

FACULTAD DE INGENIERIA

CARRERA DE INGENIERIA DE SISTEMAS COMPUTACIONALES

Generadores Léxico y Sintáctico para la Creación de una Derivada

CURSO : Compiladores y Lenguajes de Programación

DOCENTE : Arroyo Taboada Ángel David

**ALUMNOS : Chavez Laguna, Carlos
Zelada Cancino, Ronald**

2017

Tabla de contenido

1. HERRAMIENTAS DE DESARROLLO	3
1.1. Herramientas para generar analizadores léxicos Flex	3
1.2. Herramientas para generar analizadores sintácticos Bison	4
2. DESARROLLO DE LOS GENERADORES	5
2.1. Analizador léxico	5
2.2. Analizador sintáctico	5
3. IDENTIFICADORES DE TOKENS PARA EL ANALIZADOR LEXICO	6
4. IDENTIFICACIÓN DE LA GRAMÁTICA PARA EL ANALIZADOR SINTÁCTICO	6
5. PROCEDIMIENTO DE LA COMPILACIÓN	6
6. CONTENIDO DE LOS ARCHIVOS LÉXICO Y SINTÁCTICO	6
6.1. Archivo léxico.l	6
6.2 Archivo sintáctico.y	6
7. DIAGRAMA GENERAL DEL LÉXICO Y SINTÁCTICO	6
8. APRECIACIÓN CRÍTICA	6

1. HERRAMIENTAS DE DESARROLLO

1.1. Herramientas para generar analizadores léxicos Flex

FLEX:

Version:

2.6.0

Commandos instalación (Linux):

sudo apt-get install flex

Descripción:

Flex es una herramienta para generar escáneres: programas que reconocen patrones léxicos en un texto. La descripción se encuentra en forma de parejas de expresiones regulares y código C, denominadas reglas. Flex genera como salida un fichero fuente en C, 'lex.yy.c', que define una rutina 'yylex ()'.

¿Cómo se guardan?

Los ficheros de entrada de Flex (normalmente con la extensión (.l))

Esquema

siguen el siguiente esquema:

%%

patrón1 {acción1}

patrón2 {acción2}

...

dónde:

patrón: expresión regular

acción: código C con las acciones a ejecutar cuando se encuentre concordancia del patrón con el texto de entrada

Flex recorre la entrada hasta que encuentra una concordancia y ejecuta el código asociado. El texto que no concuerda con ningún patrón lo copia tal cual a la salida. *Por ejemplo:*

```
%%
```

```
a*b {printf ("X") ;};
```

```
re;
```

Ejecución

El ejecutable correspondiente transforma el texto:

Pues ha escrito X cada vez que ha encontrado ab o aab y nada cuando ha encontrado re.

Flex genera como salida un fichero fuente en C, 'lex.yy.c', que define una función 'yylex ()'.

El fichero de entrada de Flex está compuesto de tres secciones, separadas por una línea donde aparece únicamente un '%%'.

Características:

El Fichero de Entrada de Flex está compuesto en 3 secciones, separadas por una línea donde únicamente un '%%', estas son:

Definiciones

```
%%
```

Reglas

```
%%
```

Código de usuario

La Sección de Declaraciones es donde se definen macros y para importar los archivos de cabecera escritos en C.

La Sección de Reglas asocia patrones a sentencias en C, estos son expresiones regulares.

La Sección de Código de Usuario es la que contiene sentencias en C y funciones que serán copiadas en el archivo fuente generado.

1.2. Herramientas para generar analizadores sintácticos Bison

BISON

Versión:

3.0.4

Commandos instalación (Linux):

sudo apt-get install bison

Sources:

<http://ftp.gnu.org/gnu/bison>

Descripción:

Es un programa generador de analizadores sintácticos de propósito general perteneciente al proyecto GNU disponible para prácticamente todos los sistemas operativos, se usa normalmente acompañado de Flex aunque los analizadores léxicos se pueden también obtener de otras formas.

Bison convierte la descripción formal de un lenguaje, escrita como una gramática libre de contexto LALR, en un programa en C, C++, o Java que realiza análisis sintáctico.

Características:

Su entrada es un fichero con la extensión '.y' que contiene la especificación de una gramática y genera una función en C que reconoce cadenas válidas de esa gramática. El fichero para la especificación de la gramática contiene 4 partes, divididas por los delimitadores, estos son:

Prólogo: Se realizan las definiciones de las macros, declaraciones de funciones y declaraciones de variables que se utilizarán.

Declaraciones de Bison: Se declaran los símbolos terminales y no terminales, se define la prioridad de los operadores.

Reglas de Gramática: Se define las reglas de la gramática, al menos debe existir una y siempre después de un %% delimitador, lo cual es obligatorio.

Epílogo: Se definen funciones en C.

2. DESARROLLO DE LOS GENERADORES

2.1. Analizador léxico

Un analizador léxico es la primera fase de un compilador consistente en un programa que recibe como entrada el código fuente de otro programa y produce una salida compuesta de tokens o símbolos.

Estos tokens sirven para una posterior etapa del proceso de traducción, siendo la entrada para el analizador sintáctico.

La especificación de un lenguaje de programación a menudo incluye un conjunto de reglas que definen el léxico.

Estas reglas consisten comúnmente en expresiones regulares que indican el conjunto de posibles secuencias de caracteres que definen un token o lexema.

En algunos lenguajes de programación es necesario establecer patrones para caracteres especiales (como el espacio en blanco) que la gramática pueda reconocer sin que constituya un token en sí.

2.2. Analizador sintáctico

Un analizador sintáctico es un programa informático que analiza una cadena de símbolos de acuerdo a las reglas de una gramática formal.

Usualmente hace parte de un compilador, en cuyo caso, transforma una entrada en un árbol de sintáctico de derivación .

El análisis sintáctico convierte el texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la entrada.

Un analizador léxico crea tokens de una secuencia de caracteres de entrada y son estos tokens los que son procesados por el analizador sintáctico para construir la estructura de datos, por ejemplo un árbol de análisis o árboles de sintaxis abstracta.

3. IDENTIFICADORES DE TOKENS PARA EL ANALIZADOR LÉXICO

Los Tokens expuestos a continuación se eligieron debido a que son indispensables para la resolución de una derivada.

Tabla/lista de tokens

%token <caracteres> NUMERO

%token IGUAL

%token SUMA

%token <caracteres> VARIABLE

%token PUNTO-COMA

%token RESTA

Nombre y descripción

Tipo	Variable	Nombre
%token	Numero	Valor numérico entero
%token	Igual	Representación del signo igual “=”
%token	Suma	Representación a un operador “+”
%token	Variable	Constante(x)
%token	Punto- Coma	Siglo (;)
%token	resta	Representación a un operador “-”

Autómata Finito Determinista

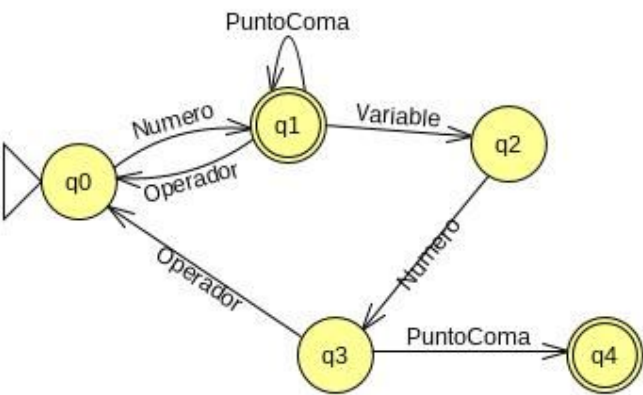
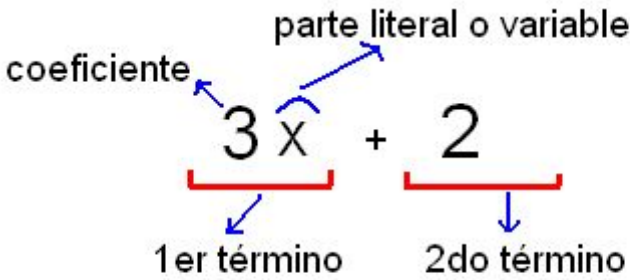


Tabla de estados:

	Numero	Variable	Numero	Operador	Punto-Coma
q0	q1	-	-	-	-
q1	-	q2	-	q0	q1
q2	-	-	q3	-	-
q3	-	-	-	q0	q4
q4	-	-	-	-	-

Porque los elegimos y porque los agrupamos de esa manera los tokens

Nosotros ,como grupo, elegimos los tokens ingresados porque al momento de obtener un resultado para la función a derivar, necesitamos primero el número luego la variable (en este caso “x”) y seguidamente el exponente (que viene a ser un token número también) más el operador(“+” o “-”), finalmente el signo del punto y coma (“;”). Este último se utiliza para finalizar la sentencia de una función a derivar cuando lo ingresamos.



4. IDENTIFICACIÓN DE LA GRAMÁTICA PARA EL ANALIZADOR SINTÁCTICO

Gramatica

TÉRMINO	número
	TÉRMINO variable número
	TÉRMINO operador número
	TÉRMINO punto-coma

Finalidad de la gramática

Estamos utilizando la siguiente gramática con la finalidad de seleccionar términos algebraicos, es decir, nos enfocamos principalmente en obtener un coeficiente, una variable y un exponente porque deseamos derivar por cada término y no operar la expresión.

Declaraciones del código en C

```

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

extern FILE *yyin;

extern int linea;

extern int yylex(void);

extern char *yytext;

void yyerror(char *s);

float suma(float a,float b);

float resta(float a, float b);

float derivarPotencia(float a, float b);

void obtenerOperador(char *s);

void imprimirResultado();

char resultado[500];

%}

```

Definición de reglas

```

termino:NUMERO {$$=$1; }
        |termino VARIABLE NUMERO {$$=$3;
        derivarPotencia(atoi($1),atoi($3));}
        |termino OPERADOR NUMERO {$$=$3obtenerOperador($2);}
        |termino PUNTO_COMA {imprimirResultado();}
;

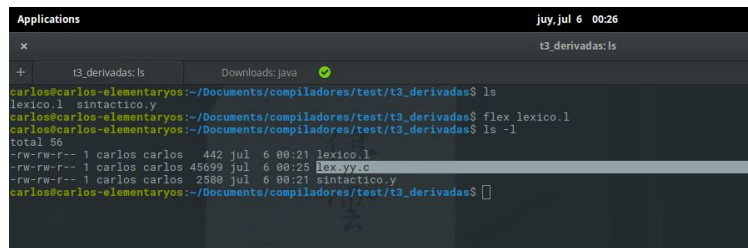
```

5. PROCEDIMIENTO DE LA COMPILACIÓN

La compilación de este proyecto se realizó en GNU Linux (Elementary OS Loki), siguiendo los siguientes pasos:

1. Con el atajo Win+T abrimos la terminal y nos dirigimos a la carpeta donde se encuentran nuestros archivos del analizador.
2. Una vez dentro usamos el programa **flex** para generar nuestro código de analizador léxico en C con el comando: **flex lexico.l**

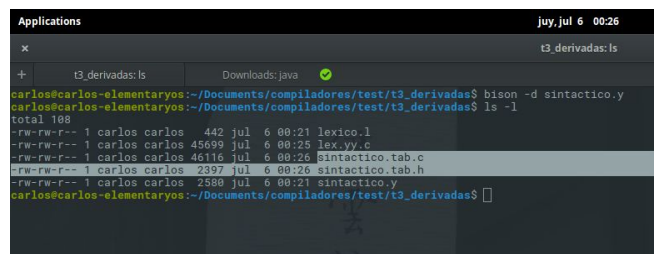
Como podemos observar, nos ha generado un archivo llamado **lex.yy.c**



```
Applications juy, jul 6 00:26
x t3_derivadas: ls
+ t3_derivadas: ls Downloads: java
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$ ls
lexico.l sintactico.y
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$ flex lexico.l
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$ ls -l
total 56
-rw-rw-r-- 1 carlos carlos 442 jul 6 00:21 lexico.l
-rw-rw-r-- 1 carlos carlos 45699 jul 6 00:25 lex.yy.c
-rw-rw-r-- 1 carlos carlos 2580 jul 6 00:21 sintactico.y
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$
```

3. Luego, procedemos a usar el programa bison para generar nuestro código de analizador sintáctico que a al , mismo tiempo, se generará un archivo necesario para el funcionamiento y comunicación comprendido entre el analizador léxico y sintáctico. Usamos el comando: **bison -d sintactico.y**

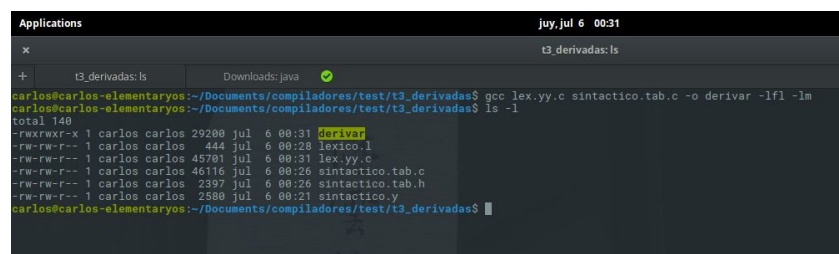
Te genera los archivos **sintactico.tab.c** y **sintactico.tab.h**



```
Applications juy, jul 6 00:26
x t3_derivadas: ls
+ t3_derivadas: ls Downloads: java
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$ bison -d sintactico.y
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$ ls -l
total 108
-rw-rw-r-- 1 carlos carlos 442 jul 6 00:21 lexico.l
-rw-rw-r-- 1 carlos carlos 45699 jul 6 00:25 lex.yy.c
-rw-rw-r-- 1 carlos carlos 46116 jul 6 00:26 sintactico.tab.c
-rw-rw-r-- 1 carlos carlos 2397 jul 6 00:26 sintactico.tab.h
-rw-rw-r-- 1 carlos carlos 2580 jul 6 00:21 sintactico.y
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$
```

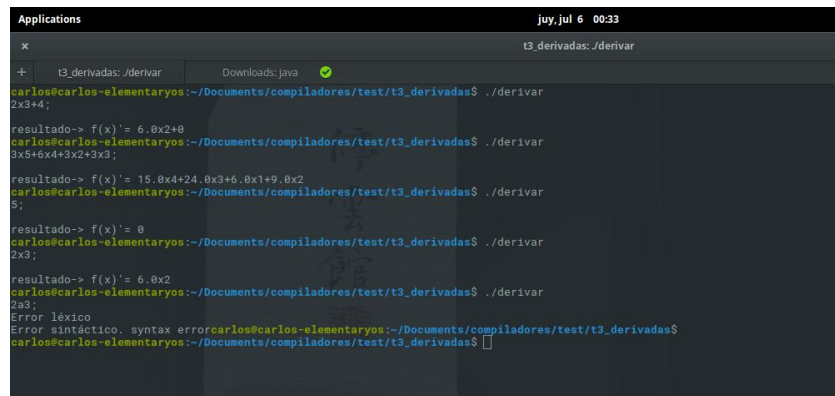
4. Por último, nos falta compilar nuestro código generado ,por bison y flex, con **gcc** para obtener un ejecutable. Utilizamos el comando: **gcc lex.yy.c sintactico.tab.c -o derivar -lfl -lm**

Nos generará un ejecutable



```
Applications juy, jul 6 00:31
x t3_derivadas: ls
+ t3_derivadas: ls Downloads: java
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$ gcc lex.yy.c sintactico.tab.c -o derivar -lfl -lm
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$ ls -l
total 140
-rwxrwxr-x 1 carlos carlos 29280 jul 6 00:31 derivar
-rw-rw-r-- 1 carlos carlos 444 jul 6 00:28 lexico.l
-rw-rw-r-- 1 carlos carlos 45701 jul 6 00:31 lex.yy.c
-rw-rw-r-- 1 carlos carlos 46116 jul 6 00:26 sintactico.tab.c
-rw-rw-r-- 1 carlos carlos 2397 jul 6 00:26 sintactico.tab.h
-rw-rw-r-- 1 carlos carlos 2580 jul 6 00:21 sintactico.y
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$
```

5. Procedemos a comprobar nuestro analizador y observamos los resultados deseados.



```
Applications juy, jul 6 00:33
x t3_derivadas: ./derivar
+ t3_derivadas: ./derivar Downloads: java
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$ ./derivar
2x3+4;
resultado-> f(x)'= 6.0x2+0
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$ ./derivar
3x5+6x4+3x2+3x3;
resultado-> f(x)'= 15.0x4+24.0x3+6.0x1+9.0x2
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$ ./derivar
5;
resultado-> f(x)'= 0
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$ ./derivar
2x3;
resultado-> f(x)'= 6.0x2
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$ ./derivar
2x3;
Error léxico
Error sintáctico. syntax errorcarlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$
carlos@carlos-elementaryos:~/Documents/compiladores/test/t3_derivadas$
```

6. CONTENIDO DE LOS ARCHIVOS LÉXICO Y SINTÁCTICO

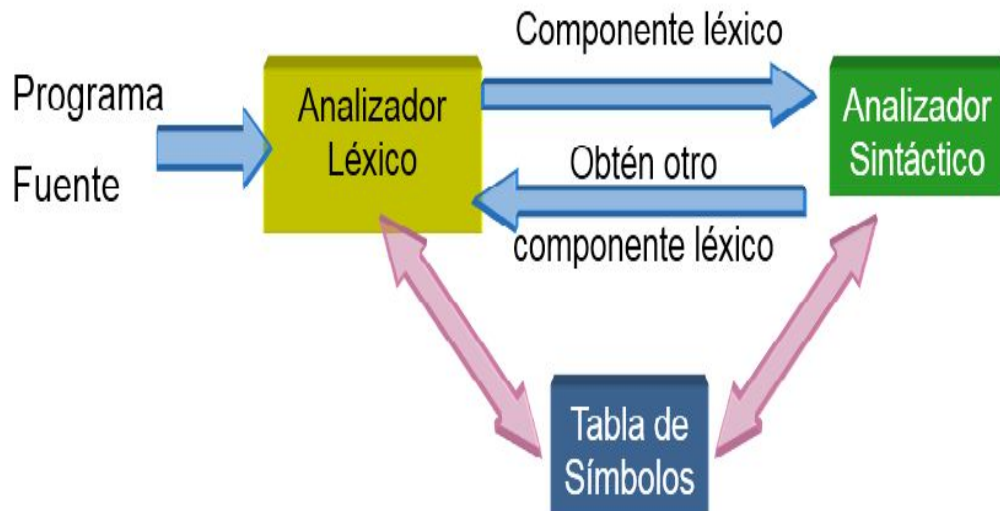
6.1. Archivo léxico.l

```
/*
SCRIPT LÉXICO
*/
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include "derivada.tab.h"
}%

NUMERO [0-9]+("."[0-9]+)?
OPERADOR ("+"|"-"")

%%
{NUMERO}          {yylval.caracteres=strdup(yytext);return(NUMERO);}
{OPERADOR}        {yylval.caracteres=strdup(yytext);return(OPERADOR);}
"x"               {return(VARIABLE);}
","               {return(PUNTO_COMA);}
.                 {printf("Error léxico");}
%%
```

6.2 Archivo sintáctico.γ&7. DIAGRAMA GENERAL DEL LÉXICO Y SINTÁCTICO



8. APRECIACIÓN CRÍTICA

En el presente proyecto utilizamos material de clase y de investigación para llevar a cabo nuestro objetivo que es: realizar un analizador que permita derivar una función. Además, gracias a Flex pudimos conocer las reglas de reconocimiento de símbolos (Tokens) a partir de expresiones regulares. En nuestra opinión el proceso de instalación y la compilación del proyecto en la plataforma Ubuntu nos resultó mucho más fácil a comparación de los pasos que necesita su par en Windows. Sin embargo, el análisis léxico y sintáctico funciona igual para los dos. En el proceso de análisis nos apoyamos en la herramienta JFLAP para dibujar nuestro autómata, extraer nuestra gramática y hacer pruebas de entrada de datos.

En conclusión, este proyecto nos ayuda a comprender más a fondo el funcionamiento de los compiladores que utilizamos de manera cotidiana y las herramientas.