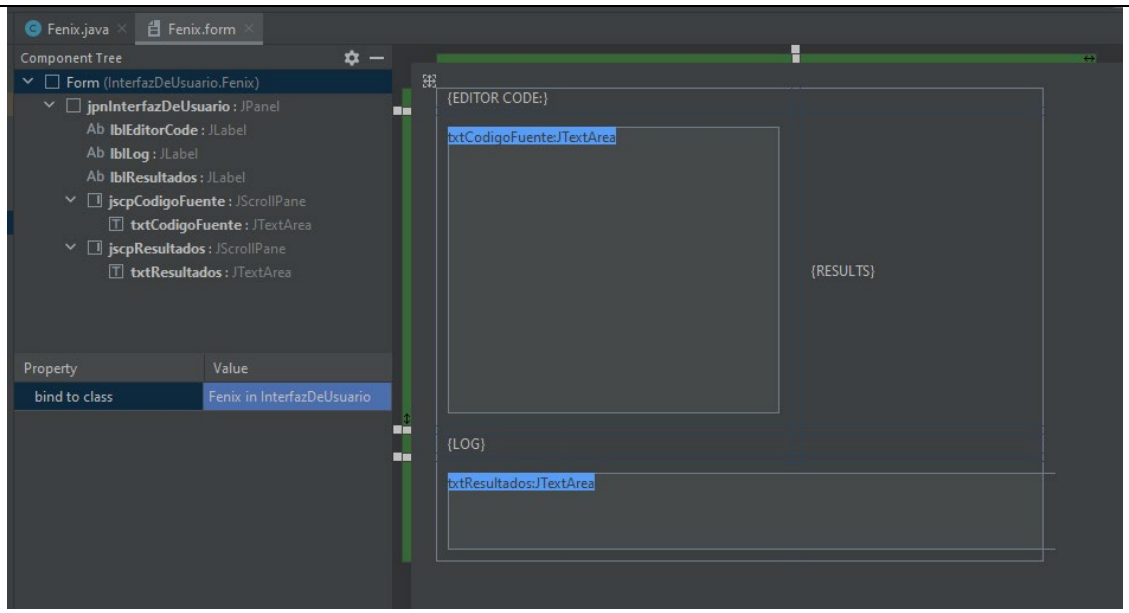

Figura 6 IDE del Proyecto

El siguiente paso es modificar el código de Fenix.java agregándole a la clase la palabra “extends JFrame”.

```
1.  public class Fenix extends JFrame{
```

Después, se crea un constructor en la clase, se agrega el título del programa con la palabra “super” y se específica cuál será el contenedor de la ventana. Quedando de la siguiente manera:

```
1.      public Fenix(){
2.          super("Fenix");
3.          setContentPane(InterfazDeUsuario);
4.      }
```

Después se creará la clase principal del programa. Para esto se volverá a crear un “package” en este caso con el nombre “Main”, dentro de ese nuevo “Package” se creará una nueva clase con el mismo nombre. Al final, se creará la función principal del programa.

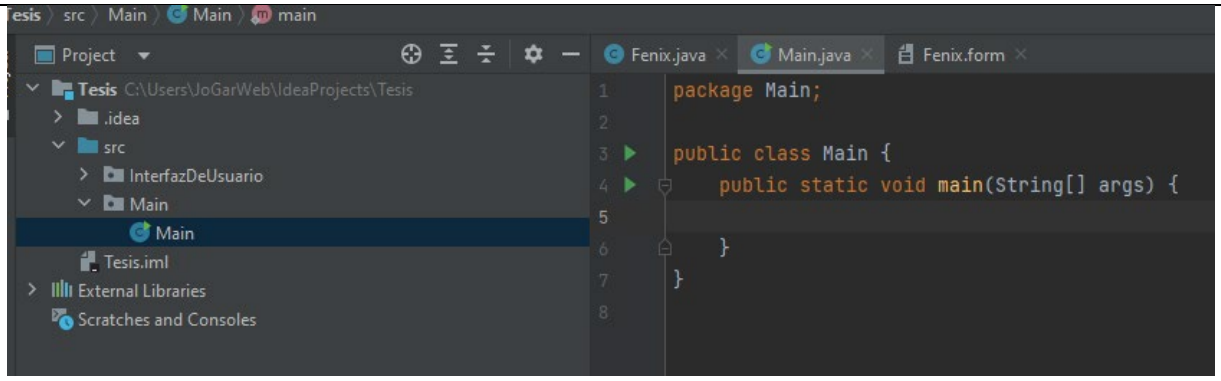

Figura 7 Función principal del proyecto

Se agregan las propiedades para abrir el programa desde el método principal. Quedará de la siguiente manera:

```
1.  public class Main {
2.      public static void main(String[] args) {
3.
4.          SwingUtilities.invokeLater(new Runnable() {
5.              @Override
6.              public void run() {
7.                  JFrame frame = new Fenix();
8.                  frame.setSize(500,400);
9.                  frame.setVisible(true);
10.             }
11.         });
12.
13.     }
14. }
15.
```

Debido a que se está utilizando un framework llamado “JGoodies” es obligado importar las librerías. Para esto en la raíz del proyecto se creará un nuevo módulo llamado “lib” que es el lugar dónde se encontrarán las librerías. Después de crearla, en la carpeta “src” se crea un “package” llamado “JGoodies”.

Dentro de esa carpeta se descargan los archivos JAR y los se colocan en esa carpeta.

Después de hacer lo anterior se debe ir a “File”, “Project Structure” y en el apartado “Libraries” se agregan las 2 librerías Java.

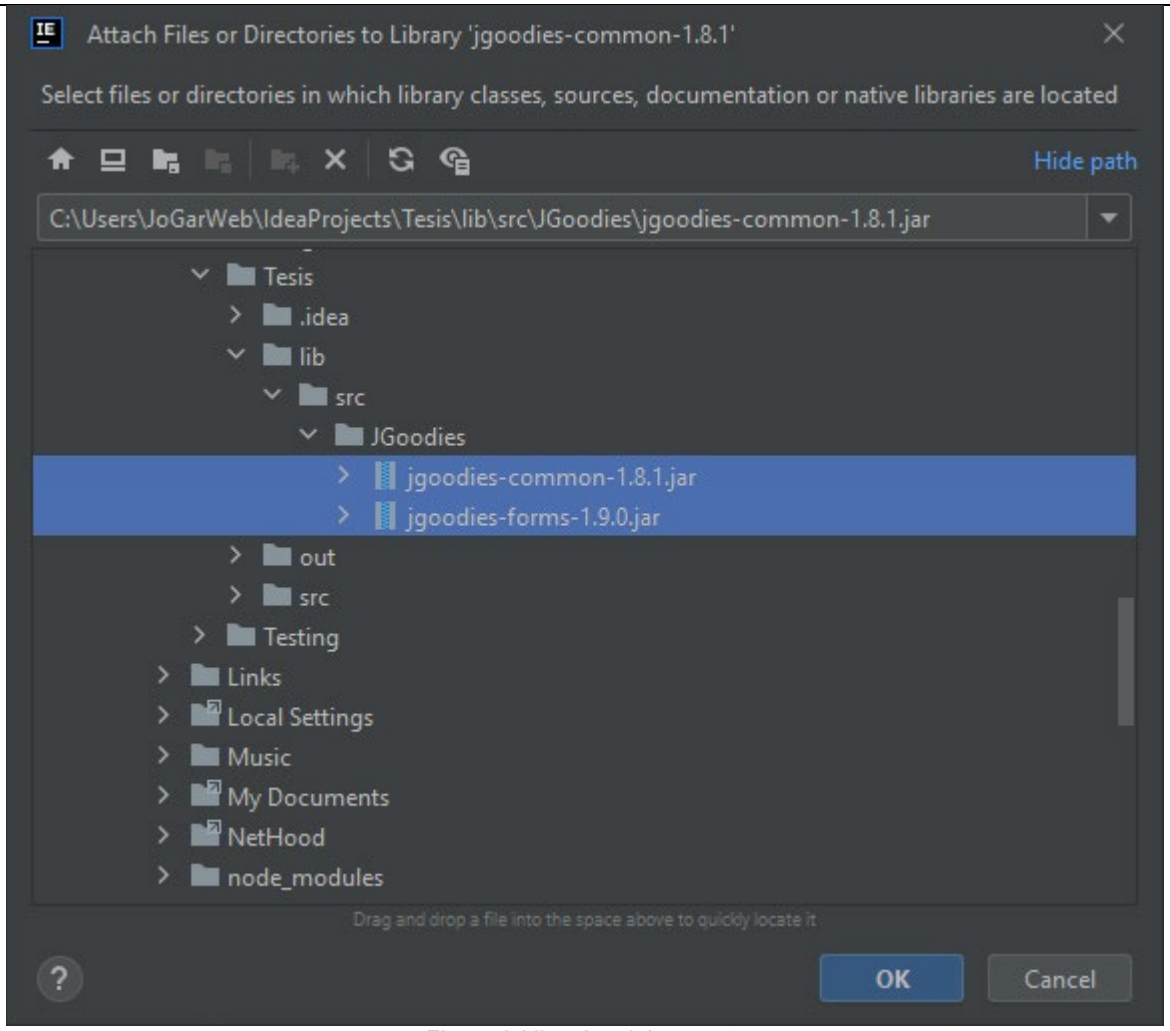

Figura 8 Librerías del proyecto

Para finalizar con la importación de librerías, haciendo clic en “OK”.

Ahora se puede ejecutar el programa como esta, para ejecutarlo hay que ir al menú “Run” y se vuelve a hacer clic en “Run”.

Si es la primera vez que se ejecuta, pedirá el método a ejecutar, en este caso será “Main”

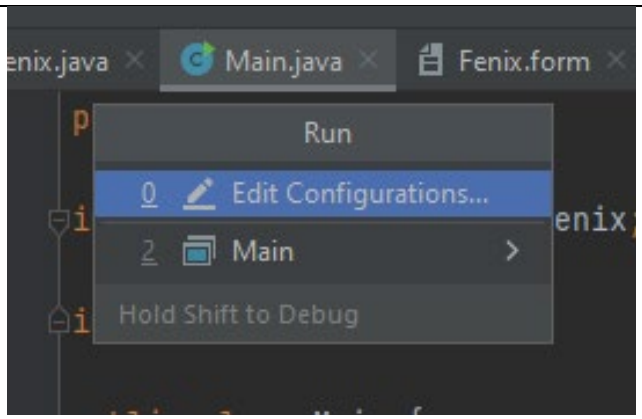

Figura 9 Ejecutar el proyecto

Y el programa abrirá por primera vez:

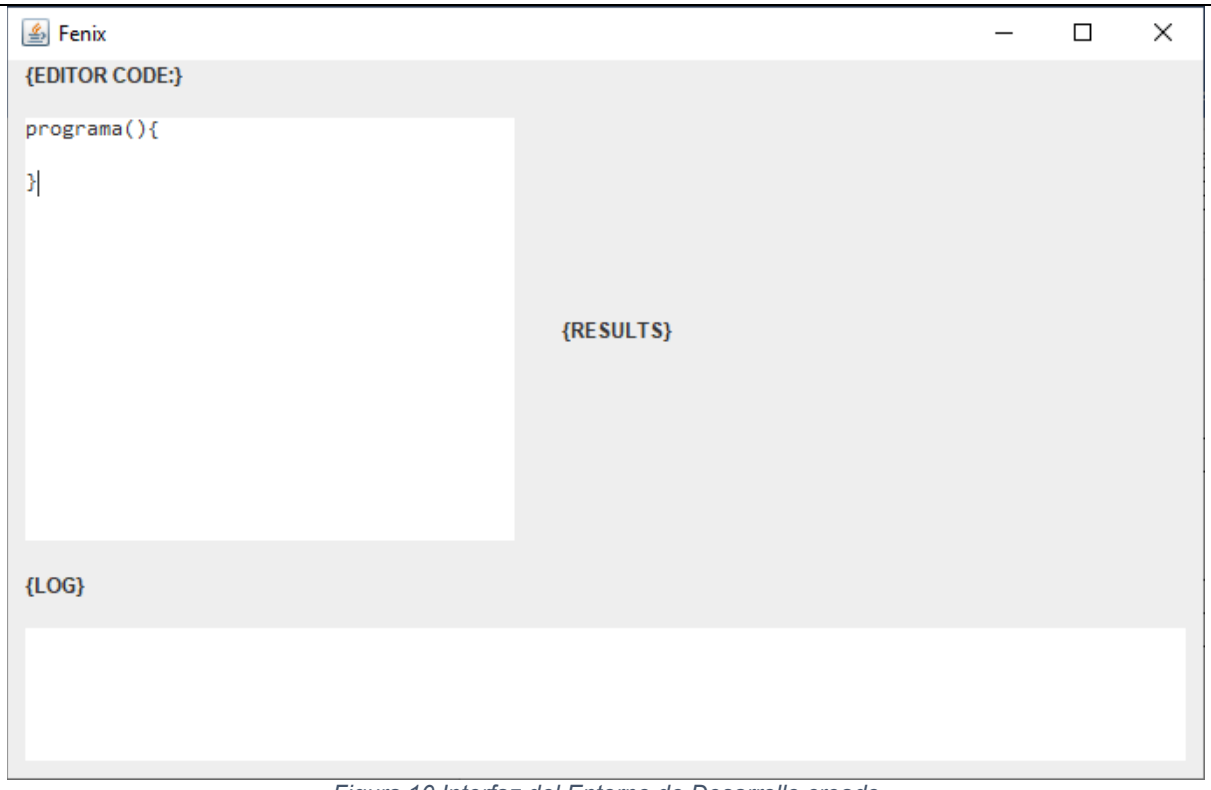

Figura 10 Interfaz del Entorno de Desarrollo creado

1.3 Creación de Menús Contextuales

Para estética de este proyecto se agregará el elemento MenuBar de forma manual.

Se crean estos objetos de forma global para que cualquier método que se genere lo puedan utilizar, quedarán más o menos de esta forma:

```
1.  //Declaración de MenuBar
2.      private JMenuBar menuBar;
3.
4.      //Elementos principales de MenuBar
5.      private JMenu archivo;
6.      private JMenu codigo;
7.      private JMenu editar;
8.      private JMenu compilador;
9.      private JMenu acercaDe;
10.
11.     //Submenus de MenuBar de Archivo
12.     private JMenuItem abrirMapa;
13.     private JMenuItem abrirCodigo;
14.
15.     //Submenus de MenuBar de codigo
16.     private JMenuItem compilar;
17.     private JMenuItem ejecutar;
18.     private JMenuItem compilarYEjecutar;
19.
20.     //Submenus de MenuBar de compilador
21.     private JMenuItem recompilarLexico;
22.     private JMenuItem recompilarSintactico;
23.     private JMenuItem recompilarTodo;
24.
```

Después, dentro del método Fenix() se inicializarán los objetos que recién se crearon

```
1.  //Inicializando los objetos creados
2.          menuBar = new JMenuBar();
3.
4.          archivo = new JMenu("{ARCHIVO}");
5.          codigo = new JMenu("{CODIGO}");
6.          editar = new JMenu("{EDITAR}");
7.          compilador = new JMenu("{COMPILADOR}");
8.          acercaDe = new JMenu("{ACERCA DE}");
9.
10.         abrirMapa = new JMenuItem("{ABRIR MAPA}");
11.         abrirCodigo = new JMenuItem("{ABRIR CODIGO}");
12.
13.         compilar = new JMenuItem("{COMPILAR}");
14.         ejecutar = new JMenuItem("{EJECUTAR}");
15.         compilarYEjecutar = new JMenuItem("{COMPILAR Y EJECUTAR}");
16.
17.         recompilarLexico = new JMenuItem("{RECOMPILAR ANALIZADOR LÉXICO}");
18.         recompilarSintactico = new JMenuItem("{RECOMPILAR ANALIZADOR SINTÁCTICO}");
19.         recompilarTodo = new JMenuItem("{RECOMPILAR ANALIZADORES}");
20.
```

Finalmente, se construye el MenuBar y se muestran.

```
1.  //Construyendo el MenuBar
2.          menuBar.add(archivo);
3.          menuBar.add(codigo);
4.          menuBar.add(editar);
5.          menuBar.add(compilador);
6.          menuBar.add(acercaDe);
7.  //Construyendo el MenuBar
8.          archivo.add(abrirMapa);
9.          archivo.add(abrirCodigo);
10.
11.         codigo.add(compilar);
12.         codigo.add(ejecutar);
13.         codigo.add(compilarYEjecutar);
14.
15.         compilador.add(recompilarLexico);
16.         compilador.add(recompilarSintactico);
17.         compilador.add(recompilarTodo);
18.
19.         //Mostrando el MenuBar
20.         setJMenuBar(menuBar);
21.
```

Al final el programa se visualizará de la siguiente manera:

Figura 11 Interfaz de usuario del Entorno de Desarrollo

Este será el prototipo (el cascarón) del programa, a partir de este prototipo se trabajará con los siguientes pasos.

Enlace 4 Código fuente del prototipo https://github.com/JonathanGarcia15/tesis-01-primer-prototipo-de-un-compilador	

2 JFlex y otras librerías

JFlex es la herramienta que ayudará a generar el Analizador Léxico. La versión que utilizará este proyecto es la versión 1.7.0

Enlace 5 La última versión de JFlex https://jflex.de/download.html	

El siguiente paso es guardar la librería de JFlex en el proyecto, para esto se crean un nuevo package en el módulo de librerías con el nombre “JFlex” y ahí se coloca la librería de JFlex.

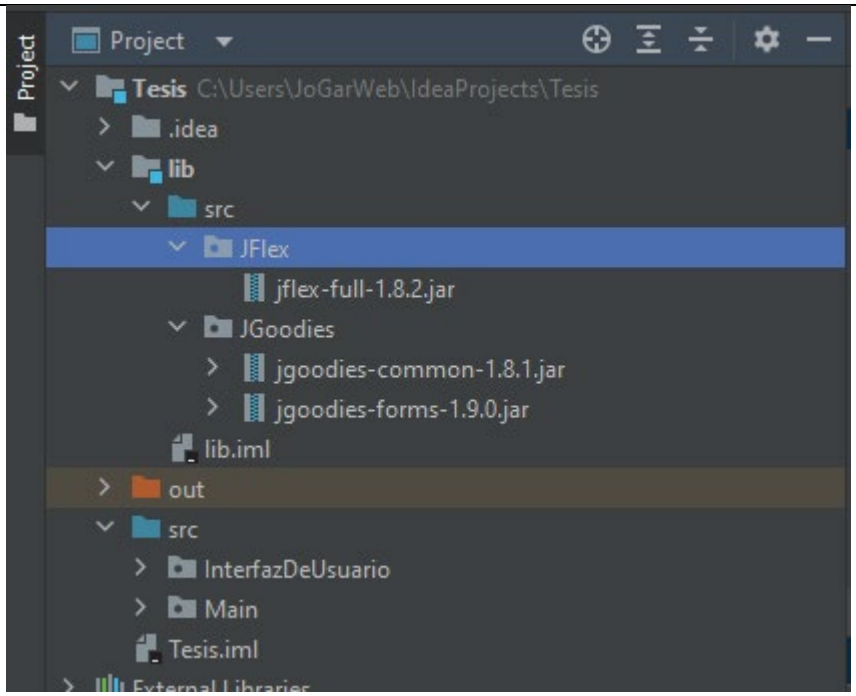

Figura 12 Librerías

2.1 Últimas modificaciones al proyecto

Algo que olvidé configurar en el diseño del prototipo del Capítulo 1 de este documento, son el resto de librerías que se utilizarán. Para este punto se utilizarán las siguientes más. Por lo que es necesario colocarlas en el respectivo proyecto. Las librerías a descargar y configurar son las siguientes:

Enlace 6 Java CUP (Analizador Sintáctico) Versión 11b (20160615) https://mvnrepository.com/artifact/com.github.vbmacher/Java-CUP/11b-20160615	
Enlace 7 JSON para Java en la versión 20220320 https://mvnrepository.com/artifact/org.json/json/20220320 (Descargar json-20220320.jar y json-20220320-Javadoc.jar)	

Al final en el proyecto lucirán las librerías de la siguiente manera:

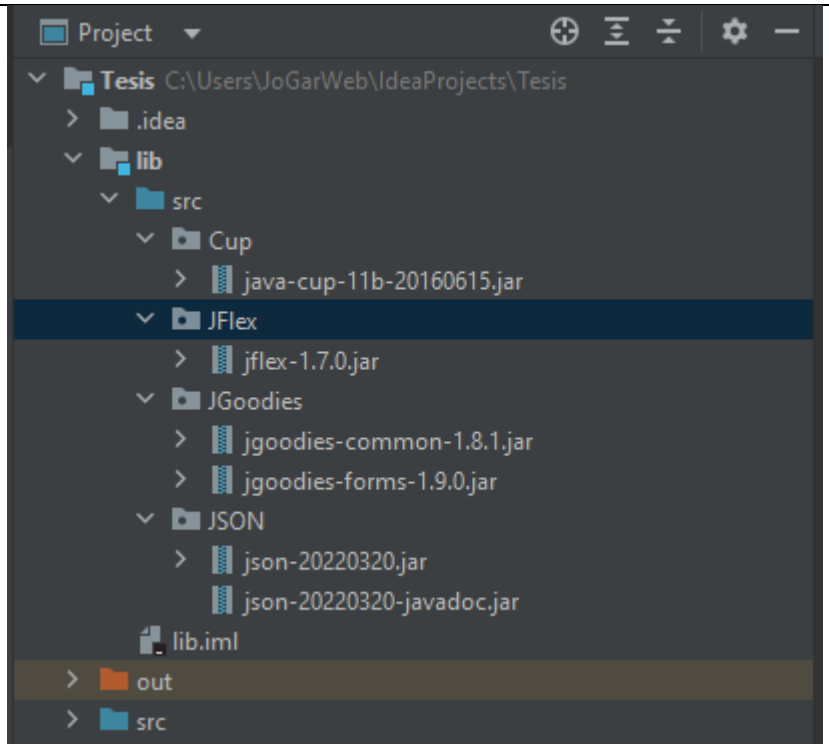

Figura 13 Librerías

3 Preparando el Entorno de Trabajo

Antes de continuar, se tiene que modificar el proyecto con los archivos que se ocuparán a lo largo del proyecto:

- Se crea un nuevo package llamado “Compilador” en la carpeta de “*src*”.
- Dentro de ese package se debe crear una clase que se llamará “*CompiladorLenguajeUsuario*”.
- Dentro del package de “Compilador” se crea otro package llamado “*Default*”.
- Después de crear el último package, se crea otro llamado “*CodigoFuente*” y “*AnalizadorLexico*”.
- Dentro del package “*AnalizadorLexico*” se crean 2 clases que se llamarán “*AnalizarLexicoDefault*” y “*Tokens*”
- Finalmente, dentro del package “*CodigoFuente*” se crea un archivo llamado: “*Lexer.flex*”.

El proyecto deberá tener los siguientes archivos en el siguiente orden:

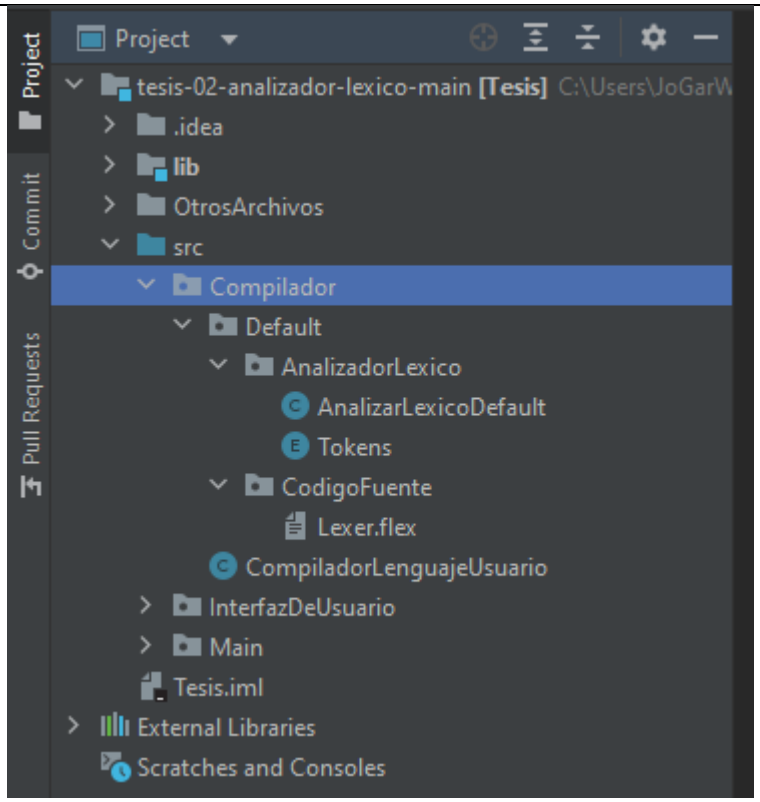

Figura 14 Directorio del proyecto

4 Modificaciones para generar el Analizador Léxico

4.1 ¿Cómo generar el Analizador Léxico desde el proyecto?

Hasta este punto se ha terminado con definir el Analizador Léxico, sin embargo, no se ha compilado con JFlex para que pueda generar el Analizador.

La idea de este capítulo es ejecutar JFlex y generar los archivos que se requieren generar desde la interfaz del proyecto. Vaya, en lugar de hacerlo por separado se generará desde el proyecto.

El usuario final no tendrá acceso a este bloque de código, así que al final de este proyecto estas opciones se deberán deshabilitar.

A partir de este momento, solo se habla de cosas estéticas para el proyecto. Pero que son necesarias para lo que se busca crear.

4.2 La clase para controlar la generación del Analizador Léxico

Para comenzar, se partirá de una clase con la que se pueda ejecutar JFlex para generar el Analizador Léxico. Esta clase será "*CompiladorLenguajeUsuario.Java*" que se crearon anteriormente.

Primero se requiere la dirección del proyecto, para esto se crea objeto de tipo *String* de nombre "*DireccionProyecto*" que indicará exactamente la dirección dónde está colocado el proyecto. En lugar de escribir la dirección de forma manual se puede utilizar el método "*System.getProperty()*", igualmente necesita reemplazar el símbolo '\' por '/'. Todo se puede hacer con una sola línea, y así quedará:

```
1.  private final String DireccionProyecto = System.getProperty("user.dir").replace("\\", "/");
```

El siguiente paso es crear un método que se pueda llamar para ejecutar los comandos necesarios para compilar en JFlex. Para este proyecto intentaré aplicar las mejores técnicas de programación, las más que pueda y pueda ver a simple vista. Por lo que separaré de la clase pública las instrucciones para generar el Analizador Léxico, con el objetivo de que las variables que utilice el método no se puedan acceder desde otro lugar en el proyecto.

```
1.      public void GenerarAnalizadorLexico(){
2.          this.GenerarLexicoDefault();
3.      }
4.
5.      private void GenerarLexicoDefault() {
6.          «CÓDIGO PARA GENERAR EL ANALIZADOR LÉXICO»
7.      }
8.
```

Antes, se tiene que crear un método que sea capaz de mover un archivo ".Java" de la ruta "*/src/Compilador/Default/CodigoFuente/*" a la ruta "*/src/Compilador/Default/AnalizadorLexico/*". La cual se podrá realizar con este método:

```
1.      private void moverArchivos(String rutaDelArchivoExistente, String rutaNuevaDelArchivo){
2.          Path path = Paths.get(DireccionProyecto + rutaNuevaDelArchivo);
3.          //Eliminando si existe un archivo previo
4.          if (Files.exists(path)) {
5.              try {
6.                  Files.delete(path);
7.              } catch (IOException ex) {
8.
9.              }
10.         }
11.
12.         try {
13.             Files.move(
14.                     Paths.get(DireccionProyecto+rutaDelArchivoExistente),
15.                     path
16.             );
17.         } catch (IOException ex) {
18.
19.         }
20.     }
21.
```

Lo único que queda por realizar es definir la ruta dónde se encuentra el archivo "*Lexer.flex*", este se encuentra en la ruta: *"/src/Compilador/Default/CodigoFuente/"* del proyecto.

Se utilizará la clase *File* para abrir el archivo, y se ejecutará la herramienta JFlex. Y por último se nombra al método que se creó anteriormente.

Al final, el código para generar el Analizador Léxico se resuelve en menos de 15 líneas de código.

```
1.      private void GenerarLexicoDefault() {
2.
3.          //Ruta de los archivos
4.          String            rutaLexico            =            DireccionProyecto            +
    "/src/Compilador/Default/CodigoFuente/Lexer.flex";
5.
6.          File archivo;
7.
8.          //Para leer Lexer.flex y generar el Analizador Léxico
9.          archivo = new File(rutaLexico);
10.         jflex.Main.generate(archivo);
11.
12.         //Moviendo el archivo Lexer.Java
13.         this.moverArchivos(               "/src/Compilador/Default/CodigoFuente/Lexer.Java",
    "/src/Compilador/Default/AnalizadorLexico/Lexer.Java");
14.
15.     }
16.
```

Para conocer el resultado final de esta clase, nuevamente pido que se abra la siguiente dirección web:

Enlace 8 Resultado final de la clase "CompiladorLenguajeUsuario" https://github.com/JonathanGarcia15/tesis-02-analizador-lexico/blob/main/src/Compilador/CompiladorLenguajeUsuario.java	

4.3 Ejecutando JFlex desde el proyecto

El último paso que falta es ejecutar JFlex para que pueda generar el Analizador Léxico. Para lograr esto, se debe programar el botón que se creó en el prototipo del proyecto. En este caso se programará la opción *"{RECOMPILAR ANALIZADOR LÉXICO}*" que está en el menú de cascada, debo aclarar que no se compila, sino que se genera el Analizador Léxico, arreglaré el nombre después.

Para esto solo se requiere añadir un "*Listener*" del menú de cascada en la opción anteriormente mencionada. Dentro de este *Listener* que se ejecuta cada que es presionado, se declara el objeto *CompiladorLenguajeUsuario* y se ejecuta el método *GenerarAnalizadorLexico*()

```
1.  //Este método se ejecuta  cuando se requiere la generación del Analizador Léxico
2.          recompilarLexico.addActionListener(e -> {
3.              //Compilador del Lexico
4.              CompiladorLenguajeUsuario compilador = new CompiladorLenguajeUsuario();
5.              compilador.GenerarAnalizadorLexico();
6.          });
```

Finalmente se compila y se ejecutará el proyecto. Simplemente se dirige a la opción "{*COMPILADOR*}", y se presiona la opción "{*RECOMPILAR ANALIZADOR LÉXICO*}"

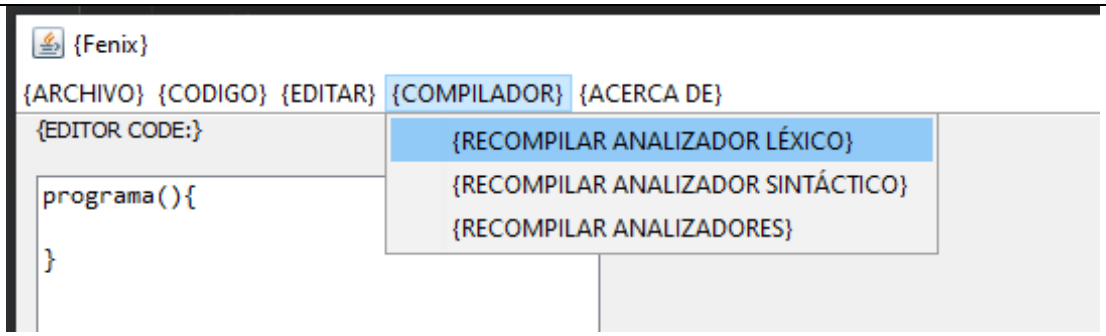

Figura 15 Generar Analizador Léxico

Regresando al IDE. Si todo ha salido a la perfección, y si el Analizador Léxico no tiene errores, tendrá que aparecer que se ha generado con éxito el Analizador Léxico y sin alguna advertencia como la que expliqué anteriormente.

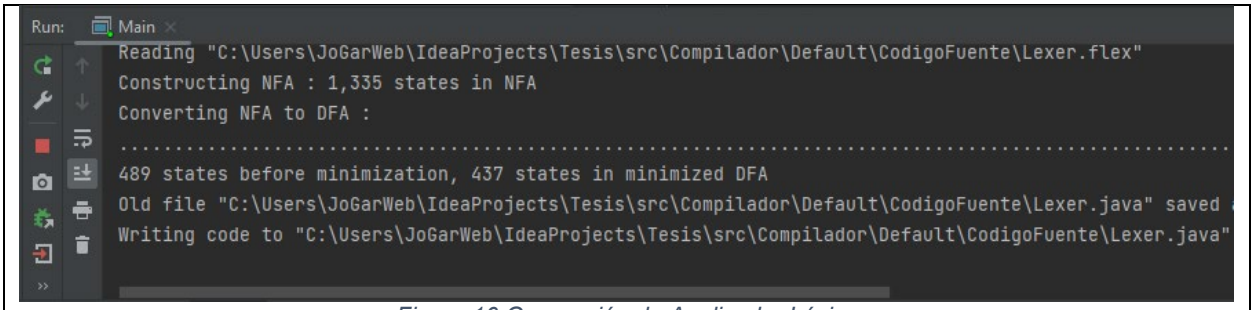

Figura 16 Generación de Analizador Léxico

Se debe de cerrar el programa. Ahora, se observa que en el package *Compilador.Default.AnalizadorLexico*. Ha aparecido una nueva clase llamada "*Lexer*". Esta clase es el Analizador Léxico.

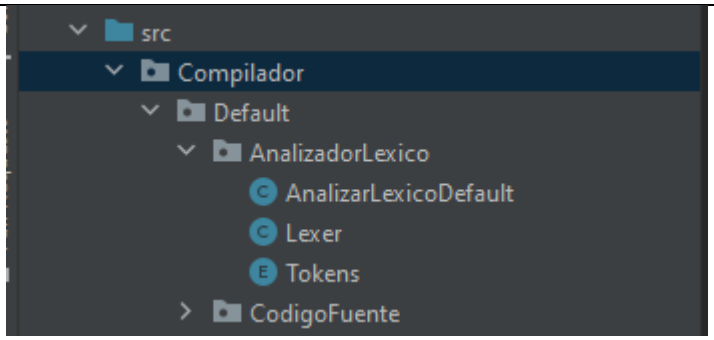

Figura 17 Analizador Léxico

Con esto, se ha terminado de generar el Analizador Léxico. Ha llegado el momento de comenzar a trabajar con él.

4.4 Preparando el Analizador Léxico

Antes de terminar este capítulo, es necesario realizar algunas pruebas al Analizador Léxico para comprobar que funcione a la perfección. Para llegar a eso, es necesario crear unas clases y métodos más. A partir de aquí se comenzará a trabajar con las estructuras JSON.

4.5 La clase del Analizador Sintáctico

Primeramente, se debe preparar un método que se encargará de pasarle el código fuente que escribió el usuario al Analizador Léxico.

Primero se crea un objeto en el cual se guardará el resultado en JSON del análisis, este resultado se va a crear, y no será el resultado que arroje el Analizador Léxico como tal. Igualmente se creará su *getter*.

```
1.      //resultadoDelAnalisis contendrá el objeto JSON con el resultado exitoso o con error
2.      private final JSONObject resultadoDelAnalisis  = new JSONObject();
3.      //Getter de resultadoDelAnalisis
4.      public JSONObject getResultadoDelAnalisis() {
5.          return resultadoDelAnalisis;
6.      }
```

Después se crean 2 métodos parecidos al anterior subtema. Un método privado para realizar el análisis léxico, y generar el objeto JSON con su análisis. Y un método público que acceda al método privado, así para proteger sus atributos en otras partes del programa.

```
1.      public void EjecutarAnalisisLexico(String ProgramaFuente){
2.          this.AnalizarLexico(ProgramaFuente);
3.      }
4.      private void AnalizarLexico(String ProgramaFuente){
5.                  «CÓDIGO PARA REALIZAR EL ANÁLISIS LÉXICO»
6.        }
7.
```

El primer paso dentro del método para comenzar el análisis léxico es precisamente ejecutar la clase *Lexer* y ésta recibirá cómo parámetro el código fuente. Hacer esto es demasiado sencillo y solo tomará una línea de código la cual es la siguiente:

```
1.          //Instrucción para leer el programa fuente con el Analizador Léxico
2.          Lexer analisisLexico = new Lexer(new StringReader(ProgramaFuente));
```

Ejecutando la instrucción anterior, se modifica el Analizador Léxico para que esté preparado para leer el programa fuente.

Aquí se comenzará a construir un objeto JSON. Este objeto JSON tiene la capacidad de pasar distintos datos en un solo objeto, lo anterior, con el objetivo de que el programa principal lea los datos del análisis de una forma más sencilla.

El siguiente paso es ejecutar del Analizador Léxico para que genere los *lexemas*. Para realizar la ejecución, se ocupa una estructura while con un loop infinito. Una vez que termine la lectura, se tendrá que ejecutar "*return*" para salir del loop.

El primer paso dentro de la estructura while es hacer generar los lexemas. Primero se declara un objeto de tipo Tokens que recibirá el token que fue identificado. Después, se ocupa un método del *Lexer* el cual es "*yylex*()". Este método realiza un escaneo del código fuente hasta que forma un lexema y se recibe (Para esto era el bloque de código que se creó al final de La estructura del Analizador Léxico para JFlex). Ejecutar el método anterior puede generar un error de tipo "*IOException*" por lo que al ejecutar esta instrucción deberá estar en una estructura "*try catch*".

Hasta ahora, el código se tendrá que ver de la siguiente manera:

```
1.  while (true) {
2.              Tokens token = null;
```

```
            //Leyendo el token a analizar.
            try {
                token = analisisLexico.yylex();
            } catch (IOException Exception) {
            }
    }
```

Puesto a que en esta parte del programa si llega a ocurrir una excepción, se tendrá que detenerse por completo el análisis, por lo tanto, dentro del bloque "catch" tiene que generar el primer JSON de error además de un "*return*" para detener la ejecución del análisis.

Antes de continuar, es necesario definir qué información tendrá el archivo JSON. Será de lo más sencillo, solo se requiere lo siguiente:

- Un booleano que indique que el Analizador Léxico realizó el análisis con error o de forma exitosa.
- Un mensaje de éxito o fracaso.
- De ser posible, información de tokens leídos. Esta información debe incluir: El nombre del lexema, la línea dónde se encuentra y el símbolo o caracteres que se reconocieron.

Un ejemplo de JSON resultante es el siguiente:

```
{
    "tokens":[«TOKENS LEÍDOS (CERO O MÁS ELEMENTOS)»
            {
                    "line":«LÍNEA DE CÓDIGO»,
                    "lexema":«LEXEMA»,
                    "token":«SÍMBOLO O CARÁTERES RECONOCIDOS»
            }
            ],
    "message":«MENSAJE EXITOSO O DE ERROR»,
    "status":«ESTADO (VERDADERO O FALSO)»
    }

```

Para el caso de la información de tokens leídos, previamente se debe crear un arreglo que guarde distintos objetos de tipo JSON, esto se logra declarando lo siguiente:

```
ArrayList<JSONObject> lecturaTokens = new ArrayList<>();
```

Definido lo anterior, el bloque catch tendrá el siguiente código:

```
            resultadoDelAnalisis.put("status",false);
            resultadoDelAnalisis.put("message",Exception);
            resultadoDelAnalisis.put("tokens",lecturaTokens);
            return;
```

Finalmente, se tiene que analizar el lexema leído, para esto se debe asegurar siempre que el lexema leído no sea *null*, cuando llega a ser *null* significa que el análisis ha terminado y se puede crear un JSON exitoso.

Con una estructura “*Switch case*” se debe poner atención en caso de un lexema Error, pues esto significa que ha encontrado un símbolo que no se definió en las expresiones regulares y se informa al usuario lo sucedido.

Para conocer con exactitud la línea de código dónde ocurre un error, se debe de apoyar en el analizador léxico y contar los saltos de línea que escribe el usuario.

Finalmente, cualquier otro lexema que se reconozca se considera correcto, por lo que se debe agregar continuamente información del lexema en un arreglo de objetos JSON.

Tomando en cuenta lo anterior, el código faltante se verá de la siguiente manera:

```
1.              if(token!=null){
2.                  JSONObject objetoTemporal;
3.                  switch (token) {
4.                      case ERROR:
5.                          resultadoDelAnalisis.put("status",false);
6.                          tmp = "Error en el análisis léxico. Símbolo no reconocido: '" +
    analisisLexico.yytext() + "' Línea: " + numeroDeLinea ;
7.                          resultadoDelAnalisis.put("message",tmp);
8.                          resultadoDelAnalisis.put("tokens",lecturaTokens);
9.                          return;
10.                     case SALTOLINEA:
11.                         numeroDeLinea++;
12.                         objetoTemporal = new JSONObject();
13.                         objetoTemporal.put("lexema",token.toString());
14.                         objetoTemporal.put("line",numeroDeLinea);
15.                         objetoTemporal.put("token", analisisLexico.yytext());
16.                         lecturaTokens.add(objetoTemporal);
17.                         break;
18.                     default:
19.                         objetoTemporal = new JSONObject();
20.                         objetoTemporal.put("lexema",token.toString());
21.                         objetoTemporal.put("line",numeroDeLinea);
22.                         objetoTemporal.put("token", analisisLexico.yytext());
23.                         lecturaTokens.add(objetoTemporal);
24.                         break;
25.                 }
26.             }
27.             if (token == null) {
28.                 //Se ha terminado de analizar el programa fuente con éxito.
29.                 resultadoDelAnalisis.put("status",true);
30.                 resultadoDelAnalisis.put("message","Análisis Lexico analizado con éxito. ");
31.                 resultadoDelAnalisis.put("tokens",lecturaTokens);
32.                 return;
33.             }
34.
```

Con esto se ha terminado esa sección, sobra decir que si se requiere ver el código de forma completa se puede acceder a la siguiente dirección web:

Enlace 9 Código Final del archivo AnalizarLexicoDefault.java https://github.com/JonathanGarcia15/tesis-02-analizador-lexico/blob/main/src/Compilador/Default/AnalizadorLexico/AnalizarLexicoDefault.java	

4.6 Un método más

Lo anterior aún no se ha probado, para esto se tiene crear una nueva clase con la que pueda ejecutar la clase creada anteriormente. No adelanto nada, pero hacer esto es necesario porque más adelante se requerirá de esta clase, y se modificará a conveniencia.

Se busca un método que ejecute el Análisis Léxico y que siga regresando el Objeto JSON creado en la clase que se acaba de crear. Este nuevo método recibirá dos parámetros: El Código Fuente y un nuevo parámetro llamado “lenguaje”. Aún no adelantaré nada, pero con los parámetros de entrada he dicho una pista.

La nueva clase deberá quedar de la siguiente manera:

```
1.      public JSONObject AnalizarSoloLexico(String expresion, String lenguaje){
2.
3.          return this.ejecutarAnalizadorLexicoDefault(expresion);
4.
5.      }
6.
7.      private JSONObject ejecutarAnalizadorLexicoDefault(String expresion){
8.          //Analisis únicamente del Analizador Léxico por Default
9.          AnalizarLexicoDefault LexicoLenguajeNatural = new AnalizarLexicoDefault();
10.         LexicoLenguajeNatural.EjecutarAnalisisLexico(expresion);
11.         return (LexicoLenguajeNatural.getResultadoDelAnalisis());
12.     }
13.
```

Aunque puse prácticamente todo el código omitiendo algunas cosas, se puede consultar el código completo de esta clase en el siguiente enlace:

Enlace 10 Consultar el código de la clase "EjecutarAnalizadores.java" https://github.com/JonathanGarcia15/tesis-02-analizador-lexico/blob/main/src/Compilador/EjecutarAnalizadores.java	

4.7 Programando el submenú

Ahora sí, teniendo los elementos anteriores, ha llegado la hora de programar los últimos detalles antes de ejecutar el Analizador Sintáctico.

Hay que recordar que se ha agregado un submenú en el programa gráfico, ahora se tendrá que programar la acción de un submenú más. Este submenú de momento estará ubicado en "{*CODIGO*}" y en "{*COMPILAR*}". Nuevamente no importa lo que diga el submenú, por una razón que aún me estoy guardando (porque falta su explicación), se tendrá que cambiar la etiqueta, más no su acción.

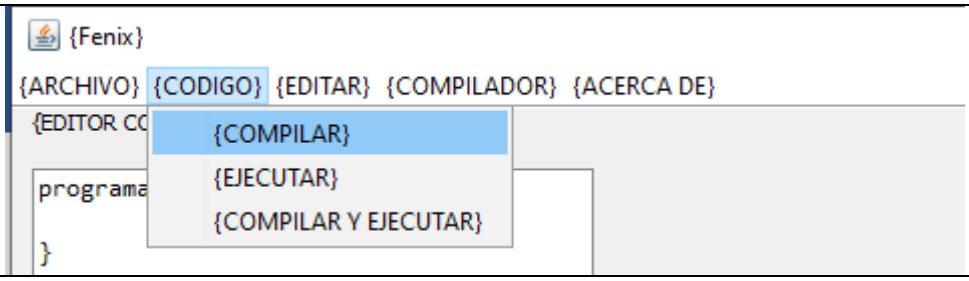

Figura 39.- Pruebas al Analizador Léxico

Para esto se creará un *Listener* que se ejecute cada vez que se presione ese submenú. Y ahora, lo primero que se tiene que hacer es leer el código fuente. El código fuente habrá sido escrito en el espacio que se destinó para ello (En el Entorno de Desarrollo creado).

Para leer el código fuente se requiere del siguiente código:

```
1.              //Guardando en un String el código fuente:
2.              String ProgramaFuente = txtCodigoFuente.getText();
```

Ahora que se conoce cuál es el código fuente del usuario, simplemente se ejecutará el Analizador Léxico. Hay que recordar guardar el resultado en un Objeto JSON local.

```
1.              //En este objeto se guardará el resultado del Analisis Léxico:
2.              JSONObject resultado;
3.              //Ejecutando el Analisis Léxico:
4.              resultado = Analizadores.AnalizarSoloLexico(ProgramaFuente,"");
```

Ahora sí, ya teniendo los resultados guardados en “resultado” se puede mostrar mensajes de éxito o error.

Para un mensaje de éxito y para propósito de esta parte del proyecto, en la parte de Log del Entorno de Desarrollo se imprime cada carácter leído y su lexema. Igualmente, toda la estructura del JSON que ha regresado el Análisis Sintáctico. Esto último con el propósito de ir conociendo a detalle los resultados de salida y detectar errores en caso de requerirlo.

Para realizar esto se tiene tener una cadena con el lexema leído, la cual se irá construyendo mientras se realiza el análisis del resultado.

Finalmente, se imprime el mensaje del JSON, seguido de la cadena que contiene lexemas y que se fue construyendo al analizar los resultados, y finalmente la cadena JSON que regresó el Analizador Léxico.

Para los mensajes de errores, se imprime el error encontrado, y de ser posible, se imprime la cadena JSON del Analizador Léxico que se generó antes de llegar a un error.

El código quedará de la siguiente manera:

```
1.              //Declaración de una cadena temporal.
2.              StringBuilder tmp = new StringBuilder();
3.
4.              //Analizando los resultados:
5.              if(resultado.getBoolean("status")){
6.                  //Enlistando los tokens recibidos
7.                  JSONObject objetoTemporal;
8.                  for(int i=0;i<resultado.getJSONArray("tokens").length();i++){
9.                      objetoTemporal = (JSONObject) resultado.getJSONArray("tokens").get(i);
10.                     if(objetoTemporal.get("token") == "SALTOLINEA"){
11.                         tmp.append("\n");
12.                     }else{
13.                         tmp.append("\"").append(objetoTemporal.get("token")).append("\"");
14.                         tmp.append("(").append(objetoTemporal.get("lexema")).append(") ");
15.                     }
16.                 }
17.                 tmp.append("\n").append("JSON recibido: ").append(resultado);
18.                 txtResultados.setText(resultado.getString("message") + "\n" + tmp);
19.                 txtResultados.setForeground(new Color(25, 111, 61));
20.             }else{
21.                 //Errores en el resultado
22.                 tmp.append("\n").append("JSON recibido: ").append(resultado);
23.                 txtResultados.setText(resultado.getString("message")+tmp);
24.                 txtResultados.setForeground(Color.red);
25.             }
26.
```

Como he estado diciendo, si se requiere consultar el código de forma completa, se puede consultar la siguiente dirección, pero en este caso solo hay que fijarse en de las líneas 106 a la 140 debido a que aquí se encuentran otros elementos que ocupa el Entorno de Desarrollo.

Enlace 11 Código final de “Fenix.java” https://github.com/JonathanGarcia15/tesis-02-analizador-lexico/blob/main/src/InterfazDeUsuario/Fenix.java	

5 Modificaciones para generar el Analizador Sintáctico

Antes de continuar, se requiere preparar el proyecto para poder generar el compilador desde el Entorno de Desarrollo.

Lo primero a realizar será crear dos nuevos archivos en el package de "*CodigoFuente*" del proyecto. Estos tendrán el nombre de *Lexercup.flex* y *Sintax.cup*

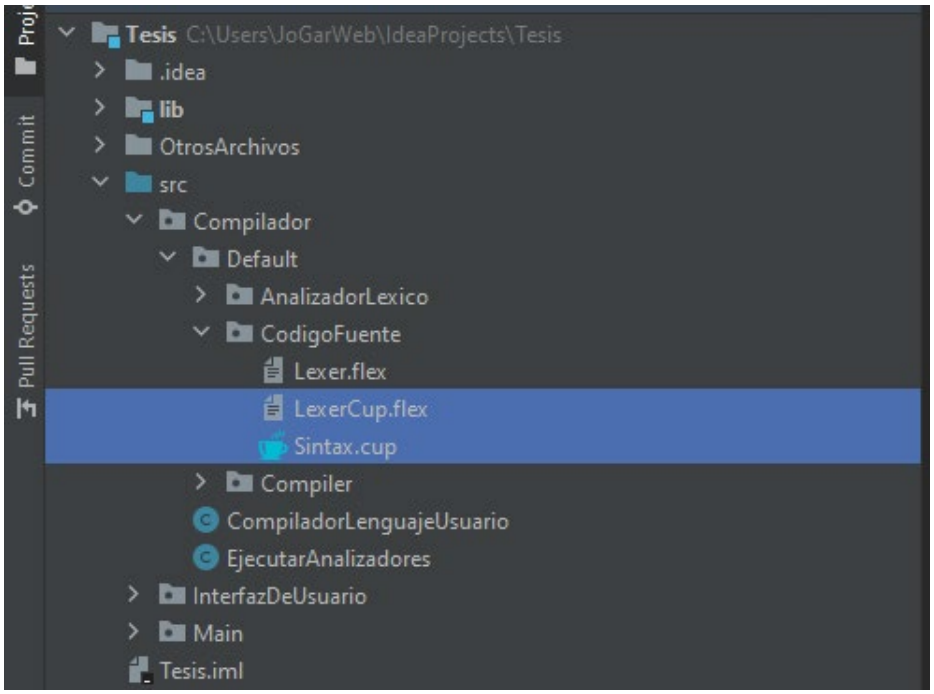

Figura 18 Archivos de CUP

Después, se crea un nuevo package dentro de Default llamado "*Compiler*". En este package irán los archivos necesarios del compilador.

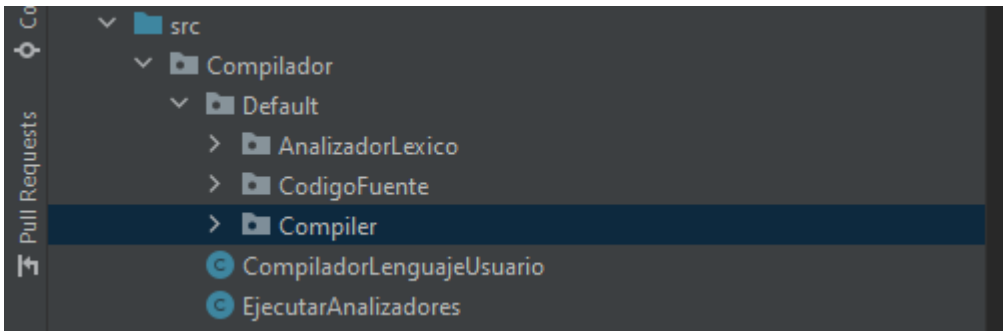

Figura 19 Directorio del proyecto

Estas dos modificaciones sencillas ayudarán a continuar con el proyecto.

5.1 Modificaciones al Analizador Léxico

Ahora que se ha preparado el proyecto, es necesario preparar el Analizador Léxico que se creó y así poderlo conectar con el Analizador Sintáctico.

Esta modificación es sencilla. Debido a que no se desea desechar lo que anteriormente se creó, se modificará un nuevo Analizador Léxico modificando el analizador creado en el Capítulo 4 de este documento. Se copia todo el contenido del archivo *Lexer.flex* a *LexerCUP.flex*. Después se modifica el encabezado hasta antes de la definición de las expresiones regulares modificando lo siguiente:

```
1.  package Compilador.Default.Compiler;
2.  import Java_CUP.runtime.Symbol;
3.
4.  %%
5.
6.  %class LexerCUP
7.  %type Java_CUP.runtime.Symbol
8.  %CUP
9.  %full
10. %line
11. %char
12.
```

Las instrucciones anteriores permitirán al Analizador Sintáctico conectarse constantemente con el Analizador Léxico. Después de realizar lo anterior, lo único que se modifica será el valor que regresarán las reglas léxicas. Esto se realiza modificando el valor de retorno a lo siguiente:

return new Symbol(sym.«TOKEN», yychar, yyline, yytext());

Omitiendo los comentarios, saltos de línea y los blancos. Las reglas léxicas tendrán la siguiente forma:

```
1.  {LLAVECIERRA} {
2.      return new Symbol(sym.LLAVECIERRA, yychar, yyline, yytext());
3.  }
4.
5.  {PARENTESISABRE} {
6.      return new Symbol(sym.PARENTESISABRE, yychar, yyline, yytext());
7.  }
8.
9.  {PARENTESISCIERRA} {
10.     return new Symbol(sym.PARENTESISCIERRA, yychar, yyline, yytext());
11. }
12.
13. {BLANCO} {/* Se ignoran los espacios en Blanco */}
14.
15. {COMENTARIO} { /* Se ignoran los comentarios */ }
16.
17. {SALTOLINEA} {/* Se ignoran los saltos de Línea */}
18.
19. . {
20.     return new Symbol(sym.ERROR, yychar, yyline, yytext());
21. }
```

Esas serán las únicas modificaciones que tendrá el Analizador Léxico.

5.2 CUP

Ahora se comenzará a trabajar con la herramienta que generará el Analizador Sintáctico. Para comenzar a trabajar con esta herramienta es necesario definir la estructura de un archivo en CUP.

```
1.  «Especificaciones»
2.  «Código del usuario»
3.  «Lista de símbolos en la gramática»
4.  «Precedencia»
5.  «Gramática»
6.
```

- En las especificaciones se indicarán los paquetes que pertenecen al Analizador Sintáctico, así como las librerías que se utilizan en el proyecto.
- Código Java que se pueden generar y que aparecerá en el archivo final.
- Se indican los símbolos terminales, no terminales y el símbolo inicial de la gramática.
- Es opcional, y se indica en caso de ser necesario, la procedencia que debería llevar un terminal sobre otro.
- En la parte final, lleva todas las gramáticas que se realizaron.

Tomando en cuenta lo anterior se comenzará a generar el archivo CUP.

En la primera parte se agregan el paquete al que pertenece la clase que generará la herramienta CUP y se declaran algunas librerías que se usarán en el **Capítulo 7 del trabajo escrito**.

```
1.  package Compilador.Default.Compiler;
2.
3.  import Java_CUP.runtime.Symbol;
4.  import Java.util.Stack;
5.  import org.json.JSONObject;
6.
```

Continuando con el Código de Usuario, en este caso se declaran algunas cosas que se utilizarán en el **Capítulo 7 del trabajo escrito** y otras que se usarán en este mismo.

```
1.  parser code
2.  {:
3.      private Symbol simbolo;
4.
5.        private JSONObject resultadoJSON;
6.        private JSONObject resultadoJSONTemporal;
7.
8.        private Stack identificadorDeInstrucciones = new Stack();
9.        private Stack instruccionesParaLoop = new Stack();
10.
11.     public JSONObject getResultadoJSON() {
12.         return resultadoJSON;
13.     }
14.
```

```
15.     public void syntax_error(Symbol simbolo){
16.         this.simbolo = simbolo;
17.     }
18.
19.     public Symbol getSimbolo(){
20.         return this.simbolo;
21.       }
22. :};
23.
```

Se continua enlistando los terminales (Que se encuentran en mayúsculas), los no terminales (Que se encuentran con minúsculas y mayúsculas) y la producción inicial que será “Inicio”.

Los terminales tienen que ser exactamente igual a los que se definieron en el Analizador Léxico. Por ejemplo:

```
1.  {DESACTIVARKABOOM} {
2.      return new Symbol(sym.DESACTIVARKABOOM, yychar, yyline, yytext());
3.  }
4.
```

```
1.  terminal DEJAPINTAR;
2.  terminal DESACTIVARKABOOM;
3.  terminal IMPRIMIRCADENA;
4.
5.  non terminal OperadoresAritmeticos;
6.  non terminal InstruccionModificacionDeValorDeVariables;
7.  non terminal InstruccionAvanzar;
8.
9.  start with Inicio;
10.
```

Finalmente se enlistarán todas las producciones de la gramática que se crearon

```
1.  Inicio::=
2.        PROGRAMA  PARENTESISABRE  PARENTESISCIERRA  LLAVEABRE  BloqueDeInstrucciones    LLAVECIERRA
    Funciones
3.  ;
4.  InstruccionEliminarObjeto::=
5.        ELIMINAR PARENTESISABRE PARENTESISCIERRA PUNTOYCOMA
6.  ;
7.  MasParametrosDeEntrada::=
8.        SEPARADOR VARIABLE IDENTIFICADOR MasParametrosDeEntrada
9.        | /* No más parámetros */
10. ;
11.
```

A diferencia del Analizador Léxico, en esta parte no importa el orden en la que se acomode la gramática o las listas con terminales o no terminales. Solo importa que aparezcan todos los elementos.

Así se ha preparado el Analizador Sintáctico y que se conectará a un nuevo Analizador Léxico.

5.3 Generando el compilador

Ahora se ha creado algo parecido a lo que se realizó con el Analizador Léxico, se modifica la clase "*CompiladorLenguajeUsuario*" para generar 2 nuevos métodos:

- Un método privado dónde se generará de nuevo el Analizador Léxico y Analizador Sintáctico. Una vez que termine, tendrá que mover estos archivos generados a un nuevo directorio,
- y otro método que llame el método anterior.

Los archivos a mover al package "*/src/Compilador/Default/Compiler/*" serán los siguientes:

- *LexerCUP.Java* que una vez generado, se encontrará en la siguiente ruta: */src/Compilador/Default/ CodigoFuente/*
- *sym.Java* y *Sintax.Java* se generarán en la raíz del proyecto.

La clase para generar ambos analizadores y cambiar de directorio los archivos generados, quedará de la siguiente forma:

```
1.      private void GenerarLexicoYSintacticoDefault(){
2.          //Generando Analizador Léxico
3.          //Ruta del archivo flex
4.          String              rutaLexico              =              DireccionProyecto              +
    "/src/Compilador/Default/CodigoFuente/LexerCUP.flex";
5.
6.          File archivo;
7.
8.          //Para leer Lexer.flex y generar el Analizador Léxico
9.          archivo = new File(rutaLexico);
10.         jflex.Main.generate(archivo);
11.
12.         //Generando Analizador Sintáctico
13.         //Ruta del archivo CUP
14.         String[]      rutaS      =      {"-parser",      "Sintax",      DireccionProyecto      +
    "/src/Compilador/Default/CodigoFuente/Sintax.CUP"};
15.
16.         try {
17.             Java_CUP.Main.main(rutaS);
18.         } catch (Exception e) {
19.             eliminarArchivoEnCasoDeError("/src/Compilador/Default/CodigoFuente/LexerCUP.Java");
20.             //throw new RuntimeException(e);
21.         }
22.
23.         //Moviendo los archivos generados al package Compilador/Default/Compiler
24.
25.         moverArchivos("/sym.Java","/src/Compilador/Default/Compiler/sym.Java");
26.         moverArchivos("/Sintax.Java","/src/Compilador/Default/Compiler/Sintax.Java");
27.
    moverArchivos("/src/Compilador/Default/CodigoFuente/LexerCUP.Java","/src/Compilador/Default/Comp
    iler/LexerCUP.Java");
28.
29.     }
30.
```

Y el método que llamará a el método anterior quedará así:

```
1.      public void GenerarAnalizadoresLexicoYSintactico(){
2.          this.GenerarLexicoYSintacticoDefault();
3.      }
4.
```

Como mencioné, se realizará lo mismo que en el Analizador Léxico con unas variaciones y un par de archivos más para cambiar de directorio.

5.3.1 Programando la instrucción

Para generar los analizadores léxico y sintáctico se requiere programar el botón que conforma el submenú creado al principio del proyecto. Para este caso será el submenú llamado "{*RECOMPILAR ANALIZADOR SINTACTICO*}". Sé que el nombre está mal escrito y que debería ser algo como: "Generar analizadores", pero como ya lo he mencionado antes, estas etiquetas se podrán cambiar.

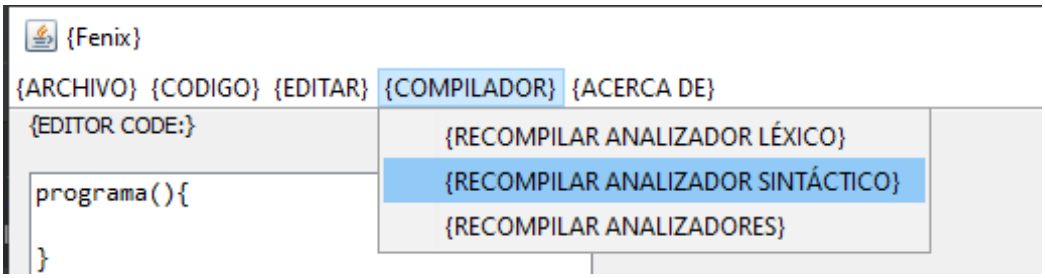

Figura 20 Generar Analizador Sintáctico

Para realizar esto, simplemente se programa un listener que se ejecute cuando se presiona esta opción y se tendrá que ejecutar el método "*GenerarAnalizadoresLexicoYSintactico();*"

```
1.  //Este método se ejecuta cuando se requiere la generación de los Analizadores Léxico y Sintáctico
2.          recompilarSintactico.addActionListener(e -> {
3.              //Generando Solamente Léxico y Sintáctico
4.              generadorDeCompilador.GenerarAnalizadoresLexicoYSintactico();
5.          });
6.
```

Al ejecutar este comando aparecerán 3 archivos en el package compilador.

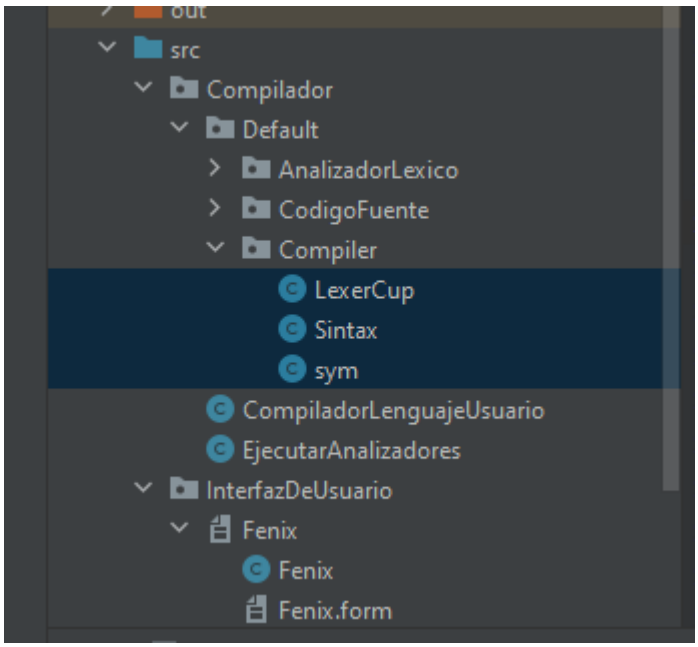

Figura 21 Analizador Sintáctico

Con esto se han generado los archivos necesarios del compilador.

5.3.2 Creando la clase que ejecutará el compilador

Como lo se realizó anteriormente, se requiere de una clase que sea capaz de ejecutar ambos analizadores. Para este propósito, se crea una nueva clase llamada "*Compiler.Java*".

Esta clase tendrá una nueva instrucción que ejecutará ambos analizadores como uno mismo. Esta instrucción es la siguiente declaración:

```
1.  Sintax compilador = new Sintax(new LexerCUP(new StringReader(codigoFuente)));
2.
```

Finalmente, para correr el compilador, se utiliza el método "*parse()*"

```
1.  compilador.parse();
2.
```

Tal cual se hizo con el Analizador Léxico, el compilador tendrá que generar un JSON con los resultados obtenidos. La diferencia es que en este caso no se regresa a los *tokens* generados, y solo se tendrá que regresar un objeto JSON con los parámetros "status" y "message".

Si llegase a generar un error, se puede recuperar exactamente el punto dónde ocurrió un error léxico. Se tiene que crear una clase *Symbol* que pertenece a CUP, y con el método "*getSimbolo()*" del Analizador Sintáctico.

- Para conocer la línea del error se ocupa la instrucción "right" que devolverá la línea del error menos 1.
- Para conocer la columna de error se ocupa la instrucción "left" que devolverá la columna dónde se encuentra el error menos 1.
- Para conocer exactamente el símbolo que falló se utilizará "value" y este regresará el símbolo no reconocido por el Analizador Sintáctico.

El método se verá de la siguiente forma:

```
1.  private void ejecutar(String codigoFuente){
2.
3.          Sintax compilador = new Sintax(new LexerCUP(new StringReader(codigoFuente)));
4.
5.          try {
6.              compilador.parse();
7.
8.              resultadoDelAnalisis.put("status",true);
9.              resultadoDelAnalisis.put("message","No hay problemas con el Análisis Léxico y
    Sintáctico");
10.
11.         } catch (Exception e) {
12.
13.             String mensaje = "";
14.             try{
15.                 Symbol simbolo = compilador.getSimbolo();
16.                 resultadoDelAnalisis.put("status",false);
17.                 mensaje += "Error de sintaxis en programa. Línea: ";
18.                 mensaje += simbolo.right + 1 ;
19.                 mensaje += " Columna: ";
20.                 mensaje += (simbolo.left + 1);
21.                 mensaje += ", Texto: ";
22.                 mensaje += simbolo.value;
23.                 resultadoDelAnalisis.put("message",mensaje);
24.
25.             }catch(Exception error){
26.                 resultadoDelAnalisis.put("status",false);
27.                 mensaje = "Error de sintaxis desconocido, revise las variables que utilizó. ";
28.                 resultadoDelAnalisis.put("message",mensaje);
29.             }
30.             //throw new RuntimeException(e);
31.         }
32.     }
33.
```

Al igual que el Analizador Léxico se requiere un método público que ejecute este método privado, y un método *get* para obtener los resultados del análisis.

```
1.      private final JSONObject resultadoDelAnalisis  = new JSONObject();
2.
3.      public JSONObject getResultadoDelAnalisis() {
4.          return resultadoDelAnalisis;
5.      }
6.
7.      public void ejecutarCompilador(String codigoFuente){
8.          this.ejecutar(codigoFuente);
9.      }
10.
```

Ahora se modifica un poco la clase "*EjecutarAnalizadores*". Simplemente se agrega una clase que ejecute el método "*ejecutarCompilador*" recién creado.

```
1.      public JSONObject EjecutarCompilador(String codigoFuente, String lenguaje){
2.          return this.ejecutarCompiladorDefault(codigoFuente);
3.      }
4.      private JSONObject ejecutarCompiladorDefault(String codigoFuente){
5.          Compiler compilador = new Compiler();
6.          compilador.ejecutarCompilador(codigoFuente);
7.          return compilador.getResultadoDelAnalisis();
8.      }
9.
```

5.3.3 Ejecutando el compilador

Finalmente, para poder ejecutar el compilador, se tendrá que modificar la acción del botón "{*EJECUTAR*}"

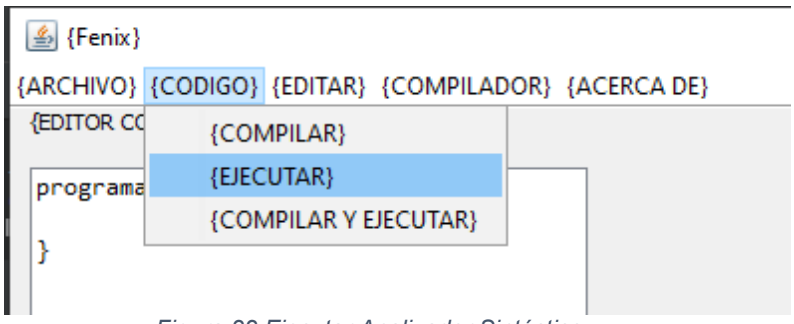

Figura 22 Ejecutar Analizador Sintáctico

Para esto, se creará un listener y se volverá a realizar el mismo código que se escribió para ejecutar el Analizador Léxico.

En este caso es el mismo código, con excepción de que, en este caso, no se analizan tokens, sino solo las respuestas válidas o incorrectas.

El código de este método quedará de la siguiente forma:

```
1.          ejecutar.addActionListener(e -> {
2.              JSONObject resultado;
3.              resultado = Analizadores.EjecutarCompilador(leerCodigoFuenteDelUsuario(),"");
4.              StringBuilder tmp = new StringBuilder();
5.              //Analizando los resultados:
6.              if(resultado.getBoolean("status")){
7.                  //Enlistando los tokens recibidos
8.                  JSONObject objetoTemporal;
9.                  tmp.append("\n").append("JSON recibido: ").append(resultado);
10.                 txtResultados.setText(resultado.getString("message") + "\n" + tmp);
11.                 txtResultados.setForeground(new Color(25, 111, 61));
12.             }else{
13.                 //Errores en el resultado
14.                 tmp.append("\n").append("JSON recibido: ").append(resultado);
15.                 txtResultados.setText(resultado.getString("message")+tmp);
16.                 txtResultados.setForeground(Color.red);
17.             }
18.
19.         });
20.
```

5.4 Pruebas del Analizador Léxico y Analizador Sintáctico.

Ahora ha llegado el turno de probar lo que será el Analizador Léxico y Analizador Sintáctico.

Lo ideal es que se fuera probando instrucción por instrucción hasta hacer un Código Fuente con todos los elementos del lenguaje. Realizando lo anterior, se puede detectar los errores de la gramática creada y corregirla. Por eso mismo, si cualquier persona que lea este texto ha realizado sus propias gramáticas, es buena práctica probar esas producciones e ir construyendo un código fuente de lo más pequeño a grande.

El programa de prueba será el siguiente y al final, deberá mostrar un mensaje de éxito.

```
1.  programa(){
2.        /// Este es un comentario
3.        var id;
4.        var id<-78;
5.        var id<-random();
6.        var id<-aleatorio(4,9);
7.        var id<-(78+78);
8.        var id<-78+(78+id);
9.        var id<-(78+78);
10.       var id<-(78+78+id);
11.       var id<-id;
12.       var id<-(id+89);
13.       var id<-(id+89+(200*78*random())+(51155+ssd)-id);
14.       var id<-(id+89);
15.       var id<-(id+89)+aleatorio(4,9);
16.       var id, id;
17.       var id, id, id;
18.
19.       ... este es otro comentario ...
20.       var id,
21.                 id<-78,
22.                 id<-random(),
23.                 id<-aleatorio(4,9),
```

```
                id<-(78+78),
                id<-78+(78+id),
                id<-(78+78),
                id<-(78+78+id),
                id<-(id+89+(200*78*random())+(51155+aleatorio(4,9))-id),
                id<-id,
                id<-(id+89),
                id<-(id+89+id),
                id<-(id+89),
                id<-(id+89)+id,
                id, id,
                id, id, id;

        id<-78;
        id<-(78+78);
        id<-78+(78+id);
        id<-(78+78);
        id<-(78+78+id);
        id<-id;
        id<-(id+89);
        id<-(id+89+(200*78*er)+(51155+ssd)-id);
        id<-(id+89);
        id<-(id+89)+id;

        avanzar();
        avanzar(random(7,98));
        avanzar(aleatorio());
        move(5);
        avanzar(id);
        move(5+78);
        move(5+78+(id+96));

        espera();
        espera(random(7,98));
        espera(aleatorio());
        wait(5);
        espera(id);
        wait(5+78);
        espera(5+78+(id+96));

        turn_left();
        turn_left(random(7,98));
        turn_left(aleatorio());
        gira_izquierda(5);
        turn_left(id);
        gira_izquierda(5+78);
        turn_left(5+78+(id+96));

        turn_right();
        turn_right(random(7,98));
        turn_right(aleatorio());
        gira_derecha(5);
        turn_right(id);
        gira_derecha(5+78);
        turn_right(5+78+(id+96));

        turn_right();
        turn_right(random(7,98));
        turn_right(aleatorio());
        gira_derecha(5);
        turn_right(id);
        gira_derecha(5+78);
        turn_right(5+78+(id+96));

        pick_up();
```

```
89.       toma_objeto();
90.
91.       put_down();
92.       soltar_objeto();
93.
94.       delete_object();
95.       eliminar_objeto();
96.
97.       disable_bomb();
98.       desactivar_bomba();
99.       disable_kaboom();
100.                desactivar_kaboom();
101.
102.                paint_floor();
103.                pintar_suelo(azul);
104.                paint_floor(blue);
105.                pintar_suelo(green);
106.                paint_floor(verde);
107.                pintar_suelo(amarillo);
108.                paint_floor(yellow);
109.                pintar_suelo(red);
110.                paint_floor(rojo);
111.                pintar_suelo();
112.
113.                stop_painting();
114.                dejar_de_pintar();
115.
116.                print_var(id);
117.                imprimir_var(id);
118.                print_variable(id);
119.                imprimir_variable(id);
120.
121.
122.                print (id);
123.                imprimir("Cadena: ");
124.                print (id+"El ID es: ");
125.                imprimir("Cadena: "+id);
126.                print (id+"El ID es: "+ id);
127.                imprimir("Cadena: "+id+ "Cadena");
128.
129.                object_in_front ();
130.                tengo_objeto_delante();
131.
132.                bomb_in_front();
133.                tengo_bomba_delante();
134.                kaboom_in_front();
135.                tengo_kaboom_delante();
136.
137.                in_front_me ();
138.                delante_de_mi();
139.
140.                wall_in_front();
141.                tengo_muro_delante();
142.
143.                llamadaAFuncion();
144.                llamadaAFuncion(id);
145.                llamadaAFuncion(848);
146.                llamadaAFuncion(id,89,id);
147.                llamadaAFuncion(random(),id,89);
148.                llamadaAFuncion(random(7,95),id,89);
149.                llamadaAFuncion(848+78+id);
150.                llamadaAFuncion(id+(78/3),(id+78)+1,id+1);
151.
152.                if(id<78){
153.                          wall_in_front();
```

```
154.                    tengo_muro_delante();
155.            }
156.
157.            if(true){
158.                    wall_in_front();
159.                    tengo_muro_delante();
160.            }
161.
162.            if(random() <= aleatorio(7,9)){
163.                    wall_in_front();
164.                    tengo_muro_delante();
165.            }
166.
167.            if(!(id<78)){
168.                    wall_in_front();
169.                    tengo_muro_delante();
170.            }
171.
172.            if((true)){
173.                    wall_in_front();
174.                    tengo_muro_delante();
175.            }
176.
177.            if(!(random() <= aleatorio(7,9))){
178.                    wall_in_front();
179.                    tengo_muro_delante();
180.            }
181.
182.            if((aleatorio(7,9) >= 78) or !(true) ){
183.                    wall_in_front();
184.                    tengo_muro_delante();
185.            }
186.
187.            if(id<78){
188.                    wall_in_front();
189.                    tengo_muro_delante();
190.            }else_if(true){
191.                    wall_in_front();
192.                    tengo_muro_delante();
193.            }
194.
195.            if(id<78){
196.                    wall_in_front();
197.                    tengo_muro_delante();
198.            }else_if(true){
199.                    wall_in_front();
200.                    tengo_muro_delante();
201.            }sinosi(random() <= aleatorio(7,9)){
202.                    wall_in_front();
203.                    tengo_muro_delante();
204.            }else_if(!(id<78)){
205.                    wall_in_front();
206.                    tengo_muro_delante();
207.            }sinosi((true)){
208.                    wall_in_front();
209.                    tengo_muro_delante();
210.            }else_if(!(random() <= aleatorio(7,9))){
211.                    wall_in_front();
212.                    tengo_muro_delante();
213.            }sinosi((aleatorio(7,9) >= 78) or !(true) ){
214.                    wall_in_front();
215.                    tengo_muro_delante();
216.            }
217.
218.            if(id<78){
```

```
219.                    wall_in_front();
220.                    tengo_muro_delante();
221.          }else_if(true){
222.                    wall_in_front();
223.                    tengo_muro_delante();
224.          }sinosi(random() <= aleatorio(7,9)){
225.                    wall_in_front();
226.                    tengo_muro_delante();
227.          }else_if(!(id<78)){
228.                    wall_in_front();
229.                    tengo_muro_delante();
230.          }sinosi((true)){
231.                    wall_in_front();
232.                    tengo_muro_delante();
233.          }else_if(!(random() <= aleatorio(7,9))){
234.                    wall_in_front();
235.                    tengo_muro_delante();
236.          }sinosi((aleatorio(7,9) >= 78) or !(true) ){
237.                    wall_in_front();
238.                    tengo_muro_delante();
239.          }
240.
241.          si(id<78){
242.                    wall_in_front();
243.                    tengo_muro_delante();
244.          }else_if(true){
245.                    wall_in_front();
246.                    tengo_muro_delante();
247.          }sinosi(random() <= aleatorio(7,9)){
248.                    wall_in_front();
249.                    tengo_muro_delante();
250.          }else_if(!(id<78)){
251.                    wall_in_front();
252.                    tengo_muro_delante();
253.          }sinosi((true)){
254.                    wall_in_front();
255.                    tengo_muro_delante();
256.          }else_if(!(random() <= aleatorio(7,9))){
257.                    wall_in_front();
258.                    tengo_muro_delante();
259.          }else{
260.                    wall_in_front();
261.                    tengo_muro_delante();
262.          }
263.
264.          comparar(id){
265.                    case 7:
266.                              wall_in_front();
267.                              tengo_muro_delante();
268.                    case <7:
269.                              wall_in_front();
270.                              tengo_muro_delante();
271.                              end;
272.                    case random():
273.                              wall_in_front();
274.                              tengo_muro_delante();
275.                              fin;
276.                    case aleatorio(7,12):
277.                              wall_in_front();
278.                              tengo_muro_delante();
279.                              fin;
280.                    default:
281.                              wall_in_front();
282.                              tengo_muro_delante();
283.                              fin;
```

```
284.                }
285.
286.                para(var i<-0, id<-(2*random()) ; i<25 ; i++, a--){
287.                          wall_in_front();
288.                          tengo_muro_delante();
289.                }
290.
291.                hacer{
292.                          wall_in_front();
293.                          tengo_muro_delante();
294.                }while(falso);
295.
296.                repite_hasta(verdadero){
297.                          wall_in_front();
298.                          tengo_muro_delante();
299.                }
300.      }
301.
302.      funcionSimple(){
303.                repite_hasta(verdadero){
304.                          wall_in_front();
305.                          tengo_muro_delante();
306.                }
307.      }
308.
309.      funcionConParametros(var id, var other, var right){
310.                funcionSimple();
311.                turn_right(id);
312.                gira_derecha(other);
313.                turn_right(right);
314.      }
315.
```

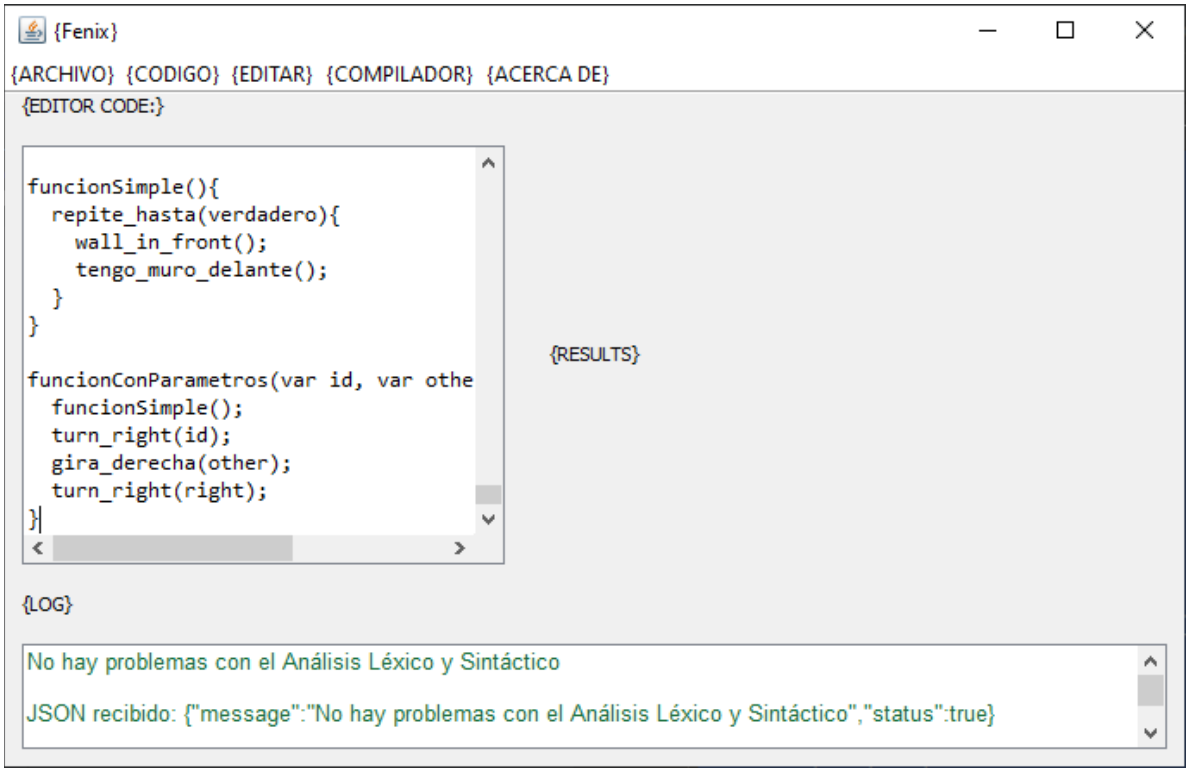

Figura 23 Mensaje de éxito del Analizador Sintáctico

Para este programa de prueba quiero aclarar las siguientes cosas:

- Se toman en cuenta casos extremos para probar la gramática. Pero así se han podido detectar errores de la gramática.
- Por si no se ha detectado, se declaran varias veces una misma variable. Para este caso de prueba se puede pasar desapercibido. En este punto se está probando solo la gramática, por lo que se puede ser altamente permisivo en ese aspecto.
- Hay algo que mencioné en la teoría que no se está tomando en cuenta por ahora, se hablan en el **Capítulo 4.3 del trabajo escrito**. Al lector de este texto: ¿logras detectar cuál es el elemento ausente?

Con esto se ha terminado esta parte del análisis léxico y sintáctico. Se puede concluir hasta ahora que ambos analizadores se conectan entre sí. En el **Capítulo 8 del trabajo escrito** detallará aún más algunas partes del Analizador Sintáctico.

6 Compilador bilingüe

He dicho muy poco sobre el soporte de idiomas del compilador. Pero para llegar a este punto era necesario llegar a generar un compilador estable y funcional, pero, sobre todo que el compilador se encuentre en una etapa final dónde casi no requerirá de modificaciones. Podría decirse que el compilador ha entrado ahora a lo que se conoce en la ingeniería de software como “etapa de mantenimiento”. Ya se ha terminado con él, sin embargo, aún se podrían agregar características o modificar ciertos componentes para mejorarlos, pero estos serían cambios menores. El trabajo pesado ha terminado.

En este capítulo, se comenzará a reciclar código que se ha estado realizando con el objetivo de generar 2 compiladores más.

Las instrucciones del compilador las reconoce indistintamente si están en inglés o español, pero ¿qué sucede si solo quiero utilizar instrucciones únicamente en inglés o únicamente en español? Es decir, que el compilador sea capaz de reconocer un error cuando se tiene configurado el compilador como “solo español” cuando una instrucción está en inglés.

El Analizador Léxico es el encargado de separar el programa fuente en tokens, gracias a las expresiones regulares que se ha creado. Si somos más observadores, estas expresiones regulares están diseñadas para reconocer ambos idiomas. Entonces lo que se debe realizar es quitarle esa característica para que solo reconozca instrucciones para el idioma que se requiera.

Ahora hablando del Analizador Sintáctico, este se comunica constantemente con el Analizador Léxico. No hay forma de indicarle a un Analizador Sintáctico para que solo utilice cierto Analizador Léxico a conveniencia del usuario. Por lo que se tendrá que generar una vez más.

En ambos casos, son actualizaciones menores porque ya está terminado todo.

6.1 Modificando archivos del proyecto

Ahora se crearán distintos *package* dentro de la estructura del proyecto. Acá tendrá sentido haber separado tanto en la estructura del proyecto los archivos para el compilador, que probablemente no tenían sentido en un principio.

Se comenzará creando 2 *package* diferentes en el *package* llamado “*Compilador*”. Estos llevarán el nombre de “*EN*” (de inglés) y “*ES*” (de español).

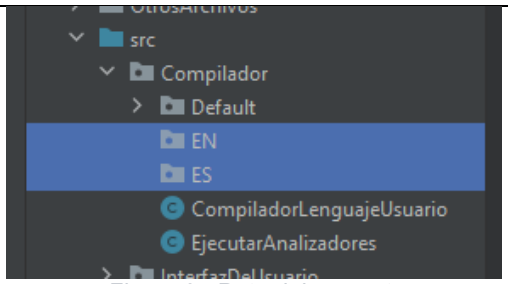

Figura 24 Ruta del proyecto

Ahora se crean los respectivos package idénticos a cómo están en el *package* "*Compilador*".

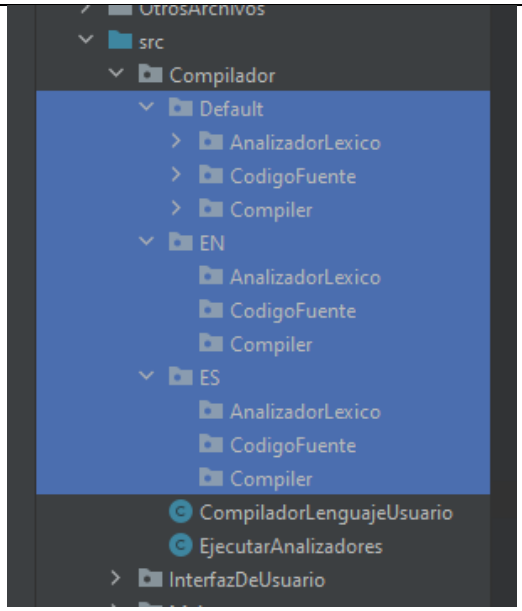

Figura 25 Ruta del proyecto

Por último, se copian 4 archivos en sus respectivos lugares, las cuales son: *Tokens.Java*, y los 3 archivos dentro del package "*CodigoFuente*".

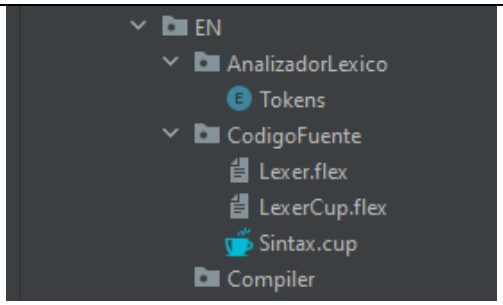

Figura 26 Archivos de compilador

Ahora para cada respectivo archivo se cambia el package dónde se encuentran. Solo se reemplaza la palabra "*Default*" por "*EN*" o "*ES*" según sea el caso.

Tokens.Java:

```
1.  package Compilador.EN.AnalizadorLexico;
```

Lexer.flex:

```
1.  package Compilador.EN.AnalizadorLexico;
2.  import static Compilador.EN.AnalizadorLexico.Tokens.*;
```

LexerCUP.flex:

```
1.  package Compilador.EN.Compiler;
```

Sintax.CUP:

```
1.  package Compilador.EN.Compiler;
```

Tabla 2.- Cambios del compilador

Por último, de cada archivo para generar el Analizador Léxico se eliminan las instrucciones según se requieran de la lista de expresiones regulares.

Para este paso hay que ser demasiado cuidadoso, pues el mínimo error podría cambiar completamente una instrucción.

Antes	Después (En inglés)
1. PARA=for|para 2. DEFAULT=default 3. SINOSI=else_if|sinosi 4. REPITEHASTA=do_over|repite_hasta 5. REPITE=while|mientras 6. HACER=do|hacer	1. PARA=for 2. DEFAULT=default 3. SINOSI=else_if 4. REPITEHASTA=do_over 5. REPITE=while 6. HACER=do

6.2 Generando los analizadores

Finalmente, lo único que faltará por realizar es generar los analizadores de cada respectivo idioma para poder comenzar a trabajar con ellos.

Para realizar esto, es necesario modificar la clase "*CompiladorLenguajeUsuario*" con el objetivo de que sea capaz de generar los analizadores y acomodarlos en sus respectivas rutas para que puedan ser utilizadas.

Para el caso del Analizador Léxico simple (aquel que solo revisa que esté bien escrito el programa) se modifica el código que ya fue escrito. Convirtiéndola en una clase diferente y que reciba como parámetros el código fuente para generar el Analizador Léxico (flex), la ruta dónde se encontrará lo generado y la ruta destino.

```
1.  private void GenerarLexico(String rutaArchivoFlex, String rutaArchivoLexerOriginal, String
    rutaArchivoLexerDestino) {
2.
3.          //Ruta de los archivos
4.          String rutaLexico = DireccionProyecto + rutaArchivoFlex;
5.
6.          File archivo;
7.
8.          //Para leer Lexer.flex y generar el Analizador Léxico
9.          archivo = new File(rutaLexico);
10.         jflex.Main.generate(archivo);
11.
12.         //Moviendo el archivo Lexer.Java
13.         this.moverArchivos(rutaArchivoLexerOriginal,rutaArchivoLexerDestino);
14.
15.     }
```

Se ha creado una nueva clase para mover archivos. Esta parte del código se podrá reciclar para el generador del Analizador Sintáctico.

```
1.  private void moverArchivos(String rutaDelArchivoExistente, String rutaNuevaDelArchivo){
2.          Path path = Paths.get(DireccionProyecto + rutaNuevaDelArchivo);
3.          //Eliminando si existe un archivo previo
4.          if (Files.exists(path)) {
5.              try {
6.                  Files.delete(path);
7.              } catch (IOException ex) {
8.                  //Logger.getLogger(Tangananica.class.getName()).log(Level.SEVERE, null, ex);
9.              }
10.         }
11.
12.         try {
13.             Files.move(
14.                     Paths.get(DireccionProyecto+rutaDelArchivoExistente),
15.                     path
16.             );
17.         } catch (IOException ex) {
18.         }
19.     }
```

Para el Analizador Léxico y Analizador Sintáctico (aquellos que trabajan en conjunto) ocurrirá algo similar, únicamente se requiere que se generen los analizadores y que se muevan los respectivos 2 archivos que genera a la carpeta deseada.

```
1.  private void GenerarLexicoYSintactico(String rutaArchivoFlex, String rutaArchivoLexerOriginal,
    String rutaArchivoLexerDestino,String rutaArchivoCUP, String rutaArchivoSYMDestino, String
    rutaArchivoSintaxDestino){
2.          //Generando Analizador Léxico
3.          //Ruta del archivo flex
4.          String rutaLexico = DireccionProyecto + rutaArchivoFlex;
5.
6.          File archivo;
7.
8.          //Para leer Lexer.flex y generar el Analizador Léxico
9.          archivo = new File(rutaLexico);
10.         jflex.Main.generate(archivo);
11.
12.         //Generando Analizador Sintáctico
13.         //Ruta del archivo CUP
14.         String[] rutaS = {"-parser", "Sintax", DireccionProyecto + rutaArchivoCUP};
15.
16.         try {
17.             Java_CUP.Main.main(rutaS);
18.         } catch (Exception e) {
19.             throw new RuntimeException(e);
20.         }
21.
22.         //Moviendo los archivos generados al package Compilador/Default/Compiler
23.
24.         moverArchivos("/sym.Java",rutaArchivoSYMDestino);
25.         moverArchivos("/Sintax.Java",rutaArchivoSintaxDestino);
26.         moverArchivos(rutaArchivoLexerOriginal,rutaArchivoLexerDestino);
27.
28.     }
29.
```

Por último, se modifica la clase “EjecutarAnalizadores”, hay que recordar que esta clase será la encargada de ejecutar el compilador y obtener el objeto JSON con el programa leído.

Para objetivos del proyecto, se ha creado 3 compiladores distintos, la única diferencia se encuentra en el lenguaje soportado de ellos. Un compilador no diferenciará el lenguaje en el que se encuentre el programa fuente, los otros 2 solo reconocerán las instrucciones en inglés o español según sea el caso.

Para esto se requiere que el método reciba un parámetro del lenguaje que se está utilizando en ese momento. Con base a eso, ejecutará el Compilador deseado.

```
1.      public JSONObject EjecutarCompilador(String codigoFuente, String lenguaje){
2.
3.          switch(lenguaje){
4.              case "ES":
5.                  if(this.ejecutarAnalizadorLexicoES(codigoFuente).getBoolean("status")){
6.                      return this.ejecutarCompiladorES(codigoFuente);
7.                  }else{
8.                      return this.ejecutarAnalizadorLexicoES(codigoFuente);
9.                  }
10.             case "EN":
11.                 if(this.ejecutarAnalizadorLexicoEN(codigoFuente).getBoolean("status")){
12.                     return this.ejecutarCompiladorEN(codigoFuente);
13.                 }else{
14.                     return this.ejecutarAnalizadorLexicoEN(codigoFuente);
15.                 }
16.             default:
17.                 //Comprobando que el análisis Léxico esta correctamente escrito
18.                 if(this.ejecutarAnalizadorLexicoDefault(codigoFuente).getBoolean("status")){
19.                     return this.ejecutarCompiladorDefault(codigoFuente);
20.                 }else{
21.                     return this.ejecutarAnalizadorLexicoDefault(codigoFuente);
22.                 }
23.         }
24.
25.     }
```

```
1.      private JSONObject ejecutarAnalizadorLexicoDefault(String codigoFuente){
2.          //Analisis únicamente del Analizador Léxico por Default
3.          AnalizarLexicoDefault LexicoLenguajeNatural = new AnalizarLexicoDefault();
4.          LexicoLenguajeNatural.EjecutarAnalisisLexico(codigoFuente);
5.          return (LexicoLenguajeNatural.getResultadoDelAnalisis());
6.      }
7.
8.      private JSONObject ejecutarCompiladorDefault(String codigoFuente){
9.          CompilerDefault compilador = new CompilerDefault();
10.         compilador.ejecutarCompilador(codigoFuente);
11.         return compilador.getResultadoDelAnalisis();
12.     }
```

Como se puede observar, cada uno de estos métodos llaman a otras clases que permiten el análisis correcto del programa fuente, según sea el caso del lenguaje de preferencia de las opciones del IDE.

Para probar que se están haciendo las cosas de manera correcta, se tendrá que modificar de momento la instrucción que llama a estas instrucciones modificando el parámetro de lenguaje español, inglés o default según sea el caso.

```
1.  resultado = Analizadores.EjecutarCompilador(leerCodigoFuenteDelUsuario(),"ES");
```

El resultado del programa (si se modifica a “solo español”) tendrá que dejar de reconocer las instrucciones en inglés.

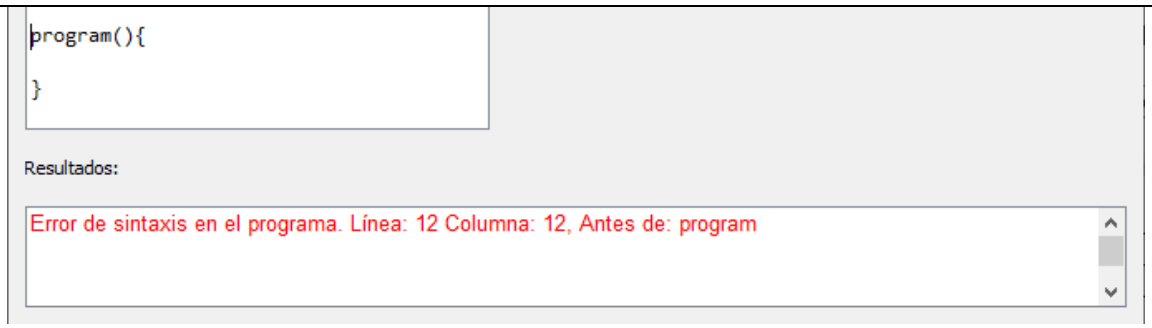

Figura 27 Programa no reconoce instrucciones en inglés

Con esto ha quedado ese soporte de idiomas del compilador. Aunque son 3 compiladores distintos, responden al diseño original y a los objetivos del proyecto.

7 Soporte de Idiomas

Uno de los objetivos del IDE es el soporte de idiomas en inglés y español. En este capítulo dotaré al IDE la capacidad de cambiar de idioma según prefiera el usuario final.

La idea de este capítulo es que se pueda configurar un lenguaje de usuario preferido, todas las instrucciones deberán mostrarse en inglés o español según sea el caso. Para lograr esto, se volverá a trabajar con los formatos JSON.

Para esto se requerirá de una clase nueva. Esta clase leerá archivos de texto con cadenas de texto en distintos idiomas.

7.1 La clase para generar cadenas de texto

Ahora se crea una clase que sea capaz de obtener una cadena de texto de un archivo de texto en formato JSON. Lo anterior, con el objetivo de buscar y encontrar una cadena de texto (inglés y español).

Esta clase utilizará algunos objetos globales las cuales son: 3 objetos JSON con las configuraciones, cadenas de texto y colores en hexadecimal; además de la dirección del proyecto actual.

```
1.      private JSONObject JSONStrings;
2.      private JSONObject JSONConfigs;
3.      private JSONObject JSONColors;
4.      private final String DireccionProyecto = System.getProperty("user.dir").replace("\\", "/");
```

Para esto se sobrescribe el constructor de la clase que reciba un objeto *String* llamado "*nombreDeLaClase*", para que cuando un objeto sea creado solo cargue los textos específicos de esa clase. Además, tendrá que tener los valores del archivo de configuración anteriormente especificado y un archivo con colores en Hexadecimal.

```
1.      public cargarTexto(String nombreDeLaClase) {
2.          try {
3.              JSONStrings = new JSONObject(
4.                      new String(Files.readAllBytes(Paths.get(DireccionProyecto + "/FILES/Strings/"
    + nombreDeLaClase + ".json")), StandardCharsets.UTF_8)
5.              );
6.              JSONConfigs = new JSONObject(
7.                      new          String(Files.readAllBytes(Paths.get(DireccionProyecto          +
    "/FILES/Config/Config.json")), StandardCharsets.UTF_8)
8.              );
9.              JSONColors = new JSONObject(
10.                     new          String(Files.readAllBytes(Paths.get(DireccionProyecto          +
    "/FILES/Strings/Colors.json")), StandardCharsets.UTF_8)
11.             );
12.         }catch(IOException ignore){
13.             JSONStrings = new JSONObject();
14.             JSONConfigs = new JSONObject();
15.             JSONColors = new JSONObject();
16.         }
17.     }
18.
```

7.2 Preferencias del usuario

La primera característica a diseñar para el IDE será un archivo de configuración. Todo programa tiene uno y permite guardar las preferencias del usuario. La estructura de este archivo de configuración será el siguiente:

```
{
  "TituloDelPrograma": "Fenix",
  "IdiomaDelCompilador": "DEFAULT",
  "Version": "0",
  "MostrarOpcionGenerarCompiladores": true,
  "TiempoInstruccion": 1500,
  "IdiomaDelPrograma": "ES",
  "TamanoTextoMapa": 16,
  "TiempoEspera": 1000
}
```

La explicación de las etiquetas son las siguientes:

- TituloDelPrograma Es la etiqueta con el nombre del proyecto.
- IdiomaDelPrograma: Es el idioma en el que se mostrará el IDE y el idioma de los resultados. En este caso, solo puede ser “ES” de español y “EN” de inglés.
- IdiomaDelCompilador: Será el idioma de preferencia con el que trabajará el compilador. Es decir, si se deben soportar instrucciones de cualquier idioma, o solo inglés o español.
- Version: Indica la versión del programa.
- MostrarOpcionGenerarCompiladores: Es una etiqueta que ocultará las opciones para generar los Analizadores Léxico y Sintáctico del proyecto. Estas opciones no deben estar al alcance del usuario final, por lo que se ocultarán desde el archivo config.
- TiempoInstruccion: Es el tiempo que se tomará el programa en realizar una instrucción a otra. Se explicará más adelante.
- TamanoTextoMapa: Es el tamaño del programa. Igualmente se explicará más adelante.
- TiempoEspera: Es el tiempo que se tomará la instrucción “*Espera*()” del lenguaje.

Lo siguiente a realizar son métodos que puedan acceder a estas configuraciones del programa. Estos métodos podrán regresar un Entero, Cadena de texto y Booleanos.

```
    public String getConfigString(String key) {
        try {
            return new JSONObject(
                    new          String(Files.readAllBytes(Paths.get(DireccionProyecto          +
"/FILES/Config/Config.json")), StandardCharsets.UTF_8)
            ).getString(key);
        }catch (Exception ignore){
            return "";
        }
    }

    public int getConfigInteger(String key) {
        try {
            return new JSONObject(
```

```
14.                     new          String(Files.readAllBytes(Paths.get(DireccionProyecto          +
    "/FILES/Config/Config.json")), StandardCharsets.UTF_8)
15.             ).getInt(key);
16.         }catch (Exception ignore){
17.             return -1;
18.         }
19.     }
20.
21.     public boolean getConfigBoolean(String key) {
22.         try {
23.             return new JSONObject(
24.                     new          String(Files.readAllBytes(Paths.get(DireccionProyecto          +
    "/FILES/Config/Config.json")), StandardCharsets.UTF_8)
25.             ).getBoolean(key);
26.         }catch (Exception ignore){
27.             return false;
28.         }
29.     }
```

7.3 Escritura de las configuraciones

Para la parte de configuraciones, necesita de algunos métodos que sean capaces de escribir el archivo Config.json, es necesario pues estas configuraciones podrán ser personalizadas por el usuario.

Para este caso se utilizará algo que se le conoce en la Programación Orientada a Objetos como “Sobrecarga de métodos”. Esto con el objetivo de tener una misma instrucción para realizar esta acción.

La sobrecarga de métodos quedará de la siguiente manera:

```
1.      public boolean putConfigValue(String key, String value) {
2.          return putString(key,value);
3.      }
4.
5.      public boolean putConfigValue(String key, int value) {
6.          return putInt(key,value);
7.      }
8.
9.      public boolean putConfigValue(String key, boolean value) {
10.         return putBoolean(key,value);
11.     }
```

Y las acciones de estos tres métodos son las siguientes:

```
1.      private boolean putString(String key, String value) {
2.          JSONConfigs.put(key,value);
3.          saveConfigFile(JSONConfigs);
4.          return true;
5.      }
6.
7.      private boolean putInt(String key, int value) {
8.          if((Objects.equals(key, "TamanoTextoMapa"))&&(value<=10)){
9.              JSONConfigs.put(key,11);
10.             saveConfigFile(JSONConfigs);
11.             return true;
12.         }else if((Objects.equals(key, "TamanoTextoMapa"))&&(value>=40)){
13.             JSONConfigs.put(key,39);
14.             saveConfigFile(JSONConfigs);
```

```
            return true;
        }else if((Objects.equals(key, "TiempoEspera"))&&(value<=0)){
            JSONConfigs.put(key,1);
            saveConfigFile(JSONConfigs);
            return true;
        }else if((Objects.equals(key, "TiempoEspera"))&&(value>2000)){
            JSONConfigs.put(key,2000);
            saveConfigFile(JSONConfigs);
            return true;
        }else if((Objects.equals(key, "TiempoInstruccion"))&&(value<=0)){
            JSONConfigs.put(key,1);
            saveConfigFile(JSONConfigs);
            return true;
        }else if((Objects.equals(key, "TiempoInstruccion"))&&(value>2000)){
            JSONConfigs.put(key,2000);
            saveConfigFile(JSONConfigs);
            return true;
        }
        JSONConfigs.put(key,value);
        saveConfigFile(JSONConfigs);
        return true;
    }

    private boolean putBoolean(String key, Boolean value) {
        JSONConfigs.put(key,value);
        saveConfigFile(JSONConfigs);
        return true;
    }

    private void saveConfigFile(JSONObject config) {
        try {
            FileWriter archivoEscrito = new FileWriter(DireccionProyecto+"/FILES/Config/Config.json");
            archivoEscrito.write(config.toString());
            archivoEscrito.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
```

7.4 Cadenas de Texto

El IDE deberá estar soportado en 2 idiomas diferentes, por lo que cada clase deberá tener un archivo JSON con las cadenas necesarias en inglés y español. Esta estructura será la siguiente:

```
{
  "key": ["«Cadena en español»", "«Cadena en inglés»"]
}
```

Dónde:

- Key: es la clave con la que se conocerá la cadena de texto.
- «Cadena en español»: Será la cadena en idioma español.
- «Cadena en inglés»: Será la cadena en idioma inglés.

Finalmente, para consultar estos valores se creará un método para regresar la cadena de texto.

```
1.      public String traerTexto(String valor){
2.          return texto(valor);
3.      }
4.      private String texto(String key){
5.          JSONObject Cadenas = this. traeObjeto();
6.          JSONObject IdiomaPreferido = this. traeConfiguracionIdiomaPrograma();
7.
8.          if ("EN".equals(IdiomaPreferido.getString("IdiomaDelPrograma"))) {
9.              return Cadenas.getJSONArray(key).getString(1);
10.         }
11.         return Cadenas.getJSONArray(key).getString(0);
12.     }
```

7.5 Colores

Los colores servirán más adelante de consulta para generar el mapa del IDE, la estructura JSON del archivo Colors.json es la siguiente:

```
1.  {
2.    "COLOR":"«valor hexadecimal»"
3.  }
```

Dónde:

- COLOR. Es la llave con la que se encuentra el valor hexadecimal.
- Valor Hexadecimal. Es el valor en hexadecimal del color al cual le está dando un valor. Por ejemplo: #FFFFFF.

Para la consulta de los hexadecimales de colores, se ocupará el siguiente método:

```
1.      public String traerColor(String valor){
2.          return JSONColors.getString(valor);
3.      }
```

8 Detalles finales

Para terminar con el proyecto, es necesario realizar unos ajustes visuales a el programa para mostrar un proyecto terminado.

8.1 Menú contextual

Primero se modificará el menú del usuario. A continuación, se muestra la estructura final que tendrá y la descripción que debe realizar cada una de ellas.

8.1.1 Archivo/File

Contiene instrucciones básicas para cargar un mapa y el código fuente del usuario. Debe contener los siguientes elementos:

Nombre	Atajo del teclado	Descripción
Abrir Mapa	Ctrl+N	Abre una ventana de selección de archivos para que el usuario seleccione la ruta del lugar dónde se encuentra un mapa en CSV.
Abrir Código	Ctrl+O	Abre una ventana de selección de archivos para que el usuario seleccione la ruta del lugar dónde se encuentra el archivo de código fuente.
Guardar Código	Ctrl+S	Guarda en un archivo de texto el programa fuente del usuario.
Preferencias	Ctrl+Alt+S	Abre una ventana de configuración del programa.
Salir	Ctrl+Alt+Q	Sale del programa.

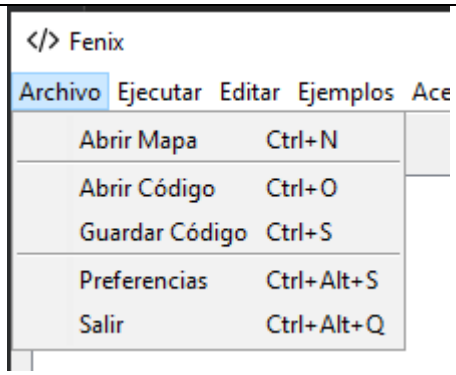

Figura 28 Menú Contextual Archivo del Entorno de Desarrollo

8.1.2 Ejecutar/Run

Contiene las instrucciones necesarias para ejecutar el código fuente del usuario. Debe contener los siguientes elementos:

Nombre	Atajo del teclado	Descripción
Análisis Léxico	F9	Realiza un análisis léxico del programa fuente del usuario.
Análisis Léxico y Sintáctico	Ctrl+F10	Realiza un análisis léxico y sintáctico del programa fuente del usuario sin ejecutar las instrucciones.
Analizar y Ejecutar	F10	Realiza un análisis léxico y sintáctico del programa fuente del usuario y ejecuta las instrucciones.

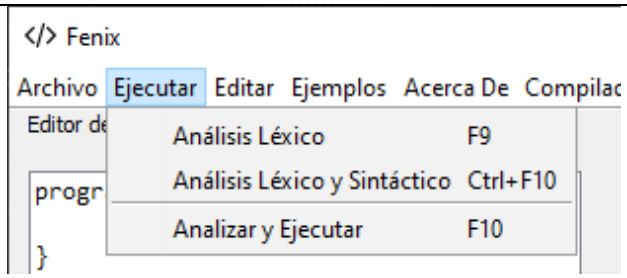

Figura 29 Menú Contextual Ejecutar del Entorno de Desarrollo

8.1.3 Editar/Edit

Contiene las opciones necesarias para controlar la máquina virtual. Debe contener los siguientes elementos:

Nombre	Atajo del teclado	Descripción
Detener Programa	F2	Detiene el programa que se esté ejecutando en ese momento.
Limpiar Mapa	Ctrl+R	Si no hay programas en ejecución, volverá el mapa a su estado cuando fue cargado.
Recargar Archivos de Editor de Código	Ctrl+M	Si se está trabajando en otro editor de texto distinto al IDE esta función es útil porque todos los cambios realizados de forma externa aparecerán en el IDE presionando este botón.
Recargar Archivos de Mapa	Ctrl+K	Si se llega a modificar el mapa y se desea volver a cargar sin necesidad de volverlo a abrir se presiona esta opción.

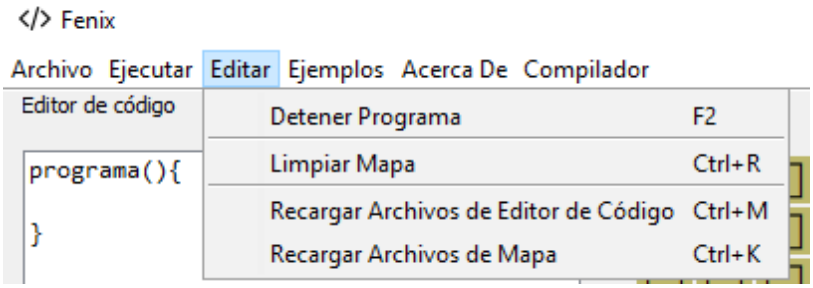

Figura 30 Menú Contextual Editar del Entorno de Desarrollo

8.1.4 Ejemplos/Examples

Contiene las opciones necesarias para cargar mapas y código de ejemplo. Debe contener los siguientes elementos:

Nombre	Atajo del teclado	Descripción
Ejemplo 1	Ctrl+1	Carga un mapa y código fuente de ejemplo.
Ejemplo 2	Ctrl+2	Carga un mapa y código fuente de ejemplo.
Ejemplo 3	Ctrl+3	Carga un mapa y código fuente de ejemplo.
Ejemplo 4	Ctrl+4	Carga un mapa y código fuente de ejemplo.

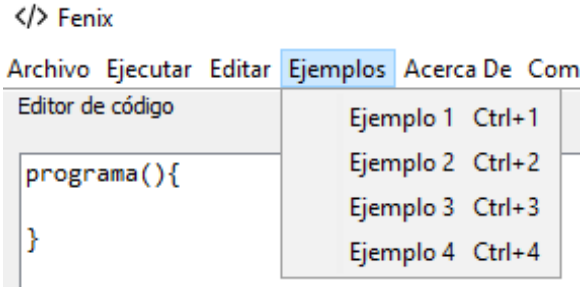

Figura 31 Menú Contextual Ejemplos del Entorno de Desarrollo

8.1.5 Acerca De/About

Contiene los siguientes elementos:

Nombre	Atajo del teclado	Descripción
Acerca de	F12	Abre una ventana con la información del programa.

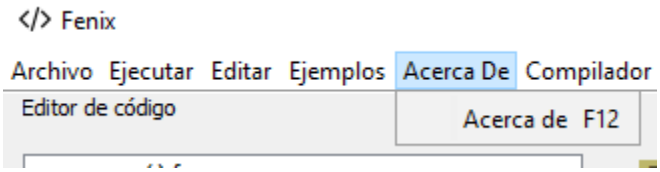

Figura 32 Menú Contextual Acerca De del Entorno de Desarrollo

8.1.6 Compilador/Compiler

Estas opciones son las que generan las clases necesarias del compilador. Estas opciones siempre deben estar ocultas si se trata de un programa en producción. Las opciones son las siguientes:

Nombre	Atajo del teclado	Descripción
Generar Analizador Léxico	(Sin atajo)	Genera el Analizador Léxico individual.
Generar Analizador Léxico y Analizador Sintáctico	(Sin atajo)	Genera el Analizador Léxico y Analizador Sintáctico del compilador.

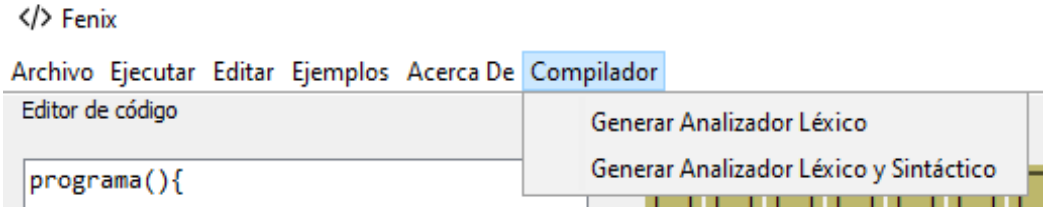

Figura 33 Menú Contextual Compilador del Entorno de Desarrollo

8.2 La ventana de configuración

La ventana de configuración es un elemento dónde el usuario podrá configurar el idioma del programa, el idioma del compilador y modificar los tiempos de espera entre instrucción.

Se debe crear la siguiente interfaz:

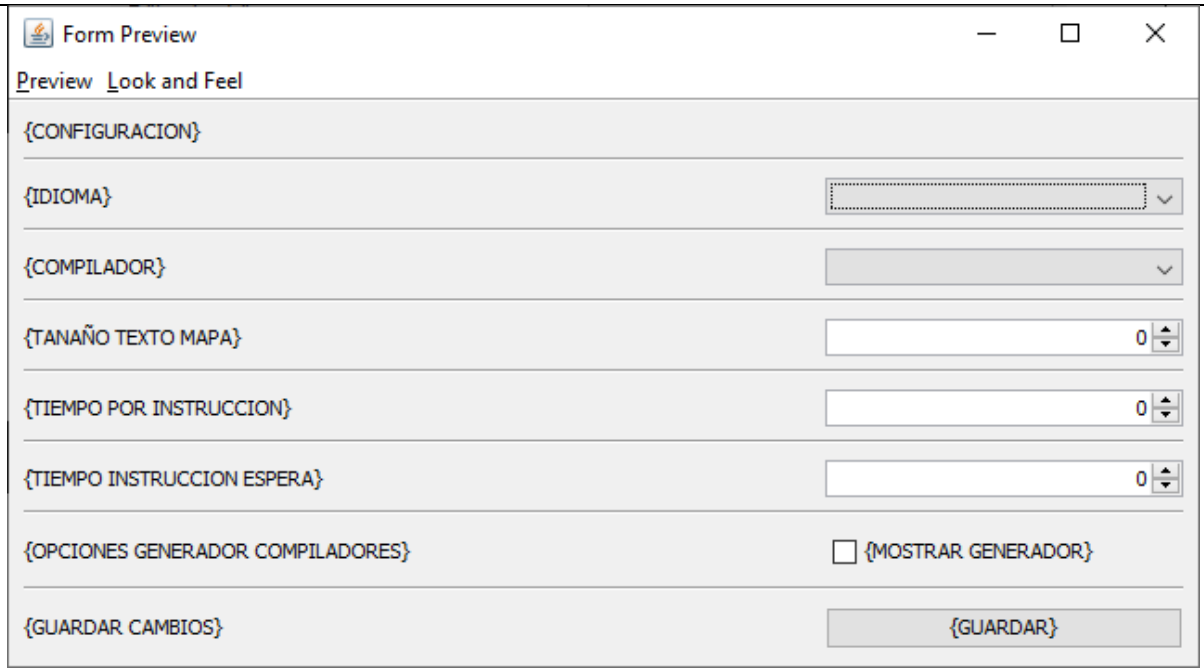

Figura 34 Ventana de configuración del Entorno de Desarrollo

No detallaré los métodos que componen esta ventana, pero tiene que ser capaz de editar el archivo config.json para guardar las opciones del usuario.

Al final se tendrá que ver algo así:

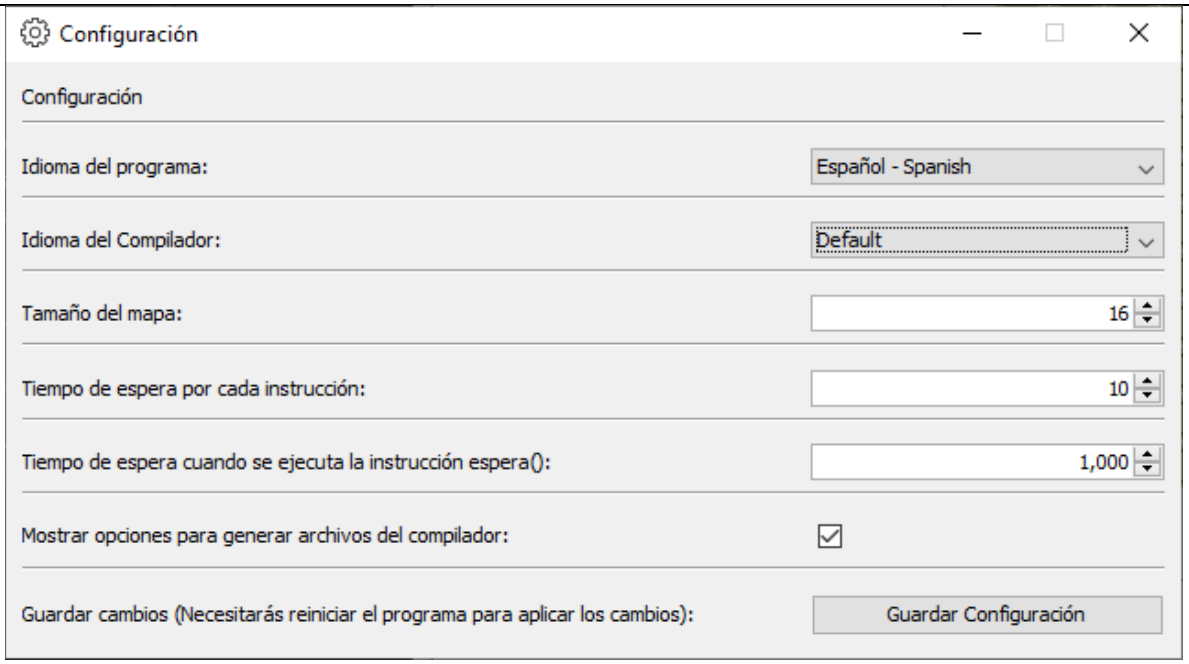

Figura 35 Ventana de configuración definitiva del Entorno de Desarrollo

Dónde:

- Idioma del programa: Solo existen 2 opciones inglés o español. Cambia todas las etiquetas del programa y se muestra en el idioma según lo indique.
- Idioma del compilador: Existen 3 opciones.
    - Si se selecciona “default” el compilador reconocerá las instrucciones en inglés y español sin distinción.
    - Si se selecciona “Solo inglés” el compilador solo reconocerá las instrucciones en inglés.
    - Si se selecciona “Solo español” el compilador solo reconocerá las instrucciones en español.
- Tamaño del mapa: Es el tamaño de la letra que se mostrará el objeto lblResultados.
- Tiempo de espera por cada instrucción: Es el tiempo de espera en milisegundos entre cada instrucción de la máquina virtual.
- Tiempo de espera cuando se ejecuta la instrucción espera(). Es el tiempo de espera en milisegundos que debe esperar el usuario cuando ejecuta la instrucción “espera()”.

8.3 Los detalles que hacen la diferencia

Finalmente, cada que el IDE sea abierto, deberá cargar siempre con un mapa de pruebas y ahorrarle la escritura al usuario final de la función principal del lenguaje.

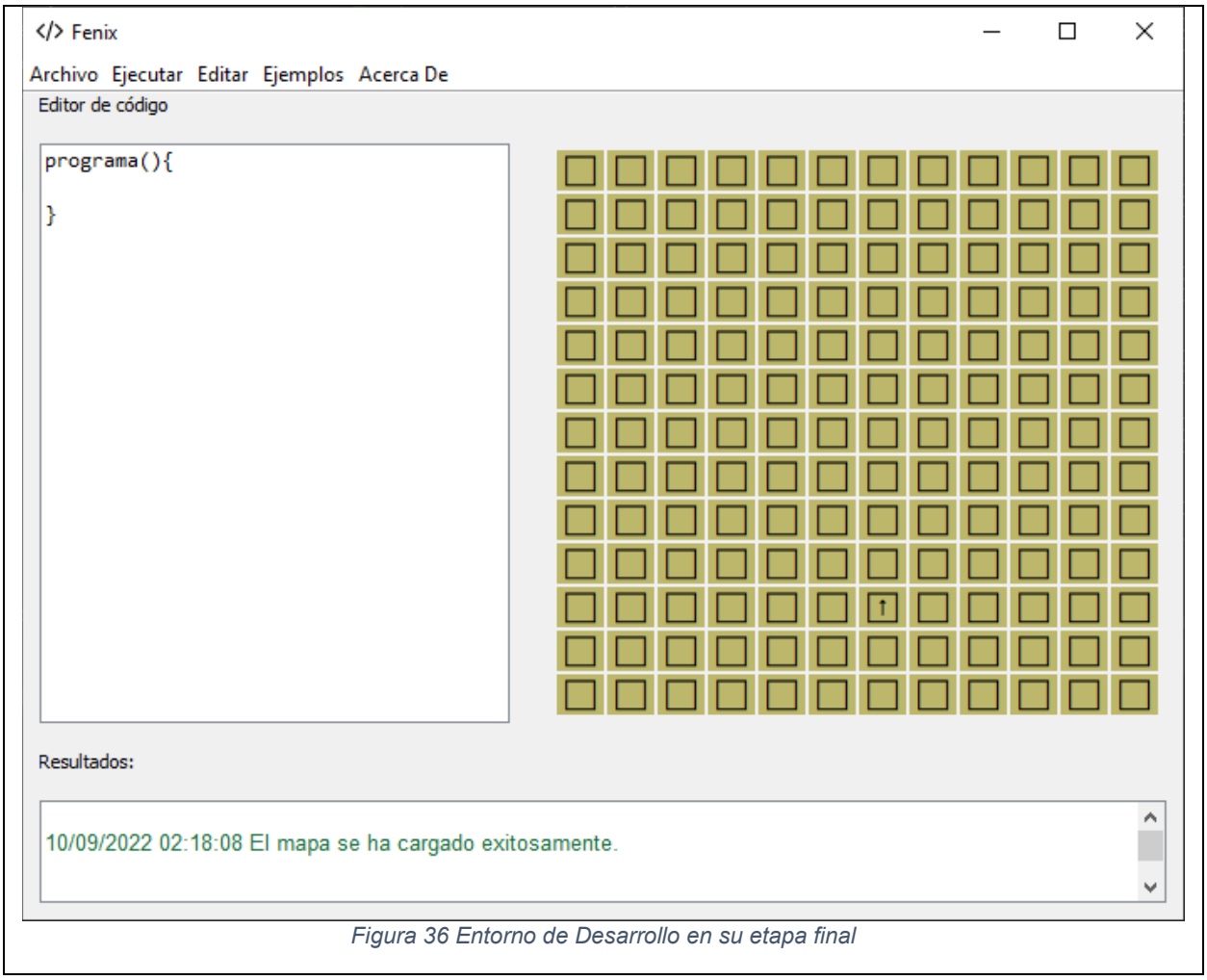

Figura 36 Entorno de Desarrollo en su etapa final

Con estos detalles finales, se finaliza el desarrollo de este proyecto.

A partir de este momento, todo el software entra en una etapa de mantenimiento.

9 Índice de Figuras

10 Índice de Enlaces