



Universidad Nacional Autónoma de México

Facultad de Ingeniería



Compiladores

Proyecto 1

Integrantes:

Díaz González Rivas Ángel Iñaqui
Gayosso Rosillo Sebastian Emiliano
Perez Delgado Erandy Estefanya
Siliano Haller Rodrigo

Profesora:

Sáenz García Elba Karen

Grupo: 1

Semestre: 2025-1

Fecha de entrega: 20/Septiembre/2024

Descripción del problema

Elaborar un analizador léxico en lex/flex que reconozca los componentes léxicos pertenecientes a las siguientes clases.

Clase	Descripción
0	Palabras reservadas (ver tabla)
1	Operadores aritméticos + - * / %
2	Operadores de asignación (ver tabla)
3	Símbolos especiales () { } [] & , :
4	Operadores relacionales (ver tabla)
5	Identificadores. Inicien con letra (mayúscula o minúscula), le sigan letras o dígitos (hasta cinco) y terminen con guion bajo.
6	Constantes numéricas enteras (base 10, sin sufijos) signados o no signados, máximo 6 dígitos, mínimo 1. Para no confundir el signo con un operador aritmético se dejará o no un espacio en los siguientes casos: -56+ +46 56- 41 72+a_
7	Constantes numéricas reales. No signados. Casos: 2.67 .85 31. Al definir la expresión regular, no debe incluir la cadena. como constante real.
8	Constantes cadenas. Iniciar y terminar con comillas dobles, cualquier carácter excepto las comillas dobles. Tamaño entre 2 y 40 (considerando las comillas).
9	Constante carácter. Cualquier carácter encerrado entre comillas sencillas (apóstrofes).

Valor	Palabra reservada
0	cadena
1	caracter
2	else
3	entero
4	For
5	if
6	real
7	return
8	void
9	while

Valor	Op. relacional	significado
0	^^	mayor que
1	^"	menor que
2	==	igual que
3	^^=	mayor o igual
4	^"=	menor o igual
5	<>	diferente

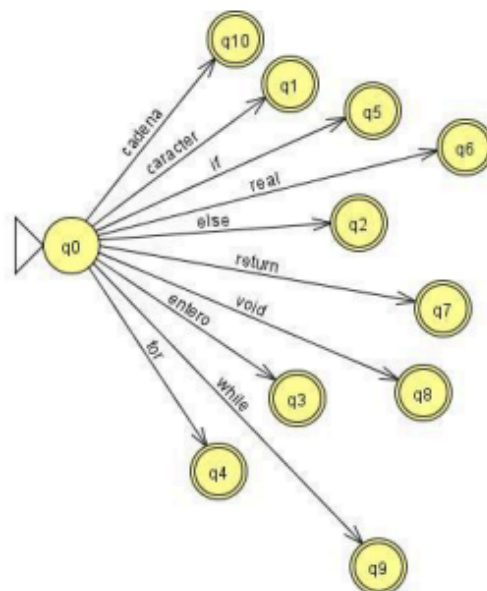
Valor	Op. asignación	significado
0	~	igual
1	+~	más igual
2	-~	menos igual
3	*~	por igual
4	/~	entre igual
5	\$~	módulo igual

Planteamiento

Para el planteamiento del analizador léxico nos basamos en las actividades anteriores donde se mostraron las expresiones regulares que Flex puede leer para los diferentes casos donde tiene que identificar entre operadores relacionales, de asignación, aritméticos, cadenas, caracteres, palabras reservadas o constantes numéricas. Dichas expresiones regulares junto con sus respectivos autómatas son los siguientes:

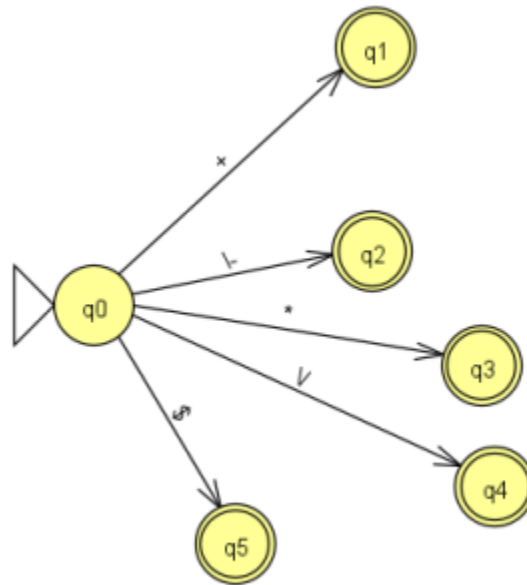
1.- Expresión regular para palabras reservadas en

Flex: "cadena"|"caracter"|"else"|"entero"|"for"|"if"|"real"|"return"|"void"|"while"



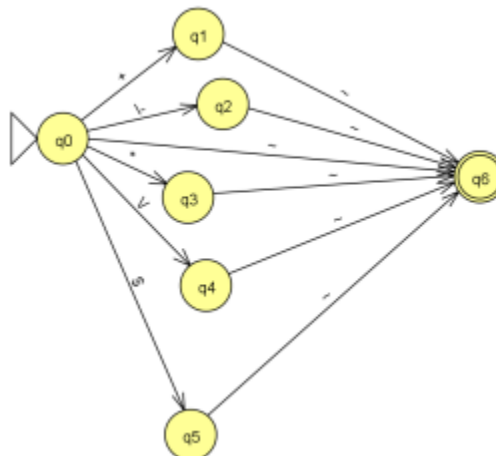
Autómata que representa la expresión regular de las palabras reservadas

2.- Expresión regular para los operadores aritméticos (+, -, *, /, \$) en Flex: `[+ \- * \ / $]`



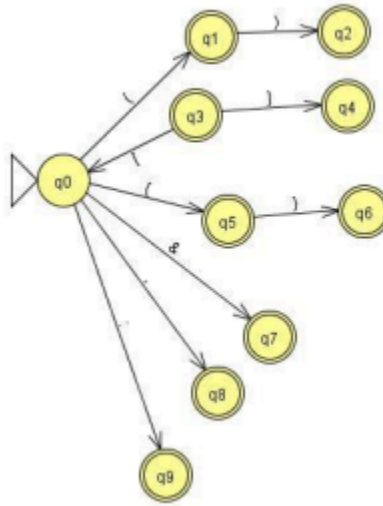
Autómata que representa la expresión regular de los operadores aritméticos

3.- Expresión regular para los operadores de asignación (\sim , $+\sim$, $-\sim$, $*\sim$, $/\sim$, $\$\sim$) en Flex: `[+ \- * \ / \$]?~`



Autómata que representa la expresión regular de los operadores de asignación

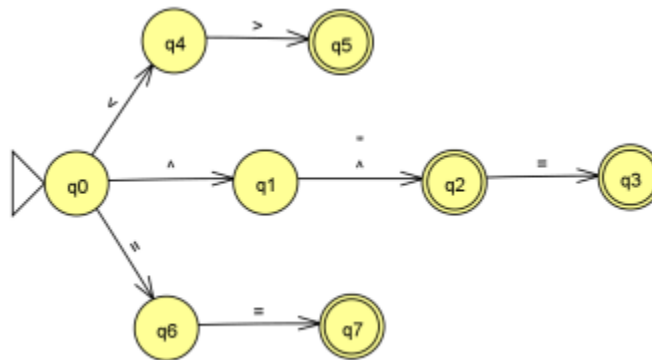
"(")"|{"|}"|["|"]|",|":|'|()|'|{|}|[]|



Autómata que representa la expresión regular de los símbolos especiales

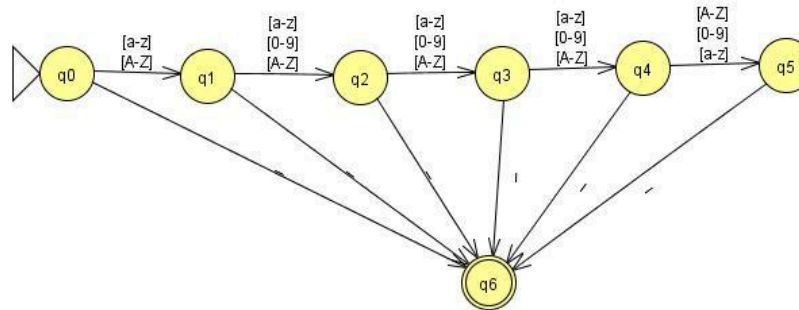
5.- Expresión regular de los operadores relacionales (^, ^", ==, ^=, ^=, <>) en Flex:

```
^(^(\^|\")=?|=|<>)$
```



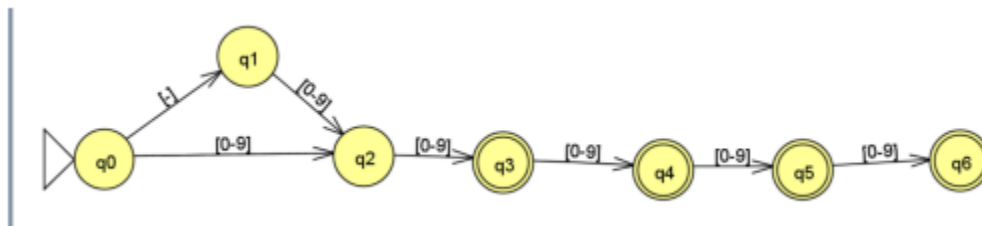
Autómata que representa la expresión regular de los operadores relacionales

6.- Expresión regular de los identificadores con la condición de que inicien con letra, sigan letras o dígitos, hasta cinco y terminan en guión bajo en Flex: $[a-zA-Z][a-zA-Z0-9]\{0,4\}_$



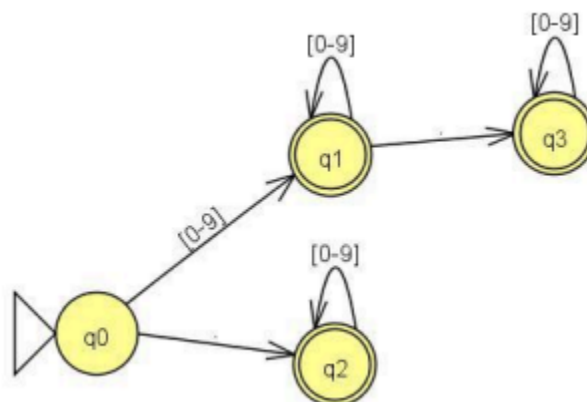
Autómata que representa la expresión regular de los identificadores

7.- Expresión regular que representa las constantes numéricas enteras en base 10 con la condición de que solo sean 6 dígitos en Flex: $[-]?[0-9]\{1,6\}$



Autómata que representa la expresión regular de las constantes enteras en base 10

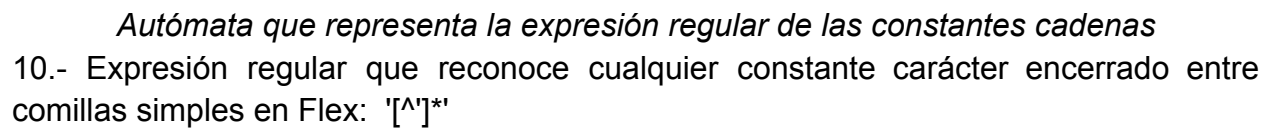
8.- Expresión regular que representa las constantes reales no signados en Flex: $[0-9]+\backslash.[0-9]*\backslash.[0-9]+$



autómata que representa la expresión regular de las constantes enteras no signadas

```

graph LR
    start(( )) --> q0((q0))
    q0 -- a --> q1((q1))
    q1 -- b --> q4((q4))
    q4 -- b --> q2((q2))
    q1 -- λ --> q2
    q2 -- λ --> q1
    q2 -- a --> q3(((q3)))
  
```



Autómata que conjunta todos los autómatas de las expresiones regulares, con el cual se busca definir el comportamiento del analizador léxico.

Análisis (Planificación)

Proyecto 1

Semana 1

Semana 2

Semana 3

Semana 4

Semana 5

	26-08/01-09	02-09/08-09	09-09/15-09	16-09/22-09	23-09/29-09
	L M X J V S D	L M X J V S D	L M X J V S D	L M X J V S D	L M X J V S D
DEFINIR REQUISITOS.					
DEFINIR LA DESCRIPCIÓN DEL PROBLEMA					
DEFINIR EXPRESIONES REGULARES.					
CONSTRUCCIÓN DE LOS AFD (AUTOMATAS FINITOS DETERMINISTAS).					
ELABORACIÓN DE DIAGRAMAS EN JFLAP					
INTEGRACIÓN DEL AFD FINAL.					
INTEGRACIÓN Y DESARROLLO DEL ANALIZADOR LÉXICO.					
DESCRIPCIÓN DEL ANALIZADOR LÉXICO.					
MANEJO DE ERRORES LÉXICOS.					
PRUEBAS Y DEPURACIÓN.					
ELABORACIÓN DE LAS INDICACIONES PARA CORRER EL PROGRAMA.					
DOCUMENTACIÓN DEL PROYECTO.					
CONCLUSIONES INDIVIDUALES Y REFERENCIAS BIBLIOGRÁFICAS.					
ENTREGA FINAL.					

Planeación	Descripción	Participación de los integrantes
Definir Requisitos.	Revisar el enunciado del proyecto.	Todos los integrantes: Determinar los requisitos específicos.
Definir la Descripción del Problema.	Redactar la descripción clara y completa del problema a resolver.	Todos los integrantes: Descripción general del problema.
Definir Expresiones Regulares.	Definir las expresiones regulares que permitirán al analizador léxico identificar los componentes léxicos	Díaz Ángel: Definió 2 expresiones regulares. Gayosso Sebastián: Definió 2

	del lenguaje.	expresiones regulares. Pérez Erandy: Definió 2 expresiones regulares. Siliano Rodrigo: Definió 3 expresiones regulares.
Construcción de los AFD (Autómatas Finitos Deterministas).	Crear los AFD para reconocer los componentes léxicos basados en las expresiones regulares definidas.	Díaz Ángel: Construyó 2 AFD y realizó pruebas de simulación. Gayosso Sebastián: Construyó 2 AFD y utilizó Flex para la generación automática. Pérez Erandy: Construyó 2 AFD Siliano Rodrigo: Construyó 3 AFD
Elaboración de diagramas en JFLAP	Creación de diagramas en el software JFLAP para representar visualmente cómo se estructura cada expresión regular y cómo se relaciona con los componentes léxicos del lenguaje.	Díaz Ángel: Elaboró 2 diagramas. Gayosso Sebastián: Elaboró 2 diagramas. Pérez Erandy: Elaboró 2 diagramas. Siliano Rodrigo: Elaboró 3 diagramas.
Integración del AFD Final.	El resultado de la integración será un AFD optimizado que permita una lectura precisa del código fuente, generando los tokens correspondientes para cada componente léxico identificado.	Díaz Ángel: Participó en la integración del AFD Gayosso Sebastián: Participó en la integración del AFD Pérez Erandy: Participó en la integración del AFD Siliano Rodrigo: Participó en la integración del AFD

Integración y Desarrollo del Analizador Léxico.	Se desarrolla el código del analizador léxico utilizando las expresiones regulares definidas previamente y las herramientas de Lex/Flex.	Gayosso Sebastián: Contribuyó al desarrollo del código del analizador léxico utilizando Lex/Flex y las expresiones regulares definidas.
Descripción del analizador léxico.	Se documenta cada sección del código del analizador léxico para facilitar su comprensión.	Todos los integrantes: Todos los integrantes colaboraron en la revisión y mejora de la documentación, asegurando que cada sección esté claramente explicada para facilitar la comprensión.
Manejo de Errores Léxicos	Los errores serán mostrados en pantalla o almacenados en un archivo para su posterior revisión.	<p>Díaz Ángel: Implementó la visualización de errores en pantalla.</p> <p>Gayosso Sebastián: Implementó el almacenamiento de errores en un archivo para su revisión posterior.</p> <p>Perez Erandy: Realizó pruebas y validaciones para asegurar que los errores se manejen correctamente.</p> <p>Siliano Rodrigo: Desarrolló mecanismos para la detección y reporte de errores léxicos.</p>
Pruebas y Depuración.	Se realizarán pruebas exhaustivas para verificar el correcto funcionamiento del analizador léxico, asegurando que todos los componentes léxicos sean reconocidos y procesados de manera adecuada.	<p>Díaz Ángel: Realizó pruebas exhaustivas para verificar el reconocimiento de tokens y el procesamiento de componentes léxicos.</p> <p>Siliano Rodrigo: Validó los resultados de las pruebas y ajustó el código para optimizar el rendimiento del analizador</p>

		léxico.
Elaboración de las indicaciones para correr el programa.	Se proporcionarán instrucciones detalladas para compilar y ejecutar el analizador léxico desde la línea de comandos, indicando el archivo de programa fuente y los formatos de entrada y salida esperados, así como la interpretación de los tokens generados.	<p>Díaz Ángel: Desarrolló las instrucciones para ejecutar el analizador léxico, incluyendo los formatos de entrada y salida esperados.</p> <p>Siliano Rodrigo: Revisó y ajustó las indicaciones finales, asegurando que la documentación cubriera todos los aspectos necesarios para ejecutar el programa correctamente.</p>
Documentación del proyecto	Descripción completa del problema, el diseño del analizador, las estructuras de datos, las pruebas realizadas, y explicaciones del código, facilitando la comprensión y extensión del proyecto.	<p>Pérez Erandy: Redactó la descripción completa del problema y el diseño del analizador léxico. Documentó las estructuras de datos utilizadas y las pruebas realizadas.</p> <p>Gayosso Sebastián: Elaboró explicaciones detalladas del código y la implementación, facilitando la comprensión y extensión del proyecto. Colaboró en la integración de toda la documentación.</p>
Conclusiones individuales y referencias bibliográficas.	Cada miembro del equipo elabora sus propias conclusiones sobre el proyecto y agrega sus referencias bibliográficas.	Todos los integrantes: Se elaboraron las conclusiones individuales, además de referencias bibliográficas, asegurando una cobertura completa de las fuentes

Entrega Final	Compilar y presentar la documentación completa del proyecto, junto con el analizador léxico funcional.	Gayosso Sebastián: Enviará la documentación final y el programa fuente definitivo.
---------------	--	---

Diseño e implementación del analizador léxico en Flex

Tablas

La generación de tokens se hará en forma de dupla (x,y), donde el valor 'x' representa la clase de entrada, 'y' representa el valor que toma de la tabla de especificaciones.

Tablas de especificaciones:

Clase	Valor	Palabra Reservada
0	0	cadena
0	1	caracter
0	2	else
0	3	entero
0	4	For
0	5	if
0	6	real
0	7	return
0	8	void
0	9	while

Tabla de especificaciones de las palabras reservadas

Ejemplos de generación de Token:

cadena: (0,0)

else: (0,2)

void: (0,8)

while: (0,9)

Clase	Valor
1	+
1	-
1	*
1	/
1	\$

Tabla de especificaciones de los operadores aritméticos

Ejemplos de generación de Token:

+: (1,+)

-: (1,-)

/: (1,/)

\$. (1,\$)

Clase	Valor
2	~
2	+~
2	-~
2	*~
2	/~
2	\$~

Tabla de especificaciones de los operadores de asignación

Ejemplos de generación de Token:

~: (2,~)

+~: (2,+~)

/~: (2,/~)

\$~: (2,\$~)

Clase	Valor
3	(
3)
3	{
3	}
3	[
3]
3	&
3	,
3	:

Tabla de especificaciones de los símbolos especiales

Ejemplos de generación de Token:

&: (3, &)

,: (3, ,)

(: (3, ()

[: (3, [)

Clase	Valor
4	^^
4	^"
4	==
4	^^=
4	^"=
4	<>

Tabla de especificaciones de los operadores relacionales

Ejemplos de generación de Token:

^^: (4, ^^)

^": (4, ^")

==: (4, ==)

<>: (4, <>)

Clase	Orden de aparición
5	n=0
5	n=1
5	n=2
5	.
5	.
5	.
5	n= n+1

Tabla de especificaciones de los identificadores

Ejemplos de generación de Token:

Pi_: (5,0)

Hola_: (5,1)

Arroba_: (5,2)

C123_: (5,3)

cClase	Orden de aparición
6	n=0
6	n=1
6	n=2
6	.

	.
6	.
	.
6	n = n+1

Tabla de especificaciones de las constantes enteras con o sin signo

Ejemplos de generación de Token:

23441: (6,0)

-97: (6,1)

6969: (6,2)

-1: (6,3)

Clase	Orden de aparición
7	n=0
7	n=1
7	n=2
7	.
7	.
7	.
7	.
7	.
7	n=n+1

Tabla de especificaciones de las constantes reales sin signo

Ejemplos de generación de Token:

72.1: (7,0)

3.141592: (7,1)

2.35. (7,2)

0.23: (7,3)

9	$n = n + 1$
---	-------------

Tabla de especificaciones de las constantes caracter

Ejemplos de generación de Token:

'a': (9,0)

'abc': (9,1)

'zzzzzz': (9,2)

'xdxdxd': (9,3)

Números enteros, reales, identificadores, constantes cadenas y caracteres son catalogados como literales, por lo cual comparten el mismo contador en la orden de aparición, es decir, además de generar su propio Token con el número de clase, se acomodan en una tabla de literales conforme el analizador léxico detecta su aparición.

Ejemplo:

Variable1_

'z'

"esto es un ejemplo"

65 + 2

38.21

Los Tokens se generarían de la siguiente forma:

Variable1_: (5,0)

'z': (9,1)

"esto es un ejemplo": (8,2)

65 + 2: (6, 3), (6,4)

38.21: (7,5)

La tabla de literales quedaría así:

0.- Variable1_

1.- 'z': (9,1)

2.- "esto es un ejemplo"

3.- 65 + 4.- 2

5.- 38.21

Estructura de datos y algoritmos de búsqueda

Se hizo uso de las bibliotecas C stdlib.h y string.h para la optimización y el manejo de cadenas, respectivamente.

Con estas bibliotecas preestablecidas, iniciamos el desarrollo de las estructuras de almacenamiento para los símbolos y literales.

```

typedef struct{
    char nombre[50];
} Simbolo;

typedef struct {
    int posicion;
    char dato[50];
} Literal;

Simbolo tablaSimbolos[100];
Literal tablaLiterales[100];
int indexSimbolos = 0;
int indexLiterales = 0;

```

Estos arreglos nos permitirán almacenar los símbolos y caracteres que el analizador léxico encuentre. Cada arreglo permite guardar hasta 100 literales y 100 símbolos.

El programa generará 2 archivos de texto donde se indicarán los tokens formados y los errores léxicos encontrados por el analizador.

```

FILE *tokensFile;
FILE *erroresFile;

```

En el código se utiliza un algoritmo de búsqueda lineal para encontrar símbolos en la tabla. La búsqueda se implementa mediante un ciclo For en la función “buscarSimbolo”. Esta función recorre la tabla de símbolos y compara el nombre de cada símbolo almacenado con el nombre proporcionado. Si se encuentra una coincidencia, la función devuelve la posición del símbolo dentro del arreglo. En caso de que el símbolo no exista en la tabla, la función devolverá -1, indicando que no fue encontrado.

Para agregar símbolos se ha implementado la función “agregarSimbolo”, la cual verificará que se agreguen nuevos símbolos y evitar duplicados, además de que actualizará el índice de símbolos.

```

int buscarSimbolo(const char *nombre) {
    for (int i = 0; i < indexSimbolos; i++) {
        if (strcmp(tablaSimbolos[i].nombre, nombre) == 0) {
            return i;
        }
    }
    return -1;
}

void agregarSimbolo(const char *nombre) {
    if (buscarSimbolo(nombre) == -1) {
        strcpy(tablaSimbolos[indexSimbolos].nombre, nombre);
        indexSimbolos++;
    }
}

```

Si el identificador no se encuentra en `tablaSimbolos` (según lo determine la función `buscarSimbolo`), entonces el identificador se inserta en el arreglo. La función `agregarSimbolo` realiza esta tarea. Primero, verifica si el identificador ya existe llamando a `buscarSimbolo`. Si no existe, copia el nombre del identificador en la posición actual de `tablaSimbolos` y luego incrementa el índice `indexSimbolos` para apuntar a la próxima posición disponible.

Las cadenas y constantes numéricas se almacenan en otro arreglo llamado `tablaLiterales`, que es un arreglo de estructuras `Literal`. Este arreglo contiene tanto la posición de cada literal como su dato (cadena o constante numérica).

Inserción de literales (cadenas y números):

Cuando se encuentra una constante numérica o una cadena durante el análisis léxico, se inserta directamente en la siguiente posición disponible en `tablaLiterales`. La función `agregarLiteral` realiza la inserción. Esta función copia el valor del literal en la posición actual y también asigna la posición de este literal. Luego, incrementa el índice `indexLiterales` para apuntar a la siguiente posición disponible.

```

void agregarLiteral(const char *dato) {
    strcpy(tablaLiterales[indexLiterales].dato, dato);
    tablaLiterales[indexLiterales].posicion = indexLiterales;
    indexLiterales++;
}

```

Además, se requieren las siguientes funciones para manejar los tokens y errores léxicos:

`guardarToken`: Guarda los tokens generados por el análisis léxico en un archivo. Esto es útil para registrar tanto los identificadores como las constantes numéricas y literales.

guardarErrorLexico: Guarda los errores léxicos encontrados durante el análisis en un archivo de errores. Esta función es llamada cuando el analizador encuentra un token no válido o fuera de las reglas definidas.

```
void guardarErrorLexico(const char *error) {
    if (erroresFile == NULL) {
        erroresFile = fopen("errores_lexicos.txt", "a");
    }
    if (erroresFile != NULL) {
        fprintf(erroresFile, "Error léxico: %s\n", error);
    }
}

void guardarToken(int clase, const char *valor) {
    if (tokensFile == NULL) {
        tokensFile = fopen("tokens.txt", "a");
    }
    if (tokensFile != NULL) {
        fprintf(tokensFile, "Token (%d, %s)\n", clase, valor);
    }
}
```

Ya teniendo establecidas las funciones que se requerirán para la búsqueda y almacenamiento de símbolos, ahora se definen las reglas léxicas que utilizará el analizador.

Empezamos por crear una expresión regular que va a ignorar cualquier sentencia de escape como lo son las tabulaciones y espacios.

Ya que se definió la manera en que se leerán los símbolos, ahora se tiene que definir las palabras reservadas, cuyo token se forma con la clase '0' y su valor establecido en la tabla.

```
[ \t\n]+ {}

"cadena"    { guardarToken(0, "0"); }
"caracter"  { guardarToken(0, "1"); }
"else"      { guardarToken(0, "2"); }
"entero"    { guardarToken(0, "3"); }
"For"       { guardarToken(0, "4"); }
"if"        { guardarToken(0, "5"); }
"real"      { guardarToken(0, "6"); }
"return"    { guardarToken(0, "7"); }
"void"      { guardarToken(0, "8"); }
"while"     { guardarToken(0, "9"); }
```

Conjunto de palabras reservadas que detectará el analizador léxico y generará su respectivo token

Seguimos definiendo las expresiones regulares de los operadores aritméticos, operadores de asignación, operadores relacionales, símbolos especiales e identificadores.

```
[\\+\\-\\*\\/\\$] { guardarToken(1, yytext); }

"~" | \\+~ | \\-~ | \\*~ | \\/~ | \\$~ { guardarToken(2, yytext); }

"<>" | "==" | "<=" | ">=" | "!=" | "^" { guardarToken(4, yytext); }

"(" | ")" | "{" | "}" | "[" | "]" | "," | ":" { guardarToken(3, yytext); }

[a-zA-Z][a-zA-Z0-9]{0,4}_ {
    agregarSimbolo(yytext);
    int pos = buscarSimbolo(yytext);
    char posStr[10];
    sprintf(posStr, "%d", pos);
    guardarToken(5, posStr);
}
```

Definición de expresiones regulares para que el analizador léxico detecte todas las literales

Se creó también una expresión para que el analizador detecte un error léxico. Este error se produce cuando la sentencia a leer no cumple con ninguna de las reglas establecidas.

```
[a-zA-Z][0-9]+ {
    guardarErrorLexico(yytext);
}
```

```
[a-zA-Z0-9_]{5,} {
    guardarErrorLexico(yytext);
}

. {
    guardarErrorLexico(yytext);
}
```

Expresión regular que detecta sentencias que no cumplen con las reglas

Las últimas expresiones regulares en definirse son las de los números enteros, los números reales, las constantes cadenas y las constantes carácter.

```
-?[0-9]{1,6} {
    agregarLiteral(yytext);
    char posStr[10];
    sprintf(posStr, "%d", indexLiterales - 1);
    guardarToken(6, posStr);
}

[0-9]+\.[0-9]*|\.[0-9]+ {
    agregarLiteral(yytext);
    char posStr[10];
    sprintf(posStr, "%d", indexLiterales - 1);
    guardarToken(7, posStr);
}
```

```
\"([^\"]|\\"){2,40}\" {
    agregarLiteralCadena(yytext); // Agrega el literal cadena a la tabla
    char posStr[10];              // Buffer para la posición del literal
    sprintf(posStr, "%d", indexLiteralesCadenas - 1); // Formatea la posición
    guardarToken(8, posStr);      // Guarda el token de tipo 8 (cadena)
}

'[^']*' {
    agregarLiteralCadena(yytext); // Agrega el literal de carácter a la tabla
    char posStr[10];              // Buffer para la posición del literal
    sprintf(posStr, "%d", indexLiteralesCadenas - 1); // Formatea la posición
    guardarToken(9, posStr);      // Guarda el token de tipo 9 (carácter)
}
```

Expresiones regulares de las últimas literales

Por último, en la función principal del programa se establecen las acciones del programa. Este funciona con un archivo de texto al cual se le desea analizar.

Ya que se tiene el archivo, el programa en flex leerá sentencia por sentencia para identificar los símbolos que cumplen con las reglas léxicas establecidas.

Una vez terminado el análisis, se crearán cuatro archivos de texto: tokens, errores_lexicos, tabla_simbolos y tabla_literales. Estos archivos nos mostrarán los tokens que se generaron, los errores que se tuvieron, los símbolos reconocidos y las literales que se encontraron, respectivamente.

```
int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Uso: %s <archivo de entrada>\n", argv[0]);
        return 1;
    }

    FILE *file = fopen(argv[1], "r");
    if (!file) {
        perror("Error al abrir el archivo");
        return 1;
    }

    yyin = file;
    tokensFile = fopen("tokens.txt", "w");
    erroresFile = fopen("errores_lexicos.txt", "w");
    yylex();
    fclose(file);

    guardarTablas();

    if (tokensFile != NULL) fclose(tokensFile);
    if (erroresFile != NULL) fclose(erroresFile);

    return 0;
}
```

Instrucciones que nos permiten el uso y creación de archivos.txt

Funcionamiento del programa

Para inicializar el programa, primero tenemos que crear un archivo de texto que queremos analizar.

prueba	15/09/2024 14:05	Documento de te...	1 KB
<pre>Pi_ 56 + 12; void ~ <> cadena caracter else entero 6996 3.1416 23.62 "prueba" 'aaaa' 'a'</pre>			

Archivo de texto que se desea analizar



Ya creado el texto a analizar, abrimos nuestro analizador de extensión '.l' desde la terminal.

```
C:\Users\Rodrigo\Documents\Facultad de Ingenieria\Compiladores>flex analizador.l
```

Inicialización del programa a través de la ruta donde se almacenó.

Una vez abierto el programa, se procede a compilar, lo que nos generará un archivo ejecutable.

```
C:\Users\Rodrigo\Documents\Facultad de Ingenieria\Compiladores>gcc lex.yy.c -o analizador
```





 lex.yy.c	15/09/2024 3:09	Archivo C	48 KB
 analizador	15/09/2024 3:09	Aplicación	54 KB

Creación del archivo '.c' y el ejecutable del programa

Por último, para analizar el texto deseado, ejecutamos el comando: 'analizador prueba.txt' en la terminal.

```
C:\Users\Rodrigo\Documents\Facultad de Ingenieria\Compiladores>analizador prueba.txt
```

Automáticamente se nos generarán los cuatro archivos.txt que son: tokens, errores_lexicos, tabla_simbolos y tabla_literales.

 tokens	15/09/2024 3:10	Documento de te...	1 KB
 errores_lexicos	15/09/2024 3:10	Documento de te...	1 KB
 tabla_simbolos	15/09/2024 3:10	Documento de te...	1 KB
 tabla_literales	15/09/2024 3:10	Documento de te...	1 KB

Archivos .txt generados por el programa

```
|Token (5, 0)
|Token (6, 0)
|Token (1, +)
|Token (6, 1)
|Token (0, 8)
|Token (2, ~)
|Token (4, <>)
|Token (0, 0)
|Token (0, 1)
|Token (0, 2)
|Token (0, 3)
|Token (6, 2)
|Token (7, 3)
|Token (7, 4)
|Token (8, 5)
|Token (8, 6)
|Token (9, 7)
|Token (9, 8)
```

Contenido del archivo tokens.txt

```
|Error léxico: ;
```

Contenido de errores_lexicos.txt

```
Tabla de Literales:
0      56
1      12
2      6996
3      3.1416
4      23.62
5      "prueba"
6      'aaaa'
7      'a'
```

Contenido del archivo tabla_literales.txt

```
Tabla de Simbolos:
0      Pi_
```

Contenido del archivo tabla_simbolos.txt

Para analizar cualquier otro archivo de texto, se sigue la siguiente sintaxis: analizador nombre_archivo.txt

Conclusiones:

Díaz González Rivas Ángel Iñiqui: El objetivo del proyecto fue desarrollar un analizador léxico utilizando Flex para reconocer los componentes léxicos de un lenguaje. Se definieron las expresiones regulares y los autómatas necesarios, apoyándose en los conceptos estudiados en clase. Luego, se crearon las tablas de símbolos y literales, diseñando una estructura de datos y un algoritmo de búsqueda para gestionar la inserción de los elementos. Finalmente, se implementó un código que, al analizar un archivo de texto, genera las tablas y tokens correspondientes, aplicando los conocimientos teóricos adquiridos.

Gayosso Rosillo Sebastian Emiliano: El proyecto cumplió sus objetivos al desarrollar un analizador léxico en lex/Flex capaz de reconocer diferentes componentes léxicos mediante expresiones regulares, y usar JFLAP para visualizar diagramas y tablas de transiciones. Además, permitió comprender mejor cómo los compiladores identifican y clasifican elementos clave de un programa, como palabras reservadas y literales.

Perez Delgado Erandy Estefanya: El desarrollo de este analizador léxico nos permitió aplicar de manera práctica los conceptos que aprendimos en clase, como los autómatas y las expresiones regulares. Al crear las tablas de símbolos y literales, también reforzamos nuestras habilidades en la manipulación de datos y la implementación de algoritmos de búsqueda e inserción. Utilizando herramientas como Flex y JFLAP, logramos comprender mejor cómo los compiladores identifican y clasifican los diferentes elementos de un programa, como identificadores, palabras reservadas y literales. Además, trabajar con C nos ayudó a afianzar nuestros conocimientos sobre la estructura y funcionamiento de los compiladores.

Siliano Haller Rodrigo: La práctica me ayudó a comprender de mejor manera el funcionamiento de los compiladores que usan algunos lenguajes como el usado en esta, que es C.

También me sirvió para repasar algunos conceptos vistos en C como la declaración de funciones y las estructuras de búsqueda.

Aprendí a manejar Flex y su sintaxis para hacer analizadores de texto.

Referencias

- Aho, A. V., Sethi, R., & Ullman, J. D. (2000). *Compiladores: Principios, técnicas y herramientas*. Addison-Wesley Iberoamericana.
- Pratt, T. W., & Zelkowitz, M. V. (1998). *Lenguajes de programación: Diseño e implementación* (1ª ed.). Prentice Hall.
- Aaby, A. (2003). *Compiler construction using flex and bison*. Walla Walla College.
- Catalán, J. R. (2010). *COMPILADORES. Teoría e implementación*. RC Libros.
- Grune, D., Reeuwijk, K. van, Bal, H. E., Jacobs, C. J. H., & Langendoen, K. (2012). *Modern compiler design* (1 recurso en línea). Springer. Disponible en la base de datos LIBRUNAM.
- Hernández, A. (n.d.). *Facultad de Ingeniería, Universidad Nacional Autónoma de México, División de Ingeniería Mecánica y Eléctrica, Departamento de Ingeniería en Computación*.
http://www.ptolomeo.unam.mx:8080/jspui/bitstream/132.248.52.100/10230/1/7J%20APUNTES%20DE%20COMPILADORES_OCR.pdf