
UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO

CENTRO UNIVERSITARIO UAEM TEXCOCO

# “Herramienta Flex para el análisis lexicográfico de lenguajes formales”

Tesina

Que para obtener el título de

Ingeniero en Computación

Presenta:

C. Guillermo Reséndiz Chávez

Director

Dr. Joel Ayala de la Vega

Revisores

Dr. en C. Alfonso Zarco Hidalgo

M. en I.S.C Irene Aguilar Juárez

Texcoco, Estado de México Diciembre del 2015

# AGRADECIMIENTOS

Al Dr. Joel Ayala de la Vega director de mi tesina por confiar en mí y por el apoyo que me ha brindado a lo largo de este proceso de titulación.

A mis revisores quienes estudiaron mi tesina y la aprobaron.

A mis docentes del Centro Universitario UAEM Texcoco quienes por sus sabios consejos y conocimientos, por su alta capacidad para enseñar me dotaron de sabiduría.

Agradezco profundamente a mis padres el apoyo y por el gran esfuerzo que hicieron para que yo estuviera aquí. ¡Este trabajo es para ustedes!

A mis hermanos que sin sus ánimos y apoyo no hubiera podido seguir adelante, siempre estuvieron ahí, ¡Gracias!

A mi novia Karla por apoyarme y estar conmigo en todo momento.

A mis compañeros y amigos que estuvieron a lo largo de mi formación como ingeniero; Tomas, Carlos Adán, Adrián, Carlos Bautista y Neyffi y a todo mi grupo por compartir tantos momentos y experiencias.

Para ellos es este agradecimiento de tesina, pues es a ellos a quienes se las debo por su apoyo incondicional.

¡MUCHAS GRACIAS!

# DEDICATORIA

A mi madre María De los ángeles Chávez por haberme apoyado en todo momento, por sus consejos, sus valores, por la motivación constante que me ha permitido ser una persona de bien, pero más que nada, por su amor.

A mi padre Reynaldo Reséndiz por los ejemplos de perseverancia y constancia que lo caracterizan y que me ha infundado siempre, por el valor mostrado para salir adelante y por su amor.

A mi hermana Cinthya quien con su apoyo moral y con su ejemplo me ha enseñado a salir adelante.

A mi hermano Jonathan que siempre ha estado junto a mí y brindándome su apoyo.

A mi amada Karla, por iluminar mi vida con la suya. El mérito de éste trabajo también te pertenece, y nunca terminaré de agradecerte lo buena que has sido conmigo.

Te amo.

A mi familia en general, porque me han brindado su apoyo incondicional y por compartir conmigo buenos y malos momento.

A mis amigos, que sin ustedes a mi lado no lo hubiera logrado, tantas desveladas sirvieron de algo y aquí está el fruto. Los quiero mucho y nunca los olvidaré.

# INDICE

INDICE ..... I
INDICE DE FIGURAS ..... III
INDICE DE TABLAS ..... IV
INTRODUCCIÓN ..... 1
Capítulo 1 ..... 3
LENGUAJES REGULARES Y AUTÓMATAS FINITOS ..... 3
1.1 Definición ..... 3
1.3 Expresiones regulares ..... 5
1.4 Autómata ..... 7
1.3.1 Autómatas finitos (AF) ..... 7
1.4.2 Autómata finito determinista (AFD) ..... 9
Capítulo 2 ..... 13
GRAMÁTICA LIBRE DE CONTEXTO Y AUTOMATA TIPO PILA ..... 13
2.1 Definición ..... 13
2.2 Gramáticas libres de contexto ..... 13
2.2.1 Notación Backus-Naur ..... 14
2.2.2 Árbol de derivación ..... 17
2.3 Autómatas tipo pila (AP) ..... 24
Capítulo 3 ..... 33
FLEX ..... 33
3.1 ANALIZADOR LEXICO ..... 33
3.2 FLEX. ..... 35
3.2.1 FORMATO DE FICHERO DE ENTRADA ..... 37
3.2.2 PATRONES ..... 39
3.2.3 EMPAREJAMIENTO DE LA ENTRADA ..... 42
3.2.4 ACCIONES ..... 42
3.2.5 EL ANALIZADOR GENERADO ..... 43
Capítulo 4 ..... 45
EJEMPLOS INTERESANTES ..... 45

EJEMPLO 1 ..... 45
EJEMPLO 2 ..... 52
EJEMPLO 3, FLEX Y BISON JUNTOS (David, 2011) ..... 59
EJEMPLO 4 ..... 70
CONCLUSIÓN ..... 78
ANEXO ..... 79
ANEXO INSTALACION DE FLEX Y BISON EN LINUX Y WINDOWS ..... 79
INSTALACIÓN DE FLEX EN LINUX ..... 79
INSTALACIÓN DE FLEX EN WINDOWS 7 ..... 80
INSTALACIÓN DE BISON EN LINUX ..... 83
INSTALACIÓN DE BISON PARA WINDOWS ..... 84
INSTALACIÓN DE UN COMPILADOR ..... 87
INSTALACIÓN DE MINGW EN WINDOWS 7 ..... 88
CONFIGURACIÓN DE FLEX, BISON Y MINGW EN WINDOWS 7 ..... 89
ANEXO MANUAL COMPILADOR GCC ..... 91
REFERENCIAS ..... 99

# INDICE DE FIGURAS

*Figura 1 Autómata que reconoce "abc"* ____ 8
*Figura 2 Autómata de un interruptor con una bombilla.* ____ 10
*Figura 3 AFN que acepta todas las cadenas que terminan en 01.* ____ 12
*Figura 4 Árbol de derivación ejemplo* ____ 22
*Figura 5 representación gráfica de un árbol con derivación por la izquierda.* ____ 23
*Figura 6 Representación gráfica de un árbol con derivación por la derecha.* ____ 24
*Figura 7 Representación gráfica de un autómata tipo pila (AP).* ____ 27
*Figura 8 Función de un analizador léxico y sintáctico* ____ 34
*Figura 9 Flex se encarga de transformar las especificaciones (reglas y acciones) que nosotros definamos a un código C que representa al analizador.* ____ 35
*Figura 10 Compilación y ejecución en sistema Linux del ejemplo verbos en inglés.* ____ 49
*Figura 11 Compilación y ejecución del ejemplo verbos en sistema Windows 7.* ____ 51
*Figura 12 Compilación y ejecución del ejemplo ejemplo2.l en sistema Linux* ____ 54
*Figura 13 Compilación y ejecución del ejemplo ejemplo2.l en sistema Windows* ____ 56
*Figura 14 Compilación del ejemplo calcu en Windows 7* ____ 66
*Figura 15 Ejecución del ejemplo calcu en Windows 7* ____ 66
*Figura 16 Compilación del ejemplo calculadora en sistema Linux (Debian 7).* ____ 69
*Figura 17 Ejecución del ejemplo calculadora en sistema Linux (Debian 7).* ____ 69
*Figura 18 Ejecución del ejemplo de analizador léxico sistema Linux.* ____ 75
*Figura19 Ejecución del ejemplo del analizador léxico en sistema Windows.* ____ 76
*Figura 20 Archivo "gram.txt"* ____ 77
*Figura 21 Gestor de paquetes Synaptic* ____ 80
*Figura 22 Página web principal de Flex* ____ 81
*Figura 23 Instalador de Flex para Windows* ____ 81
*Figura 24 Instalador de Flex para Windows* ____ 82
*Figura 25 Instalador de Flex para Windows* ____ 82
*Figura 26 Gestor de paquete Synaptic* ____ 83
*Figura 27 Página web principal de Bison* ____ 84
*Figura 28 Instalador de Bison para Windows* ____ 85
*Figura 29 Instalador de Bison para Windows* ____ 85
*Figura 30 Instalador de Bison para Windows* ____ 86
*Figura 31 Gestor de paquetes Synaptic* ____ 87
*Figura 32 Pagina web descarga del compilador MinGW para Windows* ____ 89
*Figura 33 Instalador de MinGw para Windows* ____ 89
*Figura 34 Configuración de Flex, Bison y MinGW en Windows 7* ____ 90

# INDICE DE TABLAS

*Tabla 1Tabla de transición de un autómata que reconoce “abc”* 9
*Tabla 2 Métodos y macros frecuentes usados en Flex.* 39
*Tabla 3 Caracteres más usados en Flex.* 41
*Tabla 4 Extensiones* 92

# INTRODUCCIÓN

INTRODUCCIÓN

El presente trabajo es el resultado del proyecto de titulación en la modalidad de Tesina, cuyo título es “Herramienta Flex para el análisis lexicográfico de lenguajes formales”; siendo responsable de dirigir dicho proyecto el Dr. en Ed. Joel Ayala de la Vega.

El desarrollo de compiladores, ya fuera en ensamblador o utilizando lenguajes intermedios, el problema de escribir un compilador “a mano” es que había que realizar muchas tareas repetitivas. Cuando se hizo patente esta necesidad de automatización aparecieron las primeras herramientas de ayuda a la construcción de compiladores. Lo que hacen estas herramientas es generar código en un lenguaje de programación (C, C++). Estas herramientas pueden hacer muchas de las tareas que realizan los compiladores, tales como la búsqueda de patrones, la escritura de código, el análisis léxico así como el análisis sintáctico y semántico.
(Arteaga Caballero Jorge Julián, 2009)

Flex y Bison son dos herramientas útiles para crear programas que reaccionen a una entrada de datos con una estructura y un lenguaje predeterminado. Como ejemplo se pueden crear compiladores intérprete y analizadores de línea de comando.

Flex define las reglas de reconocimiento de símbolos (Tokens) a partir de expresiones regulares, cuando un token es reconocido por uno de estos patrones se le define una acción. Por lo general esta acción es devolver el tipo y/o el valor (lexema).

Bison es un programa generador de analizadores sintácticos de propósito general, se usa normalmente acompañado de Flex. Es utilizado para crear analizadores para muchos lenguajes desde simples calculadoras hasta lenguajes complejos. (García, 2010) (Compiladores, 2015)

El objetivo de esta investigación es mostrar la interacción entre la herramienta llamada FLEX para el análisis lexicográfico de lenguajes regulares con la herramienta Bison para el análisis sintáctico, parte fundamental de la creación de compiladores, por medio de ejemplos claros y sencillos.

La investigación está orientada para uso de indagación y demostración a todo aquel alumno o persona interesada en usar estas herramientas para desarrollo de analizadores léxicos y sintácticos.

# Capítulo 1

## LENGUAJES REGULARES Y AUTÓMATAS FINITOS

### 1.1 Definición

La base de todo lenguaje es el alfabeto (Σ), el cual es un conjunto finito de símbolos.

Ejemplos de alfabeto: $\Sigma_1 = \{0,1\}$ $\Sigma_2 = \{a, b, c\ldots, z\}$ $\Sigma_3 = \{+, -, *, /\}$

Una cadena o palabra (w) es una concatenación de elementos pertenecientes a un alfabeto.

### 2.2 Operaciones sobre los lenguajes, con expresiones regulares.

-Unión

La unión de dos lenguajes L y M, designada como L U M, es el conjunto de cadenas que pertenecen a L o a M.

Por ejemplo, si L = {10, 001, 111} y M = {ε, 01}, haciendo una ordenación lexicográfica queda como: L U M = {ε, 10, 01, 111, 001}.

-Concatenación

La concatenación de los lenguajes L y M es el conjunto de cadenas que se puede formar tomando cualquier cadena de L y concatenándola con cualquier cadena de M. Retomando los lenguajes del ejemplo anterior tenemos que la concatenación es:

L•M = {10,001, 111, 1001, 00101, 11101}

-Clausura de Kleene

En lógica matemática y en ciencias de la computación, la clausura de Kleene (también llamada estrella Kleene o cierre estrella) es una operación unaria que se aplica sobre un conjunto de cadenas de caracteres o un conjunto de símbolos o caracteres (alfabeto), y representa el conjunto de las cadenas que se pueden formar tomando cualquier número de cadenas del conjunto inicial, posiblemente con repeticiones, y concatenándolas entre sí.

La aplicación de la clausura de Kleene a un conjunto *V* se denota como *V**. Es muy usada en expresiones regulares y fue introducida en este contexto por Stephen Kleene (1909-1994) para caracterizar un cierto autómata.

La clausura (o asterisco, o clausura de Kleene) de un lenguaje L se designa mediante L* y representa el conjunto de cadenas que se pueden formar tomando cualquier número de cadenas de L, posiblemente con repeticiones (es decir, la misma cadena se puede seleccionar más de una vez) y concatenando todas ellas.

Por ejemplo, si L = {0,1}, entonces L* es igual a todas las cadenas de 0s y 1s, incluyendo a la cadena vacía.

Si L = {0,11}, entonces L* constará de aquellas cadenas de 0s y 1s tales que los 1s aparezcan por parejas, como por ejemplo 011, 11110 y ε, pero no 01011 ni 101.
Con estos operadores, se puede definir a un lenguaje formal como:

$L = \{w \in \sum^{*} \mid w \text{ tiene una propiedad P}\}$

## 1.3 Expresiones regulares

Una expresión regular está compuesta de operadores que inciden sobre un alfabeto en particular, los dos operadores más importantes son la unión y la clausura de kleene.
Cada expresión regular x define un lenguaje regular L(x). Las reglas de definición especifican como se forma L(x) combinando de varias formas los lenguajes representados por las subexpresiones x.

Una expresión regular sobre el alfabeto Σ, son los caracteres sobre el alfabeto:

Σ ∪ {(, ), Ø, ∪, *, +}

Por lo tanto las reglas que rigen las expresiones regulares son las siguientes:

- Ø : Es una expresión regular y cada miembro de Σ es una expresión regular.
- ϱ : es una expresión regular que denota el lenguaje consistente de solo la cadena vacía {ϱ}
- Si α y β son expresiones regulares, lo es también α β
- Si α y β son expresiones regulares, lo es también α | β
- Si α es expresión regular, lo es también α*
- Toda expresión regular describe en forma concisa ciertos lenguajes infinitos, sin embargo no puede describir todos los lenguajes, por lo que un lenguaje es regular si y solo si puede ser descrito por una expresión regular.

El termino lenguaje es una expresión regular se refiere a cualquier conjunto de cadenas de un alfabeto fijo, a los lenguajes les pueden aplicar operaciones de unión, concatenación, Kleene-Star o cerradura positiva. Dichas operaciones funcionan de igual forma que en las expresiones regulares. Un lenguaje denotado por el conjunto “a” (siendo a el conjunto de reglas o expresiones regulares que lo define) se representa como L(a).

L(a) es una función, por lo que a cada valor que puede tomar “a” le corresponde un valor definido en la gramática, la tarea de dicha función es la verificación de una entrada que deberá cumplir con las reglas de la gramática, es decir si se tiene:
L(a) = {w | w contiene pares de b’s}
La función L(a) deberá verificar que toda cadena w se reciba cumpla con contener pares de b’s, si esto es así el resultado será positivo, de lo contario el resultado es negativo.

Los lenguajes tienen las siguientes propiedades:

- ✓ L(Ø) = Ø y L(a) = {a} para cada a ϵ Σ
- ✓ Si r es una expresión regular, entonces L(r) = {r}
- ✓ Si r y s son expresiones regulares, entonces L(rs) = L(r)L(s)
- ✓ Si r y s son expresiones regulares, entonces L(r | s) = L(r) | L(s)
- ✓ Si r es una expresión regular, entonces L(r*) = L(r)*

Las expresiones regulares cuentan con los símbolos Kleene-Star y cerradura positiva para indicar que un símbolo o símbolos se repetirán, sin embargo, éstas repeticiones son indeterminadas, la única “restricción” que se pondría dentro de las expresiones regulares es si se quiere que al menos se repita una vez (utilizando cerradura positiva) o dejar abierta la opción utilizando Kleene-Star, es decir, indicar que pueden o no existir repeticiones.

Ejemplo.
n=1: $a^n b^n = a^1 b^1 = ab,$

$(ab)^+$ = ab, considerando que aleatoriamente solo existiera una repetición de ab, ya que de no ser así se obtendría cadenas como ababab, que si bien es cierto cuenta con el mismo número de a’s y b’s es diferente a la cadena resultante de $a^n b^n$, con n=3, la cadena aaabbb, por lo tanto aaabbb ≠ ababab.

## 1.4 Autómata

Los autómatas vienen a ser mecanismos formales que "realizan" derivaciones en gramáticas formales. La manera en que las realizan es mediante la noción de reconocimiento. Una palabra será generada en una gramática si y sólo si la palabra hace transitar al autómata correspondiente a sus condiciones terminales.

Tipos de autómatas

- ✓ Autómata finito: reconocen lenguajes regulares.
- ✓ Autómata tipo pila: Reconocen lenguajes de contexto-libre.
- ✓ Autómata bidireccional: Reconocen lenguajes dependientes del contexto.
- ✓ Máquina de Turing: Reconocen lenguajes del tipo 0 de la jerarquía de Chomsky, llamados también lenguajes con estructura de frase

### 1.3.1 Autómatas finitos (AF)

Un autómata finito o máquina de estado finito es un modelo matemático de un sistema que recibe una cadena constituida por símbolos de un alfabeto y determina si esa cadena pertenece al lenguaje que el autómata reconoce.

Este modelo está conformado por un alfabeto, un conjunto de estados finitos, una función de transición, un estado inicial y un conjunto de estados finales.
Se puede definir mediante una quíntupla (E, Q, f, q0, F) donde:

E: alfabeto de entrada.
Q: conjunto de estados; es conjunto finito no vacío.
f: función de transición. f(Q,E)
q0: (perteneciente a Q) estado inicial.
F: (perteneciente a Q) conjunto de estados finales o de aceptación.

La representación gráfica de un autómata:

 Estado inicial

 Estado final

 Paso de un estado a otro dependiendo de la entrada “X”

Ejemplo Figura 1.

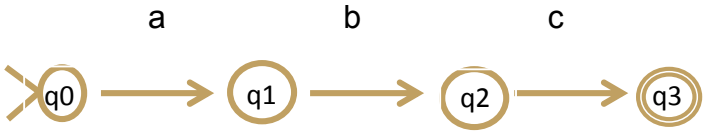



*Figura 1 Autómata que reconoce “abc”*

El funcionamiento de un autómata finito se basa en una función de transición que recibe a partir de un estado inicial, una cadena de caracteres pertenecientes al alfabeto (la entrada) y que va leyendo dicha cadena a medida que el autómata se desplaza de un estado a otro, para finalmente detenerse en un estado final o de aceptación, que representa la salida.

Representación como tabla de transiciones.

Otra manera de describir el funcionamiento de un autómata finito es mediante el uso de tablas de transiciones o matrices de estados. Retomando el ejemplo anterior tenemos la tabla 1.

| Estado inicial | Transición | Estado final |
|---|---|---|
| **q0** | A | q1 |
| **q1** | B | q2 |
| **q2** | C | q3 |

*Tabla 1Tabla de transición de un autómata que reconoce "abc"*

Los autómatas finitos son capaces de reconocer, solamente, un determinado tipo de lenguajes, llamados Lenguajes Regulares, que pueden ser caracterizados también, mediante un tipo de gramáticas llamadas también regulares.

### 1.4.2 Autómata finito determinista (AFD)

Un autónoma finito determinista es aquel que solo puede estar en un único estado después de leer cualquier secuencia de entradas. El término "determinista" hace referencia al hecho de que para cada entrada solo existe uno y solo un estado al que el autómata puede hacer la transición a partir de un estado actual.

Formalmente, un autómata finito determinista es una quíntupla (Q, E, f, q0, F) donde:

Q: conjunto finito no vacío de estados.

E: alfabeto de entrada.
f: Es un subconjunto de "QxE → Q", función de transición que especifica a qué estado pasa el autómata desde el estado actual al recibir un símbolo de entrada.

Esta función se define para todas las parejas posibles de estados y de símbolos de entrada f(q, a)= q' significa que del estado "q" con el símbolo "a", el autómata pasa al estado q'.

q0 ϵ Q: estado inicial del autómata.

F C Q: conjunto de estados finales.

Ejemplo.

Considere un sistema formado por una lámpara y un interruptor. La lámpara puede estar encendida o apagada. El sistema sólo puede recibir un estímulo exterior: pulsar el interruptor. El funcionamiento es habitual: si se pulsa el interruptor y estaba apagada la lámpara, se pasa al estado de encendido, o si está encendida, pasa a pagada. Se desea que la bombilla este inicialmente apagada.

Considere que 0: encendido, 1: apagado y la única entrada posible (pulsar interruptor) es “p”.

Sea A = (Q, E, f, q0, F), donde:

E= {p}
Q= {$q_0$, $q_1$}
f= ($q_0$, p) = $q_1$, f($q_1$, p) = $q_0$
F={q0}

Gráficamente el autómata quedaría como se muestra en la Figura 2.

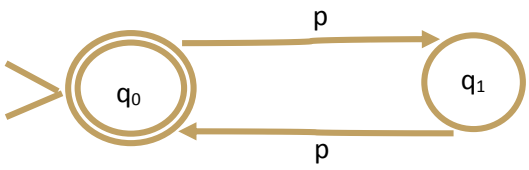



*Figura 2 Autómata de un interruptor con una bombilla.*

### 2.4.3 Autómata finito no determinista (AFN)

Un autómata finito no determinista es un autómata finito que, a diferencia de los autómatas finitos deterministas (AFD), posee al menos un estado $q \in Q$, tal que para un símbolo $a \in \Sigma$ del alfabeto, existe más de una transición $\delta(q,a)$ posible, también permite la transición con cadena vacía o con una palabra de tamaño mayor a uno.

Formalmente, si bien un autómata finito determinista se define como una quíntupla $(Q, \Sigma, q_0, \delta, F)$ donde:

$Q$ es un conjunto de estados;
$\Sigma$ es un alfabeto;
$q_0 \in Q$ es el estado inicial;
$\delta: Q \times \Sigma \to Q$ es una función de transición;
$F \subseteq Q$ es un conjunto de estados finales o de aceptación.

en un AFND la función de transición se define como:

$$\delta : Q \times \Sigma \to \mathcal{P}(Q)$$

Para el caso de los AFND, se suele expresar la función de transición de la forma:

$$\delta : Q \times \{\Sigma \cup \epsilon\} \to \mathcal{P}(Q)$$

donde $P(Q)$ es el conjunto potencia de $Q$. Esto significa que los autómatas finitos deterministas son un caso particular de los no deterministas, puesto que $Q$ pertenece al conjunto $P(Q)$.

La interpretación que se suele hacer en el cómputo de un AFND es que el autómata puede pasar por varios estados a la vez, generándose una ramificación de las *configuraciones* existentes en un momento dado.

Ejemplo Figura 3.

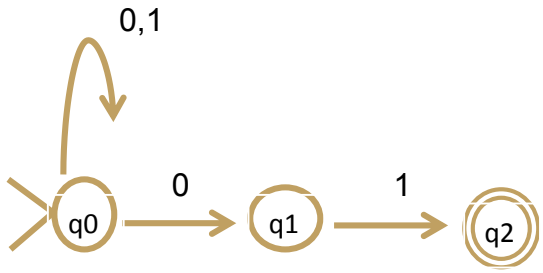



*Figura 3 AFN que acepta todas las cadenas que terminan en 01.*

# Capítulo 2

## GRAMÁTICA LIBRE DE CONTEXTO Y AUTOMATA TIPO PILA

### 2.1 Definición

Proviene del latín “grammatica” que significa “gramática”. Este a su vez desciende del griego “grammatikós” que significa “que sabe leer y escribir, gramático” y que se compone de la palabra “grámma” que significa “signo escrito, letra” (Definiciona, 2015).

La gramática es un ente formal para especificar, de una manera finita, el conjunto de símbolos que constituye un lenguaje, así tenemos la gramática del español, del francés, etc. (L., 2010)

### 2.2 Gramáticas libres de contexto

Estas gramáticas, conocidas también, por la jerarquía de Chomsky, como gramáticas de tipo 2 o gramáticas independientes del contexto, son las que generan los lenguajes libres o independientes del contexto. Los lenguajes libres del contexto son aquellos que pueden ser reconocidos por un autómata de pila determinístico o no determinístico.

Como toda gramática se definen mediante una cuádrupla G = (N, T, P, S), siendo:

N es un conjunto finito de símbolos no terminales
T es un conjunto finito de símbolos terminales $N \cap T = \emptyset$
P es un conjunto finito de producciones
S es el símbolo distinguido o axioma $S \notin (N \cup T)$

En una gramática libre del contexto, cada producción de P tiene la forma

$$A \rightarrow \omega \quad \begin{cases} A \in N \cup \{S\} \\ \omega \in (N \cup T)^* - \{ \varepsilon \} \end{cases}$$

Es decir, que en el lado izquierdo de una producción pueden aparecer el símbolo distinguido o un símbolo no terminal y en el lado derecho de una producción cualquier cadena de símbolos terminales y/o no terminales, incluyendo la cadena vacía.

Ejemplo 1:

La siguiente gramática genera las cadenas del lenguaje $L_1 = \{wcw^r \mid w \in \{a, b\}^* \}$
$G_1 = (\{A\}, \{a, b, c\}, P1, S1)$, y P1 contiene las siguientes producciones
$S1 \rightarrow A$
$A \rightarrow aAa$
$A \rightarrow bAb$
$A \rightarrow c$

### 2.2.1 Notación Backus-Naur

Las gramáticas libres del contexto se escriben, frecuentemente, utilizando una notación conocida como BNF (Backus-Naur Form). BNF es la técnica más común para definir la sintaxis de los lenguajes de programación.

El BNF se utiliza extensamente como notación para las gramáticas de los lenguajes de programación de la computadora, de los sistemas de comando y de los protocolos de comunicación, así como una notación para representar partes de las

gramáticas de la lengua natural (por ejemplo, el metro en la poesía de Venpa). La mayoría de los libros de textos para la teoría o la semántica del lenguaje de programación documentan el lenguaje de programación en BNF.

En esta notación se deben seguir las siguientes convenciones:

- ✓ los no terminales se escriben entre paréntesis angulares < >
- ✓ los terminales se representan con cadenas de caracteres sin paréntesis angulares
- ✓ el lado izquierdo de cada regla debe tener únicamente un no terminal (ya que es una gramática libre del contexto)
- ✓ el símbolo ::=, que se lee “se define como” o “se reescribe cómo”, se utiliza en lugar de →

Varias producciones del tipo

<A> ::= <B1>
<A> ::= <B2>
.
.
.
<A> ::= <Bn>

se pueden escribir como <A> ::= <B1> | <B2> | ...| <Bn>

Ejemplo 1:
La siguiente es una definición BNF del lenguaje que consiste de cadenas de paréntesis anidados:

<cadena_par> :: = <cadena_par> <paréntesis> | <paréntesis>
<paréntesis> ::= (<cadena_par> ) | ( )

Por ejemplo las cadenas ( ) ( ( ) ) y ( ) ( ) ( ) son cadenas válidas. En cambio las cadenas ( () y ( ) ) ) no pertenecen al lenguaje.

Ejemplo 2:

<simbolo> ::= <expresión con símbolos>

donde <símbolo> es un no terminal, y la expresión consiste en secuencias de símbolos o secuencias separadas por la barra vertical, '|', indicando una opción, el conjunto es una posible substitución para el símbolo a la izquierda. Los símbolos que nunca aparecen en un lado izquierdo son terminales.

Otros ejemplos:

<dirección postal> ::= <nombre> <dirección> <apartado postal>

Se traduce al español como:

- Una dirección postal consiste de un nombre, seguido por una dirección, seguida por un apartado postal.

<personal> ::= <primer nombre> | <inicial> "."

- Una parte "personal" consiste en un nombre o una inicial seguido(a) por un punto [Alan]

### 2.2.2 Árbol de derivación

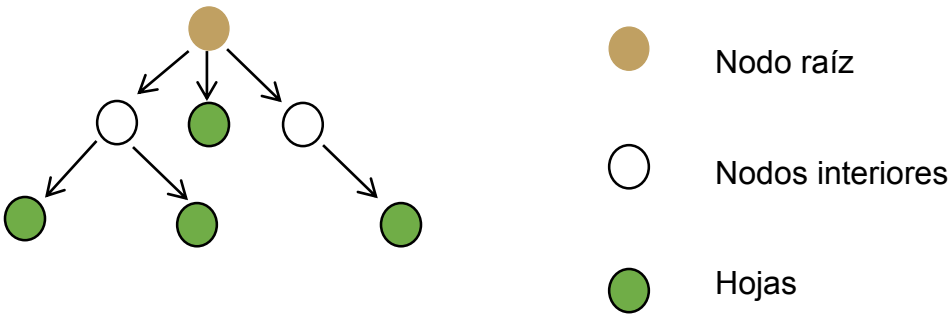
Un árbol de derivación permite mostrar gráficamente cómo se puede derivar cualquier cadena de un lenguaje a partir del símbolo distinguido de una gramática que genera ese lenguaje.

Un árbol es un conjunto de puntos, llamados nodos, unidos por líneas, llamadas arcos. Un arco conecta dos nodos distintos. Para ser un árbol un conjunto de nodos y arcos debe satisfacer ciertas propiedades:

- ✓ hay un único nodo distinguido, llamado raíz (se dibuja en la parte superior) que no tiene arcos incidentes.
- ✓ todo nodo c excepto el nodo raíz está conectado con un arco a otro nodo k, llamado el padre de c (c es el hijo de k). El padre de un nodo, se dibuja por encima del nodo.
- ✓ todos los nodos están conectados al nodo raíz mediante un único camino.
- ✓ los nodos que no tienen hijos se denominan hojas, el resto de los nodos se denominan nodos interiores.

El árbol de derivación tiene las siguientes propiedades:

- ✓ el nodo raíz está rotulado con el símbolo distinguido de la gramática;
- ✓ cada hoja corresponde a un símbolo terminal o un símbolo no terminal;
- ✓ cada nodo interior corresponde a un símbolo no terminal.



Para cada cadena del lenguaje generado por una gramática es posible construir (al menos) un árbol de derivación, en el cual cada hoja tiene como rótulo uno de los símbolos de la cadena.

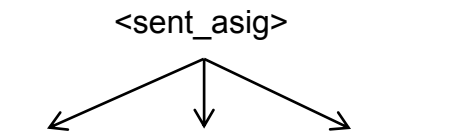
Ejemplo:

La siguiente definición BNF describe la sintaxis (simplificada) de una sentencia de asignación de un lenguaje tipo Pascal:

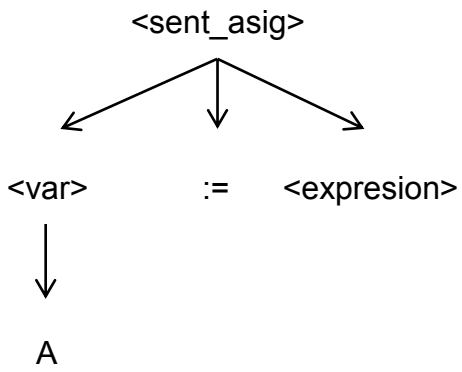
<sent_asig> ::= <var> := <expresion>
<expresion> ::= <expresion> + <termino> | <expresion> - <termino> | <termino>
<termino> ::= <termino> * <factor> | <termino> / <factor> | <factor>
<factor> ::= ( <expresion> ) | <var> | <num>
<var> ::= A  B  C  D  ...  Z
<num> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Por ejemplo, la sentencia A:= A + B es una sentencia de asignación que pertenece al lenguaje definido por la definición BNF dada, y cuyo árbol de derivación se construye como se muestra en las siguientes Figuras de 1 a 10.

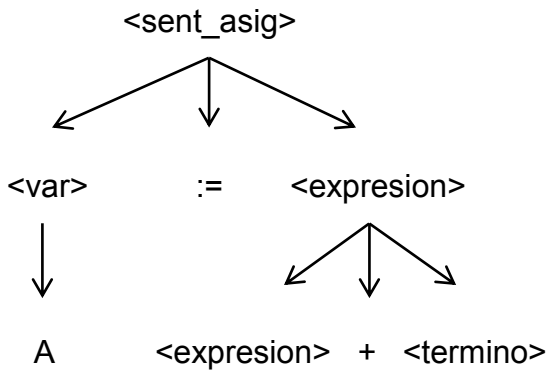
1)



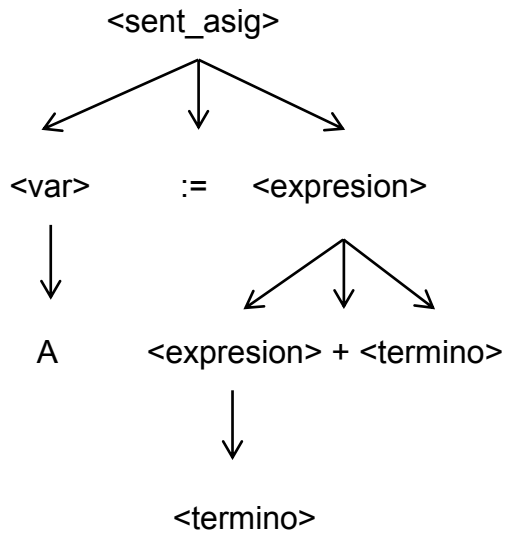

2)



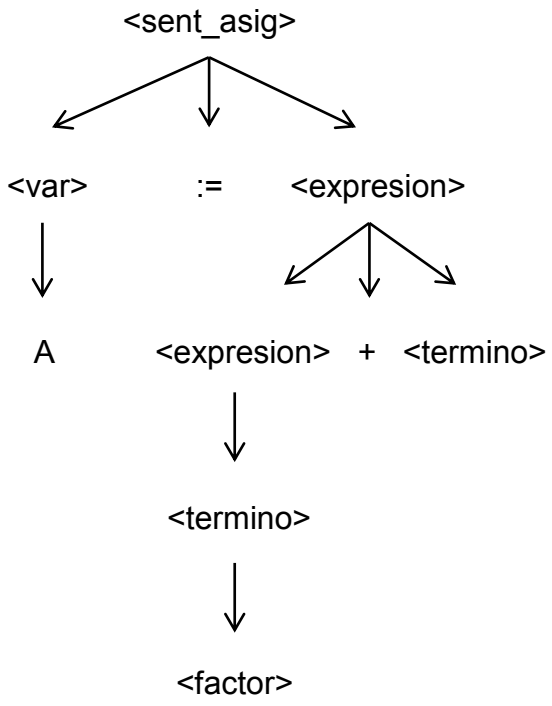
3)


<sent_asig>
<var>
:=
<expresion>
A
<expresion>
+
<termino>

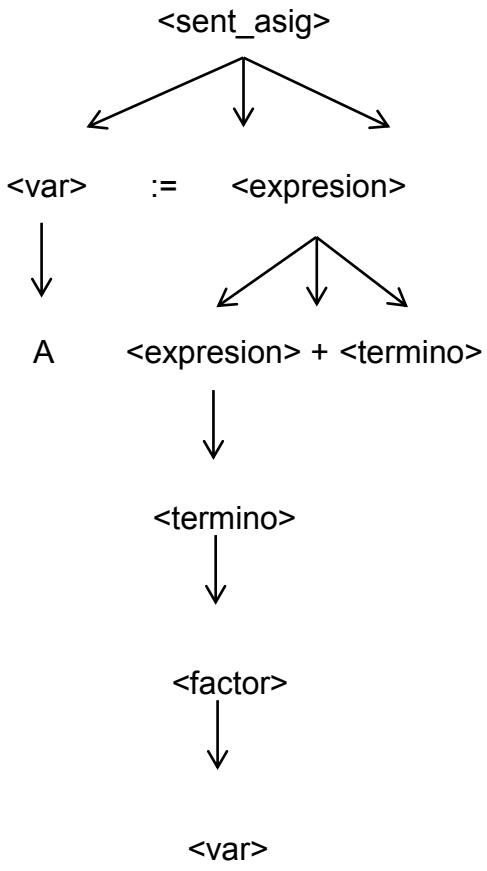
4)


<sent_asig>
<var>
:=
<expresion>
A
<expresion> + <termino>
<termino>

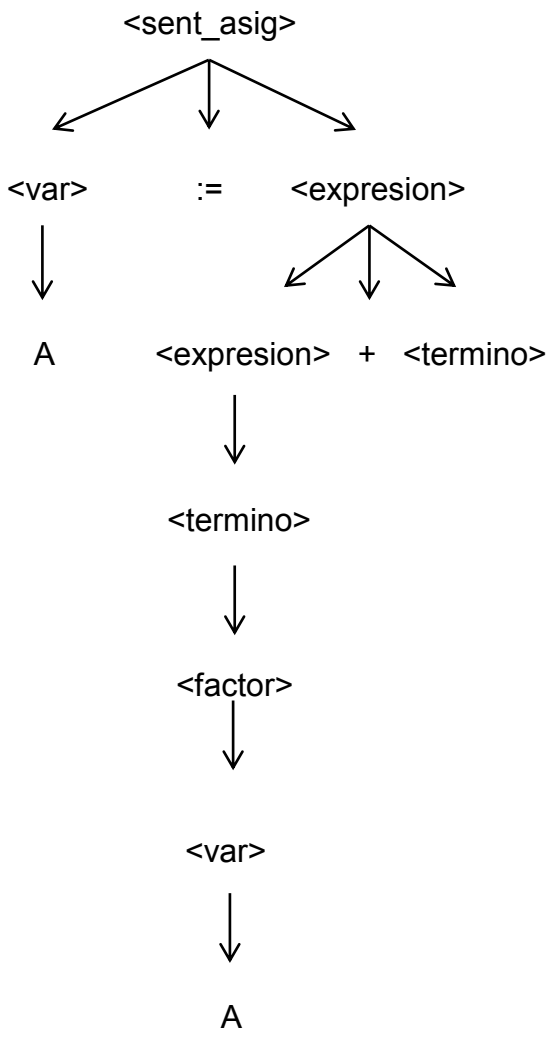
5)


<sent_asig>
<var>
:=
<expresion>
A
<expresion>
+
<termino>
<termino>
<factor>

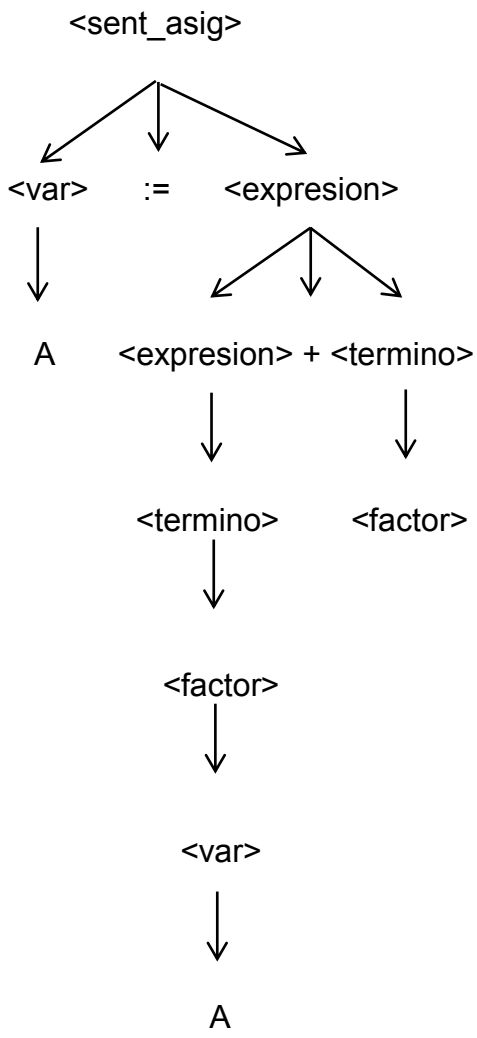
6)


<sent_asig>
<var>
:=
<expresion>
A
<expresion> + <termino>
<termino>
<factor>
<var>

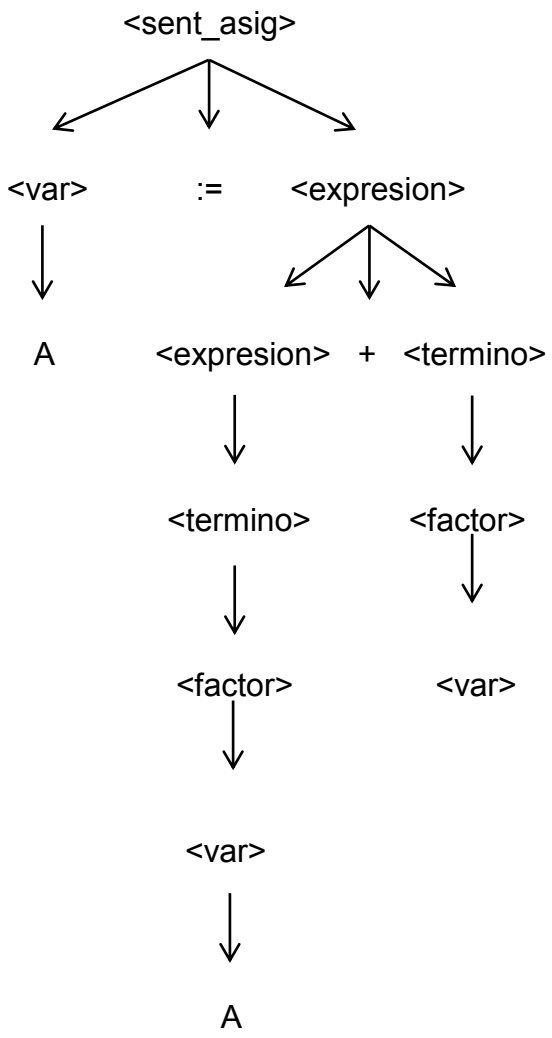
7)


<sent_asig>
<var>
:=
<expresion>
A
<expresion>
+
<termino>
<termino>
<factor>
<var>
A

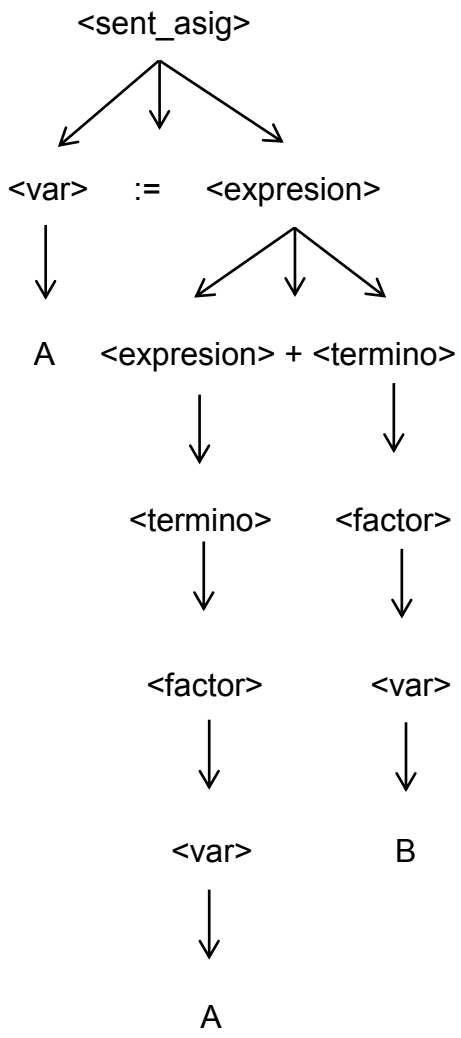
8)


<sent_asig>
<var>
:=
<expresion>
A
<expresion>
+
<termino>
<termino>
<factor>
<factor>
<var>
A

9)


<sent_asig>
<var>
:=
<expresion>
A
<expresion>
+
<termino>
<termino>
<factor>
<factor>
<var>
<var>
A

10)


<sent_asig>
<var>
:=
<expresion>
A
<expresion>
+
<termino>
<termino>
<factor>
<factor>
<var>
<var>
B
A

Ejemplo 2

El árbol de derivación correspondiente a la sentencia C:= D – E * F se muestra en la Figura 4.

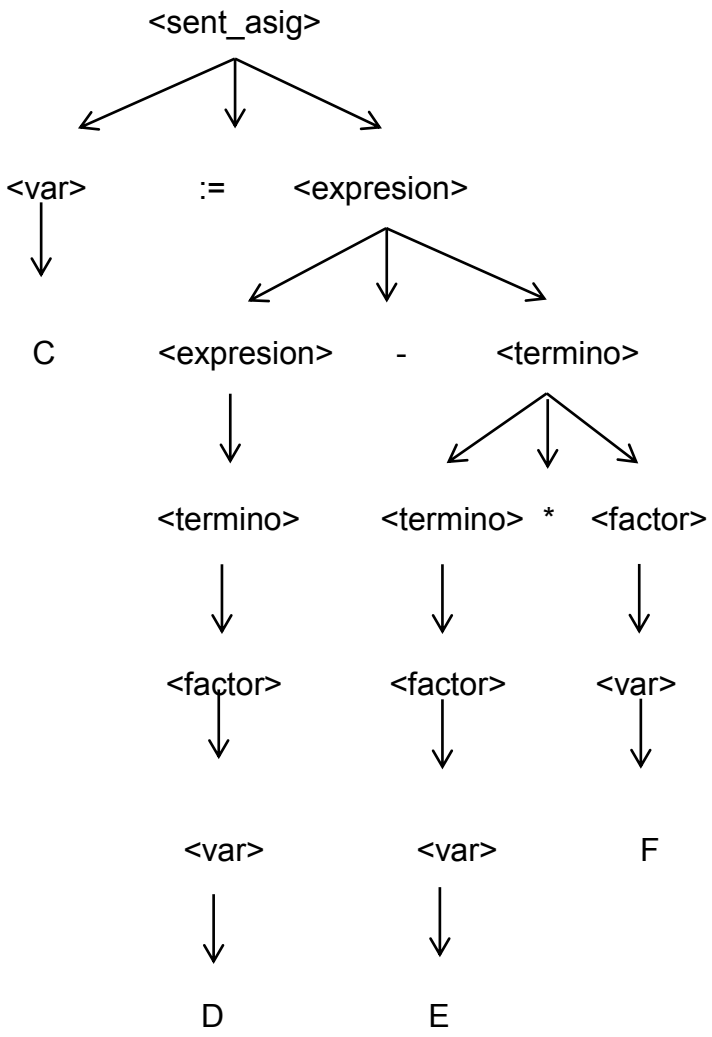



*Figura 4 Árbol de derivación ejemplo*

### 3.2.3 Ambigüedad

Una gramática es ambigua cuando tiene la capacidad de producir, para la misma cadena, más de una derivación o árbol, ya sea por la izquierda o por la derecha. Es decir, se puede sustituir los no terminales en más de un orden y llegar al mismo conjunto de símbolos terminales.
Ejemplo:

Considerando que se tiene la siguiente regla gramatical:

$E \rightarrow E + E$
$E \rightarrow E – E$
$E \rightarrow x \mid y \mid z$

La forma de llegar a la expresión y+x-z, podría ser:

Derivación por la izquierda se muestra en la Figura 5:

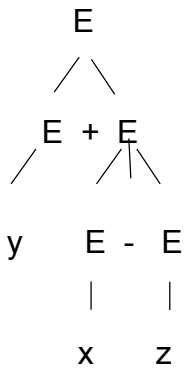



*Figura 5 representación gráfica de un árbol con derivación por la izquierda.*

Retomando el ejemplo anterior tenemos que la derivación por la derecha se muestra en la Figura 6:

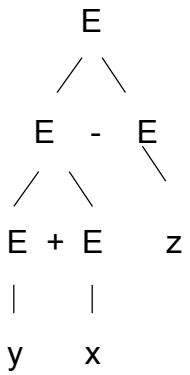



*Figura 6 Representación gráfica de un árbol con derivación por la derecha.*

Las gramáticas no ambiguas tienen la característica de que solo pueden seguir una derivación, por lo que puede ser utilizadas para la construcción de compiladores, gramáticas recursivas izquierdas.

## 2.3 Autómatas tipo pila (AP)

Los autómatas tipo pila son máquinas reconocedoras que aceptan lenguajes de contexto libre, su estructura es semejante a los autómatas finitos, un Autómata tipo Pila difiere de un Autómata Finito en que le primero utiliza una memoria auxiliar con estructura de pila para llevar a cabo sus transiciones.

Un autómata tipo pila es un dispositivo que tiene acceso a:

- ✓ Una secuencia de símbolos de entrada, que en general se representa por una cinta que se desplaza frente a un mecanismo de captación de dichos símbolos.
- ✓ El símbolo superior de una memoria en pila (LIFO).

Un autómata a pila se encuentra en cada momento en un estado determinado y el estado siguiente depende de los tres elementos siguientes:

- ✓ estado actual.
- ✓ símbolo de entrada.
- ✓ símbolo superior de la pila.

Un autómata tipo pila puede realizar dos tipos de operaciones elementales:

1. Dependientes de la entrada.

Se lee la cinta y se avanza la cabeza lectora,
En función:

- ✓ del estado actual (qi)
- ✓ del símbolo leído en la cinta (a)
- ✓ del símbolo en la cima de la pila (Z)

Se pasa a un nuevo estado, se elimina el elemento Z de la cima de la pila y se introduce en su lugar una cadena de símbolos.

2. Independientes de la entrada.

Las mismas operaciones que en el caso anterior, sólo que no se lee la cinta, ni se avanza la cabeza lectora. Se maneja la pila sin la información de entrada.

Definición formal

Un autómata tipo pila es una séptupla M=(Q, Σ, ∆, q0 ,δ, F) donde:
Q= conjunto finito de estados
Σ= alfabeto de entrada

∆= alfabeto de pila

q0∈Q estado inicial

F⊆Q, F≠∅, conjunto de estados finales

δ es la función de transición, definida de la siguiente forma

$$Qx(\Sigma \cup \{\lambda\}) \times (\Delta \cup \{\lambda\}) \xrightarrow{\delta} P(Qx\Delta^*)$$

Representación gráfica de un AP.

Es similar a la de un autómata finito:

- ✓ Dibujamos un círculo por cada estado no final y un doble circulo por cada estado final.
- ✓ Marcamos el estado inicial con una flecha de entrada, sin etiquetar.
- ✓ Por cada (r, w) ∈ δ (q, a, Z) dibujamos una flecha de q a r etiquetada a,Z;w

El grafico de un autómata tipo pila lo describe completamente

Ejemplo

Sea M8 = (Q, Σ, ∆, q0, δ, F) con Q = {p, q, r, s}, Σ = {0,1}, ∆ = {#, a}, p estado inicial F = {s}, y función de transición δ definida de la manera siguiente:

δ(p,λ,λ)={(q,#)}

δ(q,0,λ)={(q,a)}

δ(q,1,a)={(r,λ)}

δ(r,1,a)={(r,λ)}

δ(r,λ,#)={(s,λ)}

La representación gráfica de este autómata tipo pila se verá en la figura 7:

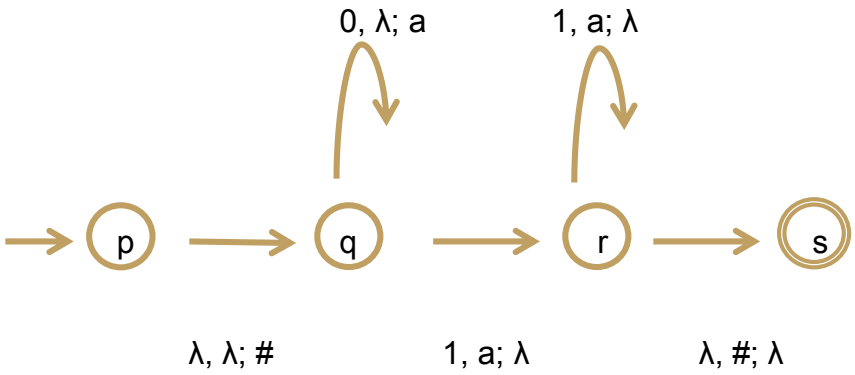



*Figura 7 Representación gráfica de un autómata tipo pila (AP).*

Se observa en el autómata que antes que nada marca el principio de la pila con # sin leer nada en la cinta y pasa al estado q; a continuación por cada 0 que lee en la entrada inserta una 'a' en la pila, sin cambiar de estado; cuando lee el primer 1, extrae una 'a' de la pila y pasa al estado r; tiene que seguir leyendo símbolos 1 a la vez que extrae los símbolos 'a' de la pila; cuando el número de 1 leídos iguala al número de 0 vuelve a aparecer el símbolo # en la cima de la pila, lo extrae y se alcanza el estado final s. Por tanto las únicas palabras que podrán alcanzar el estado final son las de la forma $0^n1^n$ con n>0

## 3.5 Jerarquía de Chomsky

En 1956, Noam Chomsky clasificó las gramáticas en cuatro tipos de lenguajes y esta clasificación es conocida como la jerarquía de Chomsky, en la cual cada lenguaje es descrito por el tipo de gramática generado. Estos lenguajes sirven como base para la clasificación de lenguajes de programación. Los cuatro tipos son: lenguajes recursivamente e numerables, lenguajes sensibles al contexto, lenguajes libres de contexto y lenguajes regulares. Dichos lenguajes también se identifican como lenguajes de tipo 0, 1, 2 y 3. (Academia, 2015)

Gramáticas Tipo 0 (sin restricciones, recursivas).
Incluyen todas las gramáticas formales. Generan todos los lenguajes que pueden ser reconocidos por una máquina de Turing.

Contiene reglas que transforman un número arbitrario de símbolos no vacío en otro número arbitrario, posiblemente vacío, de símbolos.

Ejemplos:

aAB -> a

bB -> aBC

Gramáticas Tipo 1 (dependientes de contexto)
Generan los lenguajes dependientes de contexto. Contienen reglas de producción de la forma:
$\alpha A \beta \rightarrow \alpha \gamma \beta$
A es un no terminal
$\alpha$, $\beta$ y $\gamma$ son cadenas de terminales y no terminales.
$\alpha$ y $\beta$ pueden ser vacíos, pero $\gamma$ ha de ser distinto del vacío.

Se denominan gramáticas dependientes del contexto, porque, como se observa, A puede ser sustituido por $\gamma$ si está acompañada de $\alpha$ por la izquierda y de $\beta$ por la derecha.

Estos lenguajes son todos los lenguajes que pueden ser reconocidos por una máquina de Turing no determinista. (Autómatas lineales acotados)

Una gramática es tipo 1 monotónica si no contiene reglas donde el lado izquierdo consista en más símbolos que el lado derecho:

Ejemplo:
|A| <= |B| A ->B

AF -> ABab

S -> aAB

Una gramática es tipo 1 sensible al contexto si todas sus reglas son sensibles al contexto. Es decir, una regla es sensible al contexto si solo un símbolo no terminal en el lado izquierdo se reemplaza por otro símbolo mientras el resto permanece inalterado y en el mismo orden.

bQc -> bbcc

o la gramática que genera el mismo número de a, b, c:

Ss -> abc | aSQ
bQc -> bbcc
cQ -> Qc

Gramáticas Tipo 2 (independientes de contexto, libre de contexto)
Generan los lenguajes libres de contexto. Están definidas por reglas de la forma:
$A \rightarrow \gamma$
A es un no terminal
$\gamma$ es una cadena de terminales y no terminales.

Se denominan independientes de contexto porque A puede sustituirse por $\gamma$ independientemente de las cadenas por las que esté acompañada.

Los lenguajes independientes de contexto constituyen la base teórica para la sintaxis de la mayoría de los lenguajes de programación. Definen la sintaxis de las

declaraciones, las proposiciones, las expresiones, etc. (es decir, la estructura de un programa).

Estos lenguajes son todos los lenguajes que pueden ser reconocidos por los autómatas de pila.

Son gramáticas libres de contexto cuando en su lado izquierdo siempre aparece un único no-terminal

A -> a|b|c

B -> A|CaA

C -> A,C|A

Gramáticas regulares (Tipo 3)

Generan los lenguajes regulares (aquellos reconocidos por un autómata finito). Son las gramáticas más restrictivas. El lado derecho de una producción debe contener un símbolo terminal y, como máximo, un símbolo no terminal. Estas gramáticas pueden ser:

Lineales a derecha, si todas las producciones son de la forma

$$A \rightarrow aB \text{ o } A \rightarrow a \quad \begin{cases} A \in N \cup \{S\} \\ B \in N \\ a \in T \end{cases}$$

(en el lado derecho de las producciones el símbolo no terminal aparece a la derecha del símbolo terminal)

Lineales a izquierda, si todas las producciones son de la forma

$$A \rightarrow Ba \text{ o } A \rightarrow a \quad \begin{cases} A \in N \cup \{S\} \\ B \in N \\ a \in T \end{cases}$$

(en el lado derecho de las producciones el símbolo no terminal aparece a la izquierda del símbolo terminal)

En ambos casos, se puede incluir la producción $S \rightarrow \varepsilon$, si el lenguaje que se quiere generar contiene la cadena vacía.

Las gramáticas formales definen un lenguaje describiendo cómo se pueden generar las cadenas del lenguaje. Las expresiones regulares describen los lenguajes regulares.

Son gramáticas regulares o de estados finitos cuando en su lado derecho sólo se contiene un no-terminal y además se encuentra al final de la producción. Esto produce dos clases de reglas:

- ✓ Un no-terminal produce cero o más terminales
- ✓ Un no-terminal produce cero o más terminales seguidos por un no-terminal.

La definición original de Chomsky lo restringe:

- ✓ Un no-terminal produce un terminal
- ✓ Un no-terminal produce un terminal seguido por un no-terminal

Por ejemplo (no-terminales empiezan en mayúsculas):

Sentence -> tom|dick|harry|List

List -> tom List Tail | dick List Tail | harry List Tail

List Tail -> ,List | and tom | and dick | and harry

Las gramáticas regulares solo pueden generar a los lenguajes regulares de manera similar a los autómatas finitos y las expresiones regulares.

Dos gramáticas regulares que generen el mismo lenguaje regular se denominan equivalente. Una gramática formal es un conjunto de reglas para describir cadenas de caracteres, junto con un símbolo inicial desde el cual debe comenzar la reescritura. Por lo tanto, una gramática formal generalmente se piensa como una generadora de lenguaje. Sin embargo, a veces también puede ser usada como la base para un “reconocedor”: una función que determina si una cadena cualquiera pertenece a un lenguaje o es gramaticalmente incorrecta.

Las expresiones regulares permiten representar de manera simplificada un lenguaje regular

Una expresión regular define un patrón (token); una palabra pertenece al lenguaje definido por esa expresión regular si y solo si sigue el patrón.

Una expresión regular que represente un lenguaje debe cumplir dos condiciones:

- Correcta: todas las palabras representados por la expresión regular debe ser parte del lenguaje.
- Completa: toda palabra del lenguaje debe ser representada por la expresión regular.

# Capítulo 3

## FLEX

### 3.1 ANALIZADOR LEXICO

Un analizador léxico es el componente de un compilador que divide el programa fuente en unidades lógicas o sintácticas formadas por uno o más caracteres que tienen un significado. Entre las unidades lógicas o sintácticas se incluyen las palabras clave (por ejemplo, while), identificadores (por ejemplo, cualquier letra seguida de cero o más letras y/o dígitos) y signos como + o =.

La fase de rastreo (scanner), tiene las funciones de leer el programa fuente como un archivo de caracteres y dividirlo en tokens. Los tokens son las palabras reservadas de un lenguaje, secuencia de caracteres que representa una unidad de información en el programa fuente. En cada caso un token representa un cierto patrón de caracteres que el analizador léxico reconoce, o ajusta desde el inicio de los caracteres de entrada. De tal manera es necesario generar un mecanismo computacional que nos permita identificar el patrón de transición entre los caracteres de entrada, generando tokens, que posteriormente serán clasificados. Este mecanismo es posible crearlo a partir de un tipo específico de máquina de estados llamado autómata finito.

Función del analizador léxico

En la primera fase de un compilador. Su principal función consiste en leer la secuencia de caracteres del programa fuente, carácter a carácter, y elaborar como salida la secuencia de componentes léxicos que utiliza el analizador sintáctico. El analizador sintáctico emite la orden al analizador léxico para que agrupe los caracteres y forme unidades con significado propio llamados componentes léxicos (tokens). Los componentes léxicos representan:

Palabras reservadas: if, while, do, ...
Identificadores: variables, funciones, tipos definidos por el usuario, etiquetas, ...
Operadores: =, >, <, >=, <=, +, *, ...
Símbolos especiales: ;, ( ), { }, ...

Constantes numéricas: literales que representan valores enteros y flotantes. Constantes de carácter: literales que representan cadenas de caracteres.

El analizador léxico opera bajo petición del analizador sintáctico devolviendo un componente léxico conforme el analizador sintáctico lo va necesitando para avanzar en la gramática. Los componentes léxicos son los símbolos terminales de la gramática. Suele implementarse como una subrutina del analizador sintáctico. Cuando recibe la orden “obtén el siguiente componente léxico”, el analizador léxico lee los caracteres de entrada hasta identificar el siguiente componente léxico.

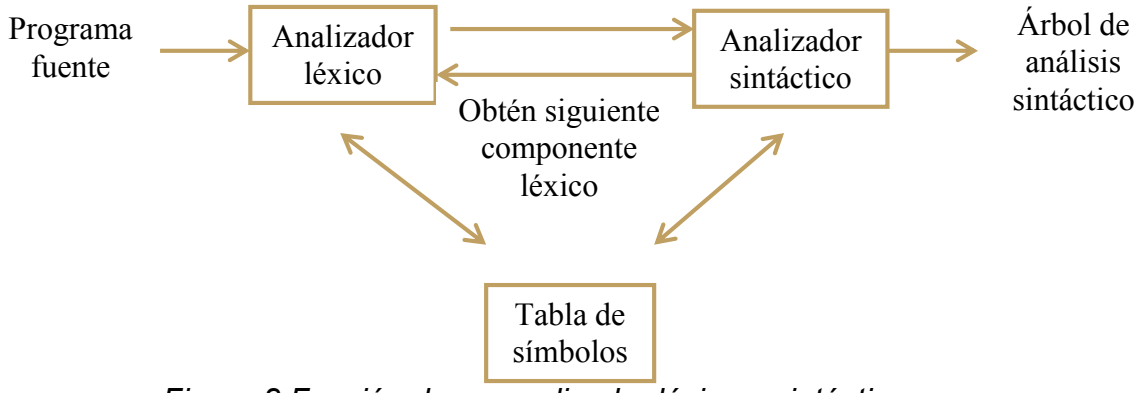



*Figura 8 Función de un analizador léxico y sintáctico*

Además, el analizador léxico es responsable de:

- ✓ Manejo de apertura y cierre de archivo, lectura de caracteres y gestión de posibles errores de apertura.
- ✓ Eliminar comentarios, espacios en blanco, tabuladores y saltos de línea.
- ✓ Inclusión de archivos y macros.
- ✓ Contabilizar número de líneas y columnas para emitir mensajes de error.

## 3.2 FLEX.

Flex es una herramienta para generar escáners: programas que reconocen patrones léxicos en un texto. En otras palabras, Flex se encarga de convertir esos patrones léxicos en tokens que pueden servir para estructurar una gramática (en el caso de un compilador) o para colorear código (En caso de un coloreador de código). Flex busca concordancias en un fichero de entrada y ejecuta acciones asociadas a estas expresiones. Es compatible casi al 100% con LEX (Lexical Analyzer), una herramienta clásica de Unix para la generación de analizadores léxicos, pero es un desarrollo diferente realizado por GNU bajo licencia GPL.

Flex genera el código C equivalente a la descripción pedida por el usuario, código que está contenido en el archivo fuente “lex.yy.c”. Este archivo contiene el código fuente de un autómata finito determinista, capaz de reconocer las expresiones regulares entregadas por el usuario en el archivo de entrada.

Después de compilar el archivo, obtendremos un ejecutable que contendrá a nuestro reconocedor. Cuando es ejecutado, el reconocedor analizará su entrada (ya sea un archivo o texto por teclado) en busca de texto que calce con las expresiones regulares definidas en el archivo inicial. Cada vez que el reconocedor encuentre un calce entre un texto leído y una regla existente, se ejecutará el código en C correspondiente a esa regla


*Figura 9 Flex se encarga de transformar las especificaciones (reglas y acciones) que nosotros definamos a un código C que representa al analizador.*

Lo anteriormente descrito se puede entender mejor observando la Figura 9, la cual nos muestra el proceso necesario para obtener el código fuente del analizador.

Los ficheros de entrada de Flex (normalmente con la extensión .l) siguen el siguiente esquema:

%%
patrón1 {acción1}
patrón2 {acción2}
...
donde:

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.

Flex lee los ficheros de entrada dados, o la entrada estándar si no se le ha indicado ningún nombre de fichero, con la descripción de un escáner a generar. 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()'. Este fichero se compila y se enlaza con la librería '-lfl' para producir un ejecutable. Cuando se arranca el fichero ejecutable, éste analiza su entrada en busca de casos de las expresiones regulares. Siempre que encuentra uno, ejecuta el código C correspondiente.

### 3.2.1 FORMATO DE FICHERO DE ENTRADA

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

definiciones
%%
reglas
%%
código de usuario

La sección de definiciones contiene declaraciones de definiciones de nombres sencillas para simplificar la especificación del escáner, y declaraciones de condiciones de arranque. Las definiciones de nombre tienen la forma:

nombre definición

El "nombre" es una palabra que comienza con una letra o un subrayado ('_') seguido por cero o más letras, dígitos, '_', o '-' (guión). La definición se considera que comienza en el primer carácter que no sea un espacio en blanco siguiendo al nombre y continuando hasta el final de la línea. Posteriormente se puede hacer referencia a la definición utilizando "{nombre}", que se expandirá a "(definición)". Por ejemplo,

```
DIGITO   [0-9]
ID       [a-z][a-z0-9]*
```

define "DIGITO" como una expresión regular que empareja un dígito sencillo, e "ID" como una expresión regular que empareja una letra seguida por cero o más letras o dígitos. Una referencia posterior a

{DIGITO}+"."{DIGITO}*

es idéntica a

([0-9])+"."([0-9])*
y empareja uno o más dígitos seguido por un '.' seguido por cero o más dígitos.

La sección de reglas en la entrada de flex contiene una serie de reglas de la forma:

patrón  acción

donde el patrón debe estar sin sangría y la acción debe comenzar en la misma línea.

Finalmente, la sección de código de usuario simplemente se copia a 'lex.yy.c' literalmente. Esta sección se utiliza para rutinas de complemento que llaman al escáner o son llamadas por éste. La presencia de esta sección es opcional: Si se omite, el segundo '%%' en el fichero de entrada se podría omitir también.

En las secciones de definiciones y reglas, cualquier texto con sangrado o encerrado entre ' %{ ' y ' %} ' se copia íntegramente a la salida (sin los %{}'s). Los %{}'s deben aparecer sin sangrar en líneas ocupadas únicamente por éstos.

En la sección de reglas, está compuesta por dos columnas una de patrones y otra de acciones. Por cada patrón hay asociada una o más acciones, en donde patrones son expresiones regulares reconocidas, y acciones son códigos C definidos por el programador.

En la sección de definiciones (pero no en la sección de reglas), un comentario sin sangría (es decir, una línea comenzando con "/*") también se copia literalmente a la salida hasta el próximo "*/".

| Métodos, macros y variables | Acción o finalidad |
|---|---|
| Variable yytext | Almacena el token actual |
| Variable yyleng | Almacena la longitud de yytex |
| Macro ECHO | Muestra yytext en stdout |
| Método yylex() | Inicia el reconcomiendo del lexemas |

*Tabla 2 Métodos y macros frecuentes usados en Flex.*

### 3.2.2 PATRONES

Los patrones en la entrada se escriben utilizando un conjunto extendido de expresiones regulares y usando como alfabeto cualquier carácter ASCII. Cualquier símbolo excepto el espacio en blanco, tabulador, cambio de línea y los caracteres especiales se escriben tal cual en las expresiones regulares (patrones) de Flex.

Los caracteres más frecuentes usados por Flex se muestran en la Tabla 3

| | |
|---|---|
| “X” | empareja el carácter ‘x’ |
| “•” | cualquier carácter (byte) excepto una línea nueva |
| “[xyz]” | una "clase de caracteres"; en este caso, el patrón empareja una ‘x’, una ‘y’, o una ‘z’ |
| “[abj-oZ]” | una "clase de caracteres" con un rango; empareja una ‘a’, una ‘b’, cualquier letra desde la ‘j’ hasta la ‘o’, o una ‘Z’ |
| “[^A-Z]” | una "clase de caracteres negada", es decir, cualquier carácter menos los que aparecen en el conjunto. En este caso, cualquier carácter EXCEPTO una letra mayúscula |

| | |
|---|---|
| “[^A-Z\n]” | cualquier carácter EXCEPTO una letra mayúscula o una línea nueva |
| “ r* “ | cero o más r’s, donde r es cualquier expresión regular |
| “ r+ “ | una o más r’s |
| “ r? “ | cero o una r (es decir, "una r opcional") |
| “ r{2,5} “ | entre dos y cinco concatenaciones de r |
| “ r{4} “ | exactamente 4 r’s |
| “ {nombre} “ | la expansión de la definición de "nombre" |
| "[xyz]\"foo" | la cadena literal: [xyz]"foo |
| “ \x “ | si x es una ‘a’, ‘b’, ‘f’, ‘n’, ‘r’, ‘t’, o ‘v’, entonces estamos hablando de que sería una interpretación ANSI-C de \x (por ejemplo \t sería un tabulador) En otro caso, si “x” es usada como literal ‘x' (es usado para indicar operadores tales como “ * “). |
| “ (r) ” | empareja una r; los paréntesis se utilizan para anular la precedencia |
| “ rs “ | la expresión regular r seguida por la expresión regular s; se denomina "concatenación" |
| “ r\|s ” | bien una r o una s |
| “ r/s “ | una r pero sólo si va seguida por una s |

| | |
|---|---|
| “ ^r “ | una r, pero sólo al comienzo de una línea |
| “ r$ “ | una r, pero sólo al final de una línea (es decir, justo antes de una línea nueva). Equivalente a "r/\n". |
| “ <s>r “ | una r, pero sólo en la condición de arranque s |
| “ <s1,s2,s3>r “ | lo mismo, pero en cualquiera de las condiciones de arranque s1, s2, o s3 |

*Tabla 3 Caracteres más usados en Flex.*

Las expresiones regulares en el listado anterior están agrupadas de acuerdo a la precedencia, desde la precedencia más alta en la cabeza a la más baja al final. Aquellas agrupadas conjuntamente tienen la misma precedencia. Por ejemplo:

foo|bar*

es lo mismo que

(foo)|(ba(r*))

Ya que el operador “ * “ tiene mayor precedencia que la concatenación, y la concatenación más alta que el operador “ | ”. Este patrón por lo tanto empareja bien la cadena "foo" o la cadena "ba" seguida de cero o más r's.

Para emparejar "foo" o, cero o más "bar"'s, use:
foo|(bar)*

y para emparejar cero o más "foo"'s o "bar"'s:
(foo|bar)*

### 3.2.3 EMPAREJAMIENTO DE LA ENTRADA

Cuando se ejecuta el analizador generado, este analiza su entrada buscando cadenas que concuerden con cualquiera de sus patrones. Si encuentra más de un coincidencia, toma el que empareje el texto más largo. Si encuentra dos o más coincidencias de la misma longitud, se escoge la regla listada en primer lugar en el fichero de entrada de Flex.

Una vez que se determina la coincidencia, el texto correspondiente a la coincidencia (denominado token) está disponible en el puntero de carácter global yytext, y su longitud en la variable global entera yyleng. Entonces la acción correspondiente al patrón encontrado se ejecuta y luego la entrada restante se analiza para otro emparejamiento.

Si no se encuentra ninguna coincidencia, entonces la regla por defecto se ejecuta: el siguiente carácter de la entrada se considera y se copia a la salida estándar.

### 3.2.4 ACCIONES

Cada patrón en una regla tiene una acción asociada, que puede ser cualquier código en C. El patrón finaliza en el primer carácter de espacio en blanco que no sea una secuencia de escape; lo que queda de la línea es su acción. Si la acción está vacía, entonces cuando el patrón se empareje el token de entrada simplemente se descarta. Por ejemplo, aquí está la especificación de un programa que borra todas las apariciones de "zap me" en su entrada:

```
%%
"zap me"
```

Si la acción contiene un ‘{‘, entonces la acción abarca hasta que se encuentre el correspondiente ‘}’, y la acción podría entonces cruzar varias líneas.

Flex es capaz de reconocer las cadenas y comentarios de C y no se dejará engañar por las llaves que encuentre dentro de éstos, pero aun así también permite que las acciones comiencen con ‘%{‘ y considerará que la acción es todo el texto hasta el siguiente ‘%}’(sin tener en cuenta las llaves ordinarias dentro de la acción).

Las acciones pueden incluir código C arbitrario, incluyendo sentencias return para devolver un valor desde cualquier función llamada ‘yylex()’. Cada vez que se llama a ‘yylex()’ ésta continúa procesando tokens desde donde lo dejó la última vez hasta que o bien llegue al final del fichero o ejecute un return.

### 3.2.5 EL ANALIZADOR GENERADO

La salida de Flex es el fichero ‘lex.yy.c’, que contiene la función de análisis ‘yylex()’, varias tablas usadas por esta para emparejar tokens, y unas cuantas rutinas auxiliares y macros. Por defecto, ‘yylex()’se declara así

```
int yylex()
{ ...
aquí van varias definiciones y las acciones ...
}
```

Siempre que se llame a ‘yylex()’, éste analiza tokens desde el fichero de entrada global yyin ( que por defecto es igual a stdin). La función continúa hasta que alcance el final del fichero (punto en el que devuelve el valor 0) o una de sus acciones ejecute una sentencia return.

Ejemplo:

El siguiente analizador cuenta el número de líneas y caracteres que hay en el fichero de entrada.

```
%{
int num_lineas = 0, num_caracteres = 0;
%}

%%

\n      num_lineas++; num_caracteres++;
.       num_caracteres++;

%%
int yywrap()
{
   return 1;
}

main()
{
   while (yylex());
   printf("No. lineas = %d, No. caracteres = %d\n", num_lineas, num_caracteres);
}
```

# Capítulo 4

## EJEMPLOS INTERESANTES

### EJEMPLO 1

Construiremos un programa simple que reorganice diferentes tipos de palabras en inglés. Nosotros empezaremos por describir partes del lenguaje Ingles (verbos).

Empezaremos por una lista de conjuntos de verbos para reorganizar:

| | | | |
|---|---|---|---|
| Is | am | are | were |
| Was | be | being | been |
| Do | does | did | will |
| Would | should | can | could |
| Has | have | had | go |

Se abrirá un archivo de texto, para este ejemplo será el block de notas, el código se guardará con la extensión “ .l ” con el nombre “ verb ” quedara de la siguiente manera: “ verb.l ”

Codigo:

```
%{

/*
* ejemplo reorganizar verbos
*/
%}
%%
```

```
[\t]+  /* ignora espacios en blanco */
is |
am |
are |
was |
were |
was |
be |
being |
been |
do |
does |
did |
wil |
would |
should |
can |
could |
has |
could |
has |
have |
had |
go      {printf("%s : es un verbo\n", yytext);}
[a-zA-Z]+      {printf("%s : no es un verbo, probar de nuevo\n", yytext);}

.|\n {ECHO;}
%%

main ()
```

```
{
        yylex();
}
```

Cuando se ejecuta el analizador léxico, una vez que reconoce un verbo, se ejecuta la acción, una declaración en C “printf”. El arreglo yytext contiene el texto que coincidió con el patrón. Esta acción imprimirá el verbo reconocido seguido de la cadena ": es un verbo".

Las últimas dos reglas:

```
[a-zA-Z]+       {printf("%s : no es un verbo, probar de nuevo\n", yytext);}
.|\n {ECHO;}
```

Cuando el patrón no coincide la acción imprimirá el texto seguido de la cadena es: “no es un verbo, probar de nuevo”

Para ejecutar en sistema Linux[1]:
FLEX[2] se puede descargar e instalar de dos formas en sistema Linux:

1. Por medio de consola “terminal”[3].
2. De modo gráfico.

Para más detalle la descarga e instalación de la herramienta FLEX, ver en el ANEXO: INSTALACION DE FLEX Y BISON EN LINUX Y WINDOWS.

También es necesario un compilador en este caso se usara el compilador GCC[4] de Unix. Ver ANEXO MANUAL COMPILADOR gcc.

[1] Distribución usada del sistema operativo Linux es Debian versión 7
[2] La versión de FLEX usada para este trabajo de investigación es 2.5
[3] La consola o terminal (Shell) es un programa informático donde interactúa el usuario con el sistema operativo mediante una ventana que espera ordenes escritas por el usuario desde el teclado
[4] La versión del compilador GCC usada para este trabajo de investigación es 4.4.

Es necesario realizar y ejecutar varios comandos dentro de la Terminal

1. Nos dirigimos a la ruta donde se encuentra nuestro archivo con extension l (archivo léxico) en mi caso es /home/guillermo/tester.

Escribimos en la consola:
cd /home/guillermo/tester.

Una vez en la ruta deseada seguimos con el siguiente paso:

2. Ejecutamos el comando  lex ejemplo.l

Entonces escribimos en consola:
lex verb.l

Con esto se nos generara un archivo fuente en C nuevo llamado lex.yy.c en la ruta donde nos encontramos en mi caso es home/guillermos/tester

Ultimo paso para crear un programa ejecutable:

3. Escribir el comando cc -w lex.yy.c –ll para llamar al compilador “gcc”, generara un archivo llamado a.out que será nuestro ejecutable.

Lex traslada las especificaciones lex dentro de un archivo fuente C llamado lex.yy.c el cual nosotros compilamos y vinculamos con la librería –ll

Escribimos:
cc –w lex.yy.c –ll

donde:

- cc es el llamado al compilador de GCC

- -w es la opción para decirle al compilador que muestre todos los mensajes de error y advertencia del compilador, incluso algunos cuestionables pero en definitiva fáciles de evitar escribiendo el código con cuidado.
- lex.yy.c es el archivo C a compilar
- -ll es un opción para decirle al compilador de GCC que se usara la librería " l " es decir la librería de Flex

Para ejecutar nuestro ejecutable el analizador, en la terminal debemos escribir ./a.out

En la Figura 10 veremos todos los pasos que describimos anteriormente para compilación y ejecución del ejemplo verbos en sistema Linux

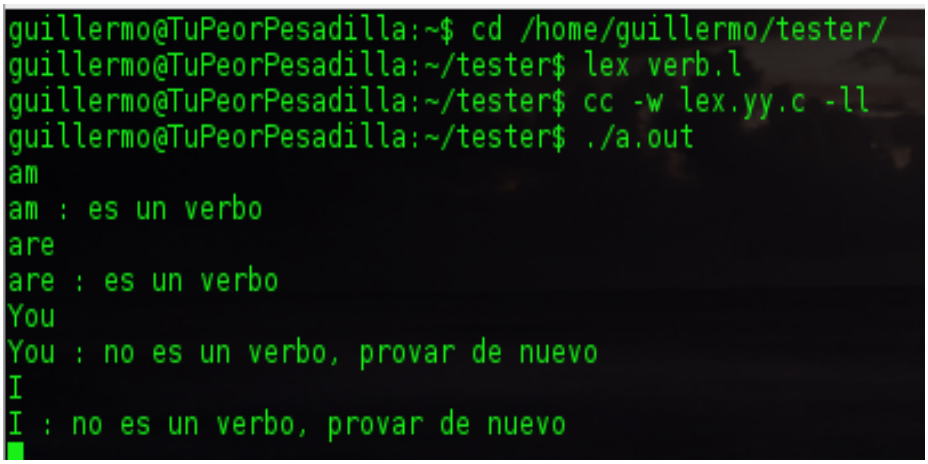



*Figura 10 Compilación y ejecución en sistema Linux del ejemplo verbos en inglés.*

Para ejecutar en sistema Windows[5].

Es necesario realizar los siguientes pasos:

1. Se guardará el archivo en la dirección donde se encuentra los archivos binarios de Flex en mi caso la dirección es C:\GnuWin32\bin se guardara con

[5] Versión utilizada fue Windows 7

la extensión " .l " quedara de la siguiente manera: verb.l , y abrimos la consola de MS-DOS (CMD.exe[6]) como administrador.

2. Ejecutamos la herramienta Flex y escribimos en el símbolo de sistema

Flex verb.l

Como en sistema Linux, generara un archivo fuente en C llamado lex.yy.c en la ruta donde nos encontramos en mi caso es C:\GnuWin32\bin.

3. Copiamos el archivo generado (lex.yy.c) a una nueva dirección, donde se encuentra los archivos binarios del compilador GCC en mi caso la dirección es C:\MinGW\bin.

4. Nos dirigimos a la dirección donde copiamos el archivo C (C:\MinGW\bin).

Escribimos en el símbolo de sistema o CMD

cd C:\MinGW\bin

5. Ejecutamos el comando: gcc lex.yy.c -lfl -o verbos

Donde

- gcc es el comando para señalar que se trabajara con el compilador gcc.
- lex.yy.c es el archivo C a compilar.
- -lfl es otra opción para decirle al compilador de GCC que se usara la librería " fl " es decir la librería de Flex
- -o verbos es una opción para decirle al compilador que el archivo ejecutable (.exe[7]) llevara como nombre "verbos".

Con esto se nos generara un ejecutable con el nombre que le indicamos al compilador.

[6] CMD es una abreviatura que significa command (en inglés) este comando sirve para abrir la consola de MS-DOS o símbolo de sistema en sistemas operativos Windows.

[7] Es una extensión que se refiere a un archivo ejecutable de código reubicable, es decir, sus direcciones de memoria son relativas.

Por último, para ejecutar el archivo .exe escribimos en el CMD: verbos.exe

En la Figura 11 se observan todos los pasos que se describieron anteriormente para compilación y ejecución del ejemplo verbos en sistema Windows.

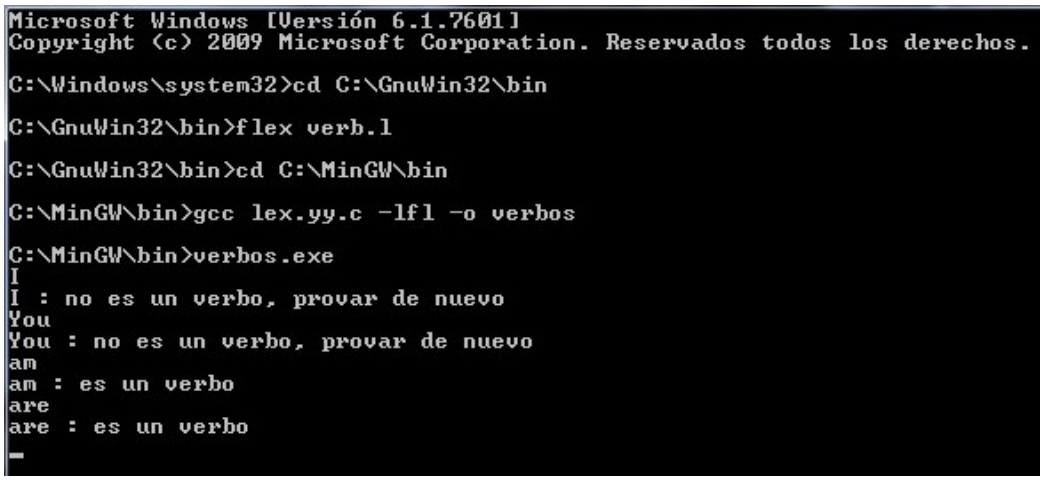



*Figura 11 Compilación y ejecución del ejemplo verbos en sistema Windows 7.*

EJEMPLO 2

En el siguen ejemplo se muestra el uso de Flex para reconocer patrones de expresiones regulares básicas, que reconoce cualquier número entero y cualquier palabra formada por letras y mayúsculas de la “a” a la “z”, sin importar si son mayúsculas o minúsculas.

Codigo:

```
%{
#include <stdio.h>
int palabra=0, numero=0;
%}
Numero -?[0-9]+
Palabra [a-zA-Z]+
%%
"bye"                   {bye();return 0;}
"quit"                  {bye();return 0;}
"resume"                {bye();return 0;}
{Numero}                {printf("Se leyo nuemro: %d",atoi(yytext)); numero++;}
{Palabra}               {printf("Se leyo la palabra: %s",yytext); palabra++;}
. printf("%s", yytext[0]);

%%
main(){
        printf("Ejemplo2.l\nEste ejemplo, distingue entre un numero entero y palabras.\nIntroduzca bye, quit o resume para terminar.\n");
        yylex();
}

bye(){
        printf("se leyeron %d entradas, de las cuales se reconocieron \n%d\tEnteros\ny\n%d\tPalabras.\n",(palabra+numero), numero, palabra);
```

}

Para ejecutar y compilar el ejemplo en sistema Linux[8] es necesario seguir los siguientes pasos:

Crear el archivo con la extension " .l " y guardarlo con el nombre que deseemos en mi caso yo le pondré ejemplo2.l, se creó con el editor de texto llamado "gedit", se guardó en la siguiente dirección: home/guillermo/tester/

Una vez que este creado y guardado el archivo ejemplo2.l abriremos una terminal y nos dirigiremos a la dirección donde se encuentra nuestro archivo.

Escribiremos
cd home/guillermo/tester/

Una vez en la dirección, haremos uso de la herramienta Flex escribiremos en la terminal:

Flex ejemplo2.l
con esto se generara un archivo fuente C llamado lex.yy.c

por ultimo haremos uso del compilador GCC, escribiremos en la terminal:
cc .w lex.yy.c .ll
donde

- cc le diremos a la terminal que se hara uso del compilador GCC}
- -w es la opción para decirle al compilador que muestre todos los mensajes de error y advertencia del compilador, incluso algunos cuestionables pero en definitiva fáciles de evitar escribiendo el código con cuidado.

[8] Versión usada para demostración del ejemplo 2 se usó Ubuntu 14.04

- lex.yy.c archvo fuente C
- -ll es un opción para decirle al compilador de GCC que se usara la librería “ l ” es decir la librería de Flex

Para ejecutar el analizador, en la terminal debemos escribir ./a.out

En la Figura 12 se observan todos los pasos que se describieron anteriormente para compilación y ejecución del ejemplo ejemplo2.l en sistema Linux

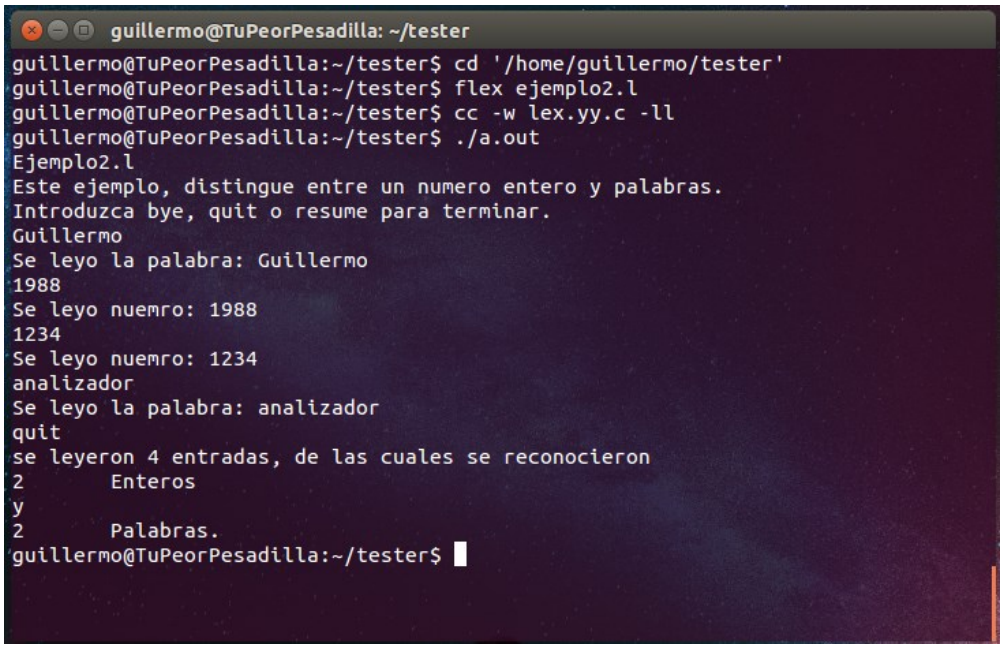



Figura 12 Compilación y ejecución del ejemplo ejemplo2.l en sistema Linux

Para poder ejecutar el analizador del ejemplo 2 en sistema Windows es necesario los siguientes pasos:

Se guarda el archivo con el nombre ejemplo2.l [9] en la dirección deseada en mi caso de se guardó en la siguiente ruta: C:\Users\Resendiz\Desktop\tester

[9] Se puede guardar el archivo con el nombre que se desee siempre y cuando tenga extensión “ .l ”

Una vez guardado se abre el símbolo de sistema de Windows (CMD) y nos dirigimos a la dirección donde se guardó el archivo (C:\Users\Resendiz\Desktop\tester), se escribe:

cd C:\Users\Resendiz\Desktop\tester
Una vez en la dirección se prosigue a hacer uso de la herramienta Flex se escribe dentro del símbolo de sistema lo siguiente.
Flex ejemplo2.l

Con esto Flex generar un archivo fuente C llamado lex.yy.c

Teniendo el archivo antes mencionado continuamos con el uso del compilador GCC para esto debemos escribir en el símbolo de sistema lo siguiente:

gcc –w lex.yy.c –o analizador –lfl

donde

- gcc indicamos que vamos a hacer uso del compilador gcc
- -w es la opción para decirle al compilador que muestre todos los mensajes de error y advertencia del compilador, incluso algunos cuestionables pero en definitiva fáciles de evitar escribiendo el código con cuidado.
- -o es la opción para decirle al compilador que queremos renombrar nuestro ejecutable con el nombre de analizador
- -lfl es un opción para decirle al compilador de GCC que se usara la librería " fl " es decir la librería de Flex

Una vez que tengamos el ejecutable se prosigue a ser ejecutado, para esto debemos escribir en el símbolo de sistema el nombre que le hayamos asignado para el ejemplo se le llamo "analizador" se escribe:

analizador.exe

En la Figura 13 se observan todos los pasos que se describieron anteriormente para compilación y ejecución del ejemplo ejemplo2.l en sistema Windows 7

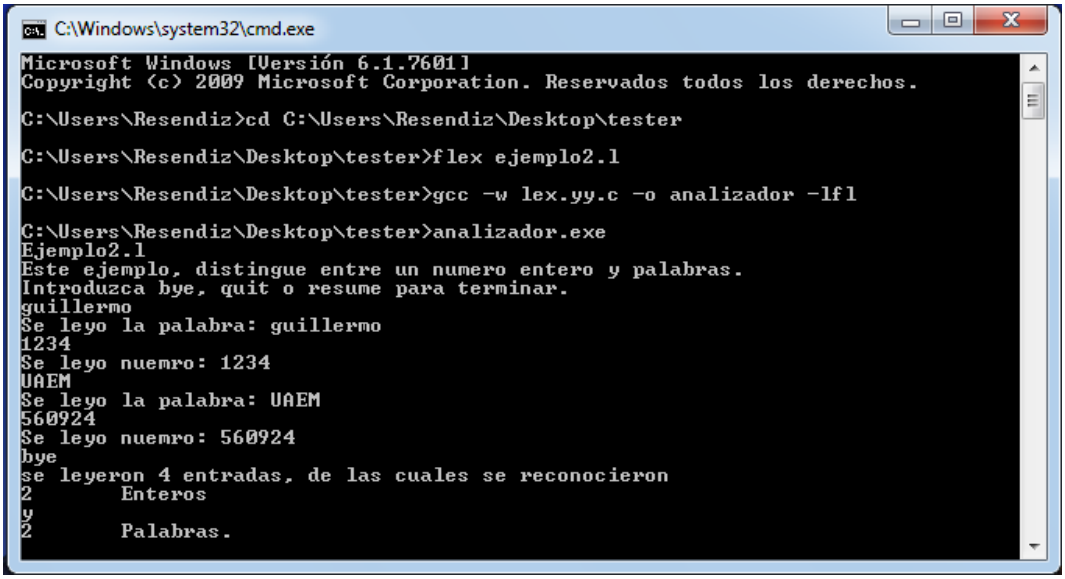



*Figura 13 Compilación y ejecución del ejemplo ejemplo2.l en sistema Windows*

FLEX Y BISON JUNTOS EN LINUX Y WINDOWS

Uno de los usos principales de Flex es como compañero del generador de analizadores sintácticos Bison. Los analizadores de Bison esperan invocar a una rutina llamada 'yylex()' para encontrar el próximo token de entrada. La rutina se supone que devuelve el tipo del próximo token además de poner cualquier valor asociado en la variable global yylval. Para usar Flex con Bison tanto en Windows como en Linux, uno especifica la opción ‘-d’ de yacc para instruirle a que genere el fichero 'y.tab.h' que contiene las definiciones de todos los '%tokens' que aparecen en la entrada de Bison. Entonces este archivo se incluye en el analizador de Flex. Por ejemplo, si uno de los tokens es "TOK_NUMERO", parte de analizador podría parecerse a:

```
%{
#include "y.tab.h"
%}

%%
[0-9]+ yylval = atoi(yytext); return TOK_NUMERO;
```

Bison https://www.gnu.org/software/bison/manual/bison.pdf

Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción para una gramática independiente del contexto en un programa en C que analiza esa gramática. Es compatible al 100% con Yacc, una herramienta clásica de Unix para la generación de analizadores léxicos, pero es un desarrollo diferente realizado por GNU bajo licencia GPL. Todas las gramáticas escritas apropiadamente para Yacc deberían funcionar con Bison sin ningún cambio. Usándolo junto a Flex esta herramienta permite construir compiladores de lenguajes.

Un fuente de Bison (normalmente un fichero con extensión .y) describe una gramática. El ejecutable que se genera indica si un fichero de entrada dado pertenece o no al lenguaje generado por esa gramática. La forma general de una gramática de Bison es la siguiente:

```
%{
declaraciones en C
%}
Declaraciones de Bison
%%
Reglas gramaticales
%%
Código C adicional
```

Los ‘%%’, ‘%{‘ y ‘%}’ son signos de puntuación que aparecen en todo archivo de gramática de Bison para separar las secciones.

Las declaraciones en C pueden definir tipos y variables utilizadas en las acciones. Puede también usar comandos del preprocesador para definir macros que se utilicen ahí, y utilizar #include para incluir archivos de cabecera que realicen cualquiera de estas cosas.

Las declaraciones de Bison declaran los nombres de los símbolos terminales y no terminales, y también podrían describir la precedencia de operadores y los tipos de datos de los valores semánticos de varios símbolos.
Las reglas gramaticales son las producciones de la gramática, que además pueden llevar asociadas acciones, código en C, que se ejecutan cuando el analizador encuentra las reglas correspondientes.

El código C adicional puede contener cualquier código C que desee utilizar. A menudo suele ir la definición del analizador léxico yylex, más subrutinas invocadas por las acciones en las reglas gramaticales. En un programa simple, todo el resto del programa puede ir aquí.

Más adelante se mostrara con ejemplos la forma en cómo se enlazan y como se ejecuta Flex y Bison juntos, tanto en Windows como en Linux.

EJEMPLO 3, FLEX Y BISON JUNTOS (David, 2011)

Para el tercer ejemplo se empleara los dos analizadores juntos (Flex y Bison).

Código:

```
/*****************
  Definiciones: Se colocan las cabeceras, variables y expresiones regulares
********************/

%{
  #include <stdio.h>
  #include <stdlib.h>
  #include "sintactico.tab.h"
  int linea=0;
%}

/*
Creamos todas las expresiones regulares

Creamos la definición llamada DIGITO, podemos acceder esta definición usando
{DIGITO}*/

DIGITO [0-9]
NUMERO {DIGITO}+("."{DIGITO}+)?

%%

 /* Creamos las reglas que reconocerán las cadenas que acepte nuestro scanner y
retornaremos el token a bison con la función return. */

{NUMERO} {yylval.real=atof(yytext); return(NUMERO);}
```

```
"="         {return(IGUAL);}
"+"         {return(MAS);}
"-"         {return(MENOS);}
";"         {return(PTOCOMA);}
"*"        {return(POR);}
"/"         {return(DIV);}
"("         {return(PAA);}
")"         {return(PAC);}
"\n"       {linea++;}
[\t\r\f] {}
" "                        {}
```

/* Si en nuestra entrada tiene algún carácter que no pertenece a las reglas anteriores, se genera un error léxico */

```
.                          {printf("Error lexico en linea %d",linea);}

%%

/*
Código de Usuario
```

Aquí podemos realizar otras funciones, como por ejemplo ingresar símbolos a nuestra tabla de símbolos o cualquier otra accione del usuario.
Todo lo que el usuario coloque en esta sección se copiara al archvo lex.yy.c tal y como esta.
*/

Guardamos el archivo como lexico.l.

Luego creamos un nuevo archivo y colocamos el siguiente código.

```
%{
/*******************
Declaraciones en C
**********************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
extern int yylex(void);
extern char *yytext;
extern int linea;
extern FILE *yyin;
void yyerror(char *s);
%}

/************************
Declaraciones de Bison
*************************/
/*  Especifica la colección completa de tipos de datos para poder usar varios tipos de datos en los terminales y no terminales*/

%union
{
float real;
}

/* Indica la producción con la que inicia nuestra gramática*/
%start Exp_l
/* Especificación de termines, podemos especificar también su tipo  */
%token <real> NUMERO
%token MAS
```

```
%token MENOS
%token IGUAL
%token PTOCOMA
%token POR
%token DIV
%token PAA
%token PAC

/* No Terminales, que también podemos especificar su tipo */
%type <real> Exp
%type <real> Calc
%type <real> Exp_l

/*  Definimos las precedencias de menor a mayor */

%left MAS MENOS
%left POR DIV
%%

/**********************
Reglas Gramaticales
**********************/

Exp_l: Exp_l Calc
                |Calc
                        ;
Calc    : Exp PTOCOMA {printf ("%4.1f\n",$1)}
```

/* Con el símbolo de $$ asignamos el valor semántico de toda la acción de la derecha y se la asignamos al no terminal de la izquierda, en la siguiente regla, se la asigna a Exp.

Para poder acceder al valor de los terminales y no terminales del lado derecho usamos el símbolo $ y le concatenamos un número que representa la posición en la que se encuentra es decir si tenemos

A --> B NUMERO C

Si queremos usar el valor que tiene el no terminal B usamos $1, si queremos usar el valor que tiene NUMERO usamos $2 y asi sucesivamente.

```
*/
Exp :           NUMERO {$$=$1;}
                        |Exp MAS Exp {$$=$1+$3;}
                        |Exp MENOS Exp {$$=$1-$3;}
                        |Exp POR Exp {$$=$1*$3;}
                        |Exp DIV Exp {$$=$1/$3;}
                        |PAA Exp PAC {$$=$2;}
                        ;
%%
/********************
Codigo C Adicional
**********************/
void yyerror(char *s)
{
  printf("Error sintactico %s",s);
}


int main(int argc,char **argv)
{
  if (argc>1)
        yyin=fopen(argv[1],"rt");
  else
```

```
        yyin=stdin;
  yyparse();
  return 0;
}
```

Guardamos este archivo con el nombre “sintáctico” con la extensión “ .y “ y con eso ya tenemos nuestro scanner y nuestro parser terminado.

Para compilar estos archivos en Windows usamos los siguientes comandos:

1. Primero tenemos que dirigirnos a la ruta donde se encuentran nuestros archivos en mi caso se guardaron en la siguiente dirección C:\user\Guillermo\Documents\tester\Ejemplo_FlexBison y abrimos el CMD.

2. Hacemos uso de la herramienta Bison, en CMD escribimos lo siguiente:

```
bison -d sintactico.y
```

Bison generara dos archivos, uno llamado sintáctico.tab.c que es que contiene la gramatica que nosotros indicamos y el archivo llamado t.tab.h. El parámetro –d, crea el fichero t.tab.h, que contiene los identificadores de los tokens de bison usados por Flex.

3. Hacemos uso de la herramienta Flex en CMD escribimos lo siguiente:

```
flex lexico.l
```

Flex generara un archivo fuente en C llamado lex.yy.c este archivo lo encontramos en la misma ruta donde se encuentra el archivo Flex (C:\user\Guillermo\Documents\tester\Ejemplo_FlexBison)

4. Compilando archivos generados y crear ejecutable, para este paso ya tenemos los siguientes archivos: lex.yy.c, sintáctico.tab.c y t.tab.h en la dirección C:\user\Guillermo\Documents\tester\Ejemplo_FlexBison. Escribimos en CMD lo siguiente:

cc lex.yy.c sintactico.tab.c -o calcu –lfl –lm

donde:

- cc comando para indicarle a CMD que vamos a usar el compilador de GCC.
- lex.yy.c archivo fuente en C
- sintáctico.tab.c archivo fuente en C
- -o calcu es la opción para decirle al compilador que el archivo ejecutable (.exe) llevara como nombre “calcu”.
- -lfl y –lm son enlaces a librerías que se usa para poder crear exitosamente el archivo ejecutable “fl” es para la librería Flex y “m” para la Liberia de matemáticas.

Con esto nos genera un ejecutable llamado calcu.

En la Figura 14 veremos todos los pasos que describimos anteriormente para compilación del ejemplo calcu en sistema Windows 7.

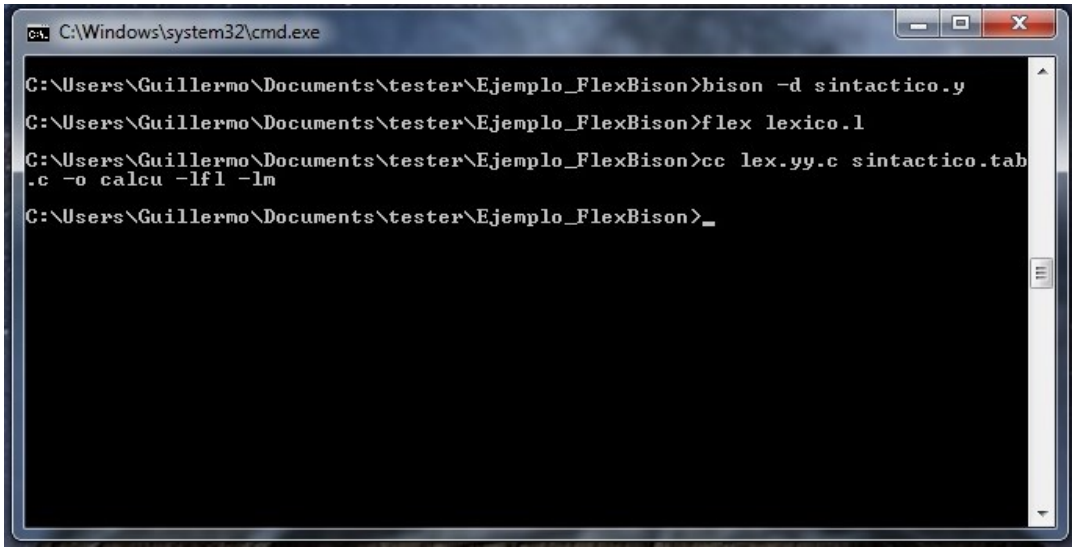



*Figura 14 Compilación del ejemplo calcu en Windows 7*

Para ejecutar este ejemplo desde la consola CMD escribimos:

calcu.exe

En la Figura 15 veremos la ejecución del ejemplo “calcu” en sistema Windows 7.

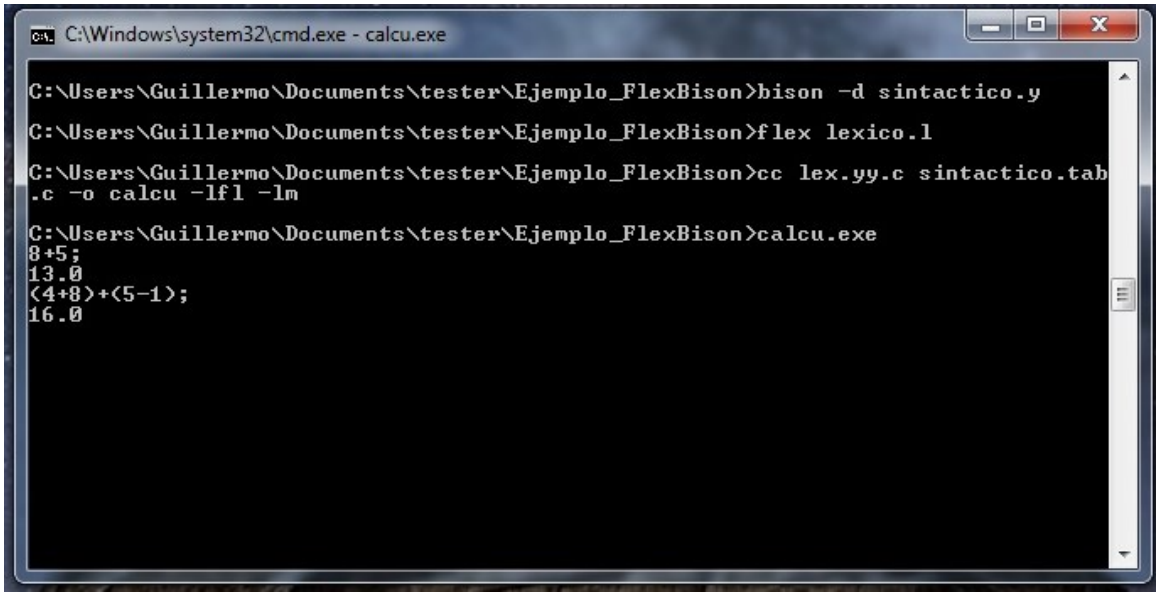



Figura 15 Ejecución del ejemplo calcu en Windows 7

Compilación y Ejecución del ejemplo calcu en Sistema Linux.

Para ejecutar nuestros archivos en un sistema Linux como Debian lo primero que tenemos que hacer es abrir una “terminal” o “consola”, dirigirnos a la dirección donde se encuentran nuestros archivos en mi caso se encuentran en la siguiente dirección: home/Guillermo/Documents/Ejemplo_FlexBison.

Escribimos en la terminal:

cd home/Guillermo/Documents/Ejemplo_FlexBison.

Una vez que ya nos encontremos en esa dirección ejecutamos los siguientes comandos en:

Para hacer uso de la herramienta Bison debemos escribir en la terminal lo siguiente:

bison –d sintáctico.y
Bison generara como en el sistema Windows, dos archivos, uno llamado sintáctico.tab.c que es que contiene la gramatica que nosotros indicamos y el archivo llamado t.tab.h. El parámetro –d, crea el fichero t.tab.h, que contiene los identificadores de los tokens de bison usados por flex.

Para hacer uso de la herramienta Flex en la terminal escribiremos lo siguiente:

flex léxico.l

Flex, como en el sistema windows generara un archivo fuente en C llamado lex.yy.c este archivo lo encontramos en la misma ruta donde se encuentra el archivo Flex es decir home/Guillermo/Documents/Ejemplo_FlexBison.

Con esto nos generara los archivos lex.yy.c y sintáctico.tab.c necesarios para crear el ejecutable.

Para generar el ejecutable debemos escribir en la terminal:

gcc lex.yy.c sintáctico.tab.c /usr/lib/x86_64-linux-gnu/libfl.a –lfl –o calculadora

donde:

- gcc le indicamos a la terminal que vamos a hacer uso del compilador GCC
- lex.yy.c es el archivo fuente C de Flex
- sintactico.tab.c archvio fuente C de Bison
- /usr/lib/x86_64-linux-gnu/libfl.a se hace referencia a la ubicación de la librería libfl.a
- -lfl enlace a la librería de Flex
- -o calculadora es la opción para decirle al compilador que el archivo ejecutable (a.out) llevara como nombre “calculadora”.

Se generara el ejecutable llamado calculadora.
NOTA: Para hacer uso del comando –lfl en Linux es necesario hacer referencia a la librería llamada libfl.a para ello escribimos la dirección donde se encuentra la librería (en este caso yo la encontré en esta dirección: /usr/lib/x86_64-linux-gnu/libfl.a) antes del comando –l

En la Figura 16 veremos la compilación del ejemplo calculadora en sistema Linux (Debian 7).

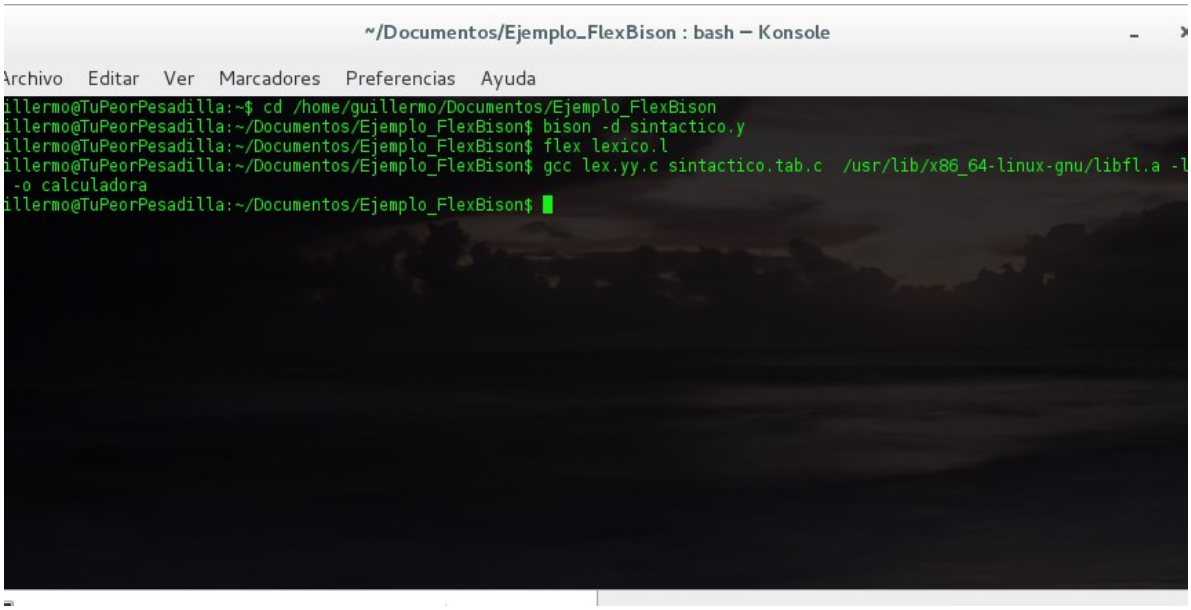



*Figura 16 Compilación del ejemplo calculadora en sistema Linux (Debian 7).*

Para ejecutar el ejemplo calculadora escribiremos en la terminal

./calculadora

En Figura 17 veremos la ejecución del ejemplo calculadora en sistema Linux (Debian 7).

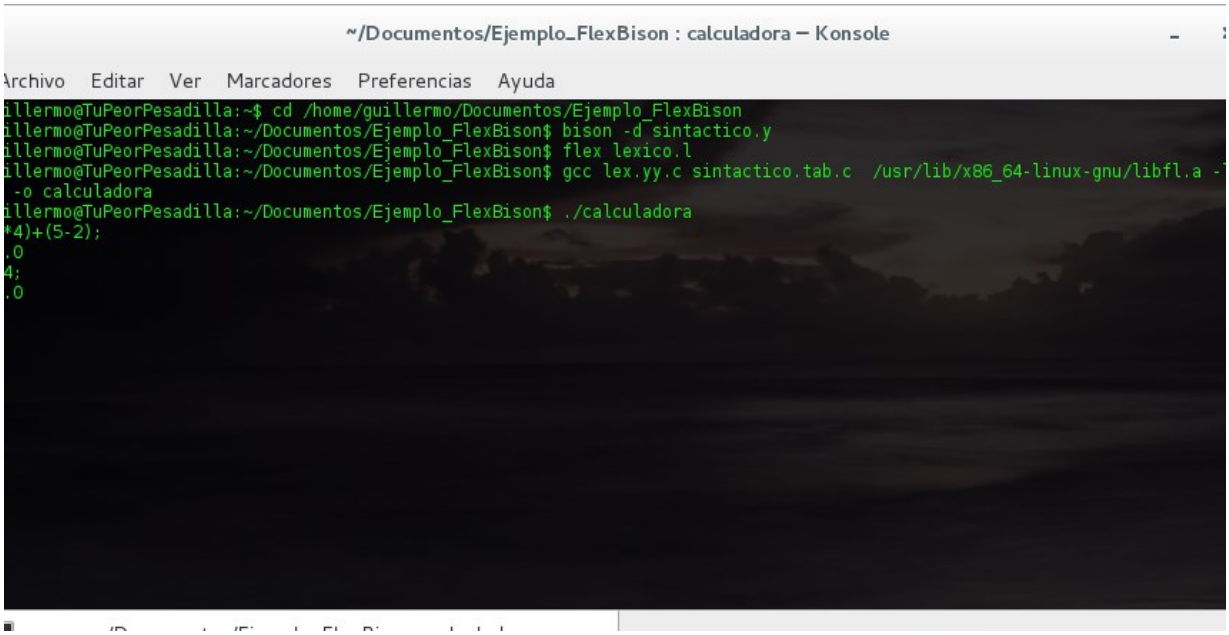



Figura 17 Ejecución del ejemplo calculadora en sistema Linux (Debian 7).

EJEMPLO 4

En el siguiente ejemplo veremos la funcionalidad de Flex junto con Bison trabajando juntos para realizar un analizador léxico sintáctico.

Código:
El siguiente código se guardara con la extensión .l es decir es el archivo Flex

```
%{
#include <stdio.h>
#include "y.tab.h"
%}

%option noyywrap
%option yylineno

ignorar " " | \t | \n
digito [0-9]
letra [a-zA-Z]
boleano "true"|"false"

%%

\t                      {}
\n                      {}
" "                     {}
"{"                     {printf("\nllave abierta\n"); return ('{');}
"}"                     {printf("\nllave cerrada\n"); return ('}');}
";"                     {printf("\npunto y coma\n"); return (';');}
","                     {printf("\ncoma\n"); return (',');}
")"                     {printf("\nparentecis cerrado\n"); return (')');}
"("                     {printf("\nparentecis abierto\n"); return ('(');}
```

```
"void"                          {printf("\ntipo vacio\n"); return VOID;}
"main"                          {printf("\ntipo principal\n"); return MAIN;}
"int"                           {printf("\ntipo entero\n"); return T_ENTERO;}
"float"                         {printf("\ntipo decimal\n"); return T_DECIMAL;}
"bool"                          {printf("\ntipo boleano\n"); return T_BOLEANO;}
"string"                        {printf("\ntipo cadena\n"); return T_CADENA;}
"+"                             {printf("\n+"); return SUMA;}
"-"                             {printf("\n-"); return RESTA;}
"*"                             {printf("\n*"); return MULTIPLICACION;}
"/"                             {printf("\n/"); return DIVISION;}
"="                             {printf("\n="); return ASIGNADOR;}
{digito}+                       {printf("\nentero\n"); return ENTERO;}
{digito}+"."{digito}            {printf("\ndeciaml\n"); return DECIMAL;}
{boleano}                       {printf("\nboleano\n"); return BOLEANO;}
"\""({digito}|{letra}|" ")*"\""  {printf("\ncadena\n"); return CADENA;}
{letra}({letra}|(digito))*      {printf("\nidentificador\n");          return
IDENTIFICADOR;}
.                               {printf("Error en linea: %d\n",yylineno);}

%%

yyerror(char * msg)
{
printf("%s\n",msg);
}

void main()
{
yyparse();
printf("\ncompilacion con exito\n");
}
```

Se guardara con el nombre lexico.l en la ubicación deseada en mi caso lo guardare en la siguiente dirección home/guillermo/tester/analizador

El siguiente código corresponde al analizador sintáctico:

```
%{
int yystopparser=0;
%}

%token VOID MAIN IDENTIFICADOR T_ENTERO T_DECIMAL T_BOLEANO T_CADENA ASIGNADOR ENTERO DECIMAL BOLEANO CADENA SUMA RESTA MULTIPLICACION DIVISION


%start funcion_principal

%%
funcion_principal:      principal
                        |VOID MAIN '(' ')' '{' setencias '}'
;
principal:              VOID MAIN '(' ')' '{' '}'
;
setencias:              setencias lineas
                        |lineas
;
lineas:                 declaracion
                        |asignar
;
declaracion:            tipo_dato declara ';'
;
```

```
declara:                IDENTIFICADOR
                        |IDENTIFICADOR asignarvalor
                        |',' IDENTIFICADOR
                        |',' IDENTIFICADOR asignarvalor
;
tipo_dato:              T_ENTERO
                        |T_DECIMAL
                        |T_BOLEANO
                        |T_CADENA
;
asignarvalor:           ASIGNADOR operacionasignacion
                        |ASIGNADOR valor
                        |ASIGNADOR IDENTIFICADOR
;
valor:                  ENTERO
                        |DECIMAL
                        |BOLEANO
                        |CADENA
;
asignar:                IDENTIFICADOR asignarvalor ';'
;
operacionasignacion:    aritmetico
;
aritmetico:             oprcomun
                        |oprcomun oprcomplemento
;
oprcomun:               valor tipoopr valor
                        |valor tipoopr IDENTIFICADOR
                        |IDENTIFICADOR tipoopr IDENTIFICADOR
                        |IDENTIFICADOR tipoopr valor
;
```

```
tipoopr:            SUMA
                    |RESTA
                    |MULTIPLICACION
                    |DIVISION
;
oprcomplemento:     oprcomplemento oprcom
                    |oprcom
;
oprcom:             tipoopr valor
                    |tipoopr IDENTIFICADOR
;
%%
```

Este código se guardara en la misma dirección donde guardamos el archivo lexico.l, lo llamaremos parser.y

Ejecución en sistema Linux[10]

En la siguiente pantalla 9 veremos la ejecución del ejemplo de analizador léxico sistema Linux.

[10] Para demostración de este ejemplo la versión de Linux usada es Ubuntu 14.04

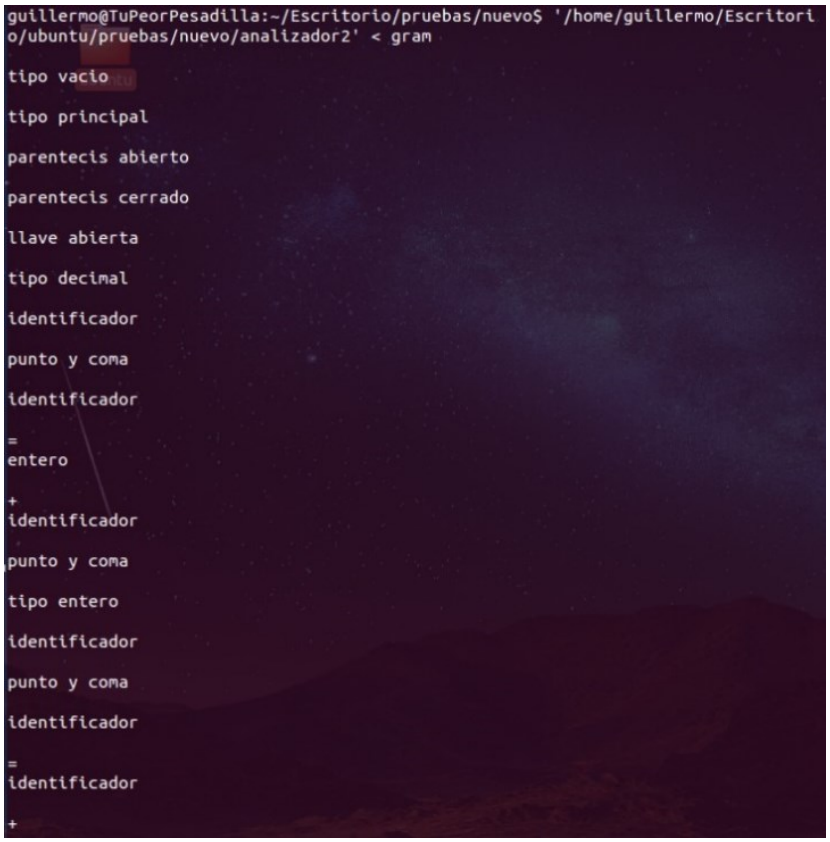



*Figura 18 Ejecución del ejemplo de analizador léxico sistema Linux.*

Ejecución en Windows[11]

En la siguiente pantalla 9 veremos la ejecución del ejemplo del analizador léxico en sistema Windows.

Para poder indicarle al ejecutable que se usara un archivo de texto como entrada es necesario hacer uso de la opción “<”.

Ejemplo:

“NombreEjecutable” < “NombreArchivo”

[11] Para demostración de este ejemplo la versión de Windows usada es Windows 7

El archivo de texto deberá estar guardado en la misma carpeta donde se encuentra el ejecutable si no se encuentra deberá indicar la ruta donde se encuentra dicho archivo

Ejemplo:
Si la ruta donde encuentra el archivo está en el escritorio del sistema se indicara como se muestra a continuación:

“NombreEjecutable” < “C:\user\Resendiz\Desktop\NombreArchivo”

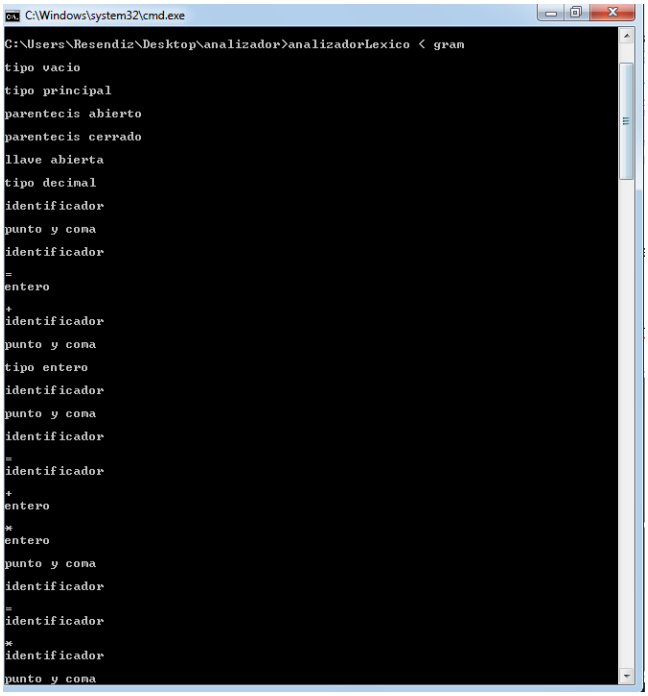



*Figura19 Ejecución del ejemplo del analizador léxico en sistema Windows.*

El archivo llamado gram es un archivo de formato txt el cual contiene el texto a analizar. En la Figura 20 se muestra el archivo gram usado para este ejemplo.

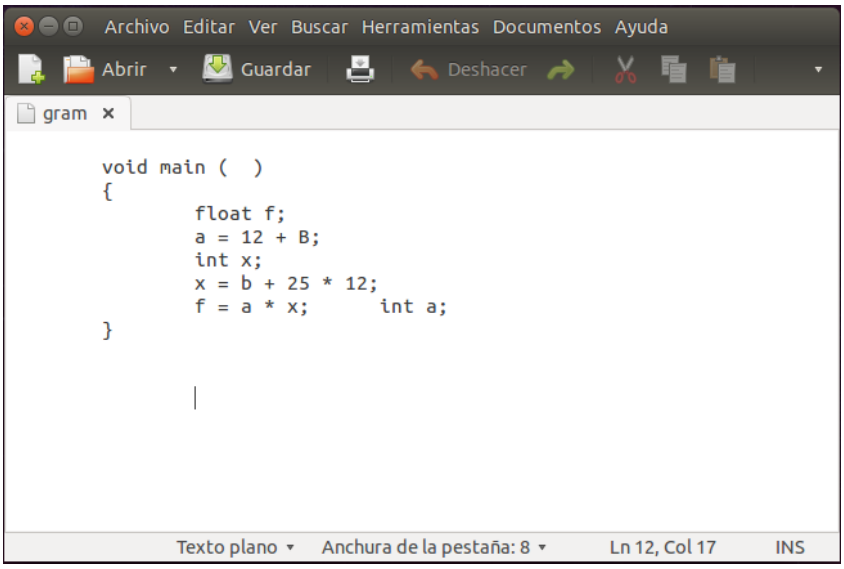



*Figura 20 Archivo "gram.txt"*

# CONCLUSIÓN

CONCLUSIÓN

Gracias a esta investigación se ha aprendido el uso de la herramienta Flex. Hay que destacar, todos los conocimientos adquiridos con la utilización de Flex así como de Bison. También esta experiencia ha servido para poner a prueba los conocimientos adquiridos durante toda la carrera. La investigación y demostración de uso de la herramienta Flex resultó satisfactoria exponiendo en ella los resultados esperados presentando con ejemplos para un mayor entendimiento.

La herramienta Flex es una herramienta y un auxiliar para generadores de analizadores léxicos. Su propósito no es más que ayudar a generar analizadores con la ayuda expresiones regulares.

La combinación de Bison y Flex permiten generar un analizador sintáctico con ayuda de un analizador léxico esto nos permite construir con excito las primeras dos fases de desarrollo de un compilador. ….

Los objetivos cumplidos:

- ✓ Explicar el funcionamiento de la herramienta Flex para el análisis léxico de lenguajes regulares y gramáticas regulares como parte fundamental de la creación de compiladores.
- ✓ Explicar que es un lenguaje regular.
- ✓ Explicar que es una gramática regular.
- ✓ Explicar el funcionamiento de FLEX como una herramienta analizadora de expresiones regulares.
- ✓ Mostrar ejemplos donde se utilizan la herramienta.

Los objetivos se lograron en base a esfuerzo y dedicación así como el compromiso fijado.

# ANEXO

## ANEXO INSTALACION DE FLEX Y BISON EN LINUX Y WINDOWS

Flex es una herramienta que genera analizadores léxicos a partir de un conjunto de expresiones regulares, en su página principal podremos encontrar más información acerca de esta herramienta http://flex.sourceforge.net/

Bison 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. En esta dirección web podremos encontrar más información acerca de esta herramienta http://www.gnu.org/software/bison/bison.html

### INSTALACIÓN DE FLEX EN LINUX

### INSTALACION DE FLEX MODO GRAFICO EN DEBIAN

En este caso se utilizará la distribución Debian, entonces para instalar Flex solo tenemos que entra al gestor de paquetes Synaptic (Synaptic es un gestor de paquetes gráfico para apt, el sistema de gestión de paquetes de Debian), ver Figura 21, luego buscamos Flex y los marcamos para ser instalados, luego solo aplicamos los cambios y se descargaran e instalaran automáticamente los paquetes.

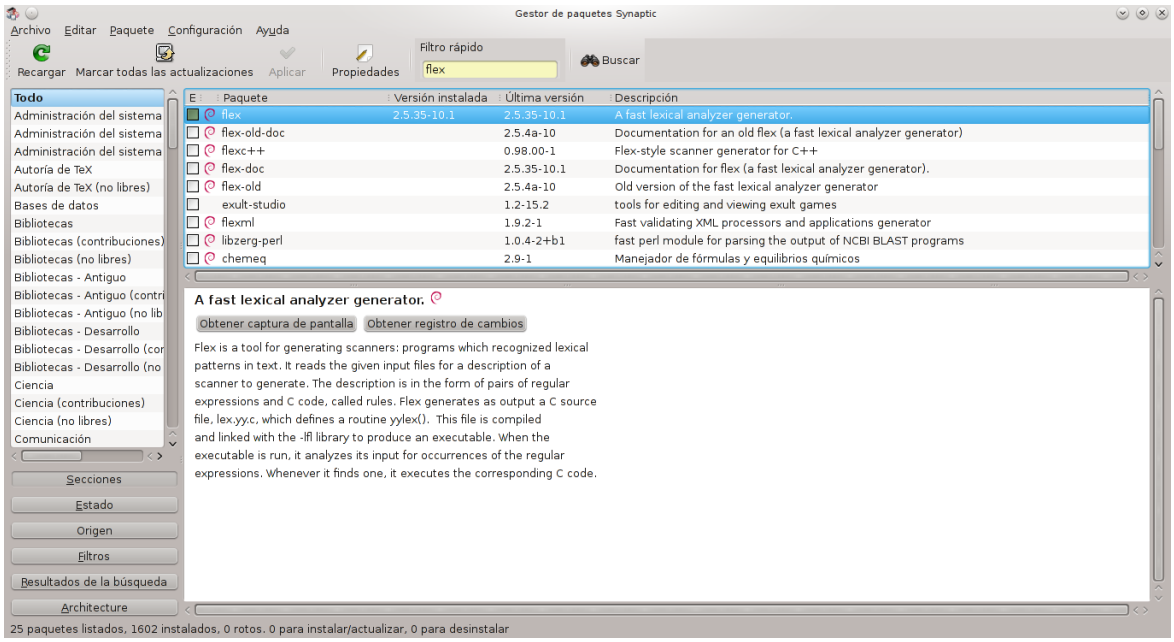



*Figura 21 Gestor de paquetes Synaptic*

## INSTALACION DE FLEX DESDE CONSOLA EN DEBIAN

Abrimos una terminal y escribimos:

sudo apt-get install flex

se escribe la contraseña de root y le damos Enter

## INSTALACIÓN DE FLEX EN WINDOWS 7

Hay dos maneras de instalar esta herramienta:

1. Descargando la lista de paquetes archivos zip.
2. Descargando un programa de instalación

Para este manual solo nos ocuparemos de la instalación por medio de un programa de instalación.

Primero tenemos que descargar en instalador de la página: http://gnuwin32.sourceforge.net/packages/flex.htm

Y elegimos la opción Setup, nos descargara un archivo .exe

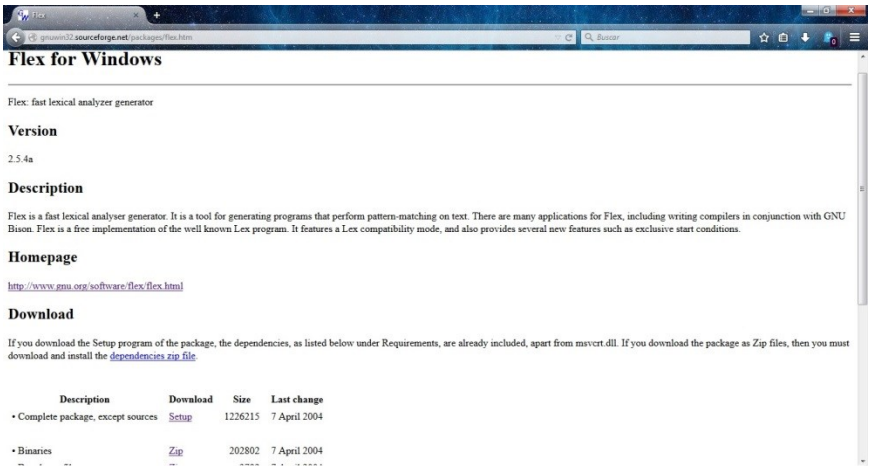



*Figura 22 Página web principal de Flex*

Ejecutamos el instalador descargado aceptamos los términos y seguimos las instrucciones que nos indiquen, ver Figura 23.

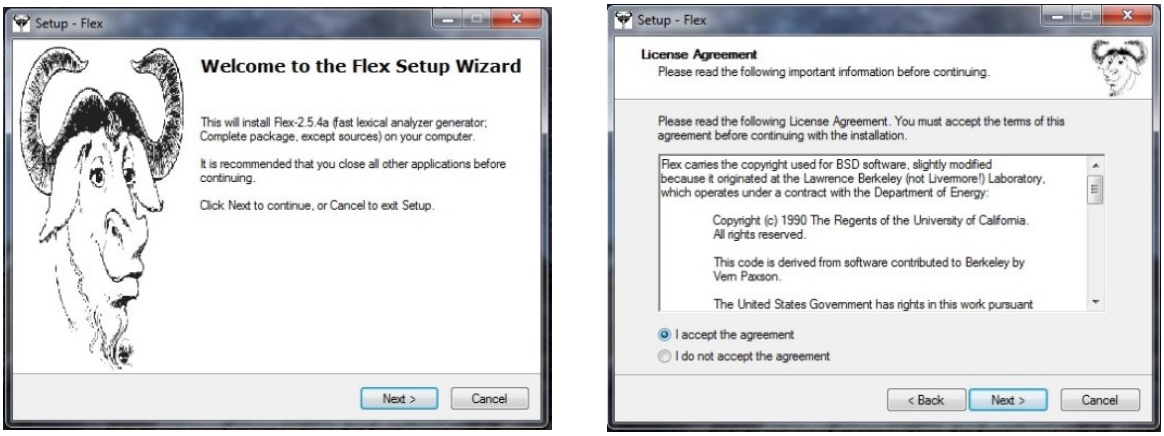



*Figura 23 Instalador de Flex para Windows*

Nos fijamos que la carpeta de instalación sea C:\GnuWin32 le damos Next y seleccionamos todos los componentes. Figura 24

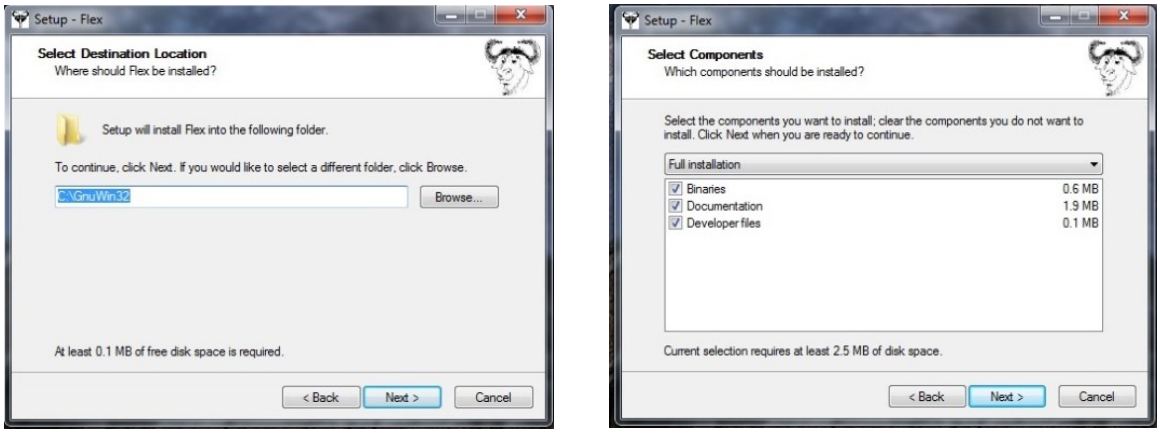



*Figura 24 Instalador de Flex para Windows*

Seleccionamos la carpeta (la dejamos tal como está) le damos Next y seleccionamos la opción de descargar fuentes (Download Source). Figura 25.

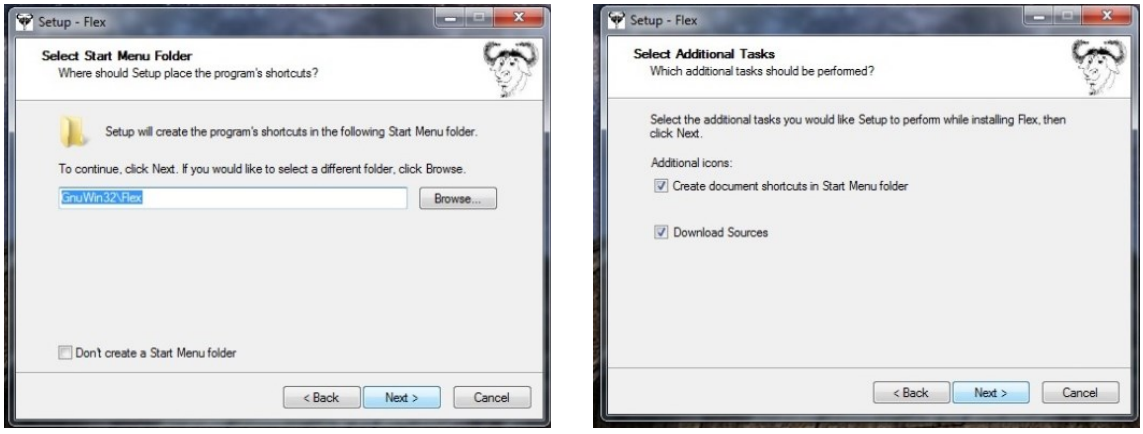



*Figura 25 Instalador de Flex para Windows*

Y por último le damos Instalar y esperamos a que el instalador termine.

INSTALACIÓN DE BISON EN LINUX

INSTALACIÓN MODO GRAFICO EN DEBIAN

En este caso se utilizara la distribución Debian, entonces para instalar Bison solo tenemos que entra al gestor de paquetes synaptic luego buscamos Bison y los marcamos para ser instalados, luego solo aplicamos los cambios y se descargaran e instalaran automáticamente los paquetes.

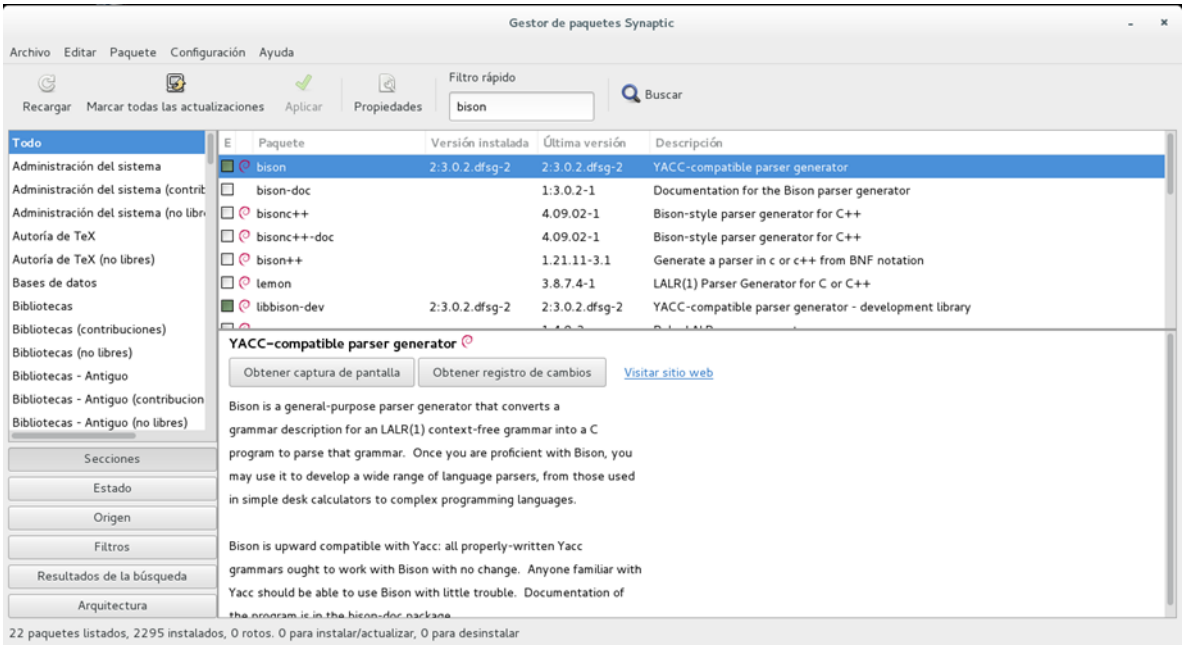



*Figura 26 Gestor de paquete Synaptic*

INSTALACIÓN POR CONSOLA EN DEBIAN

Abrimos una terminal y escribimos:

sudo apt-get install bison

se escribe la contraseña de root y le damos Enter

INSTALACIÓN DE BISON PARA WINDOWS

Para instalación de Bison es casi el mismo proceso que la instalación de Flex.

Primero tenemos que descargar en instalador de la página: http://gnuwin32.sourceforge.net/packages/bison.htm

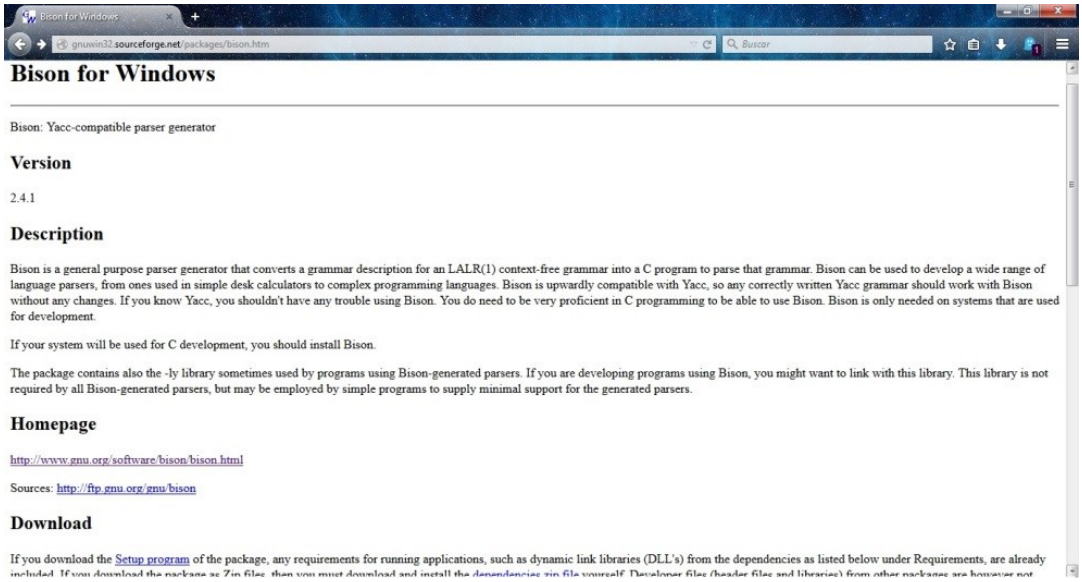



*Figura 27 Página web principal de Bison*

Y elegimos la opción Complete package, except sources (Setup), nos descargara un archivo .exe

Ejecutamos el instalador descargado aceptamos los términos y seguimos las instrucciones que nos indiquen:

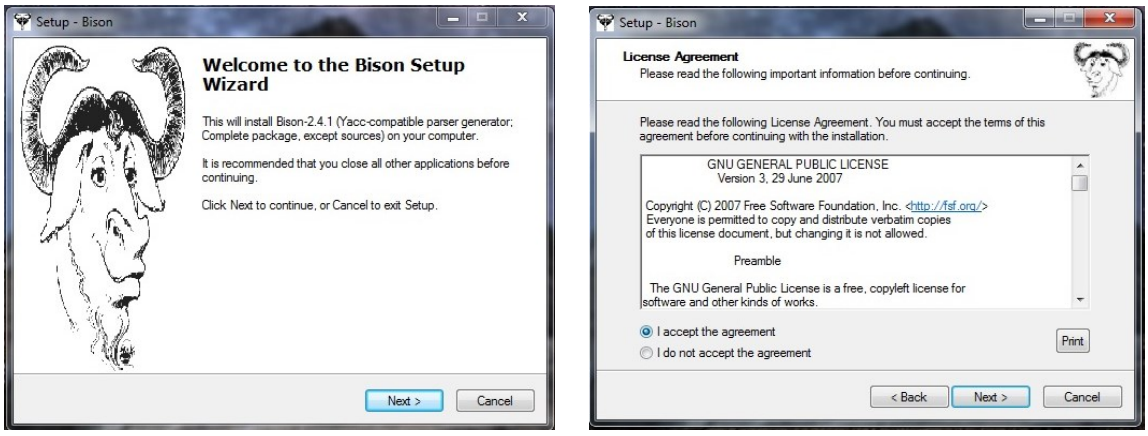



*Figura 28 Instalador de Bison para Windows*

Nos fijamos que la carpeta de instalación sea C:\GnuWin32 le damos Next y seleccionamos todos los componentes.

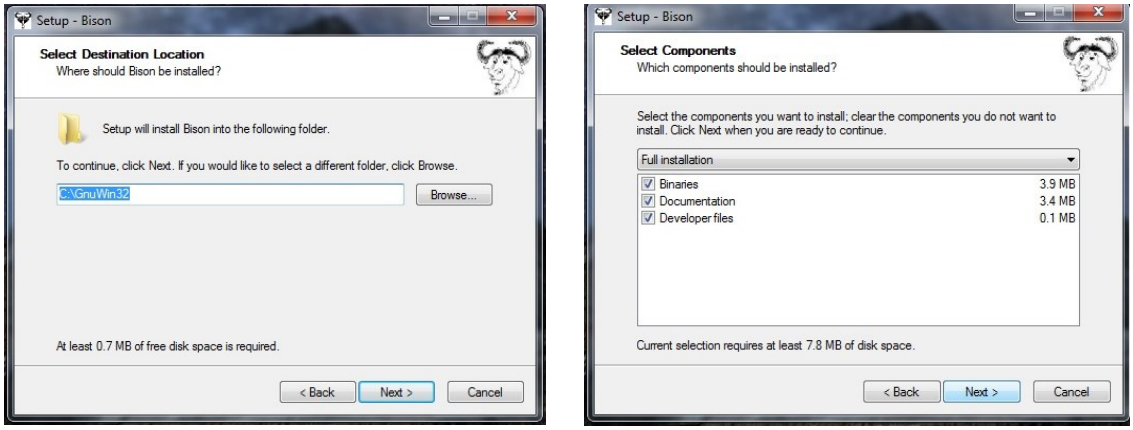



*Figura 29 Instalador de Bison para Windows*

Seleccionamos la carpeta (la dejamos tal como está) le damos Next y seleccionamos la opción de descargar fuentes (Download Source).

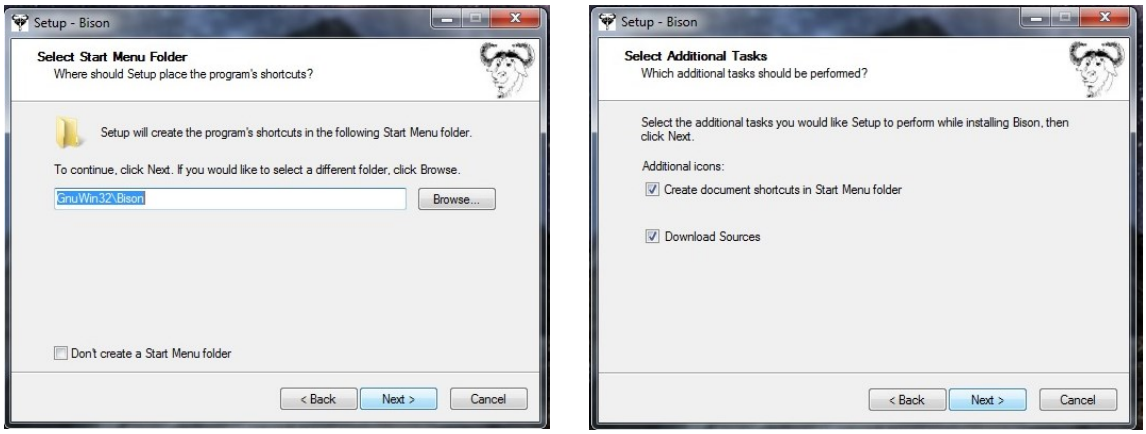



*Figura 30 Instalador de Bison para Windows*

Y por último le damos Instalar y esperamos a que el instalador termine.

INSTALACIÓN DE UN COMPILADOR

INSTALACIÓN DE GCC EN DEBÍAN

GCC es un compilador integrado del proyecto GNU para C, C++, Objective C y Fortran; es capaz de recibir un programa fuente en cualquiera de estos lenguajes y generar un programa ejecutable binario en el lenguaje de la máquina donde ha de correr. La sigla GCC significa "GNU Compiler Collection". Originalmente significaba "GNU C Compiler"; todavía se usa GCC para designar una compilación en C. G++ refiere a una compilación en C++.

INSTALACION DE GCC MODO GRAFICO EN DEBIAN

En este caso se utilizara la distribución Debian, entonces para instalar GCC solo tenemos que entra al gestor de paquetes synaptic luego buscamos GCC y los marcamos para ser instalados, luego solo aplicamos los cambios y se descargaran e instalaran automáticamente los paquetes.

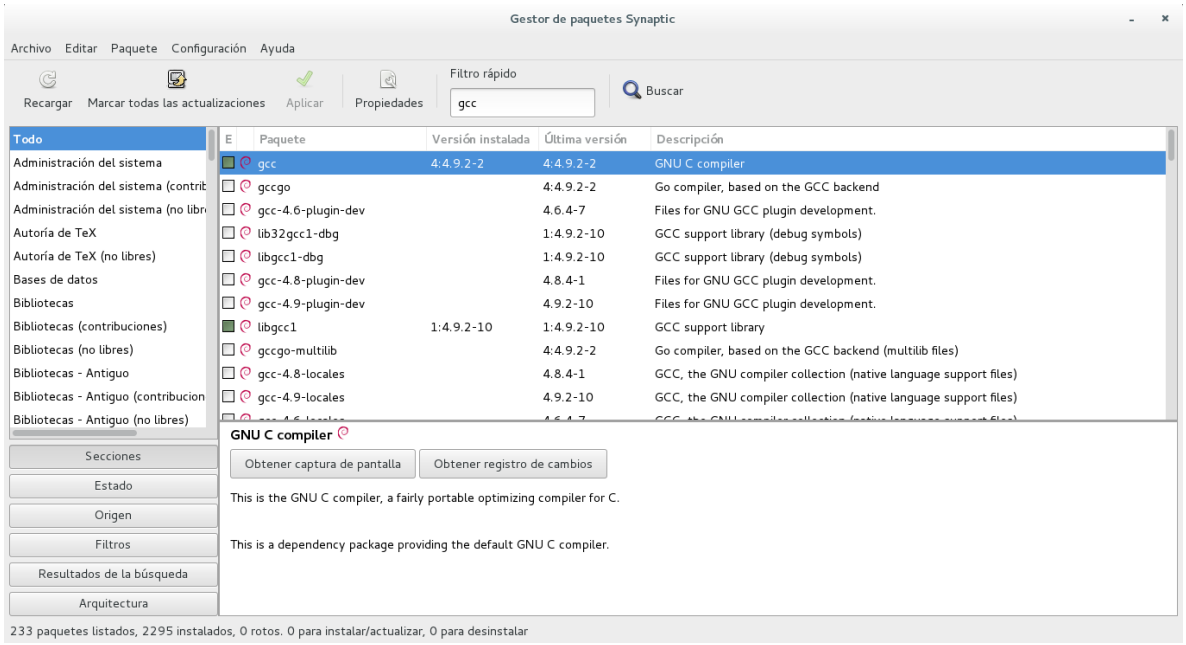



*Figura 31 Gestor de paquetes Synaptic*

## INSTALACIÓN DE GCC POR CONSOLA EN DEBIAN

Abrimos una terminal y escribimos:
sudo apt-get install gcc
se escribe la contraseña de root y le damos Enter

## INSTALACIÓN DE MINGW EN WINDOWS 7

MinGW es un ambiente de desarrollo para el sistema operativo Windows que provee herramientas similares a las existentes en GNU/Linux. Como su filosofía lo dice, es minimalista, es decir, sólo provee las herramientas necesarias y suficientes para crear aplicaciones nativas en Windows, entre las que se incluyen una versión de la suite GCC y GNU Binutils para Windows (ensamblador y enlazador). Además incluye opcionalmente un intérprete de línea de comandos llamado MSYS, con un conjunto selecto de herramientas Unix que facilitan el desarrollo de aplicaciones.

La manera recomendada de instalar MinGW es descargar el instalador automatizado disponible en el sitio del proyecto en SourceForge http://sourceforge.net/projects/mingw/files/ este instalador descarga los archivos necesarios, los descomprime y los copia a la ubicación especificada durante el proceso de instalación es recomendable que al programa instalador le indiquemos que el programa se instale en la ubicación de raíz, es decir lo ubiquemos de esta manera: C:\MinGW. La versión de mingw usada para este tutorial es 0.5-beta-20120426-1 y lo podemos encontrar en la siguiente dirección web: http://www.mediafire.com/download/v1u501njqhhbqr7/mingw-get-inst-20120426.exe

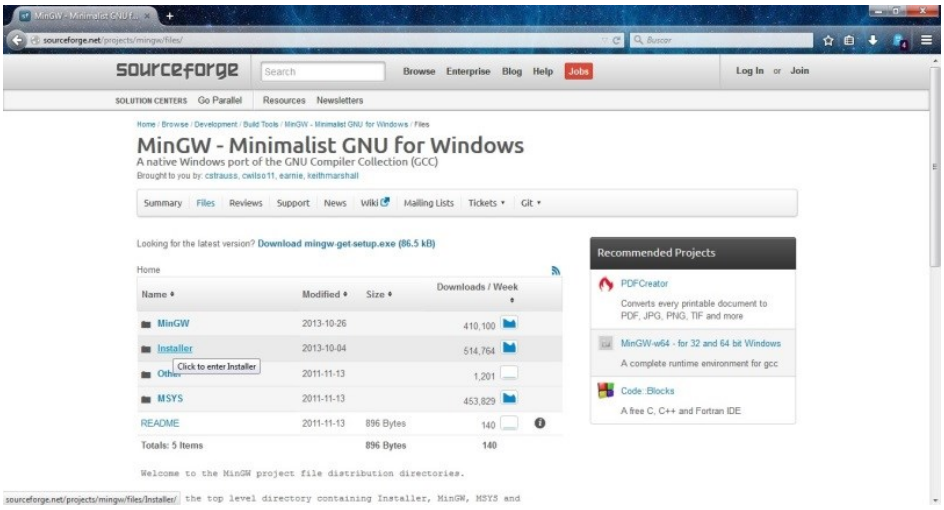



*Figura 32 Pagina web descarga del compilador MinGW para Windows*

Realmente es un procedimiento muy sencillo pues es casi automático, solamente es necesario aceptar la licencia GNU GPL del sistema, seleccionar la ubicación donde se instalará y los compiladores que se incluirán, además de la posibilidad de instalar MSYS Basic System.


*Figura 33 Instalador de MinGw  para Windows*

CONFIGURACIÓN DE FLEX, BISON Y MINGW EN WINDOWS 7

La parte más compleja sigue después de haber instalado Flex, Bison y MinGW, ya que para usar las herramientas y compiladores desde cualquier línea de comandos

(la línea de comandos convencional cmd.exe). Es necesario agregar el directorio donde se encuentran los ejecutables a la variable de entorno PATH. Para esto hay que ir al Panel de Control -> Sistema y Seguridad -> Sistema. En esta ubicación, en la columna a la izquierda aparecen varias opciones, una de ellas es Configuración avanzada del sistema. Le damos click (necesitaremos permisos de administrador) y nos aparece una pequeña ventana de nombre Propiedades del Sistema con muchas pestañas. Seleccionamos la pestaña Opciones avanzadas y hasta abajo aparece un botón que dice Variables de Entorno. Le damos clic y nos aparece otra ventana con dichas variables. Por último se copia la dirección de donde se encuentra nuestro archivos binarios en mi caso son estas direcciones: C:\GnuWin32\bin y C:\MinGW, se pega en nuestras variables de entorno, SIN BORRAR VARIABLES AGREGADAS ANTERIORMENTE, antes de pegar las direcciones debemos poner atención a que este escrito un " ; " (punto y coma) en la última variable de entorno, en caso de que no se encuentre lo debemos escribir, por ultimo escribimos " ; " después de a ver pegado nuestras direcciones, aceptamos y listo.

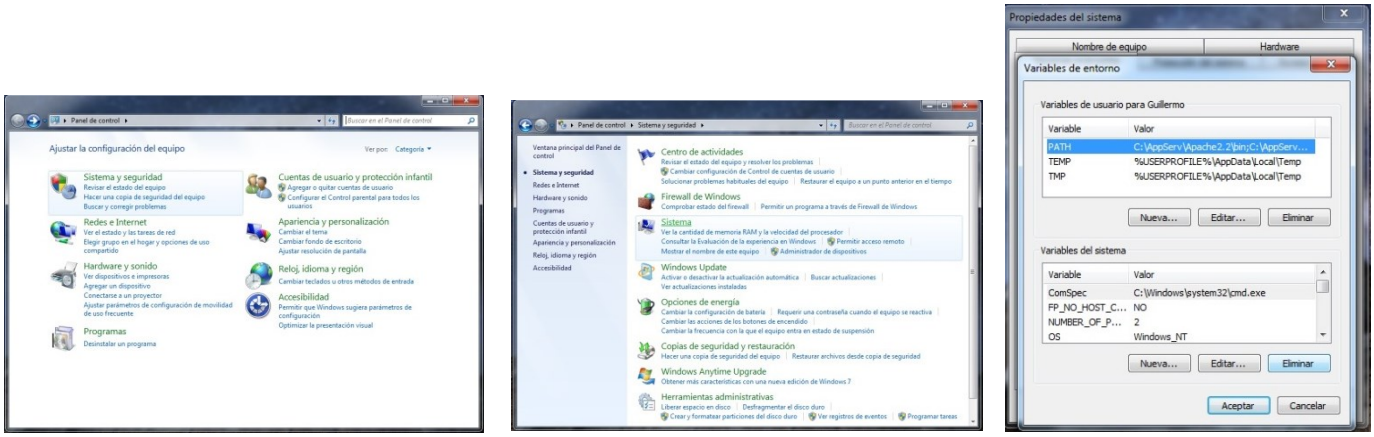



*Figura 34 Configuración de Flex, Bison y MinGW en Windows 7*

Por último para configurar bien Flex y Bison, debemos copiar la librería libfl.a ubicada en la carpeta lib de la carpeta GNUWIN32 en mi caso se encuentra en la dirección C:\GnuWin32\lib y la pegamos en la carpeta lib de MingGW en mi caso se encuentra en C:\MinGW\lib.

# ANEXO MANUAL COMPILADOR GCC

GCC es un compilador integrado del proyecto GNU para C, C++, Objective C y Fortran; es capaz de recibir un programa fuente en cualquiera de estos lenguajes y generar un programa ejecutable binario en el lenguaje de la máquina donde ha de correr.

La sigla GCC significa "GNU Compiler Collection". Originalmente significaba "GNU C Compiler"; todavía se usa GCC para designar una compilación en C. G++ refiere a una compilación en C++.

Sintaxis.

```
gcc [ opción | archivo ] ...
g++ [ opción | archivo ] ...
```

Las opciones van precedidas de un guion, como es habitual en UNIX, pero las opciones en sí pueden tener varias letras; no pueden agruparse varias opciones tras un mismo guion. Algunas opciones requieren después un nombre de archivo o directorio, otras no. Finalmente, pueden darse varios nombres de archivo a incluir en el proceso de compilación.

Ejemplos.

```
gcc hola.c
```

compila el programa en C hola.c, genera un archivo ejecutable a.out.

```
gcc -o hola hola.c
```

compila el programa en C hola.c, genera un archivo ejecutable hola.

```
g++ -o hola hola.cpp
```

compila el programa en C++ hola.c, genera un archivo ejecutable hola.

```
gcc -c hola.c
```

no genera el ejecutable, sino el código objeto, en el archivo hola.o. Si no s indica un nombre para el archivo objeto, usa el nombre del archivo en C y le cambia la extensión por .o.

gcc -c -o objeto.o hola.c

genera el código objeto indicando el nombre de archivo.

g++ -c hola.cpp

igual para un programa en C++.

g++ -o ~/bin/hola hola.cpp

genera el ejecutable hola en el subdirectorio bin del directorio propio del usuario.

g++ -L/lib -L/usr/lib hola.cpp

indica dos directorios donde han de buscarse bibliotecas. La opción -L debe repetirse para cada directorio de búsqueda de bibliotecas.

g++ -I/usr/include hola.cpp

indica un directorio para buscar archivos de encabezado (de extensión .h).

Sufijos en nombres de archivo.

Son habituales las siguientes extensiones o sufijos de los nombres de archivo:

| **.c** | **fuente en C** |
|---|---|
| **.C .cc .cpp .c++ .cp .cxx** | fuente en C++; se recomienda .cpp |
| **.m** | fuente en Objective-C |
| **.i** | C preprocesado |
| **.ii** | C++ preprocesdo |
| **.s** | fuente en lenguaje ensamblador |
| **.o** | código objeto |
| **.h** | archivo para preprocesador (encabezados), no suele figurar en la linea de comando de gcc |

*Tabla 4 Extensiones*

Opciones.

- c

realiza preprocesamiento y compilación, obteniento el archivo en código objeto; no

realiza el enlazado.

- E

realiza solamente el preprocesamiento, enviando el resultado a la salida estándar.

-o *archivo*

indica el nombre del archivo de salida, cualesquiera sean las etapas cumplidas.

-I*ruta*

especifica la ruta hacia el directorio donde se encuentran los archivos marcados para incluir en el programa fuente. No lleva espacio entre la I y la ruta, así: -I/usr/include

-L

especifica la ruta hacia el directorio donde se encuentran los archivos de biblioteca con el código objeto de las funciones referenciadas en el programa fuente. No lleva espacio entre la L y la ruta, así: -L/usr/lib

-Wall

muestra todos los mensajes de error y advertencia del compilador, incluso algunos cuestionables pero en definitiva fáciles de evitar escribiendo el código con cuidado.

-g

incluye en el ejecutable generado la información necesaria para poder rastrear los errores usando un depurador, tal como GDB (GNU Debugger).

-v

muestra los comandos ejecutados en cada etapa de compilación y la versión del compilador. Es un informe muy detallado.

Etapas de compilación.

El proceso de compilación involucra cuatro etapas sucesivas: preprocesamiento, compilación, ensamblado y enlazado. Para pasar de un programa fuente escrito por un humano a un archivo ejecutable es necesario realizar estas cuatro etapas en forma sucesiva. Los comandos gcc y g++ son capaces de realizar todo el proceso de una sola vez.

1. Preprocesado.

En esta etapa se interpretan las directivas al preprocesador. Entre otras cosas, las variables inicializadas con #define son sustituídas en el código por su valor en todos los lugares donde aparece su nombre.

Usaremos como ejemplo este sencillo programa de prueba, circulo.c:

```
/* Circulo.c: calcula el área de un círculo.
   Ejemplo para mostrar etapas de compilación.
*/
#define PI 3.1416
main()
{
  float area, radio;
  radio = 10;
  area = PI * (radio * radio);
  printf("Circulo.\n");
  printf("%s%f\n\n", "Area de circulo radio 10: ", area);
}
```

El preprocesado puede pedirse con cualquiera de los siguientes comandos; cpp alude específicamente al preprocesador.

```
  $ gcc -E circulo.c > circulo.pp
  $ cpp circulo.c > circulo.pp
```

Examinando circulo.pp

```
  $ more circulo.pp
```

puede verse que la variable PI ha sido sustituída por su valor, 3.1416, tal como había sido fijado en la sentencia #define.

2. Compilación.

La compilación transforma el código C en el lenguaje ensamblador propio del procesador de nuestra máquina.

```
$ gcc -S circulo.c
```

realiza las dos primeras etapas creando el archivo circulo.s; examinándolo con

```
$ more circulo.s
```

puede verse el programa en lenguaje ensamblador.

3. Ensamblado.

El ensamblado transforma el programa escrito en lenguaje ensamblador a código objeto, un archivo binario en lenguaje de máquina ejecutable por el procesador.

El ensamblador se denomina as:

```
$ as -o circulo.o circulo.s
```

crea el archivo en código objeto circulo.o a partir del archivo en lenguaje ensamblador circulo.s. No es frecuente realizar sólo el ensamblado; lo usual es realizar todas las etapas anteriores hasta obtener el código objeto así:

```
$ gcc -c circulo.c
```

donde se crea el archivo circulo.o a partir de circulo.c. Puede verificarse el tipo de archivo usando el comando

```
$ file circulo.o
circulo.o: ELF 32-bit LSB relocatable, Intel 80386, version 1, not stripped
```

En los programas extensos, donde se escriben muchos archivos fuente en código C, es muy frecuente usar gcc o g++ con la opción -c para compilar cada archivo fuente por separado, y luego enlazar todos los módulos objeto creados. Estas operaciones se automatizan colocándolas en un archivo llamado makefile, interpretable por el comando make, quien se ocupa de realizar las actualizaciones mínimas necesarias toda vez que se modifica alguna porción de código en cualquiera de los archivos fuente.

4. Enlazado

Las funciones de C/C++ incluídas en nuestro código, tal como printf() en el ejemplo, se encuentran ya compiladas y ensambladas en bibliotecas existentes en el sistema. Es preciso incorporar de algún modo el código binario de estas funciones a nuestro ejecutable. En esto consiste la etapa de enlace, donde se reúnen uno o más módulos en código objeto con el código existente en las bibliotecas.

El enlazador se denomina ld. El comando para enlazar

```
$ ld -o circulo circulo.o -lc
ld: warning: cannot find entry symbol _start; defaulting to 08048184
```

da este error por falta de referencias. Es necesario escribir algo como

```
$ ld -o circulo /usr/lib/gcc-lib/i386-linux/2.95.2/collect2 -m elf_i386 -dynamic-linker
/lib/ld-linux.so.2 -o circulo /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc-lib/i386-
linux/2.95.2/crtbegin.o -L/usr/lib/gcc-lib/i386-linux/2.95.2 circulo.o -lgcc -lc -lgcc
/usr/lib/gcc-lib/i386-linux/2.95.2/crtend.o /usr/lib/crtn.o
```

para obtener un ejecutable.

El uso directo del enlazador ld es muy poco frecuente. En su lugar suele proveerse a gcc los códigos objeto directamente:

```
$ gcc -o circulo circulo.o
```

crea el ejecutable circulo, que invocado por su nombre

```
$ ./circulo
Circulo.
Area de circulo radio 10: 314.160004
```

da el resultado mostrado.

Todo en un solo paso.

En programa con un único archivo fuente todo el proceso anterior puede hacerse en un solo paso:

```
$ gcc -o circulo circulo.c
```

No se crea el archivo circulo.o; el código objeto intermedio se crea y destruye sin verlo el operador, pero el programa ejecutable aparece allí y funciona.

Es instructivo usar la opción -v de gcc para obtener un informe detallado de todos los pasos de compilación:

```
$ gcc -v -o circulo circulo.c
```

Enlace dinámico y estático.

Existen dos modos de realizar el enlace:
- estático: los binarios de las funciones se incorporan al código binario de nuestro ejecutable.
- dinámico: el código de las funciones permanece en la biblioteca; nuestro ejecutable cargará en memoria la biblioteca y ejecutará la parte de código correspondiente en el momento de correr el programa.

El enlazado dinámico permite crear un ejecutable más chico, pero requiere disponible el acceso a las bibliotecas en el momento de correr el programa. El enlazado estático crea un programa autónomo, pero al precio de agrandar el tamaño del ejecutable binario.

Ejemplo de enlazado estático:

```
$ gcc -static -o circulo circulo.c
$ ls -l circulo
-rwxr-xr-x    1 victor   victor     237321 ago  4 11:24 circulo
```

Si no se especifica -static el enlazado es dinámico por defecto.

Ejemplo de enlazado dinámico:

```
$ gcc -o circulo circulo.c
$ ls -l circulo
-rwxr-xr-x    1 victor   victor       4828 ago  4 11:26 circulo
```

Notar la diferencia en tamaño del ejecutable compilado estática o dinámicamente.

Los valores pueden diferir en algo de los mostrados; dependen de la plataforma y la versión del compilador.

El comando ldd muestra las dependencias de bibliotecas compartidas que tiene un ejecutable:

```
$ gcc -o circulo circulo.c
$ ldd circulo
        libc.so.6 => /lib/libc.so.6 (0x40017000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$ gcc -static -o circulo circulo.c
$ ldd circulo
        statically linked (ELF)
```

La compilación estática no muestra ninguna dependencia de biblioteca.

Resumen.

Para producir un ejecutable con fuente de un solo archivo:

```
$ gcc -o circulo circulo.c
```

Para crear un módulo objeto, con el mismo nombre del fuente y extensión .o:

```
$ gcc -c circulo.c
```

Para enlazar un módulos objeto:

```
$ gcc -o circulo circulo.o
```

Para enlazar los módulos objeto verde.o, azul.o, rojo.o, ya compilados separadamente, en el archivo ejecutable colores:

```
$ gcc -o colores verde.o azul.o rojo.o
```

# REFERENCIAS


Libros

John R. Levine, Tony Mason & Doug Brown. (1995). Using Lex. En Lex & Yacc(27-47). EU: O'Reilly.

Alfred V. Aho, Monica S. Lam, Ravi Sethi & Jeffrey D. Ullman. (1990). Compiladores: principios, técnicas y herramientas. EU: Addison-Wesley.

Libros electrónicos

John R. Levine. (2009). Using Flex. En Flex y Bison(19-45). EU: O'Reilly.

Tom Niemann. (2015). LEX & YACC TUTORIAL. Noviembre, 2015, de epaperpress.com Sitio web: http://epaperpress.com/lexandyacc/download/LexAndYaccTutorial.pdf

Universidad Nacional de Colombia. (2015). Lenguajes regulares. Noviembre, 2015, de http://www.virtual.unal.edu.co Sitio web: http://www.virtual.unal.edu.co/cursos/ciencias/2001018/lecciones/PDFs/Cap1/Cap1s13.pdf

Alfaomega. (2008). GRAMATICAS REGULARES - EXPRESIONES REGULARES. Noviembre, 2015, de Alfaomega Sitio web: http://libroweb.alfaomega.com.mx/catalogo/matematicasparalacomputacion2/libreacceso/libreacceso/reflector/ovas_statics/cap9/Gramaticas%20regulares%20y%20Expresiones%20regulares.pdf

Héctor E. Medellín Anaya. (2015). Lenguajes Regulares. Noviembre, 2015, de Héctor E. Medellín Anaya Sitio web: galia.fc.uaslp.mx/~medellin/AcetTa/LenguajesRegulares.ppt

Páginas web.

Arteaga Caballero Jorge Julián, E. H. (Noviembre de 2009). *Blogger*. Obtenido de http://cursocompiladoresuaeh.blogspot.mx/2010/11/unidad-iv-herramientas-basicas-para.html

[Nicolás] Prof. Dr. Nicolás Luis Fernández García. (Septiembre 21, 2010). García, D. N. (21 de Septiembre de 2010). *Universidad de Córdoba*. Recuperado el 2015, de http://www.uco.es/users/ma1fegan/Comunes/manuales/pl/ANTLR/Introduccion-ANTLR.pdf

Compiladores, T. d. (Agosto de 2015). EcuRed. Obtenido de http://www.ecured.cu/index.php/Teor%C3%ADa_de_compiladores

David. (11 de Abril de 2011). *El Blog del Ing. Hobbit*. Obtenido de http://inghobbit.blogspot.mx/2011/04/ejemplo-flex-y-bison.html

L., R. (Noviembre de 2010). *Gramaticas Regulares*. Obtenido de https://robsitemas.wordpress.com/2010/10/27/gramaticas-regulares

Definiciona. (2015). *Definiciona*. Obtenido de Definiciona: http://definiciona.com/gramatica

Academia. (2015). *Academia*. Obtenido de Jerarquia de Chomsky y Fases de un compilador: http://www.academia.edu/8475322/Jerarquia_de_Chomsky_y_Fases_de_un_compilador

[Alan] Alan Hernandez. (2010). Notación Backus-Naur. Noviembre, 2015, de Alan Hernandez Sitio web: http://alan-lenpro.blogspot.mx/2010/10/notacion-backus-naur.html

Victor. (2009). Autómatas Finitos. Noviembre, 2015, de Victor Sitio web: http://automatas-finitos.blogspot.mx/

Otros Recursos:

http://unse-prog2.comxa.com/downloads/flex1.pdf

http://plan9.bell-labs.com/magic/man2html/1/lex

http://iie.fing.edu.uy/~vagonbar/gcc-make/gcc.htm

http://es.tldp.org/Manuales-LuCAS/FLEX/flex-es-2.5.pdf

http://flex.sourceforge.net/manual/

http://nereida.deioc.ull.es/~plgrado/javascriptexamples/node20.html

http://www.eis.uva.es/~fergay/III/enlaces/gcc.html

http://es.tldp.org/Manuales-LuCAS/FLEX/flex-es-2.5.html#SEC15

http://iie.fing.edu.uy/~vagonbar/gcc-make/gcc.htm

http://www.ecured.cu/index.php/Flex

https://es.wikipedia.org/wiki/Lenguaje_regular

http://webdiis.unizar.es/asignaturas/LGA/material_2004_2005/Intro_Flex_Bison.pdf