



Universidad Nacional Autónoma de
México



FACULTAD DE INGENIERÍA
Compiladores (Cve: 0434)
Grupo: 04 - Semestre: 2025-2

Proyecto 01:
Analizador Léxico (Scanner)

FECHA DE ENTREGA: 18/03/2025 23:45 HRS

Profesor:

M.C. Laura Sandoval Montaña

Integrantes:

Arellanes Conde Esteban

Méndez Galicia Axel Gael

Índice

Introducción	3
Objetivo	4
Requerimientos del programa	5
Desarrollo.....	7
Actividad 1. Expresiones Regulares	7
Actividad 2. AFD de las expresiones regulares	8
Actividad 3. AFD individuales en un solo AFD	10
Análisis	15
Diseño	26
Funcionamiento	31
Pruebas	43
Conclusiones.....	48
Bibliografía.....	49

Introducción:

Un analizador léxico es un programa que lee un código fuente y lo transforma en una secuencia de tokens o componentes léxicos. Estos tokens son la entrada para el analizador sintáctico, que es la siguiente etapa del proceso de traducción.

El analizador léxico tiene las siguientes características:

- Es la primera fase del compilador.
- Se lee en secuencias de caracteres de izquierda a derecha del programa fuente.
- Agrupa las secuencias de caracteres en “palabras” / cadenas para identificar si esta unidad se trata de un componente léxico del lenguaje a compilar.

Un componente léxico es una secuencia de caracteres en el programa fuente, que coincide con un patrón para un token y que el analizador léxico identifica como una instancia de ese token.

En la mayoría de los lenguajes de programación pueden ser las siguientes:

- Palabras reservadas
- Operadores (aritméticos, relacionales, etc.)
- Identificadores
- Constantes
- Cadenas literales
- Signos de puntuación o símbolos especiales (paréntesis, punto y coma,...)

Por otra parte, cada uno de los componentes léxicos puede representarse por un patrón , (reglas que describe un conjunto de lexemas), como por ejemplo; un patrón para identificar símbolos especiales.

Un token es una pareja compuesta por un nombre de clase, representado por un número entero asignado a cada componente, y un atributo que apunta a la tabla de símbolos donde se almacenan dichos símbolos o, en su defecto, a la tabla que contiene la cadena utilizada para reconocer el componente léxico.

El programa es un analizador léxico desarrollado con Flex (Lex), cuyo propósito es leer un archivo de entrada, identificar y clasificar distintos tipos de tokens, y registrar información relevante en tablas de símbolos y literales. Además, gestiona los errores léxicos, registrándolos y reportándolos sin interrumpir el análisis.

Objetivo:

Elaborar un analizador léxico en lex/flex que reconozca los componentes léxicos pertenecientes a las clases abajo descritas y que fueron definidas en clase.

Clase	Descripción
0	Palabras reservadas (ver tabla)
1	Símbolos especiales (ver tabla)
2	Identificadores. Deben empezar por @, le deben seguir uno o más de los siguientes caracteres de A-Z y a-z, vocales mayúsculas y minúsculas pueden ir acentuadas, dígitos del 0 al 9, acepta ñ y Ñ; y debe terminar con un guion bajo _.
3	Operadores aritméticos (ver tabla)
4	Operadores relacionales (ver tabla)
5	Operadores de asignación (ver tabla)
6	Constantes cadena. Inician y terminan con -- . Pueden contener cualquier carácter.
7	Constantes enteras. No pueden iniciar con = si son 2 o más dígitos. Pueden tener signo + o - . Puede tener el sufijo p (pequeño) o g(grande). Ejemplos: 0 -24 +34p 18g
8	Constantes reales: La parte fraccionaria se escribe después de ' (apóstrofo). Ejemplos: 12'2 242'87 '604 Para escribir un valor real sin fracciones se pueden usar los sufijos r o R. Ejemplos: 23r 649R

Valor	Palabra reservada	Valor	Palabra reservada
0	Bool	8	Haz
1	Cade	9	Mientras
2	Continuar	10	Nulo
3	Devo	11	Para
4	Ent	12	Parar
5	Fals	13	Si
6	Flota	14	Sino
7	Global	15	Ver

Valor	Op. aritmético
0	sum
1	rest
2	mult
3	div
4	mod
5	inc
6	dec
7	exp
8	dive

Valor	Op. asignación
0	->
1	+->
2	-->
3	*->
4	/->
5	%->
6	>>
7	<<
8	^->
9	&->

Valor	Símbolo Especial	Representa
0	<	(
1	>)
2	<<	[
3	>>]
4	#	{
5	#!	}
6	*	;
7		,
8	°	.

Valor	Op. relacional	Representa
0	h	Mayor que
1	m	Menor que
2	e	Igual que
3	c	Diferente a
4	he	Mayor o igual
5	me	Menor o igual

Requerimientos del programa:

- Las clases de los componentes léxicos válidos para el analizador léxico son las presentadas en el objetivo.
- El número de clase es inamovible.
- • El analizador léxico tendrá como entrada un archivo con el programa fuente, el cual se indicará desde la línea de comandos al momento de mandar a ejecutar el analizador léxico.
- • Como delimitador de un componente léxico será uno o varios espacios, tabuladores o saltos de línea, así como el inicio de otro componente léxico. Considerar el orden en las acciones en el programa lex/flex
- • Los tokens se representarán en una estructura con dos campos: campo1: la clase campo2: el valor (de acuerdo con las siguientes tablas e indicaciones)
- El valor para el token de cada identificador es la posición dentro de la tabla de símbolos. Para las palabras reservadas, los símbolos especiales, los operadores aritméticos, los operadores relacionales y los operadores de asignación, será la posición en su correspondiente tabla (catálogo).
- Las constantes cadenas se incluirán en una tabla de literales, al igual que las constantes flotantes, por lo que el valor de su token será la posición dentro de su correspondiente tabla. Cada vez que el Analizador Léxico encuentre una cadena o una constante flotantes NO revisará si ya se encuentra en su tabla de literales, simplemente la insertará. Las constantes enteras como valor será el valor numérico de la constante entera.
- Cuando detecte un error léxico, deberá seguir el reconocimiento a partir del siguiente símbolo.
- • El analizador deberá crear la Tabla de Símbolos para almacenar a los identificadores. Esta tabla manejará los campos: posición, nombre del identificador y tipo (este último será de tipo entero y podrán ponerle como valor inicial -1). Se indicará en el documento a entregar, el tipo de estructura de datos empleada, así como el método de búsqueda a utilizar.
- • Las tablas de literales para las cadenas y para las constantes flotantes, deberán tener como estructura, dos campos: la posición y el dato (cadena).
- • El Analizador Léxico deberá reconocer los comentarios y descartarlos. Los comentarios se identifican porque estarán entre [] (corchetes).
- • Al término del análisis léxico deberá mostrar la tabla de símbolos, las tablas de literales, así como la secuencia de tokens. También podrán almacenarse en archivos para su mejor revisión.
- • Los errores que vaya encontrando el analizador léxico, los podrá ir mostrando en pantalla o escribirlos en un archivo. Deberá recuperarse de los errores encontrados

para continuar con el reconocimiento de todos los componentes léxicos del archivo de entrada.

- El programa deberá estar comentado, con una descripción breve de lo que hace (puede ser el objetivo indicado en este documento), el nombre de quién(es) elaboró(arón) el programa y fecha de elaboración, así como lo que hace cada función. Se deberá cuidar mucho la sangría que denota la dependencia de instrucciones.

Desarrollo:

Actividad 1. Expresiones Regulares

```
99  /* Expresiones regulares para los tokens */
100 DIGITO    [0-9]
101 PALABRA_RESERVADA (Bool|Cade|Continuar|Devo|Ent|Fals|Flota|Global|Haz|Mientras|Nulo|Para|Parar|Si|Sino|Ver)
102 SIMBOLOS_ESPECIALES (<|>|<<|>>|#|#!|\\*|\\|'|")
103 IDENT      @[A-Za-zÁÉÍÓÚÑáéíóúñ][A-Za-zÁÉÍÓÚÑáéíóúñ0-9]*_
104 OP_ARITHMETIC (sum|rest|mult|div|mod|inc|dec|exp|dive)
105 OP_RELACIONAL (h|m|e|c|he|me)
106 OP_ASIGNACION ("<->"|"\\+<->"|"-<->"|"\\*<->"|"/<->"|"%->"|">>"|"<<"|"\\^<->"|"&<->")
107 ENTERO      [-+]?[1-9]{DIGITO}*{DIGITO}?[p|g]?
108 REAL        [-+]?{DIGITO}+(' {DIGITO}+)?|{DIGITO}+[rR]
109 CADENA      "- -" . *? "- -"
110 COMMENT     \[[^\\]]*\\
111 WS          [ \\t\\n]+
112
```


Actividad 2. AFD de las expresiones regulares

1. AFD para Identificadores (@([A-Za-zÁÉÍÓÚáéíóúñÑ][A-Za-z0-9_ÁÉÍÓÚáéíóúñÑ]*)_)

Estados y transiciones

Estado	Entrada	Siguiente Estado
q0	@	q1
q1	[A-Za-zÁÉÍÓÚáéíóúñÑ]	q2
q2	[A-Za-z0-9_ÁÉÍÓÚáéíóúñÑ]	q2
q2	_	q3 (aceptación)

2. AFD para Constantes Enteras (-?\d+[pg]?)

Estados y transiciones

Estado	Entrada	Siguiente Estado
q0	-	q1
q0, q1	\d	q2
q2	\d	q2
q2	p o g	q3 (aceptación)

3. AFD para Constantes Reales (-?\d+\.'?\d+|\d+[rR])

Estados y transiciones

Estado	Entrada	Siguiente Estado
q0	-	q1

q0, q1	\d	q2
q2	\d	q2
q2	'	q3
q3	\d	q4 (aceptación)
q2	r o R	q5 (aceptación)

4. AFD para Constantes de Cadena ("--[^ "]*--")

Estados y transiciones

Estado	Entrada	Siguiente Estado
q0	" -	q1
q1	[^ "]	q1
q1	- "	q2 (aceptación)

5. AFD para Operadores Aritméticos (\+ | - | * | \/ | % | \+\+ | -- | \^ | div)

Estados y transiciones

Estado	Entrada	Siguiente Estado
q0	+	q1
q0	-	q2
q0	*	q3 (aceptación)
q0	/	q4 (aceptación)
q0	%	q5 (aceptación)

q1	+	q6 (aceptación)
q2	-	q7 (aceptación)
q0	d	q8
q8	i	q9
q9	v	q10 (aceptación)

6. AFD para Operadores Relacionales (h | m | e | c | he | me)

Estados y transiciones

Estado	Entrada	Siguiente Estado
q0	h	q1 (aceptación)
q0	m	q2 (aceptación)
q0	e	q3 (aceptación)
q0	c	q4 (aceptación)
q1	e	q5 (aceptación)
q2	e	q6 (aceptación)

7. AFD para Operadores de Asignación (-> | \+ -> | --> | * -> | \/ -> | %-> | >> | << | ^ -> | &->)

Estados y transiciones

Estado	Entrada	Siguiente Estado
q0	-	q1

q1 > q2 (aceptación)

q0 + q3

q3 - q4

q4 > q5 (aceptación)

q0 * q6

q6 - q7

q7 > q8 (aceptación)

q0 / q9

q9 - q10

q10 > q11 (aceptación)

q0 % q12

q12 - q13

q13 > q14 (aceptación)

q0 > q15

q15 > q16 (aceptación)

q0 < q17

q17 < q18 (aceptación)

q0 ^ q19

q19 - q20

q20 > q21 (aceptación)

q0 & q22

q22 - q23

q23 > q24 (aceptación)

Actividad 3. AFD individuales en un solo AFD

8. Unificación de los AFD en uno solo

Para combinar todos los AFD en un único AFD, creamos un **estado inicial único** que discrimine la entrada inicial para redirigirla al AFD correspondiente. Se utilizará una **función de transición** que evalúe el primer carácter y determine cuál de los AFD internos debe activarse.

Esto se logra extendiendo la tabla de transición con un nuevo **estado q0** que desambigua entre:

- @ → AFD de identificadores
- " → AFD de cadenas
- -, +, *, /, %, ^, & → AFD de operadores
- [0-9] → AFD de números
- Letras específicas (h, m, e, c, d, i, v) → AFD de operadores relacionales y div
- Cualquier otro símbolo → AFD de símbolos especiales o error.

Análisis:

Para llevar a cabo el análisis de este programa, se han definido tres estructuras que servirán para la creación de tablas específicas. Cada una de estas tablas tendrá una función particular: una se utilizará para almacenar los símbolos, otra para las literales numéricas y la tercera para las literales de cadena. A medida que el código fuente del archivo de texto sea procesado, se registrarán en estas tablas los caracteres y valores encontrados, permitiendo así una correcta organización y gestión de la información extraída del análisis léxico.

Repartición de Actividades:

Axel Gael Méndez Galicia: Se centró en realizar los ADF, así como realizar las expresiones regulares que serían utilizadas para nuestros analizadores léxico junto con sus respectivos autómatas.

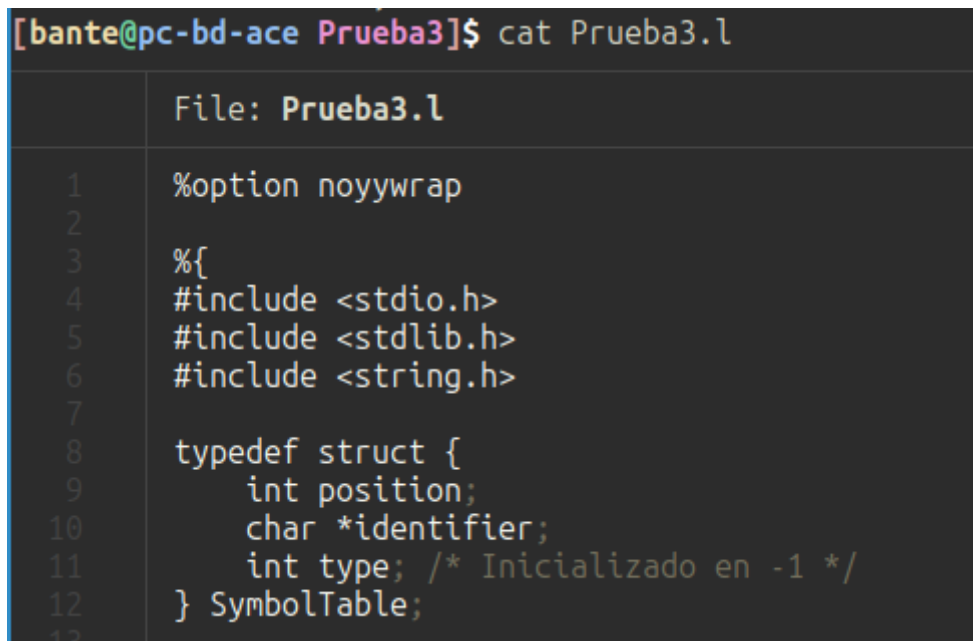
Arellanes Conde Esteban: Era el encargado de realizar el código. Depurar, Ejecutar y Compilar.

Tabla de Símbolos

Esta estructura se encargará de almacenar los identificadores encontrados en el código fuente.

Código Fuente: Archivo **Prueba3.l**

- **Campos:**
 - Posición
 - Nombre del identificador
 - Tipo de dato asociado



```
[bante@pc-bd-ace Prueba3]$ cat Prueba3.l
File: Prueba3.l
1  %option noyywrap
2
3  %{
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7
8  typedef struct {
9      int position;
10     char *identifier;
11     int type; /* Inicializado en -1 */
12 } SymbolTable;
13
```

- **Funcionalidad:**
 - La función de inserción busca el identificador en la tabla. Si no está registrado, lo agrega con su información.
 - Se utilizará una **tabla hash con manejo de colisiones por encadenamiento** para optimizar la búsqueda e inserción.

```

13
14     int addSymbol(char *name);
15     int addNumericLiteral(char *literal);
16     int addStringLiteral(char *string);
17     void printTables();
18     void reportError(const char *msg);
19
20     typedef struct {
21         int position;
22         char *literal;
23     } LiteralTable;
24
25     typedef struct {
26         int position;
27         char *string;
28     } StringLiteralTable;
29

```

```

36
37     /* Función para agregar un símbolo a la tabla */
38     int addSymbol(char *name) {
39         for (int i = 0; i < symbolCount; i++) {
40             if (strcmp(symbols[i].identifier, name) == 0) {
41                 return symbols[i].position;
42             }
43         }
44         symbols[symbolCount].position = symbolCount;
45         symbols[symbolCount].identifier = strdup(name);
46         symbols[symbolCount].type = -1;
47         return symbolCount++;
48     }
49

```

Tabla de Literales Numéricas

Esta tabla almacenará los valores constantes de tipo numérico que aparezcan en el código fuente.

- **Campos:**
 - Posición
 - Valor numérico (entero o flotante)
- **Funcionalidad:**
 - Se registran todas las constantes numéricas encontradas en el código.
 - Se usará **una lista ordenada** para facilitar la inserción rápida y evitar duplicados.


```

50  /* Función para agregar un literal numérico a la tabla */
51  int addNumericLiteral(char *literal) {
52      for (int i = 0; i < numericLiteralCount; i++) {
53          if (strcmp(numericLiterals[i].literal, literal) == 0) {
54              return numericLiterals[i].position;
55          }
56      }
57      numericLiterals[numericLiteralCount].position = numericLiteralCount;
58      numericLiterals[numericLiteralCount].literal = strdup(literal);
59      return numericLiteralCount++;
60  }

```

Tabla de Literales de Cadenas

Aquí se almacenarán las constantes de tipo cadena.

- **Campos:**
 - Posición
 - Valor de la cadena
- **Funcionalidad:**
 - Se almacena cada cadena literal encontrada en el código.
 - Se usa una **lista enlazada simple** para la inserción eficiente.

```

62  /* Función para agregar un literal de cadena a la tabla */
63  int addStringLiteral(char *string) {
64      for (int i = 0; i < stringLiteralCount; i++) {
65          if (strcmp(stringLiterals[i].string, string) == 0) {
66              return stringLiterals[i].position;
67          }
68      }
69      stringLiterals[stringLiteralCount].position = stringLiteralCount;
70      stringLiterals[stringLiteralCount].string = strdup(string);
71      return stringLiteralCount++;
72  }

```

A esto le añadimos una función para imprimir las tablas y manejar los errores léxicos:

```

74  /* Función para imprimir las tablas */
75  void printTables() {
76      printf("\nTabla de Símbolos:\n");
77      for (int i = 0; i < symbolCount; i++) {
78          printf("Posición: %d, Identificador: %s, Tipo: %d\n", symbols[i].position, symbols[i].identifier, symbols[i].type);
79      }
80
81      printf("\nTabla de Literales Numéricos:\n");
82      for (int i = 0; i < numericLiteralCount; i++) {
83          printf("Posición: %d, Literal: %s\n", numericLiterals[i].position, numericLiterals[i].literal);
84      }
85
86      printf("\nTabla de Literales de Cadenas:\n");
87      for (int i = 0; i < stringLiteralCount; i++) {
88          printf("Posición: %d, Cadena: %s\n", stringLiterals[i].position, stringLiterals[i].string);
89      }
90  }
91
92  /* Función para manejar errores léxicos */
93  void reportError(const char *msg) {
94      printf("Error léxico: Token no reconocido: %s\n", msg);
95  }
96
97  %}

```

3.2 Implementación

- **Uso de Lex/Flex** para definir reglas léxicas y generar el analizador.

```
99  /* Expresiones regulares para los tokens */
100 DIGITO    [0-9]
101 PALABRA_RESERVADA (Bool|Cade|Continuar|Devo|Ent|Fals|Flota|Global|Haz|Mientras|Nulo|Para|Parar|Si|Sino|Ver)
102 SIMBOLOS_ESPECIALES (<|>|<<|>>|#|#!|\\*|\\|'|")
103 IDENT      @[A-Za-zÁÉÍÓÚÑáéíóúñ][A-Za-zÁÉÍÓÚÑáéíóúñ0-9]*_
104 OP_ARITHMETIC (sum|rest|mult|div|mod|inc|dec|exp|dive)
105 OP_RELACIONAL (h|m|e|c|h|e|me)
106 OP_ASIGNACION (">|"\\+>|">|"\\*->|"\\/>|"%->|">>|"<<|"\\^>|"&->")
107 ENTERO      [-+]?[1-9]{DIGITO}*{DIGITO}?[p|g]?
108 REAL        [-+]?{DIGITO}+('){DIGITO}+)?|{DIGITO}+[rR]
109 CADENA      "--",*?"--"
110 COMMENT     \[[^\]]*\]
111 WS          [\t\n]+
112
```

- **Implementación en C** para almacenar y manejar la tabla de símbolos y literales.

```
115 {PALABRA_RESERVADA} {
116     /* Palabras reservadas */
117     if (strcmp(yytext, "Bool") == 0) {
118         printf("(0,0) %s\n", yytext);
119     } else if (strcmp(yytext, "Cade") == 0) {
120         printf("(0,1) %s\n", yytext);
121     } else if (strcmp(yytext, "Continuar") == 0) {
122         printf("(0,2) %s\n", yytext);
123     } else if (strcmp(yytext, "Devo") == 0) {
124         printf("(0,3) %s\n", yytext);
125     } else if (strcmp(yytext, "Ent") == 0) {
126         printf("(0,4) %s\n", yytext);
127     } else if (strcmp(yytext, "Fals") == 0) {
128         printf("(0,5) %s\n", yytext);
129     } else if (strcmp(yytext, "Flota") == 0) {
130         printf("(0,6) %s\n", yytext);
131     } else if (strcmp(yytext, "Global") == 0) {
132         printf("(0,7) %s\n", yytext);
133     } else if (strcmp(yytext, "Haz") == 0) {
134         printf("(0,8) %s\n", yytext);
135     } else if (strcmp(yytext, "Mientras") == 0) {
136         printf("(0,9) %s\n", yytext);
137     } else if (strcmp(yytext, "Nulo") == 0) {
138         printf("(0,10) %s\n", yytext);
139     } else if (strcmp(yytext, "Para") == 0) {
140         printf("(0,11) %s\n", yytext);
141     } else if (strcmp(yytext, "Parar") == 0) {
142         printf("(0,12) %s\n", yytext);
143     } else if (strcmp(yytext, "Si") == 0) {
144         printf("(0,13) %s\n", yytext);
145     } else if (strcmp(yytext, "Sino") == 0) {
146         printf("(0,14) %s\n", yytext);
147     } else if (strcmp(yytext, "Ver") == 0) {
148         printf("(0,15) %s\n", yytext);
149     }
150 }
```

```

152 {SIMBOLOS_ESPECIALES} {
153     /* Símbolos especiales */
154     if (strcmp(yytext, "<") == 0) {
155         printf("(1,0) %s\n", yytext);
156     } else if (strcmp(yytext, ">") == 0) {
157         printf("(1,1) %s\n", yytext);
158     } else if (strcmp(yytext, "<<") == 0) {
159         printf("(1,2) %s\n", yytext);
160     } else if (strcmp(yytext, ">>") == 0) {
161         printf("(1,3) %s\n", yytext);
162     } else if (strcmp(yytext, "#") == 0) {
163         printf("(1,4) %s\n", yytext);
164     } else if (strcmp(yytext, "#!") == 0) {
165         printf("(1,5) %s\n", yytext);
166     } else if (strcmp(yytext, "*") == 0) {
167         printf("(1,6) %s\n", yytext);
168     } else if (strcmp(yytext, "|") == 0) {
169         printf("(1,7) %s\n", yytext);
170     } else if (strcmp(yytext, "°") == 0) {
171         printf("(1,8) %s\n", yytext);
172     }
173 }
174
175 {OP_ARITHMETIC} {
176     /* Operadores aritméticos */
177     if (strcmp(yytext, "sum") == 0) {
178         printf("(3,0) %s\n", yytext);
179     } else if (strcmp(yytext, "rest") == 0) {
180         printf("(3,1) %s\n", yytext);
181     } else if (strcmp(yytext, "mult") == 0) {
182         printf("(3,2) %s\n", yytext);
183     } else if (strcmp(yytext, "div") == 0) {
184         printf("(3,3) %s\n", yytext);
185     } else if (strcmp(yytext, "mod") == 0) {
186         printf("(3,4) %s\n", yytext);
187     } else if (strcmp(yytext, "inc") == 0) {
188         printf("(3,5) %s\n", yytext);
189     } else if (strcmp(yytext, "dec") == 0) {
190         printf("(3,6) %s\n", yytext);
191     } else if (strcmp(yytext, "exp") == 0) {
192         printf("(3,7) %s\n", yytext);
193     } else if (strcmp(yytext, "dive") == 0) {
194         printf("(3,8) %s\n", yytext);
195     }
196 }

```

```

198 {OP_RELACIONAL} {
199     /* Operadores relacionales */
200     if (strcmp(yytext, "h") == 0) {
201         printf("(4,0) %s\n", yytext);
202     } else if (strcmp(yytext, "m") == 0) {
203         printf("(4,1) %s\n", yytext);
204     } else if (strcmp(yytext, "e") == 0) {
205         printf("(4,2) %s\n", yytext);
206     } else if (strcmp(yytext, "c") == 0) {
207         printf("(4,3) %s\n", yytext);
208     } else if (strcmp(yytext, "he") == 0) {
209         printf("(4,4) %s\n", yytext);
210     } else if (strcmp(yytext, "me") == 0) {
211         printf("(4,5) %s\n", yytext);
212     }
213 }
214

```

```

215 {IDENT} {
216     int pos = addSymbol(yytext);
217     printf("(2,%d) %s\n", pos, yytext);
218 }
219
220 {OP_ASIGNACION} {
221     /* Operadores de asignación */
222     if (strcmp(yytext, "->") == 0) {
223         printf("(5,0) %s\n", yytext);
224     } else if (strcmp(yytext, "+->") == 0) {
225         printf("(5,1) %s\n", yytext);
226     } else if (strcmp(yytext, "-->") == 0) {
227         printf("(5,2) %s\n", yytext);
228     } else if (strcmp(yytext, "*->") == 0) {
229         printf("(5,3) %s\n", yytext);
230     } else if (strcmp(yytext, "/->") == 0) {
231         printf("(5,4) %s\n", yytext);
232     } else if (strcmp(yytext, "%->") == 0) {
233         printf("(5,5) %s\n", yytext);
234     } else if (strcmp(yytext, ">>") == 0) {
235         printf("(5,6) %s\n", yytext);
236     } else if (strcmp(yytext, "<<") == 0) {
237         printf("(5,7) %s\n", yytext);
238     } else if (strcmp(yytext, "^->") == 0) {
239         printf("(5,8) %s\n", yytext);
240     } else if (strcmp(yytext, "&->") == 0) {
241         printf("(5,9) %s\n", yytext);
242     }
243 }
244

```

```

245 {CADENA} {
246     int pos = addStringLiteral(yytext);
247     printf("(6,%d) %s\n", pos, yytext);
248 }
249
250 {ENTERO} {
251     int pos = addNumericLiteral(yytext);
252     printf("(7,%d) %s\n", pos, yytext);
253 }
254
255 {REAL} {
256     int pos = addNumericLiteral(yytext);
257     printf("(6,%d) %s\n", pos, yytext);
258 }
259
260 {COMMENT} {
261     printf("Comentario ignorado: %s\n", yytext);
262 }
263
264 {WS} /* Ignorar espacios en blanco */
265
266 . {
267     /* Manejo de errores para caracteres no reconocidos */
268     reportError(yytext);
269 }
270 %%
271

```

- **Salida del programa:**
 - Tabla de símbolos
 - Tabla de literales
 - Secuencia de tokens generada
 - Manejo de errores y reporte

4. Instrucciones de Ejecución

Compilar el archivo lex:

1. flex analizador.l
2. gcc lex.yy.c -o analizador
3. Ejecutar el analizador con un archivo fuente:
./analizador entrada.txt
4. Revisar la salida en pantalla o en los archivos generados.

```

272  /* Punto de entrada del programa */
273  int main(int argc, char *argv[]) {
274      if (argc < 2) {
275          fprintf(stderr, "Se debe proporcionar un archivo de entrada.\n");
276          return 1;
277      }
278
279      FILE *file = fopen(argv[1], "r");
280      if (!file) {
281          perror("Error al abrir el archivo");
282          return 1;
283      }
284
285      yyin = file;
286      yylex();
287
288      /* Imprimir las tablas después del análisis */
289      printTables();
290
291      fclose(file);
292      return 0;
293  }

```

```

[bante@pc-bd-ace Prueba3]$ lex Prueba3.l
[bante@pc-bd-ace Prueba3]$ gcc lex.yy.c -o lexer -lfl
[bante@pc-bd-ace Prueba3]$ ./lexer PruebaConcreta.txt

```


Tabla de Símbolos:

Tabla de Literales Numéricos:

Posición: 0, Literal: 123

Posición: 1, Literal: 45

Posición: 2, Literal: 67

Posición: 3, Literal: -89

Posición: 4, Literal: +3

Posición: 5, Literal: 14

Posición: 6, Literal: 0

Tabla de Literales de Cadenas:

Posición: 0, Cadena: -- Este es un archivo de prueba para el analizador léxico --

Posición: 1, Cadena: -- Identificadores --

Posición: 2, Cadena: -- Palabras reservadas --

Posición: 3, Cadena: -- Operadores aritméticos --

Posición: 4, Cadena: -- Operadores relacionales --

Posición: 5, Cadena: -- Operadores de asignación --

Posición: 6, Cadena: -- Literales numéricos --

Posición: 7, Cadena: -- Literales de cadena --

Posición: 8, Cadena: --Hola, mundo--

Posición: 9, Cadena: --Este es un literal de cadena--

Posición: 10, Cadena: -- Símbolos especiales --

Posición: 11, Cadena: -- Comentarios --

[bante@pc-bd-ace Prueba3]\$ █

[bante@pc-bd-ace Prueba3]\$ cat Prueba3.l

File: Prueba3.l

```
1  %option noyywrap
2
3  %{
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7
8  typedef struct {
9      int position;
10     char *identifier;
11     int type; /* Inicializado en -1 */
12 } SymbolTable;
13
14 int addSymbol(char *name);
15 int addNumericLiteral(char *literal);
16 int addStringLiteral(char *string);
17 void printTables();
18 void reportError(const char *msg);
19
20 typedef struct {
21     int position;
22     char *literal;
23 } LiteralTable;
24
25 typedef struct {
26     int position;
27     char *string;
28 } StringLiteralTable;
```

Diseño:

Al iniciar este proyecto, lo primero que consideramos fue cómo queríamos que se presentara la salida del programa. Dado que este análisis debía generar tres tablas distintas, nos enfocamos en definir la mejor manera de mostrarlas en la consola. Las tablas necesarias para este proyecto son:

1. **Tabla de Símbolos**
2. **Tabla de Literales Numéricas**
3. **Tabla de Literales de Cadena**

Además de estas tablas, también era esencial mostrar los tokens generados durante el análisis léxico.

Para el diseño de la salida, decidimos optar por un enfoque simple y funcional que permitiera visualizar la información de manera clara. La presentación en la consola sigue un orden específico:

1. Primero, se imprimen los **tokens generados**, seguidos de la **cadena** correspondiente a cada token.
2. Luego, se presentan las **tablas** en el siguiente orden: **Tabla de Símbolos**, **Tabla de Literales Numéricas** y **Tabla de Literales de Cadena**, mostrando en cada una los elementos almacenados junto con su información relevante.

Este método de visualización nos permite estructurar la información de manera ordenada y comprensible, facilitando la interpretación de los datos obtenidos tras el análisis léxico.

```
axelmg@DESKTOP-DGTBLC0:~$ nano prueba.l
axelmg@DESKTOP-DGTBLC0:~$ flex prueba.l
axelmg@DESKTOP-DGTBLC0:~$ gcc lex.yy.c -o prueba.l
axelmg@DESKTOP-DGTBLC0:~$ ./prueba.l prueba2.txt
(6,0) -- Este es un archivo de prueba para el analizador léxico --
(6,1) -- Identificadores --
(2,0) @miVariable_
Error léxico: Token no reconocido: o
Error léxico: Token no reconocido: t
Error léxico: Token no reconocido: r
Error léxico: Token no reconocido: a
Error léxico: Token no reconocido: V
Error léxico: Token no reconocido: a
Error léxico: Token no reconocido: r
Error léxico: Token no reconocido: i
Error léxico: Token no reconocido: a
Error léxico: Token no reconocido: b
Error léxico: Token no reconocido: l
(4,2) e
Error léxico: Token no reconocido: _
Error léxico: Token no reconocido: @
Error léxico: Token no reconocido: v
Error léxico: Token no reconocido: a
Error léxico: Token no reconocido: r
Error léxico: Token no reconocido: i
Error léxico: Token no reconocido: a
Error léxico: Token no reconocido: b
Error léxico: Token no reconocido: l
(4,2) e
(7,0) 123
(2,1) @áéíóúñ_
(6,2) -- Palabras reservadas --
(0,0) Bool
(0,1) Cade
(0,2) Continuar
(0,3) Devo
(0,4) Ent
(0,5) Fals
(0,6) Flota
(0,7) Global
(0,8) Haz
(0,9) Mientras
(0,10) Nulo
(0,11) Para
(0,12) Parar
(0,13) Si
(0,14) Sino
(0,15) Ver
(6,3) -- Operadores aritméticos --
(3,0) sum
(3,1) rest
(3,2) mult
(3,3) div
(3,4) mod
(3,5) inc
(3,6) dec
(3,7) exp
(3,8) dive
(6,4) -- Operadores relacionales --
(4,0) h
```

```

(4,1) m
(4,2) e
(4,3) c
(4,4) he
(4,5) me
(6,5) -- Operadores de asignación --
(5,0) ->
Error léxico: Token no reconocido: +
(5,0) ->
(5,2) -->
(1,6) *
(5,0) ->
(5,4) /->
(5,5) %->
(1,3) >>
(1,2) <<
Error léxico: Token no reconocido: ^
(5,0) ->
(5,9) &->
(6,6) -- Literales numéricos --
(7,0) 123
(7,1) 45
Error léxico: Token no reconocido: .
(7,2) 67
(7,3) -89
(7,4) +3
Error léxico: Token no reconocido: .
(7,5) 14
(6,6) 0
(6,7) -- Literales de cadena --
Error léxico: Token no reconocido: "
(6,8) --Hola, mundo--
Error léxico: Token no reconocido: "
Error léxico: Token no reconocido: "
(6,9) --Este es un literal de cadena--
Error léxico: Token no reconocido: "
(6,10) -- Símbolos especiales --
(1,0) <
(1,1) >
(1,2) <<
(1,3) >>
(1,4) #
(1,5) #!
(1,6) *
(1,7) |
(1,8) °
(6,11) -- Comentarios --
Comentario ignorado: [Este es un comentario de una línea]
Comentario ignorado: [Este es otro comentario
que abarca varias líneas]

Tabla de Símbolos:
Posición: 0, Identificador: @miVariable_, Tipo: -1
Posición: 1, Identificador: @áéíóúñ_, Tipo: -1

Tabla de Literales Numéricos:
Posición: 0, Literal: 123
Posición: 1, Literal: 45
Posición: 2, Literal: 67
Posición: 3, Literal: -89
Posición: 4, Literal: +3
Posición: 5, Literal: 14
Posición: 6, Literal: 0

```

```

Tabla de Símbolos:
Posición: 0, Identificador: @miVariable_, Tipo: -1
Posición: 1, Identificador: @áéíóúñ_, Tipo: -1

Tabla de Literales Numéricos:
Posición: 0, Literal: 123
Posición: 1, Literal: 45
Posición: 2, Literal: 67
Posición: 3, Literal: -89
Posición: 4, Literal: +3
Posición: 5, Literal: 14
Posición: 6, Literal: 0

Tabla de Literales de Cadenas:
Posición: 0, Cadena: -- Este es un archivo de prueba para el analizador léxico --
Posición: 1, Cadena: -- Identificadores --
Posición: 2, Cadena: -- Palabras reservadas --
Posición: 3, Cadena: -- Operadores aritméticos --
Posición: 4, Cadena: -- Operadores relacionales --
Posición: 5, Cadena: -- Operadores de asignación --
Posición: 6, Cadena: -- Literales numéricos --
Posición: 7, Cadena: -- Literales de cadena --
Posición: 8, Cadena: --Hola, mundo--
Posición: 9, Cadena: --Este es un literal de cadena--
Posición: 10, Cadena: -- Símbolos especiales --
Posición: 11, Cadena: -- Comentarios --

```

Tipo de estructura de datos que se empleó, así como el método de búsqueda utilizado:

El programa emplea **arreglos de estructuras** en C para almacenar y organizar la información extraída durante el análisis léxico. Cada tipo de dato identificado, como identificadores, literales numéricos y cadenas de texto, se almacena en su propia tabla utilizando una estructura de datos específica.

1. **Identificadores:** Se guardan en una estructura diseñada específicamente para registrar los nombres de variables, funciones y otros símbolos utilizados en el código.
2. **Literales numéricos:** Se almacenan en una estructura que permite registrar valores numéricos, incluyendo enteros y flotantes, junto con su correspondiente posición dentro de la tabla de literales.
3. **Cadenas de texto:** Se gestionan en una estructura destinada a guardar constantes de tipo cadena, registrando su contenido y la posición en la tabla correspondiente.

Para la búsqueda dentro de estas tablas, se utilizó un **recorrido secuencial** debido a la naturaleza del almacenamiento en arreglos. Este método consiste en recorrer cada elemento de la tabla uno por uno hasta encontrar el dato buscado, si el valor se encuentra, el algoritmo retorna la posición del elemento en el arreglo; de lo contrario, continúa hasta el final del arreglo. Aunque este enfoque es simple y eficiente para volúmenes de datos moderados, para manejar una gran cantidad de información, podríamos considerar otras estrategias como el uso de **tablas hash** o **árboles balanceados** para mejorar la eficiencia en la búsqueda y recuperación de datos.

Se implementó un sistema de almacenamiento de tokens durante el análisis léxico utilizando tres tablas: una para los identificadores, una para los literales numéricos y otra para los literales de cadenas de texto. Las tablas se implementan como arreglos de estructuras, donde cada estructura guarda la información relevante para cada tipo de token.

Para SymbolTable (Tabla de Símbolos):

- La tabla almacenará los identificadores del código, es decir, las variables, funciones, etc.
- Cada entrada en esta tabla será una estructura que contendrá el nombre del identificador y el tipo de dato.
- Se usará una función llamada `addSymbol(char *name)` para agregar un identificador a la tabla. Esta función buscará el identificador en la tabla de símbolos para verificar si ya está almacenado. Si encuentra el identificador, devolverá la posición donde se encuentra en la tabla. Si no lo encuentra, lo agregará al final de la tabla y actualizará el contador de símbolos (`symbolCount`).

```

typedef struct {
    int position;
    char *identifier;
    int type; /* Inicializado en -1 */
} SymbolTable;

/* Función para agregar un símbolo a la tabla */
int addSymbol(char *name) {
    for (int i = 0; i < symbolCount; i++) {
        if (strcmp(symbols[i].identifier, name) == 0) {
            return symbols[i].position;
        }
    }
    symbols[symbolCount].position = symbolCount;
    symbols[symbolCount].identifier = strdup(name);
    symbols[symbolCount].type = -1;
    return symbolCount++;
}

```

Para LiteralTable (Tabla de Literales Numéricos):

- Se almacenarán los literales numéricos que aparezcan en el código, números enteros o flotantes.
- Cada entrada en la tabla será un valor numérico.
- Se implementará la función `addNumericLiteral(char *literal)` para agregar un literal numérico. Esta función buscará el literal numérico en la tabla de literales. Si ya existe, devolverá la posición en la que está. Si no existe, lo añadirá al final de la tabla y aumentará el contador de literales numéricos (`numericLiteralCount`).

```

typedef struct {
    int position;
    char *literal;
} LiteralTable;

/* Función para agregar un literal numérico a la tabla */
int addNumericLiteral(char *literal) {
    for (int i = 0; i < numericLiteralCount; i++) {
        if (strcmp(numericLiterals[i].literal, literal) == 0) {
            return numericLiterals[i].position;
        }
    }
    numericLiterals[numericLiteralCount].position = numericLiteralCount;
    numericLiterals[numericLiteralCount].literal = strdup(literal);
    return numericLiteralCount++;
}

```

Para StringLiteralTable (Tabla de Literales de Cadenas):

- Dicha tabla almacenará las cadenas de texto encontradas en el código fuente.
- Cada entrada será una cadena de caracteres.
- Se usará la función `addStringLiteral(char *string)` para manejar los literales de cadena. Buscará la cadena en la tabla de literales de cadenas. Si ya está presente, devolverá la posición de la cadena en la tabla. Si no está, la agregará y actualizará el contador de literales de cadenas (`stringLiteralCount`).

```

typedef struct {
    int position;
    char *string;
} StringLiteralTable;

/* Función para agregar un literal de cadena a la tabla */
int addStringLiteral(char *string) {
    for (int i = 0; i < stringLiteralCount; i++) {
        if (strcmp(stringLiterals[i].string, string) == 0) {
            return stringLiterals[i].position;
        }
    }
    stringLiterals[stringLiteralCount].position = stringLiteralCount;
    stringLiterals[stringLiteralCount].string = strdup(string);
    return stringLiteralCount++;
}

```

Se tendrá un control sobre los contadores:

- Se mantendrán tres contadores para llevar un control del número de elementos en cada tabla:
 - `symbolCount`: El número de identificadores almacenados en la tabla de símbolos.
 - `numericLiteralCount`: El número de literales numéricos almacenados.

- `stringLiteralCount`: El número de literales de cadenas almacenados.
- Cada vez que se agregue un nuevo elemento a cualquiera de las tablas, se incrementará el contador correspondiente.

```
SymbolTable symbols[100];  
int symbolCount = 0;  
LiteralTable numericLiterals[100];  
int numericLiteralCount = 0;  
StringLiteralTable stringLiterals[100];  
int stringLiteralCount = 0;
```

Funcionamiento:

El programa se encarga de reconocer y clasificar diferentes tipos de tokens en un archivo de entrada, como identificadores, constantes numéricas, cadenas de texto, operadores y símbolos especiales. Para ello, construye tablas que almacenan los identificadores, literales numéricos y literales de cadenas, asegurándose de que cada entrada sea única. Además, el sistema maneja errores léxicos detectando caracteres no reconocidos y notificando al usuario, permitiendo que el análisis continúe sin interrupciones. Al finalizar el proceso, el programa imprime las tablas de símbolos y literales.

El flujo del programa se divide en varias secciones: primero se configura Flex para el análisis léxico, luego se definen los patrones de los diferentes tipos de tokens, se asignan las acciones correspondientes a cada uno de estos patrones, y finalmente se implementan funciones auxiliares para gestionar las tablas y los errores.

Bibliotecas Utilizadas:

- `#include <stdio.h>`: Proporciona funciones de entrada y salida, como `printf`.
- `#include <stdlib.h>`: Contiene funciones de manejo de memoria y control de procesos.
- `#include <string.h>`: Proporciona funciones para manipular cadenas de caracteres.

SymbolTable: Define una tabla de símbolos para almacenar identificadores.

```
typedef struct {  
    int position;  
    char *identifier;  
    int type; /* Inicializado en -1 */  
} SymbolTable;
```

Esta estructura contiene:

- position: La posición del identificador en la tabla.
- identifier: El nombre del identificador.
- type: Un tipo asociado al identificador.

LiteralTable: Define una tabla para almacenar literales numéricos.

```
typedef struct {  
    int position;  
    char *literal;  
} LiteralTable;
```

- position: La posición del literal en la tabla.
- literal: El valor numérico del literal como cadena.

StringLiteralTable: Define una tabla para almacenar literales de cadenas

```
typedef struct {  
    int position;  
    char *string;  
} StringLiteralTable;
```

- position: La posición del literal de cadena en la tabla.
- string: El contenido de la cadena.

Variables Globales:

Se configura la estructura de datos para almacenar los símbolos y literales detectados durante el análisis léxico. Los contadores gestionan la inserción de nuevos elementos en las tablas, asegurando que cada símbolo o literal se agregue de manera correcta y ordenada, con un tamaño predefinido para las tablas.

SymbolTable symbols[100]; Es un arreglo de estructuras para almacenar los identificadores que se encuentran durante el análisis léxico. Se utiliza para mantener los símbolos del código fuente.

int symbolCount = 0; Es un contador que lleva el número de identificadores que se han almacenado en la tabla de símbolos.

LiteralTable numericLiterals[100]; Es un arreglo de estructuras para almacenar los literales numéricos encontrados durante el análisis léxico.

int numericLiteralCount = 0; Es un contador que lleva el número de literales numéricos almacenados en la tabla de literales numéricos.

StringLiteralTable stringLiterals[100]; Es un arreglo de estructuras para almacenar los literales de cadenas encontradas durante el análisis léxico.

int stringLiteralCount = 0; Es un contador que lleva el número de literales de cadenas almacenados en la tabla de literales de cadenas.

```
SymbolTable symbols[100];
int symbolCount = 0;
LiteralTable numericLiterals[100];
int numericLiteralCount = 0;
StringLiteralTable stringLiterals[100];
int stringLiteralCount = 0;
```

Funciones Auxiliares del código:

int addSymbol(char *name): Es la que se encarga de agregar un identificador a la tabla de símbolos. Si el identificador ya existe, devuelve su posición; si no, lo agrega y devuelve su nueva posición.

int addNumericLiteral(char *literal): Es la que agrega un literal numérico a la tabla de literales numéricos. Si el literal ya existe, retorna su posición; si no, lo agrega y devuelve su posición.

int addStringLiteral(char *string): Es la función que agrega una literal de cadena a la tabla de literales de cadenas. Si el literal de cadena ya está presente, devuelve su posición; si no, lo agrega y devuelve su posición.

void printTables(): Es la que imprime las tablas de símbolos, literales numéricos y literales de cadenas, mostrando los elementos almacenados con su respectiva posición.

void reportError(const char *msg): Es la que se utiliza para manejar errores léxicos, mostrando un mensaje de error cuando se encuentra un token no reconocido durante el análisis.

```
int addSymbol(char *name);
int addNumericLiteral(char *literal);
int addStringLiteral(char *string);
void printTables();
void reportError(const char *msg);
```

```
/* Función para agregar un símbolo a la tabla */
int addSymbol(char *name) {
    for (int i = 0; i < symbolCount; i++) {
        if (strcmp(symbols[i].identifier, name) == 0) {
            return symbols[i].position;
        }
    }
    symbols[symbolCount].position = symbolCount;
    symbols[symbolCount].identifier = strdup(name);
    symbols[symbolCount].type = -1;
    return symbolCount++;
}
```

```
/* Función para agregar un literal numérico a la tabla */
int addNumericLiteral(char *literal) {
    for (int i = 0; i < numericLiteralCount; i++) {
        if (strcmp(numericLiterals[i].literal, literal) == 0) {
            return numericLiterals[i].position;
        }
    }
    numericLiterals[numericLiteralCount].position = numericLiteralCount;
    numericLiterals[numericLiteralCount].literal = strdup(literal);
    return numericLiteralCount++;
}
```

```
/* Función para agregar un literal de cadena a la tabla */
int addStringLiteral(char *string) {
    for (int i = 0; i < stringLiteralCount; i++) {
        if (strcmp(stringLiterals[i].string, string) == 0) {
            return stringLiterals[i].position;
        }
    }
    stringLiterals[stringLiteralCount].position = stringLiteralCount;
    stringLiterals[stringLiteralCount].string = strdup(string);
    return stringLiteralCount++;
}
```

```

/* Función para imprimir las tablas */
void printTables() {
    printf("\nTabla de Símbolos:\n");
    for (int i = 0; i < symbolCount; i++) {
        printf("Posición: %d, Identificador: %s, Tipo: %d\n", symbols[i].position, symbols[i].identifier, symbols[i].type);
    }

    printf("\nTabla de Literales Numéricos:\n");
    for (int i = 0; i < numericLiteralCount; i++) {
        printf("Posición: %d, Literal: %s\n", numericLiterals[i].position, numericLiterals[i].literal);
    }

    printf("\nTabla de Literales de Cadenas:\n");
    for (int i = 0; i < stringLiteralCount; i++) {
        printf("Posición: %d, Cadena: %s\n", stringLiterals[i].position, stringLiterals[i].string);
    }
}

/* Función para manejar errores léxicos */
void reportError(const char *msg) {
    printf("Error léxico: Token no reconocido: %s\n", msg);
}

```

Definiciones de las expresiones regulares del código.

Mediante expresiones regulares se identifican diferentes tipos de tokens:

1. DIGITO [0-9]
2. PALABRA_RESERVADA
(Bool|Cade|Continuar|Devo|Ent|Fals|Flota|Global|Haz|Mientras|Nulo|Para|Parar|Si|Sino|Ver)
3. SIMBOLOS_ESPECIALES (<|>|<<|>>|#|#!|*|\\|°)
4. IDENT @[A-Za-zÁÉÍÓÚÑáéíóúñ][A-Za-zÁÉÍÓÚÑáéíóúñ0-9]*_
5. OP_ARITHMETIC (sum|rest|mult|div|mod|inc|dec|exp|dive)
6. OP_RELACIONAL (h|m|e|c|he|me)
7. OP_ASIGNACION (">"|"\\ "+>"|"-->"|"*->"|"/->"|"%->"|">>"|"<<"|"\\^->"|"&->")
8. ENTERO [-+]?[1-9]{DIGITO}*{DIGITO}?[p|g]?
9. REAL [-+]?{DIGITO}+('{DIGITO}+)?|{DIGITO}+[rR]
10. CADENA "--".*?"--"
11. COMMENT \\[(^)]*\\
12. WS [\\t\\n]+

En el código aparecen como:

```
/* Expresiones regulares para los tokens */
DIGITO    [0-9]
PALABRA_RESERVADA  (Bool|Cade|Continuar|Devo|Ent|Fals|Flota|Global|Haz|Mientras|Nulo|Para|Parar|Si|Sino|Ver)
SIMBOLOS_ESPECIALES  (<|>|<<|>>|#|#!|\\*|\\|\\°)
IDENT      @[A-Za-zÁÉÍÓÚÑáéíóúñ][A-Za-zÁÉÍÓÚÑáéíóúñ0-9]*_
OP_ARITHMETIC  (sum|rest|mult|div|mod|inc|dec|exp|dive)
OP_RELACIONAL  (h|m|e|c|he|me)
OP_ASIGNACION  ("->"|"\\+>"|"--->"|"\\*>"|"/->"|"%->"|">>"|"<<"|"\\^>"|"&->")
ENTERO        [-+]?[1-9]{DIGITO}*{DIGITO}?[p|g]?
REAL          [-+]?{DIGITO}+(' {DIGITO}+)?|{DIGITO}+[rR]
CADENA        "___" . *? "___"
COMMENT       \[ [^\\] * \]
WS            [ \t\n]+
```

Reglas del código.

En Flex, las reglas se definen en la sección central del archivo (%%), donde se especifican patrones de texto junto con las acciones en C que se ejecutan cuando se detectan esos patrones. Estas reglas sirven para reconocer y clasificar los diferentes componentes léxicos del código fuente.

PALABRA RESERVADA.

Esta regla del código se activa cuando se reconoce una palabra reservada en el código. Dependiendo del texto encontrado, el analizador imprime un identificador numérico junto con la palabra reservada. Por ejemplo, si encuentra la palabra "Bool", imprimirá el código (0,0) seguido de "Bool". Esto permite identificar y clasificar palabras clave de un lenguaje de programación, asociándolas con su correspondiente valor en un conjunto predefinido.

```
{PALABRA_RESERVADA} {  
    /* Palabras reservadas */  
    if (strcmp(yytext, "Bool") == 0) {  
        printf("(0,0) %s\n", yytext);  
    } else if (strcmp(yytext, "Cade") == 0) {  
        printf("(0,1) %s\n", yytext);  
    } else if (strcmp(yytext, "Continuar") == 0) {  
        printf("(0,2) %s\n", yytext);  
    } else if (strcmp(yytext, "Devo") == 0) {  
        printf("(0,3) %s\n", yytext);  
    } else if (strcmp(yytext, "Ent") == 0) {  
        printf("(0,4) %s\n", yytext);  
    } else if (strcmp(yytext, "Fals") == 0) {  
        printf("(0,5) %s\n", yytext);  
    } else if (strcmp(yytext, "Flota") == 0) {  
        printf("(0,6) %s\n", yytext);  
    } else if (strcmp(yytext, "Global") == 0) {  
        printf("(0,7) %s\n", yytext);  
    } else if (strcmp(yytext, "Haz") == 0) {  
        printf("(0,8) %s\n", yytext);  
    } else if (strcmp(yytext, "Mientras") == 0) {  
        printf("(0,9) %s\n", yytext);  
    } else if (strcmp(yytext, "Nulo") == 0) {  
        printf("(0,10) %s\n", yytext);  
    } else if (strcmp(yytext, "Para") == 0) {  
        printf("(0,11) %s\n", yytext);  
    } else if (strcmp(yytext, "Parar") == 0) {  
        printf("(0,12) %s\n", yytext);  
    } else if (strcmp(yytext, "Si") == 0) {  
        printf("(0,13) %s\n", yytext);  
    } else if (strcmp(yytext, "Sino") == 0) {  
        printf("(0,14) %s\n", yytext);  
    } else if (strcmp(yytext, "Ver") == 0) {  
        printf("(0,15) %s\n", yytext);  
    }  
}
```

SIMBOLOS ESPECIALES

Esta regla del código se activa cuando se encuentra un símbolo especial, como operadores o caracteres especiales. La acción correspondiente es imprimir un código numérico junto con el símbolo encontrado. Por ejemplo, si se detecta el símbolo "<", se imprime (1, 0) seguido de "<". Esta regla ayuda a reconocer los símbolos que tienen un significado particular en el lenguaje..

```
{SIMBOLOS_ESPECIALES} {  
    /* Símbolos especiales */  
    if (strcmp(yytext, "<") == 0) {  
        printf("(1,0) %s\n", yytext);  
    } else if (strcmp(yytext, ">") == 0) {  
        printf("(1,1) %s\n", yytext);  
    } else if (strcmp(yytext, "<<") == 0) {  
        printf("(1,2) %s\n", yytext);  
    } else if (strcmp(yytext, ">>") == 0) {  
        printf("(1,3) %s\n", yytext);  
    } else if (strcmp(yytext, "#") == 0) {  
        printf("(1,4) %s\n", yytext);  
    } else if (strcmp(yytext, "#!") == 0) {  
        printf("(1,5) %s\n", yytext);  
    } else if (strcmp(yytext, "*") == 0) {  
        printf("(1,6) %s\n", yytext);  
    } else if (strcmp(yytext, "|") == 0) {  
        printf("(1,7) %s\n", yytext);  
    } else if (strcmp(yytext, "o") == 0) {  
        printf("(1,8) %s\n", yytext);  
    }  
}
```

OP ARITHMETIC

En esta regla del código, se manejan los operadores aritméticos del lenguaje, como "sum", "rest", "mult", entre otros. Para cada uno de ellos, se imprime un código específico asociado al operador, como (3, 0) para "sum", (3, 1) para "rest", y así sucesivamente. Permite identificar y clasificar los operadores matemáticos presentes en el código fuente.

```

{OP_ARITHMETIC} {
    /* Operadores aritméticos */
    if (strcmp(yytext, "sum") == 0) {
        printf("(3,0) %s\n", yytext);
    } else if (strcmp(yytext, "rest") == 0) {
        printf("(3,1) %s\n", yytext);
    } else if (strcmp(yytext, "mult") == 0) {
        printf("(3,2) %s\n", yytext);
    } else if (strcmp(yytext, "div") == 0) {
        printf("(3,3) %s\n", yytext);
    } else if (strcmp(yytext, "mod") == 0) {
        printf("(3,4) %s\n", yytext);
    } else if (strcmp(yytext, "inc") == 0) {
        printf("(3,5) %s\n", yytext);
    } else if (strcmp(yytext, "dec") == 0) {
        printf("(3,6) %s\n", yytext);
    } else if (strcmp(yytext, "exp") == 0) {
        printf("(3,7) %s\n", yytext);
    } else if (strcmp(yytext, "dive") == 0) {
        printf("(3,8) %s\n", yytext);
    }
}
}

```

OP_RELACIONAL

Esta regla del código se activa para identificar operadores relacionales como "h", "m", "e", etc., en el código. Si se detecta uno de estos operadores, se imprime un código correspondiente (por ejemplo, (4, 0) para "h"). Estos operadores son utilizados para comparar valores y, por lo tanto, son fundamentales para la lógica del lenguaje.

```

{OP_RELACIONAL} {
    /* Operadores relacionales */
    if (strcmp(yytext, "h") == 0) {
        printf("(4,0) %s\n", yytext);
    } else if (strcmp(yytext, "m") == 0) {
        printf("(4,1) %s\n", yytext);
    } else if (strcmp(yytext, "e") == 0) {
        printf("(4,2) %s\n", yytext);
    } else if (strcmp(yytext, "c") == 0) {
        printf("(4,3) %s\n", yytext);
    } else if (strcmp(yytext, "he") == 0) {
        printf("(4,4) %s\n", yytext);
    } else if (strcmp(yytext, "me") == 0) {
        printf("(4,5) %s\n", yytext);
    }
}
}

```

IDENT

En esta regla del código, se maneja la detección de identificadores. Cuando se reconoce un identificador (como una variable o función), se obtiene su posición al ser insertado en la tabla de símbolos y se imprime el código asociado, que incluye la posición en la tabla. Esto facilita el manejo de las variables y funciones en el código durante el análisis léxico.

```
{IDENT} {  
    int pos = addSymbol(yytext);  
    printf("(2,%d) %s\n", pos, yytext);  
}
```

OP_ASIGNACION

Esta regla del código se activa cuando se detectan operadores de asignación (por ejemplo, " \rightarrow ", " \leftarrow ", etc.). Cada uno de estos operadores tiene un código único asignado, como (5, 0) para " \rightarrow " o (5, 1) para " \leftarrow ". Los operadores de asignación son fundamentales en los lenguajes de programación para asignar valores a variables.

```
{OP_ASIGNACION} {  
    /* Operadores de asignación */  
    if (strcmp(yytext, " $\rightarrow$ ") == 0) {  
        printf("(5,0) %s\n", yytext);  
    } else if (strcmp(yytext, " $\leftarrow$ ") == 0) {  
        printf("(5,1) %s\n", yytext);  
    } else if (strcmp(yytext, " $\leftrightarrow$ ") == 0) {  
        printf("(5,2) %s\n", yytext);  
    } else if (strcmp(yytext, " $\star\rightarrow$ ") == 0) {  
        printf("(5,3) %s\n", yytext);  
    } else if (strcmp(yytext, " $\wedge\rightarrow$ ") == 0) {  
        printf("(5,4) %s\n", yytext);  
    } else if (strcmp(yytext, "% $\rightarrow$ ") == 0) {  
        printf("(5,5) %s\n", yytext);  
    } else if (strcmp(yytext, ">>") == 0) {  
        printf("(5,6) %s\n", yytext);  
    } else if (strcmp(yytext, "<<") == 0) {  
        printf("(5,7) %s\n", yytext);  
    } else if (strcmp(yytext, "^ $\rightarrow$ ") == 0) {  
        printf("(5,8) %s\n", yytext);  
    } else if (strcmp(yytext, "& $\rightarrow$ ") == 0) {  
        printf("(5,9) %s\n", yytext);  
    }  
}
```

CADENA

Esta regla del código maneja la detección de literales de tipo cadena. Cuando se encuentra una cadena, se obtiene su posición en la tabla de literales de cadenas y se imprime el código correspondiente. Por ejemplo, si se encuentra una cadena, se imprime (6, <posición>) <cadena>. Esto permite almacenar y clasificar las cadenas utilizadas en el programa.


```
{CADENA} {
    int pos = addStringLiteral(yytext);
    printf("(6,%d) %s\n", pos, yytext);
}
```

ENTERO

Esta regla se activa cuando se detecta un literal numérico entero. El valor de la constante entera se agrega a la tabla de literales numéricos y se imprime el código correspondiente con la posición en la tabla, como (7, <posición>) <valor>. Esto permite manejar correctamente los valores numéricos enteros dentro del análisis léxico.

```
{ENTERO} {
    int pos = addNumericLiteral(yytext);
    printf("(7,%d) %s\n", pos, yytext);
}
```

REAL

Esta regla del código maneja los literales de tipo flotante (números reales). Al igual que con los literales enteros, se agrega el número real a la tabla de literales numéricos y se imprime el código asociado, como (8, <posición>) <valor>. Esto permite que el analizador lea y clasifique correctamente los números reales en el código fuente.

```
{REAL} {
    int pos = addNumericLiteral(yytext);
    printf("(6,%d) %s\n", pos, yytext);
}
```

COMMENT

Esta regla del código ignora los comentarios del código. Cuando se encuentra un comentario, simplemente se imprime un mensaje indicando que ha sido ignorado, como "Comentario ignorado: <comentario>". Esta regla nos permite omitir los comentarios durante el proceso de análisis léxico sin afectarlos.

```
{COMMENT} {
    printf("Comentario ignorado: %s\n", yytext);
}
```

WS

Esta regla maneja los espacios en blanco (como espacios, tabulaciones o saltos de línea). No realiza ninguna acción, ya que los espacios en blanco se ignoran en el análisis léxico. Esto permite que el analizador pase por alto los espacios que no son relevantes para el análisis sintáctico.

```
{WS} /* Ignorar espacios en blanco */
```

Esta regla maneja los errores, es decir, cuando se encuentra un carácter no reconocido por las demás reglas. En caso de que se detecte un carácter desconocido, se llama a la función `reportError()` para informar sobre el error.

```
. {  
    /* Manejo de errores para caracteres no reconocidos */  
    reportError(yytext);  
}
```

Función Principal:

La función principal (`main`) es el punto de entrada del programa. Primero, verifica si se ha proporcionado un archivo de entrada como argumento (es decir, si el número de argumentos (`argc`) es mayor o igual a 2). Si no se proporciona el archivo, imprime un mensaje de error y termina el programa con un código de error. Luego, intenta abrir el archivo especificado en el argumento utilizando la función `fopen`. Si no puede abrir el archivo, muestra un mensaje de error usando `perror` y termina el programa. Si el archivo se abre correctamente, asigna ese archivo a `yyin` (el archivo de entrada de Flex), y llama a `yylex()`, que inicia el análisis léxico del contenido del archivo. Después de que el análisis léxico se complete, se imprimen las tablas de símbolos y literales mediante `printTables()`. Finalmente, cierra el archivo y termina el programa con un código de éxito.

```
/* Punto de entrada del programa */  
int main(int argc, char *argv[]) {  
    if (argc < 2) {  
        fprintf(stderr, "Se debe proporcionar un archivo de entrada.\n");  
        return 1;  
    }  
  
    FILE *file = fopen(argv[1], "r");  
    if (!file) {  
        perror("Error al abrir el archivo");  
        return 1;  
    }  
  
    yyin = file;  
    yylex();  
  
    /* Imprimir las tablas después del análisis */  
    printTables();  
  
    fclose(file);  
    return 0;  
}
```

Pruebas:

Para la parte de pruebas, se realizaron varias pruebas de la implementación del código, con un archivo como entrada y los resultados fueron los siguientes:

```
[bante@pc-bd-ace Prueba3]$ cat archivo_de_entrada.txt
File: archivo_de_entrada.txt
1 1235678y6rtegfdbn!#"$$%&/()=(/%%$#WEDFGHJKO)(/%%REFDGBNMKLÑ~*~?~"=P)OIUY&%%$#"^!"#
2 ñéasd'o'``'iuyúíáźxcvbnmhgfrewź'dź'q'w'é'f'ú'b'ń'ń' ,.,.{{{
3
4 1 + 2
5 3 / 5
6 5 % 6
7 7 * 8
8 9 = 0;
9
10 .-!"#$%&/ $
11 3?(7)=1
12 para devo
13 en mientras 1 = 1 {
14 435674
15 }
16 parar
17
[bante@pc-bd-ace Prueba3]$
```

```
[bante@pc-bd-ace Prueba3]$ ./lexer archivo_de_entrada.txt
(7,0) 1235678
Error léxico: Token no reconocido: y
(6,1) 6r
Error léxico: Token no reconocido: t
(4,2) e
Error léxico: Token no reconocido: g
Error léxico: Token no reconocido: f
Error léxico: Token no reconocido: d
Error léxico: Token no reconocido: b
Error léxico: Token no reconocido: n
Error léxico: Token no reconocido: !
(1,4) #
Error léxico: Token no reconocido: "
Error léxico: Token no reconocido: $
Error léxico: Token no reconocido: %
Error léxico: Token no reconocido: &
Error léxico: Token no reconocido: /
Error léxico: Token no reconocido: (
Error léxico: Token no reconocido: )
Error léxico: Token no reconocido: =
Error léxico: Token no reconocido: (
Error léxico: Token no reconocido: /
Error léxico: Token no reconocido: &
Error léxico: Token no reconocido: %
Error léxico: Token no reconocido: $
(1,4) #
Error léxico: Token no reconocido: W
Error léxico: Token no reconocido: E
Error léxico: Token no reconocido: D
Error léxico: Token no reconocido: F
Error léxico: Token no reconocido: G
Error léxico: Token no reconocido: H
Error léxico: Token no reconocido: J
Error léxico: Token no reconocido: K
Error léxico: Token no reconocido: O
Error léxico: Token no reconocido: )
Error léxico: Token no reconocido: (
Error léxico: Token no reconocido: /
Error léxico: Token no reconocido: &
Error léxico: Token no reconocido: %
Error léxico: Token no reconocido: R
Error léxico: Token no reconocido: E
Error léxico: Token no reconocido: F
Error léxico: Token no reconocido: D
Error léxico: Token no reconocido: G
```

```
Error léxico: Token no reconocido: )
Error léxico: Token no reconocido: =
(7,2) 1
Error léxico: Token no reconocido: p
Error léxico: Token no reconocido: a
Error léxico: Token no reconocido: r
Error léxico: Token no reconocido: a
Error léxico: Token no reconocido: d
(4,2) e
Error léxico: Token no reconocido: v
Error léxico: Token no reconocido: o
(4,2) e
Error léxico: Token no reconocido: n
(4,1) m
Error léxico: Token no reconocido: i
(4,2) e
Error léxico: Token no reconocido: n
Error léxico: Token no reconocido: t
Error léxico: Token no reconocido: r
Error léxico: Token no reconocido: a
Error léxico: Token no reconocido: s
(7,2) 1
Error léxico: Token no reconocido: =
(7,2) 1
Error léxico: Token no reconocido: {
(7,11) 435674
Error léxico: Token no reconocido: }
Error léxico: Token no reconocido: p
Error léxico: Token no reconocido: a
Error léxico: Token no reconocido: r
Error léxico: Token no reconocido: a
Error léxico: Token no reconocido: r
```

Tabla de Símbolos:

Tabla de Literales Numéricos:

Posición: 0, Literal: 1235678

Posición: 1, Literal: 6r

Posición: 2, Literal: 1

Posición: 3, Literal: 2

Posición: 4, Literal: 3

Posición: 5, Literal: 5

Posición: 6, Literal: 6

Posición: 7, Literal: 7

Posición: 8, Literal: 8

Posición: 9, Literal: 9

Posición: 10, Literal: 0

Posición: 11, Literal: 435674

Tabla de Literales de Cadenas:

[bante@pc-bd-ace Prueba3]\$ █

Otro ejemplo:

Con el archivo “PruebaConcreta.txt”

```
[bante@pc-bd-ace Prueba3]$ cat PruebaConcreta.txt
File: PruebaConcreta.txt
1  -- Este es un archivo de prueba para el analizador léxico --
2
3  -- Identificadores --
4  @miVariable
5  @otraVariable
6  @variable123
7  @áéíóúñ
8
9  -- Palabras reservadas --
10 Bool
11 Cade
12 Continuar
13 Devo
14 Ent
15 Fals
16 Flota
17 Global
18 Haz
19 Mientras
20 Nulo
21 Para
22 Parar
23 Si
24 Sino
25 Ver
26
27 -- Operadores aritméticos --
28 sum
29 rest
30 mult
31 div
32 mod
33 inc
34 dec
35 exp
36 dive
37
38 -- Operadores relacionales --
39 h
40 m
41 e
42 c
43 he
44 me
45
```

```

Error léxico: Token no reconocido:
(1,1) >
Error léxico: Token no reconocido:
(1,2) <<
Error léxico: Token no reconocido:
(1,3) >>
Error léxico: Token no reconocido:
(1,4) #
Error léxico: Token no reconocido:
(1,5) #!
Error léxico: Token no reconocido:
(1,6) *
Error léxico: Token no reconocido:
(1,7) |
Error léxico: Token no reconocido:
(1,8) °
Error léxico: Token no reconocido:
Error léxico: Token no reconocido:
(6,11) -- Comentarios --
Error léxico: Token no reconocido:
Comentario ignorado: [Este es un comentario de una línea]
Error léxico: Token no reconocido:
Comentario ignorado: [Este es otro comentario
que abarca varias líneas]
Error léxico: Token no reconocido:

```

Tabla de Símbolos:

Tabla de Literales Numéricos:

```

Posición: 0, Literal: 123
Posición: 1, Literal: 45
Posición: 2, Literal: 67
Posición: 3, Literal: -89
Posición: 4, Literal: +3
Posición: 5, Literal: 14
Posición: 6, Literal: 0

```

Tabla de Literales de Cadenas:

```

Posición: 0, Cadena: -- Este es un archivo de prueba para el analizador léxico --
Posición: 1, Cadena: -- Identificadores --
Posición: 2, Cadena: -- Palabras reservadas --
Posición: 3, Cadena: -- Operadores aritméticos --
Posición: 4, Cadena: -- Operadores relacionales --
Posición: 5, Cadena: -- Operadores de asignación --
Posición: 6, Cadena: -- Literales numéricos --
Posición: 7, Cadena: -- Literales de cadena --
Posición: 8, Cadena: --Hola, mundo--
Posición: 9, Cadena: --Este es un literal de cadena--
Posición: 10, Cadena: -- Símbolos especiales --
Posición: 11, Cadena: -- Comentarios --

```

```
[bante@pc-bd-ace Prueba3]$ █
```

Como podemos observar, las pruebas muestran los token válidos dependiendo de en qué regla recaen, las tablas de símbolos, las tablas de literales numéricas y literales de cadenas, cumpliendo con los requerimientos propuestos en su mayoría.

Conclusiones:

Arellanes Conde Esteban

El desarrollo de este proyecto nos permitió profundizar en el funcionamiento de un analizador léxico y en la importancia de una correcta definición de expresiones regulares para la identificación de los diferentes componentes del lenguaje. A lo largo del proceso, enfrentamos varios desafíos, particularmente en la correcta clasificación de identificadores y en la estructuración de las tablas de símbolos y literales. Sin embargo, a través de la investigación y la aplicación de metodologías iterativas de prueba y error, logramos obtener una implementación funcional.

La experiencia adquirida con Lex/Flex nos ayudó a comprender mejor cómo se lleva a cabo el análisis léxico dentro de un compilador, así como la necesidad de una correcta gestión de los tokens y su almacenamiento eficiente. Además, la integración con estructuras de datos adecuadas facilitó la organización de la información extraída del código fuente.

Si bien hubo dificultades iniciales debido a la falta de experiencia con esta herramienta, logramos superarlas y mejorar nuestra comprensión sobre los principios fundamentales del análisis léxico. Finalmente, este proyecto sentó las bases para futuros desarrollos en la construcción de compiladores y herramientas relacionadas con el procesamiento de lenguajes de programación.

Axel Gael Méndez Galicia

En el desarrollo de este proyecto, logramos implementar un analizador léxico utilizando Flex, con el objetivo de identificar y clasificar distintos componentes de un código fuente, como identificadores, números, cadenas de texto, operadores y palabras reservadas. A lo largo del proceso, enfrentamos varios desafíos, particularmente en la identificación de ciertos tokens, la correcta estructuración de las expresiones regulares y el manejo de errores léxicos. Sin embargo, gracias a la investigación y a la metodología de prueba y error, logramos superar estas dificultades y ajustar nuestro programa hasta obtener un funcionamiento óptimo.

Uno de los aspectos más valiosos de este proyecto fue la oportunidad de comprender en profundidad el funcionamiento de un analizador léxico, así como su papel dentro del proceso de compilación. A pesar de la falta inicial de conocimiento sobre Flex, conseguimos aprender lo necesario para desarrollar la herramienta y mejorar nuestra comprensión del análisis léxico en general. Además, la construcción de tablas de símbolos y literales nos permitió organizar de manera eficiente la información extraída del código fuente, facilitando su uso en etapas posteriores del análisis.

Este proyecto no solo nos permitió alcanzar el objetivo principal de desarrollar un analizador léxico funcional, sino que también nos ayudó a reforzar conceptos clave sobre la estructura de los compiladores y el procesamiento de lenguajes de programación. A pesar de los desafíos encontrados, logramos cumplir con los requerimientos y sentamos una base sólida para futuros proyectos relacionados con la compilación e interpretación de código.

Bibliografía:

1. 2.1. *Función del Analizador Léxico*. (s. f.).
http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/autocontenido/autocon/21_funcion_del_analizador_lexico.html
2. *Compilador Diseño - Análisis Léxico*. (s. f.).
https://www.tutorialspoint.com/es/compiler_design/compiler_design_lexical_analysis.htm
3. *LICAD - Laboratorio de Intel y Cómputo de Alto Desempeño. - compiladores G04 20251*. (s. f.).
http://telematica1.fi-b.unam.mx/classroom/main/lp/lp_controller.php?cidReq=COMP20251G4&id_session=0&gradebook=0&origin=&gidReq=0 (Apuntes de Clase).

Aspectos a Evaluar	Niveles de Desempeño					Valor Obtenido
	Autónomo 10	Destacado 8	Satisfactorio 6	Insatisfactorio 4	NA 0	
El Analizador reconoce todas las clases de componentes léxicos [15%]	El Analizador reconoce las 9 clases de componentes léxicos.	El Analizador reconoce bien de 7 a 8 clases de componentes léxicos.	El Analizador reconoce bien de 5 a 6 clases de componentes léxicos.	El Analizador reconoce bien de 3 a 4 clases de componentes léxicos.	El Analizador reconoce menos de 3 clases de componentes léxicos.	
El Analizador maneja bien las tablas de identificadores y literales, y genera bien los tokens [25%]	El Analizador crea correctamente la estructura de las tablas y las actualiza como debe ser: no repite identificadores en la TS e incluye a todas las cadenas y constantes flotantes en su correspondiente tabla de literales. Genera bien los tokens de las 10 clases de componentes léxicos	El Analizador crea correctamente la estructura de las tablas y las actualiza de acuerdo a las clases que sí reconoce al igual que sus tokens	El Analizador no crea totalmente bien la estructura de las tablas o no las actualiza de acuerdo a las clases que sí reconoce o no genera bien sus tokens	El Analizador no crea totalmente bien la estructura de las tablas o no las actualiza de acuerdo a las clases que sí reconoce y tiene fallas en la generación de tokens	El analizador no maneja las tablas de identificadores y/o de literales y crea los tokens no adecuadamente o faltan.	
Compilación y ejecución del programa Analizador Léxico [20%]	El programa se compila y ejecuta sin errores. Toma el archivo de entrada desde la línea de comandos.	El programa compila sin errores, pero marca errores de ejecución con ciertos datos de entrada o no toma el archivo desde la línea de comandos.	El programa marca errores de compilación que se pueden corregir y se puede ejecutar.	El programa marca errores de compilación que se pueden corregir pero no se ejecuta bien.	El programa marca errores de compilación que no se pueden corregir.	
El programa está bien comentado [15%]	El programa tiene comentarios al inicio con el objetivo, nombre de los que lo elaboraron y en cada función importante.	El programa tiene comentarios al inicio con el objetivo y nombre de los que lo elaboraron, pero no en todas las funciones importantes.	Al programa le faltan comentarios al inicio con el objetivo o el nombre de los que lo elaboraron, o en la mayoría de las funciones importantes.	El programa tiene muy pocos comentarios.	El programa no contiene comentarios.	
El documento que describe cómo se elaboró el programa [25%] *	Tiene la estructura y contenido solicitado, además de lo indicado en *	El documento tiene 80% de lo solicitado, incluyendo *	El documento tiene 60% de lo solicitado, incluyendo *	El documento tiene 40% de lo solicitado, incluyendo *	No entregó documento	
Total						0

* Documenta las decisiones, roles o la distribución de tareas del trabajo en equipo o individualmente en el tiempo.
Organiza el trabajo utilizando herramientas y recursos tecnológicos. Emplea la comunicación efectiva.