UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Materia:

COMPILADORES

Profesor:

M.I. ELBA KAREN SAENZ GARCÍA

Alumnos:

López Gonzáles José Sadam
Paredes Pacab Rosaura Elena
Ruiz Godoy Franco

PROYECTO: “ANALIZADOR LÉXICO”

Grupo: 01

Números de cuenta:
-314279810
-419049582
-317159019

Semestre 2024 - 1
Fecha de entrega: 24 de octubre de 2023

ÍNDICE

OBJETIVO

Aplicar los conocimientos adquiridos durante la clase de compiladores para implementar un analizador léxico. Utilizaremos la gramática propuesta por el grupo para identificar y validar la estructura léxica de un archivo fuente, aplicando las características específicas del analizador léxico previamente desarrollado.

INTRODUCCIÓN

Los compiladores desempeñan un papel esencial en el mundo de la programación y la informática, ya que son responsables de traducir el código fuente escrito por los programadores en lenguaje de alto nivel a un código ejecutable por la máquina. Éste se encuentra integrado por varios procesos, y uno de los más importantes es el analizador léxico, que se encarga de escanear el código fuente y descomponerlo en unidades léxicas significativas, conocidas como "tokens". Estos últimos incluyen palabras clave, identificadores, literales, operadores, etc. Son la base para la posterior fase de análisis sintáctico y generación de código intermedio.

El presente proyecto nos enfocamos en la planeación y creación de un analizador léxico utilizando la herramienta "Flex". A lo largo de éste, desarrollamos el proceso de diseño, implementación y pruebas de este componente esencial en la construcción de compiladores.

Uno de los puntos claves en el desarrollo del analizador léxico fue la definición de expresiones regulares que describen los patrones léxicos del lenguaje de programación en cuestión. Estas expresiones regulares son como la columna vertebral del analizador léxico, ya que nos permiten reconocer y clasificar los tokens. Nuestra precisión al desarrollar las expresiones regulares fue muy importante para el correcto funcionamiento del analizador.

La implementación del analizador léxico en "Flex" implicó la creación de un archivo de especificación que asocie las expresiones regulares con las acciones correspondientes. Estas acciones se ejecutan cuando se encuentra un patrón coincidente, permitiendo la identificación y extracción de los tokens.
Del mismo modo, las pruebas hechas al código juegan un papel importante en el desarrollo del proyecto, ya que permitieron verificar que el analizador léxico funcione correctamente en una variedad de situaciones.

Por último, la documentación a lo largo de las diversas etapas del proyecto se fue por ende fundamental para nosotros, puesto que no solo nos ayudó a mantener una resolución más organizada y precisa, sino que también desempeñó un papel clave en la identificación y corrección de errores durante las fases de planificación y codificación del analizador.

DESARROLLO

DESCRIPCIÓN DEL PROBLEMA

El proyecto tiene como objetivo el desarrollo de un analizador léxico utilizando lex/flex para la identificación y clasificación de componentes léxicos en un lenguaje de programación o dominio específico. Los componentes léxicos pertenecen a una serie de clases inamovibles y deben ser reconocidos y procesados de acuerdo con las siguientes especificaciones:

1. Clases de Componentes Léxicos:
   - Las clases de los componentes léxicos válidos para el analizador léxico son las siguientes:
     - Palabras reservadas

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

- Operadores aritméticos (+, -, *, /, $)

1	Operadores aritméticos + - * / $

- Operadores de asignación.

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

- Símbolos especiales (,), {, }, [,], &, ,, :)

3	Símbolos especiales ( ) { } [ ] & , :

- Operadores relacionales.

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

- Identificadores

- Constantes numéricas enteras (base 10, signadas o no signadas, máximo 6 dígitos, mínimo 1)
- Constantes numéricas reales (no signadas)
- Constantes de cadenas (comienzan y terminan con comillas dobles)
- Constantes de caracteres (encerradas entre comillas sencillas)

2. Entrada del Analizador Léxico:
- El analizador léxico tomará como entrada un archivo que contiene el programa fuente. La ubicación de este archivo se especificará desde la línea de comandos al ejecutar el analizador léxico.

3. Delimitadores de Componentes Léxicos:
- Los componentes léxicos se delimitará por uno o varios espacios, tabuladores o saltos de línea, así como el inicio de otro componente léxico. Se debe tener en cuenta el manejo de constantes numéricas enteras para evitar ambigüedades.

4. Representación de Tokens:
- Los tokens se representarán en una estructura con dos campos:
  - Campo 1: La clase del componente léxico.
  - Campo 2: El valor de acuerdo con las tablas e indicaciones proporcionadas.

5. Tablas de Símbolos:
- El analizador léxico deberá crear una tabla de símbolos para almacenar identificadores. Esta tabla contendrá los siguientes campos:
  - Posición
  - Nombre del identificador
  - Tipo (inicializado a -1)
- Se indicará en el documento a entregar la estructura de datos empleada y el método de búsqueda utilizado para la tabla de símbolos.

6. Tablas de Literales:
- Se utilizarán dos tablas de literales separadas, una para las constantes numéricas y otra para las cadenas. Cada tabla contendrá dos campos:
  - Posición
  - Dato (cadena o constante numérica según el caso)

7. Manejo de Errores Léxicos:
- En caso de detectar un error léxico, el analizador léxico deberá continuar el reconocimiento a partir del siguiente símbolo.
- Los errores léxicos encontrados podrán mostrarse en pantalla o escribirse en un archivo de registro. El programa debe ser capaz de recuperarse de estos errores y continuar el análisis.

8. Salida del Analizador Léxico:

- Al término del análisis léxico, el programa deberá mostrar en pantalla la tabla de símbolos, las tablas de literales y los tokens. También se podrán almacenar estos resultados en archivos para una revisión posterior.

9. Documentación y Comentarios:

- El programa deberá estar debidamente comentado, con descripciones breves de su funcionamiento, el nombre del o los desarrolladores, la fecha de elaboración y detalles sobre la funcionalidad de cada función. Se debe prestar especial atención a la sangría para indicar la dependencia de instrucciones.

El proyecto tiene también como finalidad facilitar el análisis y procesamiento del código fuente en un lenguaje de programación específico o en un dominio particular. La correcta implementación del analizador léxico es esencial para detectar errores léxicos y producir una salida coherente y estructurada que sea utilizable en fases posteriores del proceso de compilación o procesamiento de texto.

ANÁLISIS

El proyecto requiere una implementación cuidadosa de reglas léxicas en lex/flex para garantizar que los componentes léxicos se reconozcan y clasifiquen correctamente.
En la fase inicial de desarrollo del analizador léxico, nos enfrentamos a diversas situaciones que requerían soluciones o ser tomadas en cuenta para lograr la completa construcción de nuestro proyecto.Como lo son:

- **La Definición de las Expresiones Regulares:** Describir las secuencias esenciales para identificar y clasificar distintos elementos de nuestro lenguaje, como números, identificadores y palabras clave. Al trabajar en esto debemos asegurarnos de que son precisas puesto que eran esenciales para cubrir todas las posibles variantes de los tokens creados.

- **La Definición de los Catálogos y las Tablas para Crear los Tokens:** Una vez identificamos los patrones, era necesario estructurar la información en tablas y catálogos que almacenan cada token y sus atributos, como tipo, clase y valor.

- **El Algoritmo de Búsqueda para Analizar los Caracteres en el Archivo:** Lo siguiente fue recrear un algoritmo con la rapidez con la que puede recorrer y clasificar los caracteres del archivo fuente. Era fundamental elegir o diseñar un algoritmo de búsqueda que sea capaz de identificar tokens incluso cuando estos están intercalados o tienen la necesidad de crearse.

- **La Gestión de Errores del Código:** En el proyecto del analizador léxico debíamos asegurarnos de que sea capaz de detectar y reportar errores de manera clara y útil para el usuario.

- **El Manejo de Espacios en Blanco, Comentarios y Delimitadores:** Otro reto fueron los espacios en blanco y los comentarios, esenciales para la legibilidad de nuestro código, los cuales tenemos que asegurarnos de que sean ignorados por el analizador léxico. Asimismo, los delimitadores, como paréntesis o corchetes deben ser identificados con precisión y el analizador debe catalogarlo correctamente.

Aunado a lo anterior, la gestión de tablas y la detección de errores son aspectos cruciales que deben abordarse de manera efectiva. La creación de una documentación detallada y comprensible es esencial para facilitar la comprensión y el mantenimiento del programa. La salida de resultados en pantalla y su almacenamiento en archivos contribuirán a la evaluación y depuración efectiva del analizador léxico.

Empleamos las siguientes expresiones regulares:

```
//EXPRESION REGULAR PARA LA CLASE 0 (PALABRAS RESERVADAS)
//RESERVADA cadena|caracter|else|entero|for|if|real|return|void|while

//EXPRESION REGULAR PARA LA CLASE 1 (OPERADOR ARITMETICO)
//OPARITMETICO [\+-\*\/\$]|\+\x20|\-\x20

//EXPRESION REGULAR PARA LA CLASE 2 (OPERADOR DE ASIGNACION)
//OP_ASIGNACION ~|\+~|\-~|\*~|\/~|\$~

//EXPRESION REGULAR PARA LA CLASE 3 (SIMBOLOS ESPECIALES)
//SIMB_ESPECIAL [\(\)\{\}\[\]&,:]

//EXPRESION REGULAR PARA LA CLASE 4 (OPERADORES RELACIONALES)
//OP_RELACIONALES \^\^|\^\x22|\=\=|\^\^\=|\^\x22=|<>

//EXPRESION REGULAR PARA LA CLASE 5 (IDENTIFICADORES)
//IDENT ^[a-zA-Z][a-zA-Z0-9]{1,5}_$

//EXPRESION REGULAR PARA LA CLASE 6 (CONSTANTES NÚMERICAS ENTERAS)
//CONST_NUMERICA_ENTERA  [-|\+]?[0-9]{1,6}

//EXPRESION REGULAR PARA LA CLASE 7 (CONSTANTES NÚMERICAS REALES)
//CONST_NUMERICA_REALES \d*\.\d*

//EXPRESION REGULAR PARA LA CLASE 8 (CONSTANTES CADENAS)
//CONST_CADENAS ^\x22[^\x22]{0,38}\x22$

//EXPRESION REGULAR PARA LA CLASE 9 (CONSTANTES CARÁCTER)
//CONST_CARACTER \x27[^.]\x27

//DELIMITADOR [ \t\n]+

//ERROR .+
```

Definamos las expresiones regulares para diferentes clases de tokens en un analizador léxico o lexer. Cada expresión regular corresponde a un tipo de token en un lenguaje de programación. Por ejemplo, la expresión regular para "CLASE 0" coincide con palabras reservadas como "cadena", "caracter", "else", etc. Las expresiones regulares para otras clases, como operadores aritméticos, operadores de asignación, símbolos especiales, identificadores, constantes numéricas enteras, reales, cadenas y caracteres, están diseñadas para identificar esos elementos en el código fuente. Además, parece haber una expresión regular llamada "DELIMITADOR" que coincide con espacios en blanco y caracteres de control. En conjunto, estas expresiones regulares se utilizarían para analizar y dividir el código fuente en tokens en un compilador o intérprete.

DISEÑO E IMPLEMENTACIÓN

Este analizador léxico solamente funciona con una entrada en formato .txt, a subes deben de tener salto de líneas para que pueda leer correctamente las entradas.

Declaraciones y bibliotecas: El código comienza incluyendo varias bibliotecas estándar de C, como stdio.h, stdlib.h, y string.h. También incluye un archivo de encabezado llamado "tabla_anlex.h", que probablemente contenga definiciones y funciones relacionadas con la gestión de tablas de símbolos. Además, define algunas constantes, variables y arreglos que se utilizarán más adelante.

```
/* DEFINICIONES */
%{
/* BIBLIOTECAS */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include "tabla_anlex.h"
#include "lex.yy.h"
```

Catálogos: Se definen varios catálogos que contienen palabras reservadas, operadores aritméticos, operadores de asignación, símbolos especiales y operadores relacionales. Estos catálogos se utilizan para identificar y clasificar los tokens encontrados en el código fuente.

```
// Catálogo de palabras reservadas
const char *reservada[TAM_RESERVADA] = {"cadena", "caracter", "else", "entero", "for", "if", "real", "return", "void", "while"};

// Catálogo de operadores aritméticos
const char *aritmeticos[TAM_ARITMETICO] = {"+", "-", "*", "/", "$"};

// Catálogo de operadores de asignación
const char *asignacion[TAM_ASIGNACION] = {"~", "+~", "-~", "*~", "/~", "$~"};

// Catálogo de simbolos especiales
const char *simb_especial[TAM_SIMB_ESP] = {"(", ")", "{", "}", "[", "]", ",", "&", ":"};

// Catálogo de operadores relacionales
const char *op_relacionales[TAM_OP_RELACIONAL] = {"^^", "^\"", "==", "^^=", "^\"=", "<>"};
```

El for de los catálogos busca la posición de los lexema en su catálogo correspondiente.

```
int posc_Cat_R(char *lexema) {
    for (int i = 0; i < TAM_RESERVADA; i++) {
        if (strcmp(lexema, reservada[i]) == 0) {
            return i;
        }
    }
    return -1;
}


int posc_Cat_Asignacion(char *op) {
    for (int i = 0; i < TAM_ASIGNACION; i++) {
        if (strcmp(op, asignacion[i]) == 0) {
            return i;
        }
    }
    return -1;
}

int posc_Cat_OpRelacional(char *op) {
    for (int i = 0; i < TAM_OP_RELACIONAL; i++) {
        if (strcmp(op, op_relacionales[i]) == 0) {
            return i;
        }
    }
    return -1;
}
```

Expresiones regulares sin nombre: En la sección, se encuentran expresiones regulares sin nombre que describen patrones de tokens, como palabras reservadas, operadores, identificadores, números, etc. Estas expresiones se utilizan en las reglas siguientes para identificar tokens específicos.

Reglas: Las reglas comienzan con %%. En esta sección, se asocian expresiones regulares con acciones específicas a realizar cuando se encuentra un token que coincide con el patrón. Por ejemplo, cuando se encuentra una palabra reservada, se llama a la función insertar_Token con la clase 0 y la posición de la palabra reservada en el catálogo de palabras reservadas.

```
/*REGLAS*/
%%
cadena|caracter|else|entero|for|if|real|return|void|while {insertar_Token(0, posc_Cat_R(yytext));}
[\+\-\*\/\$]|\+\x20|\-\x20   {insertar_Token(1, (int)(yytext[0]));}
~|\+~|\-~|\*~|\/~|\$~       {insertar_Token(2, posc_Cat_Asignacion(yytext));}
[\(\)\{\}\[\]&,:]       {insertar_Token(3, (int)(yytext[0]));}
\^\^|\^\x22|\=\=|\^\^\=|\^\x22=|<>    {insertar_Token(4, posc_Cat_OpRelacional(yytext)); }
^[a-zA-Z][a-zA-Z0-9]{1,5}_$              {insertar_Token(5, 0);}
[\-|\+]?[0-9]{1,6} {
                    char* lexema = (char*)malloc(yyleng + 1); // Asignar memoria para la copia
                    strncpy(lexema, yytext, yyleng);
                    lexema[yyleng] = '\0';
                    insertar_LitNum(cont_num_ent,lexema);insertar_Token(6, cont_num_ent++);
                }

^\d*\.\d*$ {insertar_LitReal(cont_num_real,yytext);insertar_Token(7, cont_num_real++); }

^\x22[^\x22]{0,38}\x22$ {
                    char* lexema = (char*)malloc(yyleng + 1); // Asignar memoria para la copia
                    strncpy(lexema, yytext, yyleng);
                    lexema[yyleng] = '\0';
                    insertar_LitString(cont_cadena,lexema);insertar_Token(8, cont_cadena++);
                }

\x27[^.]\x27     {insertar_Token(9, (int)(yytext[1])); }

[\x20\t\n]+         {/* Ignore */ }

.+              {printf("Error de sintaxis: %s\n", yytext); }

%%
```

Funciones de inserción: Las funciones insertar_Token, insertar_LitNum e insertar_LitReal se utilizan para insertar información sobre tokens en alguna estructura de datos o tabla de símbolos. Estas funciones parecen ser parte de la gestión de tablas de símbolos.

Función main: En la función main, se configura la entrada de datos (yyin) para que provenga de un archivo proporcionado como argumento en la línea de comandos o desde la entrada estándar (terminal) si no se proporciona un archivo. Luego, se llama a la función yylex para iniciar el análisis léxico del código fuente. Finalmente, se llama a la función imprimirTablas para imprimir información sobre los tokens encontrados.

```
//PARA PROBAR EL ANALIZADOR TENEMOS DOS FORMAS (POR TERNIMAL O INGRESANDO UN ARCHIVO)
main( argc, argv )
    int argc;
    char **argv;
{
    ++argv, --argc;
    if ( argc > 0 )
    yyin = fopen( argv[0], "r" ); //PARA QUE FUNCIONE BIEN CON UN ARCHIVO, CADA LEXEMA O PATRON DEBE IR SEGUIDO POR UN SALTO DE LINEA
    else
    yyin = stdin; //entrada estandar (terminal) FUNCIONA CORRECTAMENTE, BASTA CON INGRESAR EL PATRON O LEXEMA Y PRESIONAR ENTER
    //printf("TOKENS:\n");
    yylex();
    imprimirTablas();
}
```

Para **TABLA_ANLEX_H** se tomó en cuenta lo siguiente:

Directivas de preprocesador:
Las líneas #ifndef TABLA_ANLEX_H y #define TABLA_ANLEX_H son directivas de preprocesador que se utilizan para evitar la inclusión múltiple de este archivo de encabezado en un mismo programa. Si TABLA_ANLEX_H no está definido, se define, lo que significa que el contenido del archivo de encabezado se procesa una sola vez. Esto ayuda a prevenir errores de compilación.

```
#ifndef TABLA_ANLEX_H
#define TABLA_ANLEX_H
```

Definición de estructuras:
Se definen dos estructuras de datos:
NodoTabla: Representa un nodo en una tabla de símbolos. Contiene la posición, el lexema y una referencia al siguiente nodo.
NodoToken: Representa un nodo en la lista de tokens. Contiene la clase, la posición y una referencia al siguiente nodo.

```
typedef struct NodoTabla {
    int posicion;
    const char* lexema;
    struct NodoTabla* siguiente;
} NodoTabla;

typedef struct NodoToken {
    int clase;
    int posicion;
    struct NodoToken* siguiente;
} NodoToken;

// Funciones para operar con la tabla de sÃ-mbolos

int insertar_Ident(const char* lexema);
void insertar_LitNum(int posc, const char* lexema);
void insertar_LitReal(int posc, const char* lexema);
void insertar_LitString(int posc, const char* lexema);
int buscar_Ident(const char* lexema);

void insertar_Token(int clase, int posc);

void imprimirTablas();
#endif
```

Para el Código de **Tabla_anlex_c**. Se hizo uso de:
Declaración de estructuras de datos:
Se declaran varios punteros a estructuras de datos, cada uno de los cuales se utilizará para mantener información específica sobre identificadores, literales numéricos, literales de punto flotante (real), literales de cadena y tokens.
Cada estructura de datos se define como un tipo de dato compuesto que contendrá información sobre la posición, el lexema y una referencia al siguiente elemento en la lista.

Función insertar_Ident:
Esta función se utiliza para insertar identificadores (variables, nombres de funciones, etc.) en la tabla de identificadores (tablaIden).

Si la tabla de identificadores está vacía, se crea un nuevo nodo y se asigna el lexema y la posición proporcionada.
Si la tabla de identificadores no está vacía, se busca si el identificador ya existe en la tabla. Si existe, se devuelve su posición. Si no existe, se crea un nuevo nodo con el lexema y se agrega al final de la lista.
Esta función parece estar diseñada para mantener un registro único de los identificadores en el código fuente.

```
int insertar_Ident(const char* lex) {
   if (tablaIden == NULL){
      NodoTabla* nuevoNodo = (NodoTabla*)malloc(sizeof(NodoTabla));
      nuevoNodo-> posicion = 0;
      nuevoNodo-> lexema = lex;
      nuevoNodo-> siguiente = tablaIden;
      tablaIden = nuevoNodo;
      return 0;
   }else{
      NodoTabla* nuevoNodo = (NodoTabla*)malloc(sizeof(NodoTabla));
      nuevoNodo-> posicion = buscar_Ident(lex);
      nuevoNodo-> lexema = lex;
      nuevoNodo-> siguiente = NULL;
      NodoTabla* actual = tablaIden;
        while (actual->siguiente != NULL) {
            actual = actual->siguiente;
        }
        actual->siguiente = nuevoNodo;
        return nuevoNodo-> posicion;
   }
}
```

Funciones insertar_LitNum, insertar_LitReal y insertar_LitString:
Estas funciones se utilizan para insertar literales numéricos enteros, literales de punto flotante (real) y literales de cadena en sus respectivas tablas (tablaLitNum, tablaLitReal y tablaLitString).
Funcionan de manera similar a insertar_Ident, creando nuevos nodos si la tabla está vacía o agregando nuevos nodos al final de la lista si ya contiene elementos.

```
void insertar_LitNum(int posc, const char* lex) {
if (tablaLitNum == NULL){
     NodoTabla* nuevoNodo = (NodoTabla*)malloc(sizeof(NodoTabla));
     nuevoNodo-> posicion = posc;
     nuevoNodo-> lexema = lex;
     nuevoNodo-> siguiente = tablaLitNum;
     tablaLitNum = nuevoNodo;
   }else{
     NodoTabla* nuevoNodo = (NodoTabla*)malloc(sizeof(NodoTabla));
     nuevoNodo-> posicion = posc;
     nuevoNodo-> lexema = lex;
     nuevoNodo-> siguiente = NULL;
     NodoTabla* actual = tablaLitNum;
       while (actual->siguiente != NULL) {
           actual = actual->siguiente;
       }
       actual->siguiente = nuevoNodo;
   }
}
```

```
void insertar_LitReal(int posc, const char* lex) {
if (tablaLitReal == NULL){
      NodoTabla* nuevoNodo = (NodoTabla*)malloc(sizeof(NodoTabla));
      nuevoNodo-> posicion = posc;
      nuevoNodo-> lexema = lex;
      nuevoNodo-> siguiente = tablaLitReal;
      tablaLitReal = nuevoNodo;
   }else{
      NodoTabla* nuevoNodo = (NodoTabla*)malloc(sizeof(NodoTabla));
      nuevoNodo-> posicion = posc;
      nuevoNodo-> lexema = lex;
      nuevoNodo-> siguiente = NULL;
      NodoTabla* actual = tablaLitReal;
        while (actual->siguiente != NULL) {
            actual = actual->siguiente;
        }
        actual->siguiente = nuevoNodo;
   }


}

void insertar_LitString(int posc, const char* lex) {
if (tablaLitString == NULL){
      NodoTabla* nuevoNodo = (NodoTabla*)malloc(sizeof(NodoTabla));
      nuevoNodo-> posicion = posc;
      nuevoNodo-> lexema = lex;
      nuevoNodo-> siguiente = tablaLitString;
      tablaLitString = nuevoNodo;
   }else{
      NodoTabla* nuevoNodo = (NodoTabla*)malloc(sizeof(NodoTabla));
      nuevoNodo-> posicion = posc;
      nuevoNodo-> lexema = lex;
      nuevoNodo-> siguiente = NULL;
      NodoTabla* actual = tablaLitString;
        while (actual->siguiente != NULL) {
            actual = actual->siguiente;
        }
        actual->siguiente = nuevoNodo;
   }


}
```

Función insertar_Token:

Esta función se utiliza para insertar información sobre tokens en la tabla de tokens (tablaToken).

Cada token se asocia con una clase (como palabras reservadas, operadores, identificadores, etc.) y una posición (la posición en su respectiva tabla, si corresponde).

Al igual que las otras funciones de inserción, esta función agrega nuevos nodos al final de la lista si la tabla ya contiene elementos.

```
void insertar_Token(int clase, int posc){
    if (tablaToken == NULL){
      NodoToken* nuevoNodo = (NodoToken*)malloc(sizeof(NodoToken));
      nuevoNodo-> clase = clase;
      nuevoNodo-> posicion = posc;
      nuevoNodo-> siguiente = tablaToken;
      tablaToken = nuevoNodo;
   }else{
      NodoToken* nuevoNodo = (NodoToken*)malloc(sizeof(NodoToken));
      nuevoNodo-> clase = clase;
      nuevoNodo-> posicion = posc;
      nuevoNodo-> siguiente = NULL;
      NodoToken* actual = tablaToken;
        while (actual->siguiente != NULL) {
            actual = actual->siguiente;
        }
        actual->siguiente = nuevoNodo;
   }
    //printf("\n(%d,%d)\n",clase,posc);

}
```

Función imprimirTablas:

Esta función imprime información sobre las tablas de símbolos y los tokens.

Recorre cada una de las tablas y muestra la posición y el lexema de cada elemento en la tabla. Finalmente, imprime información sobre los tokens encontrados.

```
void imprimirTablas() {
    struct NodoTabla* actual;
    /*printf("Tabla de Simbolos (Identificadores):\n");
    struct Nodo* actual = cabeza;
    while (actual != NULL) {
        printf("\n(%d,%d)\n",clase,posc);
        printf("%d ", actual->valor);
        actual = actual->siguiente;
    }
    printf("\n");*/

    printf("\nTabla de literales (numericas):\n");
    actual = tablaLitNum;
    while (actual != NULL) {
        printf("\n(%d,%s)\n",actual->posicion,actual->lexema);
        actual = actual->siguiente;
    }
    printf("\n");

    printf("\nTabla de literales (cadenas):\n");
    actual = tablaLitString;
    while (actual != NULL) {
        printf("\n(%d,%s)\n",actual->posicion,actual->lexema);
        actual = actual->siguiente;
    }
    printf("\n");

    struct NodoToken* act = tablaToken;
    printf("TOKENS:\n");
    while (act != NULL) {
        printf("\n(%d,%d)\n", act->clase, act->posicion);
        act = act->siguiente;
    }
    printf("\n");
```

Resultado.

De la cadena:

```
cadena
for
+
/
$
~
+~
$~
(
)
&
:
^"
^^=
<>
Iden1_
78
+90
-1
.89
2.4
7.
"hola mundo"
"debo funcionar"
'a'
nofunciono
a
d
```

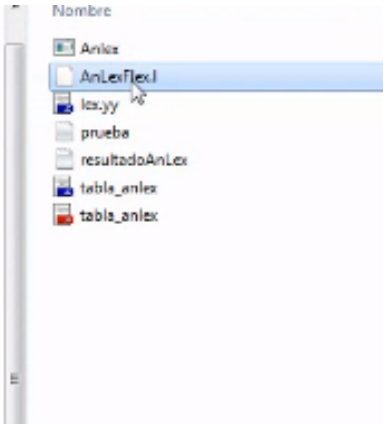
Dio como resultado.

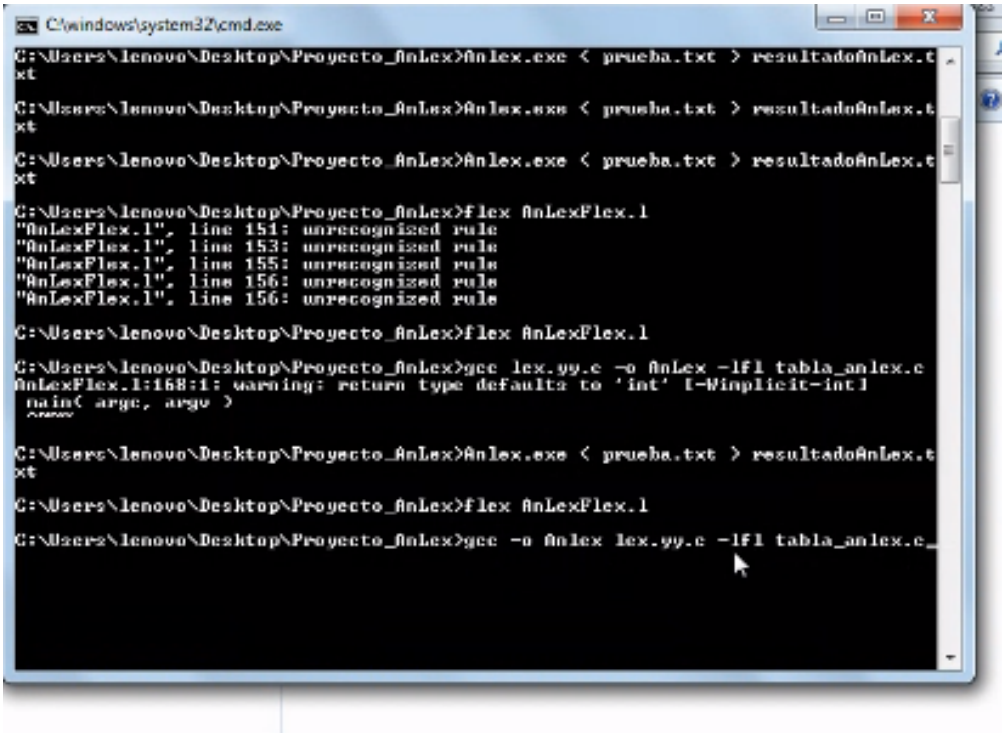
```
Error de sintaxis: .89
Error de sintaxis: 2.4                    (2,1)
Error de sintaxis: 7.
Error de sintaxis: nofunciono             (2,5)
Error de sintaxis: a
Error de sintaxis: d                      (3,40)

Tabla de literales (numericas):           (3,41)

(0,78)                                    (3,38)

(1,+90)                                   (3,58)

(2,-1)                                    (4,1)

Tabla de literales (cadenas):             (4,3)

(0,"hola mundo")                          (4,5)

(1,"debo funcionar")                      (5,0)

TOKENS:                                   (6,0)

(0,0)                                     (6,1)

(0,4)                                     (6,2)

(1,43)                                    (8,0)

(1,47)                                    (8,1)

(1,36)                                    (9,97)

(2,0)
```

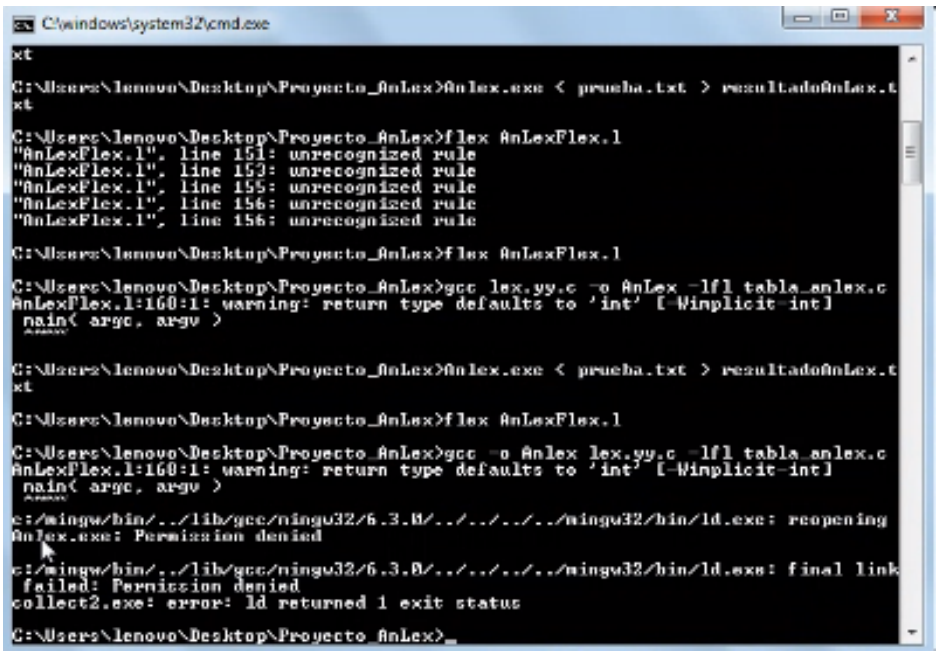
Compilación

1. **Abre la consola de equipo Windows (o linux).**
2. **Usar el comando: flex AnLexFlex.l**
3. **gcc lex.yy.c -o AnLex -lfl tabla_anlex.c (Se entrelaza con el archivo tabla_anlex.c porque en él se encuentra el archivo yylex, es decir, está compilando dos archivos a la vez).**
4. **Anlex.exe < prueba.txt < resultadoAnlex.txt (Donde prueba.txt es el archivo a analizar y resultadoAlex.txt es donde se arrojarán los resultados del analizador léxico).**

5. **Antes de compilar, se deberán tener todos los archivos dentro de una carpeta.**







CONCLUSIONES

López Gonzáles José Sadam

Hemos creado un programa especial que funciona bien para encontrar y clasificar palabras clave en un archivo fuente, algo muy importante al hacer un programa de computadora. También usamos formas de organizar y guardar información de manera inteligente, y evitamos tener la misma palabra clave repetida, lo que hace que el análisis de las palabras clave sea más rápido y exacto.

Paredes Pacab Rosaura Elena

La realización de este proyecto me permitió entender un poco más a fondo la estructura de los lenguajes y cómo las máquinas interpretan el texto a nivel interno, lo cual es el propósito del analizador léxico. Me ayudó a trabajar sistemáticamente, siguiendo pasos, cómo definir los catálogos y las expresiones regulares, los tokens y la inserción de éstos. Esta experiencia pulió un poco más mis habilidades de programación y me hizo estructurar los pasos a seguir antes de pasar a la codificación.

Ruiz Godoy Franco

Este proyecto de compiladores nos ha brindado la oportunidad de poner en práctica lo que aprendimos en nuestras clases teóricas. Hemos creado un analizador léxico que puede identificar y clasificar las partes importantes de un programa en un archivo fuente, lo cual es fundamental cuando se está construyendo un programa que traduce código. Además, hemos utilizado herramientas para organizar y manejar de manera efectiva la información relacionada con estas partes del programa.

BIBLIOGRAFÍA

- Componentes Léxicos, Patrones y Lexemas. (n.d.). Recuperado el 10 de octubre del 2023 de: http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/22_componentes_lxicos_patrones_y_lexemas.html
- Educacion OnLine. (2021, 24 abril). *FLEX Analizador Lexico - Explicacion, compilacion y pruebas.* [Vídeo]. YouTube. Recuperado el 14 de octubre de https://www.youtube.com/watch?v=AyB7gVNor9U