Universidad Nacional Autónoma de México

Facultad de Ingeniería

Primer Programa: Analizador Léxico.

Alumnos:
- García López Erik
- Serapio Hernández Alexis Arturo

Materia: Compiladores

Profesor: M.I. Laura Sandoval Montaño

Semestre 2024-1

Fecha de Entrega: 03/10/2023

- **Objetivo:**

Elaborar un analizador léxico en lex/flex que reconozca los componentes léxicos vistos y propuestos para este programa.

- **Descripción del Problema:**

Un analizador léxico, también conocido como escáner, es una parte fundamental de un compilador o intérprete de lenguaje de programación.
Su función principal es analizar un flujo de texto o código fuente y dividirlo en unidades más pequeñas llamadas "tokens". Estos tokens son las unidades mínimas de un lenguaje de programación y representan palabras clave, identificadores, números, operadores y otros elementos significativos en el código fuente.

El problema de este proyecto, fue el planteamiento y realización de un analizador léxico, para esto se tiene que cumplir con una serie de tareas dentro de nuestro código, tales tareas deberán ser reconocimiento y generación de la secuencia de tokens, detección de errores léxicos y la creación de catálogos.

Para nuestro proyecto, que se desarrolló en lex se tomaron en cuenta los siguientes puntos, donde se incluyen las clases que definimos al momento de la profesora solicitar el proyecto.

- ☐ **Características de los componentes léxicos a cumplir:**

Los componentes léxicos que se aceptaran en nuestro analizador léxico son los siguientes:

Clase	Descripción
0	Operadores aritméticos + - / * %
1	Operadores lógicos (ver tabla).
2	Operadores relacionales (ver tabla).
3	Constantes numéricas enteras. Sólo en base 10. Si está con signo, encerrarlo entre ( ). Ejemplos: 0 672 (-265) (+49)
4	Palabras reservadas (ver tabla).
5	Identificadores. Inician con _ y le sigue una letra minúscula o mayúscula, después pueden contener letras minúsculas o mayúsculas, dígitos y _
6	Símbolos especiales ( ) { } ; , [ ] : #
7	Operadores de asignación (ver tabla).
8	Constantes cadenas. Encerradas entre comillas (“) cualquier secuencia de caracteres incluye salto de línea.
9	Operadores sobre cadenas (ver tabla).

Como sabemos, el analizador léxico se encarga de generar tokens.
Dichos tokens tienen una estructura basada en dos campos (x,y), en donde el primer término es la clase y el segundo es el valor de acuerdo a las clases y sus descripciones.

Valor	Op. lógico
0	&&
1	||
2	!

Tabla de la Clase 1: Operadores Lógicos

Valor	Op. relacional
0	==
1	!=
2	>
3	<
4	>=
5	<=

Tabla de la Clase 2: Operadores Relacionales

Valor	Palabra reservada	Equivale en C a
0	assinado	void
1	caso	case
2	enquanto	while
3	fazer	do
4	flutuador	float
5	inteiro	int
6	para	for
7	quebrar	break
8	retorno	return
9	se	if
10	trocar	switch

Tabla de la Clase 4: Palabras Reservadas

Valor	Op. asignación
0	=
1	+=
2	-=
3	*=
4	/=
5	%=

Tabla de la Clase 7: Operadores de Asignación

Valor	Op. Sobre cadenas
0	&
1	like

Tabla de la Clase 9: Operadores Sobre Cadenas

Entonces, el valor para el token de cada identificador será la posición dentro de la tabla de símbolos. Para las palabras reservadas, operadores relacionales, los operadores lógicos, operadores sobre cadenas y los operadores de asignación será la posición en su correspondiente tabla (catálogo).

- ## Expresión Regular de cada clase:

Para la elaboración del analizador léxico, es necesario hacer una expresion regular para las clases, las expresiones regulares a las que llegamos fueron las siguientes:

Para la Clase 0 : Operadores Aritméticos:

```
op_arit    [+\-/*%]
```

Para la Clase 1 : Operadores Lógicos:

```
op_logic   (&&)|(\|\|)\(!)
```

Para la Clase 2 : Operadores Relacionales:

```
op_rel     ==|!=|>|<|>=|<=
```

Para la Clase 3 : Constantes Numéricas Enteras:

```
constantes_ent   (\([-+][0-9]+\)|[0-9]|_)*
```

Para la Clase 4: Palabras Reservadas:

```
pal_res        assinado|caso|enquanto|fazer|flutuador|inteiro|para|quebrar|retorno|se|trocar
```

Para la Clase 5: Identificadores:

```
identif    _[a-zA-Z]+([a-zA-Z]|[0-9]|_)*
```

Para la Clase 6: Símbolos Especiales:

```
simb_esp   [(){};,:#\[|\]]
```

Para la Clase 7: Operadores de Asignación:

```
op_asig    “=”|“+=”|“-=”|“.=”|“*=”|“/=”|“%=”
```

Para la Clase 8: Constantes Cadenas:

```
cadena     \”[^\”]*\”
```

Para la Clase 9: Operadores sobre Cadenas:

```
cadena     &|like
```

Expresiones Regulares Adicionales en el código:

```
salto_linea     [\r\n]
  espacio [“ ”]
```

- **Propuesta de solución del programa**

Para la realización del analizador, propusimos una función principal en la que pueda leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos de los que hará uso un posterior análisis sintáctico.

- **Fases del desarrollo del programa**
  - **Analisis(Planificacion, Indicando participantes por cada actividad)**

Actividad	Participantes
1. Creación de expresiones regulares para las 10 clases	Ambos
2. Probar que funciones las expresiones regulares y si están mal corregirlas.	Ambos
3. Creación de los archivos (archivo de salida y el archivo que guarda los tokens generados)	Alexis
4. Creación de estructuras para almacenar los nodos	Erik
5. Crear listas para los catálogos (palabras reservadas, operadores relacionales, operadores lógicos, operadores sobre cadenas y operadores de asignación).	Ambos
6. Generar las estructuras de datos para las tablas y los catálogos.	Alexis
7. Llenar las listas	Alexis
8. Completar la tabla de símbolos	Erik
9. Completar la tabla de literales	Erik

10. Hacer una función que imprima las tablas	Alexis
11. Comprobar el programa y si hay errores, corregirlos	Ambos
12. Documentar	Ambos

- **Diseño (Indicar exactamente cómo se definirá la tabla de símbolos, la tabla de literales y los tokens, la técnica de búsqueda e inserción de los identificadores)**

El diseño del analizador se propuso como un programa con estructura básica en lex.

Se comenzó a idear el programa a partir de la creación de las expresiones regulares, con estas pudimos hacernos una idea de la manera en la que podríamos implementar las funciones que retornan las cadenas que acepta nuestro analizador.

Al observar y llevar a cabo una prueba de estas expresiones regulares y confirmar que aceptan las cadenas solicitadas.

Uno de los aspectos más importantes del desarrollo de un Analizador Léxico, es la creación de la tabla de símbolos, esta tabla que tiene como característica poseer elementos, posición, nombre y tipo de cada uno de estos símbolos.
Por lo tanto, comenzamos con la declaración de nuestras estructuras dentro de las que encontramos la estructura **Iden** y posee atributos como los antes mencionados. Por lo tanto Iden nos ayuda a manejar a los identificadores como símbolos para ingresarlos en la tabla de símbolos.

Esto sirve para definir la estructura que poseerá finalmente nuestra tabla de símbolos, por lo que propusimos que los valores que retornarán las funciones adicionales retroalimentaran esta función para encontrar nuestros identificadores en la tabla de símbolos.

A partir de esta estructura, se planteó y se realizó la creación de una estructura llamada ListaS, en donde cada elemento obtenido tendrá las características de posición, nombre y tipo.

ListaS tendrá como atributo para indicar cual es el último y principal para manejar en qué posición vamos y el total.
Este indicador nos ayuda a obtener la posición en la que se encuentra nuestro identificador en la tabla de símbolos, así como para tener almacenados todos los identificadores.

El manejo necesario para las 3 tablas fue realizado de una manera semejante, es decir que llevamos la misma estructura en la creación de nuestras tablas de símbolos, tablas de cadenas y las de reales.

Como estructura de datos para el manejo del programa hicimos uso de las listas, pues consideramos que con esta estructura se tiene un mejor manejo de los datos y la información que ahí se contiene.

Es por esto que creamos un nodo para poder tener una lista de un tipo de dato en específico, para los símbolos de los identificadores utilizamos Iden como ya se mencionó y para las cadenas utilizamos nodoCadena.

Posteriormente, ocupamos estas funciones para el manejo de los datos para que cuando el programa encuentre un identificador, este la retorna para almacenarla en la tabla de símbolos. Esto se hace haciendo llamar a las funciones desde las definiciones. Como se muestra a continuación:

```
{identif} {printf("%s es un identificador", yytext);
           fprintf(archSal,"%s es un identificador (5,%d)\n", yytext,Tabla
           fprintf(tokens,"(5,%d)\n",TablaSimbolos(yytext));}
```

De esta manera entonces hacemos que cada que se encuentre un elemento a almacenar, se hace uso de una llamada a la función. Esta función tiene distintos usos, es por eso que se tiene como parámetro el argumento “yytext” y dependiendo de la tabla a utilizar se puede realizar una llamada a la función o no.

En el caso específico de la tabla de símbolos, recibimos el parámetro y se verifica que la integridad de la lista de símbolos fuera vacía, ya que si se encuentra en este estado entonces es posible asignar valores del identificador al nodo raíz de nuestra lista por medio del atributo asignado a la posición y dependiendo de esta misma posición, nos ayuda a contar el número de elementos en la lista y cada uno de los id de los elementos en la estructura de datos.

```
//Funcion que toma un identificador como entrada y lo agrega a la tabla de símbolos si es la primera
//vez que se encuentra. Si el identificador ya existe en la tabla de símbolos, se recupera su posición.
//Regresa 02 su no se encuentra en la tabla
int TablaSimbolos(char *identifi){
    int p =0;  //posicion
    if(im==0){ //Si es la primera vez que se llama a esta funcion se crea un nuevo identificador y se agrega a la tabla de simbolos
        struct Iden *iden0;
        iden0 = (struct Iden*)malloc(sizeof(struct Iden));
        iden0->next = NULL,
        iden0->posicion = Simbolos.total;
        strcpy(iden0->nombre, identifi);
        iden0->tipo = -1;
        Simbolos.raiz = iden0;
        Simbolos.ultimo = iden0;
        p = Simbolos.total;
        Simbolos.total++;
        im++;
        }
        else{ //Si no es la primera vez lo busca en la TS. Regresa -2 si no se enceuntra en la tabla, se crea un nuevo identificador y se agrega a la tabla
            p=buscarenTS(identifi);
                if(p==-2){
                    struct Iden *iden1;
                    iden1 = (struct Iden*)malloc(sizeof(struct Iden));
                    iden1->next = NULL;
                    iden1->posicion = Simbolos.total;
                    strcpy(iden1->nombre, identifi);
                    iden1->tipo = -1;
                    Simbolos.ultimo->next = iden1;
                    Simbolos.ultimo = iden1;
                    p=Simbolos.total;
                    Simbolos.total++;
                    im++;
                }
            }
                return p; //Devuelve la posicion del identificador en la tabla de simbolos
```

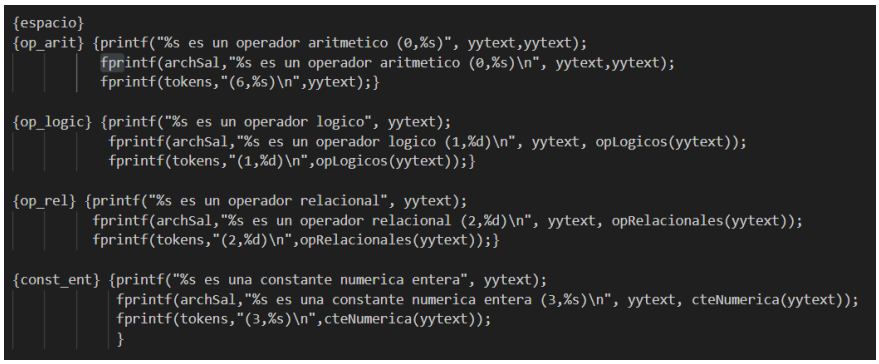
De esta forma, es por esto que el total de nodos que se encuentren hasta ese momento va a ser la posición del nodo actual, ya que es lo que se va a ingresar a los tokens. De esta manera entonces hicimos el llenado de las tablas para las literales de cadena.
Lo que se hace es entonces manipular una lista específica antes creada para asignar valores a los nodos dependiendo de en qué posición será, es por esto que las tablas las hicimos con esa lógica para proporcionar la posición del nodo y lo que se mandaría como token.

Para la tabla de símbolos es necesario que no se repitan, por lo que dentro de la estructura, cada vez que se tiene un símbolo nuevo, había que hacer una búsqueda. Esta se hizo de manera lineal en donde le damos un parámetro yytext y con un nodo auxiliar realizamos comparaciones internas para ver si alguno coincide con los nombres dentro, esto podemos observar en la siguiente captura:

```
// Recorre la lista ligada y regresa la posición de ocurrencia.
// Si no hay coincidencias regresa el valor -2.
int buscarenTS( char *ide){
    struct Iden *a = Simbolos.raiz;
    while (a!=NULL) {
        if(!strcmp(ide, a->nombre)){
            return a->posicion;
            }
            a = a->next;
            };
            return -2;
}
```

Si dicha búsqueda no daba resultado, lo que hace es asignar los valores del nodo actual de símbolos de la tabla de símbolos como el final, además que hacer que el siguiente quede desocupada para que le pueda entrar algún otro nodo posterior.
Esta misma lógica se repite de igual manera con las tablas de literales, solo que en esta última tabla si se tiene que realizar una búsqueda, ya que los tokens iban a ser distintos.

Finalmente para la parte de los tokens se llegó a la conclusión de que sería más óptimo y podríamos observar correctamente el resultado de dichos tokens si se arrojaban a un archivo. Por lo que declaramos una variable y una estructura para poder realizar la manipulación de los archivos de datos. Esto aplica para todas nuestras tablas, imprimiendo en el archivo lo siguiente:



```
{espacio}
{op_arit} {printf("%s es un operador aritmetico (0,%s)", yytext,yytext);
           fprintf(archSal,"%s es un operador aritmetico (0,%s)\n", yytext,yytext);
           fprintf(tokens,"(6,%s)\n",yytext);}

{op_logic} {printf("%s es un operador logico", yytext);
            fprintf(archSal,"%s es un operador logico (1,%d)\n", yytext, opLogicos(yytext));
            fprintf(tokens,"(1,%d)\n",opLogicos(yytext));}

{op_rel} {printf("%s es un operador relacional", yytext);
          fprintf(archSal,"%s es un operador relacional (2,%d)\n", yytext, opRelacionales(yytext));
          fprintf(tokens,"(2,%d)\n",opRelacionales(yytext));}

{const_ent} {printf("%s es una constante numerica entera", yytext);
             fprintf(archSal,"%s es una constante numerica entera (3,%s)\n", yytext, cteNumerica(yytext));
             fprintf(tokens,"(3,%s)\n",cteNumerica(yytext));
             }
```

Entonces haciendo uso del fprintf lo que se hizo fue escribir en nuestro archivo los tokens para almacenarlos y poder en un caso hipotético, pasarlos a un analizador sintáctico.

- ## Implementación

Haremos un resumen de la estructura de nuestro programa:

1.- Declaraciones e Includes

Al inicio del documento encontraremos las directivas y declaraciones de variables necesarias para la correcta implementación del analizador.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


FILE *archSal;  //Archivo para la salida
FILE *tokens;   //Archivo para los tokens

int im = 0;   //Bandera auxiliar en la tabla de simbolos
int im1 = 0;  //Bandera auxiliar en la tabla de literales para cadenas
int im2 = 0;
```

2.- Declaraciones de las Estructuras

Posteriormente encontramos las estructuras de nuestro programa, estas se declaran aquí debido a las buenas prácticas al momento de realizar un código en lenguaje c.
Se hacen uso de estas estructuras tanto para la manipulación de los nodos, los identificadores de la tabla de símbolos y la tabla de literales.

```c
//Estructura para los nodos. Tiene valor y clase
typedef struct nodo{
    int valor;
    char *clase;
    struct nodo *siguiente;
}Nodo;

//Estructura para manejar a los identificadores como simbolos para ingresa
struct Iden{
    struct Iden *next;
    int posicion;
    char nombre[63];
    int tipo;
};

//Estructura para la lista de simbolos y tener almacenados todos los ident
struct ListaS{
    struct Iden *raiz;
    int total;
    struct Iden *ultimo;
};

//Estructura para almacenar listas de cualquier tipo, se tiene un nodo apu
typedef struct lista{
    Nodo *raiz;
    int total;
    Nodo *ultimo;
}Lista;
```

3.- Declaraciones del Funcionamiento del Analizador

Encontramos también, las declaraciones del funcionamiento del analizador, por ejemplo nuestro operador lógico, el cual recibe como parámetro una cadena, esta cadena es el operador lógico identificado en la ER.

```
//Devuelve un entero correspondiente a la posicion del operador logico en el catalogo
//Recibe como parametro una cadena que es el operador logico identificado en la ER
//Si no enceuntra ninguna coincidencia devuelve -1
int opLogicos(char *opl){

    if (!strcmp(opl,"&&")){
        return 0;
    } else if (!strcmp(opl,"||")){
        return 1;
    } else if (!strcmp(opl,"!")){
        return 2;
    } else {
        return -1;
    }

```

4.- Definición de las expresiones regulares ER

En una parte más profunda de nuestro código, observamos la declaración en lex de nuestras expresiones regulares.

```
%}

salto_linea [\r\n]

op_arit     [+\-/*%]
op_logic    (&&)|(\|\|)|(!)
op_rel      ==|!=|>|<|>=|<=
const_ent   (\([-+][0-9]+\)|[0-9]+)
pal_res     assinado|caso|enquanto|fazer|flutuador|inteiro|para|quebrar|retorno|se|trocar
identif     _[a-zA-Z]+([a-zA-Z]|[0-9]|_)*
simb_esp    [(){};,:#\[|\]]
op_asig     "="|"+="|"-="|"*="|"-="|"*="|"/="|"%="
cadena      \"[^\"]*\"
op_cade     &|like

espacio    [" "]


%%
```

5.- Impresión de los resultados dentro del archivo resultante.

Como ya se comentó en la parte superior, el resultado se arroja por medio de la función fprintf a nuestro archivo que contiene una lista de los tokens con su respectiva clase y cadena.

```
{espacio}
{op_arit} {printf("%s es un operador aritmetico (0,%s)", yytext,yytext);
           fprintf(archSal,"%s es un operador aritmetico (0,%s)\n", yytext,yytext);
           fprintf(tokens,"(6,%s)\n",yytext);}

{op_logic} {printf("%s es un operador logico", yytext);
            fprintf(archSal,"%s es un operador logico (1,%d)\n", yytext, opLogicos(yytext));
            fprintf(tokens,"(1,%d)\n",opLogicos(yytext));}

{op_rel} {printf("%s es un operador relacional", yytext);
          fprintf(archSal,"%s es un operador relacional (2,%d)\n", yytext, opRelacionales(yytext));
          fprintf(tokens,"(2,%d)\n",opRelacionales(yytext));}

{const_ent} {printf("%s es una constante numerica entera", yytext);
             fprintf(archSal,"%s es una constante numerica entera (3,%s)\n", yytext, cteNumerica(yytext));
             fprintf(tokens,"(3,%s)\n",cteNumerica(yytext));
             }
```

6.- Función Principal

Finalmente, todo el código se pone en marcha debido a que la declaración de las funciones principales por ejemplo se encuentran en la parte inferior del programa, esta función es la que inicializa tanto el manejador de archivos como el que crea y manipula las listas con las que se trabaja.

```
//Funcion principal:
void main(int argn, char *argv[]){
    yyin = fopen(argv[1],"r");                          /
    archSal = fopen("salida.txt","w");                  /
                                                        /
    tokens = fopen("tokens.txt", "w");                  /
    fprintf(tokens, "Tokens generados: \n");            /

    /*s
    Generacion de listas ligadas para los catalogos de
    y op. asignacion
    */

    Lista *tablaPalRes = crearLista();
    Lista *tablaOpRel = crearLista();
    Lista *tablaOpLog = crearLista();
    Lista *tablaOpCadenas = crearLista();
    Lista *tablaOpAsig = crearLista();
```

- ## Indicaciones de como correr el programa

Dentro de una terminal de linux debemos estar en la ruta donde se encuentra nuestro archivo. Una vez hecho lo anterior escribimos los siguientes comandos:

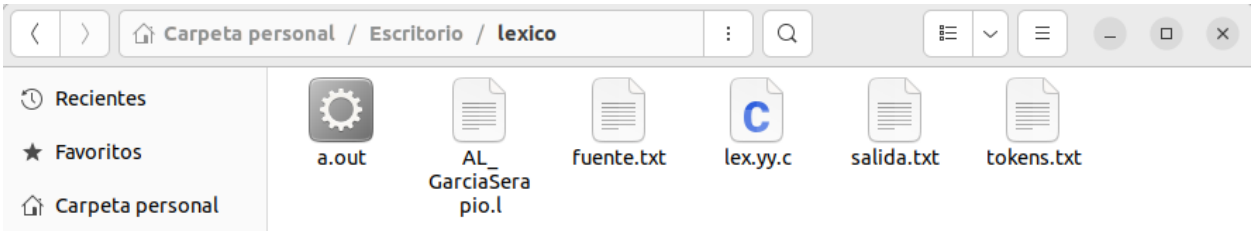
flex AL_GarciaSerapio.l
gcc lex.yy.c -lfl -o salida.out
./a.out <entrada.txt

entrada.txt es el nombre del archivo que va a analizar, puede llevar cualquier nombre siempre y cuando tenga la extensión “.txt”.

Ejemplo:

```
erik@erik-VirtualBox:~/Escritorio/lexico$ flex AL_GarciaSerapio.l
erik@erik-VirtualBox:~/Escritorio/lexico$ gcc lex.yy.c -lfl
erik@erik-VirtualBox:~/Escritorio/lexico$ ./a.out <fuente.txt
```

Una vez que se ejecute el programa imprimirá en pantalla todos los catálogos, después los reconocimientos que fue haciendo el analizador léxico, la tabla de símbolos y la tabla de literales.
Adicionalmente se crearán los archivos salida.txt y tokens.txt como se muestra a continuación.



- ## Conclusiones

  - ### García López Erik:

Realizar este programa fue útil para mi aprendizaje ya que al programar un analizador léxico pude entender a profundidad cómo funciona, abarcando desde la creación de las expresiones regulares, creación de las tablas (catálogos, símbolos, literales) e ir llenandolas según sus respectivos criterios, también crear los tokens.

A pesar de lo mencionado anteriormente, considero que fue difícil la elaboración de este programa principalmente porque tenía tiempo sin utilizar C entonces varias cosas ya se me había olvidado cómo manejar las estructuras, crear estructuras de datos (listas ligadas con todo y sus nodos), manejo de archivos entre otras cosas.

Según las pruebas que he realizado pienso que el programa funciona correctamente y ya está listo para entregar los tokens que recibe como entrada el analizador sintáctico.

  - ### Serapio Hernández Alexis Arturo:

Gracias al desarrollo de este programa pude comprender a profundidad el funcionamiento básico interno de un analizador léxico y el papel de este último en el diseño y uso en la mayoría de compiladores modernos.
Asimismo, aprendimos a manipular los errores que nos iban surgiendo con el fin de no “transmitir” este tipo de errores a nuestro próximo programa.
De igual manera creo que la parte de documentar el programa, nos ayudó a retroalimentarnos de igual manera a darle una última revisión general al programa para verificar que todo se cumpliera de manera satisfactoria.

Finalmente puedo concluir que el diseño y programación de este programa me ayudó a entender de una mejor manera la importancia de este analizador en todo el proceso de compilación y los elementos adquiridos a lo largo del desarrollo de este proyecto serían de mucha utilidad al momento del desarrollo de los programas posteriores.