Universidad Nacional Autónoma de México

Facultad de Ingeniería

División de Ingeniería Eléctrica

Compiladores

Profesor: Sandoval Montaño Laura Ing.

Semestre 2019-1

"Analizador Sintáctico"

Grupo: 02

Elaborado por:

- **Corella Pérez Elda**
- **Moreno Guerra Marco Antonio**

Cd. Universitaria a 25 de octubre del 2018.

ÍNDICE

PARTE 2.

ANALIZADOR SINTÁCTICO EN C

1. Descripción del problema

Se necesita implementar un analizador Léxico y Sintáctico en lenguaje C, apoyándonos de diversas herramientas (Flex y Estructuras de datos). El analizador recibirá un archivo por línea de comandos como entrada. Lo primero que debe de realizar es el analisis lexico, dentro de este análisis se deben indicar los errores léxicos que se vayan encontrando así como la línea del programa en donde se encontró dicho error, posteriormente se generarán tres archivos de salida, uno con la lista de los *identificadores*, el segundo con las *constantes cadenas* que contiene el programa de entrada y el archivo que contiene la ***tabla de tokens***, finalmente el analizador léxico debe de generar una **cadena de átomos**. La cadena de átomos será la entrada del analizador sintáctico.

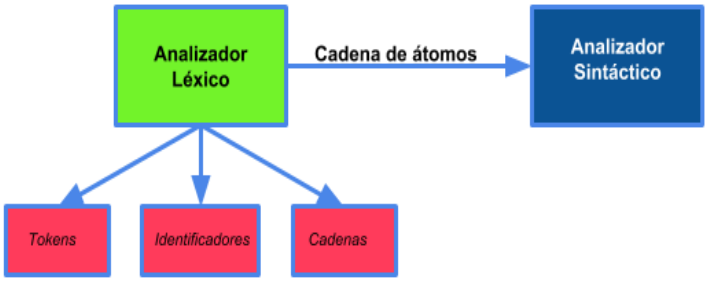


Imagen 1. Flujo de datos en el programa

1.1 Características de la cadena de átomos

La cadena de átomos debe estar basada en la generación, de los tokens. Cada que se encuentre alguna clase se le asignará un átomo diferente. La cadena debe de ir concatenando los átomos que vaya reconociendo en la lectura del programa de entrada. A continuación en la tabla 1, tabla 2, y 3. Se muestran los respectivos átomos que se deben de colocar, dependiendo de la clase.

Tabla No.1 Átomos correspondientes a las diferentes clases

Clase	Descripción	Átomo(s)
0	Palabras reservadas	***VER TABLA 2***
1	Identificadores	a
2	Símbolos especiales	( ) , ; [ ]
3	Operador de asignación	=
4	Operadores relacionales	***VER TABLA 3***
5	Operadores aritméticos	+ - * / %
6	Constantes cadenas	s
7	Constantes enteras	n
8	Constantes reales	r

Tabla No.2 Átomos para la clase de Palabras Reservadas		
Valor	**Palabra Reservada**	**Átomo**
0	Bul	b
1	Cadena	c
2	Cierto	t
3	Entero	e
4	Falso	f
5	Haz	h
6	Mientras	m
7	Para	p
8	Real	d
9	Si	i
10	Sino	o

Tabla No. 3 Átomos para los Operadores Relacionales		
Valor	**Operador Relacional**	**Átomo**
0	.DIF.	!
1	.IGL.	q
2	.MN.	<
3	.MNI.	l
4	.MY.	>
5	.MYI.	g

** La cadena de átomos además tendrá al final una concatenación con el símbolo de fin de cadena, para nuestro caso el símbolo de fin de cadena fue representado por un ***'&"***

* * Es importante que se tenga en cuenta que para las clases, 2, 3 y 5. ***El átomo será el propio carácter.***

2. Propuesta de solución

Para poder desarrollar el analizador léxico, y cumplir todas los requerimientos establecidos, llegamos a las siguientes propuestas:

- Uso del lenguaje de programación C, basado en el estándar ANSI C.
- Uso la herramienta Flex de C, y listas ligadas como nuestra principal estructura de datos.
- La metodología de trabajo seleccionada fue nuevamente la **Kanban.** Pues al ver que se obtuvieron buenos resultados en la parte 1 del programa (analizador Léxico), seguir con la misma metodología es lo más eficiente.
- Puesto que el tiempo del desarrollo de software es muy corto se propuso presentar un solo tablero para visualizar el flujo de trabajo a mediados del desarrollo del proyecto,y así afinar detalles, además se identifican por colores las actividades con prioridad alta y baja.

3. Fases del desarrollo

Para poder tener una implementación adecuada tanto del analizador léxico como el sintáctico en un solo programa y que ambos cumplan los objetivos establecidos, es necesario llevar un estricto control de las estructuras de datos a utilizar, así como considerar las alternativas más convenientes para la solución del problema.

3.1 Análisis

Una parte fundamente en la solución de cualquier problema , es detectar las actividades más relevantes y distribuirlas entre los integrantes del equipo para así tener un trabajo más organizado y completo. En la tabla de actividades destacadas, se puede visualizar dicha distribución.

Tabla de Actividades Detectadas		
ANÁLISIS		
	Actividad 1	Identificar aquellas producciones y No Terminales que sean vacíos.
	Actividad 2	Realizar el first de cada producción (Todo el equipo)
	Actividad 3	Realizar el follow de cada uno de los No Terminales de la gramática definida.
	Actividad 4	Conjunto Selección de cada producción
	Actividad 5	Definir las variables auxiliares globales
	Actividad 6	Definir las funciones globales auxiliares
	Actividad 7	Definir estructura de datos para manejar cadena de átomos
	Actividad 8	Implementar el código para la cadena de átomos
	Actividad 9	Definir funciones necesarias para el analizador Sintáctico
	Actividad 10	Implementación de las funciones para el análisis sintáctico

Corella Perez Elda

Moreno Guerra Marco

	Actividad 11	**Implementación de la detección de errores sintácticos**
	Actividad 12	**Creación De funciones de impresión de errores sintácticos**
	Actividad 13	**Documentar el código Final**
	Actividad 14	**Realizar trabajo de Documentación**
	Actividad 15	**Realizar pruebas de escritorio (todo el equipo)**

A continuación se muestran los tableros que se generaron a lo largo del proyecto, cada uno con una diferencia de 3 días.

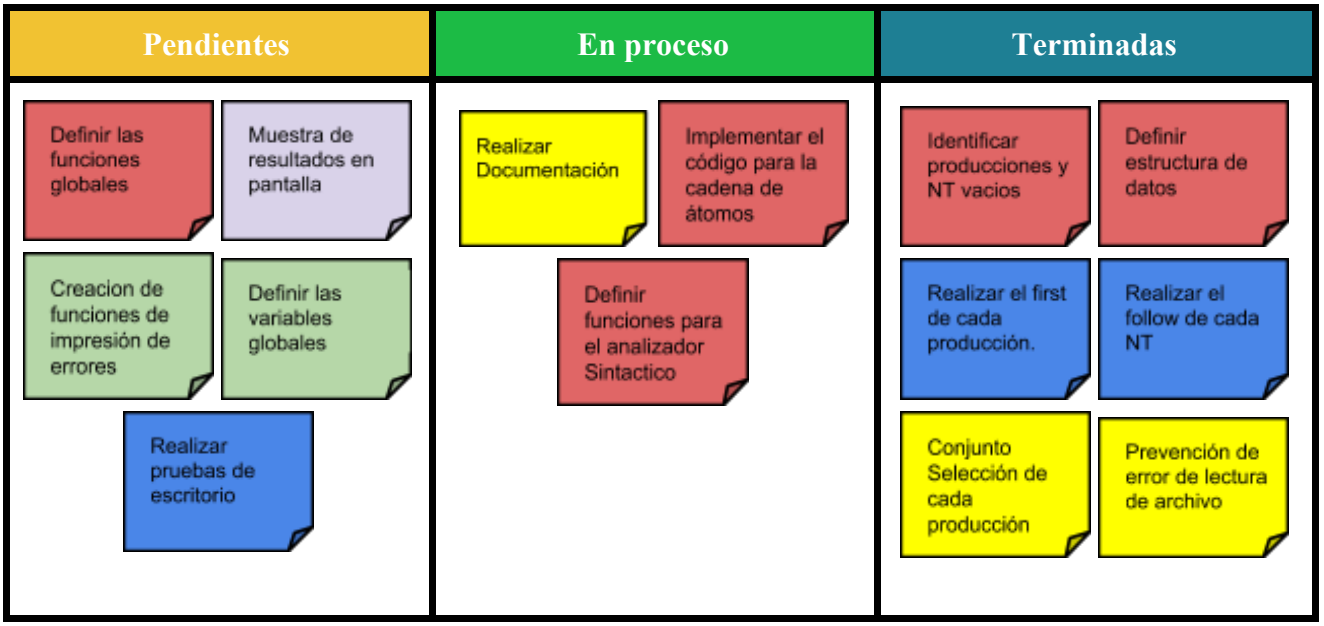


Figura No. 2
Tablero generado al uso de la metodología Kanban

3.2 Diseño

Antes de comenzar con la programación, es importante comprobar que la gramática de nuestro lenguaje,sea LL(1). Ya que solo una gramática LL(1) puede ser analizada con los algoritmos que se implementarán.

La gramática de nuestro lenguaje se puede visualizar en la tabla 4

Tabla No. 4 Gramática para el lenguaje Pu+

1.	G → [Z]	19.	S → I	37.	O → g
2.	Z → DZ	20.	A → a= K	38.	E→ TE’
3.	Z → ε	21.	H → h[Y]m(R);	39.	E’ → +TE’
4.	Z → Y	22.	M → m(R)[Y]	40.	E’→ -TE’
5.	Y →SX	23.	P → p(A;R;A)[Y]	41.	E’→ ε
6.	X →Y	24.	I → i(R)[Y]N	42.	T→ FT’
7.	X →ε	25.	N → ε	43.	T’→ *FT’
8.	D→JaV	26.	N → o[Y]	44.	T’→ /FT’
9.	J→b	27.	K → s	45.	T’→ %FT’
10	J→c	28.	K → E	46.	T’→ ε
11.	J→e	29.	R → EQ	47.	F→ (E)
12.	J→d	30.	Q→ OE	48.	F→ a
13.	V→,aV	31.	Q → ε	49.	F→ n
14.	V→;	32.	O → !	50.	F→ r
15.	S→A;	33.	O → q	51.	K→ t
16	S→H	34.	O → <	52.	K → f
17.	S→M	35.	O → l		
18.	S→P	36.	O → >		

Para poder calcular los Conjuntos de selección de cada una de las producciones de la tabla No. 4, debes seguir una serie de pasos:

a) Producciones y No Terminales anulables

- Producciones Anulables: 3, 7, 25, 31, 41, 46
- No Terminales Anulables: Z, X, N, Q, E', T'

b) Calcular el First de cada producción.

First(1) = { [}

First(2) = { b c e d }

First(3) = { }

First(4) = { a h m p i }

First(5) = { a h m p i}

First(6) = { a h m p i }

First(7) = { }

First(8) = { b c e d }

First(9) = { b }

First(10) = { c }

First(11) = { e }

First(12) = { d }

First(13) = { , }

First(14) = { ; }

First(15) = { a }

First(16) = { h }

First(17) = { m }

First(18) = { p }

First(19) = { i }

First(20) = { a }

First(21) = { h }

First(22) = { m }

First(23) = { p }

First(24) = { i }

First(25) = { }

First(26) = { o }

First(27) = { [}

First(28) = { b c d e }

First(29) = { }

First(30) = { ! q < l > g}

First(31) = { }

First(32) = { ! }

First(33) = { q }

First(34) = { < }

First(35) = { l }

First(36) = { > }

First(37) = { g }

First(38) = { (a n r }

First(39) = { + }

First(40) = { - }

First(41) = { }

First(42) = { (a n r }

First(43) = { * }

First(44) = { / }

First(45) = { % }

First(46) = { }

First(47) = { (}

First(48) = { a }

First(49) = { n }

First(50) = { r }

First(51) = { t }

First(52) = { f }

c) Calcular el First de cada No Terminal

First(G) = { [}
First(Z) = { b c e d a h m p i }
First(Y) = { a h m p i }
First(X) = { a h m p i }
First(D) = { b c e d }
First(J) = { b c e d }
First(V) = { , ; }
First(S) = { a h m p i }
First(A) = { a }
First(H) = { h }
First(M) = { m }
First(P) = { p }
First(I) = { i }
First(N) = { o }
First(K) = { s (a n r t f }
First(R) = { (a n r }
First(Q) = { ! q < l > g }
First(O) = { ! q < l > g }
First(E) = { (a n r }
First(E')= { + - }
First(T) = { (a n r }
First(T')= { * / % }
First(F) = { (a n r }

d) Conjunto Follow de cada No Terminal

Follow(G) = { }
Follow(Z) = {] }
Follow(Y) = {] } = Follow(Z) ∪ Follow(X) ∪ "]"
Follow(X) = {] } = Follow(Y)
Follow(D) = { b c e d a h m p i] } = First(Z) ∪ Follow(Z)
Follow(J) = { a }
Follow(V) = { b c e d a h m p i] } = Follow(D)
Follow(S) = { a h m p i] } = First(X) ∪ Follow(X)
Follow(A) = { ;) }
Follow(H) = { a h m p i] } = Follow(S)
Follow(M) = { a h m p i] } = Follow(S)
Follow(P) = { a h m p i] } = Follow(S)
Follow(I) = { a h m p i] } = Follow(S)
Follow(N) = { a h m p i] } = Follow(I)
Follow(K) = { ;) } = Follow(A)
Follow(R) = {) ; }
Follow(Q) = {) ; } = Follow(R)
Follow(O) = { (a n r } = First(E)
Follow(E) = { ;) ! q < l > g } = Follow(K) ∪ First(Q) ∪ Follow(Q) ∪ ")"
Follow(E')= { ;) ! q < l > g } = Follow(E)
Follow(T) = { + - ;) ! q < l > g } = First(E') ∪ Follow(E')
Follow(T')= { + - ;) ! q < l > g } = Follow(T)
Follow(F) = { * / % + - ;) ! q < l > g } = First(T') ∪ Follow(T')

e) Conjunto Selección de cada producción

C.S.(01) = { [}
C.S.(02) = { b c e d }
C.S.(03) = {] }
C.S.(04) = { a h m p i }
C.S.(05) = { a h m p i }
C.S.(06) = { a h m p i }
C.S.(07) = {] }
C.S.(08) = { b c e d }
C.S.(09) = { b }
C.S.(10) = { c }
C.S.(11) = { e }
C.S.(27) = { s }
C.S.(28) = { (a n r }
C.S.(29) = { (a n r }
C.S.(30) = { ! q < l > g}
C.S.(31) = {) ; }
C.S.(32) = { ! }
C.S.(33) = { q }
C.S.(34) = { < }
C.S.(35) = { l }
C.S.(36) = { > }
C.S.(37) = { g }

C.S.(12) = { d }
C.S.(13) = { , }
C.S.(14) = { ; }
C.S.(15) = { a }
C.S.(16) = { h }
C.S.(17) = { m }
C.S.(18) = { p }
C.S.(19) = { i }
C.S.(20) = { a }
C.S.(21) = { h }
C.S.(22) = { m }
C.S.(23) = { p }
C.S.(24) = { i }
C.S.(25) = { a h m p i] }
C.S.(26) = { o }

C.S.(38) = { (a n r }
C.S.(39) = { + }
C.S.(40) = { - }
C.S.(41) = { ;) ! q < l > g }
C.S.(42) = { (a n r }
C.S.(43) = { * }
C.S.(44) = { / }
C.S.(45) = { % }
C.S.(46) = { + - ;) ! q < l > g }
C.S.(47) = { (}
C.S.(48) = { a }
C.S.(49) = { n }
C.S.(50) = { r }
C.S.(51) = { t }
C.S.(52) = { f }

**** *Al tener todos los conjuntos de Selección disjuntos para los mismos No Terminales, se puede concluir que es una gramática LL(1)***

3.2.1 Estructura de Datos

Inicialmente, ya se tenían definidos cuáles serían los átomos para cada una de las clases que se encontraban en el analisis lexico. En algunas casos el átomo no era genérico, pues el átomo depende del símbolo o palabra reservada que se estaba detectando, y no de la clase.

Como se mencionó en la descripción del problema, para poder dar inicio al análisis sintáctico, es importante tener una cadena de átomos. La solución planteada para resolver este problema fue implementar la estructura de datos **listas ligadas simples,** ya que estas permiten tener un mejor control de los átomos que se van introduciendo, así como la optimización de memoria al implementar *malloc*.

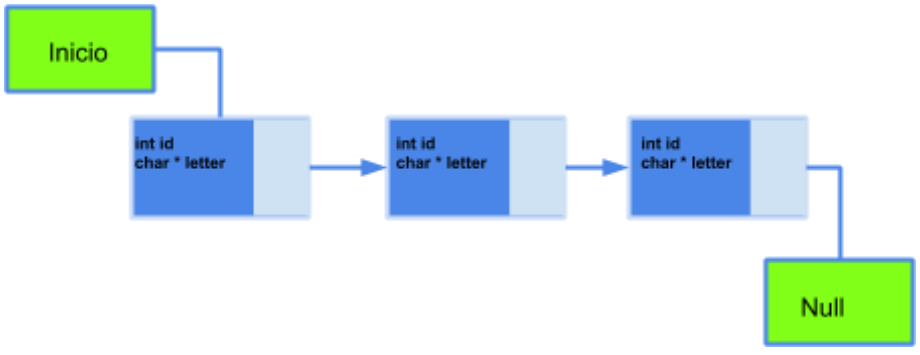


Figura No. 3 Representación de la lista ligada simple utilizada para la cadena de átomos

3.3.2 Técnicas de Inserción

Para la inserción de átomos a la cadena se implementó una función llamada *insertAtom*. Dicha función no devuelve nada, pues su único propósito es concatenar los átomos entre sí, para así generar la cadena resultante de átomos una vez que se ha leído todo el programa de entrada. Esta función recibe dos parámetros de entrada, el átomo a insertar y la lista donde se insertará.
La función de inserción *insertAtom* se implementó de la siguiente forma:

```
void insertAtom(strAtom *listAtom, char *character)
{
        atom *newNode = (atom *)
malloc(sizeof(atom));
        newNode -> id = listAtom -> size;
        newNode -> letter = strdup(character);

        if(listAtom -> head == NULL) {
        newNode -> next = NULL;
        listAtom -> head = newNode;
        listAtom -> tail = newNode;
        } else {
        newNode -> next = NULL;
        listAtom -> tail -> next = newNode;
        listAtom -> tail = newNode;
        }
        listAtom -> size ++;
        return;
}
```

Esta función es llamada en cada una de las clases que se definieron e implementaron con ayuda de flex (Palabras Reservadas, Identificadores, Operadores Relacionales, etc).

Además, en el caso de las palabras reservadas y los operadores relacionales, se implementaron dos funciones auxiliares para realizar la búsqueda de esos componentes léxicos ya definidos. Estas funciones evalúan el contenido de yytext y devuelven el átomo que corresponde al componente léxico reconocido. Estas dos funciones se explicarán a detalle en el siguiente apartado.

Ejemplo:

```
{palRes}        {
                insertAtom(&stringAtoms, knownPalRes(yytext));
                insertComponentLex(pReservadas, 11, yytext, tokensFile, &componentesLexicos, 0);
        }
```

3.2.3 Técnicas de Búsqueda

La técnica de búsqueda para el analizador sintáctico consto de dos funciones llamadas *knowOpREl* y *knowPalREs* dichas funciones implementaron una estructura de control switch-case y el uso de una función definida en el analizador léxico *findComp* .

La función findComp retorna la posición en el arreglo de la palabra que se envía como parámetro de entrada.

```
int findComp(char **array, int lenght, char *key) {
        int i = 0;
        while(i < lenght) {
        if(!strcmp(*(array + i), key)) {
        break;
        }
        i++;
        }

        if(i == lenght) {
        return -1;
        } else {
        return i;
        }
}
```

Las funciones *knowOpRel* y *knowPalRes* retornan el átomo que se debe de ingresar a la cadena, dependiendo de la posición que reciben al llamar a *findComp*.

```
char* knownOpRel(char *key) {
        int i = findComp(opRelacional, 6, key);

        switch(i) {
        case(0): return "!";
        break;
        case(1): return "q";
        break;
        case(2): return "<";
        break;
        case(3): return "l";
        break;
        case(4): return ">";
        break;
        case(5): return "g";
        break;
        default: return "&"; //Se agrega fin de cadena si hubo algun
error
        printf("\n\t***Error en la generacion de cadena de
atomos****\n");
        break;
        }
}
```

```
char* knownPalRes(char *key) {

        int i = findComp(pReservadas, 11, key);

        switch(i) {
        case(0): return "b";
        break;
        case(1): return "c";
        break;
        case(2): return "t";
        break;
        case(3): return "e";
        break;
        case(4): return "f";
        break;
        case(5): return "h";
        break;
        case(6): return "m";
        break;
        case(7): return "p";
        break;
        case(8): return "d";
        break;
        case(9): return "i";
        break;
        case(10): return "o";
        break;
        default: return "&"; //Se agrega fin de cadena si hubo algun error
        printf("\n\t***Error en la generacion de cadena de
atomos****\n");
        break;
        }
```

```
}
```

3.3 Implementación

En esta parte se muestra el código de todas las estructuras utilizadas , así como el cuerpo de las funciones.

Primero, fue necesario identificar los tipos de lista necesarias para poder llevar a cabo el manejo óptimo de la cadena de átomos. Como resultado de un análisis, se concluyó que sería conveniente crear la cadena de átomos con la siguiente estructura:

```
typedef struct atom {
        int id;
        char *letter;
        struct atom *next;
} atom;
```

Además, para tener un control adecuado de las listas, fue necesario implementar una estructura de datos más para definir dos apuntadores, uno que apunte al inicio (*head*) de la lista y otro que indique el fin de la lista o la cola de la lista (*tail*).

```
typedef struct strAtom {
        struct atom *head;
        struct atom *tail;
        int size;
} strAtom;
```

El analizador sintáctico, tiene un comportamiento particular, puesto que para su correcto funcionamiento, este debe de llamar una primera función (que debe estar previamente analizada, estructurada e implementada), esta llama a su vez llama a otra función y así sucesivamente hasta terminar con los No Terminales que se definieron de nuestro lenguaje pu+ . En la tabla No. 4. A continuación se muestra el cuerpo de las funciones para el analizador sintáctico.

Función principal para el analizador Sintáctico

```
void parser() {
        c = nextAtom(&stringAtoms, contAtom++);
        G();
        if(!strcmp(c, "&")) {
                printf("\n\t***NO HUBO ERRORES SINTÁCTICOS***\n");
        } else {
                printf("\n\t***HUBO ERRORES SINTÁCTICOS***\n");
        }
        return;
}
```

Cuerpo de la Funciones para cada uno de los No Terminales del lenguaje pu+

- FUNCION G( )

```
void G() {
        if(!strcmp(c, "[")) {
        c = nextAtom(&stringAtoms, contAtom++);
        Z();
        if(!strcmp(c, "]")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else {
        printf("\nSyntax Error: expecting \"]\"\n");
        }
        } else {
        printf("\nSyntax Error: expecting \"[\"\n");
        }
        return;
}
```

- FUNCION Z( )

```
void Z() {
        if(!strcmp(c, "b") || !strcmp(c, "c") || !strcmp(c, "e") || !strcmp(c, "d")) {
        D();
        Z();
        } else if(!strcmp(c, "]")) {
        return;
        } else if(!strcmp(c, "a") || !strcmp(c, "h") || !strcmp(c, "m") || !strcmp(c, "p") || !strcmp(c, "i")) {
        Y();
        } else {
        printf("\nSyntax Error: expecting Data Type or \"]\" or An Assignment/Control Structure\n");
        }
        return;
}
```

- FUNCION D( )

```
void D() {
        if(!strcmp(c, "b") || !strcmp(c, "c") || !strcmp(c, "e") || !strcmp(c, "d"))
{
        J();
        if(!strcmp(c, "a")) {
        c = nextAtom(&stringAtoms, contAtom++);
        V();
        } else {
        printf("\nSyntax Error: expecting An Identifier\n");
        }
        } else {
        printf("\nSyntax Error: expecting A Data Type\n");
        }
        return;
}
```

❖ FUNCION J( )

```
void J() {
        if(!strcmp(c, "b")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else if(!strcmp(c, "c")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else if(!strcmp(c, "e")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else if(!strcmp(c, "d")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else {
        printf("\nSyntax Error: expecting An Bul/Cadena/Entero/Real\n");
        }
        return;
}
```

❖ FUNCION V( )

```
void V() {
        if(!strcmp(c, ",")) {
        c = nextAtom(&stringAtoms, contAtom++);
        if(!strcmp(c, "a")) {
        c = nextAtom(&stringAtoms, contAtom++);
        V();
        } else {
        printf("\nSyntax Error: expecting Another Identifier\n");
        c = nextAtom(&stringAtoms, contAtom++);
        }
        } else if(!strcmp(c, ";")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else {
        printf("\nSyntax Error: expecting \";\" or \",\" For More Identifiers\n");
        }
        return;
}
```

❖ FUNCION Y( )

```
void Y() {
        if(!strcmp(c, "a") || !strcmp(c, "h") || !strcmp(c, "m") || !strcmp(c, "p") || !strcmp(c, "i"))
{
        S();
        X();
        } else {
        printf("\nSyntax Error: expecting An Assignment or A Control Structure\n");
        }
        return;
}
```

❖ FUNCION X( )

```
void X() {
        if(!strcmp(c, "a") || !strcmp(c, "h") || !strcmp(c, "m") || !strcmp(c, "p") || !strcmp(c, "i")) {
        Y();
        } else if(!strcmp(c, "]")) {
        return;
        } else {
        printf("\nSyntax Error: expecting Another Control Structure or Another Assignment\n");
        }
        return;
}
```

❖ FUNCION S( )

```
void S() {
        if(!strcmp(c, "a")) {
        A();
        if(!strcmp(c, ";")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else {
        printf("\nSyntax Error: expecting \";\"\n");
        }
        } else if(!strcmp(c, "h")) {
        H();
        } else if(!strcmp(c, "m")) {
        M();
        } else if(!strcmp(c, "p")) {
        P();
        } else if(!strcmp(c, "i")) {
        I();
        } else {
        printf("\nSyntax Error: expecting A Control Structure or An Assignment\n");
        }
        return;
}
```

❖ FUNCION A( )

```
void A() {
        if(!strcmp(c, "a")) {
        c = nextAtom(&stringAtoms, contAtom++);
        if(!strcmp(c, "=")) {
        c = nextAtom(&stringAtoms, contAtom++);
        K();
        } else {
        printf("\nSyntax Error: expecting \":=\"\n");
        }
        } else {
        printf("\nSyntax Error: expecting An Identifier\n");
        }
        return;
}
```

❖ FUNCION H( )

```
void H() {
        if(!strcmp(c, "h")) {
        c = nextAtom(&stringAtoms, contAtom++);
        if(!strcmp(c, "[")) {
        c = nextAtom(&stringAtoms, contAtom++);
        Y();
        if(!strcmp(c, "]")) {
        c = nextAtom(&stringAtoms, contAtom++);
        if(!strcmp(c, "m")) {
                c = nextAtom(&stringAtoms, contAtom++);
                if(!strcmp(c, "(")) {
                c = nextAtom(&stringAtoms, contAtom++);
                R();
                if(!strcmp(c, ")")) {
                c = nextAtom(&stringAtoms, contAtom++);
                if(!strcmp(c, ";")) {
                c = nextAtom(&stringAtoms, contAtom++);
                } else {
                printf("\nSyntax Error: expecting ; \n");
                }
                } else {
                printf("\nSyntax Error: expecting ')' \n");
                }
                } else {
                printf("\nSyntax Error: expecting '(' \n");
                }
        } else {
                printf("\nSyntax Error: expecting Mientras Structure \n");
        }
        } else {
        printf("\nSyntax Error: expecting ']' \n");
        }
        } else {
        printf("\nSyntax Error: expecting '[' \n");
        }
        } else {
        printf("\nSyntax Error: expecting Haz Structure \n");
        }
        return;
}
```

❖ FUNCION M( )

```
void M() {
        if(!strcmp(c, "m")) {
        c = nextAtom(&stringAtoms, contAtom++);
        if(!strcmp(c, "(")) {
        c = nextAtom(&stringAtoms, contAtom++);
        R();
        if(!strcmp(c, ")")) {
        c = nextAtom(&stringAtoms, contAtom++);
        if(!strcmp(c, "[")) {
                c = nextAtom(&stringAtoms, contAtom++);
                Y();
                if(!strcmp(c, "]")) {
                c = nextAtom(&stringAtoms, contAtom++);
                } else {
                printf("\nSyntax Error: expecting ']' \n");
                }
        } else {
                printf("\nSyntax Error: expecting '[' \n");
```

```
        }
        } else {
        printf("\nSyntax Error: expecting ')' \n");
        }
        } else {
        printf("\nSyntax Error: expecting '(' \n");
        }
        } else {
        printf("\nSyntax Error: expecting Mientras Structure \n");
        }
        return;
}
```

- FUNCION P( )

```
void P() {
        if(!strcmp(c, "p")) {
        c = nextAtom(&stringAtoms, contAtom++);
        if(!strcmp(c, "(")) {
        c = nextAtom(&stringAtoms, contAtom++);
        A();
        if(!strcmp(c, ";")) {
        c = nextAtom(&stringAtoms, contAtom++);
        R();
        if(!strcmp(c, ";")) {
                c = nextAtom(&stringAtoms, contAtom++);
                A();
                if(!strcmp(c, ")")) {
                c = nextAtom(&stringAtoms, contAtom++);
                if(!strcmp(c, "[")) {
                c = nextAtom(&stringAtoms, contAtom++);
                Y();
                if(!strcmp(c, "]")) {
                c = nextAtom(&stringAtoms, contAtom++);
                } else {
                printf("\nSyntax Error: expecting ']' \n");
                }
                } else {
                printf("\nSyntax Error: expecting '[' \n");
                }
                } else {
                printf("\nSyntax Error: expecting ')' \n");
                }
        } else {
                printf("\nSyntax Error: expecting \";\" \n");
        }
        } else {
        printf("\nSyntax Error: expecting \";\" \n");
        }
        } else {
        printf("\nSyntax Error: expecting '(' \n");
        }
        } else {
        printf("\nSyntax Error: expecting Para Structure \n");
        }
        return;
}
```

❖ FUNCION I( )

```
void I() {
        if(!strcmp(c, "i")) {
        c = nextAtom(&stringAtoms, contAtom++);
        if(!strcmp(c, "(")) {
        c = nextAtom(&stringAtoms, contAtom++);
        R();
        if(!strcmp(c, ")")) {
        c = nextAtom(&stringAtoms, contAtom++);
        if(!strcmp(c, "[")) {
                c = nextAtom(&stringAtoms, contAtom++);
                Y();
                if(!strcmp(c, "]")) {
                c = nextAtom(&stringAtoms, contAtom++);
                N();
                } else {
                printf("\nSyntax Error: expecting ']' \n");
                }
        } else {
                printf("\nSyntax Error: expecting '[' \n");
        }
        } else {
        printf("\nSyntax Error: expecting ')' \n");
        }
        } else {
        printf("\nSyntax Error: expecting '(' \n");
        }
        } else {
        printf("\nSyntax Error: expecting \"Si\" \n");
        }
        return;
}
```

❖ FUNCION K( )

```
void K() {
        if(!strcmp(c, "s")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else if(!strcmp(c, "t")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else if(!strcmp(c, "f")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else if(!strcmp(c, "(") || !strcmp(c, "a") || !strcmp(c, "n") || !strcmp(c, "r")) {
        E();
        } else {
        printf("\nSyntax Error: expecting A String/Cierto/Falso or An Expression\n");
        }
        return;
}
```

- FUNCION E( )

```
void E() {
        if(!strcmp(c, "(") || !strcmp(c, "a") || !strcmp(c, "n") || !strcmp(c, "r")) {
        T();
        EP();
        } else {
        printf("\nSyntax Error: expecting An Expression\n");
        }
        return;
}
```

- FUNCION T( )

```
void T() {
        if(!strcmp(c, "(") || !strcmp(c, "a") || !strcmp(c, "n") || !strcmp(c, "r")) {
        F();
        TP();
        } else {
        printf("\nSyntax Error: expecting ')' or Identifier or Integer or Real\n");
        }
        return;
}
```

- FUNCION EP( )

```
void EP() {
        if(!strcmp(c, "+")) {
        c = nextAtom(&stringAtoms, contAtom++);
        T();
        EP();
        } else if(!strcmp(c, "-")) {
        c = nextAtom(&stringAtoms, contAtom++);
        T();
        EP();
        } else if(!strcmp(c, ";") || !strcmp(c, ")") || !strcmp(c, "!") || !strcmp(c, "q") || !strcmp(c, "<") ||
        !strcmp(c, "l") || !strcmp(c, ">") || !strcmp(c, "g")) { //En caso de ser producción anulable
        return;
        }
        return;
}
```

- FUNCION F( )

```
void F() {
        if(!strcmp(c, "(")) {
        c = nextAtom(&stringAtoms, contAtom++);
        E();
        if(!strcmp(c, ")")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else {
        printf("\nSyntax Error: expecting ')' \n");
        }
        } else if(!strcmp(c, "a")) {
```

```
        c = nextAtom(&stringAtoms, contAtom++);
        } else if(!strcmp(c, "n")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else if(!strcmp(c, "r")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else {
        printf("\nSyntax Error: expecting '('/Identifier/Integer/Real\n");
        }
        return;
}
```

- FUNCION TP( )

```
void TP() {
        if(!strcmp(c, "*")) {
        c = nextAtom(&stringAtoms, contAtom++);
        F();
        TP();
        } else if(!strcmp(c, "/")) {
        c = nextAtom(&stringAtoms, contAtom++);
        F();
        TP();
        } else if(!strcmp(c, "%")) {
        c = nextAtom(&stringAtoms, contAtom++);
        F();
        TP();
        } else if(!strcmp(c, "+") || !strcmp(c, "-") || !strcmp(c, ";") || !strcmp(c, ")") || !strcmp(c, "!") ||
        !strcmp(c, "q") || !strcmp(c, "<") || !strcmp(c, "l") || !strcmp(c, ">") || !strcmp(c, "g") ) {
        return;
        } else {
        printf("\nSyntax Error: expecting + - ; ')' ! q < l > g \n");
        }
        return;
}
```

- FUNCION R( )

```
void R() {
        if(!strcmp(c, "(") || !strcmp(c, "a") || !strcmp(c, "n") || !strcmp(c, "r")) {
        E();
        Q();
        } else {
        printf("\nSyntax Error: expecting '(' or Identifier/Integer/Real/Cierto/Falso \n");
        }
        return;
}
```

- FUNCION Q( )

```
void Q() {
        if(!strcmp(c, "!") || !strcmp(c, "q") || !strcmp(c, "<") || !strcmp(c, "l") || !strcmp(c, ">") || !strcmp(c, "g")) {
        O();
        E();
        } else if(!strcmp(c, ")") || !strcmp(c, ";") ) { //En caso de ser producción anulable
        return;
        } else {
        printf("\nSyntax Error: expecting An Relational Symbol or ')' or ; \n");
        }
        return;
```

```
}
```

- FUNCION O( )

```
void O() {
        if(!strcmp(c, "!")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else if(!strcmp(c, "q")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else if(!strcmp(c, "<")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else if(!strcmp(c, "l")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else if(!strcmp(c, ">")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else if(!strcmp(c, "g")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else {
        printf("\nSyntax Error: expecting A Relational Symbol\n");
        }
        return;
}
```

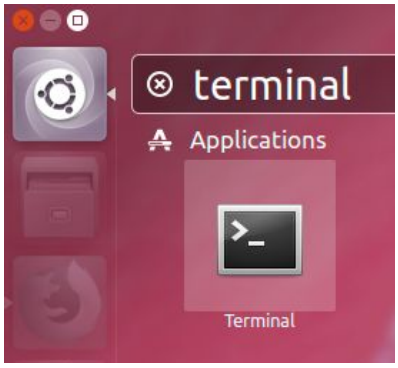
- FUNCION N( )

```
void N() {
        if(!strcmp(c, "a") || !strcmp(c, "h") || !strcmp(c, "m") || !strcmp(c, "p") || !strcmp(c, "i") || !strcmp(c, "]"))
{
        } else if(!strcmp(c, "o")) {
        c = nextAtom(&stringAtoms, contAtom++);
        if(!strcmp(c, "[")) {
        c = nextAtom(&stringAtoms, contAtom++);
        Y();
        if(!strcmp(c, "]")) {
        c = nextAtom(&stringAtoms, contAtom++);
        } else {
        printf("\nSyntax Error: expecting ']'\n");
        }
        } else {
        printf("\nSyntax Error: expecting '['\n");
        }
        } else {
        printf("\nSyntax Error: expecting Identifier or ']' or Control Structure \n");
        }
        return;
}
```

4. Indicaciones de cómo correr el programa

Para realizar la correcta ejecución del analizador es necesario lo siguiente:

1. Ejecutar la Terminal de una distribución de GNU/Linux



2. Tener el archivo “parser.l”

3. Tener ubicado el directorio de dicho archivo “parser.l”

```
a-ThinkPad-L420: ~/Desktop
elda@elda-ThinkPad-L420:~$ cd Desktop
elda@elda-ThinkPad-L420:~/Desktop$ ls
COMPILADORES  parser.l  prueba.txt  SISTEMAS_OPERATIVOS
elda@elda-ThinkPad-L420:~/Desktop$
```

4. Ejecutar siguiente comando: ***$ flex parser.l*** Como resultado, se genera un archivo llamado *lex.yy.c*

```
elda@elda-ThinkPad-L420:~/Desktop$ flex parser.l
elda@elda-ThinkPad-L420:~/Desktop$
```

5. Realizar la compilación de dicho archivo llamando a la librería de flex (lfl) y, opcionalmente, asignarle un nombre en particular que sea distintivo del programa (analizador), para ello se ejecuta el siguiente comando: ***$ gcc lex.yy.c -lfl -o analizador***
   Como resultado, se genera el archivo ejecutable llamado *analizador*

```
elda@elda-ThinkPad-L420:~/Desktop$ gcc lex.yy.c -lfl -o analizador
elda@elda-ThinkPad-L420:~/Desktop$
```

6. Se procede a ejecutarlo, con una observación importante: **es necesario indicarle el archivo fuente del cual realizará el análisis léxico**. Para ello se ejecuta el siguiente comando en la terminal:

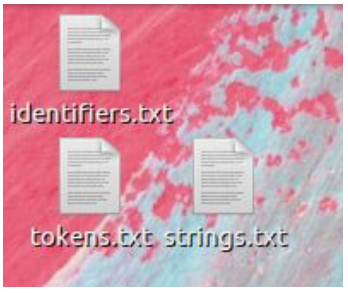
   $ ./analizador prueba.txt

```
elda@elda-ThinkPad-L420:~/Desktop$ ./analizador prueba.txt
```

   Donde *prueba.txt* es el nombre del archivo que del que se desea hacer el análisis léxico y sintáctico.

   En caso de haber seguido correctamente los pasos anteriores, se desplegará en la pantalla de la terminal las tablas que corresponden a *identificadores, cadenas, tokens, cadena de átomos y errores léxicos y sintácticos*. Además, las tablas de identificadores, cadenas y tokens se guardarán en tres archivos diferentes, en el mismo directorio donde se ejecutó el programa. Dichos archivos tienen los siguientes nombres:

identifiers.txt
strings.txt
tokens.txt



   La terminal se podrá visualizar de la siguiente manera:

```
Error lexico en la linea 14: }
Error lexico en la linea 22: =
Error lexico en la linea 29: S
Error lexico en la linea 29: I
Error lexico en la linea 30: N
Error lexico en la linea 33: {
Error lexico en la linea 35: =
Error lexico en la linea 38: {
```

Errores Léxicos, indicando la línea.

```
***Tabla de Tokens***
CLASE    ID
  2      [
  1      0
  7      8
  4      0
  4      5
  4      1
  4      4
  4      1
  4      3
```

Tabla de Tokens

```
***Identificadores (Clase 1)***
 ID      VALOR           TIPO
  0      hola
  1      o
  2      mn
  3      kjlc
  4      condicio
  5      nal
  6      mientras
  7      si
  8      sino
  9      a

***Constantes Cadenas (Clase 6)***
 ID      VALOR
  0      "hola como estas"
  1      "me llamo Elda"
```

Tabla de identificadores y cadenas

```
CADENA DE ATOMOS
!gq>ql<s=nr;,/)()*n*/%()-++=hcansoai*aa+++q()mcan=n;%%aaaaa//+a=n&
```

Cadena de Átomos

```
Syntax Error: expecting ":="

Syntax Error: expecting ";"

Syntax Error: expecting Another Control Structure or Another Assignment

Syntax Error: expecting "]"
```

Errores Sintácticos

En caso de no indicar el nombre del archivo o ingresar uno correcto, se desplegará en la pantalla de la terminal un mensaje indicando que existe un error en la lectura del archivo, como se muestra en la siguiente figura:

```
marcomorenoag@marco-HP:~/Desktop/compilador$ ./scanner
Error de apertura de archivo fuente
Verificar que exista el archivo e ingresar su nombre como parametro

marcomorenoag@marco-HP:~/Desktop/compilador$
```

5. Conclusiones

- **Corella Perez Elda**

Una vez que hemos implementado el analizador léxico, el segundo paso para nuestro compilador es el analizador sintáctico. En esta ocasión, nos dimos cuenta de la importancia que tiene hacer bien uno de los programa pues la salida de estos, es la entrada a otro. Tal es este caso, pues al generarse la cadena de átomos esta debe de ser la entrada a la función del analizador sintáctico. Los algoritmos que se vieron en clase, en el analizador sintáctico cobran un sentido más formal. Pues es bastante diferente hacer el algoritmo en libreta que programado. Además de ir visualizando el comportamiento que tienen las gramáticas una vez que se les aplica los algoritmos. El hacer estos analizadores refuerzas muchos conocimientos, desde la implementación de estructuras de datos que se vieron en los primero semestres, hasta el uso de nuevas herramientas como en su momento lo fue flex.

Verificamos antes de realizar alguna modificación o implementación, que el lenguaje propuesto pu+ fuera una gramática con las características suficientes para ser LL(1) para posteriormente poder codificar las estructuras de datos.

- **Moreno Guerra Marco Antonio**

Con la realización del analizador léxico, como primer parte de un compilador, fue una pieza fundamental para poder realizar de manera exitosa el analizador sintáctico o parser; es por ello, que antes de realizar el desarrollo del parser fue conveniente realizar algunas correcciones y mejoras a la primera versión que se entregó del analizador léxico. Una de los cambios más importantes fue el manejo del campo “value” de la estructura de esas listas de tokens y demás, como tipo String en lugar de Int, lo que permitió manejar todo de manera más práctica.

Otro punto importante, fue que antes de realizar cualquier codificación de las funciones de la gramática, fue el realizar el desarrollo teórico de los conjuntos de selección para poder determinar que la gramática cumple la propiedad LL(1) y con base en los conjuntos de selección realizar la declaración de las funciones de cada No Terminal.

Con esto, se logró visualizar el potencial que tiene el estudio de una gramática formal para un lenguaje, con lo que se puede construir un nuevo lenguaje de programación con las reglas léxicas y sintácticas que se deseen o sean convenientes, de acuerdo con el paradigma al que se enfoca.