
LENGUAJES AUTÓTAS II

Profesor: Olga Gabriela Delgado Cansino



Compilador Léxico y Sintáctico



Integrantes del Equipo:

Andrea Jacqueline Bautista Salas

Guadalupe Nataly Ruiz Torres

Victor Emanuel Marquez Echevarria

Carlos Alejandro Villareal Lopez

Hugo Alejandro Quezada Rodríguez

Jesús Uriel Teran Vazquez

Índice

Desarrollo de un Compilador.....	1
Objetivo Principal.....	1
Tipos de Analizadores	2
Identificadores / Alfabeto:	2
Operadores:.....	3
○ Introducir Comentarios:	3
Delimitadores:.....	3
Palabras Reservadas: Comunes del lenguaje Java.	4
Palabras Reservadas: Proyecto compilador léxico.....	4
Error Léxico:	5
Proyecto: Descripción.....	5
Concepto: Analizador Sintáctico.....	6
Funciones: Analizador Sintáctico.....	6
Gramática utilizada por un analizador sintáctico.....	7
Derivaciones	8
Combinaciones	9
Notación BNF	10
Manual Técnico del Compilador	12
IDE para desarrollo del programa.....	12
Librería JFlex.jar	12
Estructura del Compilador: Elementos.	12
Creación y contenido:	13
Lexer.flex.....	13
Contenido: Lexer.flex y Tokens.java	14
Descripción: Lexer.flex.....	15
Descripción: Tokens.java	15
Nombres asignados de los operadores, palabras reservadas y delimitadores:	15
Descripción: generadora.java	16
Generar Lexer:	17
Precauciones:	17

Descripción: Lexer.java	18
LexerCup.flex.....	19
Syntax.cup	20
Botón Analizador Sintáctico:	21
Diseño de interfaz gráfica	22
Color de paneles y botones:.....	23
Áreas de texto:	24
Texto de etiquetas:.....	24
Función de los botones:.....	25
Error Léxico:.....	25
Ejemplo de funcionamiento de interfaz gráfica:.....	26
Manual de Usuario	27
Ejecutar programa:	27
Uso de Programa:.....	28
Nombres asignados de los operadores, palabras reservadas y delimitadores:	28
Elementos de la interfaz:	29

Desarrollo de un Compilador

Objetivo Principal.

El objetivo de este proyecto es realizar un compilador diseñando las herramientas necesarias para así poder elaborar un programa analizador de los datos ingresados por el usuario; en otras palabras: diseñaremos un programa con la capacidad de construir otros programas, de manera que, interprete un lenguaje de alto nivel (el del usuario), traduciéndolo a otro de bajo nivel (lenguaje máquina).

Tipos de Analizadores

Si un compilador tuviera que procesar sólo programas correctos, su diseño e implementación se simplificarían mucho. Las primeras versiones de los programas suelen ser incorrectas, y un buen compilador debería ayudar al programador a identificar y localizar errores. Es más, considerar desde el principio el manejo de errores puede simplificar la estructura de un compilador y mejorar su respuesta a los errores. Los errores en la programación pueden ser de los siguientes tipos:

- Léxicos, producidos al escribir mal un identificador, una palabra clave o un operador.
- Sintácticos, por una expresión aritmética o paréntesis no equilibrados.
- Semánticos, como un operador aplicado a un operando incompatible.
- Lógicos, puede ser una llamada infinitamente recursiva.
- De corrección, cuando el programa no hace lo que el programador realmente deseaba.

Los compiladores actuales se centran en el reconocimiento de los tres primeros tipos de errores; ya que resulta evidente que los errores de corrección no pueden ser detectados por un compilador, ya que en ellos interviene el concepto abstracto que el programador tiene sobre el programa que construye.

Identificadores / Alfabeto:

Abarca todas las letras del abecedario, desde la A a la Z, tanto mayúsculas como y minúsculas, y caracteres alfanuméricos, dígitos que van del 0 al 9:

$$[A,...,Z],[a,...,z],[0,...,9]$$

Operadores:

Incluiremos los caracteres del código ASCII, estos son:

Operador: Aquél que realiza un efecto de concatenación.

<code>< , > , << , >> = , >= , <= , != , * , & , && , % , , / , + , - , ++ , -- , == , ' , " , \ , :</code>

- **Introducir Comentarios:**

Sustituir `//` por `&&`:

Delimitadores:

<code>" , " , " , () , [] , { }</code>
--

Símbolos utilizados como separadores de las distintas construcciones de un lenguaje de programación.

- **() PARENTESIS:** Lista de parámetros en la definición y llamada a métodos, precedencia en expresiones para control de flujo y conversiones de tipo.
- **{ } LLAVES:** Inicialización de arrays, bloques de código, clases métodos y ámbitos locales.
- **[] CORCHETES:** Para uno de arreglos (arrays).
- **(";") PUNTO Y COMA:** Separador de sentencias.
- **(" , ") COMA:** Identificadores consecutivos en una declaración de variables y sentencias encadenadas dentro de una sentencia **for**.
- **(" . ") PUNTO:** Separado de nombres de paquetes, subpaquetes y clases; separador entre variables y métodos/miembros.

Palabras Reservadas: Comunes del lenguaje Java.

Emplearemos las palabras traducidas al español y convertidas al femenino.

Ejemplo: “**abstract**” cambiará a “**abstracta**”.

Switch: cambiara.

abstract	continue	for	new	switch
boolean	default	goto	null	synchronized
break	do	if	package	this
byte	double	implements	private	threadsafe
byvalue	else	import	protected	throw[s]
case	extends	instanceof	public	transient
catch	false	int	return	true
char	final	interface	short	try
class	finally	long	static	void
const	float	native	super	while
cast	future	generic	inner	
operator	outer	rest	var	

Palabras Reservadas: Proyecto compilador léxico.

Hasta el momento tenemos considerados los siguientes posibles caracteres, dejando espacios en blanco para nuevas consideraciones a emplear en nuestro proyecto.

<i>Java</i>	<i>Nuevo</i>	<i>Java</i>	<i>Nuevo</i>	<i>Java</i>	<i>Nuevo</i>
if	entonces	int	entera	boolean	boleana
else	ademas	char	caracter	break	rota
while	mientrasque	float	flotante	private	privada
for	para	double	doble		
new	nueva				

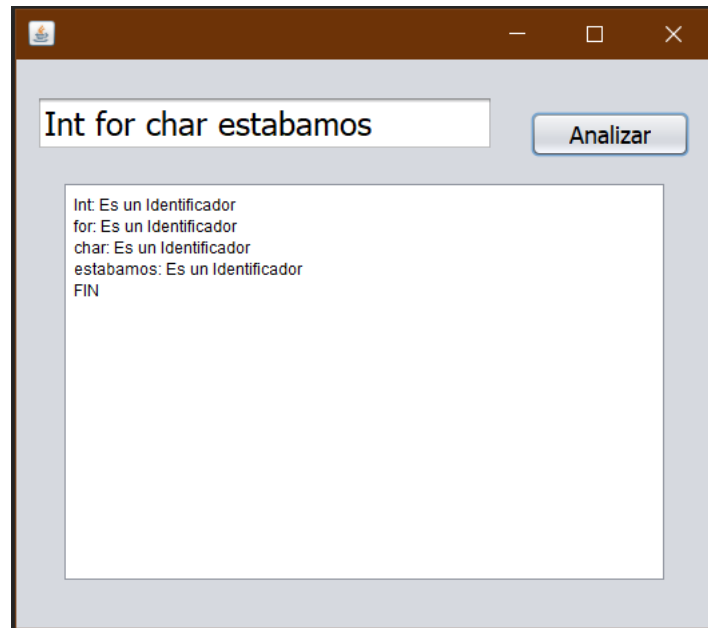
Error Léxico:

Al momento de no reconocer alguno de los símbolos introducidos al programa, éste emitirá un mensaje de alerta:

“Error! Símbolo no definido”.

Proyecto: Descripción.

El objetivo principal es realizar un analizador léxico el cual pueda reconocer los elementos que se le introducen. El ejemplo que se muestra a continuación está realizado con el lenguaje java.



Concepto: Analizador Sintáctico

Como concepto: es la fase del analizador que se encarga de chequear la secuencia de tokens que representa al texto de entrada, en base a una gramática dada. En caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce en base a una representación computacional. Este árbol es el punto de partida de la fase posterior de la etapa de análisis: el analizador semántico.

Funciones: Analizador Sintáctico

- Incorpora acciones semánticas en las que colocar el resto de las fases del compilador (excepto el analizador léxico): desde el análisis semántico hasta la generación de código.
- Informa de la naturaleza de los errores sintácticos que encuentra e intenta recuperarse de ellos para continuar la compilación.
- Controla el flujo de tokens reconocidos por parte del analizador léxico.

En definitiva, realiza casi todas las operaciones de la compilación, dando lugar a un método de trabajo denominado compilación dirigida por sintaxis.

El manejo de errores de sintaxis es el más complicado desde el punto de vista de la creación de compiladores. Nos interesa que cuando el compilador encuentre un error, no cancele definitivamente la compilación, sino que se recupere y siga buscando errores. Recuperar un error no quiere decir corregirlo, sino ser capaz de seguir construyendo el árbol sintáctico a pesar de los errores encontrados. En vista de esto, el manejador de errores de un analizador sintáctico debe tener como objetivos:

- Indicar los errores de forma clara y precisa. Debe informar mediante los correspondientes mensajes del tipo de error y su localización.
- Recuperarse del error, para poder seguir examinando la entrada.

- Distinguir entre errores y advertencias. Las advertencias se suelen utilizar para informar sobre sentencias válidas pero que, por ser poco frecuentes, pueden constituir una fuente de errores lógicos.
- No ralentizar significativamente la compilación.

Gramática utilizada por un analizador sintáctico

Gramática Propuesta:

① A	→	A + C
②		C
③ C	→	C * R
④		R
⑤ R	→	No
⑥		numero
⑦		(A)

Las derivaciones a izquierda quedarían: (primera manera de representarlo)

$A \rightarrow A + A \rightarrow A * A + A \rightarrow \text{No} * A + A \rightarrow \text{No}_1 * \text{No}_2 + A \rightarrow \text{No}_1 + \text{No}_2 + \text{No}_3$

Sustituyendo a la primera "A" quedaría secuencialmente de la siguiente manera:

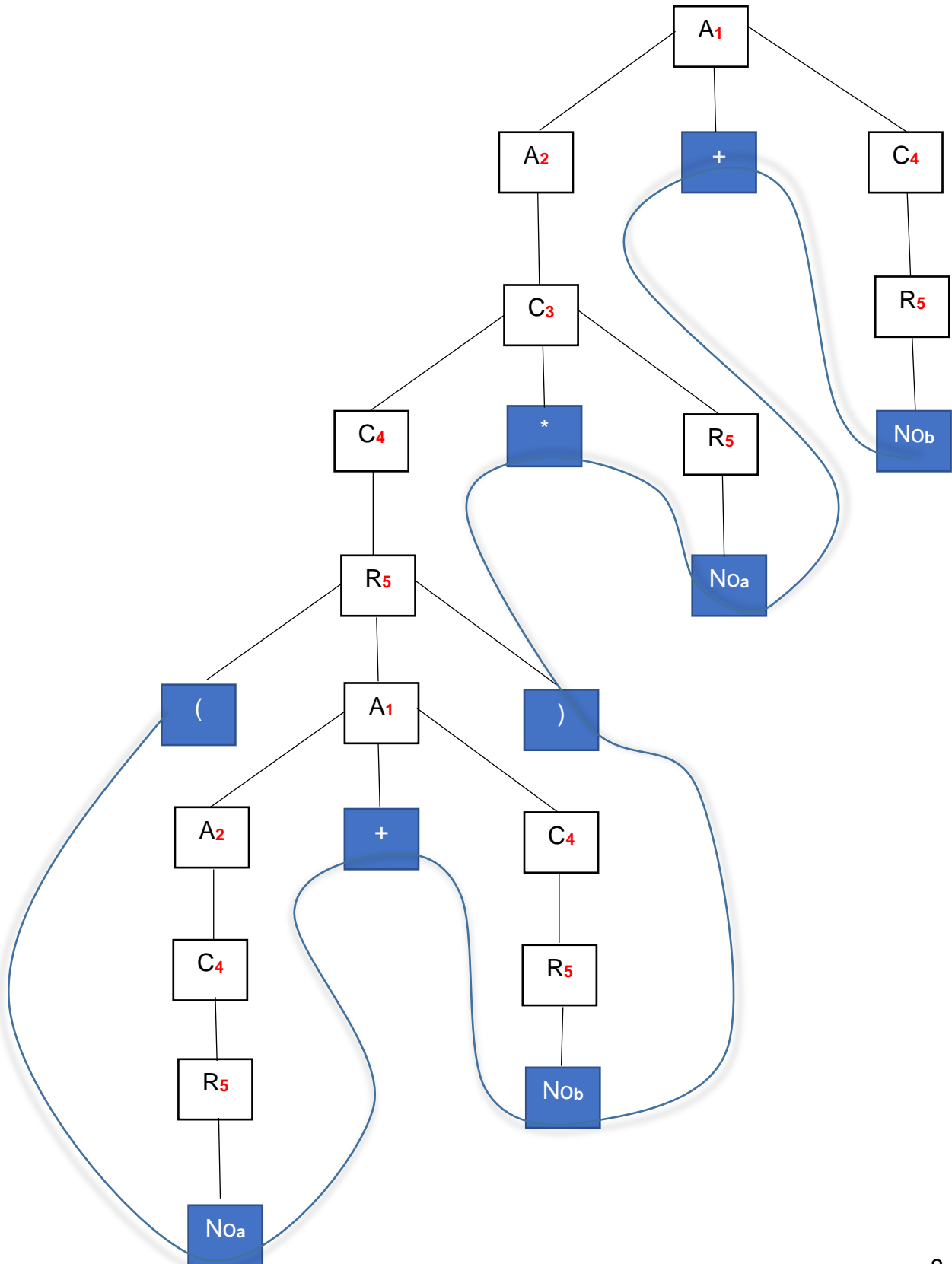
(Segunda manera de representarlo)

A → A+A

A	→	A	+	A
	→	A	*	A
	→	No	*	A
	→	No ₁	*	No ₂
	→	No ₁	*	No ₂
	→	No ₁	*	No ₂
	→	No ₁	*	No ₂

Derivaciones

$$(No_a + No_b) * No_a + No_b$$

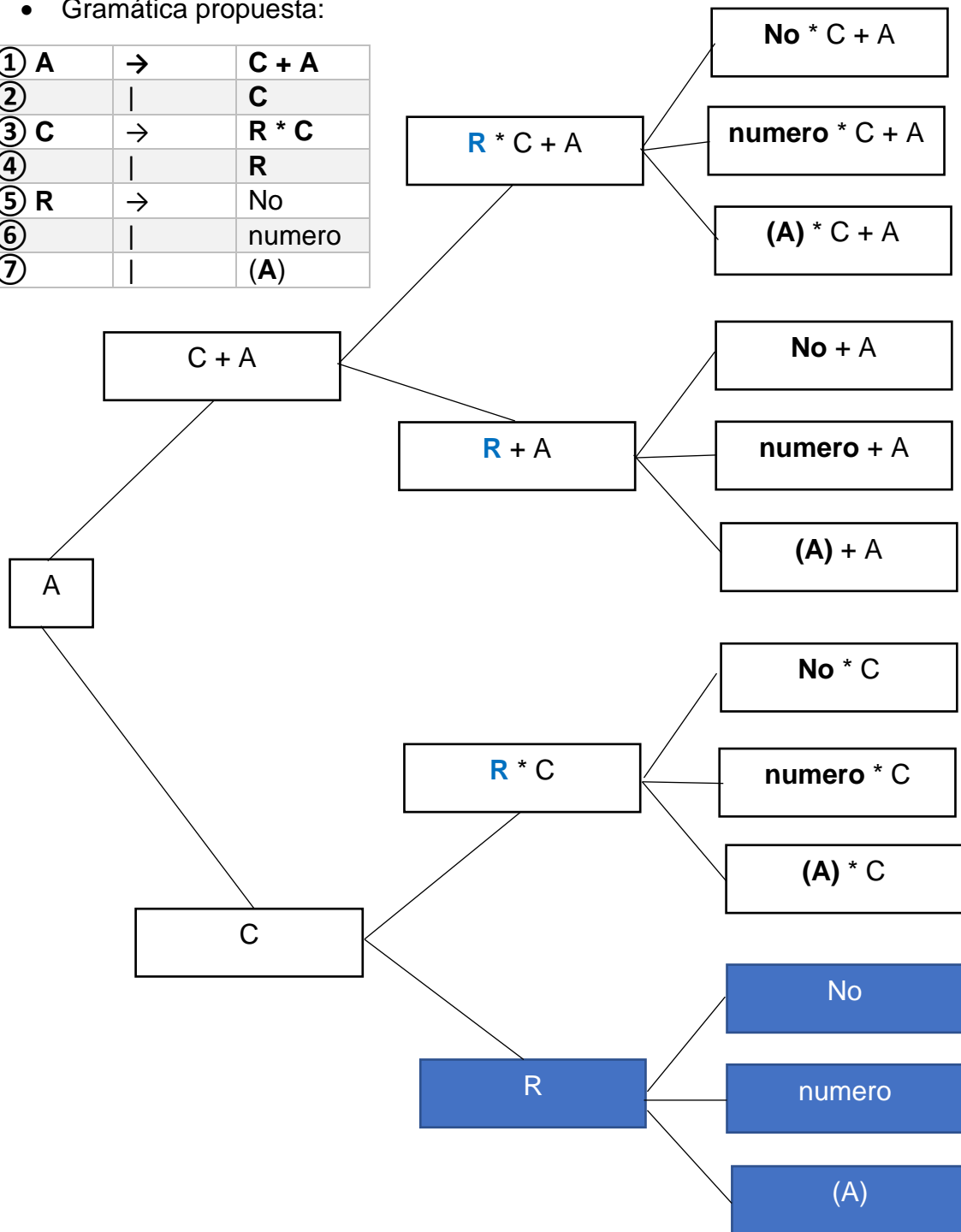


Combinaciones

Se pueden combinar todas las posibles sentencias reconocibles por la gramática propuesta que consiste en reemplazar con valores que un elemento representa.

- Gramática propuesta:

①	A	→	C + A
②			C
③	C	→	R * C
④			R
⑤	R	→	No
⑥			numero
⑦			(A)



Notación BNF

Para la descripción de un lenguaje de programación.

<programa>	=	<listaDeFunciones>
<listaDeFunciones>	=	<función> <listaDeFunciones> <funcion>
<funcion>	=	FUNC<variable>(<ListaDeParametros>)<Sentencia>
<listadeDeParametros>	=	<ListaDeVariable> ε
<ListaDeVariable>	=	<variable> <ListaDeVariables> , <variable>
<alfanumerico>	=	<letra> <digito>
<letra>	=	a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
<digito>	=	0 1 2 3 4 5 6 7 8 9
<sentencia>	=	<sentenciaDeAsignacion> <sentenciaDeRetorno> <sentenciaDelImpresion> <sentenciaNula> <sentenciaCondicional> <sentenciaWhile> <bloque>
<sentenciaDeAsignacion>	=	<variable> = <expresion>
<expresión>	=	<expresión> <operadorBinario> <expresión> <operadorUnario> <expresion> (<expresion>) <entero> <variable> (<listaDeArgumentos>)
<operadorBinario>	=	+ - * / ~
<operadorUnario>	=	-
<operadorIgualA>	=	==
<operadorMenorQue>	=	<
<operadorMayorQue>	=	>
<operadorMayorOigualA>	=	>=
<operadorMenorOigualA>	=	<=
<operadorDesplazamientoIzquierda>	=	<<
<operadorDesplazamientoDerecha>	=	>>
<operadorDiferenteA>	=	!=
<operadorTambien>	=	&
<operadorComentario>	=	&&
<operadorModula>	=	%
<operadorComentada>	=	#
<operadorObien>	=	
<operadorIncremento>	=	++
<operadorDecremento>	=	--
<operadorBinarioABinario>	=	~

<operadorComilladoSimple>	=	'
<operadorComilladoDoble>	=	"
<operadorContinuaY>	=	^
<operadorComa>	=	,
<operadorPuntoYcoma>	=	;
<operadorParentesisA>	=	(
<operadorParentesisB>	=)
<operadorCorcheteA>	=	[
<operadorCorcheteB>	=]
<operadorLlaveA>	=	{
<operadorLlaveB>	=	}
<entero>	=	<digito> <entero><digito>
<listaDeArgumentos>	=	<listaDeExpresiones> <expresiones> , <listaDeExpresiones>
<sentenciaDelImpresion>	=	PRINT<listaDelImpresion> <listaDelImpresion>,<elementoDelImpresion>
<elementoDelImpresion>	=	<Expresion> "<Texto>"
<texto>	=	<caracter> <caracter><texto>
<carácter>	=	<caracterImprimible> <caracterEscapado>
<caracterImprimible>	=	"Cualquier carácter ASCII imprimible"
<caracterEscapado>	=	\n
<sentenciaDeRetorno>	=	RETURN <expresion>
<sentenciaCondicional>	=	If <expresion> then <sentencia> fi if <expresion> then <sentencia> else <sentencia> entonces <sentencia> fi if <expresion> then <sentencia> else <sentencia> entonces <sentencia> ademas <sentencia> fi for <expresión> then <sentencia> fi for <expresión> then <sentencia> para <sentencia> fi
<sentenciaWhile>	=	while <expresion> do <sentencia> done
<bloque>	=	{<listaDeDeclaraciones><listaDeSentencias>}
<listaDeDeclaraciones>	=	<declaración><listaDeDeclaraciones> ε
<declaracion>	=	VAR<listaDeVariables>
<listaDeSentencias>	=	<sentencia> <listaDeSentencias><sentencia>

Manual Técnico del Compilador

IDE para desarrollo del programa

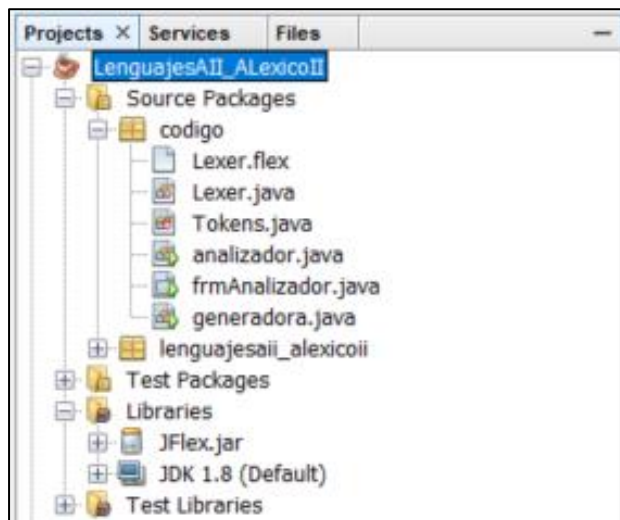
Descargamos e instalamos la versión de Java NetBeans 8.2.

(**Nota:** En caso de fallos, tener cuidado con la versión de jdk que se está ejecutando desde java NetBeans 8.2. Procurar que esta versión no sea la más reciente).

Librería JFlex.jar

Descargaremos la librería JFlex.jar y se tomará en cuenta la dirección donde éste se alojará.

Estructura del Compilador: Elementos.



El Programa está compuesto por un paquete de nombre “código”. Dentro contiene una serie de clases, una librería, un archivo anteriormente vacío, y un JFrame.

Archivo (anteriormente vacío):

- Lexer.flex

Clases:

- Tokens.java
- generadora.java
- Lexer.java

Librería:

- jFlex.jar

JFrame:

- frmAnalizador.java

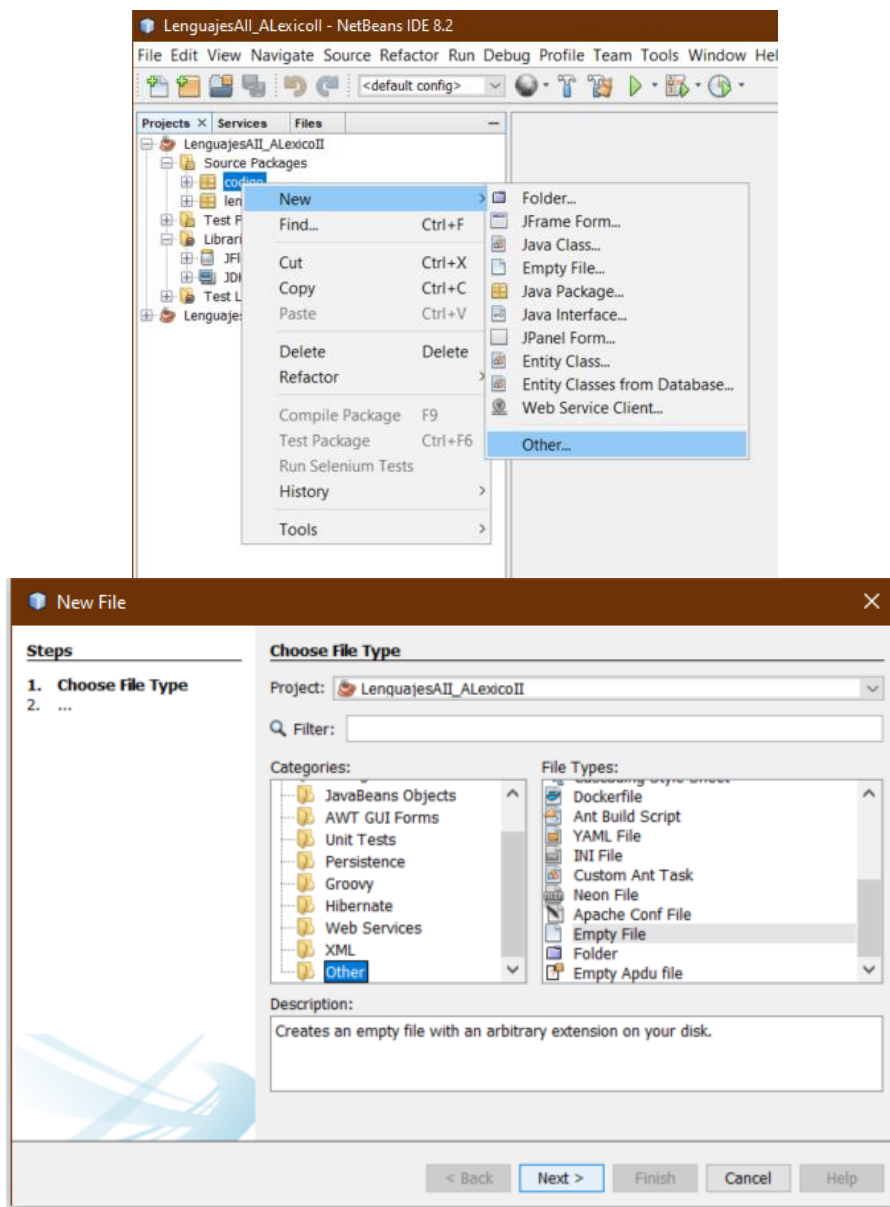
Creación y contenido:

Lexer.flex

Creación de documento:

Una vez creado el proyecto y el paquete “código”; presionaremos el segundo botón del ratón para desglozar las siguientes opciones:

Crear un archivo nuevo desde: new / other / other / Empty File.



Contenido: Lexer.flex y Tokens.java

Lexer.flex	Tokens.java
<pre> package codigo; import static codigo.Tokens.*; %% %class Lexer %type Tokens L=[a-zA-Z_]+ D=[0-9]+ espacio=[\t,\r]+ %{ public String lexeme; }% %% if {lexeme = yytext(); return entonces;} else {lexeme = yytext(); return ademas;} while {lexeme=yytext(); return mientrasque;} for {lexeme=yytext(); return para;} int {lexeme = yytext(); return entera;} char {lexeme = yytext(); return caracter;} float {lexeme = yytext(); return flotante;} double {lexeme = yytext(); return doble;} boolean {lexeme = yytext(); return booleana;} {espacio} {/*Ignore*/} "/" {return Linea;} "=" {lexeme = yytext(); return asignaA;} "+" {lexeme = yytext(); return Suma;} "-" {lexeme = yytext(); return Resta;} "*" {lexeme = yytext(); return Multiplicacion;} "/" {lexeme = yytext(); return Division;} "==" {lexeme = yytext(); return igualA;} "<" {lexeme = yytext(); return menorQue;} ">" {lexeme = yytext(); return mayorQue;} ">=" {lexeme = yytext(); return mayorOigualA;} "<=" {lexeme = yytext(); return menorOigualA;} "<<" {lexeme = yytext(); return desplazal Izquierda;} ">>" {lexeme = yytext(); return desplaza Derecha;} "!=" {lexeme = yytext(); return diferenteA;} "&" {lexeme = yytext(); return tambien;} "&&" {lexeme = yytext(); return comentario;} "%" {lexeme = yytext(); return modula;} " " {lexeme = yytext(); return obien;} "++" {lexeme = yytext(); return incremento;} "--" {lexeme = yytext(); return decremento;} "'" {lexeme = yytext(); return comillado Simple;} "\"" {lexeme = yytext(); return comillado DobleA;} "." {lexeme = yytext(); return dosPuntos;} "," {lexeme = yytext(); return coma;} ";" {lexeme = yytext(); return puntoYcoma;} "(" {lexeme = yytext(); return parentesisA;} ")" {lexeme = yytext(); return parentesisB;} "[" {lexeme = yytext(); return corcheteA;} "]" {lexeme = yytext(); return corcheteB;} "{" {lexeme = yytext(); return llaveA;} "}" {lexeme = yytext(); return llaveB;} {L}{D}* {lexeme=yytext(); return Identificador;} {"(-{D}+)" {D}+ {lexeme=yytext(); return Numero;} . {return ERROR;} </pre>	<pre> package codigo; public enum Tokens { Reservadas, entonces, ademas, mientrasque, para, entera, caracter, flotante, doble, booleana, Linea, asignaA, Igual, Suma, Resta, Multiplicacion, Division, igualA, menorQue, mayorQue, mayorOigualA, menorOigualA, desplazal Izquierda, desplaza Derecha, diferenteA, tambien, comentario, modula, obien, incremento, decremento, comillado Simple, comillado DobleA, dosPuntos, coma, puntoYcoma, parentesisA, parentesisB, corcheteA, corcheteB, llaveA, llaveB, Identificador, Numero, ERROR } </pre>

Descripción: Lexer.flex

En el archivo Lexer.flex se redacta los operadores, delimitadores y palabras reservadas, mediante “ **lexeme = yytext();** ” y donde después de “ **return** ” se describe el elemento que devolverá.

Descripción: Tokens.java

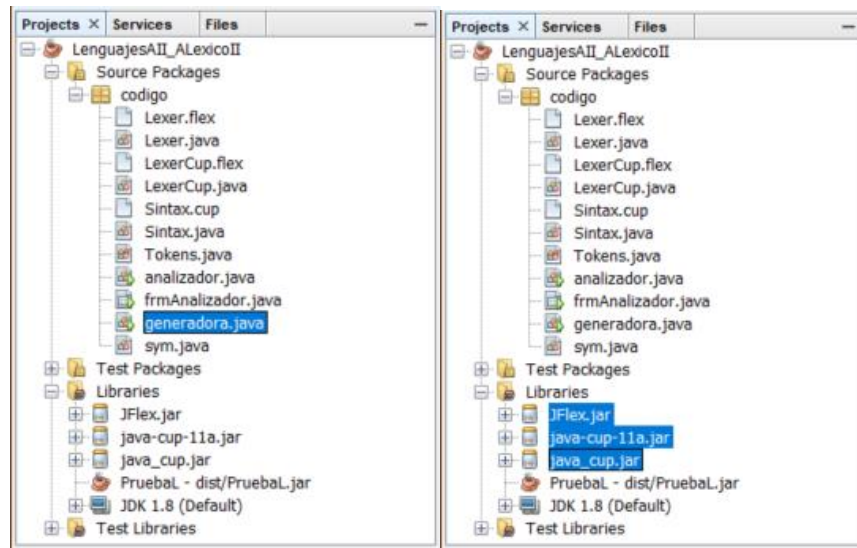
Se crea el método “Tokens” donde se incluyen los nombres que se asignaron en el archivo Lexer.flex para así poder ser llamados cuando se les requiera.

Nombres asignados de los operadores, palabras reservadas y delimitadores:

<ul style="list-style-type: none">• if → entonces• else → ademas• while → mientrasque• for → para• int → entera• char → caracter• float → flotante• double → doble• boolean → boleana• "=" → asignaA• "+" → Suma• "-" → Resta• "*" → Multiplicacion• "/" → Division• "==" → igualA• "<" → menorQue• ">" → mayorQue• ">=" → mayorOigualA• "<=" → menorOigualA	<ul style="list-style-type: none">• "<<" → desplazalzquierda• ">>" → desplazaDerecha• "!=" → diferenteA• "&" → tambien• "&&" → comentario• "%" → modula• " " → obien• "++" → incremento• "--" → decremento• " " " → comilladoSimple• " " " " → comilladoDobleA• ":" → dosPuntos• "," → coma• ";" → puntoYcoma• "(" → parentesisA• ")" → parentesisB• "[" → corcheteA• "]" → corcheteB• "{" → llaveA• "}" → llaveB
--	--

Descripción: generadora.java

Éste es un archivo ejecutable que contiene dentro de sí la ruta para acceder a la librería “JFlex.jar”, con el cual mediante el método **generarLexer** y la librería importada “**java.io.File**”, se tendrá acceso a la librería JFlex.jar.



Importamos dentro de las librerías **jFlex.jar**, **java-cup11a.jar**, y **java_cup.jar**.

Generar Lexer:

```
generadora.java x
Source History
1 package codigo;
2
3 import java.io.File;
4 import java.io.IOException;
5 import java.nio.file.Files;
6 import java.nio.file.Path;
7 import java.nio.file.Paths;
8
9 public class generadora {
10     public static void main(String[] args) throws Exception
11     {
12         String ruta = "C:\\Users\\Hugo Quezada\\OneDrive\\1.JavaProyects\\LenguajesAII_AlexicoII\\src\\codigo\\Lexer.flex";
13         String ruta2 = "C:\\Users\\Hugo Quezada\\OneDrive\\1.JavaProyects\\LenguajesAII_AlexicoII\\src\\codigo\\LexerCup.flex";
14         String[] rutaS = {"-parser", "Syntax", "C:\\Users\\Hugo Quezada\\OneDrive\\1.JavaProyects\\LenguajesAII_AlexicoII\\src\\codigo\\syntax.cup"};
15         generarLexer(ruta, ruta2, rutaS);
16     }
17     public static void generarLexer(String ruta, String ruta2, String[] rutaS) throws IOException, Exception{
18         File archivo;
19         archivo = new File(ruta);
20         JFlex.Main.generate(archivo);
21         archivo = new File(ruta2);
22         JFlex.Main.generate(archivo);
23         java_cup.Main.main(rutaS);
24
25         Path rutaSym = Paths.get("C:\\Users\\Hugo Quezada\\OneDrive\\1.JavaProyects\\LenguajesAII_AlexicoII\\src\\codigo\\sym.java");
26         if (Files.exists(rutaSym)) {
27             Files.delete(rutaSym);
28         }
29         Files.move(
30             Paths.get("C:\\Users\\Hugo Quezada\\OneDrive\\1.JavaProyects\\LenguajesAII_AlexicoII\\src\\codigo\\sym.java"),
31             Paths.get("C:\\Users\\Hugo Quezada\\OneDrive\\1.JavaProyects\\LenguajesAII_AlexicoII\\src\\codigo\\sym.java")
32         );
33         Path rutaSin = Paths.get("C:\\Users\\Hugo Quezada\\OneDrive\\1.JavaProyects\\LenguajesAII_AlexicoII\\src\\codigo\\Syntax.java");
34         if (Files.exists(rutaSin)) {
35             Files.delete(rutaSin);
36         }
37         Files.move(
38             Paths.get("C:\\Users\\Hugo Quezada\\OneDrive\\1.JavaProyects\\LenguajesAII_AlexicoII\\src\\codigo\\Syntax.java"),
39             Paths.get("C:\\Users\\Hugo Quezada\\OneDrive\\1.JavaProyects\\LenguajesAII_AlexicoII\\src\\codigo\\Syntax.java")
40         );
41     }
42 }
```

En este apartado, conforme a las rutas; se generan los archivos con terminación “.java”, como: **lexer.java**, **lexer.java** y finalmente **sym.java**. En los cuales se establecen los símbolos predefinidos por el programador. También, cada que se realicen cambios a los archivos **Lexer.flex**, **Tokens**, se tendrá que volver a ejecutar el programa para que éste se actualice.

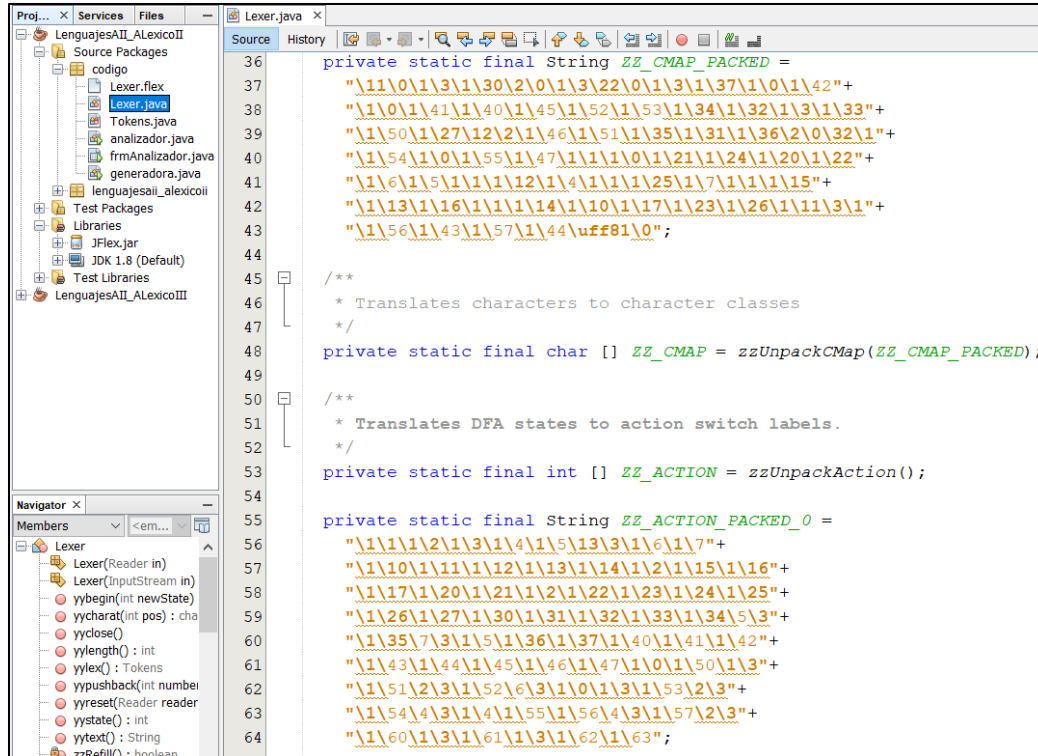
Precauciones:

Como se observa, tenemos rutas muy similares, que generan el mismo archivo en diferentes lugares. Si se descuida, el programa buscará generarlos en dentro de la misma carpeta generando conflicto, y por ello, es necesario crear la ruta donde estos no coincidan.

```
Files.move(
    Paths.get("C:\\Users\\Hugo Quezada\\OneDrive\\1.JavaProyects\\LenguajesAII_AlexicoII\\src\\codigo\\sym.java"),
    Paths.get("C:\\Users\\Hugo Quezada\\OneDrive\\1.JavaProyects\\LenguajesAII_AlexicoII\\src\\codigo\\sym.java")
);
```

Descripción: Lexer.java

Una vez ejecutado el programa **generadora.java** automáticamente se generará el archivo: **Lexer.java**.



```
36 private static final String ZZ_CMAP_PACKED =
37     "\1\0\1\3\1\30\2\0\1\3\22\0\1\3\1\37\1\0\1\42"+
38     "\1\0\1\41\1\40\1\45\1\52\1\53\1\34\1\32\1\3\1\33"+
39     "\1\50\1\27\1\2\1\46\1\51\1\35\1\31\1\36\2\0\32\1"+
40     "\1\54\1\0\1\55\1\47\1\1\1\0\1\21\1\24\1\20\1\22"+
41     "\1\6\1\5\1\1\1\12\1\4\1\1\1\25\1\7\1\1\1\15"+
42     "\1\13\1\16\1\1\1\14\1\10\1\17\1\23\1\26\1\11\3\1"+
43     "\1\56\1\43\1\57\1\44\ufff81\0";
44
45 /**
46  * Translates characters to character classes
47  */
48 private static final char [] ZZ_CMAP = zzUnpackCMap(ZZ_CMAP_PACKED);
49
50 /**
51  * Translates DFA states to action switch labels.
52  */
53 private static final int [] ZZ_ACTION = zzUnpackAction();
54
55 private static final String ZZ_ACTION_PACKED_0 =
56     "\1\1\1\2\1\3\1\4\1\5\1\13\3\1\6\1\7"+
57     "\1\10\1\11\1\12\1\13\1\14\1\2\1\15\1\16"+
58     "\1\17\1\20\1\21\1\2\1\22\1\23\1\24\1\25"+
59     "\1\26\1\27\1\30\1\31\1\32\1\33\1\34\5\3"+
60     "\1\35\7\3\1\5\1\36\1\37\1\40\1\41\1\42"+
61     "\1\43\1\44\1\45\1\46\1\47\1\0\1\50\1\3"+
62     "\1\51\2\3\1\52\6\3\1\0\1\3\1\53\2\3"+
63     "\1\54\4\3\1\4\1\55\1\56\4\3\1\57\2\3"+
64     "\1\60\1\3\1\61\1\3\1\62\1\63";
```

LexerCup.flex

Este apartado lo creamos describiendo todas las palabras reservadas y los operadores con el fin de describirle a nuestro programa las simbologías y el tipo al que corresponden. Como se muestra en la vista siguiente:

LexerCup.flex	LexerCup.java
<pre>1 package codigos; 2 import java_cup.runtime.Symbol; 3 %% 4 %class LexerCup 5 %type java_cup.runtime.Symbol 6 %cup 7 %full 8 %line 9 %char 10 L=[a-zA-Z_]+ 11 D=[0-9]+ 12 espacio=[\t,\r,\n]+ 13 %{ 14 private Symbol symbol(int type, Object value){ 15 return new Symbol(type, yyline, yycolumn, value); 16 } 17 private Symbol symbol(int type){ 18 return new Symbol(type, yyline, yycolumn); 19 } 20 %} 21 %% 22 /* Espacios en blanco */ 23 (espacio) { /* ignore */ } 24 /* Comentarios */ 25 (/*(/*.*) /* ignore */ } 26 /* Tipos de datos */ 27 /* Palabra reservada entonces */ 28 (if) { return new Symbol(sym.entonces, yychar, yyline, yytext()); } 29 /* Palabra reservada ademas */ 30 (else) { return new Symbol(sym.ademas, yychar, yyline, yytext()); } 31 /* Palabra reservada mientrasque */ 32 (while) { return new Symbol(sym.mientrasque, yychar, yyline, yytext()); } 33 /* Palabra reservada para */ 34 (for) { return new Symbol(sym.para, yychar, yyline, yytext()); } 35 /* Palabra reservada entera */ 36 (int) { return new Symbol(sym.entera, yychar, yyline, yytext()); } 37 /* Palabra reservada doble */ 38 (double) { return new Symbol(sym.doble, yychar, yyline, yytext()); } 39 40 /* Operador asigna */ 41 ("=") { return new Symbol(sym.asigna, yychar, yyline, yytext()); } 42 /* Operador suma */ 43 ("+") { return new Symbol(sym.suma, yychar, yyline, yytext()); } 44 /* Operador resta */ 45 ("-") { return new Symbol(sym.resta, yychar, yyline, yytext()); }</pre>	<pre>620 621 // store back cached position 622 zzMarkedPos = zzMarkedPosL; 623 624 switch (zzAction < 0 ? zzAction : ZZ_ACTION[zzAction]) { 625 case 37: 626 { return new Symbol(sym.doble, yychar, yyline, yytext()); } 627 } 628 case 38: break; 629 case 3: 630 { return new Symbol(sym.Identificador, yychar, yyline, yytext()); } 631 } 632 case 39: break; 633 case 20: 634 { return new Symbol(sym.tambien, yychar, yyline, yytext()); } 635 } 636 case 40: break; 637 case 14: 638 { return new Symbol(sym.parentesisB, yychar, yyline, yytext()); } 639 } 640 case 41: break; 641 case 10: 642 { return new Symbol(sym.Multiplicacion, yychar, yyline, 643 yytext()); } 644 } 645 case 42: break; 646 case 15: 647 { return new Symbol(sym.llaveA, yychar, yyline, yytext()); } 648 } 649 case 43: break; 650 case 29: 651 { return new Symbol(sym.desplazaIzquierda, yychar, yyline, yytext()); } 652 } 653 case 44: break; 654 case 23: 655 { return new Symbol(sym.corcheteA, yychar, yyline, yytext()); } 656 } 657 case 45: break; 658 case 4: 659 { return new Symbol(sym.Numero, yychar, 660 yyline, yytext()); }</pre>

Éste tendrá efecto una vez que sea ejecutado nuestro archivo “generadora”.

En la terminal, el código al que tiene acceso con **LexerCup.java** se encuentra dentro de nuestro botón analizador Léxico.

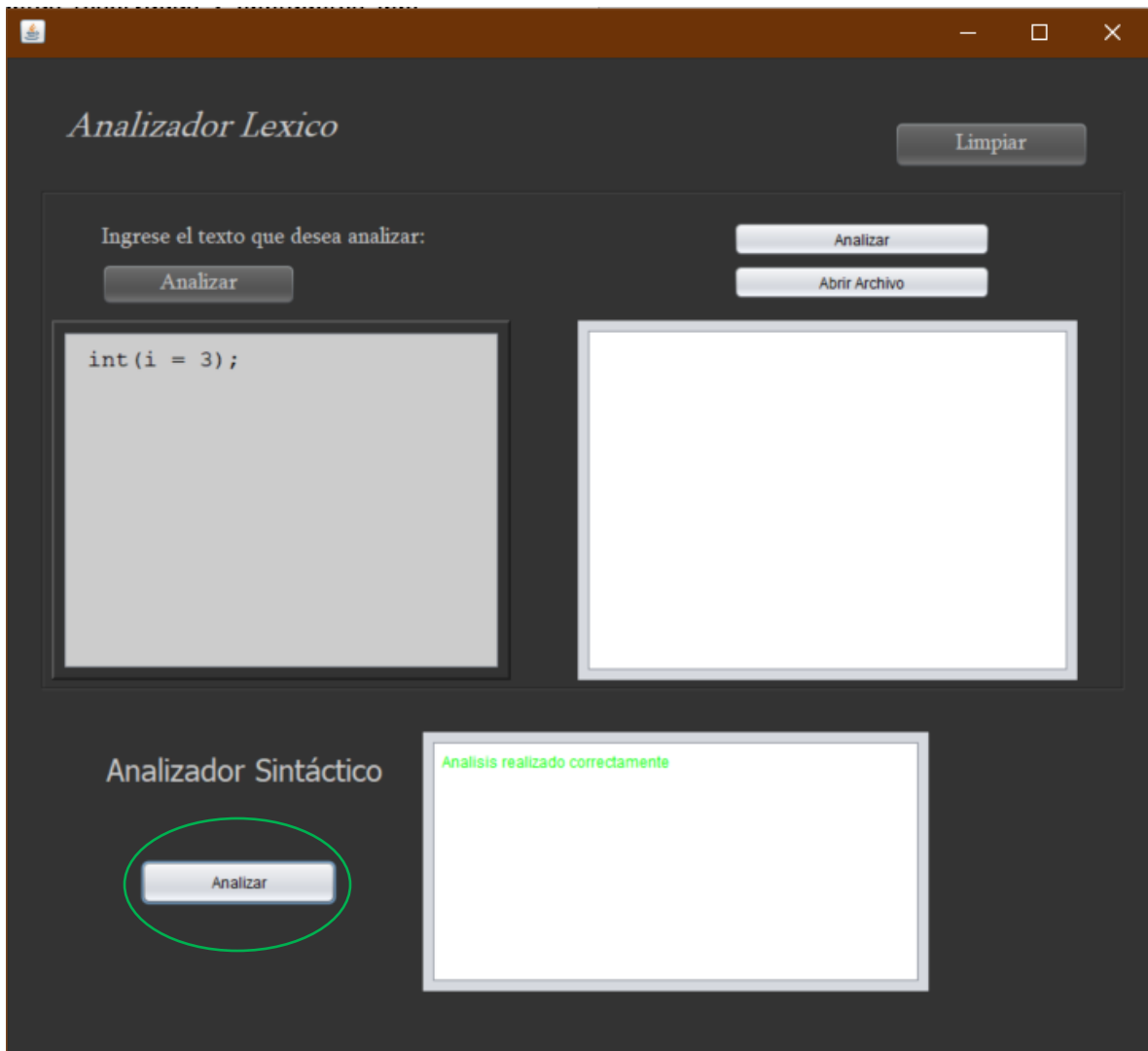
```
private void btnAnalizador2ActionPerformed(java.awt.event.ActionEvent evt) {
    try {
        analizadorLexico();
    } catch (IOException ex) {
        Logger.getLogger(frmAnalizador.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Syntax.cup

Syntax.cup	Syntax.java
<pre> 1 package codigo; 2 import java_cup.runtime.Symbol; 3 parser code 4 { 5 private Symbol s; 6 7 public void syntax_error(Symbol s) { 8 this.s = s; 9 } 10 public Symbol get\$() { 11 return this.s; 12 } 13 ;; 14 15 terminal Lines, entonces, ademas, mientrasque, para, entera, caracter, flotante, 16 doble, boolean, Main, 17 asignaA, suma, resta, Multiplicacion, Division, 18 igualA, menorQue, mayorQue, mayorOigualA, menorOigualA, 19 desplazaIzquierda, desplazaDerecha, diferenteA, tambien, 20 comentario, modulo, obien, incremento, decremento, 21 comilladoSimple, comilladoDobleA, dosPuntos, coma, puntoYcoma, 22 parentesisA, parentesisB, corcheteA, corcheteB, llaveA, llaveB, 23 Identificador, Numero, ERROR; 24 25 non terminal INICIO, SENTENCIAint, SENTENCIAfor, SENTENCIAwhile; 26 27 start with INICIO; 28 INICIO ::= 29 SENTENCIAint 30 SENTENCIAfor 31 SENTENCIAwhile 32 ; 33 SENTENCIAint ::= 34 entera parentesisA Identificador asignaA Numero parentesisB puntoYcoma 35 ; 36 37 SENTENCIAfor ::= 38 para parentesisA Identificador asignaA Numero parentesisB puntoYcoma 39 ; 40 41 SENTENCIAwhile ::= 42 mientrasque parentesisA Identificador asignaA Numero parentesisB puntoYcoma 43 ; </pre>	<pre> 142 143 144 Object RESULT =null; 145 146 CUP\$Syntax\$result = parser.getSymbolFactory().newSymbol("SENTENCIAwhile",1 147) 148 149 return CUP\$Syntax\$result; 150 151 /* */ 152 case 6: // SENTENCIAfor ::= para parentesisA Identificador asignaA Numero para 153 { 154 Object RESULT =null; 155 156 CUP\$Syntax\$result = parser.getSymbolFactory().newSymbol("SENTENCIAfor",2, 157) 158 return CUP\$Syntax\$result; 159 160 /* */ 161 case 5: // SENTENCIAint ::= entera parentesisA Identificador asignaA Numero pa 162 { 163 Object RESULT =null; 164 165 CUP\$Syntax\$result = parser.getSymbolFactory().newSymbol("SENTENCIAint",1, 166) 167 return CUP\$Syntax\$result; 168 169 /* */ 170 case 4: // INICIO ::= 171 { 172 Object RESULT =null; 173 174 CUP\$Syntax\$result = parser.getSymbolFactory().newSymbol("INICIO",0, ((java 175) 176 return CUP\$Syntax\$result; 177 178 /* */ 179 case 3: // INICIO ::= SENTENCIAwhile 180 { 181 Object RESULT =null; 182 183 CUP\$Syntax\$result = parser.getSymbolFactory().newSymbol("INICIO",0, ((java 184) 185 return CUP\$Syntax\$result; 186 </pre>

Como se observa; ingresamos nuestras palabras reservadas y operadores que serán utilizados a través de nuestro analizador Sintáctico, y será ejecutado con nuestra clase “**generadora.java**”; esto para generar nuestro archivo “**Syntax.java**”, donde se registran nuestras palabras asignadas que darán inicio a las sentencias establecidas, ejemplo:

INICIO ::= SENTENCIAint ;	SENTENCIAint ::= entera parentesisA Identificador asignaA Numero parentesisB puntoYcoma ;
Y esto quiere decir: que nuestra <u>SENTENCIAint</u> equivale a = int (i = 3) ;	Y lo anterior es lo mismo a: SENTENCIA int ::= int (i = 3) ; ;



Botón Analizador Sintáctico:

Éste se encuentra dentro en la programación del botón “Analizar”, de nuestra terminal.

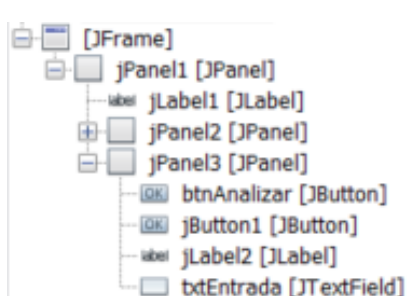
```
private void btbAnalizador3ActionPerformed(java.awt.event.ActionEvent evt) {
    String ST = txtEntrada.getText();
    Sintax s = new Sintax(new codigo.LexerCup(new StringReader(ST)));
    try {
        s.parse();
        txtSalida2.setText("Análisis realizado correctamente");
        txtSalida2.setForeground(Color.green);
    } catch (Exception ex) {
        Symbol sym = s.getS();
        txtSalida2.setText("Error en la sintaxis. Línea: " + (sym.right+1) + " texto:\"\" + sym.value + \"\" );
        txtSalida2.setForeground(Color.red);
        //Logger.getLogger(frmAnalizador.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```


Symbol: botón Analizador Sintáctico	Symbol: Syntax.java
<pre> private void btnAnalizador3ActionPerformed String ST = txtEntrada.getText(); Syntax s = new Syntax(new codigo.LexerCup try { s.parse(); txtSalida2.setText("Análisis realizado"); txtSalida2.setForeground(Color.green); } catch (Exception ex) { Symbol sym = s.getS(); txtSalida2.setText("Error en la sí"); txtSalida2.setForeground(Color.red); //Logger.getLogger(frmAnalizador.clas } </pre>	<pre> 127 /** Method with the actual generated action code. */ 128 public final java_cup.runtime.Symbol CUP\$Syntax\$do_action(129 int CUP\$Syntax\$act_num, 130 java_cup.runtime.lr_parser CUP\$Syntax\$parser, 131 java.util.Stack CUP\$Syntax\$stack, 132 int CUP\$Syntax\$stop, 133 throws java.lang.Exception 134) { 135 /* Symbol object for return from actions */ 136 java_cup.runtime.Symbol CUP\$Syntax\$result; 137 138 /* select the action based on the action number */ 139 switch (CUP\$Syntax\$act_num) 140 { 141 /* ... */ 142 case 7: // SENTENCIAwhile ::= mientrasque parentesisA Identificador asignaA Numero parentesisB puntoYcoma 143 { 144 Object RESULT =null; 145 CUP\$Syntax\$result = parser.getSymbolFactory().newSymbol("SENTENCIAwhile",3, ((java_cup.runtime.Symbol)CUP\$Syntax\$stack.e1 146) 147); 148 return CUP\$Syntax\$result; 149 } 150 /* ... */ 151 case 6: // SENTENCIAfor ::= para parentesisA Identificador asignaA Numero parentesisB puntoYcoma 152 { 153 Object RESULT =null; 154 CUP\$Syntax\$result = parser.getSymbolFactory().newSymbol("SENTENCIAfor",2, ((java_cup.runtime.Symbol)CUP\$Syntax\$stack.e1 155) 156); 157 return CUP\$Syntax\$result; 158 } 159 /* ... */ 160 case 5: // SENTENCIAint ::= entera parentesisA Identificador asignaA Numero parentesisB puntoYcoma 161 { 162 Object RESULT =null; 163 CUP\$Syntax\$result = parser.getSymbolFactory().newSymbol("SENTENCIAint",1, ((java_cup.runtime.Symbol)CUP\$Syntax\$stack.e1 164) 165); 166 return CUP\$Syntax\$result; </pre>

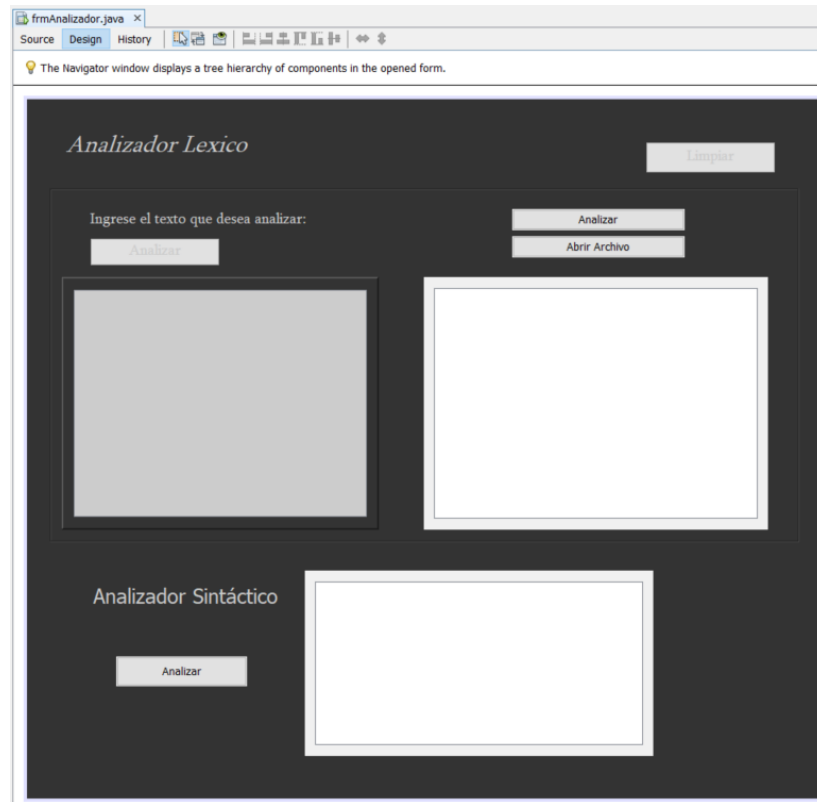
Y es de donde se accede a nuestro Syntax.java generado por nuestra “**generadora.java**” y analiza si el texto ingresado a nuestra cosola corresponde a lo establecido en nuestro **Syntax.cup**.

Diseño de interfaz gráfica

Se generó una interfaz gráfica con nombre: “frmAnalizador.java”. Y tiene las siguientes características:

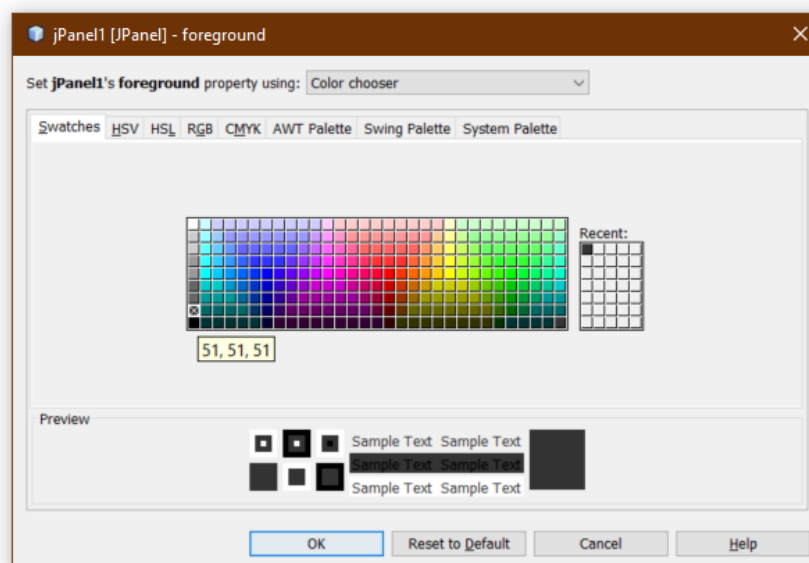
<p>Diagrama de dependencias</p>  <pre> graph TD JFrame[JFrame] --> JPanel1[JPanel1] JPanel1 --> JLabel1[JLabel1] JPanel1 --> JPanel2[JPanel2] JPanel2 --> JPanel3[JPanel3] JPanel3 --> btnAnalizar[JButton] JPanel3 --> JButton1[JButton1] JPanel3 --> JLabel2[JLabel2] JPanel3 --> txtEntrada[JTextField] </pre>	<p>JPanel1 – contiene dentro dos paneles y una etiqueta (JLabel1) con el nombre Compilador Léxico.</p> <p>JPanel2- resalta con un marco el Área de Texto.</p> <p>JPanel3 – contiene dentro de si el campo de texto (txtEntrada), botón Analizar (btnAnalizar), botón Recet (JButton1), y una etiqueta con la instrucción “ingrese el texto que desea analizar”(JLabel2).</p>
--	---

Diseño de la interfaz gráfica:



Color de paneles y botones:

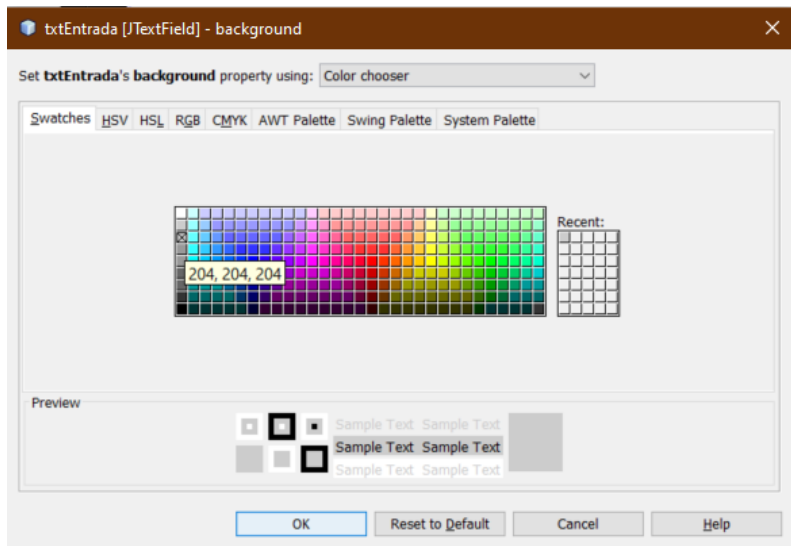
Foreground (color principal): (51,51,51)



Áreas de texto:

Background (color del fondo): (204, 204, 204)

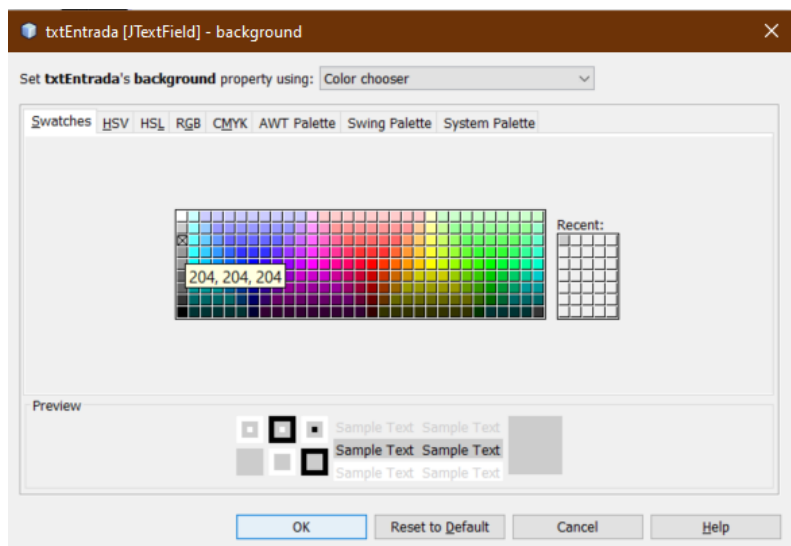
Estilo de letra: Thoman, 18



Texto de etiquetas:

Foreground (color): (204, 204, 204)

Estilo: Sylfaen, 18

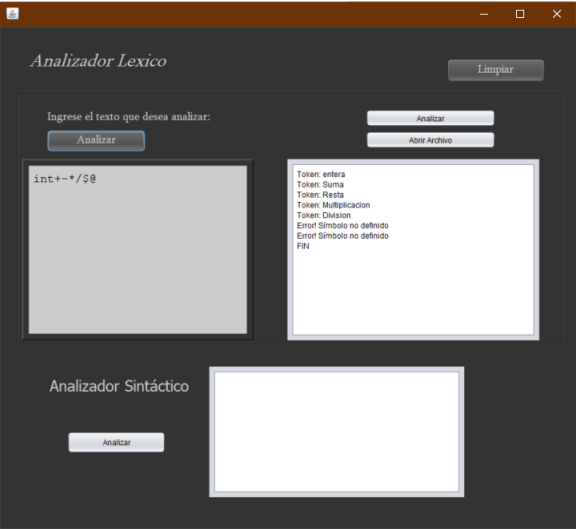


Función de los botones:

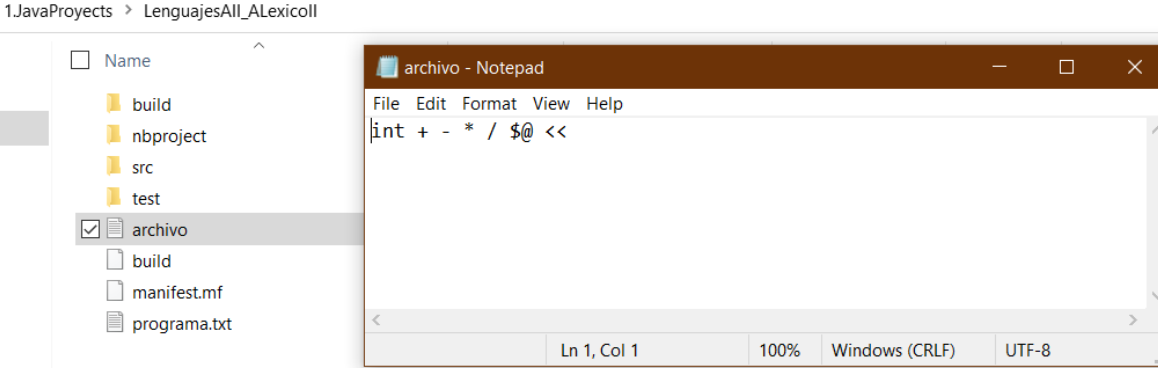
Botón Analizar: Está compuesto por dos try-catch:

<pre> PrintWriter escribir; try { escribir = new PrintWriter(archivo); escribir.print(txtEntrada.getText()); escribir.close(); } catch (FileNotFoundException ex) { Logger.getLogger(frmAnalizador.class.getName()).log(L evel.SEVERE, null, ex); } </pre>	<p>El primer try-catch (PrintWriter) tiene como función crear un archivo de texto dentro del proyecto de nombre “archivo”. El contenido del texto dependerá de lo que el usuario ingrese dentro de él.</p>
<pre> Reader lector; try { lector = new BufferedReader(new FileReader("archivo.txt")); Lexer lexer = new Lexer(lector); String resultado = ""; while (true) { Tokens tokens = lexer.yylex(); if (tokens == null) { resultado += "FIN"; txtSalida.setText(resultado); //System.out.println(resultado); (proyectar en consola de NetBeans) return; } switch (tokens) { case ERROR: resultado += "Error! Símbolo no definido\n"; break; case Identificador: case Numero: case Reservadas: resultado += lexer.lexeme + ": Es un " + tokens + "\n"; break; default: resultado += "Token: " + tokens + "\n"; break; } } } catch (FileNotFoundException ex) { Logger.getLogger(frmAnalizador.class.getName()).log(L evel.SEVERE, null, ex); } catch (IOException ex) { Logger.getLogger(frmAnalizador.class.getName()).log(L evel.SEVERE, null, ex); } </pre>	<p>Éste segundo try-catch (Reader lector), tiene como función reconocer cada uno de los elementos introducidos previamente por el usuario.</p> <p>Estos son introducidos por la función, se almacenaran de un archivo de texto con nombre: “archivo.txt”, y éste leerá su contenido y posteriormente serán impresos en txt.Salida.</p> <h3>Error Léxico:</h3> <p>Al no poder reconocer alguno de los elementos de texto, éste arrojará una alerta con el enunciado: “Error! Símbolo no definido”.</p> <p>Sin embargo, cada uno de los elementos identificados han sido definidos previamente en los archivos “Lexer.flex” y “Tokens.java”, de los elementos introducidos por el usuario serán identificados como “Tokens”.</p>

Ejemplo de funcionamiento de interfaz gráfica:



Como se observa de manera un poco más ilustrado, se introducen elementos dentro de la interfaz gráfica, y una vez presionado el botón de “Analizar”, el programa genera un archivo nuevo con nombre “archivo”, dentro de la carpeta del programa; que visualizando el contenido será el mismo al texto introducido previamente. En seguida éste contenido también será impreso dentro del campo de texto (txtSalida), y al finalizar se describe como FIN.



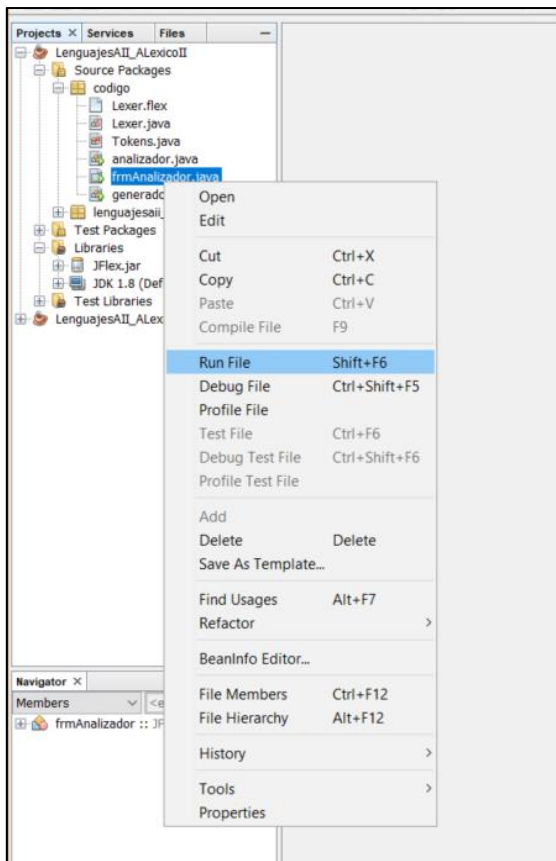
Y este a su vez se genera el archivo de texto dentro de la carpeta del proyecto.

Manual de Usuario

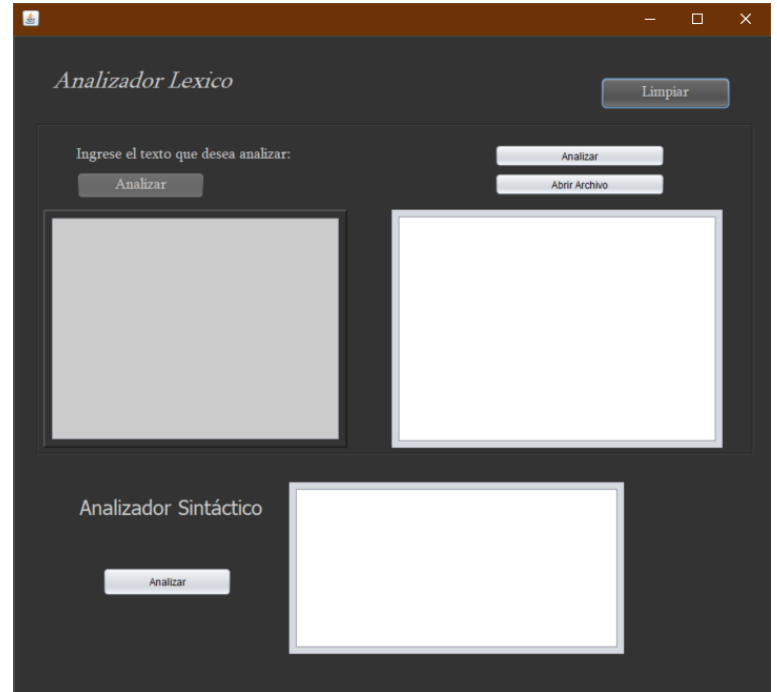
Ejecutar programa:

Usando el IDE java NetBeans 8.2, una vez exportado el proyecto LenguajesAll_ALexicoII, ejecutar la interfaz gráfica: frmAnalizador.java con el segundo botón del ratón, donde si abrirá la interfaz gráfica.

Cómo ejecutar programa:



Interfaz Gráfica



Uso de Programa:

Tome en cuenta los siguientes símbolos y palabras que el programa puede identificar:

Nombres asignados de los operadores, palabras reservadas y delimitadores:

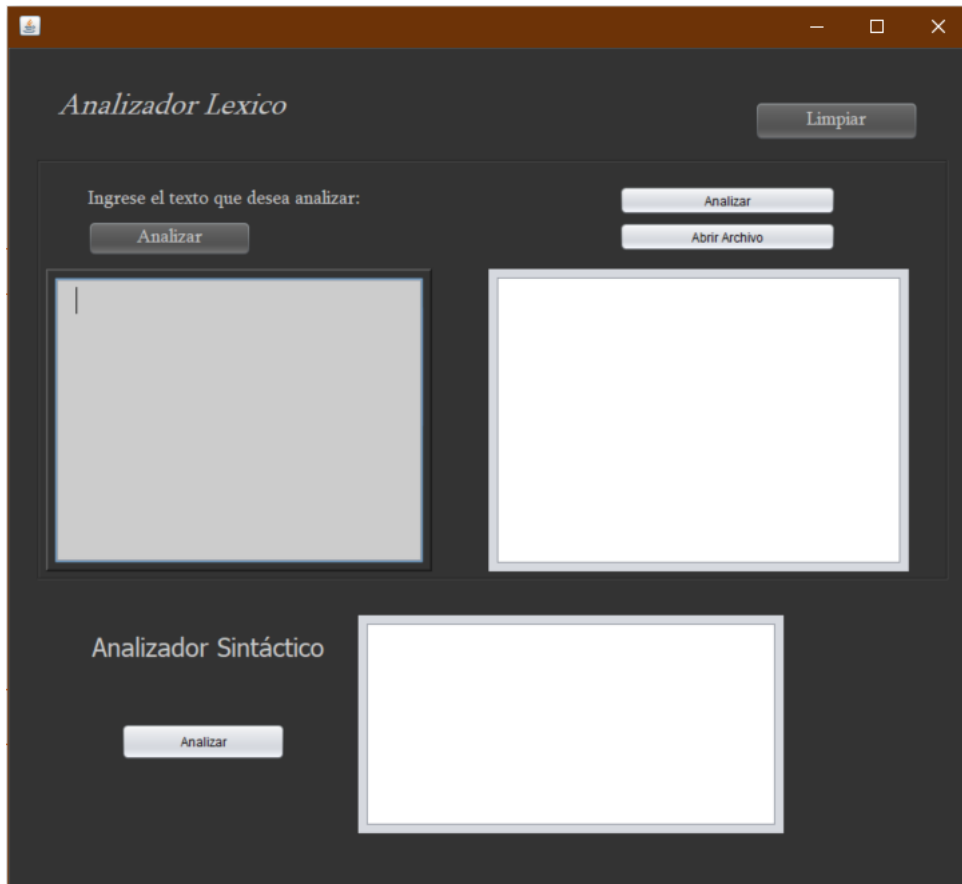
<ul style="list-style-type: none">• if → entonces• else → ademas• while → mientrasque• for → para• int → entera• char → caracter• float → flotante• double → doble• boolean → boleana• "=" → asignaA• "+" → Suma• "-" → Resta• "*" → Multiplicacion• "/" → Division• "==" → igualA• "<" → menorQue• ">" → mayorQue• ">=" → mayorOigualA• "<=" → menorOigualA	<ul style="list-style-type: none">• "<<" → desplazalzquierda• ">>" → desplazaDerecha• "!=" → diferenteA• "&" → tambien• "&&" → comentario• "%" → modula• " " → obien• "++" → incremento• "--" → decremento• " " " → comilladoSimple• " " " " → comilladoDobleA• ":" → dosPuntos• "^" → continuaY• "." → punto• "," → coma• ";" → puntoYcoma• "(" → parentesisA• ")" → parentesisB• "[" → corcheteA• "]" → corcheteB• "{" → llaveA• "}" → llaveB
--	--

Cualquier símbolo que no se encuentre dentro de ésta lista, el programa no podrá identificarlo, y emitirá un mensaje de error de identificación:

Ejemplo: Símbolos no identificados (\$, @).

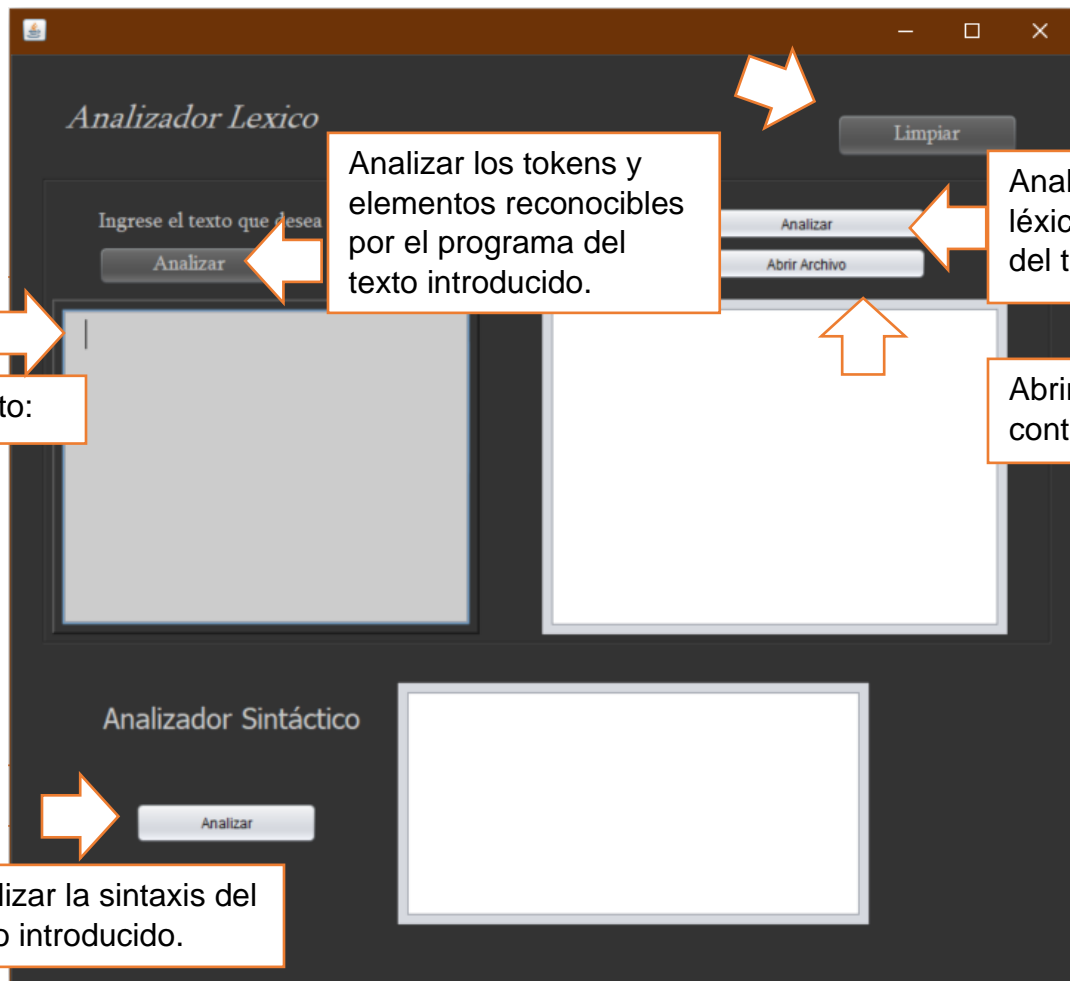
Elementos de la interfaz:

Vista normal de la consola:



Descripción de los elementos de la consola:

Borrar cualquier símbolo
introducido e impreso en
el campo de salida.



Nota:

- Cualquier elemento analizado en ésta consola será generada en un archivo de texto dentro de la carpeta del proyecto con nombre "archivo.txt", el cual se irá actualizando cada que se presione el botón "Analizar".
- Para ingresar nuevos elementos al sistema como palabras y/o símbolos, favor de revisar el manual técnico.