	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Laboratorio de Computación Salas A y B

Profesor(a): René Adrián Dávila Pérez

Asignatura: Compiladores

Grupo: 04

No de Práctica(s): Analizador Léxico

Integrante(s): 318307921

318035327

318051363

318045351

No. de lista o brigada: Equipo 11

Semestre: 2024-1

Fecha de entrega: 05 de Octubre de 2023

Observaciones:

CALIFICACIÓN: _____

Índice

1. Introducción

En este trabajo se muestra el desarrollo de un analizador léxico que reconoce operaciones entre números reales y que además nos indiqué a que tipo de token pertenecen los caracteres ingresados en las operaciones, para ello, se plantea y se desarrolla el proceso de obtención de la gramática con base en un lenguaje creado por nosotros para determinar las reglas que definirán si una expresión es válida o no, todo esto con el apoyo de conocimientos de gramáticas libres de contexto y de autómatas finitos deterministas y no deterministas que nos ayudarán en la etapa posterior a este analizador el cual corresponde a la etapa del parsing.

La finalidad, es implementar el funcionamiento básico del analizador léxico en un lenguaje de programación, en este caso se realizará en el lenguaje C, haciendo uso de las bibliotecas básicas que se explicarán posteriormente en el desarrollo de este proyecto. Además, se verificará la realización correcta de este mediante pruebas que permitirán refutar la hipótesis que se plantea y se explicará la forma en que, mediante el código, se resolvieron los problemas presentados durante esta implementación.

1.1. Hipótesis

Realizar un analizador léxico de operaciones con números reales, debe reconocer expresiones que involucren números con o sin signo inicial, punto decimal obligatorio y por lo menos debe de reconocer al menos una de las cuatro operaciones aritméticas básicas entre los números reales de la expresión (pueden ser más de una), también debe tener la posibilidad de aceptar expresiones que contengan paréntesis, pero es importante que todo paréntesis que abra debe cerrarse. Dicho analizador será programado en lenguaje C. Para probarlo se introducirá una cadena que involucre las características mencionadas. Se realizará un conteo de tokens inválidos y se mostrará dónde se encuentran, así como, también se mostrará el tipo de token correspondiente para aquellos que se detectaron como válidos.

2. Desarrollo

2.1. Revisión Documental

2.1.1. ¿Qué es un analizador léxico?

Un analizador léxico es la primera fase de un compilador, su principal función es leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis.[1]

El analizador léxico toma un flujo de caracteres como entrada y los divide en tokens (unidades léxicas), que son estructuras de datos que representan unidades léxicas significativas para el lenguaje en cuestión. Los tokens pueden ser palabras clave, identificadores, números, operadores, signos de puntuación, etc. [1]

2.1.2. Bibliotecas Usadas

Las bibliotecas usadas para el desarrollo de nuestro código fueron las siguientes:

- ***stdio.h (Standard Input and Output Library)***

stdio.h es una biblioteca estándar en C que proporciona funciones y macros para realizar operaciones de entrada y salida en un programa. Estas operaciones incluyen leer datos del teclado, escribir datos en la pantalla y trabajar con archivos en el sistema de archivos.[3]

Funciones y Macros Relevantes:

- **printf():** Utilizada para imprimir texto y datos formateados en la salida estándar (generalmente en la pantalla).
- **scanf():** Utilizada para leer datos desde la entrada estándar (generalmente desde el teclado).
- **fopen(), fclose(), fread(), fwrite(), fprintf(), fscanf():** Funciones para trabajar con archivos, como abrir, cerrar, leer y escribir en archivos.
- **getchar(), putchar():** Funciones para leer y escribir caracteres individualmente.
- **fgets(), fputs():** Funciones para leer y escribir cadenas de caracteres (líneas) desde y hacia archivos o la entrada/salida estándar.

- ***stdlib.h (Standard Library)***

stdlib.h es una biblioteca estándar en C que proporciona funciones y macros relacionados con operaciones de propósito general, como gestión de memoria, control de procesos y funciones matemáticas.[3]

Funciones Relevantes:

- **malloc(), calloc(), realloc(), free():** Funciones para la gestión de memoria dinámica, que permiten asignar y liberar memoria en tiempo de ejecución.
- **exit():** Utilizada para finalizar la ejecución de un programa y devolver un código de salida al sistema operativo.

- **rand(), srand():** Funciones para generar números pseudoaleatorios.
- **system():** Ejecuta un comando del sistema operativo desde el programa.
- **atoi(), atof(), atol():** Funciones para convertir cadenas en valores numéricos.

❖ *stdbool.h (Boolean Type Library)*

stdbool.h es una biblioteca que se introdujo en el estándar C99 para proporcionar un tipo de dato booleano y valores true y false. Antes de su introducción, se usaban convenciones como 0 para falso y cualquier otro valor para verdadero.[3]

Tipo de Dato:

- bool: Un tipo de dato que puede tener dos valores: true o false. Se utiliza para representar valores booleanos en C.

Valores Relevantes:

- true: Representa el valor verdadero en una expresión booleana.
- false: Representa el valor falso en una expresión booleana.

❖ *string.h (String Library)*

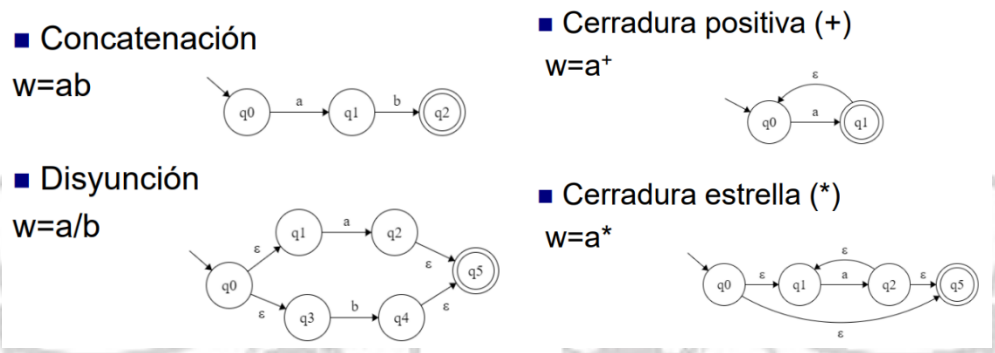
string.h es una biblioteca estándar en C que proporciona funciones y macros para trabajar con cadenas de caracteres (strings). Estas funciones permiten la manipulación, comparación y búsqueda de cadenas, entre otras operaciones.[3]

Funciones Relevantes:

- **strlen():** Calcula la longitud de una cadena (número de caracteres).
- **strcpy(), strncpy():** Copian una cadena en otra.
- **strcat(), strncat():** Concatenan una cadena al final de otra.
- **strcmp(), strncmp():** Comparan dos cadenas y devuelven un valor que indica su relación (igual, mayor o menor).
- **strstr():** Busca una subcadena dentro de una cadena.
- **strtok():** Divide una cadena en tokens (fragmentos) basados en un delimitador.
- **memset(), memcpy():** Funciones para establecer valores de bytes en memoria o copiar bloques de memoria.

2.1.3. Gramáticas, AFND Y AFD

Recordando la metodología de Thompson se compone de la siguiente manera



Eliminación de transiciones ε

1.Buscar un nodo del que salga una transición con ε. Si lo hay se finaliza.

2. Siga esa transición hasta el siguiente nodo y continue a través de las transiciones ε hasta llegar a un nodo del que ya no salgan transiciones con ε. Si no existe tal nodo pase al inciso 8.

3.Llame A y B a los dos últimos nodos en la secuencia anterior.

4. Si hay una secuencia de transiciones etiquetadas ε desde un nodo inicial hasta A, haga de B un nodo inicial, o si A es un nodo inicial haga de B un nodo inicial, o si A es un nodo inicial haga de B un nodo inicial.

5. Si B es un estado final, haga de A un estado final.

6. Elimine la transición ε que va del nodo A al nodo B.

7. Para cada transición que parte de B a un nodo C etiquetada con un símbolo del alfabeto agregar una transición desde A hasta C con el símbolo de B a C. Regrese al inciso 1.

8. Si no existe un nodo del tipo buscado, entonces debe existir una cadena cerrada de la siguiente forma:

$$A_1 \rightarrow A_2 \rightarrow A_n \rightarrow \ldots \rightarrow A_1$$

9.Haga un solo nodo con todos los nodos que forman la cadena cerrada y elimine las transiciones ε restantes.

10. Si algún estado A_n es final, entonces el nuevo estado también es final.

11. Si algún estado A_n es inicial, entonces el nuevo estado también es inicial. [2] Regrese al inciso 1.

Obtención de un AFD a partir de un AFND sin ε

1.El AFD tiene el mismo alfabeto de entrada que el AFND.

2.Etiqueta el primer renglón de la tabla del AFD con todos los estados iniciales del AFND.

3.Tomar de esta tabla un renglón para el que no se han calculado los estados siguientes y calcularlos, empleando los símbolos de entrada a cada estado del conjunto que etiqueta a ese renglón, obteniéndose así el conjunto de estados siguientes.

4.Si entre los conjuntos de estados siguientes hay algunos que no corresponda a la etiqueta de un renglón, úselo para etiquetar un nuevo renglón.

5.Repita desde el paso 3 hasta que no existan más renglones nuevos.

6.Los renglones de la tabla se marcan como estados de aceptación, si alguno de los estados que etiquetan ese renglón es de aceptación, en caso contrario es de rechazo.

7.Renombre los estados actuales y obtenga la tabla de transición con los nuevos estados siguientes.

8.La tabla resultante es la del AFD, dibuje al autómata. [2]

Obtención de un AFD mínimo

1.Partir de un AFD.

2. Separar los estados en dos grupos, por un lado, los de aceptación y por otro los de rechazo.

3. Aplicar los símbolos de entrada a cada estado y separar en nuevos grupos de acuerdo con el o los estados a los que pertenecen los estados siguientes.

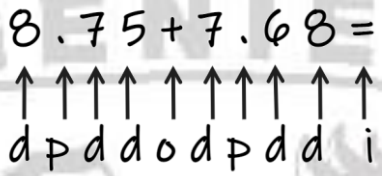
4. Repetir el paso 3 hasta que no se generen nuevos conjuntos.

5. Los estados agrupados en cada conjunto pueden ser sustituidos por uno sólo. [2]

2.2. Gramática.

2.2.1. Desarrollo de la Expresión Regular

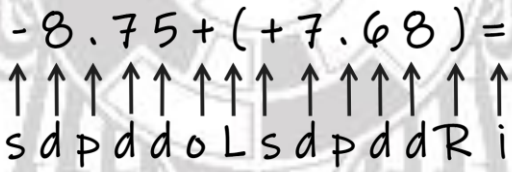
Para el desarrollo de la expresión regular debemos de analizar la estructura de una operación la cual es la siguiente:



Donde podemos definir los siguientes conjuntos:

$$d = \{0,1,2,3,4,5,6,7,8,9\}$$
$$o = \{+, -, *, /\}$$
$$p = \{.\}$$
$$i = \{=\}$$

Sin embargo, podemos agregar más complejidad a la estructura de nuestra operación



Finalmente, nuestros conjuntos quedan de la siguiente manera

$$d = \{0,1,2,3,4,5,6,7,8,9\}$$
$$o = \{+, -, *, /\}$$
$$p = \{.\}$$
$$i = \{=\}$$
$$s = \{+, , -\}$$
$$L = \{(\}$$
$$R = \{)\}$$

Considerando las restricciones para desarrollar la expresión regular las cuales son:

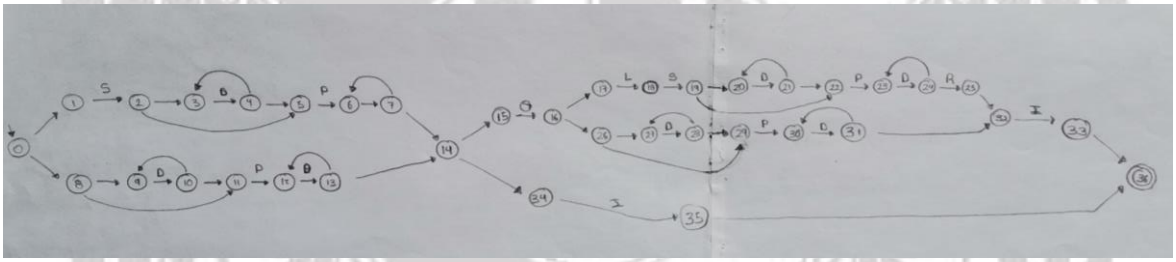
- Podemos recibir una cifra e inmediatamente el signo de igual, este caso es correcto.
- Si recibimos algún signo después del operador este debe de estar entre paréntesis para que la expresión sea correcta.
- Si recibimos paréntesis y no recibimos un signo este caso es incorrecto.
- Debemos recibir más de 2 operaciones entre números reales.
- Los números deben de ser reales, esto lo damos a entender que llevan punto y dígitos decimales.

Por lo que nuestra expresión queda de la siguiente manera:

$$(S\,D^*\,P\,D+ \mid D^*\,P\,D+)(I \mid [O\ (L\,S\,D^*\,P\,D+R \mid D^*\,P\,D+]+\ I)$$

2.2.2. Obtención del AFND mediante la metodología de Thompson

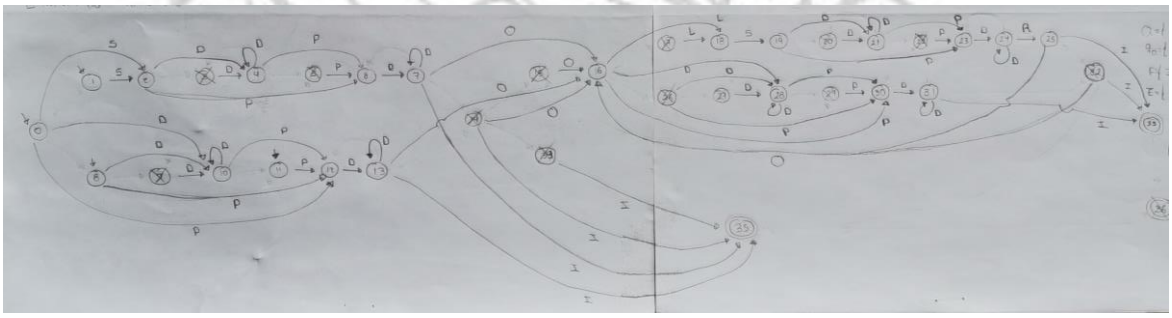
Obtenemos el siguiente Autómata Finito No Determinístico con transiciones vacías



$$Q = \{0,1,2,3,4,5,6,7,8,9,\ldots,36\}$$
$$q0 = \{0\}$$
$$Z = \{S,D,P,I,L,R\}$$
$$F = \{36\}$$

2.2.3. Eliminación de Transiciones vacías del AFND

Obtenemos el siguiente Autómata Finito No Determinístico sin transiciones vacías ξ



$Q = \{0,1,2,3,4,5,6,7,8,9,\ldots,36\}$

$q0 = \{0,1,8,11\}$

$\Sigma = \{S,D,P,I,L,R\}$

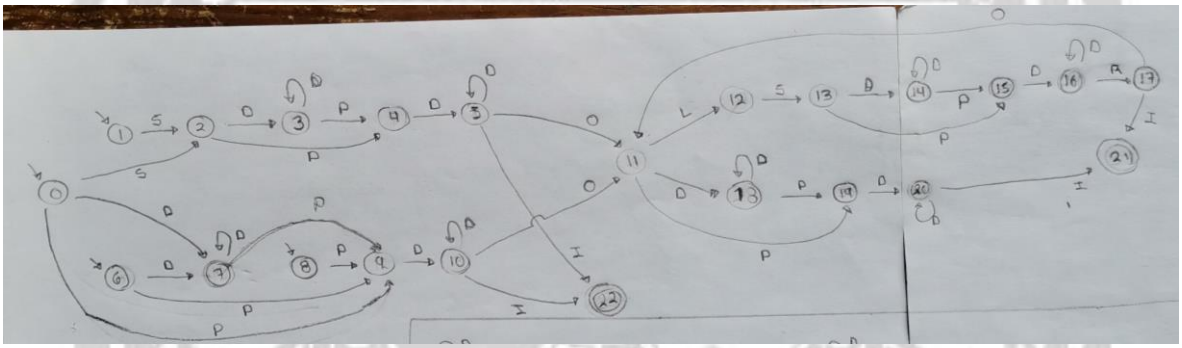
$F = \{33,35,36\}$

Nodos Extraños =

$\{36,32,20,29,17,27,26,15,14,34,5,3,9,22\}$

2.2.4. Pasar de un AFND sin transiciones vacías a AFD

Renombramos el número de estados y volvemos a dibujar el autómata

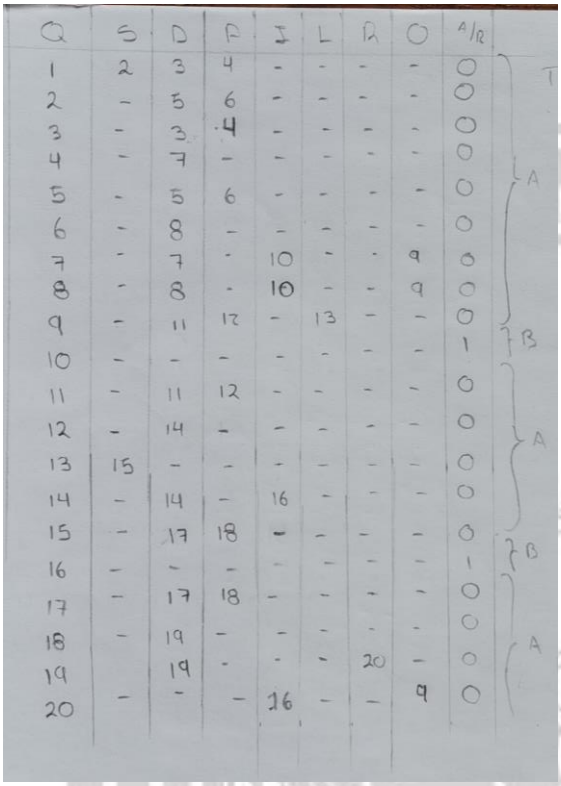


Obteniendo la tabla del AFD:

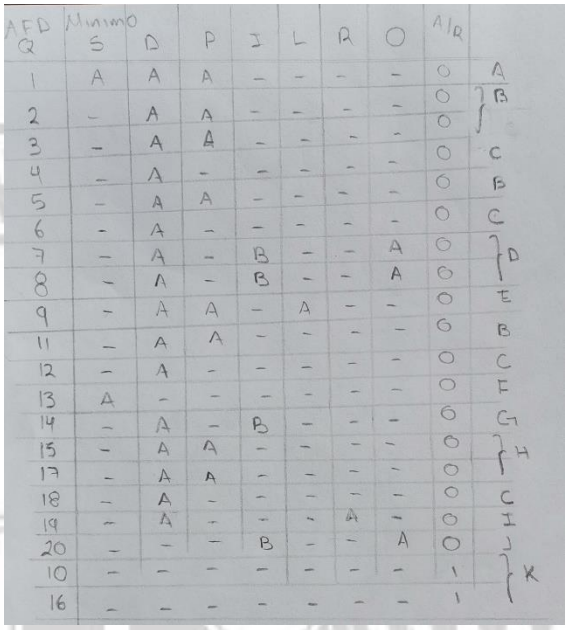
	Q	S	D	P	I	L	R	O	A/R
1	0,6,18	2	7	9	-	-	-	-	0
2	2	-	3	4	-	-	-	-	0
3	7	-	7	9	-	-	-	-	0
4	9	-	10	-	-	-	-	-	0
5	3	-	3	4	-	-	-	-	0
6	4	-	5	-	-	-	-	-	0
7	10	-	10	-	22	-	-	11	0
8	5	-	5	-	22	-	-	11	0
9	11	-	18	19	-	12	-	-	0
10	22	-	-	-	-	-	-	-	1
11	18	-	18	19	-	-	-	-	0
12	19	-	20	-	-	-	-	-	0
13	12	13	-	-	-	-	-	-	0
14	20	-	20	-	21	-	-	-	0
15	13	-	14	15	-	-	-	-	0
16	21	-	-	-	-	-	-	-	1
17	14	-	14	15	-	-	-	-	0
18	15	-	16	-	-	-	-	-	0
19	16	-	16	-	-	-	17	-	0
20	17	-	-	-	21	-	-	11	0

2.2.5. Minimizar el AFD

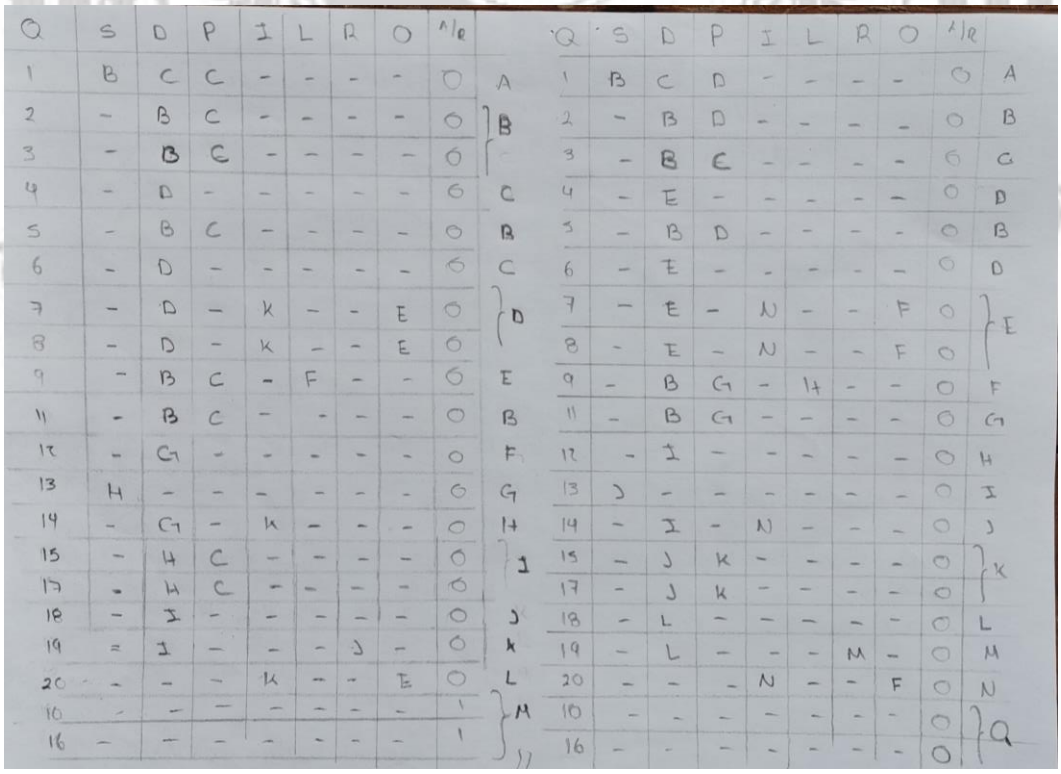
Obteniendo la tabla del AFD mínimo y su representación obtenemos:



Q	S	D	P	I	L	R	O	A/R	
1	2	3	4	-	-	-	-	0	T
2	-	5	6	-	-	-	-	0	
3	-	3	4	-	-	-	-	0	
4	-	7	-	-	-	-	-	0	A
5	-	5	6	-	-	-	-	0	
6	-	8	-	-	-	-	-	0	
7	-	7	-	10	-	-	9	0	
8	-	8	-	10	-	-	9	0	
9	-	11	12	-	13	-	-	0	
10	-	-	-	-	-	-	-	1	B
11	-	11	12	-	-	-	-	0	
12	-	14	-	-	-	-	-	0	A
13	15	-	-	-	-	-	-	0	
14	-	14	-	16	-	-	-	0	
15	-	17	18	-	-	-	-	0	
16	-	-	-	-	-	-	-	1	B
17	-	17	18	-	-	-	-	0	
18	-	19	-	-	-	-	-	0	
19		19	-	-	-	20	-	0	A
20	-	-	-	16	-	-	9	0	



AFD Mínimo Q	S	D	P	I	L	R	O	A/R	
1	A	A	A	-	-	-	-	0	A
2	-	A	A	-	-	-	-	0	B
3	-	A	A	-	-	-	-	0	
4	-	A	-	-	-	-	-	0	C
5	-	A	A	-	-	-	-	0	B
6	-	A	-	-	-	-	-	0	C
7	-	A	-	B	-	-	A	0	D
8	-	A	-	B	-	-	A	0	
9	-	A	A	-	A	-	-	0	E
11	-	A	A	-	-	-	-	0	B
12	-	A	-	-	-	-	-	0	C
13	A	-	-	-	-	-	-	0	E
14	-	A	-	B	-	-	-	0	G
15	-	A	A	-	-	-	-	0	H
17	-	A	A	-	-	-	-	0	
18	-	A	-	-	-	-	-	0	C
19	-	A	-	-	-	A	-	0	I
20	-	-	-	B	-	-	A	0	J
10	-	-	-	-	-	-	-	1	K
16	-	-	-	-	-	-	-	1	

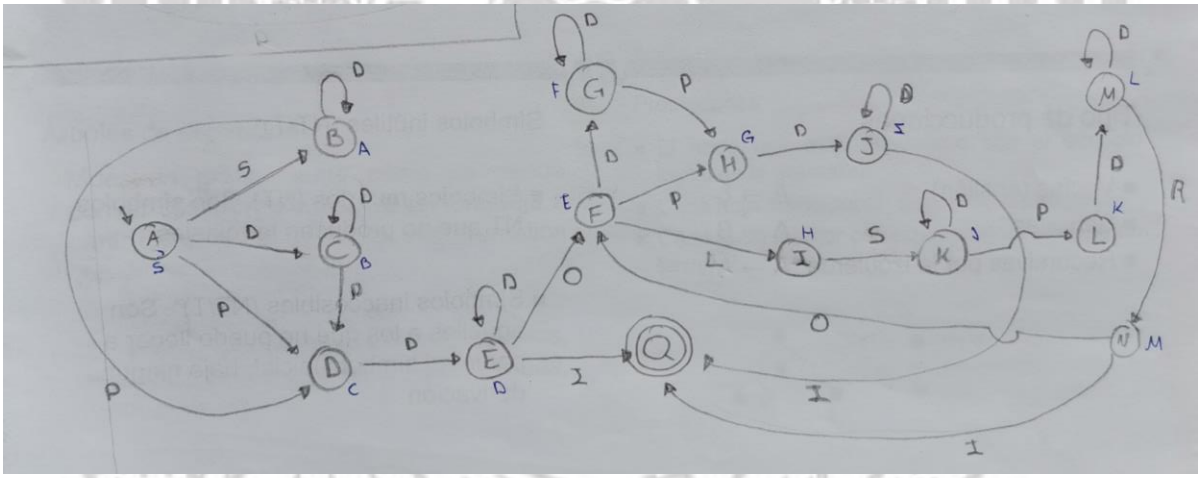


Q	S	D	P	I	L	R	O	A/R	
1	B	C	C	-	-	-	-	0	A
2	-	B	C	-	-	-	-	0	B
3	-	B	C	-	-	-	-	0	
4	-	D	-	-	-	-	-	0	C
5	-	B	C	-	-	-	-	0	B
6	-	D	-	-	-	-	-	0	C
7	-	D	-	K	-	-	E	0	D
8	-	D	-	K	-	-	E	0	
9	-	B	C	-	F	-	-	0	E
11	-	B	C	-	-	-	-	0	B
12	-	G	-	-	-	-	-	0	F
13	H	-	-	-	-	-	-	0	G
14	-	G	-	K	-	-	-	0	H
15	-	H	C	-	-	-	-	0	I
17	-	H	C	-	-	-	-	0	
18	-	I	-	-	-	-	-	0	J
19	-	I	-	-	-	J	-	0	K
20	-	-	-	K	-	-	E	0	L
10	-	-	-	-	-	-	-	1	M
16	-	-	-	-	-	-	-	1	

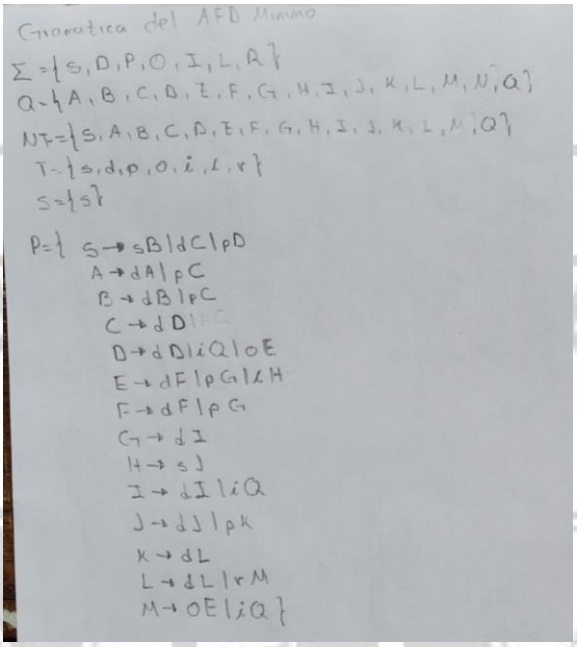
Q	S	D	P	I	L	R	O	A/R	
1	B	C	D	-	-	-	-	0	A
2	-	B	D	-	-	-	-	0	B
3	-	B	C	-	-	-	-	0	C
4	-	E	-	-	-	-	-	0	D
5	-	B	D	-	-	-	-	0	B
6	-	E	-	-	-	-	-	0	D
7	-	E	-	N	-	-	F	0	E
8	-	E	-	N	-	-	F	0	
9	-	B	G	-	H	-	-	0	F
11	-	B	G	-	-	-	-	0	G
12	-	I	-	-	-	-	-	0	H
13	J	-	-	-	-	-	-	0	I
14	-	I	-	N	-	-	-	0	J
15	-	J	K	-	-	-	-	0	K
17	-	J	K	-	-	-	-	0	
18	-	L	-	-	-	-	-	0	L
19	-	L	-	-	-	M	-	0	M
20	-	-	-	N	-	-	F	0	N
10	-	-	-	-	-	-	-	0	Q
16	-	-	-	-	-	-	-	0	

Q	S	D	P	I	L	A	O	
A	B	C	D	-	-	-	-	A
2	-	B	D	-	-	-	-	B
3	-	C	B	-	-	-	-	C
4	-	E	-	-	-	-	-	D
5	-	B	D	-	-	-	-	B
6	-	E	-	-	-	-	-	D
7	-	E	-	Q	-	-	F	E
8	-	E	-	Q	-	-	F	
9	-	G	H	-	I	-	-	F
11	-	G	H	-	-	-	-	G
12	-	J	-	-	-	-	-	H
13	K	-	-	-	-	-	-	I
14	-	J	-	Q	-	-	-	J
15	-	K	L	-	-	-	-	K
17	-	K	L	-	-	-	-	
18	-	M	-	-	-	-	-	L
19	-	M	-	-	-	N	-	M
20	-	-	-	Q	-	-	F	N
10	-	-	-	-	-	-	-	Q
16	-	-	-	-	-	-	-	

Por lo tanto, nuestro AFD mínimo queda de la siguiente manera:



2.2.6. Gramática del AFD Mínimo



Gramatica del AFD Minimo

$\Sigma = \{S, D, P, O, I, L, R\}$

$Q = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, Q\}$

$NT = \{S, A, B, C, D, E, F, G, H, I, J, K, L, M, Q\}$

$T = \{s, d, p, o, i, l, r\}$

$S = \{s\}$

P={ S → sB | dC | pD
A → dA | pC
B → dB | pC
C → dD | pC
D → dD | iQ | oE
E → dF | pG | lH
F → dF | pG
G → dI
H → sJ
I → dI | iQ
J → dJ | pK
K → dL
L → dL | rM
M → oE | iQ }

2.2.7. Obtener la tabla de transiciones del AFD

Q	S	D	P	I	L	R	O	A|R
A	B	C	D	-	-	-	-	0
B	-	B	D	-	-	-	-	0
C	-	C	D	-	-	-	-	0
D	-	E	-	-	-	-	-	0
E	-	E	-	Q	-	-	F	0
F	-	G	H	-	I	-	-	0
G	-	G	H	-	-	-	-	0
H	-	J	-	-	-	-	-	0
I	K	-	-	-	-	-	-	0
J	-	J	-	Q	-	-	-	0
K	-	K	L	-	-	-	-	0
L	-	M	-	-	-	-	-	0
M	-	M	-	-	-	N	-	0
N	-	-	-	Q	-	-	F	0
Q	-	-	-	-	-	-	-	1

2.3. Fragmentos Importantes

Se declara un puntero a un archivo y se realiza la apertura con permisos de lectura, en caso de que no se pueda abrir el archivo o no exista deberá arrojar “No se logró abrir el archivo” en caso de que fuera exitoso continua el programa de forma normal.

```
1  int main(int argc, char *argv[]){
2      FILE *archivo;                          //Declaraci□n de un puntero a un archivo
3      char exp[100][1000];                    // Matriz para almacenar las lineas del archivo
4      char linea[10000];                      // Tama□o del buffer para una linea
5
6      archivo = fopen("archivo.txt", "r");    //Apertura del Archivo "archivo.txt" en modo lectura
7
8      /*Validaci□n para verificar si se pudo abrir el archivo en caso contrario
9      mostrara el mensaje de "No se pudo abrir el archivo"*/
10
11
12     if (archivo == NULL) {
13         printf("No se pudo abrir el archivo.\n");
14         return 1;                           //Se regresa 1 para indicar un error
15     }
```

En el siguiente fragmento se realiza la lectura de cada línea de nuestro archivo que tendrá las expresiones que pasaremos por el analizador léxico.

```
1  // Bucle que se ocupa para leer y procesar cada l□nea del archivo
2          while (fgets(linea, sizeof(linea), archivo)) {
3              // Buscar el salto de l□nea en la l□nea
4              char *salto = strchr(linea, '\n');
5
6              if (salto != NULL) {
7                  // Reemplazar el salto de l□nea con un terminador nulo
8                  *salto = '\0';
9
10                 // Copiar la l□nea a la matriz exp en la fila i
11                 strcpy(exp[cont1], linea);
12                 len1=strlen(exp[cont1]);
13                 if( len1 == 0 ){ break; }
14                 printf("\n=============================================================================================");
15                 printf("\n\t\t\t\tEJECUTANDO LA EXPRESI%cN #%d\n", 224, cont1+1);
16                 printf( "La expresi%cn es: %s\n", 162, exp[cont1] );
17
```

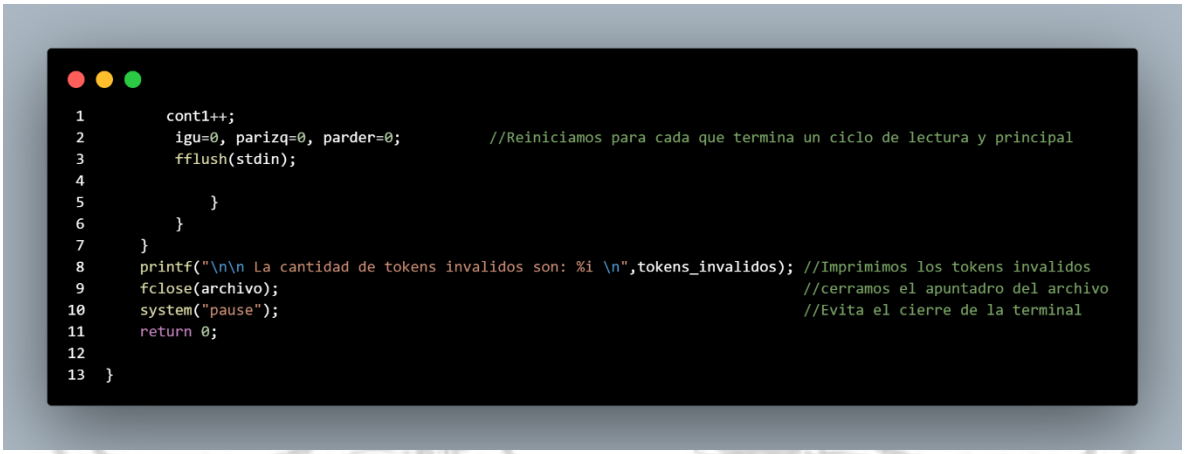
En la línea 6 para que se distinga una cadena de otra, usamos el salto de línea como un terminador de cadena. En la línea 11 se copia el contenido de “línea” en nuestra matriz “exp”. Esto se hace para su posterior procesamiento.

```
//Caso especial para cuando la expresi□n inicia en punto
        if( exp[cont1][0] == 46 ){
            printf("\n\t La cadena est%c formada por:\n\t\tp ", 160);
            if( len1 == 1 ){//Si solo es el punto es invalido
                printf("X<-- Se esperaba '0..9'--\n", 162, 160);
                tokens_invalidos++;
                break;
            }
            for( int j = 1; j < len1-1; j++){ //Ciclo para validar los digitos
                if( exp[cont1][j] >= 48 && exp[cont1][j] <=57 ){    //Puede haber 1 o m□s digitos
                    printf("d ");
                    if( exp[cont1][j+1] == 61 ){ //Debe terminar en igual
                        printf("i \n\n");
                    }
                }else{  //No puede tener otro terminal
                    printf("X <-- Se esperaba '0..9','=' \n", 162, 160);
                    tokens_invalidos++;
                    break;
                    }

            }
            if( exp[cont1][len1-1] != 61 ){ //Debe terminar en igual
                printf("X--Se esperaba '=' \n", 162, 160);
                tokens_invalidos++;
                }
            break;
        }
```

En este caso analizamos si nuestra expresión inicia en un punto “.” Entonces solamente acepta un número real y el signo igual; si es diferente entonces lo marca como error.

```
if(len1<8){ //La cadena m□s corta que acepta es con longitud=8, por lo que si es m□s chica no se acepta
        }
        else{
                //Si se acepta la cadena, comienza a contar la cantidad de iguales (s□lo debe haber uno al final de la expresi□n),
                //parentesis izquierdos y derechos, de estos debe haber la misma cantidad
            for (i1=0; i1<len1; i1++){
                if(exp[cont1][i1]==61){
                    igu++;
                }
                if(exp[cont1][i1]==40){
                    parizq++;
                }
                if(exp[cont1][i1]==41){
                    parder++;
                }
            }
            printf("\nSe encontraron %i iguales, %i parentesis izquierdos y %i parentesis derechos\n", igu, parizq, parder);

```

Si la cadena que se analiza en su momento es de longitud mayor a 8 entonces verificamos que existan signo igual y paréntesis tanto izquierdo como derecho. Esta parte es parte de las pruebas del código, pero decidimos dejarlo.



```
        cont1++;
         igu=0, parizq=0, parder=0;          //Reiniciamos para cada que termina un ciclo de lectura y principal
         fflush(stdin);

             }
         }
     }
     printf("\n\n La cantidad de tokens invalidos son: %i \n",tokens_invalidos); //Imprimimos los tokens invalidos
     fclose(archivo);                                                            //cerramos el apuntadro del archivo
     system("pause");                                                            //Evita el cierre de la terminal
     return 0;

}
```

El código anterior muestra el cierre del puntero del archivo y la impresión de la cantidad de tokens inválidos.

3. Conclusión

Un analizador léxico es esencial para reconocer y clasificar los componentes léxicos de una cadena de entrada en un lenguaje de programación específico. El uso de herramientas como Flex facilitaría el trabajo, sin embargo, en este caso no se utilizó, esto para que podamos entender de mejor manera como funciona y cómo se elabora un analizador, asimismo, para que al momento de desarrollar el proyecto final del curso la carga de trabajo sea menor.

Para el caso específico de este analizador léxico: e identifican operaciones con números reales. El analizador es capaz de reconocer expresiones que involucren números reales con o sin signo inicial, un punto decimal obligatorio para asegurar que es un número real y al menos una de las cuatro operaciones aritméticas básicas entre los números reales, también permite ingresar paréntesis en la expresión considerando que todo paréntesis que abre debe cerrarse. La construcción del analizador léxico también toma en cuenta la gramática regular utilizada, los tokens válidos y los inválidos, el AFD (Autómata Finito Determinista) y las acciones semánticas.

Por todo lo anterior, se concluye que la hipótesis de este proyecto si se cumple, además de que servirá como una buena base para poder realizar el proceso de parsing en el siguiente analizador: el semántico.

4. Referencias

- [1] L.S.C. Monroy Cedillo Jair Jonathan (s.f.). Autómatas y Compiladores (Unidad 2) [Online]. Available: http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/21_funcin_del_analizador_lxico.html
- [2] "Gramáticas regulares y autómatas finitos", class notes for 0442, División de Ingeniería Eléctrica, Universidad Nacional Autónoma de México, Winter, 2022.
- [3] M. A. Vine, *C programming for the absolute beginner*. Course Technology Ptr, 2007.