

Instituto Tecnológico de Costa Rica

Escuela de Computación

Compiladores e Intérpretes IC-5701 Grupo 60

## **Proyecto II**

Preparado por:

Emanuel Rodríguez Oviedo c.2022108678

Kendall Guzmán Ramírez c.2019076561

Verano 2023

6 de Enero del 2024

Prof. Allan Gabriel Rodríguez Davila

<b>Manual de uso:</b> .....	<b>3</b>
<b>Pruebas de funcionalidad:</b> .....	<b>3</b>
<b>Descripción del problema:</b> .....	<b>3</b>
<b>Diseño del programa:</b> .....	<b>3</b>
<b>Librerías usadas:</b> .....	<b>3</b>
<b>Análisis de resultados:</b> .....	<b>3</b>
<b>Bitácora:</b> .....	<b>3</b>
<b>Estatus de objetivos:</b> .....	<b>4</b>

## Manual de uso:

Para poder ejecutar la aplicación se debe tener como prerequisite instalado un JDK de java, java y la herramienta de Visual Studio Code con su configuración para ejecutar proyectos de java. Desde un buscador, ingresar a la página oficial del repositorio en GitHub:

<https://github.com/ERodbot/Compiladores---Proyecto-I->

Una vez allí, seleccionar el botón código, donde se encuentra la opción *clonar* y copiar el enlace o los dos cuadrados que se solapan.

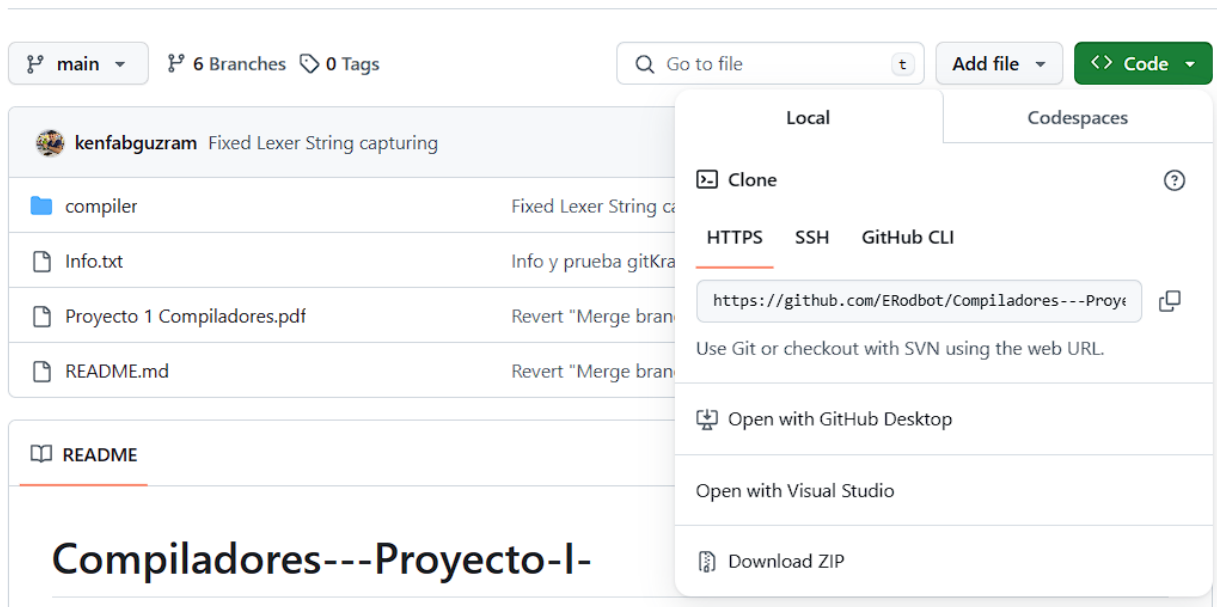
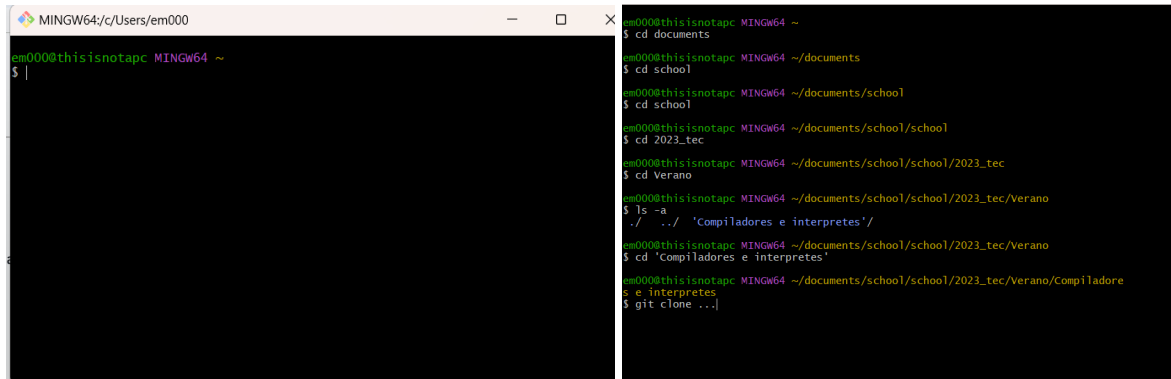


Imagen 1: clonar en git

En la carpeta en la que se desea colocar el proyecto, se utiliza la herramienta git bash para llegar hasta esta y se clona el proyecto usando *git clone* + *el enlace* antes copiado.



The image shows two terminal windows side-by-side. The left window has a title bar 'MINGW64: c:/Users/em000' and shows a prompt 'em000@thisisnotapc MINGW64 ~' with a cursor on a new line. The right window has a title bar 'em000@thisisnotapc MINGW64 ~' and shows a series of commands and directory changes: 'cd documents', 'cd school', 'cd school', 'cd 2023\_tec', 'cd Verano', 'ls -a' (showing './' and '../'), 'cd 'Compiladores e interpretes'', and 'git clone ...'.

```
em000@thisisnotapc MINGW64 ~
$

em000@thisisnotapc MINGW64 ~
$ cd documents
em000@thisisnotapc MINGW64 ~/documents
$ cd school
em000@thisisnotapc MINGW64 ~/documents/school
$ cd school
em000@thisisnotapc MINGW64 ~/documents/school/school
$ cd 2023_tec
em000@thisisnotapc MINGW64 ~/documents/school/school/2023_tec
$ cd Verano
em000@thisisnotapc MINGW64 ~/documents/school/school/2023_tec/Verano
$ ls -a
./  ../  'Compiladores e interpretes'
em000@thisisnotapc MINGW64 ~/documents/school/school/2023_tec/Verano
$ cd 'Compiladores e interpretes'
em000@thisisnotapc MINGW64 ~/documents/school/school/2023_tec/Verano/Compiladore
e interpretes
$ git clone ...
```

Imagen 2: clonar en carpeta

Opcionalmente, se puede descargar el proyecto como un .zip y extraerlo en la carpeta deseada, esto desde la página de Git en download zip, como se ve en la imagen 1.

Luego se debe compilar usando Visual Studio Code, abriendo el proyecto en la pestaña superior izquierda *file + open project*, en este punto se busca la carpeta donde está el proyecto y se abre. Luego, yendo a la carpeta *compiler/src/Main/* y se ejecuta con *click derecho + run* sobre la clase *App.java*.

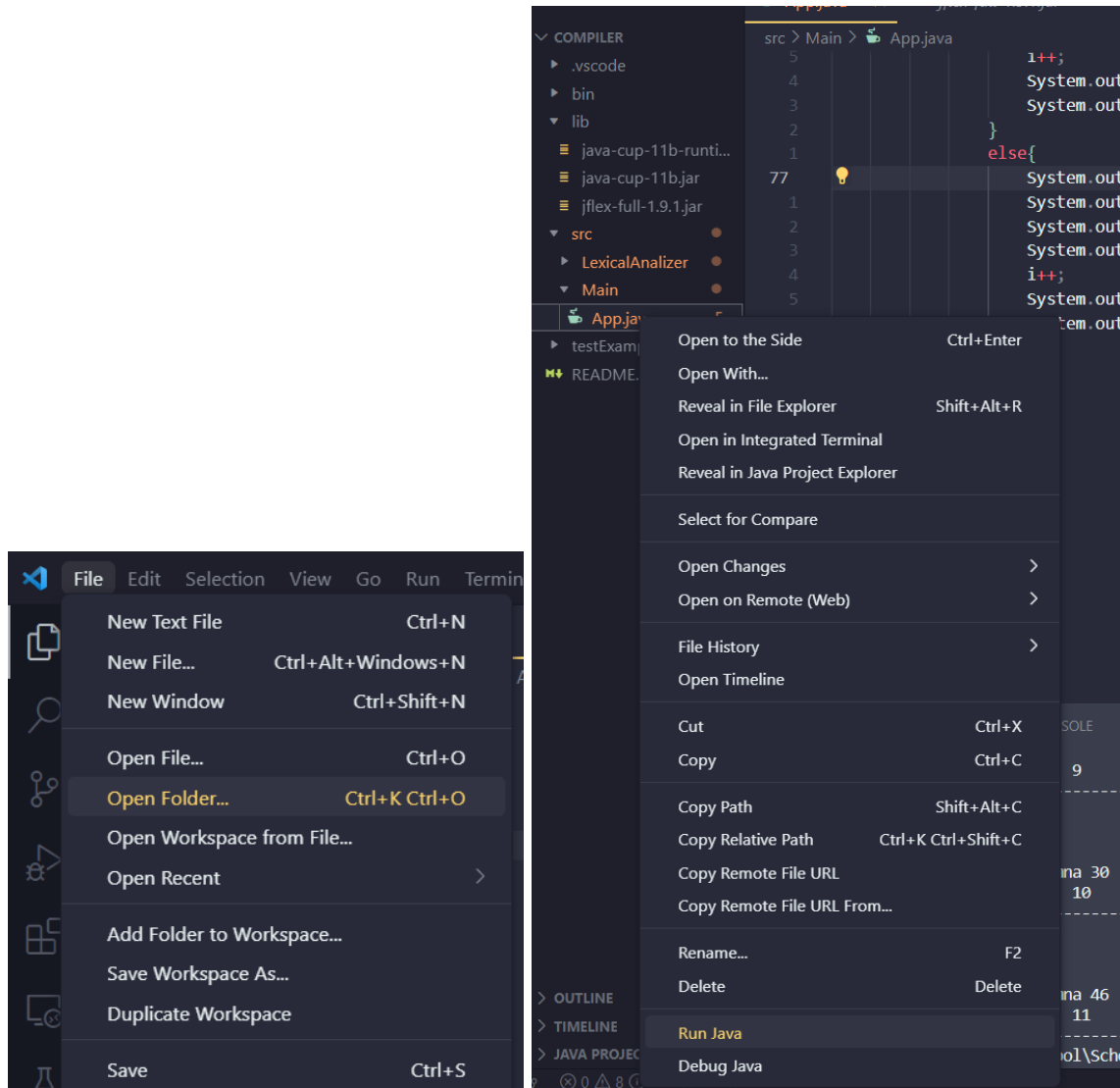


Imagen 4: Compilar y ejecutar proyecto.

## Pruebas de funcionalidad:

Para probar la funcionalidad de nuestro proyecto, se usó un archivo .txt al que se le realizaron varios cambios y errores que demostraban la implementación correcta del manejo de errores a nivel sintáctico. En la **imagen 6** podemos apreciar cómo se logra correctamente identificar el token '>' al principio de la función y se presenta su error. En el segundo ejemplo, representado

en la **imagen 7** se observa un error dentro de la definición de argumentos de una función, mientras que en la **imagen 8** se muestra su respectivo error proporcionado. En este caso se logra recuperar el error a partir del estatuto de definición de la variable hola.

```

1
4  > integer function hola (integer volumen, float nicolasRojo ) { You, 24
1    local float nicolasRojo <= "grey god"|
2    local integer i <= 1|
3    local char holaMundo <= "strong"|
4    local char six <= '6'|
5
6    local char array[] <= {1,2,3,4}|
7    local char array[]|
8    local char array[4] <= {1,2,3,4} |
9    local char array[1]| = 1 |
10

```

Imagen 5: código con error '<' en línea 4

```

Code written to "Sintax.java", and "sym.java".
----- (CUP v0.11b 20160615 (GIT 4ac7450))
Error de sintaxis en la línea 4 y columna 1. Token que generó el error: >
Exception in thread "main" java.lang.Exception: Error de sintaxis no puede ser recuperado. Encontrado en la línea 4 columna 1. Tok
en que generó el error: >
    at LexicalAnalyzer.Sintax.unrecovered_syntax_error(Sintax.java:651)
    at java_cup.runtime.lr_parser.parse(lr_parser.java:731)
    at Main.App.testLexer(App.java:58)
    at Main.App.main(App.java:29)
PS C:\Users\em000\Documents\School\School\2023_TEC\Verano\Compiladores e interpretes\Proyecto-1\Compiladores---Proyecto-I-\compile
r>

```

Imagen 6: mensaje de error en consola.

```

2
1
4  integer function hola (integer volumen, float nicolasRojo, integer ) {
1    local float nicolasRojo <= "grey god"|
2    local integer i <= 1|

```

Imagen 7: error en la definición de argumentos en la función hola, línea 4

```

----- (CUP v0.11b 20160615 (GIT 4ac7450))
Inicio de parseo
Error de sintaxis en la línea 4 y columna 68. Token que generó el error: )

```

Imagen 8: mensaje de error con posible recuperación

Para el caso de la definición de variables, podemos observar en los ejemplos de la **imagen 9** dos errores en su sintaxis, seguidamente en la **imagen 10** se observa la respuesta del programa en forma de error. Estos errores se recuperan en las siguientes líneas inmediatamente posteriores a las dos declaraciones.

```
10
9  ✓ integer function hola (integer volumen, float nicolasRojo ) {
8      float nicolasRojo <= "grey god"|
7      integer i <= 1|
6      char holaMundo <= "strong"|
5      char six <= '6'|
4
3      char array[] <= {1,2,3,4}|
2      char array[]|
1      char array[4] <= {1,2,3,4} |
13     char array[1]| = 1 |      You, 1 second ago • Uncommitted change
1
2
```

Imagen 9: error de declaración de variables en las líneas 12 y 13.

```
Inicio de parseo
Error de sintaxis en la línea 12 y columna 5. Token que generó el error: char
Error de sintaxis en la línea 13 y columna 26. Token que generó el error: =
```

Imagen 10: mensajes de error para error en declaraciones de variables.

Los errores de estatutos por su parte se muestran en la **imagen 11**, con las **imágenes 12 y 13** representando la respuesta del analizador sintáctico.

```
21
22     if nicolasRojo == holaMundo) {
23         do {
24             print("hola mundo ") |
25         }
26         until (1 >= nicolasRojo )|
27     }|      kenfabguzram, 2 weeks ago • Change on
28     elif (true) {
29         for (local <= 1| , i <= 10 , ++i) {
30             read(123)
31         }|
32     }
33     else {
34         return integer hola <= error404 |
35     }|
36 }|
37
```

Imagen 11: Errores sintácticos en estatutos (líneas 22, 30 y 34).

```

Inicio de parseo
Error de sintaxis en la línea 22 y columna 8. Token que generó el error: nicolasRojo
Error de sintaxis en la línea 36 y columna 2. Token que generó el error: }

```

Imagen 12: Respuesta a error en línea 22.

```

Inicio de parseo
Error de sintaxis en la línea 29 y columna 20. Token que generó el error: <=
Error de sintaxis en la línea 34 y columna 16. Token que generó el error: integer

```

Imagen 13: Errores en líneas 29 y 34

Por último, la **imagen 14** nos muestra la impresión presente del árbol sintáctico para el código encontrado en el **anexo 1**. La imagen 15 enseña los archivos generados de este exitoso análisis.

Archivo	Editar	Ver		
Tipo de símbolo	Símbolo	Número de símbolo	Línea	Columna
NOEL	integer	2	3	0
RECORRIDO	function	51	3	8
PERSONA	hola	53	3	17
ABRECUESTO	(	7	3	22
NOEL	integer	2	3	23
PERSONA	volumen	53	3	31
CHIMENEA	,	13	3	38
NICOLAS	float	3	3	40
PERSONA	nicolasRojo	53	3	46
CIERRACUESTO	)	8	3	57
ABREREGALO	{	11	3	59
LOCALCOLOCARREGALO	local	52	4	4
NICOLAS	float	3	4	10
PERSONA	nicolasRojo	53	4	16
ENTREGA	<=	39	4	28
1_SANTA	"grey god"	16	4	40
FINREGALO		54	4	41
LOCALCOLOCARREGALO	local	52	5	4
NOEL	integer	2	5	10
PERSONA	i	53	5	18
ENTREGA	<=	39	5	20
1_NOEL	1	15	5	23
FINREGALO		54	5	24
LOCALCOLOCARREGALO	local	52	6	4
COLACHO	char	5	6	10
PERSONA	holaMundo	53	6	15
ENTREGA	<=	39	6	25
1_SANTA	"strong"	16	6	35
FINREGALO		54	6	36
LOCALCOLOCARREGALO	local	52	7	4
COLACHO	char	5	7	10
PERSONA	six	53	7	15
ENTREGA	<=	39	7	19
1_COLACHO	'6'	17	7	24
FINREGALO		54	7	25
LOCALCOLOCARREGALO	local	52	9	4
COLACHO	char	5	9	10
PERSONA	array	53	9	15
ABREEMPAQUE	[	9	9	20
CIERRAEMPAQUE	]	10	9	21
ENTREGA	<=	39	9	23
ABREREGALO	{	11	9	26
1_NOEL	1	15	9	27
PERSONA	array	53	9	28

Ln 1, Col 3

Valor	Tipo	Lexema
gogo	integer	function
var	integer	variable local

Valor	Tipo	Lexema
hola	integer	function
volumen	integer	argumento de funcion
nicolasRojo	float	argumento de funcion
nicolasRojo	float	variable local
i	integer	variable local
holaMundo	char	variable local
six	char	variable local
array	char	variable local
array	char	variable local
array	char	variable local
array	char	variable local
g	integer	variable local

Imagen 15: archivos generados para el archivo fuente (.txt y markdown).



## Descripción del problema:

El desarrollo del proyecto para la creación del lenguaje "Nochebuena" es un desafío significativo que implica la creación de un compilador capaz de traducir el código escrito en este nuevo lenguaje a instrucciones comprensibles para los procesadores de chips. Uno de los aspectos clave en este proceso es el parseo, que es la etapa en la que el compilador analiza los tokens detectados por el analizador léxico en el código fuente para convertirlos en una representación interna que pueda ser utilizada en las etapas posteriores del proceso de compilación.

La importancia del proceso de parseo radica en que es esencial para garantizar que el código escrito en "Nochebuena" sea gramaticalmente correcto y cumpla con la sintaxis definida para el lenguaje. Un parser eficiente y preciso permite detectar errores en el código fuente y generar mensajes de error claros para los desarrolladores, facilitando la corrección de posibles bugs.

La problemática que enfrenta un compilador durante el parseo incluye la gestión de ambigüedades en la gramática del lenguaje, la identificación correcta de tokens y la resolución de conflictos sintácticos. Además, en el contexto de un lenguaje imperativo para configuración de chips, la definición precisa de la sintaxis es crucial para evitar errores que podrían tener consecuencias graves en el funcionamiento de los chips.

Es por eso que nuestros objetivos para la segunda fase de nuestro proyecto son:

**Objetivo Principal:** desarrollar un programa que lea un código fuente y utilizando la herramienta Cup realice el parseo de acuerdo con la gramática del lenguaje denominado "Nochebuena".

### **Objetivos secundarios:**

1. Crear una serie de producciones que sirvan como base sintáctica del nuevo lenguaje de programación.
2. Identificar adecuadamente los errores que puede presentar la gramática y manejarlos para evitar una caída del proceso de parseo.
3. Preservar información relevante de los tokens detectados en el código fuente por medio de la tabla de símbolos.

## Diseño del programa:

Para lograr cumplir el propósito de nuestro programa se utilizó el lenguaje de programación Java. En efecto, este posee herramientas como las librerías CUP y JFLEX, que sirven como generadores léxicos y analizadores sintácticos y aportan una gran facilidad de uso en su objetivo.

El programa consta de 2 principales paquetes:

- Main
- LexicalAnalyzer

El paquete Lexical Analyzer, como su nombre lo indica, se encarga del proceso de creación y definición del analizador léxico. Para ello se emplea un archivo LexerCup.flex, que posee la definición de los lexemas del lenguaje navideño, por medio del uso de expresiones regulares y definiciones regulares. Al mismo tiempo, utiliza su capacidad para definir comportamiento en lenguaje java para retornar los símbolos que representan a cada lexema por medio de una clase Symbol. Estos lexemas constituyen los siguientes elementos base del lenguaje:

- Operadores aritméticos binarios
- Operadores aritméticos unarios
- Operadores relacionales
- Operadores lógicos
- identificador
- tipos
- literales
- paréntesis
- paréntesis cuadrado
- llaves
- lexemas de estructuras de control (if, elif, else, for, do, until, return, break)
- print, read
- lexema de fin de expresión
- lexema de asignar
- lexema separador(coma)

```

//Parenthesis
LeftParenthesis= \(
RightParenthesis= \)
LeftBracket= \[
RightBracket= \]
LeftKey= \{
RightKey= \}h

//separator
Separator= ,

//literaldecimalinteger
LiteralDecimalInteger= (0|-?[1-9][0-9]*)

//literaldecimalfloat
LiteralDecimalFloat= ({LiteralDecimalInteger})\.[0-9]*

//unarian operators
Decrement = --
Increment = \+\+

//arithmetic binary operators
Sum= \+
Subtraction= \-
Multiplication=\*

```

```

//literaltrue
<YYINITIAL> "true"          {return symbol(sym.l_t_CLAUS);}

//literalfalse
<YYINITIAL> "false"         {return symbol(sym.l_f_CLAUS);}

<YYINITIAL> {
    \"                      { string.setLength(0); yybegin(STRING); }

    {SingleLineComment}     { /*skip*/ }

    {MultipleLineComment}   { /*skip*/ }

    /*decimal integer literal*/
    {LiteralDecimalInteger}  {return symbol(sym.l_NOEL);}

    /*decimal float literal*/

```

Imagen 16: archivo LexerCup.flex

Luego, por medio del uso de la librería CUP y el archivo llamado “Syntax.CUP” se logra definir código de constructor personalizado para el parser, que le otorga un DefaultSymbolFactory, para la construcción de símbolos (tokens).

Además aporta una inicialización del analizador y define cómo se obtienen los tokens del analizador léxico.

Igualmente, define los nombres de los tokens para cada símbolo no terminal dentro del lenguaje de programación.

```
parser code {:  
    LexerCup lex;  
  
    @SuppressWarnings("deprecation")  
    public void parser(LexerCup lex){  
        this.lex=lex;  
        this.symbolFactory= new DefaultsymbolFactory();  
    }  
:}  
  
init with { : :};  
  
scan with { : return lex.next_token(); :};  
  
// Terminals  
  
// Data Types  
terminal NOEL, NICOLAS, SANTA, COLACHO, CLAUS;  
  
//Parenthesis  
terminal ABRECUESTO, CIERRACUESTO, ABREEMPAQUE, CIERRAEMPAQUE, ABREREGALO, CIERRAREGALO;  
  
//Separator  
terminal CHIMENEA;
```

Imagen 17: archivo .Cup

A continuación, el archivo App.java realiza una llamada a su método main. Inmediatamente se llama a una función que genera los archivos: Syntax.java, Lexer.java y sym.java. Estos sirven para definir la sintaxis del lenguaje, nuestro analizador léxico y el archivo que referencia los tokens y su id.

```

Run | Debug
public static void main(String[] args) throws Exception {

    String pathLexCup = System.getProperty(key:"user.dir") + "/src/LexicalAnalyzer/LexerCup.flex";
    String[] paths = { "-parser", "Sintax", (System.getProperty(key:"user.dir") + "/src/LexicalAnalyzer/Sintax.cup") };
    generar(pathLexCup, paths);
    testLexex(System.getProperty(key:"user.dir") + "/testExamples/example.txt");
}

public static void generar(String pathLexCup, String[] paths) throws IOException, Exception {
    String[] paths = { pathLexCup };
    jflex.Main.generate(paths);
    java_cup.Main.main(paths);

    Path pathSym = Paths.get((System.getProperty(key:"user.dir") + "/src/LexicalAnalyzer/sym.java"));
    if (Files.exists(pathSym)) {
        Files.delete(pathSym);
    }

    Files.move(Paths.get((System.getProperty(key:"user.dir") + "/sym.java")),
        Paths.get((System.getProperty(key:"user.dir") + "/src/LexicalAnalyzer/sym.java")));

    Path pathSin = Paths.get((System.getProperty(key:"user.dir") + "/src/LexicalAnalyzer/Sintax.java"));
    if (Files.exists(pathSin)) {
        Files.delete(pathSin);
    }
    Files.move(Paths.get((System.getProperty(key:"user.dir") + "/Sintax.java")),
        Paths.get((System.getProperty(key:"user.dir") + "/src/LexicalAnalyzer/Sintax.java")));
}

```

Imagen 18: función main y generar

Tras la lectura del *path* a leer, definido dentro del método main al llamar a generar. Se comprueba la validez del path y se realiza la lectura, mostrando los resultados en la misma consola.

```

public static void testLexex(String scannerPath) throws IOException {
    Reader reader = new BufferedReader(new FileReader(scannerPath));
    reader.read();
    LexerCup lexer = new LexerCup(reader);
    int i = 1;
    Symbol token;

    // Print table header
    System.out.printf(format:"%-20s %-20s %-20s %-20s %-20s %-20s %n", ...args:"Tipo de símbolo", "Símbolo", "Número de símbolo", "Línea", "Columna");
    System.out.println(x:"-----");

    while (true) {
        token = lexer.next_token();

        if (token.sym != 0) {
            String symbolType = sym.terminalNames[token.sym];
            String symbol = (token.sym == 24) ? token.value.toString() : lexer.yytext();
            int symbolNumber = token.sym;
            int line = token.left;
            int column = token.right;

            You, 10 hours ago • new tokens, change in main interface, follow-up t...
            // Print table row
            System.out.printf(format:"%-20s %-20s %-20s %-20d %-20d %n", symbolType, symbol.toString(), symbolNumber, line, column);
            i++;
        } else {
            break;
        }
    }

    System.out.println("Cantidad de lexemas encontrados: " + i);
    System.out.println(x:"-----");
}

```

Imagen 19: tester para lexemas con menú de información

## Librerías usadas:

Para el desarrollo del programa se decidieron usar las librerías: . Su función específica se detalla en la siguiente tabla:

Libreria	Funcion
java_cup	Utilizado como base de generador de tokens para los lexemas, este además sirve como interfaz para el analizador léxico.
Java Standard I/O	Utilizada para la lectura de archivos, creación de buffers de lectura y utilidades de manejo de archivos.
Java New I/O	Utilizada para la manipulación de archivos dentro del programa main, que reconoce por un parámetro de entrada los problemas encontrados en un archivo, así como los errores.
Java Jflex	Utilizada para generar el analizador léxico, aporta la base de la definición de lexemas presentes en el lenguaje de programación por medio de expresiones regulares y definiciones regulares, retorna los símbolos definidos para los mismos lexemas en el archivo .flex.

## Análisis de resultados:

El criterio de evaluación para nuestro proyecto se basará en la implementación de los objetivos definidos en la sección [Descripción del problema](#): por medio de la utilización de etiquetas que ayudarán a describir el estado de cada uno de ellos. Además, se aportarán comentarios en caso de que no se haya logrado completar un objetivo o que se considere necesario.

Las etiquetas a utilizar son:

1. No realizado
2. Parcialmente realizado
3. No probado
4. No funcional
5. Parcialmente funcional
6. Funcional.

**Objetivo 1:** Crear una serie de producciones que sirvan como base sintáctica del nuevo lenguaje de programación.

Para este objetivo se tomará los requerimientos de la gramática BNF presentados a continuación:

- a. Permitir la creación de funciones, y dentro de ellas, estructuras de control, bloques de código ({} ) y sentencias de código.  
Estado: Funcional
- b. Manejar los tipos de variables enteras, flotantes, booleanas, caracteres, cadenas de caracteres (string) y arreglo estático.  
Estado: Funcional
- c. Se permite crear arreglos de tipo entero o char. Además, se permite obtener y modificar sus elementos, y ser utilizados en expresiones.  
Estado: Funcional
- d. Permitir sentencias para creación de variables, creación y asignación de expresiones y asignación de expresiones a variables, y algunos casos, sólo expresiones sin asignación. La asignación se hará por medio de “<=”.  
Estado: Funcional
- e. Las expresiones permiten combinar literales, variables y/o funciones, de los tipos reconocidos en la gramática.  
Estado: Funcional
- f. Debe permitir operadores y operandos, respetando precedencia (usual matemática)

y permitiendo el uso de paréntesis.

Estado: Funcional

g. Permitir expresiones aritméticas binarias de suma (+), resta (-), división (/) –entera o decimal según el tipo--, multiplicación (\*), módulo (~) y potencia (\*\*). Para enteros o flotantes.

Estado: Funcional

h. Permitir expresiones aritméticas unarias de negativo (-), ++, --, antes del operando (preorden); esto para enteros, el negativo adicionalmente se puede aplicar a flotantes. El negativo a literales y el ++ y -- a variables.

Estado: Funcional

i. Permitir expresiones relacionales (sobre enteros y flotantes) de menor, menor o igual, mayor, mayor o igual, igual y diferente. Los operadores igual y diferente permiten adicionalmente tipo booleano. Sólo permiten un operando y 2 operadores.

Estado: Funcional

j. Permitir expresiones lógicas de conjunción (^), disyunción (#) y negación (esta debe ser de tipo caracter (!)).

Estado: Funcional

k. Debe permitir sentencias de código para las diferentes expresiones mencionadas anteriormente y su combinación, el delimitador de final de expresión será el carácter pipe (|). Además, dichas expresiones pueden usarse en las condicionales y bloques de las siguientes estructuras de control.

Estado: Funcional

l. Debe permitir el uso de tipos y la combinación de expresiones aritméticas (binarias y unarias), relacionales y lógicas, según las reglas gramaticales, aritméticas, relacionales y lógicas del Paradigma Imperativo, por ejemplo, tomando como referencia el lenguaje C.

Estado: Funcional

m. Debe permitir las estructuras de control if-[elif]-[else], do-until y for, además, permitir return y break. Las expresiones de las condiciones deberán ser valores booleanos combinando expresiones aritméticas, lógicas y relacionales.

Estado: Funcional

n. Debe permitir las funciones de leer (enteros y flotantes) y escribir en la salida estándar (cadena carácter, enteros y flotantes), se pueden escribir literales o variables, se lee a identificadores.

Estado: Funcional

o. Debe permitir la creación y utilización de funciones, definición clásica, estos deben retornar valores (entero, flotantes, char o booleanos) y recibir parámetros (con tipo).

Estado: Funcional

p. Debe existir un único procedimiento inicial main, por medio de la cual se inicia la ejecución de los programas.

Estado: Funcional

q. Además, debe permitir comentarios de una línea (@) o múltiples líneas (/\_ \_/).

Estado: Funcional



**Objetivo 2:** Identificar adecuadamente los errores que puede presentar la gramática y manejarlos para evitar una caída del proceso de parseo.

- Errores asociados a la declaración de funciones:  
Estado: Funcional
- Errores asociados a los estatutos (líneas de código):  
Estado: Funcional
- Errores asociados a la definición de argumentos en funciones:  
Estado: Funcional
- Errores asociados a las declaraciones de variables:  
Estado: Funcional

**Objetivo 3:** Preservar información relevante de los tokens detectados en el código fuente por medio de la tabla de símbolos.

- Hace la lectura completa de un archivo fuente:  
Estado: Funcional
- Identifica la lista completa de tokens presentados así como sus errores.  
Estado: Funcional
- Identifica las producciones en el análisis sintáctico por medio de la creación de un árbol sintáctico.  
Estado: Funcional
- Identifica los errores de sintaxis en el código fuente.  
Estado: Funcional
- Imprime los errores encontrados.  
Estado: Funcional

- Escribe los tokens encontrados en un archivo.  
Estado: Funcional
- Escribe el árbol sintáctico generado a un archivo.  
Estado: Funcional

## Bitácora:

Se proporciona automáticamente por medio de la herramienta de github:

[Commits · ERodbot/Compiladores---Proyecto-I- \(github.com\)](https://github.com/ERodbot/Compiladores---Proyecto-I-)

## Anexo 1

Código usado para el test:

```
integer function hola (integer volumen, float nicolasRojo) {
  local float nicolasRojo <= "grey god"|
  local integer i <= 1|
  local char holaMundo <= "strong"|
  local char six <= '6'|

  local char array[] <= {1,2,3,4}|
  local char array[]|
  local char array[4] <= {1,2,3,4} |
  local char array[1]|

  local integer g <= (5 + ++er - --nicolasRojo * (5)~1 >= 60) # true |

  if (12 > nicolasRojo) {
    return|
  }

  if (nicolasRojo == holaMundo) {
    do {
      print("hola mundo ") |
    }
    until (1 >= nicolasRojo )|
  }
}
```

```
elif (true) {  
    for (local integer i <= 1|, i <== 10 , ++i) {  
        read(123)|  
    }  
}  
else {  
    return error404 |  
}  
}|
```

```
integer function gogo () {  
    local integer var <= hola(mal, vola)||  
    return var|  
    return hola(mal, vola)||  
}|
```