UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO

CENTRO UNIVERSITARIO UAEM TEXCOCO

“APLICACIÓN DE LAS HERRAMIENTAS JFLEX Y CUP, PARA EL ANÁLISIS LEXICOGRÁFICO Y SINTÁCTICO DE LENGUAJES FORMALES”

T E S I S

QUE PARA OBTENER EL TÍTULO DE

INGENIERO EN COMPUTACIÓN

PRESENTA

ALAN ISAAC ARANDA CORAZA

ASESORA

DR. EN ED. JOEL AYALA DE LA VEGA

REVISORES

M. EN I. S. IRENE AGUILAR JUÁREZ

DR. EN C. JAIR CERVANTES CANALES

TEXCOCO, ESTADO DE MÉXICO, NOVIEMBRE DE 2018.

Universidad Autónoma del Estado de México

Centro Universitario UAEM Texcoco

Texcoco, México a 20 de Abril del 2018.

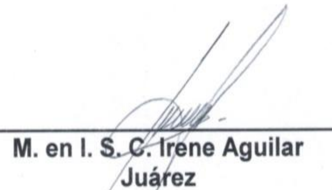
Asunto: Etapa de digitalización

M en C. Ed. Viridiana Banda Arzate.
Sub Directora Académica del
Centro Universitario UAEM Texcoco.
PRESENTE.

AT'N, M. en C. Leticia Arevalo Cedillo
Responsable del Departamento de Titulación

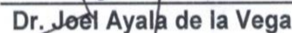
Con base en las revisiones efectuadas al trabajo escrito titulado "***Aplicación de las herramientas JFlex y CUP, para el análisis lexicográfico y sintáctico de lenguajes formales"*** que para obtener el título de Licenciado en Ingeniería en Computación presenta el sustentante Alan Isaac Aranda Coraza con número de cuenta 1024185, se concluye que cumple con los requisitos teórico-metodológicos por lo que se le otorga el voto aprobatorio para su sustentación, pudiendo **continuar con la etapa de digitalización** del trabajo escrito.

A T E N T A M E N T E



Dr. Jair Cervantes Canales

M. en I. S. C. Irene Aguilar Juárez



Dr. Joel Ayala de la Vega

Ccp Alan Isaac Aranda Coraza
Ccp- Dr. Joel Ayala de la Vega
Ccp. M. en C. Leticia Arévalo Cedillo.

Centro Universitario UAEM Texcoco
Av. Jardín Zumpango s/n. Fracc. El Tejocote
C.P. 56259 Texcoco, Estado de México.
Tels. (595) 9211216 - 9211247 - 9210368 - 9210493
e-mail: cutex.uaem@gmail.com.

CUTex

Contenido

INTRODUCCION 1

PLANTEAMIENTO DEL PROBLEMA 3

OBJETIVOS 4

JUSTIFICACIÓN 5

METODOLOGÍA.. 6

1. CAPITULO - LENGUAJES FORMALES. 7

1.1. LENGUAJES FORMALES. 8

1.1.1. DEFINICIÓN 9

1.2. LENGUAJES REGULARES. 10

1.2.1. EXPRESIONES REGULARES 11

2. CAPITULO – AUTOMATAS. 17

2.1. AUTOMATAS.. 18

2.1.1. AUTOMATAS FINITOS DETERMINISTAS (AFD) 19

3. CAPITULO – LENGUAJES LIBRES DE CONTEXTO Y GRAMATICAS REGULARES 37

3.1. LENGUAJES LIBRES DE CONTEXTO 38

3.2. DEFINICION DE GRAMATICA 39

3.2.1. JERARQUIA DE LAS GRAMATICAS 40

3.2.2. DERIVACION UTILIZANDO UNA GRAMATICA.. 44

3.2.3. ARBOLES DE DERIVACION... 48

3.2.4. NOTACIONES DE LOS LENGUAJES LIBRES DE CONTEXTO 54

3.2.5. AUTOMATAS TIPO PILA 57

4. CAPITULO – JFLEX Y CUP.. 66

4.1. ANALIZADOR LEXICOGRAFICO.. 67

4.1.1. FUNCION DEL ANALIZADOR LEXICO 67
4.2. ¿QUE ES JFLEX? 68
4.2.1. FUNCIONES Y ESTRUCTURA DE JFLEX 69
4.2.2. APLICANDO JFLEX 76
4.2.3. EJEMPLOS DE INTERES 87
4.3. ANALIZADOR SINTACTICO 89
4.4. ¿QUE ES CUP? 90
4.4.1. FUNCIONES Y ESTRUCTURA DE CUP 91
4.4.2. APLICANDO CUP 96
4.5. INTEGRACION JFLEX Y CUP 105
4.6. CASO DE USO JFLEX Y CUP 112
CONCLUSIONES 127
BIBLIOGRAFÍA 128

INTRODUCCION

Dentro de las Ciencias de la Computación se trata con varios modelos matemáticos abstractos que describen en diversos grados de precisión partes y tipos de computadores. Generalmente, para describir la funcionalidad o detalles más prácticos de las capacidades de un ordenador. Algunos campos que se abarcan dentro de las Ciencias de la Computación son: Arquitectura de Computadores, Teoría de Circuitos, Algoritmia y Estructura de Datos, Teoría de Autómatas, Sistemas Operativos, por hacer mención de algunos.

Fue a raíz de los grandes logros de las matemáticas al inicio del siglo XX que se obtuvo una gran evolución en el campo de las Ciencias de la Computación. De igual forma la máquina de Turing, la lógica matemática, la teoría de conjuntos, pudo cuestionar la dificultad de poder computar una función en base a alguna otra función establecida, dando hincapié a realizar cambios drásticos en maquetación y manejo de memoria, carga de programas y procesos, entre otros. De esta forma evolucionaron diversas áreas de operatividad en este campo, y en especial la introducción de los programas que ejercen las debidas instrucciones y operaciones que el computador ejecutar para obtener un resultado concreto. (Navarrete Sanchez, y otros, 2008)

De lo anteriormente mencionado, comienza a surgir una nueva área de los lenguajes formales “los lenguajes de programación de alto nivel”, basándose en una sintaxis y semántica propia, permitiendo ejercer instrucciones encaminadas a la algorítmica, facilitando su empleo y desempeño. Ante estos intentos, comenzaron a surgir y a generarse diversas gramáticas y expresiones que en conjunto dan lógica y solución a un problema planteado. (Moral)

Para el análisis y diseño de los lenguajes formales se requirió el manejo de metalenguajes. Dentro del análisis léxico se utilizan las expresiones regulares y en el análisis sintáctico se manejan los lenguajes libres de contexto. Para poder facilitar el uso de estos metalenguajes surge la necesidad de manejar diversas herramientas

como lo son los autómatas finitos y autómatas tipo pila. (E. Hopcroft, Motwani, & D. Ullman, 2007)

Al hacerse uso de los diversos lenguajes de programación existentes de alto nivel, existe una versatilidad de poder conjuntar instrucciones más complejas de manera más fluida, permitiendo ejercer las soluciones requeridas a los problemas establecidos de manera más efectiva. De esta forma, se hace ver la necesidad de la generación de analizadores sintácticos y lexicográficos para estos campos tan amplios que necesitan una coherencia y orden al ser ejecutadas. En el campo de la computación existen diversos analizadores en varios lenguajes, pero al ser un campo en constante cambio y actualización, siempre es necesario hacer uso de lo que se tiene al día, sin despreciar algunos de los lenguajes más “antiguos”, por su usabilidad, potencia y congruencia que han marcado a lo largo de la línea temporal.

El presente trabajo pretende dar como resultado, una efímera explicación e introducción a los autómatas y los lenguajes formales para demostrar el uso de las herramientas JFLEX y CUP, como analizador lexicográfico y sintáctico, que facilitan el manejo y creación de compiladores, siguiendo las reglas de los lenguajes formales y sus gramáticas, evitando la tediosa tarea de generar compiladores a mano y haciendo uso de las tecnologías recurrentes que han ido surgiendo para facilitar tareas tan laboriosas como la que se ha venido mencionando. JFLex y CUP, son herramientas que reaccionan a una entrada de datos con una estructura y un lenguaje predeterminado, el cual se le da a conocer previamente para poder trabajar con los datos de entrada.

JFLex, es un analizador lexicográfico, desarrollado en Java, el cual está basado en JLex y pensado a trabajar en conjunto con CUP, si así se desea. El cual trabaja de manera independiente para poder ser integrado con Java. (Urquina Fuentes, 2008)

CUP por otro lado, además de ser una herramienta también independiente para ser integrada con Java y JFLex, es un analizador sintáctico LALR (Look – Ahead Left to Right parser) el cual recibe de entrada un archivo con la estructura de la gramática y su salida es un archivo con los datos necesarios a ser usados.

PLANTEAMIENTO DEL PROBLEMA

El desarrollo de compiladores “a mano”, recae en la realización de muchas tareas repetitivas. Cuando se hizo patente esta necesidad de automatización, ya fuera en ensamblador o utilizando lenguajes intermedios, aparecieron las primeras herramientas de ayuda a la construcción de compiladores. Lo que hacen estas herramientas es generar código en un lenguaje de programación. Estas herramientas pueden hacer muchas de las tareas que realizan los compiladores, tales como la búsqueda de patrones, el análisis léxico, así como el análisis sintáctico y semántico.

Aun dentro de la automatización, se busca la eficiencia en los resultados, la facilidad de codificar y el menor costo posible en tiempos. Existen herramientas programadas en diversos lenguajes de programación que se prestan para la realización de compiladores, cada uno con sus características; reconociendo el gran potencial de los lenguajes de bajo nivel, pero de igual forma retomando su manera tediosa y extensa de codificación, es necesario el uso de lenguajes de alto nivel que permiten apegarse más a un lenguaje “natural”, y aun ante estos lenguajes tan potentes y de practica usabilidad, comienza a surgir la necesidad de implementar herramientas que eviten la ostentosa codificación de funciones abruptas, de ello es necesario la implementación de estas herramientas analizadoras.

JFLex y CUP, son dos herramientas que generan 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, dando una opción a la solvencia necesitada.

OBJETIVOS

General

- Explicar el funcionamiento de dos herramientas para el análisis léxico y sintáctico de lenguajes formales, expresiones regulares y sus gramáticas libres de contexto como parte fundamental de la creación de compiladores.

Particulares

- Explicar que es una expresión regular.
- Explicar que es una gramática libre de contexto.
- Explicar que son los lenguajes formales.
- Explicar el funcionamiento de JFLex como herramienta para el análisis lexicográfico.
- Explicar el funcionamiento de CUP como herramienta para el análisis sintáctico.
- Mostrar ejemplos donde se utilizan las dos herramientas.

JUSTIFICACIÓN

Siendo parte de la Teoría de Autómatas y una herramienta fundamental para la generación de compiladores, los lenguajes formales propician una gran diferencia entre los programas cargados a la memoria principal en los ordenadores en la década de los 40's, al nuevo surgimiento de los lenguajes de programación de alto nivel. Permitiendo reglas léxicas, sintácticas y semánticas rígidas, concretas y bien definidas. De esta manera, y ante lo abstracto o complejo de los lenguajes formales, surge la necesidad de hacer uso de diversas herramientas que permitan implementar las gramáticas que pueden llegar a generarse, evitando la manera tediosa de generación de código de programación, tomando en cuenta el apoyo que se tiene ante los ordenadores por la cantidad de información e instrucciones que pueden llegar a manejar en instantes.

De esta forma, se pretende dar a conocer el uso y funcionamiento de algunas de las herramientas existentes para el manejo lexicográfico y sintáctico de las gramáticas que se generan dentro del análisis y manejo de los lenguajes formales. Así mismo, la interacción entre ambas herramientas bajo el ambiente que se esté trabajando. Las herramientas a utilizar son: JFLex como analizador lexicográfico y CUP como analizador sintáctico, basadas en uno de los lenguajes de programación más poderosos, y competitivos de alto nivel que se utiliza hoy en día tanto académica y laboralmente hablando: JAVA.

A lo largo de esta investigación se pretende demostrar el funcionamiento, usabilidad y facilidad de aplicación de las herramientas JFLex y CUP, ante lo descrito con anterioridad. Así mismo y citando, la conexión entre ambas. Permitiendo abrir un nuevo esquema e "innovación" de uso de herramientas de trabajo, que faciliten el aprendizaje de los lenguajes formales y sus gramáticas, dentro del CU UAEM Texcoco y para la comunidad Académica en general.

METODOLOGÍA

Para la presente investigación se llevará a cabo la siguiente metodología:

- Análisis de los lenguajes formales.
- Análisis de las gramáticas y expresiones regulares.
- Análisis y competencia de JFLex como analizador lexicográfico ante las opciones existentes en el mercado competitivo.
- Análisis y competencia de CUP como analizador sintáctico ante las opciones existentes en el mercado competitivo.
- Interacción y conexión entre JFLex y CUP.
- Instalación y pruebas de las dos herramientas.
- Implementación de JFLex y CUP a un problema aplicativo.
- Conclusiones del análisis sobre JFLex y CUP como análisis léxico y sintáctico, dentro de los compiladores.
- Revisión bibliográfica.
- Programación de ejemplos de interés.
- Impresión de la Investigación.

Para el desarrollo de las aplicaciones se usará:

- Lenguaje Orientado a Objetos (Java).
- Plataforma “Netbeans IDE”.
- Plataforma “Sublime Text”.
- Java SE Development Kit 7.

El equipo mínimo para que funcione la aplicación es:

- Sistema Operativo “Windows" o “Ubuntu”.
- Memoria RAM mínima de 512 MB.
- Procesador Intel Pentium o Atom.

1. CAPÍTULO - LENGUAJES FORMALES.

1.1. LENGUAJES FORMALES.

Existen dos tipos básicos y reconocidos de lenguajes: los lenguajes naturales y los lenguajes formales. El origen y desarrollo de los primeros, como pueden ser el castellano, el inglés o el francés, es natural, es decir, sin el control de ninguna teoría. Las teorías de lenguajes naturales y las gramáticas, fueron establecidas a priori, esto es, después de que el lenguaje ya había madurado. Por otro lado, los lenguajes formales como las matemáticas y la lógica, fueron desarrollados generalmente a través del establecimiento de una teoría, la cual le da las bases para dichos lenguajes.

El desarrollo de los computadores en la década de los 40's, tuve un gran auge en el campo de la computación, ya que se tuvo una nueva maquetación de memoria al poder cargar los programas a la memoria principal, lo cual permitía interpretar el conjunto de instrucciones de manera más rápida y eficiente. (Moral). Posteriormente surgió la introducción de lenguajes de programación de alto nivel, propiciando reglas sintácticas y semánticas más rígidas, concretas y bien definidas, apegándose aún más al *lenguaje natural,* pero con la distinción de tener un control más riguroso, dando pasó a la construcción de gramáticas y reglas de lenguaje.

La Teoría de Gramáticas y Lenguajes Formales como una herramienta matemática que permite dar un rigor más amplio al diseño de lenguajes de programación, además de desarrollar las bases necesarias para la construcción de Autómatas, tiene un origen fuera del basto campo de las ciencias de la computación: La Lingüística. (Cueva Lovelle, 2001). El campo de las ciencias de la computación, comenzó a adquirir este *concepto* de gramáticas y lenguajes formales, tras diversas publicaciones de *Chomsky*, un célebre lingüista y matemático que permitió definir la sintaxis y semántica del *lenguaje natural*, permitiendo que surgieran uno de los primeros lenguajes de programación de alto nivel que implementaron gramáticas libres de contexto: ALGOL 60. (Cueva Lovelle, 2001). Permitiendo un diseño más riguroso de algoritmos de traducción y compiladores. (Quiroga Rojas, 2008)

La teoría de los lenguajes formales sin la intención de su creador resultó tener una relación muy fuerte y sorprendente con la teoría de autómatas y la computabilidad.

Donde se hizo notar que los lenguajes propuestos por Chomsky tienen una equivalencia entre las máquinas abstractas, en donde a partir de restricciones de las gramáticas, le corresponde un tipo de máquina abstracta. (Ibarra Florencio, 2011)

1.1.1. DEFINICIÓN

En matemáticas, lógica, y las ciencias computacionales, un **lenguaje formal** es un conjunto de palabras (cadenas de caracteres) de longitud finita formadas a partir de un alfabeto (conjunto de caracteres) finito. (Quiroga Rojas, 2008). Informalmente, el termino *lenguaje formal*, es utilizado para referirse a la formalización de un lenguaje, al cual, le precede una teoría, previamente establecida, que le permite mantener una sintaxis y semántica *bien formada*. De esta forma, se logra la formalización de un lenguaje, permitiendo tomar bases en teorías que han sido establecidas con anterioridad, permitiendo proporcionar una consistencia más elocuente a las "oraciones" que llegan a formarse. (Polanco Fernandez, 2000)

Definición 1.1 (Alfabeto). Al igual que todo lenguaje, su base es un **alfabeto**, el cual se define como un conjunto finito y no vacío de símbolos, es denotado por el símbolo: ∑. A continuación, se muestran algunos ejemplos de alfabetos:

$$\sum\nolimits_1 = \{0, 1, \dots\} \qquad \sum\nolimits_2 = \{a, b, c, \dots\}$$

Definición 1.2 (Cadena). Una **cadena** o **palabra** (ω), es una seria arbitraria de símbolos, unidos por concatenación, por ejemplo: *aaabbbccc*, es una cadena, al no tener un tamaño u orden especifico puede definirse simplemente que ω es una cadena, de igual forma existe la cadena vacía, representada con el símbolo **ε.**

Definición 1.3 (Lenguaje). Un **lenguaje** (L), es un conjunto finito de palabras o cadenas. Los siguientes, son ejemplos de lenguajes.

$$L_1 = \{\varepsilon, ab, aabb, aaabbb\} \quad L_2 = \{001, 011, 111\}$$

En base a las definiciones anteriormente mencionadas, se sigue que existe un espacio o universo U infinito de lenguajes. No todos los lenguajes que se encuentran

es este universo, son de interés para la teoría de los lenguajes formales, cuyo objetivo es investigar si existe un orden dentro de ese universo, y así poder estudiar las diversas propiedades con las cuales se puede calificar como *interesantes* a los lenguajes. (Baliri, 2014)

Una de las maneras de determinar si un lenguaje es de interés o no, es observando si se sigue alguna pauta regular en la construcción de las cadenas. De esta forma, se puede determinar que el lenguaje L_3 es razonable de estudio, pero el lenguaje L_4 no lo es.

$$L_3 = \{abc, aabbcc, aaabbbccc, \ldots\} \quad L_4 = \{a, cab, bdac, \ldots\}$$

Un segundo punto para determinar si un lenguaje es de *interés*, es representado por la disposición de recursos que podrían determinar si una cadena dada pertenece o no a un lenguaje determinado. Permitiendo este par de connotaciones definir los dos objetivos principales de las dos grandes disciplinas matemáticas que dan orden y sentido al universo de los lenguajes formales: la Teoría de los Lenguajes Formales y la Teoría de la Complejidad Computacional. (Baliri, 2014)

Definición 1.4 (Teoría de lenguajes formales). La **Teoría de los Lenguajes Formales**, estudia los lenguajes presentando atención únicamente a sus propiedades estructurales, definiendo clases de complejidad estructural y estableciendo relaciones entre las diferentes clases.

Definición 1.5 (Teoría de la complejidad computacional). La **Teoría de la Complejidad Computacional** estudia los lenguajes prestando atención a los recursos que utilizaría un dispositivo mecánico para completar un procedimiento de decisión, definiendo así diferentes clases de complejidad computacional y las relaciones que existen entre ellas.

1.2. LENGUAJES REGULARES.

El estudio de los lenguajes formales, es bastante amplio y complejo, por esta razón la gran mayoría de las investigaciones, se centra más al tema de los lenguajes regulares. Este tipo de lenguajes obtiene este nombre por la *regularidad* o por la

repetición de un mismo componente, es decir, se tiene una precepción intuitiva del lenguaje. (Quiroga Rojas, 2008)

Los lenguajes regulares, contienen una estructura más simple dentro de la *Jerarquía de Chomsky*, de la cual se hablará más adelante; una de las características más relevantes de los lenguajes regulares es la dependencia lineal que poseen sus cadenas, es decir, la presencia de un símbolo u otro en una posición determinada de la cadena, depende exclusivamente del símbolo que lo precede inmediatamente.

Un ejemplo de un lenguaje regular, se logra apreciar en el Lenguaje L_1.

$$L_1 = \{ab, abab, ababab, \ldots\}$$

En este ejemplo, logra apreciarse la simple repetición de la cadena *ab*, hasta un número infinito o finito de veces, de aquí es que se define la *regularidad* de este tipo de lenguajes. De lo demostrado anteriormente, nace un punto de vista práctico, donde los *lenguajes regulares*, son utilizados para especificar la construcción de analizadores léxicos (programas dedicados al análisis de texto y obtener lexemas o unidades léxicas que existen dentro del mismo texto). (Dean, 2001)

Definición 1.6 (Lenguaje Regular). Un lenguaje L es regular, si y solo si, se cumple al menos una de las siguientes condiciones.

- L es finito.
- L es la unión o la concatenación de otros lenguajes regulares R_1 y R_2,
- L = R_1 U R_2 o L = $R_1 R_2$ respectivamente.
- L es la cerradura de Kleene de algún lenguaje regular, L = R*.

Esta definición, permite la creación de expresiones, basadas en la notación de conjuntos que son representados en los lenguajes regulares.

1.2.1. EXPRESIONES REGULARES

Las expresiones regulares, surgen como una forma de expresar los *lenguajes regulares*, dicho en otras palabras, son un metalenguaje. Este tipo de expresiones no son propiamente de los *lenguajes regulares*, de igual forma sirven para

representar otro tipo de lenguajes y son una forma de entrada de datos y reconocimiento por parte de los autómatas finitos. (Cueva Lovelle, 2001)

Como se mencionó en la sección anterior, un *lenguaje regular*, puede ser representado por la notación de conjuntos, pero es mejor el manejo de una notación en la que al representar un lenguaje sea con simple texto (cadenas de caracteres). De esta forma el representar un *lenguaje regular* seria simplemente mediante palabras de un lenguaje (Ibarra Florencio, 2011). Una de las grandes ventajas con las que cuentan las *Expresiones Regulares* y que un autómata no puede llegar a ofrecer, recae en la forma declarativa que poseen para expresar las cadenas que desean ser aceptadas; una ventaja adicional es que se puede convertir en una máquina (autómata finito), el cual puede automáticamente decidir si una cadena (palabra) pertenece o no al lenguaje denotado por la *expresión regular* (Vilca Huayta, 2008). Por lo tanto, las *expresiones regulares*, son una gran forma de entrada de datos en muchos sistemas de procesamiento de este tipo de información. Algunos ejemplos de sistemas de este tipo, son:

- Comandos de búsqueda tales como el comando *grep* de UNIX o comandos equivalentes para localizar cadenas en los exploradores web o en los sistemas de formateo de texto. Estos sistemas emplean una notación de tipo *expresión regular* para describir los patrones que el usuario desea localizar en un archivo.
- Generadores de analizadores léxicos, como lo son JLex o JFLex. Tomando en cuenta que 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 =.

Definición 1.7 (Expresiones Regulares). Una expresión regular es una fórmula *r* cuya denotación es un lenguaje L (r) definido sobre un alfabeto ∑. Hay dos tipos de

expresiones regulares: las *expresiones regulares atómicas* y las *expresiones regulares compuestas*. Cada tipo posee tres subtipos.

La sintaxis y la denotación de las *expresiones regulares atómicas* son:

- Sea α, tal que a ∈ ∑, es una *expresión regular,* entonces L (α) = {a}.
- La cadena vacía ε es una *expresión regular*, entonces L (ε) = {ε}.
- El conjunto vacío Φ es una *expresión regular*, entonces L (Φ) = {}.

La sintaxis y la denotación de las *expresiones regulares compuestas* son:

- ($r_1 r_2$) es una expresión regular, entonces L ($r_1 r_2$) = L (r_1) · L (r_2).
- ($r_1 + r_2$) es una expresión regular, entonces L ($r_1 + r_2$) = L (r_1) U L (r_2).
- (r)* es una *expresión regular*, entonces L ((r)*) = (L (r)) *

1.2.1.1. OPERACIONES CON LAS EXPRESIONES REGULARES

Como se demostró, la definición de *expresión regular*, explota la propiedad única de los *lenguajes regulares*, según la cual, cualquier *lenguaje regular* puede expresarse como la concatenación, unión o clausura de dos o más *lenguajes regulares* por la propiedad basada en la notación de conjuntos. (Baliri, 2014)

1.2.1.1.1. UNIÓN

Definición 1.8 (Unión de lenguajes). Si L_1 y L_2 son lenguajes, la unión de L_1 y L_2 es $L_1 \cup L_2 = \{x \mid x \in L_1 \vee x \in L_2\}$. Donde cada lenguaje está definido sobre el mismo alfabeto. (Cueva Lovelle, 2001)

Dada la **Definición 1.8** (Unión de lenguajes). Si L_1 y L_2 son lenguajes, la unión de L_1 y L_2 es $L_1 \cup L_2 = \{x \mid x \in L_1 \vee x \in L_2\}$. queda a denotar que el conjunto resultante de la unión es aquel conjunto que contiene las cadenas pertenecientes a los lenguajes L_1 y L_2 indistintamente uno de otro. Por lo tanto, si tenemos que $L_1 = \{001, 101, 111\}$ y $L_2 = \{\varepsilon, 001, 000, 010\}$. Entonces:

$$L_1 \cup L_2 = \{\varepsilon, 000,001,010,101,111\}.$$

Otro ejemplo se aprecia a continuación, donde $L_1 = \{ab, c, df\}$ y $L_2 = \{xy, ab, je\}$. Entonces:

$$L_1 \cup L_2 = \{ab, c, df, je, xy\}.$$

El operador que se utiliza para denotar la unión, es el mismo utilizado para las operaciones de conjuntos (U).

1.2.1.1.2. CONCATENACIÓN

Definición 1.9 (Concatenación de Lenguajes). Si L_1 y L_2 son lenguajes, la concatenación de L_1 y L_2 es $L_1L_2 = \{x_1, x_2 \mid x_1 \in L_1 \wedge x_2 \in L_2\}$. Donde cada lenguaje está definido sobre el mismo alfabeto. Se define concatenación de lenguajes a todas las cadenas formadas concatenando una palabra del primer lenguaje, con una del segundo lenguaje. (Cueva Lovelle, 2001)

Tomando la **Definición 1.9**, la concatenación es el conjunto de cadenas que pueden llegar a formarse al tomar cualquier cadena de L_1 y concatenándola con cualquiera de L_2 obteniendo así una nueva cadena resultante, donde la primera posición pertenece al primer lenguaje y el segundo, por ende, al segundo lenguaje y así sucesivamente. Para definir con un operador la concatenación, se hace uso del operador punto (·) o de la omisión de algún operador para darla a denotar. (E. Hopcroft, Motwani, & D. Ullman, 2007)

Para demostrar su funcionamiento, se tiene $L_1 = \{001, 10, 111\}$ y $L_2 = \{\varepsilon, 01\}$. Entonces:

$$L_1L_2 o\ L_1 \cdot L_2 = \{001, 10, 111, 00101, 1001, 11101\}.$$

Como podemos observar las tres primeras cadenas son pertenecientes al lenguaje L_1 y las consiguientes son la concatenación de L_1 con L_2. La razón de las tres primeras cadenas, es dada a una de las propiedades de las expresiones regulares, la cual mantiene toda cadena sin alteración alguna, al ser concatenada a la cadena vacía. (Navarrete Sanchez, y otros, 2008). Estas propiedades serán listadas más adelante.

1.2.1.1.3. POTENCIA DE UN LENGUAJE

Definición 1.10 (Potencia de un lenguaje). También conocida como *cerradura positiva*, se define como la unión de una o más potencias de una cadena de L_1 en un alfabeto. El resultado contiene todas las cadenas, con excepción de la cadena vacía ε. Se denota como $L_1^+ = \bigcup_{n=1}^{\infty} L_1^n$. (Padilla Beltran, 2006)

Desde un punto de vista estricto, se define como la potencia i-ésima de un lenguaje, concatenándolo consigo mismo i-veces. Teniendo el caso de $n = 0$, da como resultado el conjunto vacío (Φ). Para demostrarlo, se tiene el siguiente ejemplo.

Si $L_1 = \{a\}$ entonces L_1^+, tendremos:

$$L_1^+ = \{a^1, a^2, a^3, \dots\}$$

Desglosándolo:

$$L_1^1 = \{a\}$$

$$L_1^2 = \{aa\}$$

$$L_1^3 = \{aaa\}$$

Así sucesivamente hasta llegar al infinito o a la cantidad de concatenaciones deseadas, siempre comenzando desde $n = 1$. Ya que $L_1^0 = \{\varepsilon\}$.

1.2.1.1.4. OPERADOR ESTRELLA O CLAUSURA DE KLEENE

Definición 1.11 (Clausura de Kleene). Dado un lenguaje L, se obtiene un nuevo lenguaje L^*, al igual que la potencia de un lenguaje, representando el conjunto de cadenas que se pueden formar tomando cualquier número de cadenas de L. Definiéndose, así como $L^* = \bigcup_{n=0}^{\infty} L^n$. (Cueva Lovelle, 2001)

Como puede apreciarse, su funcionamiento es idéntico al de la *potencia de un lenguaje*, con la diferencia de que aquí si se toma en cuenta la cadena vacía (ε) y comenzando $n = 0$.

La clausura de Kleene, toma su nombre de quien ideo la notación para las expresiones regulares, Stephen Cole Kleene (1909-1994), quien de igual forma dio pauta a la existencia de este operador. (E. Hopcroft, Motwani, & D. Ullman, 2007).

Como ejemplo, si se tiene $L_1 = \{a\}$ entonces L_1^*, tendremos:

$$L_1^* = \{a^0, a^1, a^2, a^3, \dots\}$$

Habitualmente desglosando cada cadena y tomando en cuenta que siempre la primera cadena es la *cadena vacía*.

1.2.1.2. PROPIEDADES DE LOS LENGUAJES REGULARES

Las expresiones regulares, toman varias de las propiedades que la notación de conjuntos posee, ya que están basadas en el álgebra de conjuntos, pero con sus debidas variedades. Tomamos que α y β son expresiones regulares equivalentes, por lo tanto, α = β y L (α) = L (β). A continuación, se enumeran algunas de las propiedades más recurrentes de las expresiones regulares (Navarrete Sanchez, y otros, 2008). Recordando que Φ es el conjunto vacío y ε es la cadena vacía.

1.- $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$

2.- $\alpha + \beta = \beta + \alpha$

3.- $\alpha + \Phi = \alpha$

4.- $\alpha + \alpha = \alpha$

5.- $\alpha \cdot \varepsilon = \alpha$

6.- $\alpha \cdot \Phi = \Phi$

7.- $\alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma$

8.- $\alpha \cdot (\beta + \gamma) = \alpha\beta + \alpha\gamma$

9.- $\varepsilon^* = \varepsilon$

10.- $\Phi^* = \varepsilon$

11.- $\alpha \cdot \alpha^* = \alpha^* \cdot \alpha$

12.- $\alpha^* = \alpha^* \cdot \alpha^* = (\alpha^*)^*$

13.- $\alpha^* = \varepsilon + \alpha \cdot \alpha^*$

2. CAPÍTULO – AUTOMATAS.

2.1. AUTOMATAS

La palabra *autómata*, hace alusión a imitar las funciones propias de un ser vivo, de una manera más elocuente y enfocada, al movimiento. Dentro del campo de procesadores, compiladores e intérpretes, el movimiento no es el campo fundamental que se estudia y maneja, ya que se evoca más a la simulación de procesos para transformar información. Un autómata dentro del campo de los compiladores, procesadores, intérpretes, por hacer mención de algunos, recibe información que es codificada en cadenas de símbolos y el autómata, que es un dispositivo que manipula las cadenas de símbolos, produce otro tipo de cadenas de símbolos para la salida de datos (Isasi Viñuelas, Martinez Fernandez, & Borrajo Millan, 1997). El resultado obtenido no solo depende del último símbolo, sino de toda la secuencia de caracteres o cadena de símbolos secuencial que se introdujo, esta definición se atribuye a un tipo de autómata llamado *Maquinas secuenciales*.

La *teoría de autómatas* tiene su origen en el campo de la *Ingeniería Eléctrica* (Navarrete Sanchez, y otros, 2008) por el matemático norteamericano Shanon estableciendo las bases para la aplicación de la Lógica Matemática a los circuitos combinados, quien posteriormente permitiría a Huffman en 1954 dar inicio a los circuitos secuenciales y utilizar el concepto *estado de un autómata* y *tabla de transición*. De esta manera Shanon fue desarrollando y formalizando la Teoría de las Maquinas Secuenciales y de los Autómatas Finitos (1956) (Quiroga Rojas, 2008). Retomando lo anteriormente mencionado, se puede definir el concepto fundamental *estado de un autómata.*

Definición 2.1 (Estado de un autómata). Es toda aquella información necesaria en un momento dado, para deducir cuál será el símbolo de salida, dado un símbolo de entrada en ese momento.

Dentro de la teoría de autómatas, existen diversos tipos de autómatas, dentro de ellos caben:

- Autómatas Finitos Deterministas. (AFD)
- Autómatas Finitos No Deterministas. (AFND)

- Autómatas Tipo Pila.
- Autómatas Finitos No Deterministas Con Transición ε. (AFND-ε)
- Máquinas de Turing determinísticas.
- Máquinas de Turing no determinísticas

En otras palabras, el conocer el *estado del autómata* es conocer todo el historial de símbolos de entrada que ha tenido, así mismo el *estado inicial*, el cual es el estado en recibir el primero de los símbolos de entrada. Para aclarar más el tema de autómatas, se tiene el siguiente ejemplo, donde se hará uso de una expresión de lenguaje formal y posteriormente se implementará un autómata en sus grafos correspondientes que lo representan más común y coloquialmente dentro del campo de las ciencias de la computación.

Ejemplo 1. Obtener un autómata que acepta el lenguaje $L = 1(1^{*}01^{*}01^{*})^{*}$.

Para el **Ejemplo 1**, se tiene el autómata representado en la **Figura 1** donde es posible agregar un sinfín de 1's consecutivos de un 0, nuevamente varios 1's y finalizar siempre con un 0. Cabe recordar, que la operación Kleene, permite añadir o no la cadena o caracteres a los que afecta, por ello es posible el autómata representado en la **Figura 1**. Posteriormente se explicará su composición y la obtención de un autómata.

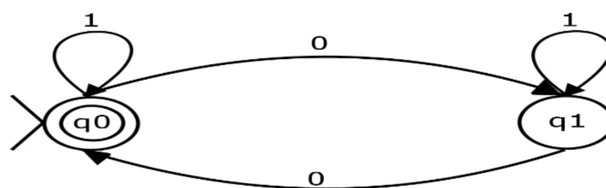


Figura 1. Diagrama de autómata

2.1.1. AUTOMATAS FINITOS DETERMINISTAS (AFD)

Un *autómata finito deterministas*, es un modelo matemático con un sistema que recibe parámetros de entrada y posee salidas discretas, tal y como se mencionó con anterioridad, poseyendo de igual forma un conjunto de estados *configuraciones internas*. 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 *Gramáticas Regulares*. (Moral)

Los AFD (*autómata finito determinista*) se utilizan generalmente para analizar cadenas de caracteres y verificar si pertenecen o no a un determinado lenguaje, así mismo son utilizados como analizadores en la traducción de algoritmos para los ordenadores, este tipo de analizadores, son mayormente conocidos como *analizadores lexicográficos*. Las gramáticas y sus reglas de producción, son utilizadas más frecuentemente para el análisis de la sintaxis de los lenguajes de programación. Todo lo mencionado anteriormente, aunado a los autómatas finitos.

Un AFD, se define de igual forma como una quíntupla, donde la colección se forma de la siguiente forma $M = (Q, \sum, S, F, \delta)$:

- Un alfabeto de entrada $\sum$.
- Una colección finita de estados Q.
- Un estado inicial S.
- Una colección de estados finales o de aceptación F.
- Una función $\delta: Q \times \sum \longrightarrow Q$ que determina el único estado siguiente para el par (q_i, σ) correspondiente al estado actual y la entrada.

Donde el nombre *determinista*, proviene de la forma en como está definida la función de transición, es decir, si en un instante *t*, se encuentra en el estado *q* y lee el símbolo *a*, entonces, en el instante *t+1*, se cambia al siguiente estado el cual se conoce con seguridad, que descrito en otras palabras se define como $\delta(q, a)$. (Quiroga Rojas, 2008)

Ejemplo 2. Considerando el lenguaje regular $\mathrm{A} = c^*(\mathrm{ab}c^*)^*$. Dada una cadena ω, verificar que ω pertenece a A.

Para dar solución al **Ejemplo 2**, no solamente se deben verificar los caracteres pertenecientes a ω, también se deben se vislumbrar sus posiciones. Por ejemplo, la cadena abc^5 se encuentra en A, pero $cabac^3$ no lo está. Para ello construye un diagrama el cual permita determinar los distintos miembros del lenguaje.

De esta forma y siguiendo con el **Ejemplo 2**. Se obtiene el autómata mostrado en la **Figura 2**.

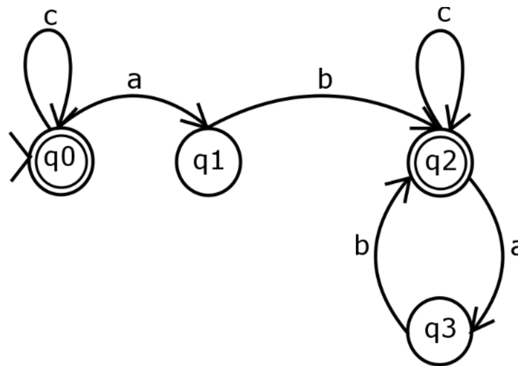


Figura 2. Grafo resultante del **Ejemplo 2**.

Los AFD, pueden ser representado mediante:

- Tablas de transición o estados.
- Diagrama de Moore.

El diagrama de Moore, es una forma de representación gráfica de las transiciones existentes en el autómata, especificando el estado actual y a que estado existe una transición, mediante un carácter determinado (Cueva Lovelle, 2001). La **Figura 2** muestra un ejemplo de un diagrama de Moore. Tiene la forma de un *grafo* dirigido con información adicional añadida, y también es llamado *diagrama de transición*. Los nodos del grafo son llamados *estados* (Ver **Figura 3**) y se van uniendo para señalar hasta que lugar se ha ido analizando la cadena entrante. Las aristas del grafo se etiquetan con caracteres del alfabeto manejado y se llaman *transiciones* (Ver **Figura 4**).

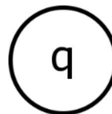


Figura 3. Estado de un autómata, representación gráfica.

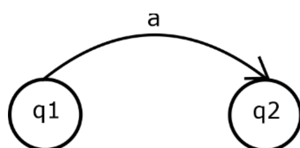


Figura 4. Transición de un autómata.

El diagrama de Moore es un grafo orientado en lo que cada nodo corresponde a un estado; y si se tiene que $\delta(\varepsilon, q_i) = q_j$ y $\delta(\varepsilon, q_i) = F$, entonces existe un arco dirigido a un nodo q_i al correspondiente q_j. La **Figura 5** representa otro autómata mediante el diagrama de Moore.

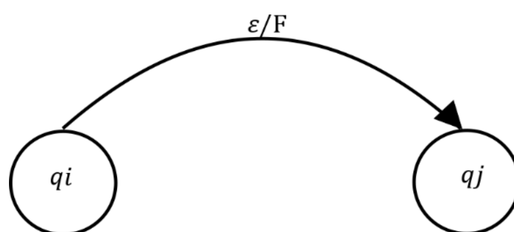


Figura 5. Ejemplo de diagrama de Moore

La tabla de transiciones es una tabla donde las filas son el conjunto de estados y las columnas las entradas con las que cuente el autómata. De esta forma un autómata representado mediante el diagrama de Moore, puede ser presentado como una tabla de transiciones y viceversa.

Para el **Ejemplo 2**, se obtiene una tabla de transiciones, pero primeramente es necesario identificar el alfabeto que acepta el autómata. Como puede observarse, es posible obtener todos los elementos de la quíntupla que se ha mencionado con anterioridad para poder definir de *formal* el autómata y la tabla de transiciones. El autómata acepta un alfabeto de entrada $\sum = \{a, b, c\}$, la cantidad de estados son $Q = \{q_0, q_1, q_2, q_3\}$, el estado inicial $S = \{q_0\}$ y el estado final $F = \{q_0, q_2\}$. Posterior a ello, es posible obtener la tabla de estados, representada de la **Figura 6**.

δ	*a*	*b*	*c*
q_0	q_1	-	q_0
q_1	-	q_2	-
q_2	q_3	-	q_2
q_3	-	q_2	-

Figura 6. Tabla de estados resultante, perteneciente a la **Figura 2**.

Como se observa, es el producto cruz del número de estados que se tienen, con el alfabeto presenta para así obtener la transición a cada estado, mediante cada símbolo. De esta forma comienzan a avanzar los caracteres de la cadena a través del *grafo*, donde si el siguiente carácter a reconocer concuerda con la etiqueta de alguna *transición* que parte del estado actual, nos desplazamos al estado siguiente que lleve la arista correspondiente. El comienzo de este grafo es el *estado inicial*, donde se coloca una cola de pato (Ver **Figura 7**) y para finalizar y saber que la cadena es aceptada por el lenguaje, es necesario un *estado final* (Ver **Figura 8**) el cual representa el término de nuestro *grafo* para el conjunto de caracteres implementados en nuestra cadena.

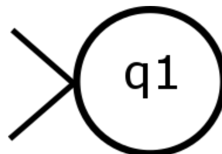


Figura 7. Representación del estado inicial de un autómata.

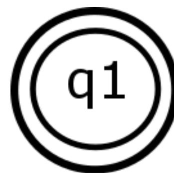


Figura 8. Estado final de un autómata.

Un AFD, es inicializado, con la entrada de una palabra ω de la siguiente manera:

1. ω es agregada al inicio de la pila de entrada, separando de esta forma, cada carácter de manera individual.
2. El apuntador de lectura del AFD, apunta al símbolo más a la izquierda de ω.
3. El estado actual, para a ser q_0.

Ejemplo 3. Considerando el lenguaje regular:

$$L = \{\omega = (a, b)^* | \omega \text{ tiene una a al final de la cadena.}$$

$$\text{Y las subcadenas de b}', \text{si las hay, seran continuas e impares.}\}$$

Por lo que el orden léxico gráfico será:

$$L = \{a, aa, ba, aaa, baa, aba, a^4, ba^3, abaa, aaba, bbba, \dots\}$$

La gramática perteneciente al lenguaje será:

$$L(G) = ((b(bb)^* \cup \varepsilon)^* a^+)^+$$

El autómata resultante que reconoce el lenguaje, está presente en la **Figura 9**.

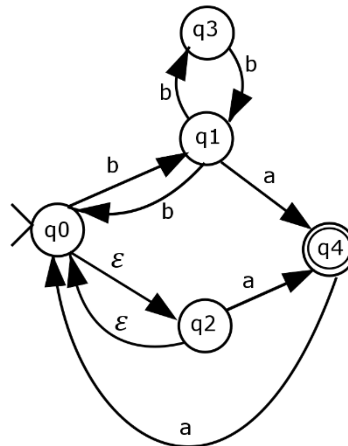


Figura 9. Autómata resultante del **Ejemplo 3**.

Retomando la **Figura 9**, se tiene a modo de ejemplo la cadena aaaba. De esta forma y con lo anteriormente mencionado, es retomar el símbolo más a la izquierda para poder *inicializar* el autómata, entonces se inicia con el carácter <<a>> (**Figura 10**).

Figura 10. Caracterización de la cadena aaaba.

Una vez que el autómata ha *iniciado*, comienza su ejecución tras el siguiente ciclo:

1. Se lee el símbolo actual, que es la cabecera de la pila de lectura. Si el cabezal, apunta al *vacío*, entonces el AFD finaliza, *aceptando* la palabra, si y solo si, el estado actual es el final.
2. Se calcula el estado siguiente en base al estado y símbolo actual según la tabla de transiciones, obteniendo de esta forma la siguiente ecuación: $\delta(estado\ actual, simbolo\ actual) = estado\ siguiente.$

Para el **Ejemplo 3**, se obtiene la siguiente tabla de estado como resultado mostrada en la **Figura 11**.

δ	*a*	*b*	*ε*
q_0	-	q_1	q_2
q_1	q_4	q_0, q_3	-
q_2	q_4	-	q_0
q_3	-	q_1	-
q_4	q_0	-	-

Figura 11. Tabla de transiciones resultante del autómata del **Ejemplo 3**.

3. El apuntador de lectura, se mueve a la siguiente posición hacia la derecha.
4. El estado siguiente, pasa a ser el estado actual y se regresa al paso 1.

Figura 12. En base al algoritmo mostrado, va cambiando el apuntador hasta finalizar la palabra ω.

Como muestra la **Figura 12**, el apuntador se va desplazando hasta finalizar la palabra según las transiciones y los estados determinados dentro del autómata hasta finalizar y aceptar la cadena.

Una de las diferencias fundamentales entre las diversas clases de autómatas que se mencionaron anteriormente, recae si dicho control o manejo de estados, es *determinista*, o *no determinista* lo cual significa que se puede estar en varios estados a la vez. De esta forma el *no determinismo* logra definir un lenguaje de una forma más rápida y *sencilla*, a diferencia de un *autómata determinista*. Una de las ventajas sobre los *autómatas no deterministas*, es que permiten diseñar soluciones más flexibles dentro de los problemas presentados a los lenguajes de alto nivel, pero no es fácil de programar. El *autómata determinista* a veces no es fácil de diseñar, pero es fácil de programar. (E. Hopcroft, Motwani, & D. Ullman, 2007)

2.1.1.1. AUTOMATAS FINITOS NO DETERMINISTAS (AFND)

Como se apreció en las secciones anteriores, se manejó solamente el determinismo de un autómata finito (AFD), para esta nueva sección, se emplea lo contrario, el no-determinismo. Al igual que los ADF, los AFND son una quíntupla, con una diferencia en la función de transiciones, donde es agregada la característica del *no determinismo*. Este *no determinismo*, se presenta como el desconocimiento de caminos hacia cada estado en base a un símbolo, esto es: a partir del estado actual, con el símbolo actual de entrada, no es posible <<determinar>> de forma exacta, cuál va a ser el estado siguiente dentro del autómata. Para explicar mejor lo anteriormente descrito, se tiene el

Ejemplo 4.

Ejemplo 4. Dado el lenguaje regular:

$$(a \cup b)^*(aa \cup bb)(a \cup b)^*$$

Se muestra el autómata en la **Figura 13** que acepta las cadenas generadas por tal expresión regular. Su tabla de transiciones se muestra en la **Figura 14**.

En la tabla de transiciones obtenida por el

Ejemplo 4, es posible apreciar que, en algunas de las transiciones de los estados, existe más de una transición con uno solo de los símbolos, esta parte viene a definirse como el *no determinismo* de esta clase de autómatas (Navarrete Sanchez, y otros, 2008), ya que como hemos de recordar, un AFD se caracteriza por tener una sola transacción por un símbolo determinado.

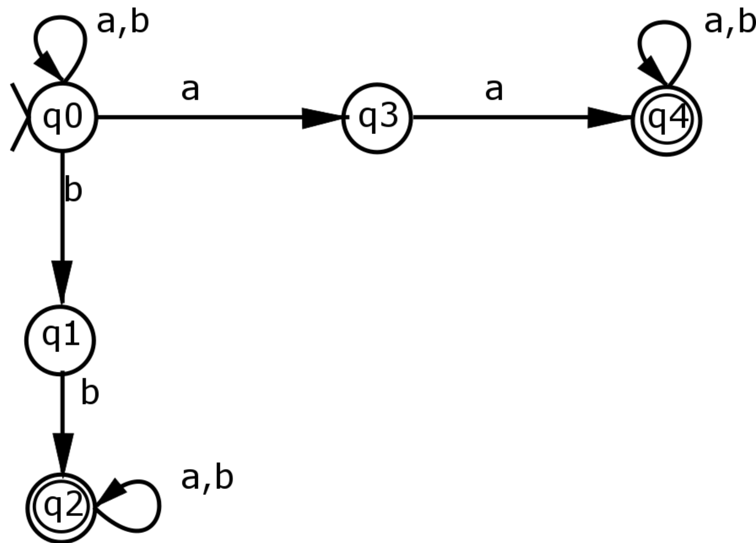


Figura 13. Ejemplo de un AFND.

δ	*a*	*b*
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	-	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	

q_4	$\{q_4\}$	$\{q_4\}$

Figura 14. Tabla de transiciones correspondiente a la **Figura 13**.

A pesar de ser autómatas, en ocasiones, *complejos*, son un concepto útil para probar teoremas y simplificar descripciones de los autómatas. Esto no significa que un autómata con diversas transiciones sea propio un AFND, también es posible que sea aceptado por un AFD, de esta forma es posible decir que cualquier autómata aceptado por un AFND es aceptado por un AFD. (Campos, 1995)

2.1.1.1.1. AUTOMATAS FINITOS NO DETERMINISTAS CON TRANSICION ε

Este tipo de autómatas, son parte de la *familia* de los AFND, con sus mismas propiedades y características, la diferencia recae en sus transiciones entre un estado a otro, ya que estas transiciones pueden ser realizadas sin la necesidad de consumir alguna entrada de un carácter como tal (Dean, 2001). Este tipo de transiciones son representadas por la cadena vacía (ε). Al igual que los AFD, los AFND, los AFND-ε, son una quíntupla que difiere de los demás en la función de transición y en la habilidad que poseen de permitir una cantidad variada de caminos a elegir en base al estado en el que se encuentran y el símbolo de entrada que se tenga (Lewis & Papadimitriou, 1998); los AFND-ε a diferencia de los AFND, que también permite la transición a varios estados con un solo carácter, se permite dicha transición en base al carácter nulo o cadena vacía. (E. Hopcroft, Motwani, & D. Ullman, 2007)

Este tipo de transiciones *vacías* o nulas, dan una nueva capacidad y versatilidad a los autómatas. Si se tiene una transición nula, el autómata puede quedarse en el estado actual o cambiar a algún otro estado que permita llegar a él sin la necesidad de consumir algún carácter como entrada. La **Figura 15**, muestra un autómata, por el cual, dos de sus transiciones son mediante el vacío y aun así cuentan con transiciones con caracteres con un valor nulo.

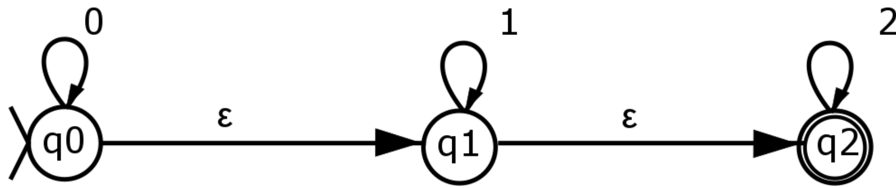


Figura 15. Representación de un AFND- ε.

Como se mencionó con anterioridad y al igual como sucede con los AFND, la tabla de transiciones es distinta a los demás, la diferencia recae que, dentro de los caracteres del lenguaje aceptado, existe el símbolo ε, de esta forma también se toman en cuenta aquellos cambios de estado que sean representados por el vacío. Aunque parece un cambio muy simple, lleva una gran funcionalidad, permitiendo de esta forma la lectura de diversos caracteres y un camino de inicio a fin, con espacios *invisibles* que no contribuyen o afectan a la cadena de entrada. El **Ejemplo 5**, muestra el uso de los AFND-ε de una manera práctica.

Ejemplo 5. Un autómata, que acepte:

- Un signo +, - o que no exista signo.
- Una cadena de dígitos.
- Un punto decimal.
- Otra cadena de dígitos.

La cadena del punto 4 o del punto 2, pueden llegar a ser vacías, aunque al menos una de las dos cadenas, debe tener dígitos.

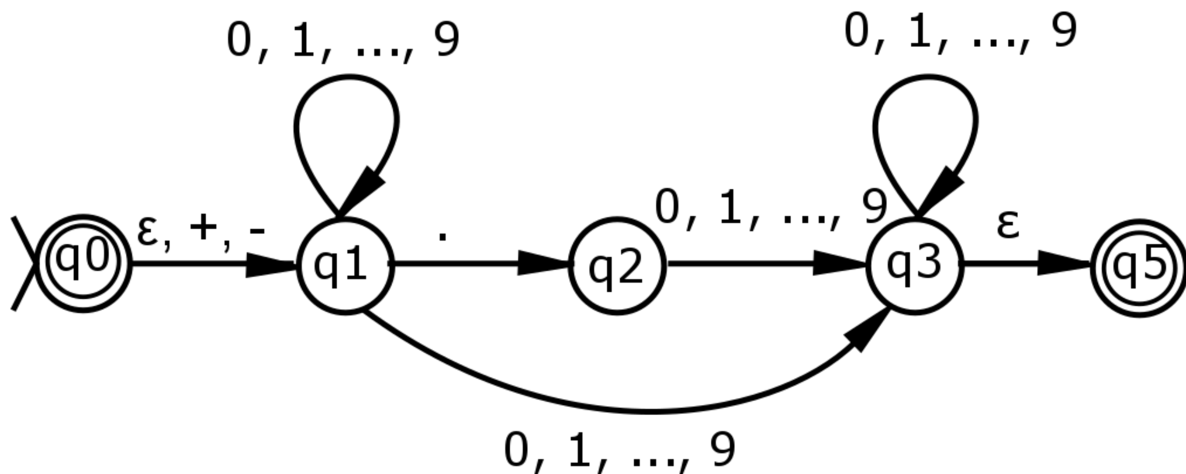


Figura 16. Autómata resultante del **Ejemplo 5**, haciendo uso de la cadena vacía en las transiciones de un estado a otro para obtener una solución *congruente*.

Como se aprecia en la **Figura 16**, la primera transición mostrada, de q_0 a q_1, se logra también, mediante el consumo de la cadena vacía (mejor dicho, sin el consumo de algún carácter) lo cual es redituable ante ciertos casos, permitiendo de esta forma resolver el ejercicio propuesto de manera temporal, pero al mismo tiempo presentando otro tipo de problemática. El problema recae en el uso de los tipos de autómatas, ante una máquina de estados, ya que las máquinas de estados solo aceptan AFD por ello, y como se ha venido comentando, el *no determinismo* es una manera compleja de dar solución a diversos problemas, pero también una solución óptima en diversos casos como el presentado con anterioridad.

Hay que recordar que todo autómata del tipo AFND, sin importar a que *familia* pertenezca, puede ser *transformado* en un AFD.

2.1.1.2. TRANSFORMACION DE AFND- ε A AFD

La transformación de un AFND-ε a AFD, lleva a cabo el siguiente algoritmo aquí presentado tomando en cuenta que $M = (Q, \sum, S, F, \delta)$ es un AFND y para poder obtener su *determinación*, es necesario construir un autómata $M' = (Q', \sum, S', F', \delta)$ equivalente a M. La idea principal es ver un AFND como *ocupante*, en cualquier momento, no en un solo estado, es decir, todos los estados a los que se puede llegar desde el estado inicial a través de la entrada consumida hasta el momento. Si M tiene 5 estados $\{q_0, \ldots, q_4\}$ y, después de leer una determinada cadena ω, y

algunos de los caracteres pertenecientes a ω aparecieran solo en q_0, q_2 o q_3, pero no en q_1 o q_4, este nuevo estado podría ser considerado por el conjunto de $\{q_0, q_2, q_3\}$ o como un indeterminado miembro del conjunto. (Lewis & Papadimitriou, 1998)

El conjunto de estados finales de M' estará constituido por un subconjunto de estado, el cual contendrá al menos un estado final de M. La definición de la función de transiciones de M', será un poco más complicada. La idea básica recae en que al ejercer algún movimiento sobre M', al leer un símbolo $a \in \Sigma$ imita un movimiento sobre M al entrar el mismo símbolo, *posiblemente seguido por algún número de e-movimientos de M.* (Lewis & Papadimitriou, 1998)

Para poder explicar con mayor veracidad el algoritmo e ir dando un ejemplo de lo comentado, se hará uso del autómata presente en la **Figura 17**. Como es de apreciarse, permite la transición de un estado a otro en base a la cadena vacía. El lenguaje aceptado por este autómata es el siguiente:

$$L = (ab)^*(ba)^* \cup aa^*$$

Para dar solución al autómata presentado, fue *eficiente*, el generar un autómata del tipo AFND-ε para las transiciones y la unión presente entre $(ab)^*(ba)^*$ y aa^*.

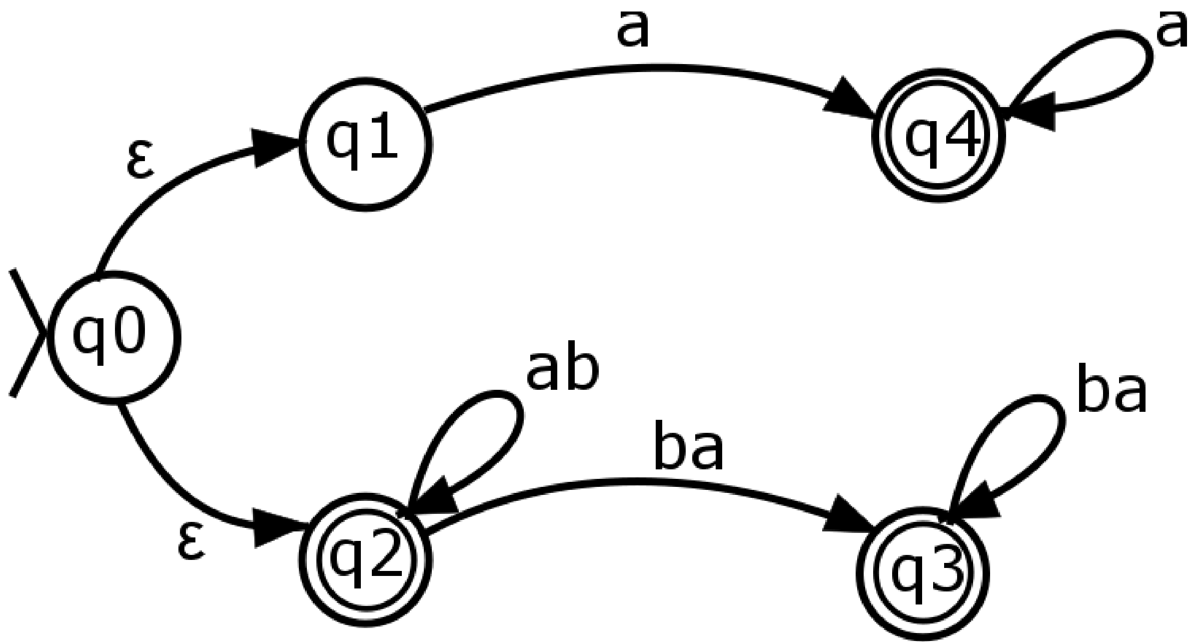


Figura 17. Autómata generado para el lenguaje mencionado anteriormente y fuente para el ejemplo presente en la explicación del algoritmo de transformación.

El primer paso dentro del algoritmo, de transformación de un AFND-ε a AFD, consiste en separar todas aquellas transiciones que contengan más de un solo carácter. Esto para poder obtener una propiedad de unicidad entre transición de un estado a otro. La **Figura 18**, muestra el autómata bajo la unicidad de transiciones.

1) Unicidad de transiciones de un estado a otro.

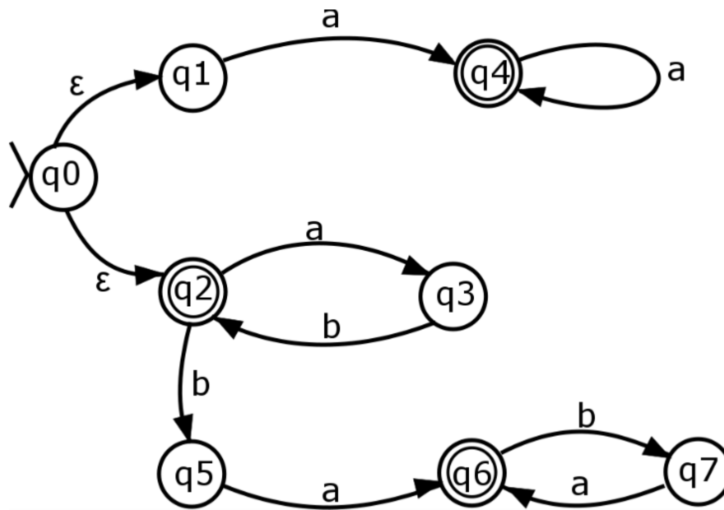


Figura 18. Autómata resultante al aplicar la unicidad de transiciones.

El siguiente paso consiste en obtener una tabla de transiciones, de todas aquellas transiciones que pueden ser logradas mediante la cadena vacía. Cabe aclarar que estas transiciones incluyen todos y cada uno de los estados, ya que para llegar a cada estado por sí mismo, puede ser realizado mediante el vacío, por lo tanto, todos los estados poseen una transición mediante el vacío.

Dentro de la tabla de transiciones formada en la **Figura 19**, es posible el obtener cual va a ser el estado inicial del nuevo autómata a generarse. Para obtener esta *definición*, se tomará siempre como nuevo estado inicial, aquella transición que posea el actual estado inicial. Por lo tanto, la transición presente en la **Figura 20**, pasa a ser el nuevo estado inicial **S**.

Esta *definición*, pasa a ser el tercer paso del algoritmo de transformación.

2) Transiciones con la cadena vacía.

$E(q_0)$	$\{q_0, q_1, q_2\}$
$E(q_1)$	$\{q_1\}$
$E(q_2)$	$\{q_2\}$
$E(q_3)$	$\{q_3\}$
$E(q_4)$	$\{q_4\}$
$E(q_5)$	$\{q_5\}$
$E(q_6)$	$\{q_6\}$
$E(q_7)$	$\{q_7\}$

Figura 19. Tabla de transiciones resultante de todas aquellas transiciones que sean realizadas mediante la cadena vacía.

3) Obtener el nuevo estado inicial del AFD.

$E(q_0)$	$\{q_0, q_1, q_2\}$

Figura 20. Nuevo estado inicial, denominado como **S**.

El cuarto paso, consiste en obtener todos los demás estados, en base a los caracteres pertenecientes al lenguaje, para así obtener las nuevas transiciones del AFD. El comienzo va en base al nuevo estado inicial **S**, donde debe obtenerse el nuevo estado en base al conjunto de estados de transición, por el cual está formado y formando los nuevos estados sucesivamente.

4) Obtener los nuevos estados y transiciones.

$S(a)$	(q_0, a)	-
	(q_1, a)	$\{q_4\}$
	(q_2, a)	$\{q_3\}$
	$E(q_4) \cup E(q_3) = \{q_3, q_4\}$	Q_1

$S(b)$	(q_0, b)	-

	$(\boldsymbol{q_1}, \boldsymbol{b})$	-
	$(\boldsymbol{q_2}, \boldsymbol{b})$	$\{q_5\}$
	$\boldsymbol{E}(\boldsymbol{q_5}) = \{\boldsymbol{q_5}\}$	Q_2

Como se puede observar en las funciones anteriores, Se han obtenido dos nuevos conjuntos de estados, $\{\boldsymbol{q_3}, \boldsymbol{q_4}\}$ y $\{\boldsymbol{q_5}\}$, dando lugar a Q_1 y Q_2 respectivamente. De esta forma, se van a ir formando los nuevos estados y transiciones, si alguno de estos conjuntos de estados pertenecientes al autómata no determinístico vuelve a presentarse en alguna función, no se genera un nuevo estado en el autómata determinístico, se indica que regresa al estado ya existente.

$\boldsymbol{Q_1}(\boldsymbol{a})$	$(\boldsymbol{q_4}, \boldsymbol{a})$	$\{\boldsymbol{q_4}\}$
	$(\boldsymbol{q_3}, \boldsymbol{a})$	-
	$\boldsymbol{E}(\boldsymbol{q_4}) = \{\boldsymbol{q_4}\}$	Q_3

$\boldsymbol{Q_1}(\boldsymbol{b})$	$(\boldsymbol{q_4}, \boldsymbol{b})$	-
	$(\boldsymbol{q_3}, \boldsymbol{b})$	$\{\boldsymbol{q_2}\}$
	$\boldsymbol{E}(\boldsymbol{q_2}) = \{\boldsymbol{q_2}\}$	Q_4

$\boldsymbol{Q_2}(\boldsymbol{a})$	$(\boldsymbol{q_5}, \boldsymbol{a})$	$\{\boldsymbol{q_6}\}$
	$\boldsymbol{E}(\boldsymbol{q_6}) = \{\boldsymbol{q_6}\}$	Q_5

$\boldsymbol{Q_2}(\boldsymbol{b})$	$(\boldsymbol{q_5}, \boldsymbol{b})$	-
	-	-

$\boldsymbol{Q_3}(\boldsymbol{a})$	$(\boldsymbol{q_4}, \boldsymbol{a})$	$\{\boldsymbol{q_4}\}$
	$\boldsymbol{E}(\boldsymbol{q_4}) = \{\boldsymbol{q_4}\}$	Q_3

$\boldsymbol{Q_3}(\boldsymbol{b})$	$(\boldsymbol{q_4}, \boldsymbol{b})$	-

	-	-

$\boldsymbol{Q_4(a)}$	$\boldsymbol{(q_2, a)}$	$\{\boldsymbol{q_3}\}$
	$\boldsymbol{E(q_3) = \{q_3\}}$	Q_6

$\boldsymbol{Q_4(b)}$	$\boldsymbol{(q_2, b)}$	$\{\boldsymbol{q_5}\}$
	$\boldsymbol{E(q_5) = \{q_5\}}$	Q_2

Como se puede apreciar en este último caso, se repite nuevamente un conjunto que apareció como resultado con anterioridad. Como se comentó, no se genera un nuevo estado, se retoma el ya generado.

$\boldsymbol{Q_5(a)}$	$\boldsymbol{(q_6, a)}$	-
	-	-

$\boldsymbol{Q_5(b)}$	$\boldsymbol{(q_6, b)}$	$\{\boldsymbol{q_7}\}$
	$\boldsymbol{E(q_7) = \{q_7\}}$	Q_7

$\boldsymbol{Q_6(a)}$	$\boldsymbol{(q_3, a)}$	-
	-	-

$\boldsymbol{Q_6(b)}$	$\boldsymbol{(q_3, b)}$	$\{\boldsymbol{q_2}\}$
	$\boldsymbol{E(q_2) = \{q_2\}}$	Q_4

$\boldsymbol{Q_7(a)}$	$\boldsymbol{(q_7, a)}$	$\{\boldsymbol{q_6}\}$
	$\boldsymbol{E(q_6) = \{q_6\}}$	Q_5

$\boldsymbol{Q_7(b)}$	$\boldsymbol{(q_7, b)}$	-

Como es de apreciarse, ya no existen nuevos estados que procesar, por lo tanto, se procede a construir el nuevo autómata. Como se mencionó anteriormente, el nuevo estado inicial es la función vacía del estado inicial del autómata no determinístico, y para los estados finales se aplica la misma conjetura, haciendo estado final del autómata determinístico a todo conjunto de estados que contenga un estado final del autómata no determinístico.

Una vez cotejados todos los estados y transiciones, se obtiene el autómata mostrado en la **Figura 21**. De esta forma, se obtiene el nuevo autómata del tipo AFD.

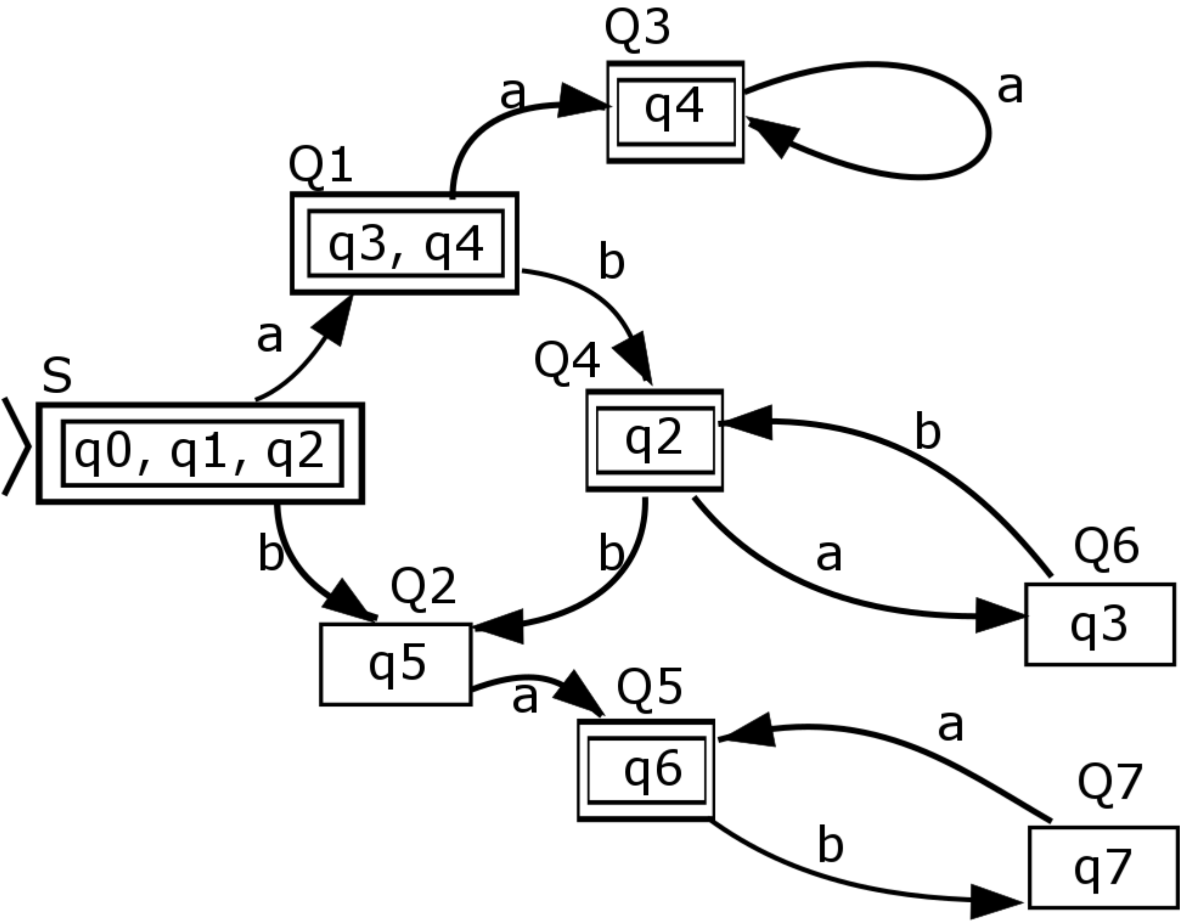


Figura 21. Autómata resultante del algoritmo de transformación.

3. CAPÍTULO – LENGUAJES LIBRES DE CONTEXTO Y GRAMÁTICAS REGULARES

3.1. LENGUAJES LIBRES DE CONTEXTO

Los Lenguajes Libres de Contexto (LLC por sus abreviaturas en español) forman una clase de lenguaje, aún más amplia que los Lenguajes Regulares, de acuerdo a lo definido en la Jerarquía de Chomsky. Estos lenguajes, son importantes, ya que, vistos desde la parte teórica, relacionan a las llamadas *Gramáticas Libres de Contexto* con los Autómatas de tipo Pila; y vistos desde un punto más práctico, casi todos los lenguajes de programación se encuentran basados en los LLC. (Vazquez Palma)

Para este tipo de lenguajes, el análisis automático, es computacionalmente más eficiente que al de otra clase de lenguajes que son más generales. Para poder comprender este tipo de lenguajes, es necesario entender algunos conceptos primero.

- **Regla.** Expresión de la forma:

$$\alpha \rightarrow \beta$$

En donde, tanto α como β son cadenas de símbolos en donde se puede aparecer tantos elementos del alfabeto Σ (llamado constantes) como símbolos nuevos (llamados variables).

- **Gramáticas.** Básicamente, es un conjunto de reglas.

Una *Gramática Libre de Contexto*, es un conjunto finito de variables (también denominados *no-terminales*) de las cuales, cada una representa un lenguaje en particular. Una de las características importantes y más notorias de las *Gramáticas Libres de Contexto*, es como se definen recursivamente en términos de otros y de diversos símbolos denominados *terminales*. (Campos, 1995)

Uno de los motivos principales, que toma parte en las *Gramáticas Libres de Contexto*, fue el poder describir los lenguajes naturales con una mayor fluidez y entendimiento a los usuarios tal y como se puede apreciar en los lenguajes de programación de alto nivel y en sus descripciones. (Campos, 1995)

3.2. DEFINICIÓN DE GRAMÁTICA

Una gramática es una cuádrupla de la forma:

$$G = (V, T, S, P)$$

Donde:

- Conjunto finito de símbolos no-terminales V.
- Conjunto finito de símbolos terminales T.
- Símbolo inicial y pertenece a V, S.
- Conjunto de producciones o reglas de derivación P.

De esta forma, todos los lenguajes formados por una gramática, están formados con símbolos del *vocabulario terminal* T. Este tipo de vocabulario, se define por enumeración de los símbolos terminales. (Cueva Lovelle, 2001)

El *vocabulario no-terminal* V, es el conjunto de símbolos introducidos como elementos auxiliares para poder definir la gramática. No figuran en la sentencia establecida y es definido por enumeración de los símbolos no-terminales. (Cueva Lovelle, 2001)

El *símbolo inicial* S, es un símbolo *no-terminal*, el cual da inicio a aplicar las reglas de la gramática para obtener las distintas cadenas del lenguaje. (Cueva Lovelle, 2001)

Las *producciones* P, son el conjunto de reglas que deben aplicarse desde el símbolo inicial S para obtener las cadenas del lenguaje. El conjunto P, se define en base a las enumeraciones de las distintas producciones, en forma de reglas o por medio de un metalenguaje, por ejemplo, BNF (Backus Naur Form) o EBNF (Extended Backus Naur Form). (Cueva Lovelle, 2001)

Al igual que las gramáticas, las expresiones regulares y los autómatas finitos, nos proporcionan medios para explicar o definir lenguajes. Las expresiones regulares proporcionan una plantilla o patrón para las cadenas del lenguaje, de esta forma, todas las cadenas que correspondan con el patrón, serán las únicas pertenecientes

al lenguaje. Igualmente, los autómatas finitos especifican un lenguaje como el conjunto de todas las cadenas que lo hacen pasar del estado inicial a uno de sus estados de aceptación. Otra forma de interpretar a los autómatas, es como un generador de cadenas del lenguaje. (Dean, 2001)

Estas gramáticas, han *jugado* un papel importante dentro de la tecnología de los compiladores desde los años 70's, han permitido la creación e implementación de analizadores sintácticos, que son una parte importante para la estructura de un programa. De esta forma, algunos trabajos rutinarios que consumían mucho tiempo, fueron simplificados en base a estas nuevas funcionalidades. Dentro de las gramáticas existe otro tipo de autómata denominado *Autómata a pila*, los cuales son encargados únicamente de describir aquellos lenguajes independientes del contexto, aun mas especifico, gramáticas independientes del contexto. (E. Hopcroft, Motwani, & D. Ullman, 2007)

3.2.1. JERARQUÍA DE LAS GRAMÁTICAS

Noam Chomsky, definió cuatro tipos de gramáticas en función de sus reglas de derivación. La clasificación, da inicio con el tipo de gramáticas que pretender ser universales, aplicando las debidas restricciones a sus reglas de derivación es como se obtienen los otros tres tipos de gramáticas. (Cueva Lovelle, 2001)

La **Figura 22**, muestra la clasificación de lenguajes determinada por Chomsky dentro del universo de los lenguajes.

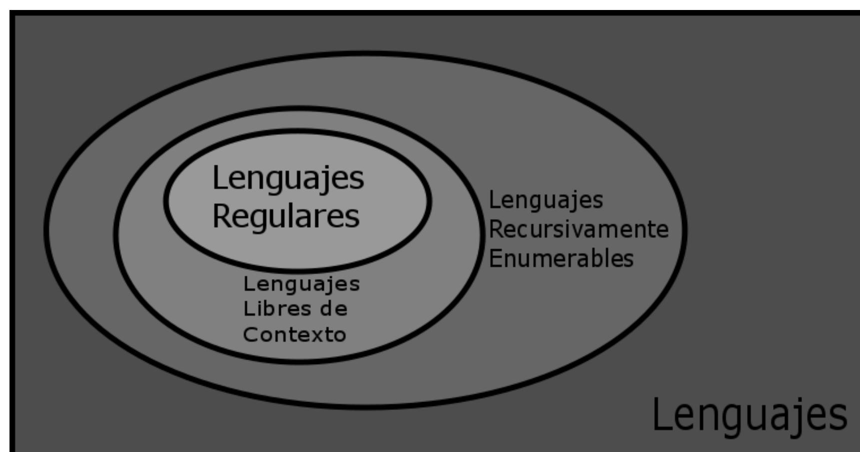


Figura 22. Clasificación de lenguajes según N. Chomsky.

Gramática tipo 0

También llamadas *gramáticas no restringidas* o *gramáticas con estructura de fase*. Es el tipo de gramáticas más generales, por ello también adquieren el nombre de *gramáticas sin restricciones*. Esto quiere decir, que las producciones pueden ser de cualquier tipo permitido. (Navarrete Sanchez, y otros, 2008)

Las gramáticas de tipo 0, son de la forma:

$$\alpha \rightarrow \beta$$

Donde:

$$\alpha \in (V \cup T)^+$$

$$\beta \in (V \cup T)^*$$

Es decir, la única restricción es que no puede haber reglas de la forma:

$$\lambda \rightarrow \beta$$

Donde, λ es la cadena vacía.

Este tipo de gramáticas, generan lenguajes *recursivamente enumerables*, que son aceptados en la *máquina de Turing*. Recordando que es un autómata de tipo *determinístico* o *no determinístico.* (Padilla Beltran, 2006)

Gramática tipo 1

También son conocidas como *sensibles al contexto*, y sus reglas de producción son de la forma:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

Siendo:

$$A \in V$$

$$\alpha, \beta \in (V \cup T)^*$$

$$\gamma \in (V \cup T)^+$$

Generalmente son llamadas *sensibles al contexto*, ya que se puede reemplazar A por γ siempre y cuando se esté en el contexto $\alpha \dots \beta$. Por lo tanto, las producciones de tipo:

$$S \rightarrow \gamma$$

Son aceptadas siempre y cuando S no pertenezca a la parte derecha de ninguna de las reglas de producción. (Navarrete Sanchez, y otros, 2008)

Otra forma de ver las reglas de producción es el especificar que una variable A puede ser reemplazada por γ en una derivación directa, cuando A aparezca en el *contexto* de α y β. Es por esta última característica, que son llamadas *sensibles al contexto* (Navarrete Sanchez, y otros, 2008). Cabe aclarar, que este tipo de producciones siempre cumplen, que la parte izquierda tiene una longitud menor o igual que la parte derecha, pero nunca mayor, por lo tanto, es una gramática *no contráctil*. La única excepción presente es el caso ya visto:

$$S \rightarrow \gamma$$

Gramática tipo 2

Se denominan *gramáticas libres de contexto*, este tipo de gramáticas solo permiten tener un símbolo no-terminal en la parte izquierda de su producción y en la parte derecha, puede ser una combinación entre *terminales* y *no-terminales*, es decir, son de la forma:

$$A \rightarrow \alpha$$

Donde:

$$A \in V$$

$$\alpha \in (V \cup T)^{+}$$

Las producciones obtenidas por este tipo de gramáticas, son aceptadas por los *autómatas de tipo pila*. Se denominan *libres de contexto*, porque puede cambiar A por α, independientemente del contexto en que aparezca A. (Padilla Beltran, 2006)

Gramática tipo 3

También llamadas *regulares*, comienzan sus reglas de producción, por un símbolo terminal, el cual puede llegar a ser seguido por un símbolo no-terminal.

Existen dos tipos:

- *Lineales por la derecha*. Todas sus producciones son de la forma:

$$A \rightarrow bC$$

Donde:

$$A, C \in V$$

$$b \in T$$

- *Lineales por la izquierda*. Todas sus producciones son de la forma:

$$A \rightarrow Cb$$

Donde:

$$A, C \in V$$

$$b \in T$$

Este tipo de lenguajes, tiene un símbolo *no-terminal* en el lado izquierdo de la producción, en el lado derecho, debe contener una cadena formada de símbolos *terminales* y *no-terminales*, teniendo como máximo, un *no-terminal*, el cual debe estar del lado derecho o izquierdo de la cadena. (Padilla Beltran, 2006)

Como es posible apreciar, y en base a todo lo mencionado hasta el momento, tanto los autómatas (ya sean *determinísticos*, *no-determinísticos* o *tipo pila*), aceptan alguno de los tipos de lenguajes manejados por Noam Chomsky, dentro de los cuales, se han basado los análisis lexicográficos y sintácticos, que han permitido y han dado paso a la resolución de diversos problemas presentes dentro del mundo de la computación y más apegado a los compiladores que permiten el interactuar

con los ordenadores en base a instrucciones en un *lenguaje* más comprensible para todos y no solamente aquel que la maquina comprende.

3.2.2. DERIVACIÓN UTILIZANDO UNA GRAMÁTICA

Existen 4 componentes importantes dentro de las gramáticas regulares y los LLC, los cuales los describen al momento de generar una derivación:

- Un conjunto finito de símbolos que forma una cadena de un lenguaje, es denominado *alfabeto terminal* o *alfabeto de símbolos terminales*. Por ejemplo, el conjunto $\{0,1\}$.
- Un conjunto finito de *variables*, es también denominado *símbolos no-terminales*.
- Una *variable*, representa el lenguaje que se está definiendo, es decir, se denomina *símbolo inicial*. Generalmente el símbolo inicial esta denominado por S, pero puede ser cualquier símbolo a elegir.
- Un conjunto finito de *producciones* o *reglas de producción*, representan la definición recursiva de un lenguaje. Cada producción consta de lo siguiente:
    - Una *variable*, la cual define parcialmente *cabeza* de la producción.
    - Un símbolo de *producción* →.
    - Una cadena formada por *símbolos terminales* y *no-terminales*. Esta cadena se denomina *cuerpo de producción.*

El **Ejemplo 6**, muestra las diversas características mencionadas anteriormente.

$$E \rightarrow I$$

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$I \rightarrow a$$

$$I \rightarrow b$$

$$I \rightarrow Ia$$

$$I \rightarrow Ib$$

$$I \rightarrow I0$$

$$I \rightarrow I1$$

Ejemplo 6. Gramática Libre de Contexto

Ejemplo 7. Considerando las *reglas de producción*, mostradas en el **Ejemplo 6**, demostrar la derivación de la cadena $a * (a + b00)$.

Una vez que se ha analizado la cadena, es necesario verificar cual, de todas las *producciones*, se acomoda más a lo solicitado, como es de apreciarse, la *producción* numero 3 es la que más se acomoda. Por lo que quedaría de la siguiente forma:

$$E \rightarrow E * E$$

Siendo así el primer paso para poder continuar y lograr obtener el resultado deseado. Como es de apreciarse, la variable *no-terminal* E genera $E * E$ el cual se acomoda, pero aun no nos muestra el resultado deseado. Posterior a este *primer movimiento*, consiste en sustituir en cada *no-terminal*, las debidas *producciones* pertinentes en nuestra gramática. La siguiente *producción* para la variable E postrada en el lado izquierdo es la visualizada en la 1º posición de nuestra *tabla de producciones*. Para el lado derecho, es funcional la *producción* presente en la 4º posición.

Con lo mencionado anteriormente se obtiene lo siguiente:

$$E \rightarrow I * (E)$$

De esta forma, se continúa sustituyendo las variables *no-terminales* con sus debidas *producciones* en sus equivalencias. Quedando de esta forma la derivación:

$$E \rightarrow a * (E + E)$$

$$E \rightarrow a * (I + E)$$

$$E \rightarrow a * (a + I)$$

$$E \rightarrow a * (a + I0)$$

$$E \rightarrow a * (a + I00)$$

$$E \rightarrow a * (a + b00)$$

3.2.2.1. DERIVACIONES IZQUIERDA Y DERECHA

Existen otros tipos de derivaciones, además del mostrado anteriormente. Este tipo de derivaciones, son nombradas: *derivación a la izquierda* y *derivación a la derecha*.

El objetivo principal de este tipo de derivaciones es restringir el gran número de opciones disponibles al generar la derivación de un determinado lenguaje. A menudo, es muy útil el permitir que solo se reemplacen aquellas *variables* que se encuentran más a la izquierda o derecha de las *producciones*.

Ejemplo 8. Demostrar la *derivación a la izquierda*, haciendo uso de la cadena mostrada en el **Ejemplo 7**.

Como bien se observó con anterioridad, el comienzo de las derivaciones de la cadena $a * (a + b00)$, se da en base a la *producción* número 3, mostrada en el **Ejemplo 6**. Los pasos a seguir son exactamente los mismos que se mostraron en el **Ejemplo 7**, sustituyendo cada *producción* con su equivalente hasta obtener la cadena de terminales requerida, con la excepción de que no se comienza al “azar” dentro de las *producciones* o sustituyendo aquella que mejor se acople. Para este tipo de derivaciones, y como bien lo dice su nombre, es necesario comenzar a

sustituir aquellos *no-terminales* que se localicen más a la izquierda de nuestra *producción*. Por lo tanto, se obtiene las siguientes *producciones*.

$$E \rightarrow E * E$$
$$E \rightarrow I * E$$
$$E \rightarrow I * (E)$$
$$E \rightarrow a * (E + E)$$
$$E \rightarrow a * (I + E)$$
$$E \rightarrow a * (a + I)$$
$$E \rightarrow a * (a + I0)$$
$$E \rightarrow a * (a + I00)$$
$$E \rightarrow a * (a + b00)$$

Ejemplo 9. Demostrar la *derivación a la derecha*, haciendo uso de la cadena mostrada en el **Ejemplo 7**.

Tal y como se ha mencionado con anterioridad, este tipo de derivación consta de tomar aquellos *no-terminales* que se encuentren más a la derecha de su *producción*. Por lo tanto, es posible obtener las siguientes *producciones*:

$$E \rightarrow E * E$$
$$E \rightarrow E * (E)$$
$$E \rightarrow E * (E + E)$$
$$E \rightarrow E * (E + I)$$
$$E \rightarrow E * (E + I0)$$
$$E \rightarrow E * (E + I00)$$
$$E \rightarrow E * (I + b00)$$
$$E \rightarrow I * (a + b00)$$
$$E \rightarrow a * (a + b00)$$

Como es de apreciarse, en ambos casos no existe diferencia alguna de manera momentánea, pero es de aclarar, que en diversos lenguajes el hacer uso de alguno

de estos tipos de derivaciones, suele darnos un camino más corto a la cadena requerida y viceversa.

3.2.3. ARBOLES DE DERIVACIÓN

Existe otro tipo de mostrar las derivaciones y donde es posible captar con mayor facilidad la parte de sustitución dentro de las *producciones*, estos son los *arboles de derivación o arboles sintácticos*. Poseen la misma funcionalidad que una derivación normal, pero con la característica de que son mostradas de una manera gráfica más “amigable” y en ocasiones, más fácil de comprender.

Una de las características presentes en los arboles de derivación, es que muestra claramente cómo se van agrupando los símbolos de una cadena *terminal*, en diversas subcadenas a lo largo de su *producción*. Una de las partes más importantes de los árboles de derivación, es notable dentro de los compiladores ya que es la estructura de datos que representa el programa fuente facilitando la traducción del programa fuente, a código ejecutable, permitiendo de esta manera que la “traducción” se lleve a cabo de manera más natural en funciones recursivas. (E. Hopcroft, Motwani, & D. Ullman, 2007)

Un árbol de derivaciones, se construye de la siguiente forma:

- Cada nodo es constituido por una *variable no-terminal* de nuestras *producciones*.
- Cada nodo puede poseer solo un nodo padre.
- Existe solo un nodo raíz, el cual no tiene nodo padre. Se encuentra en la parte superior del árbol, los nodos que se desprenden de este nodo, son denominados *hojas*.
- Los nodos que finalizan en una variable de tipo *terminal*, son la parte final de la derivación.

Ejemplo 10. En base a las siguientes producciones, obtener el *árbol de derivación* para la expresión $(id + id) * id$.

$$< expresion > \rightarrow < expresion > \ + < expresion >$$
$$< expresion > \rightarrow < expresion > * < expresion >$$
$$< expresion > \rightarrow (< expresion >)$$
$$< expresion > \rightarrow id$$

Un *árbol de derivación*, se genera *casi* de la misma forma que una derivación normal, es por esta razón que comenzaremos visualizando la derivación para poder ir comprendiendo como se va a ir generando nuestro árbol. En primera instancia, se obtiene el siguiente resultado mostrado en la

$$< expresion > \rightarrow < expresion > * < expresion >$$
$$< expresion > \rightarrow (< expresion >) * < expresion >$$
$$< expresion > \rightarrow (< expresion >) * id$$
$$< expresion > \rightarrow (< expresion > \ + < expresion >) * id$$
$$< expresion > \rightarrow (< expresion > \ + id) * id$$
$$< expresion > \rightarrow (id + id\ ) * id$$

Figura 23. Derivación resultante del **Ejemplo 10**.

Como es de apreciarse, se obtiene una derivación de forma rápida y sencilla, pero ahora, en base a un *árbol de derivaciones*, es necesario identificar primordialmente cual será nuestro nodo raíz, de esta forma es posible apreciar que la variable denominada $< expresion >$ partirá como nodo principal o raíz. Posteriormente, seguiremos armando el *árbol de derivaciones*, donde al tener el nodo raíz, se desprende la primera *derivación* que resulta ser:

$$< expresion > \rightarrow < expresion > * < expresion >$$

Donde cada uno de los elementos del lado derecho o *producción*, es un nuevo nodo hijo, del cual se desprenderán nuevos nodos hasta llegar a un elemento *terminal*. En la **Figura 24**, es posible apreciar lo comentado con anterioridad, donde el nodo

raíz, es $< expresion >$ y cada nodo hijo, es cada parte que constituye a la *derivación* obtenida de este primer “paso”.

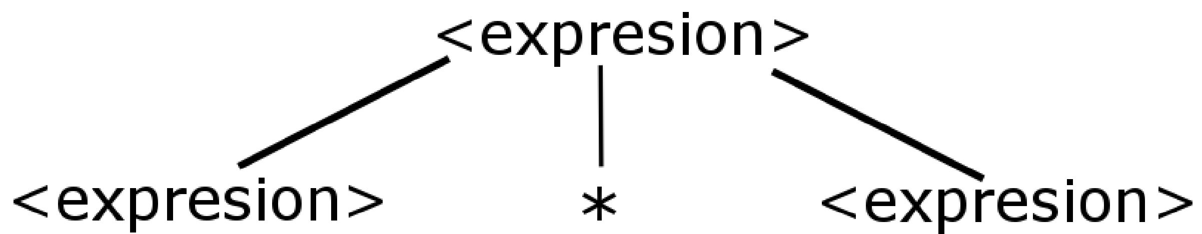


Figura 24. Primera *derivación*, de la **Figura 23**, mostrada en un *árbol de derivación*.

De esta forma, y por cada nodo hijo *no-terminal*, se realiza la misma *derivación* a lo largo del árbol hasta llegar al resultado requerido. Cabe hacer notar, que al igual que en las *derivaciones* vistas con anterioridad, existen también dentro de los *árboles*, ya sea con una *derivación hacia la derecha* o con una *derivación hacia la izquierda*. De manera práctica y teórica, se aplica el mismo concepto, donde la *derivación por izquierda* implica que los nodos para llegar al resultado, sean derivados en base a que se encuentren más apegados al lado izquierdo como su nombre lo indica, y viceversa para la *derivación por derecha*.

En la **Figura 25**, es posible apreciar el resultado de derivar la expresión:

$$(id + id) * id$$

En base a un *árbol*, también es posible apreciar que su *derivación*, es más apegada a la izquierda.

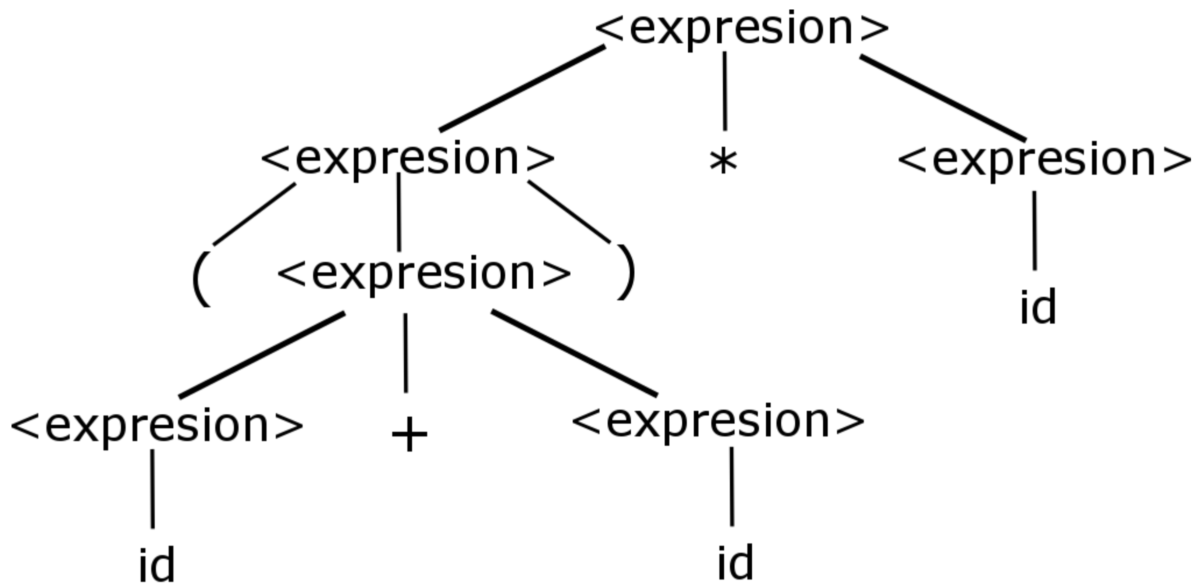


Figura 25. *Árbol de derivación* resultante para el **Ejemplo 10**.

3.2.3.1. AMBIGÜEDAD EN LAS GRAMÁTICAS

Una gramática, se dice que es ambigua cuando produce más de un *árbol sintáctico* para una determinada sentencia o expresión. Otra forma de decir que una gramática es ambigua, es cuando produce más de una *derivación* ya sea a la izquierda o derecha. (Sanchez Dueñas & Valverder Andreu, 1988)

Ejemplo 11. Se tiene la siguiente gramática:

$$E \to E + E$$

$$E \to E * E$$

$$E \to x$$

$$E \to y$$

Con esta gramática, representar su *derivación* y su *árbol de derivación* de la expresión $x + y * x$.

Para dar solución a esta expresión, existen dos caminos.

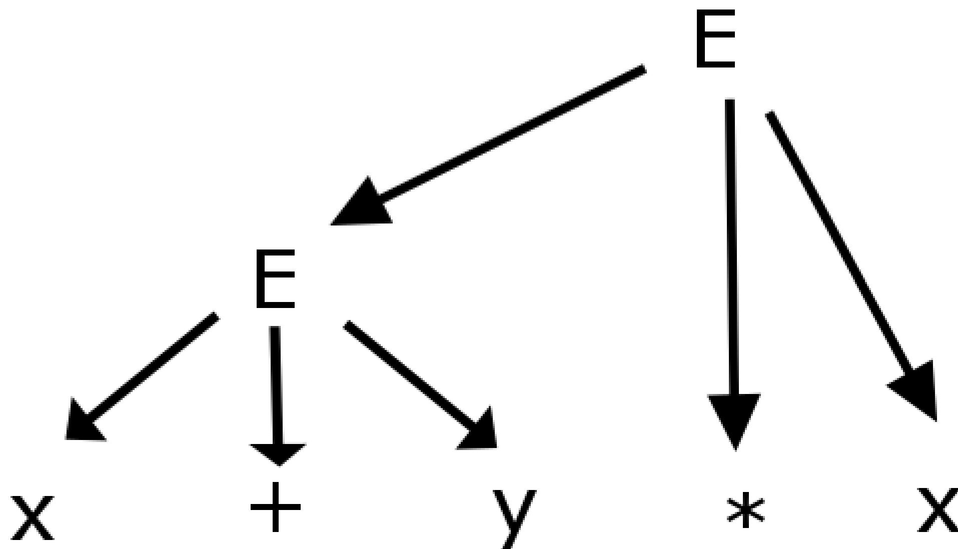


Figura 26. Árbol con derivación más a la izquierda.

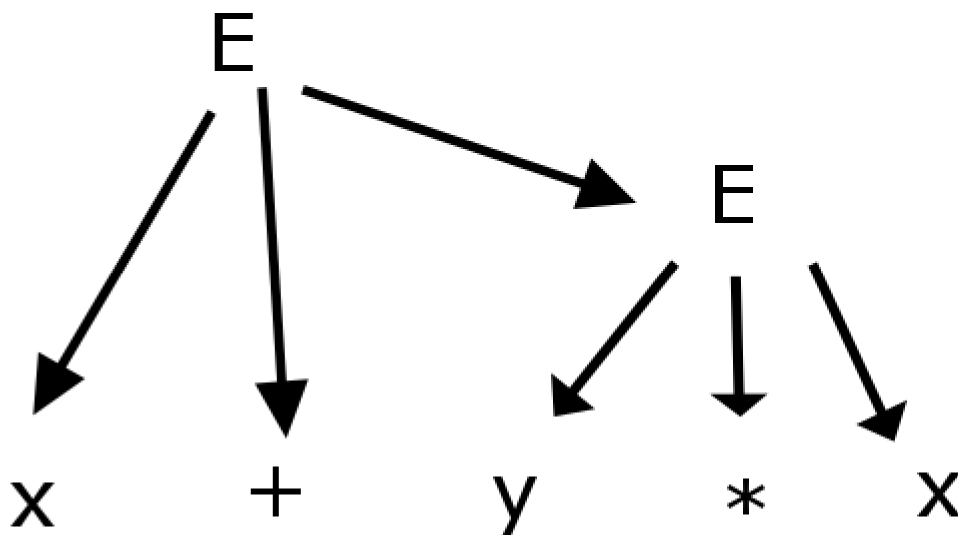


Figura 27. Árbol con derivación más a la derecha.

En este ejemplo, el hecho de la existencia de dos *árboles de derivación* para una misma expresión es indeseable, ya que cada *árbol* indica una forma distinta de estructurar y representar la expresión. Esta forma de estructurar y representar la expresión $x + y * x$, no solo afecta la forma en como es visualizada por el analizador sintáctico, también tiene su repercusión dentro del resultado de la misma expresión, ya que en el *árbol* que se muestra en la **Figura 26**, el resultado obtenido en la suma de $x + y$, es multiplicado posteriormente por x. Mientras que en el *árbol* que se visualiza en la **Figura 27**, primero se ejecuta la multiplicación $y * x$ y posteriormente

la adición de la segunda x. Por lo tanto, el significado que se da a ambas expresiones, difiere, provocando resultados no deseados.

Una de las formas más utilizadas para eliminar la ambigüedad, consiste en introducir *no-terminales* para la eliminación de los *árboles de derivación* no deseados. De esta forma, y utilizando las *producciones* mostradas en el **Ejemplo 11**, se obtiene lo siguiente:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow x$$

$$F \rightarrow y$$

Con esta nueva gramática, el *árbol* mostrado en la **Figura 26** se elimina, quedando solamente el *árbol* de la **Figura 27**, eliminando en conjunto la ambigüedad. De esta forma, el nuevo árbol queda tal y como se muestra en la **Figura 28**.

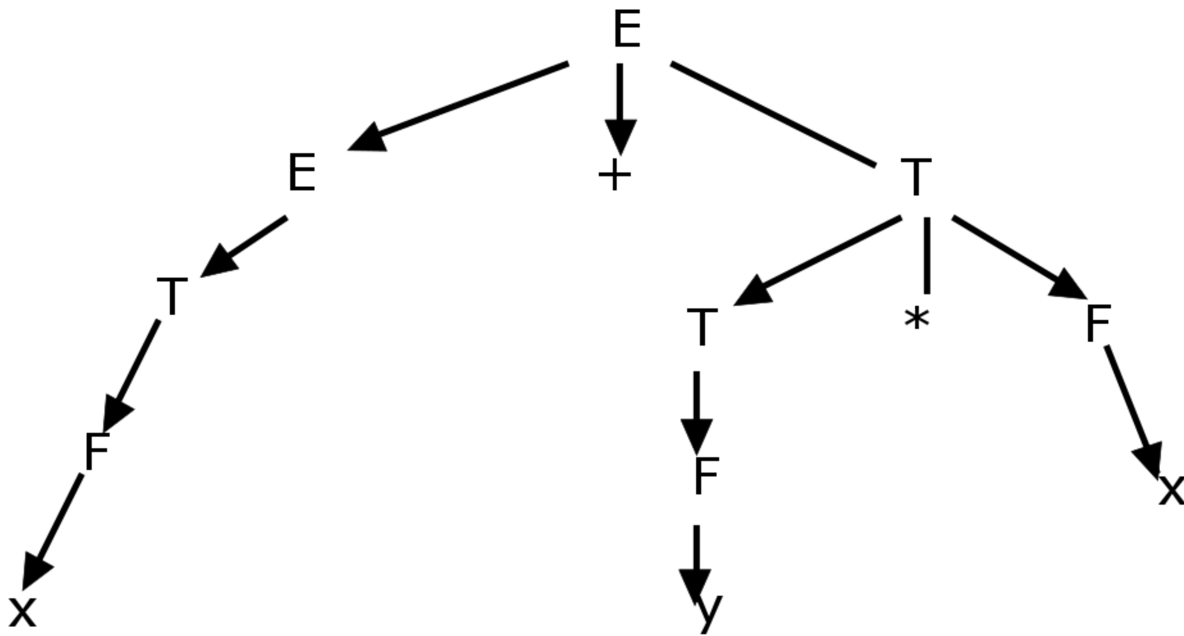


Figura 28. Árbol generado por las nuevas reglas de derivación.

3.2.4. NOTACIONES DE LOS LENGUAJES LIBRES DE CONTEXTO

Dentro de las gramáticas libres de contexto, existe un tipo de notación para poder *representar* la sintaxis de un lenguaje, esta notación, es un *metalenguaje*, ya que se utiliza para representar otro lenguaje de manera compacta y precisa, basándose en constructores sintácticos de símbolos y reglas (Zeugmann). Una de las notaciones más utilizadas, es el *Backus-Naur-Form*, también conocido como *Panini-Backus Form* o *Backus Normal Form* (BNF), de igual forma existe otra alternativa que se deriva del BNF que es el *Extended-BNF* (EBNF) los cuales, como ya se mencionó anteriormente, son formas matemáticas para poder expresar Gramáticas Libres de Contexto y al ser un *metalenguaje*, generan un *estándar* o *normalización* de cómo se comenzaron a representar los lenguajes de programación, esto por la precisión con la cual logran describir el objetivo de lo representado en el programa, evitando la ambigüedad que anteriormente se tenía al programar, permitiendo que cualquiera pudiera comprender lo que se intentaba decir.

Dentro de la representación BNF, existen diversos símbolos que dan significado a equivalencias dentro de las Gramáticas Libres de Contexto:

- Los *no-terminales*, son representados entre "<*no-terminal*>".
- Los *terminales*, son representados sin los símbolos "<>".
- "::=" o ":=" representan las definiciones de la *producción* en lugar del símbolo →, también es pronunciado como "se define como".
- "|" define un simple "OR".

Como es posible observar, existe una relación bastante apegada a las Gramáticas Libres de Contexto, refiriéndonos a su representación dentro de las producciones, las cuales posteriormente fueron una base fundamental para generar los lenguajes de programación. Pero posteriormente la necesidad de poder expresar mayores descripciones y funcionalidades, el BNF no fue suficiente, por ello, fue necesario el EBNF, dentro del cual se siguió la misma convención anterior, pero agregando las nuevas soluciones a las necesidades

que se tuviesen. Con esta nueva función, se logró dar también un gran paso para hacer más riguroso y consolidar los lenguajes de programación que se utilizan hoy en día.

Dentro del EBNF, se encuentran las siguientes nuevas reglas de sintaxis:

- "*" (Clausula Kleene) representa de 0 a más ocurrencias de la sentencia donde se utilice.
- "+" (Potencia o Kleene Cross) representa 1 o más ocurrencias de la sentencia donde sea aplicada.
- "?" representa 0 o 1 ocurrencias, en algunos casos llega a ser representado también por corchetes "[…]".
- Los paréntesis "(…)" representan agrupaciones.

Ejemplo 12. Supóngase que se tienen las siguientes *producciones*, representarlas en la notación BNF.

$$O \rightarrow FN\ FV$$

$$FV\ \ \rightarrow VERBO\ FN \mid VERBO$$

$$FN \rightarrow SUST \mid ART\ SUST \mid SUST\ ADJ \mid ART\ SUST\ ADJ$$

$$SUST \rightarrow cosa \mid arbol \mid nombre$$

$$ADJ \rightarrow azul \mid grande$$

$$VERBO \rightarrow tiene \mid estudia$$

$$ART \rightarrow la \mid los \mid el \mid un$$

Dentro de las notaciones BNF, quedaría de la siguiente manera:

$$< oracion > \coloneqq < fn >< fv >$$

$$< fn > \coloneqq < verbo >< fn > \mid < verbo >$$

$$< fn > \coloneqq < sust > \mid < art >< sust > \mid < sust >< adj > \mid < art >< sust >< adj >$$

$$< sust > \coloneqq cosa \mid arbol \mid nombre$$

$$< adj > \coloneqq azul \mid grande$$

$$< verbo > \coloneqq tiene \mid estudia$$

$$< art > \coloneqq la \mid los \mid el \mid un$$

Como es posible observar dentro del **Ejemplo 12**, la diferencia existente entre las Gramáticas Libres de Contexto y las notaciones BNF, no es grande, solo se definió una *normalización* de símbolos y conjeturas para expresar las *producciones*.

La gran diferencia, recae entre las notaciones BNF y EBNF, en el **Ejemplo 13** es posible observar esta gran diferencia.

Ejemplo 13. Representar la gramática de los números decimales en las notaciones BNF y EBNF.

En BNF:

$$< expr > ≔ '-' < num > \mid < num >$$
$$< num > ≔ < digito > \mid < digito > '.' < digito >$$
$$< digito > ≔ < digito > \mid < digito > < digito >$$
$$< digito > ≔ '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'$$

En EBNF:

$$< expr > ≔ '-' ? < digito > + ('.' < digito > +)?$$
$$< digito > ≔ '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'$$

Como es posible observar, el EBNF en comparación al BNF, es más conciso en la definición de la gramática. Su lectura es simple, al igual que su implementación, permitiendo que sea de mayor utilidad dentro de los lenguajes de programación. Por ello, las Gramáticas Libres de Contexto, con ayuda de estas notaciones, son la base primordial de los lenguajes de programación de alto nivel.

3.2.5. AUTOMATAS TIPO PILA

Un autómata tipo pila, es una 6-tupla, de la forma $AP = (Q, \Sigma, \Gamma, \Delta, q_0, F)$ que se define cada elemento como:

- Q es el conjunto finito de estados.
- Σ es el alfabeto finito de entrada de la cadena.
- Γ es el alfabeto que pertenece a la pila.
- Δ es la función de transición.
- q_0 es el estado inicial.
- F es el conjunto de estados finales.

La función de transición, se define tal y como se muestra en la **Figura 29**.

$$\Delta = (Q \times \Sigma^* \times \Gamma^*) \times (Q \times \Gamma^*)$$

Estado Actual
Leer de cadena
Elemento a sacar de la pila
Nuevo estado
Elemento a introducir en la pila

Figura 29. Definición de la función de transición en los autómatas tipo pila.

Los autómatas de tipo pila, son básicamente un autómata finito con control sobre los elementos de entrada y sobre el *stack* que posee una capacidad infinita, de igual forma, los lenguajes generados por las Gramáticas Libres de Contexto, son aceptados en los autómatas tipo pila. (Campos, 1995)

Los autómatas tipo pila o AP, son autómatas finitos no deterministas con transición ε. El hecho de tener una *pila* dentro de un autómata, es dar la capacidad de "recordar" toda la información que pasa a través del mismo autómata. Una de las desventajas presentes en los AP, recae en la misma capacidad que tiene de almacenar infinita información dentro de *stack*, ya que solo puede acceder a la información disponible en su pila de acurdo a la forma en como la pila la manipula. Al ser una pila del tipo FIFO (first - in, first - out), solo podrá manipular el último elemento que accedió al AP. (Vazquez Palma)

De manera informal, es posible interpretar un AP como un dispositivo, tal y como se muestra en la **Figura 30**. En esta imagen, se logra leer claramente la funcionalidad de un simple autómata finito, donde se tiene una entrada de datos, un control de entrada de datos, el cual lee el dato que accedió y valida si es aceptado o no dentro de la máquina. Dentro de los AP, lleva una máquina de estados apilada, la cual define la próxima transición en base al estado actual. (E. Hopcroft, Motwani, & D. Ullman, 2007)

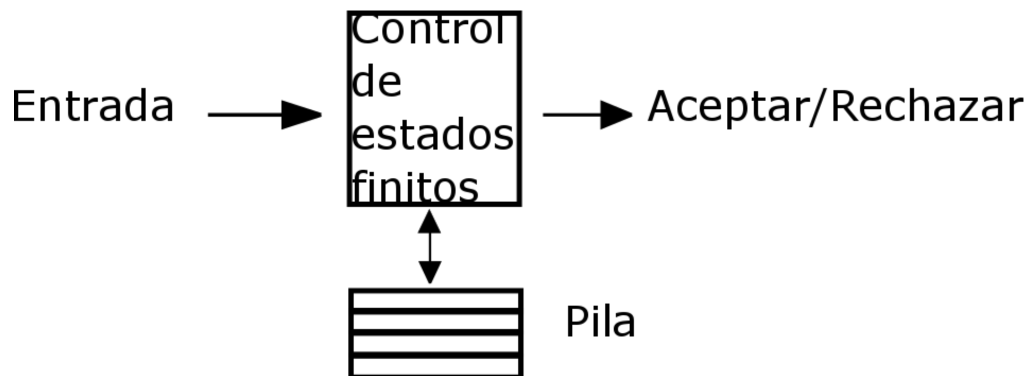


Figura 30. Autómata tipo pila, es un autómata finito con una estructura de datos de pila.

En los AP, se tiene el siguiente comportamiento:

- Consume el símbolo de entrada en la transición. Si se llega a utilizar ε, no se consume ningún símbolo.
- Pasa al nuevo estado, el cual puede o no puede ser el mismo que el estado anterior.
- Reemplaza el símbolo de la parte superior de la pila con cualquier cadena. La cadena puede ser ε, lo que corresponde a una extracción de la pila. En caso de no realizar ningún cambio en la pila, se conserva el mismo símbolo que estaba anteriormente. Es posible reemplazar el símbolo de la cima de la pila, lo cual no extrae ningún otro símbolo, pero cambia el estado.

Los AP, al igual que los autómatas finitos, poseen una notación gráfica, donde en cada nodo existe una transición del tipo *"ω/α/β".*

Donde:

- ω es la entrada de caracteres que consume el autómata.
- α es lo que va a salir de la pila.
- β es lo que va a entrar a la pila.

La **Figura 31**, muestra un ejemplo en las diversas variaciones que existen en los tipos de transición de cada autómata y como es que varían en los que son del tipo pila.

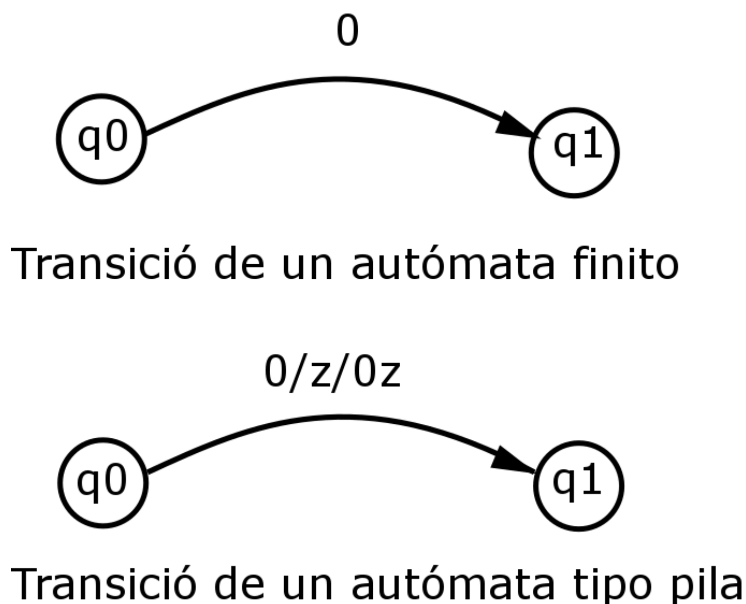


Figura 31. Diferencia en las transiciones de un Autómata Finito y un Autómata Pila.

Los AP, al igual que los autómatas finitos, poseen estados finales, los cuales permiten distinguir que la palabra de entrada, fue aceptada correctamente. Para que una palabra de entrada, sea aceptada, debe cumplir las siguientes condiciones:

- La palabra debe haber sido consumida en su totalidad.
- El AP, debe encontrarse en un estado final.
- La pila debe estar vacía.

Ejemplo 14. Se tiene el lenguaje $L = \{0^n 1^n | n \geq 1\}$

La primera parte importante a notar, es identificar que, al intentar construir un autómata finito con este lenguaje, es algo casi imposible, ya que los autómatas finitos, no cuentan con la capacidad para contar el número de elementos que son insertados. Por ello, es necesario hacer uso de un AP. Una vez aclarado, es dar a notar el grafo resultante de AP, el cual se muestra en la **Figura 32**. El autómata que se visualiza, no es la solución final, ya que hace falta definir las entradas y salidas de la pila.

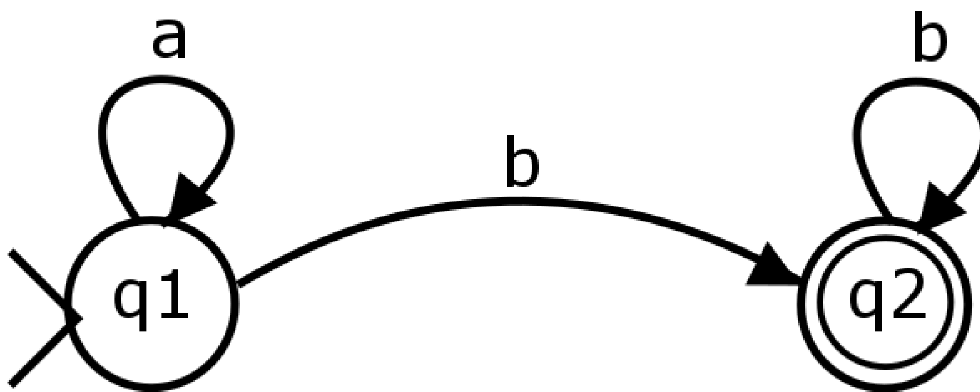


Figura 32. Primer esquema del autómata para el **Ejemplo 14**.

Una vez que se tiene este *esquema*, es necesario tener en cuenta cuáles serán los elementos que entrarán al autómata, que se espera que exista en la pila y cuál sería el resultado final en la pila. Como es posible apreciar, se tienen los estados q_0 y q_1, donde q_1 además de ser el estado final, también es el estado de aceptación de la cadena.

Con este esquema de los AP, es necesario tener en cuenta la pila en todo momento, para ello es necesario mantener un identificador el cual nos del estado actual de la pila, cuando esta se encuentre vacía. Este *identificador*, lo definiremos como ε (vacío). De esta forma, es posible comenzar con la construcción del autómata.

Para este tipo de autómatas, su función va en base a la pila, tal y como se muestra en la **Figura 30**, por ello y en base al ejemplo, es necesario verificar y validar el

estado de la pila para poder hacer uso de la siguiente transición ya que estas dependen del estado actual de la pila.

Como el primer elemento en entrar a la pila es una a, se tiene en cuenta que la pila en ese momento está vacía, la **Figura 33**, muestra el comportamiento tras la entrada del primer elemento al AP.

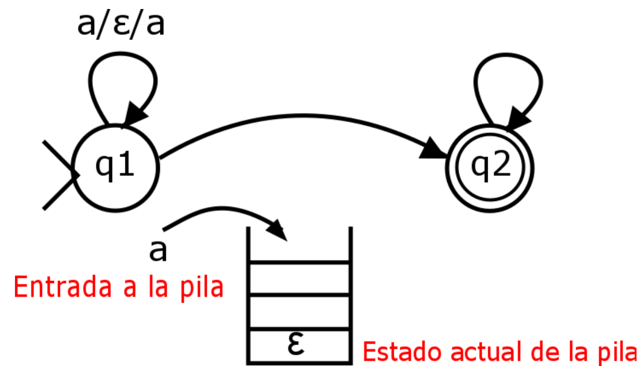


Figura 33. Reacción del autómata tipo pila, tras introducir el primer elemento.

Si se vuelve a introducir otra a, la pila quedaría tal y como se muestra en la **Figura 34**.

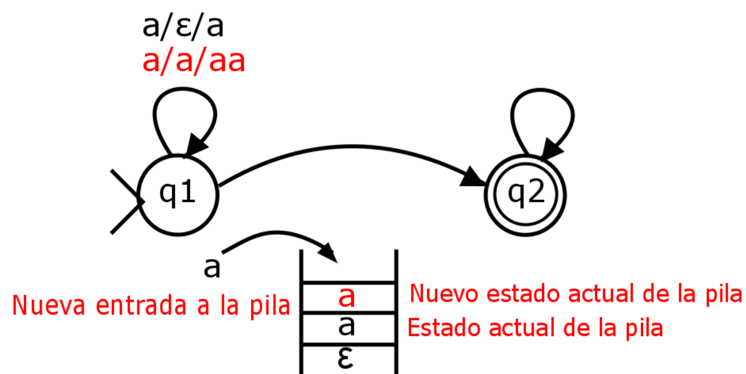


Figura 34. Movimientos realizados en la pila tras la introducción de otro elemento.

Como es de apreciarse, el estado de la pila comienza a cambiar según el número de elementos que se introduzcan a ella y sigue apilándolos, mientras más $a's$ sean introducidas. Pero ahora, si se introduce una b, el procedimiento cambia ya que al introducir una b, se debe verificar que en la pila exista algo más que un elemento vacío, y en base al ejemplo que estamos trabajando, se espera que exista una a

dentro de la pila. El procedimiento se muestra en la **Figura 35**, donde por cada entrada de una b, en la pila debe encontrarse una a la cual es eliminada de la pila y en su lugar se deja un estado vacío (ε).

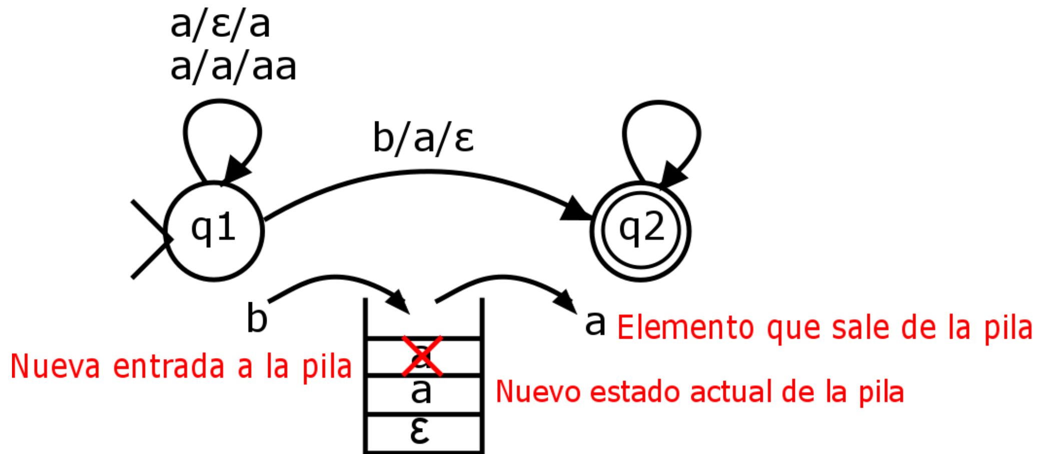


Figura 35. Al introducir un elemento b a la pila, una a se saca de la pila.

Con la transición que se mencionó anteriormente, ahora se encuentra el autómata en el estado q_2, el cual es el estado final y el estado de aceptación de la cadena. Como podremos notar, aun se cuenta con un elemento en la pila tras el ejercicio que se está llevando a cabo y como se mencionó con anterioridad, el estado de aceptación solo es válido hasta que se hayan consumido todos los elementos de la cadena y que la pila se encuentre vacía.

Si recordamos el lenguaje marca que deben ser el mismo número de entradas de una a y una b, por ello en el estado final, es necesario aplicar el mismo criterio que se ocupó en la transición anterior. De esta forma se obtiene el resultado mostrado en la **Figura 36**.

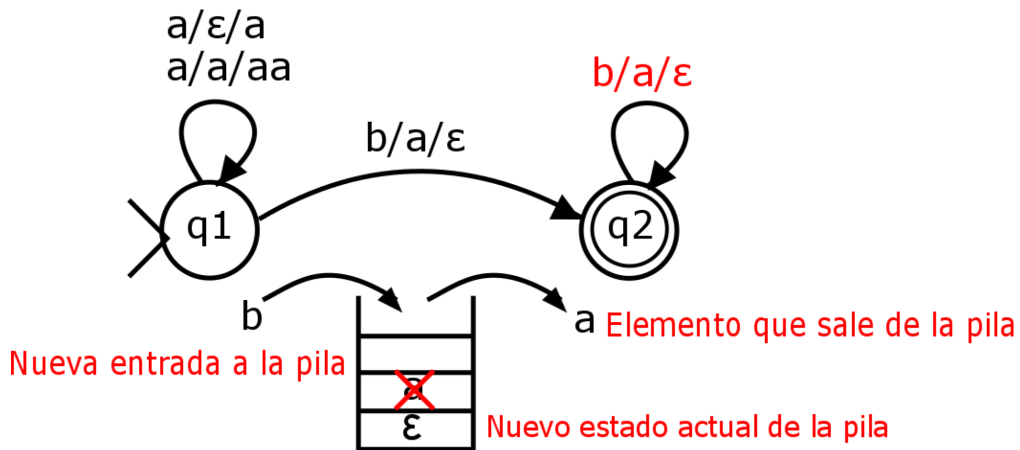


Figura 36. Resultado final de la pila y el autómata.

Ya logramos visualizar el funcionamiento de los AP, los cuales se encuentran ampliamente relacionados con los Lenguajes Libres de Contexto ya que los AP, aceptan exactamente a los LLC. La prueba de lo mencionado, se divide en dos partes:

- Si M es un AP, entonces L(M) es un LLC.
- Si L es un LLC, entonces hay un AP M tal que L(M) = L.

De esta forma procedemos a obtener la gramática del **Ejemplo 14**.

Ejemplo 15. Obtener la gramática y el AP con su definición formal, validando la entrada de la cadena “aaabbb”.

Como se pudo observar en el **Ejemplo 14**:

- $L = \{0^n 1^n | n \geq 1\}$
- $L(M)$ puede verse en la **Figura 32**.
- $L(G) = S \rightarrow aSb|ab$

La función de transiciones para $L(M)$, en base a la función definida en la **Figura 29**, donde se define aquel estado donde se encuentra el autómata, que carácter va a ser leído, el elemento a sacar de la pila, el nuevo estado al cual va a pasar y cuál va a será el nuevo elemento a introducir en la pila, seria:

$$(q_0, a, \varepsilon)(q_0, a)$$

$$(q_0, b, a)(q_1, \varepsilon)$$

$$(q_1, b, a)(q_1, \varepsilon)$$

De igual forma, esta función de transición para $L(G)$, quedaría de la siguiente forma, donde se introducen los elementos de la gramática.

Δ=

$$(q, \varepsilon, S)(q, aSb)$$

$$(q, \varepsilon, S)(q, ab)$$

$$(q, a, a)(q, \varepsilon)$$

$$(q, b, b)(q, \varepsilon)$$

De forma desglosada, por cada elemento, se obtiene de la siguiente forma para la función se transición:

Δ=

1	$(P, \varepsilon, \varepsilon)(q, S)$
2	$(q, a, \varepsilon)(q_a, \varepsilon)$
3	$(q_a, \varepsilon, a)(a, \varepsilon)$
4	$(q, b, \varepsilon)(q_b, \varepsilon)$
5	$(q_b, \varepsilon, b)(q, \varepsilon)$
6	$(q, \$, \varepsilon)(q_\$, \varepsilon)$
7	$(q_a, \varepsilon, S)(q_a, aSb)$
8	$(q_b, \varepsilon, S)(q_b, \varepsilon)$

Donde:

$P = aaabbb\$$

$\$$ es el fin de cadena, para definir la aceptación de la cadena.

En base a lo mencionado anteriormente, es posible obtener la siguiente tabla con el desglose por cada transición y cuál es la que se aplica a ese carácter.

	Estado actual	Leer de cadena	Sacar de la Pila	Nuevo estado	Introducir a la pila	Cadena	Pila
1	P	ε	ε	q	S	aaabbb$	ε
2	q	a	ε	q_a	ε	aaabbb$	S
7	q_a	ε	S	q_a	aSb	aabbb$	S
3	q_a	ε	a	q	ε	aabbb$	aSb
2	q	a	ε	q_a	ε	aabbb$	Sb
7	q_a	ε	S	q_a	aSb	abbb$	Sb
3	q_a	ε	a	q	ε	abbb$	aSbb
2	q	a	ε	q_a	ε	abbb$	Sbb
7	q_a	ε	S	q_a	aSb	bbb$	Sbb
3	q_a	ε	a	q	ε	bbb$	aSbbb
4	q	b	ε	q_b	ε	bbb$	Sbbb
8	q_b	ε	S	q_b	ε	bb$	Sbbb
5	q_b	ε	b	q	ε	bb$	bbb
4	q	b	ε	q_b	ε	bb$	bb
5	q_b	ε	b	q	ε	b$	bb
4	q	b	ε	q_b	ε	b$	b
5	q_b	ε	b	q	ε	$	b
6	q	$	ε	$q_\$$	ε	$	ε
	$q_\$$					ε	ε

Como se logró apreciar, y se ha ido analizando, los autómatas finitos permiten describir y analizar la parte lexicográfica, verifican que cada cadena que sea introducida, se encuentre *bien escrita*, mientras que los autómatas tipo pila, y en relación a las gramáticas y los lenguajes libres de contexto, definen la parte sintáctica, donde se valida el proceso que llevara la máquina virtual según lo que se introduzca en ella y hacia qué camino llevar el resultado.

4. CAPÍTULO – JFLEX Y CUP

4.1. ANALIZADOR LEXICOGRÁFICO

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, la función `while`, identificadores, ya sea cualquier letra seguida de cero o más letras y/o dígitos, y signos como + o =, por hacer mención de algunos. La fase de rastreo, lectura o *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 (AF).

4.1.1. FUNCIÓN DEL ANALIZADOR LEXICO

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 o *tokens*. Los componentes léxicos representan:

- Palabras reservadas. Por ejemplo, if, while, do.
- Identificadores. Como lo son las variables, funciones, tipos definidos por el usuario, etiquetas, por hacer mención del algunas.
- Operadores. Tales como son, =, >, <, >=, <=, +, *.
- Símbolos especiales. Por ejemplo, ( ), { }.
- Constantes numéricas. literales que representan valores enteros o 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. El análisis lexicográfico, suele implementarse como una subrutina del analizador sintáctico, cuando recibe la orden "obtén el siguiente componente léxico", por ejemplo, el analizador léxico lee los caracteres de entrada hasta identificar el siguiente componente léxico. La **Figura 37** muestra gráficamente, lo mencionado anteriormente.

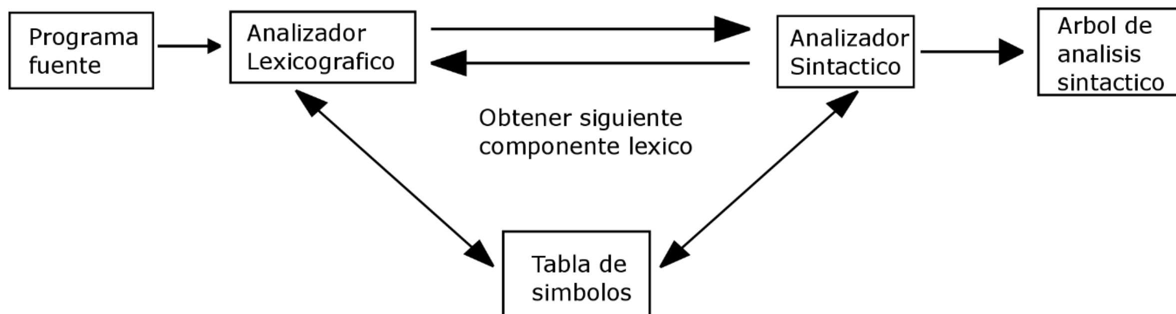


Figura 37. Comportamiento del analizador lexicográfico y en analizador sintáctico.

Además, el analizador lexicográfico, también es responsable de las siguientes funcionalidades:

- Manejo de apertura y cierre de archivos.
- 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.

4.2. ¿QUÉ ES JFLEX?

JFLex es un metacompilador que permite la rápida generación de analizadores lexicográficos, desarrollado en Java, para Java, el cual posee una tecnología basada en autómatas finitos deterministas (ADF), esta característica le permite una

gran fluidez y velocidad de lectura, en base a ficheros de texto que contienen las debidas expresiones regulares que definirán al lenguaje. (Vega Castro, 2008)

JFLex, fue diseñado principalmente para el procesamiento léxico, basado en un conjunto de reglas léxicas y con base en la herramienta JLex de Elliot Berk de la Universidad de Princeton, ya que JFLex es una “reescritura” de JLex y en conjunto se integró la herramienta Flex, la cual está desarrollada en C/C++ (Gerwin, 2004). De esta forma, permite la fluidez que posee y su fácil uso como analizador lexicográfico permitiendo la generación de los analizadores léxicos, en base a una “descripción” del lenguaje, tal y como lo son las Expresiones Regulares y los convierte en *tokens* comprensibles para el analizador. (Daviana)

JFLex está diseñado para trabajar en conjunto con el analizador sintáctico de tipo LALR (Look – Ahead Left to Right parser) como CUP de Scott Hudson y la adaptación de YACC de Berkeley, denominada BYACC/J por Bob Jamison. (Gerwin, 2004)

Algunas de las características más sobresalientes de JFLex son:

- Generación rápida de analizadores léxicos.
- Sintaxis cómoda de manipular y fácil de interpretar.
- Independencia de plataforma, ya que esta generado para ser integrado con Java.
- Fácil integración con CUP.
- Es OpenSource.

4.2.1. FUNCIONES Y ESTRUCTURA DE JFLEX

JFLex, necesita un archivo de configuración, en donde se especifique la descripción de la estructura de los *tokens* o expresiones regulares, para obtener la validación de los *tokens* en el analizador léxico. Las especificaciones o reglas, son escritas en un archivo con extensión “*.flex”, el cual lee JFLex e integra con una clase “Scanner.java” la cual permite analizar los *tokens* enviados y producir un programa llamado Yylex que reconoce las cadenas que cumplan las reglas establecidas para posteriormente ejecutar las acciones asociadas. (JFlex, 2009)

Definido de otra forma, JFLex obtiene un archivo denominado “Reglas.flex” el cual contiene las expresiones a cumplir, posteriormente genera un archivo llamado *Yylex.java* el cual posee la implementación de las reglas (también denominadas Autómata Finito Determinista) y finalmente ejecuta las acciones asociadas a cada regla. La **Figura 38**, define en forma gráfica su funcionamiento.

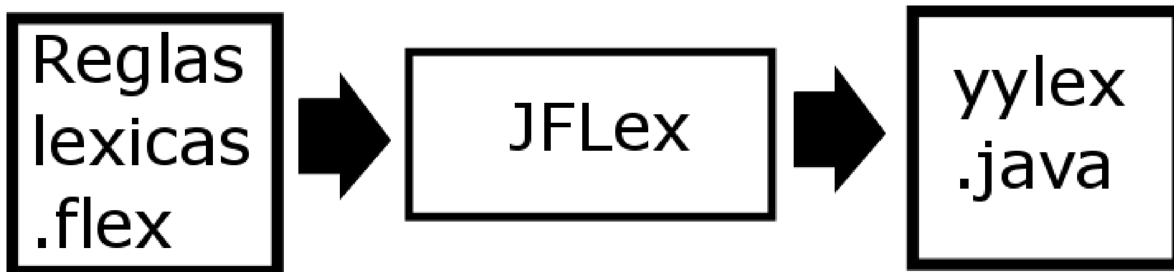


Figura 38. Funcionamiento de JFLex.

La velocidad de lectura, suele ser proporcional a la cantidad de caracteres con los que se alimente a JFLex, mientras que la complejidad, depende del número de reglas o producciones que se tengan por parte del autómata.

JFLex, maneja dentro de los archivos “*.flex” una estructura, la cual se divide en 3 secciones que se listan a continuación:

- Código del usuario.
- Opciones y declaraciones.
- Reglas lexicográficas.

Cada sección es separada por los símbolos “%%” y posee el siguiente formato, mostrado en la **Figura 39**.

```
1  {Codigo de usuario}
2
3  %%
4
5  {Opciones y declaraciones}
6
7  %%
8
9  {Reglas lexicograficas}
```

Figura 39. Estructura del archivo de configuración de JFLex.

4.2.1.1. FORMATO DEL FICHERO DE ENTRADA

Como se mencionó, JFLex necesita un archivo de entrada que contenga los debidos parámetros de configuración y directivas que le ayuden a ejecutar su funcionamiento de forma eficaz o en algunos casos, ejercer acciones específicas para el tipo de datos de entrada. Este archivo es dividido en tres secciones, cada sección lleva a cabo las siguientes funciones:

- **Código de Usuario**: Es código auxiliar, el cual permite añadir a JFLex librerías, importar alguna función propia del usuario, asignar paquetes, entre otras funciones, por ejemplo, si es necesario hacer uso de alguna función de la librería `util` de Java, se agrega simplemente la importación del paquete.

```
import java.util.*;
```

  Este código, es copiado tal cual, dentro del archivo fuente del generador lexicográfico y colocado hasta el principio para posteriormente declarar la clase “scanner”.

- **Opciones y declaraciones**: Dentro de esta sección, se da inicio la definición del bloque para la configuración de las directivas del analizador, el cual contiene un conjunto de parámetros que indicaran el comportamiento del analizador. Cada parámetro debe ser especificado en cada línea y en forma

de lista comenzando con un símbolo de porcentaje (%). Algunas de las directivas utilizadas, son:

- %class *classname*: Notifica a JFLex que el archivo *.java, llevara el nombre *classname*.
- %unicode: Permite trabajar con archivos que posean este tipo de caracteres.
- %cup: Habilita la integración de JFLex con CUP.
- %line: Habilita el contador de líneas del analizador.
- %column: Habilita el contador de columnas del analizador.
- %implements *interface*: La clase generada, implementa la interface *interface*.
- %extends *classname*: La clase generada, es una subclase de la clase *classname*.
- %public: Hace a la clase generada, publica.
- %char: Habilita el contador de caracteres, el resultado se encuentra en *yychar*.
- %function *name*: Cambia el método del scanner, provocando que sea renombrado a *name*. Si esta directiva no es definida, el nombre por default del método será *yylex*.
- %{

  <code>

  %}

  El código Java, que sea integrado en una sección de este tipo, será copiado dentro de la clase generada. De esta forma, el usuario podrá definir sus propias variables y funciones dentro del scanner.
- %{

  <init>

  %}

  El código Java que sea ingresado en esta sección, será incluido directamente dentro del constructor de la clase.

- **Reglas lexicográficas**: Esta sección, forma parte importante dentro del funcionamiento del analizador ya que aquí se definirá el conjunto de expresiones regulares que se utilizaran durante el proceso de análisis. Las reglas, tienen la siguiente forma:

$$Regla = Expresión\ Regular + Accion$$

Donde la Acción, es un fragmento de código en Java que se ejecuta una vez que se cumpla con la expresión regular, cabe mencionar que, si existe más de una coincidencia respecto a las reglas definidas, JFLex, tomara aquella con la que tenga una mayor longitud en su coincidencia, en dado caso de coincidir también en tamaño, tomara la primera que encuentre según su definición. Una regla funcional, se define tal y como se aprecia en la **Figura 40**.

```
(a|b)* {System.out.println("*** match");}
```

Expresion Regular — Accion

Figura 40. Ejemplo de la implementación de una regla lexicográfica.

Al igual que las directivas, es posible generar macros, las cuales poseen la definición de alguna expresión regular y pueden ser nombradas al gusto del usuario para poder identificarlas fácilmente o para definir su funcionamiento. Las macros dentro de JFLex, tiene la siguiente estructura:

$$macroID = expresion\ regular$$

De esta forma, si es generada una macro:

$$whitespaces = [\backslash t \backslash n] +$$

$$letter = [a - zA - Z]$$

$$digit = [0 - 9]$$

$$number = (\{digit\}) +$$

Es posible utilizarlas posteriormente dentro de la sección *Reglas lexicográficas*, donde en se hace un llamado a la macro, nombrando su identificador entre corchetes $\{whitespaces\}$.

Los métodos y variables, van en conjunto con las macros y la sección de *Reglas lexicográficas* permitiendo obtener información de forma fácil, sin la necesidad de reprogramarlas, algunos de estos métodos y variables son:

- `int yychar`: Representa el número de caracteres procesados desde el inicio de lectura. Solo si se incluye la directiva %char.
- `int yyline`: Representa el número de líneas procesadas, desde el inicio de lectura. Solo si se incluye la directiva %line.
- `int yycolumn`: Representa el número de columnas procesadas, desde el inicio de lectura. Solo si se incluye la directiva %column.
- `String yytext()`: Devuelve la cadena que coincidió con la respectiva expresión regular.
- `int yylength()`: Devuelve el tamaño de la cadena que coincidió con la respectiva expresión regular.
- `int yystate()`: Devuelve el estado léxico actual del analizador.
- `void yybegin(int lexicalState)`: Cambia el estado léxico actual del analizador, por el nuevo estado especificado.
- `char yycharat(int pos)`: Devuelve el carácter que se encuentra la posición *pos* de la cadena que coincidió con la respectiva expresión regular.
- `void yyclose()`: Cierra el flujo de entrada de datos.

Adicionalmente a la definición de las reglas, es posible hacer uso de los estados léxicos. Los estados léxicos actúan como una condición de arranque, esto significa que, si el *scanner* se encuentra en el estado `<ESTADO>`, solo aquellas reglas que estén contenidas posterior a este estado, serán ejecutadas. El estado, generalmente es una expresión regular, en dado caso de no declarar ningún estado, JFLex toma de forma predeterminada el estado `<YYINITIAL>` para comenzar la exploración de coincidencias con las reglas establecidas. De igual forma, si al

declarar una regla, no está establecido un estado léxico como condición de inicio, serán aplicadas a todos los estados encontrados.

Un estado se define tal y como se muestra en la **Figura 41**.

```
<ESTADO> {
    Exp1 {accion1}
    Exp2 {accion2}
}
```

Figura 41. Definición del cuerpo de un estado léxico.

Como es posible observar, la configuración de JFLex, tiene un gran parecido a la estructura de los archivos de configuración de Flex/Lex, con la diferencia de que JFLex integra mayores directivas, macros, reglas con veracidad en configuración y poder manejar algunas reglas directamente en lenguaje Java lo cual permite un mayor alcance en funcionalidad y un gran poder por el camino que está llevando este lenguaje de programación a través del desarrollo de aplicaciones. Otra de las razones por la cual JFLex es amigable y una competencia aceptable de uso ante este par de generadores lexicográficos, es que permite tanto a los nuevos usuarios, como a los usuarios de Flex/Lex, integrarse de forma *amigable* a JFLex, por el parecido y existe entre ellos, además de que JFLex, está basado en las herramientas ya mencionadas sin la necesidad de entrometerse ampliamente a una nueva herramienta, ahorrando la ardua tarea de aprender el uso de una nueva herramientas, disminuyendo significativamente la curva de aprendizaje.

Existe otro generador de analizadores lexicográficos especializado para Java el cual también se encuentra basado en Lex, llamado JLex para Java, una de las diferencias más significativas, recae en que JFLex es más sofisticado y versátil en

comparación a JLex, además de que JFLex está basado en JLex, el tiempo de análisis y lectura es menor en JFLex por el soporte del tipo de analizadores que posee como soporte a diferencia de JLex que solo es soportado por Lex. Aunque ambas son buenas opciones al hacer uso de un metacompilador, JFLex va a la cabeza de la competencia por su conveniente sintaxis, por la implementación de los Autómatas Finitos Deterministas (AFD) facilitando la fluidez de lectura sin la necesidad de retornos y soportando más de 64,000 estados, el soporte de caracteres UNICODE ampliando la gama de lectura y generación de analizadores, además de la identificación automática del EOF (End Of File o Fin de Archivo) ya sea en sistemas Windows o basados en Unix.

4.2.2. APLICANDO JFLEX

Como bien es sabido, JFLex está basado en Flex que a su vez es un derivado de Lex los cuales comparten diversas similitudes respecto al archivo de configuración donde se puede evitar el trabajar directamente con autómatas para especificar la secuencia de aceptación y así sustituirlos por expresiones regulares para un manejo más eficiente. En el siguiente ejemplo se muestra la aceptación del siguiente lenguaje:

- Operaciones aritméticas entre números (+, -, *, /, permitir el símbolo como operación y asignación =).
- Números.
- Identificadores los cuales permitieran primero una letra y posteriormente una *n* cantidad de números en combinación de letras.

El IDE a utilizar será Netbeans, donde se creará un proyecto llamado "Ejemplo1JFLex" la **Figura 42** ejemplifica la generación del proyecto.

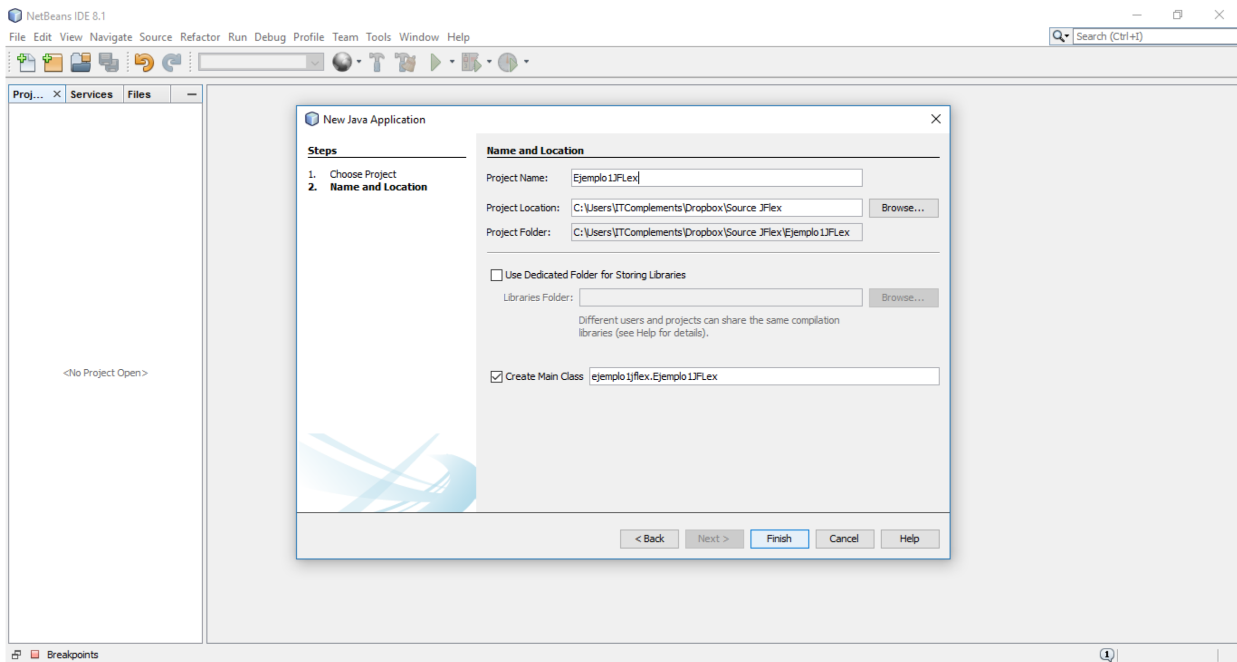


Figura 42. Creación del proyecto “Ejemplo1JFLex”.

Una vez generado, tal y como se muestra en la **Figura 43**, es necesario agregar el archivo de configuración que en este caso será nombrado `Lexer.flex`, ver **Figura 44**, y donde se contendrán todas las configuraciones y expresiones regulares necesarias, así mismo reglas, macros y código necesario para poder definir el correcto funcionamiento de JFLex para la generación del analizador lexicográfico.

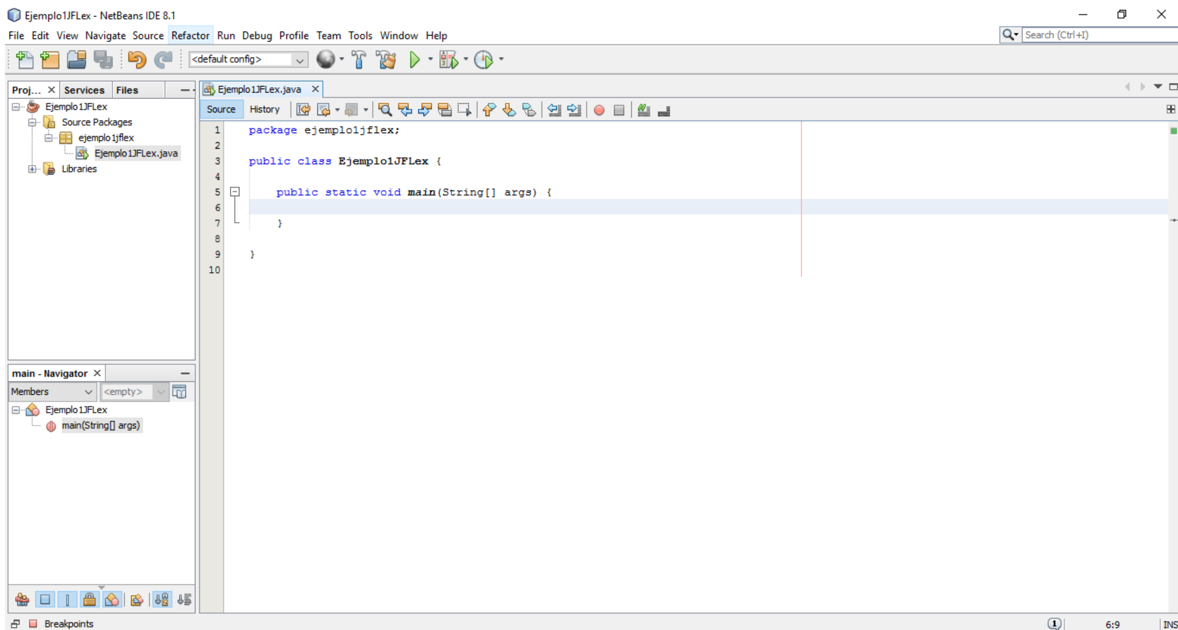


Figura 43. Creación de la clase principal, nombrada igual que el proyecto.

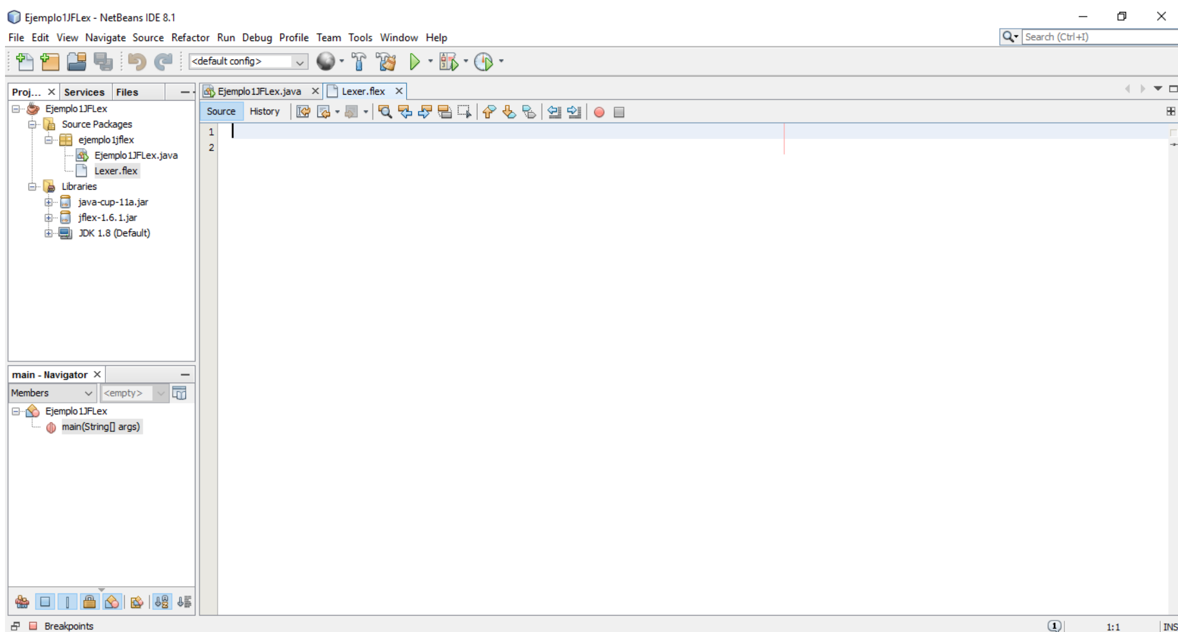


Figura 44. Generación del archivo `Lexer.flex` para configurar JFLex.

Posterior a tener el archivo de configuración, y recordando las secciones y opciones que se denotaron en la sección anterior, comienza la declaración de configuración donde se ha de declarar en que paquete se va a localizar la clase del analizador léxico y su funcionamiento.

La primera sección incluye el código del usuario, indicando el paquete donde se encontrará el analizador léxico y en conjunto de los `imports` necesarios, para ello se integrarán las debidas librerías para la escritura de ficheros en base a un buffer de datos para poder visualizar los resultados del análisis. Se ha de incluido una clase denominada `Yytoken` donde se ha de mostrar cada *token*, en que línea y en que columna aparece, el tipo de *token* y cuál fue la coincidencia encontrada. Habrá que recordar que todo lo que se coloca en la primera sección de este documento, se transcribe de la misma forma, literalmente, dentro de la nueva clase a generar del analizador lexicográfico antes de la declaración de la nueva clase.

La primera sección puede visualizarse en la **Figura 45** donde es posible apreciar la creación de la clase `Yytoken`, la declaración de sus atributos y la función que ha de regresar la cadena formada por el conjunto de valores asignados en el constructor de la clase. Habrá que verificar cuidadosamente los identificadores de acceso al declarar una clase en esta sección, recordando las reglas de programación en Java.

Continuando con la segunda sección, mostrada en la **Figura 46**, donde son declaradas diversas opciones propias de JFLex en conjunto de las declaraciones de las expresiones regulares que conformaran el funcionamiento del autómata para dar forma al nuevo analizador lexicográfico.

Como es de apreciar, se hace uso de diversas opciones de JFLex ya mencionadas, la principal es el asignar un nuevo nombre de la función scanner con `function nextToken`, consiguiente, cambiar el identificador de acceso de la clase a `public` y asignar un nuevo nombre para la clase analizador, sustituyendo el nombre por defecto a `AnalizadorLexico`.

```
1   /*Seccion de codigo de usuario*/
2   package ejemplo1jflex;
3
4   import java.io.BufferedWriter;
5   import java.io.FileWriter;
6   import java.io.IOException;
7   import java.util.ArrayList;
8
9   //Clase de los token devueltos
10  class Yytoken {
11      //Metodos de los atributos de la clase
12      public int numToken;
13      public String token;
14      public String tipo;
15      public int linea;
16      public int columna;
17
18      Yytoken (int numToken,String token, String tipo, int linea, int columna){
19          //Contador para el numero de tokens reconocidos
20          this.numToken = numToken;
21          //String del token reconocido
22          this.token = token;
23          //Tipo de componente lexico encontrado
24          this.tipo = tipo;
25          //Numero de linea
26          this.linea = linea;
27          //Columna donde empieza el primer caracter del token
28          this.columna = columna;
29      }
30
31      //Metodo que devuelve los datos necesarios que se escribiran en un archivo de salida
32      public String toString() {
33          return "Token #"+numToken+": "+token+" C.Lexico: "+tipo+" ["+linea
34          + "," +columna + "]";
35      }
36  }
```

Figura 45. Sección del código del usuario dentro del archivo de configuración `Lexer.flex`.

```
38  /* Seccion de opciones y declaraciones de JFlex */
39  %% //inicio de opciones
40  //Cambiamos el nombre la funcion para el siguiente token por nextToken
41  %function nextToken
42  //Clase publica
43  %public
44  //Cambiamos el nombre de la clase del analizador
45  %class AnalizadorLexico
46  //Agregamos soporte a unicode
47  %unicode
48  //Codigo java
49  %{
50
51      private int contador;
52      private ArrayList<Yytoken> tokens;
53
54       private void writeOutputFile() throws IOException{
55              String filename = "file.out";
56              BufferedWriter out = new BufferedWriter(
57                  new FileWriter(filename));
58            System.out.println("\n*** Tokens guardados en archivo ***\n");
59              for(Yytoken t: this.tokens){
60                  System.out.println(t);
61                  out.write(t + "\n");
62              }
63              out.close();
64       }
65  %}
66  //Creamos un contador para los tokens en el constructor de la clase
67  %init{
68      contador = 0;
69       tokens = new ArrayList<Yytoken>();
70  %init}
71  //Se escribe en un fichero los tokens al encontrar el EOF
72  %eof{
73       try{
74           this.writeOutputFile();
75         System.exit(0);
76       }catch(IOException ioe){
77           ioe.printStackTrace();
78       }
79  %eof}
```

```
1   //Activar el contador de lineas, variable yyline
2   %line
3   //Activar el contador de columna, variable yycolumn
4   %column
5   //Fin de opciones
6
7   //Expresiones regulares
8   //Declaraciones
9   EXP_ALPHA=[A-Za-z]
10  EXP_DIGITO=[0-9]
11  EXP_ALPHANUMERIC={EXP_ALPHA}|{EXP_DIGITO}
12  NUMERO=({EXP_DIGITO})+
13  IDENTIFICADOR={EXP_ALPHA}({EXP_ALPHANUMERIC})*
14  ESPACIO=" "
15  SALTO=\n|\r|\r\n
16  //fin declaraciones
```

Figura 46. Configuración de la sección de opciones y declaraciones del archivo `Lexer.flex`.

Se habilita la entrada de caracteres Unicode y se inicializa un contador en conjunto con un arreglo de `Yytoken` los cuales se escribirán posteriormente en un archivo de salida por medio de la intromisión del código denotado entre la sección `%{…}%` el cual es colocado antes de generar alguno de los constructores de la clase.

El código contenido en la sección `%init{…%init}` es transcrita al constructor de la clase del analizador, por lo tanto, por cada constructor existente, existirá el código contenido en esta sección, por ello se inicializa el contador y el arreglo de *tokens* existentes.

En la sección `%eof{…%eof}` indica que aquel código encontrado en esta parte, será ejecutado una vez que sea encontrado el fin del archivo (End Of File) por ello se hace el llamado al método encargado de escribir en un archivo todos aquellos *tokens* encontrados. Y para finalizar la sección de opciones, se habilitan los contadores de líneas y columnas con las funciones `line` y `column` respectivamente.

Para finalizar esta sección, se tiene la subsección de las expresiones regulares, donde se declaran todas aquellas expresiones que van a definir nuestro lenguaje, pero de la forma a como lo muestran las gramáticas libres de contexto anteriormente mencionadas. De esta forma es más fácil y rápido definir el lenguaje sin la necesidad de hacer uso de los autómatas deterministas donde es tedioso y tardado definir cada uno de los estados perteneciente a las producciones que serán aceptadas.

La última sección es aquella donde serán definidas las diversas reglas lexicográficas con sus respectivas acciones, tal y como es mostrado en la **Figura 47**. En cada regla, se define una expresion regular y que accion se debe tomar al encontrar cada una de estas coincidencias, en el ejemplo en curso, por cada cioncidencia o regla, se creara un objeto del tipo `Yytoken` para posteriormente agregarlo al arreglo de *tokens* que seran escritos en el fichero de salida y aumentando el contador por cada token encontrado. Es pos ello que las reglas y las expresiones regulares son dependientes, hasta cierto punto, ya que por cada expresion regular que sea de suma importancia o que se desee identificar su aparicion, es necesario generar una regla para llevar a cabo la accion correspondiente, si ese es el caso.

```
96
97   /* Seccion de reglas lexicas */
98   %%
99   //Regla      {Acciones}
100
101  {NUMERO}    {
102      contador++;
103      Ytoken t = new Ytoken(contador,yytext(),"num",yyline,yycolumn);
104      tokens.add(t);
105      return t;
106  }
107  {IDENTIFICADOR}   {
108      contador++;
109      Ytoken t = new Ytoken(contador,yytext(),"id",yyline,yycolumn);
110      tokens.add(t);
111      return t;
112  }
113  "+"   {
114      contador++;
115      Ytoken t = new Ytoken(contador,yytext(),"suma",yyline,yycolumn);
116      tokens.add(t);
117      return t;
118  }
119  "-"   {
120      contador++;
121      Ytoken t = new Ytoken(contador,yytext(),"resta",yyline,yycolumn);
122      tokens.add(t);
123      return t;
124  }
125  "*"   {
126      contador++;
127      Ytoken t = new Ytoken(contador,yytext(),"multiplicacion",yyline,yycolumn);
128      tokens.add(t);
129      return t;
130  }
```

```
1   "/"  {
2       contador++;
3       Ytoken t = new Ytoken(contador,yytext(),"division",yyline,yycolumn);
4       tokens.add(t);
5       return t;
6   }
7   "="  {
8       contador++;
9       Ytoken t = new Ytoken(contador,yytext(),"igualdad",yyline,yycolumn);
10      tokens.add(t);
11      return t;
12  }
13  {ESPACIO} {
14      //ignorar
15  }
16  {SALTO} {
17      contador++;
18      Ytoken t = new Ytoken(contador,"","fin_linea",yyline,yycolumn);
19      tokens.add(t);
20      return t;
21  }
22  [^] {
23      contador++;
24      Ytoken t = new Ytoken(contador,"","error",yyline,yycolumn);
25      tokens.add(t);
26      return t;
27  }
```

Figura 47. Configuración de las reglas de la última sección del archivo `Lexer.flex`.

De esta forma es más sencillo y más rápido definir la funcionalidad de JFLex en conjunto con la definición de la funcionalidad del analizador léxico, evitando la tediosa tarea de definir el autómata con cada uno de sus estados de aceptación y las transiciones correspondientes entre ellos, para que posteriormente, al ser ejecutado, JFLex construya el AFD que se define en base a las expresiones regulares, en conjunto con las reglas, en un AFND, el cual, posteriormente minimiza su longitud a la mayor fracción posible para finalmente construir la clase del analizador léxico.

Para poder generar la nueva clase del analizador lexicográfico haciendo uso de JFLex, es necesario ejecutar las instrucciones mostradas en la **Figura 48**, donde se hace uso del método `jflex.Main.generate` que recibe como parámetro un archivo, para el ejemplo que se ha venido manejando en curso, se indica la ruta del archivo `Lexer.flex`. Al hacer uso de este método, se genera la clase indicada dentro del archivo *.flex, en la misma ubicación donde se encuentre el archivo de configuración. En la **Figura 49** se puede observar el resultado de la ejecución del archivo `Lexer.flex`, indicando la generación de los estados del autómata y el resultado al reducirlo.

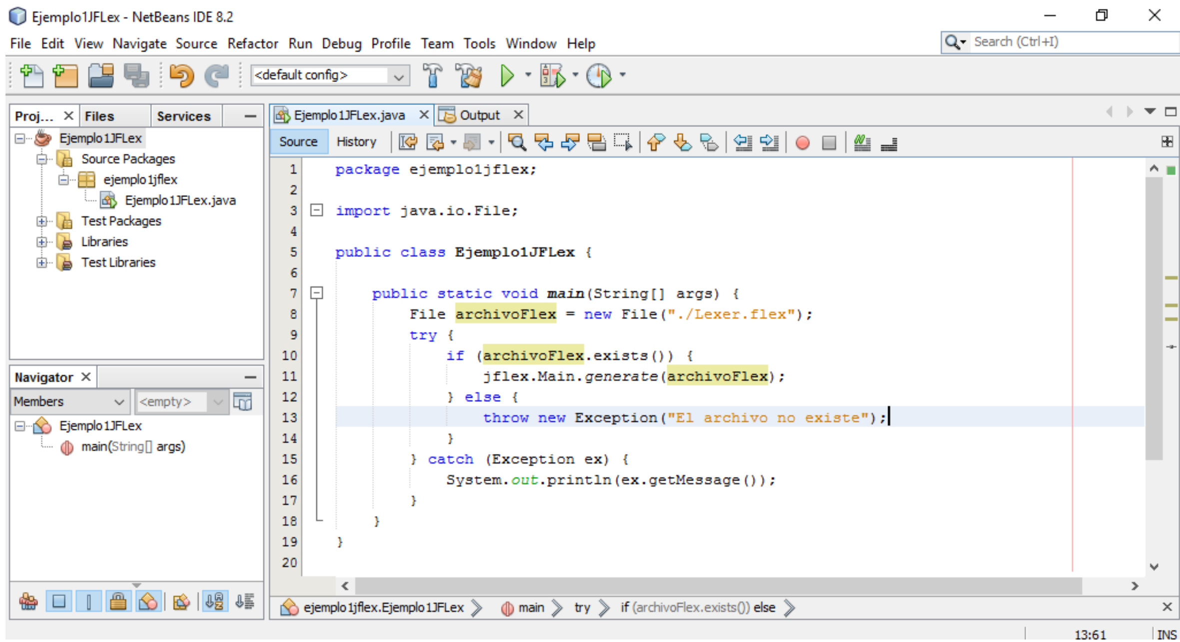


Figura 48. Generación de la clase `AnalizadorLexico`, utilizando JFLex.

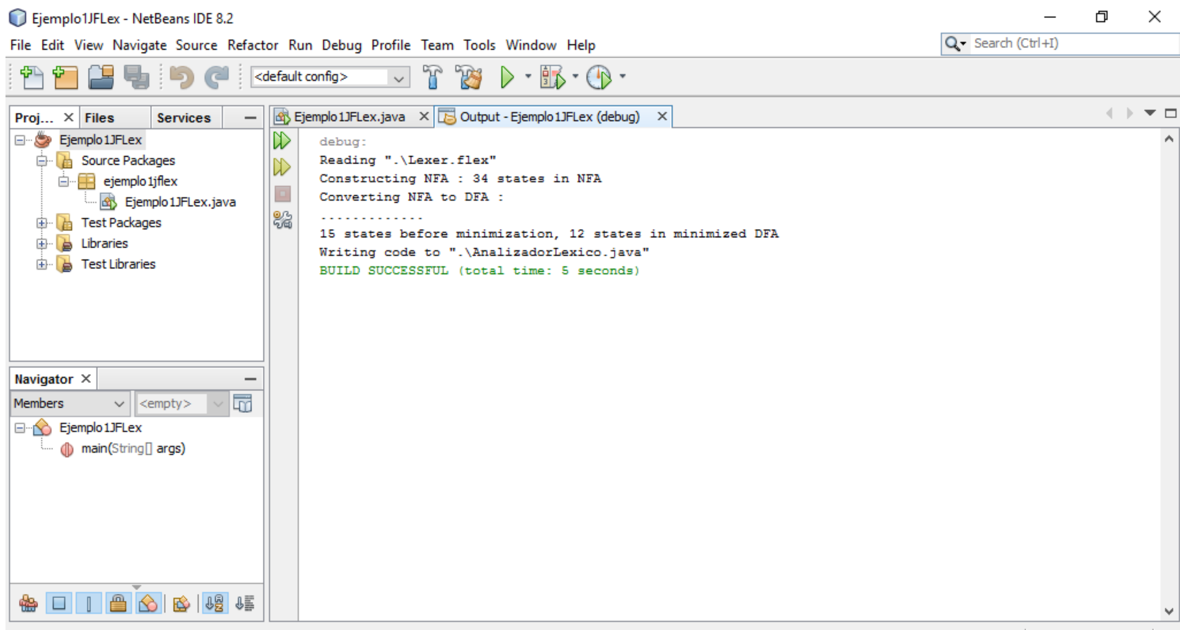


Figura 49. Resultado de la ejecución del archivo `Lexer.flex`.

Nombre	Fecha de modifica...	Tipo	Tamaño
build	20/11/2016 20:18	Carpeta de archivos	
libs	20/11/2016 11:16	Carpeta de archivos	
nbproject	20/11/2016 11:17	Carpeta de archivos	
src	20/11/2016 11:13	Carpeta de archivos	
test	20/11/2016 20:08	Carpeta de archivos	
AnalizadorLexico	20/11/2016 20:59	Archivo JAVA	21 KB
build	01/11/2016 12:44	Documento XML	4 KB
Lexer	16/11/2016 9:11	Archivo FLEX	4 KB
manifest.mf	01/11/2016 12:44	Archivo MF	1 KB

Figura 50. Ubicación de la clase `AnalizadorLexico`.

En la **Figura 50**, es posible apreciar la generación de la clase `AnalizadorLexico`, la cual va a ser agregada al paquete `ejemplo1jflex` para poder hacer uso de la misma clase por medio del IDE NetBeans.

Para poder probar la funcionalidad de la clase `AnalizadorLexico`, y verificar que está cumpliendo con su debida funcionalidad, se va a generar un archivo de entrada de texto nombrado `FileToRead.txt`, donde su contenido es el mostrado en la **Figura 51**.

```
1   12 a -5555 asr 14 sy y 13445
2   12
3   3
4   4
5   afr
6   int a += 5
7   6 + 7 + 9
8   float z = -5
9   x = 72
```

Figura 51. Contenido del archivo `FileToRead.txt`.

Posterior a tener el archivo de prueba, el cual también puede ser ingresado por medio de la consola y él envió de parámetros o como un simple `String` estático, se procede a la ejecución y validación por medio del AnalizadorLexico, tal y como se muestra en la **Figura 52** donde el principal parámetro es el archivo a leer para ir validando cada *token*.

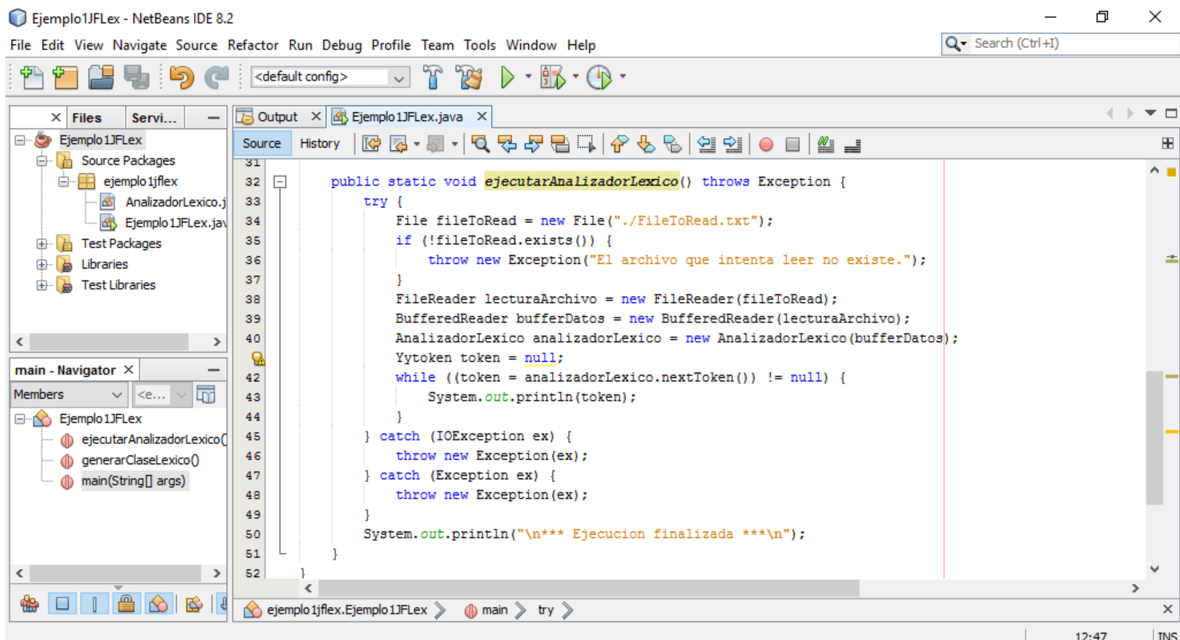


Figura 52. Lectura y ejecución del `AnalizadorLexico` en el archivo `FileToRead.txt`.

Al ejecutar la lectura del archivo, se obtiene un nuevo archivo, el cual fue indicado dentro del archivo de configuración, que contendrá el resultado de la lectura y las coincidencias encontradas. La **Figura 53** muestra los *tokens* encontrados y sus respectivas ubicaciones y cuál fue su correspondiente asimilación según las reglas establecidas y el lenguaje definido en base a las expresiones regulares. Para el ejemplo ejecutado, fue una ejecución "limpia", ya que, solo se introdujeron aquellos caracteres aceptados por el lenguaje, pero en dado caso de introducir un carácter no valido para el `AnalizadorLexico`, tal como se muestra en **Figura 54**, se marcaria la existencia de un error.

```
1   Token #1: 12 C.Lexico: num [0,0]
2   Token #2: a C.Lexico: id [0,3]
3   Token #3: - C.Lexico: resta [0,5]
4   Token #4: 5555 C.Lexico: num [0,6]
5   Token #5: asr C.Lexico: id [0,11]
6   Token #6: 14 C.Lexico: num [0,15]
7   Token #7: sy C.Lexico: id [0,18]
8   Token #8: y C.Lexico: id [0,21]
9   Token #9: 13445 C.Lexico: num [0,23]
10  Token #10:  C.Lexico: fin_linea [0,28]
11  Token #11: 12 C.Lexico: num [1,0]
12  Token #12:  C.Lexico: fin_linea [1,2]
13  Token #13: 3 C.Lexico: num [2,0]
14  Token #14:  C.Lexico: fin_linea [2,1]
15  Token #15: 4 C.Lexico: num [3,0]
16  Token #16:  C.Lexico: fin_linea [3,1]
17  Token #17: afr C.Lexico: id [4,0]
18  Token #18:  C.Lexico: fin_linea [4,3]
19  Token #19: int C.Lexico: id [5,0]
20  Token #20: a C.Lexico: id [5,4]
21
```

```
1   Token #21: + C.Lexico: suma [5,6]
2   Token #22: = C.Lexico: igualdad [5,7]
3   Token #23: 5 C.Lexico: num [5,9]
4   Token #24:  C.Lexico: fin_linea [5,10]
5   Token #25: 6 C.Lexico: num [6,0]
6   Token #26: + C.Lexico: suma [6,2]
7   Token #27: 7 C.Lexico: num [6,4]
8   Token #28: + C.Lexico: suma [6,6]
9   Token #29: 9 C.Lexico: num [6,8]
10  Token #30:  C.Lexico: fin_linea [6,9]
11  Token #31: float C.Lexico: id [7,0]
12  Token #32: z C.Lexico: id [7,6]
13  Token #33: = C.Lexico: igualdad [7,8]
14  Token #34: - C.Lexico: resta [7,10]
15  Token #35: 5 C.Lexico: num [7,11]
16  Token #36:  C.Lexico: fin_linea [7,12]
17  Token #37: x C.Lexico: id [8,0]
18  Token #38: = C.Lexico: igualdad [8,2]
19  Token #39: 72 C.Lexico: num [8,4]
20
```

Figura 53. Resultado de la lectura del archivo `FileToRead.txt`.

```
1   12 a -5555 asr 14 sy y 13445
2   12
3   3
4   4
5   afr
6   int a += 5
7   6 + 7 + 9
8   float z = -5
9   x = 7\2
```

Figura 54. Introducción del carácter "\", mostrándose como carácter no correspondiente al lenguaje.

Posterior a la segunda ejecución con el carácter no perteneciente al lenguaje, se obtiene lo mostrado en la **Figura 55**, donde al encontrar un carácter no perteneciente al lenguaje, como lo es el carácter "\", se marca como un error, en el ejemplo en curso, el manejo de los errores solo son almacenados y mostrados bajo la etiqueta "error" pero la flexibilidad de JFLex, permite el emitir el error a tal grado de detener la ejecución del sistema o realizar la acción correspondiente según la necesidad y las instrucciones proporcionadas por el usuario, por ello JFLex sigue siendo uno de los mejores y más versátiles generadores de analizadores lexicográficos por la facilidad y flexibilidad que tiene con el usuario de poder generar sus propias funciones y manejar cada caso según le sea necesario.

```
Token #1: 12 C.Lexico: num [0,0]
Token #2: a C.Lexico: id [0,3]
Token #3: - C.Lexico: resta [0,5]
Token #4: 5555 C.Lexico: num [0,6]
Token #5: asr C.Lexico: id [0,11]
Token #6: 14 C.Lexico: num [0,15]
Token #7: sy C.Lexico: id [0,18]
Token #8: y C.Lexico: id [0,21]
Token #9: 13445 C.Lexico: num [0,23]
Token #10:  C.Lexico: fin_linea [0,28]
Token #11: 12 C.Lexico: num [1,0]
Token #12:  C.Lexico: fin_linea [1,2]
Token #13: 3 C.Lexico: num [2,0]
Token #14:  C.Lexico: fin_linea [2,1]
Token #15: 4 C.Lexico: num [3,0]
Token #16:  C.Lexico: fin_linea [3,1]
Token #17: afr C.Lexico: id [4,0]
Token #18:  C.Lexico: fin_linea [4,3]
Token #19: int C.Lexico: id [5,0]
Token #20: a C.Lexico: id [5,4]
```

```
Token #21: + C.Lexico: suma [5,6]
Token #22: = C.Lexico: igualdad [5,7]
Token #23: 5 C.Lexico: num [5,9]
Token #24:  C.Lexico: fin_linea [5,10]
Token #25: 6 C.Lexico: num [6,0]
Token #26: + C.Lexico: suma [6,2]
Token #27: 7 C.Lexico: num [6,4]
Token #28: + C.Lexico: suma [6,6]
Token #29: 9 C.Lexico: num [6,8]
Token #30:  C.Lexico: fin_linea [6,9]
Token #31: float C.Lexico: id [7,0]
Token #32: z C.Lexico: id [7,6]
Token #33: = C.Lexico: igualdad [7,8]
Token #34: - C.Lexico: resta [7,10]
Token #35: 5 C.Lexico: num [7,11]
Token #36:  C.Lexico: fin_linea [7,12]
Token #37: x C.Lexico: id [8,0]
Token #38: = C.Lexico: igualdad [8,2]
Token #39: 7 C.Lexico: num [8,4]
Token #40: \ C.Lexico: error [8,5]
Token #41: 2 C.Lexico: num [8,6]
```

Figura 55. Manejo del carácter “no aceptado” por el `AnalizadorLexico`.

4.2.3. EJEMPLOS DE INTERES

Continuando con los ejemplos sobre el funcionamiento y usabilidad de JFLex y haciendo uso del ejemplo anterior, se visualizó el uso de la palabra reservada `int` como un simple identificador y no como un mnemónico que identifica un tipo de dato, por ello, en esta modificación va a ser utilizada una regla, la cual va a identificar aquellas palabras que se deseen identificar como una función específica y no como un simple identificador como se logró apreciar anteriormente. El principal motivo, y como ya fue mencionado, JFLex identifica aquellas cadenas de caracteres que más se acerquen a empatar las expresiones regulares definidas en las declaraciones, por esta razón, al definir una cadena en una regla, ofrece una mayor prioridad a todas aquellas cadenas que empaten con la expresión definida dentro de la macro. Dentro de la **Figura 56** se logra apreciar la integración de los mnemónicos como reglas que identificaran algunas de las palabras reservadas, tal y como normalmente suele suceder en los típicos lenguajes de programación. Cabe hacer notar, que al definir los mnemónicos, que desean ser identificados, es necesario declararlos antes de la definición del “IDENTIFICADOR”, el cual es una expresión regular que define la combinación de caracteres alfanuméricos, si al declarar los mnemónicos posterior a esta definición, JFLex los tomaría como un identificador y no como una palabra reservada, ya que, la jerarquía de prioridades recae en el “IDENTIFICADOR” el cual abarca toda combinación de caracteres alfanuméricos.

Figura 56. Configuración del archivo `Lexer.flex`, integrando algunos mnemónicos de tipos de dato.

Posteriormente a la lectura del archivo `Lexer.flex`, él puede visualizarse en la **Figura 57**, y a la generación de la nueva clase AnalizadorLexico, se procese a leer nuevamente el archivo de texto con las diversas cadenas que se desean identificar, obteniendo el resultado mostrado en la **Figura 58**.

```
1   12 a -5555 asr 14 sy y 13445
2   12
3   3
4   4
5   afr
6   int a += 5
7   double z = 6 + 7 + 9
8   float z = -5
9   String string
10  x = 7\2
```

Figura 57. Nuevo archivo `Lexer.flex` con la integración de algunos mnemónicos utilizados en los lenguajes de programación de alto nivel.

```
Token #1: 12 C.Lexico: num [0,0]
Token #2: a C.Lexico: id [0,3]
Token #3: - C.Lexico: resta [0,5]
Token #4: 5555 C.Lexico: num [0,6]
Token #5: asr C.Lexico: id [0,11]
Token #6: 14 C.Lexico: num [0,15]
Token #7: sy C.Lexico: id [0,18]
Token #8: y C.Lexico: id [0,21]
Token #9: 13445 C.Lexico: num [0,23]
Token #10:  C.Lexico: fin_linea [0,28]
Token #11: 12 C.Lexico: num [1,0]
Token #12:  C.Lexico: fin_linea [1,2]
Token #13: 3 C.Lexico: num [2,0]
Token #14:  C.Lexico: fin_linea [2,1]
Token #15: 4 C.Lexico: num [3,0]
Token #16:  C.Lexico: fin_linea [3,1]
Token #17: afr C.Lexico: id [4,0]
Token #18:  C.Lexico: fin_linea [4,3]
Token #19: int C.Lexico: entero [5,0]
Token #20: a C.Lexico: id [5,4]
Token #21: + C.Lexico: suma [5,6]
Token #22: = C.Lexico: igualdad [5,7]
Token #23: 5 C.Lexico: num [5,9]
Token #24:  C.Lexico: fin_linea [5,10]
```

```
Token #25: double C.Lexico: doble_presicion [6,0]
Token #26: z C.Lexico: id [6,7]
Token #27: = C.Lexico: igualdad [6,9]
Token #28: 6 C.Lexico: num [6,11]
Token #29: + C.Lexico: suma [6,13]
Token #30: 7 C.Lexico: num [6,15]
Token #31: + C.Lexico: suma [6,17]
Token #32: 9 C.Lexico: num [6,19]
Token #33:  C.Lexico: fin_linea [6,20]
Token #34: float C.Lexico: flotante [7,0]
Token #35: z C.Lexico: id [7,6]
Token #36: = C.Lexico: igualdad [7,8]
Token #37: - C.Lexico: resta [7,10]
Token #38: 5 C.Lexico: num [7,11]
Token #39:  C.Lexico: fin_linea [7,12]
Token #40: String C.Lexico: cadena_caracteres [8,0]
Token #41: string C.Lexico: id [8,7]
Token #42:  C.Lexico: fin_linea [8,13]
Token #43: x C.Lexico: id [9,0]
Token #44: = C.Lexico: igualdad [9,2]
Token #45: 7 C.Lexico: num [9,4]
Token #46: \ C.Lexico: error [9,5]
Token #47: 2 C.Lexico: num [9,6]
```

Figura 58. Resultado de la ejecución del AnalizadorLexico, con los nuevos estados del archivo `Lexer.flex`.

De esta forma, es posible ir definiendo el analizador lexicográfico de forma gradual para la generación del compilador, donde posteriormente será posible integrarlo con otra herramienta llamada CUP, la cual se encarga del análisis sintáctico.

4.3. ANALIZADOR SINTÁCTICO

Todo lenguaje de programación obedece un conjunto de reglas que describen la estructura "bien formada" del código fuente del programa. Es posible describir la sintaxis de las construcciones de los lenguajes de programación por medio de las gramáticas de libre contexto o por alguna notación como lo es Backus Naur Form. Al hacer uso de las gramáticas formales, se ofrece diversas ventajas significativas al momento del diseño y el desarrollo de los compiladores, tales como lo son:

- A partir de la gramática, es posible generar el analizador sintáctico.
- Las gramáticas permiten el descubrimiento de ambigüedades.
- Una gramática proporciona la estructura a un lenguaje de programación, haciendo más fácil la generación de código y la detección de errores.
- Es más sencillo realizar una ampliación o modificación al lenguaje al estar generado por una gramática.

El funcionamiento primordial del analizador sintáctico es verificar la secuencia de los *tokens* en base a la gramática establecida, alimentando el árbol sintáctico, validando los *tokens* de entrada, siendo esta parte el punto de partida, pero el

analizador sintáctico, no solamente realiza la validación de los *tokens* de entrada, también dirige el proceso de compilación tales como:

- Incorporar acciones semánticas, análisis semántico, hasta generación de código.
- Informa sobre la naturaleza de los errores sintácticos presentes e intenta la recuperación posterior a los errores para continuar con la compilación.
- Controla el flujo de *tokens* reconocidos por parte del analizador lexicográfico.

Como es posible observar, realiza casi todas las operaciones de la compilación dando lugar a la compilación dirigida por sintaxis. La **Figura 59** muestra un diagrama, de forma resumida, sobre el funcionamiento del analizador sintáctico.

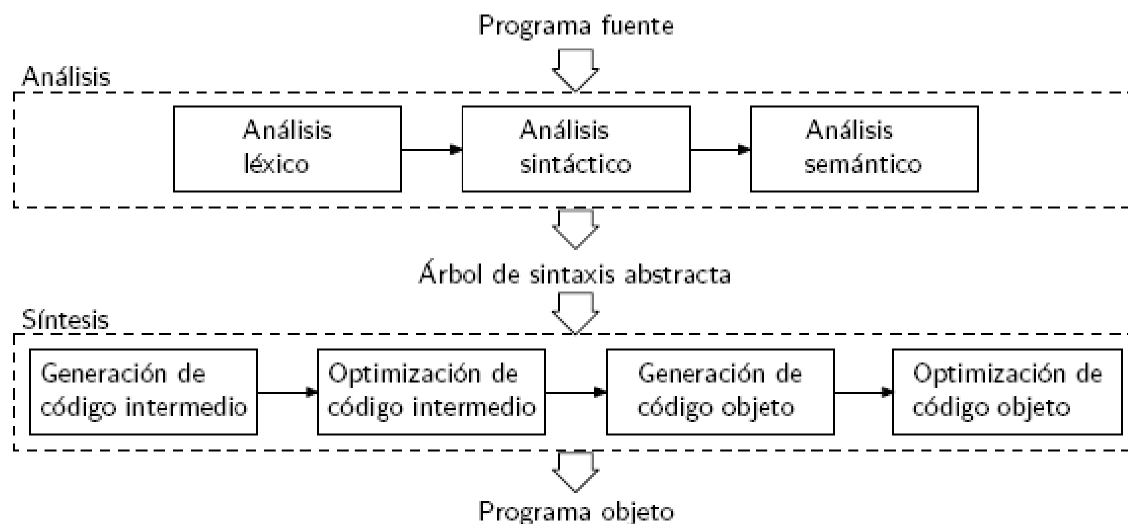


Figura 59. Diagrama del funcionamiento del análisis sintáctico.

4.4. ¿QUÉ ES CUP?

CUP es una base para la construcción de analizadores sintácticos, obtiene su nombre por sus siglas en ingles Based Constructor of Useful Parsers. CUP es un sistema que permite la generación de analizadores o parsers de tipo LALR (Look – Ahead Left to Right parser) para Java, cumpliendo de esta forma el mismo rol que ejerce YACC como generador de analizadores sintácticos, pero con un mayor número de funcionalidades. Fue desarrollado por Scott Ananian, Frank Flannery,

Dan Wang, Andrew Apple y Michael Petter en Java, en el Instituto de Tecnología de Georgia.

CUP al utilizar un parser del tipo LR permite la generación de árboles sintácticos con una derivación más a la derecha, dando la oportunidad de reconocer una gran cantidad de lenguajes de programación que pueden llegar a ser generados por medio de una gramática de libre contexto, así mismo permite el reconocimiento de errores dentro de la sintaxis del programa, de forma instantánea y precisa, otra de las grandes ventajas al utilizar un analizador LR, recae en que, si la gramática contiene una gran cantidad de ambigüedades o alguna construcción difícil de comprender, el metacompilador es capaz de reconocer cual es la regla de producción que ocasiona la problemática e informar sobre ella al desarrollador.

Una de las grandes ventajas que existen en CUP, además de su velocidad, del gran número de nuevas características que presenta y su eficiencia en implementación al trabajar en conjunto con Java, CUP permite la integración de usuarios de otros metacompiladores, como lo son, el anteriormente mencionado, YACC o Bison ya que los fundamentos y parte del conocimiento semántico de ambos metacompiladores, son de ayuda para comprender el funcionamiento de CUP.

4.4.1. FUNCIONES Y ESTRUCTURA DE CUP

CUP permite la generación de los analizadores sintácticos, por medio de especificaciones propiciadas por una gramática libre de contexto, haciendo uso de la notación BNF, además de que permite su integración con algún scanner como lo es JFLex, JLex, Lex o Flex, siendo JFLex uno de los más populares para su uso por las características y funcionalidades ya mencionadas anteriormente.

CUP, al igual que JFLex o algún otro metacompilador, permite un fichero de entrada, el cual se encarga de la configuración sobre las acciones, así mismo, la declaración de la gramática a analizar, la **Figura 60** muestra la estructura del archivo, el cual debe corresponde a la extensión *.cup.

```
//Package e Imports

package name;
import package_name.class_name;

//Componentes de codigo de usuario

action code {: ... :};
parser code {: ... :};
init with {: ... :};
scan with {: ... :};

//Lista de simbolos

terminal classname name1, name2, ...;
non terminal classname name1, name2, ...;
terminal name1, name2, ...;
non terminal name1, name2, ...;

//Declaraciones de Precedencia y Asociatividad

precedence left     terminal[, terminal...];
precedence right    terminal[, terminal...];
precedence nonassoc terminal[, terminal...];

//Gramatica

start with non-terminal;
Producciones de la Gramatica
```

Figura 60. Estructura del archivo de configuración `Parser.cup`.

4.4.1.1. FORMATO DE FICHERO DE ENTRADA

Tal y como se mencionó, para la configuración y funcionamiento de CUP, es necesaria la implementación de un fichero del cual CUP, lee y reconoce aquellas funciones o características que se desean habilitar o utilizarse. La estructura del archivo, se divide en 5 secciones, de la misma forma en cómo se listan a continuación:

- **Especificaciones de importación y empaquetamiento:** Dentro de esta sección, se incluye la información del paquete donde se encontrará localizada la clase del analizador, siguiendo las reglas establecidas en el empaquetamiento de Java y con el mnemónico `package`; la importación, indica el uso de aquellas librerías o paquetes necesarios para el funcionamiento del analizador, de igual forma utilizando las reglas de importación establecidas en Java por el mnemónico `import`. Las importaciones, son agregadas a la clase del analizador sintáctico en conjunto de los paquetes e importaciones que establece CUP por defecto, cabe mencionar que es opcional el hacer uso de esta sección.

- **Código de usuario:** Permite agregar, de forma opcional, código Java que el usuario genera para posteriormente incluirlo como parte del analizador sintáctico, para ello, es necesario hacer uso de las siguientes funciones, las cuales, su uso es opcional.
  - `action code {: ...:};`: Dentro de esta función, el usuario agrega código propio, el cual, posteriormente, es agregado a una clase no publica para ser incluido a la clase del analizador, de esta forma es posible declarar variables o rutinas.
  - `parser code {: ...:};`: Permite declarar métodos y variables que serán integrados directamente a la clase del analizador sintáctico, y aunque no sea una acción muy común, es posible agregar un método de análisis lexicográfico o alguna otra rutina, según sea el caso.
  - `init with {: ...:};`: Permite agregar código que es declarado y ejecutado, antes de que sea leído el primer *token*, generalmente utilizado para inicializar alguna variable o la ejecución y declaración del análisis lexicográfico para la previa validación de las cadenas de caracteres a leer, esta sección retorna un valor vacío o `void`.
  - `scan with {: ...:};`: Define como el analizador sintáctico debería verificar o validar sobre el siguiente *token* a leer, esta subsección, al igual que la anterior, posee un valor de retorno, pero a diferencia del anterior, retorna un objeto del tipo `java_cup.runtime.Symbol`.
- **Lista de símbolos:** Esta sección, es estrictamente requerida ya que posee la lista de símbolos o, en otros términos, la declaración de los símbolos pertenecientes a la gramática. Dentro de esta sección, se declara la nomenclatura o nombramiento de cada símbolo terminal y no-terminal. Para definir dentro del analizador que tipo de objeto es cada símbolo, basta con definir si son `terminal` o `non terminal`, tal y como se muestra a continuación.
  - `terminal` *classname*, name1, name2…;
  - `non terminal` *classname*, name1, name2…;
  - terminal name1, name2…;

- `non terminal` name1, name2…;

Donde, *classname* puede ser definido en un nombre de múltiples partes separados por ".", además, el tipo de valor para cada terminal o no-terminal, en caso de no ser definido, no tendrán valor alguno de retorno. La definición de la lista de símbolos, no deben ser una palabra reservada de CUP.

- **Declaraciones de Precedencia y Asociatividad**: Esta sección es opcional y funcional con gramáticas ambiguas permitiendo definir la precedencia y cuál será la asociatividad por cada declaración, CUP permite tres formas:

  `precedence left terminal` [, terminal...];

  `precedence right terminal` [, terminal...];

  `precedence nonassoc terminal` [, terminal...];

  Donde cada lista de terminales, define la asociación especifica del nivel precedente de la declaración, definidas en orden, del más alto al más bajo y de abajo hacia arriba. Al declarar las procedencias de esta forma, es más sencillo resolver los problemas de reducción de operaciones.

  CUP define una procedencia para cada una de las terminales, según una de estas declaraciones, y toda terminal que no se encuentre en esta declaración, posee la más baja procedencia al igual que toda terminal o no-terminal declarada con alguna producción.

- **Gramática:** La última sección del fichero de configuración de CUP, generalmente empieza con la sección `start with non-terminal;` pero de forma opcional. De esta forma se indica cual no-terminal es el comienzo para el analizador sintáctico. En dado caso de no ser declarado, el primer no-terminal declarado a la izquierda de la producción, será utilizado. Al finalizar un análisis exitoso, CUP retorna un objeto del tipo `java_cup.runtime.Symbol`. Al definir cada una de las producciones de la gramática, comienza con un no-terminal definido a la izquierda, seguido por el símbolo "`::=`" que es seguido por una nula o variada cantidad de acciones,

símbolos terminales o no-terminales, seguido por una asignación procedente contextual, la cual puede ser opcional, y finalizado por "`;`". Cada uno de los símbolos localizados a la derecha, pueden ser etiquetados, donde cada etiqueta es nombrada posteriormente al símbolo "`:`" que se localizada inmediatamente de cada símbolo definido, además de ser única la etiqueta y con la posibilidad de ser utilizada en diversas acciones al hacer referencia a la etiqueta. La gramática, también permite el uso de diversas producciones, para ello se hace uso de "`|`". Las acciones son definidas dentro de "`{: …:}`", generalmente código Java que determina dicha acción, estas acciones son excluidas del analizador, pero llamadas al llegar al ser reconocida la producción. De igual forma es posible integrar procedencias contextuales las cuales permiten asignar un símbolo de forma precedente a la producción, para ello es necesario incluir "`%prec`" al término de la declaración de la producción.

Para poder hacer uso de CUP, y generar un analizador sintáctico, es necesario ejecutar el método `java_cup.Main` el cual genera dos ficheros:

- `sym.java` – Contiene las definiciones de constantes de la clase `sym`, el cual asigna un valor entero a cada uno de los terminales y a cada uno de los no-terminales, si fuera necesario.
- `Parser.java` – Es la clase que contiene la lógica del analizador sintáctico que contiene CUP.

Adicional a lo ya mencionado, la clase del analizador sintáctico `parser.java`, posee dos clases encargadas de las acciones de la gramática y de su implementación:

- `public class parser extends java_cup.runtime.lr_parser {...}`: Clase publica y propia del analizador, e implementa la tabla de acciones de un analizador LALR al ser una subclase de `java_cup.runtime.lr_parser`.
- `class CUP$parser$actions`: Clase no publica que se encuentra encapsulada en la clase del analizador sintáctico y posee las acciones que han sido descritas en la gramática, además de poseer un método encargado

de ejecutar las diversas acciones que han sido definidas en cada regla sintáctica: `public final java_cup.runtime.Symbol CUP$parser$do_action`.

De igual forma, la clase del analizador sintáctico, posee métodos encargados de la tabla de producciones y acciones recurrentes de la gramática.

- `protected static final short [] [] _production_table`: Contiene la tabla de producciones con el número de símbolo a la izquierda de la producción, por cada producción de la gramática.
- `protected static final short [] [] _action_table`: Contiene la tabla de acciones donde se indica que acción (desplazamiento, reducción o error) debe utilizarse para cada símbolo anticipado (*lookahead*) en función del estado en el cual se encuentre el análisis.
- `protected static final short [] [] _reduce_table`: Contiene la tabla de saltos a estado, el cual indica a que estado debe ir después de una reducción, en función del no-terminal y el estado en el cual se encuentre el análisis.

4.4.2. APLICANDO CUP

Para continuar con el funcionamiento de CUP, se ejecutará un ejemplo de una calculadora que cumpla con los siguientes puntos:

- La calculadora debe permitir más de una ecuación dentro de una sola expresión, esto es:

$$2 + 5 * 6$$

- Al encontrar el símbolo “;”, se indica que es el final de la ecuación y es posible evaluarla.
- Debe de permitir operaciones binarias y unitarias, dentro de las operaciones binarias permitidas, se encuentra la suma, resta, multiplicación, modulo y división, para las unitarias, solo el símbolo negativo para un digito o un simple digito, por ejemplo:

$2 + 2$, o sea el caso unitario, -7

- Son aceptables las operaciones agrupadas entre paréntesis.

$$(ecuacion)$$

Una vez que se tienen las debidas acciones que puede ejecutar la calculadora, es posible generar la gramática perteneciente a la misma calculadora y posteriormente integrar dichas instrucciones a CUP. La gramática se muestra en **Figura 61**, recordando las reglas e instrucciones que indica CUP para la generación de las gramáticas.

```
1   EXP_LIST ::= EXP_LIST EXP_PART | EXP_PART
2   EXP_PART ::= EXP ';'
3   EXP      ::= EXP '+' EXP
4                | EXP '-' EXP
5                | EXP '*' EXP
6                | EXP '/' EXP
7                | EXP '%' EXP
8                | '(' EXP ')'
9                | '-' EXP
10               | NUMERO
```

Figura 61. Gramática perteneciente al ejemplo de la calculadora.

Posteriormente a tener la gramática definida, es necesario identificar aquellos símbolos terminales y no-terminales, los cuales se muestran en la **Figura 62**.

```
1   No-Terminales
2   EXP_LIST, EXP_PART, EXP
3
4   Terminales
5   SUMA, RESTA, MULT, DIV, MOD, SEMI, NUMERO, PIZQ, PDER, SIMNEG
```

Figura 62. Terminales y no-terminales pertenecientes a la gramática de la calculadora.

Posteriormente a tener la gramática e identificar los terminales y no-terminales, es posible generar un archivo `Parser.cup`. El archivo completo, se puede observar en la **Figura 63**.

```
Parser.cup
1   //Package e Imports
2
3   package ejemplo1cup;
4
5   import java_cup.runtime.*;
6
7   // Componentes de codigo de usuario
8
9   init with {: scanner.init();              :};
10  scan with {: return scanner.next_token(); :};
11
12  //Lista de simbolos
13
14  /* Terminales */
15  terminal           SEMI, SUMA, RESTA, MULT, DIV, MOD;
16  terminal           SIMNEG, PIZQ, PDER;
17  terminal Integer   NUMERO;
18
19  /* No Terminales */
20  non terminal            EXP_LIST, EXP_PART;
21  non terminal Integer    EXP;
22
23  //Declaracion de procedencias y asociatividad
24
25  precedence left SUMA, RESTA;
26  precedence left MULT, DIV, MOD;
27  precedence left SIMNEG;
28
29  //Gramatica
30
31  EXP_LIST ::= EXP_LIST EXP_PART | EXP_PART;
32
33  EXP_PART ::= EXP : e {: System.out.println(" = " + e); :} SEMI;
34
35  EXP      ::= EXP : e1 SUMA EXP : e2   {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
36             | EXP : e1 RESTA EXP : e2  {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
37             | EXP : e1 MULT EXP : e2  {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
38             | EXP : e1 DIV EXP : e2 {: RESULT = new Integer(e1.intValue() / e2.intValue()); :}
39             | EXP : e1 MOD EXP : e2    {: RESULT = new Integer(e1.intValue() % e2.intValue()); :}
40             | NUMERO : n               {: RESULT = n; :}
41             | RESTA EXP : e            {: RESULT = new Integer(0 - e.intValue()); :} %prec SIMNEG
42             | PIZQ EXP : e PDER    {: RESULT = e; :};
```

Figura 63. Contenido del archivo `Parser.cup`.

Dentro del archivo `Parser.cup`, se logra visualizar que el analizado será integrado al proyecto Ejemplo1CUP tal y como se muestra en la **Figura 64**.

```
Parser.cup
1  //Package e Imports
2
3  package ejemplo1cup;
4
5  import java_cup.runtime.*;
6  |
```

Figura 64. Muestra de la integración del analizador sintáctico, al proyecto Ejemplo1CUP.

Al tener nuestro archivo `Parser.cup`, es necesario ejecutarlo dentro de nuestro proyecto, el cual contiene la estructura mostrada en la **Figura 65**, donde se han agregado las librerías `java-cup-11b-runtime.jar` y `java-cup-11b.jar`, las cuales son pertenecientes a CUP y nos permitirán generar el metacompilador.

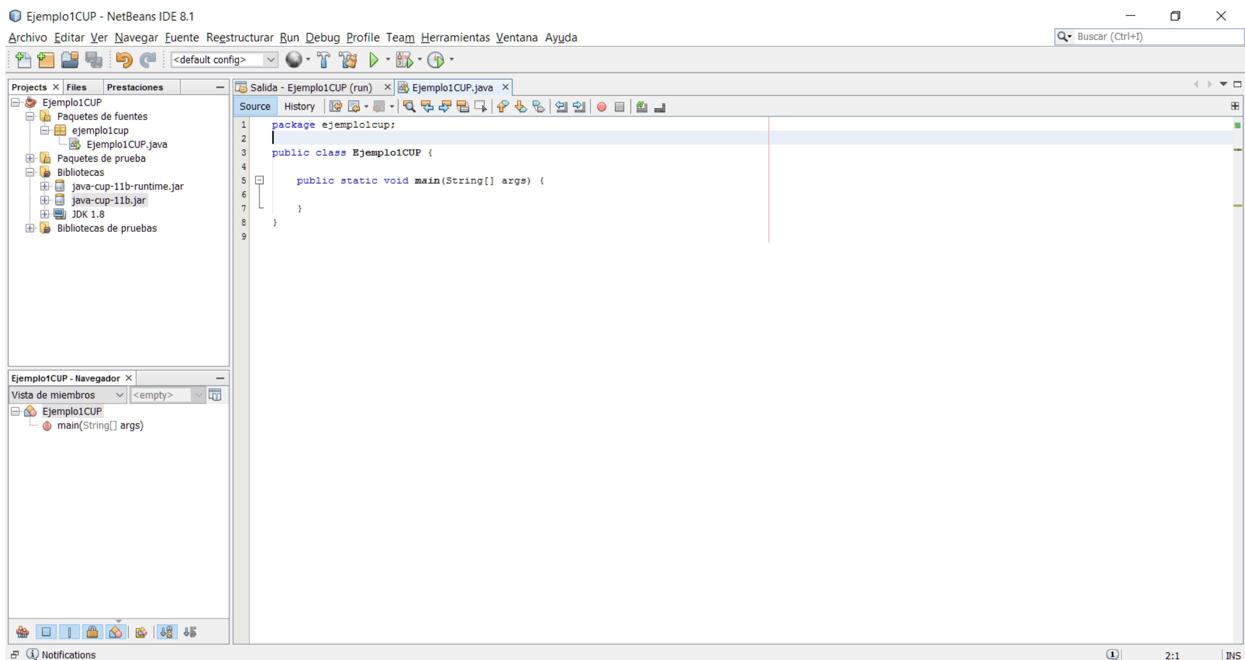


Figura 65. Estructura del proyecto Ejemplo1CUP.

Una vez que se tiene el proyecto, es necesario mostrar las indicaciones adecuadas para la generación del metacompilador, dentro de este paso se generan dos clases, la clase `Parser` y la clase `sym`, las cuales ya han sido mencionadas anteriormente.

La **Figura 66** contiene un ejemplo de las debidas instrucciones o código, con el cual es posible generar el metacompilador haciendo uso del método `java_cup.Main.main()`, el cual permite un arreglo de cadena de caracteres como entrada, dentro de este arreglo se contienen aquellos parámetros para indicar las especificaciones del comportamiento al generar el metacompilador y finalmente donde se encuentra el archivo `Parser.cup` que posee la gramática.

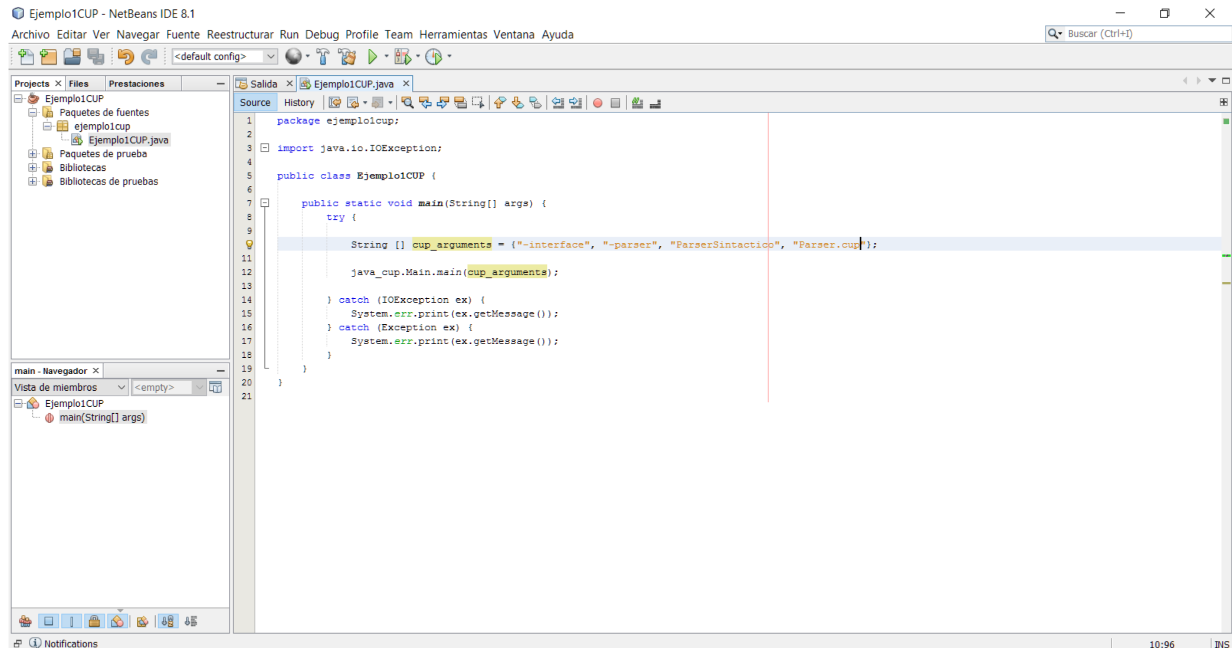


Figura 66. Código para la ejecución y generación del metacompilador, haciendo uso de CUP.

Dentro de estas especificaciones están:

- `-package` *name*: Especifica que las clases `Parser` y `sym`, se encontraran dentro del paquete *name*. Por default, al generar estas clases, no contienen ningún paquete.
- `-parser` *name*: La clase `Parser`, se genera con el nombre *name*.
- `-symbols` *name*: La clase `sym`, se genera con el nombre *name*.
- `-interface`: Genera la clase `sym` como un tipo `interface` y no del tipo `class`.

- `-nonterms`: Genera las contantes para los no-terminales dentro de la clase sym, el Parser no necesita los símbolos de los no-terminales, por esta razón, no los genera, pero generarlos de forma manual, suele ser de ayuda para el debugging.
- `-expect` *number*: Durante la construcción del `parser`, el sistema suele detectar situaciones de ambigüedad o conflictos. Por lo general el `parser`, no es capaz de decidir si es un `shift` (leer otro símbolo) o un `reduce` (reemplazar el lado derecho reconocido de una producción con su lado izquierdo), esto es denominado conflicto `shift/reduce`. De igual forma existen conflictos de reducción con dos producciones diferentes, a esto se le denomina conflicto `reduce/reduce`. Normalmente, si existe uno o más de estos conflictos, el `parser` aborta su operación. Para estos casos, que suelen ser muy arbitrarios, CUP hace uso de las convenciones de YACC para darle solución a estos conflictos, tomando aquellas producciones con la más alta prioridad de producción. Por ello, con esta instrucción, es posible indicar cuantos conflictos pueden llegar a esperarse.
- `-compact_red`: Permite la optimización y reducción de la tabla de reducciones, la cual, en ocasiones suele poseer grandes dimensiones. Permitiendo compactar los errores en un solo campo.
- `-nowarn`: Inhabilita los mensajes de advertencia, que pueden llegar a ser generados por el `parser`.
- `-nosummary`: Normalmente, al generar el `parser`, se genera un resumen de salida, esta opción inhabilita la generación de dicho resumen.
- `-progress`: Esta opción permite obtener diversos mensajes sobre el estatus, etapas y progreso al generar el `parser`.
- `-dump_grammar`.
- `-dump_states`.
- `-dump_tables`.
- `-dump`: Habilita la generación de un vocablo legible de la gramática, los estados de construcción y las tablas del `parser` respectivamente.

- `-time`: Agrega detalles de las estadísticas del tiempo dentro del resumen de resultados.
- `-debug`: Produce toda la información interna sobre el debugging.
- `-nopositions`: Evita que CUP genere código de propagación de izquierda a derecha, de los terminales y no-terminales, si es que no van a ser utilizados. Salvando ciclos de ejecución dentro del computador.
- `-locations`: Permite la generación de accesos `xleft/xright` para el acceso a los objetos `Location` desde los símbolos `start/end` dentro de las acciones.
- `-xmlactions`: Permite la generación de acciones que producen elementos XML por cada símbolo.
- `-genericlabels`: Esta opción genera un archivo completo XML, es una versión extendida de `-xmlactions`.
- `-version`: Imprime la versión de CUP que se esté utilizando.

Posterior a definir las opciones existentes y validas dentro de CUP, se utilizará dentro de este ejemplo `-interface` y `-parser` donde se renombrará la clase `Parser` a `ParserSintactico`. De esta forma, y al ejecutar las operaciones se obtiene la siguiente salida, que es mostrada en la **Figura 67**.

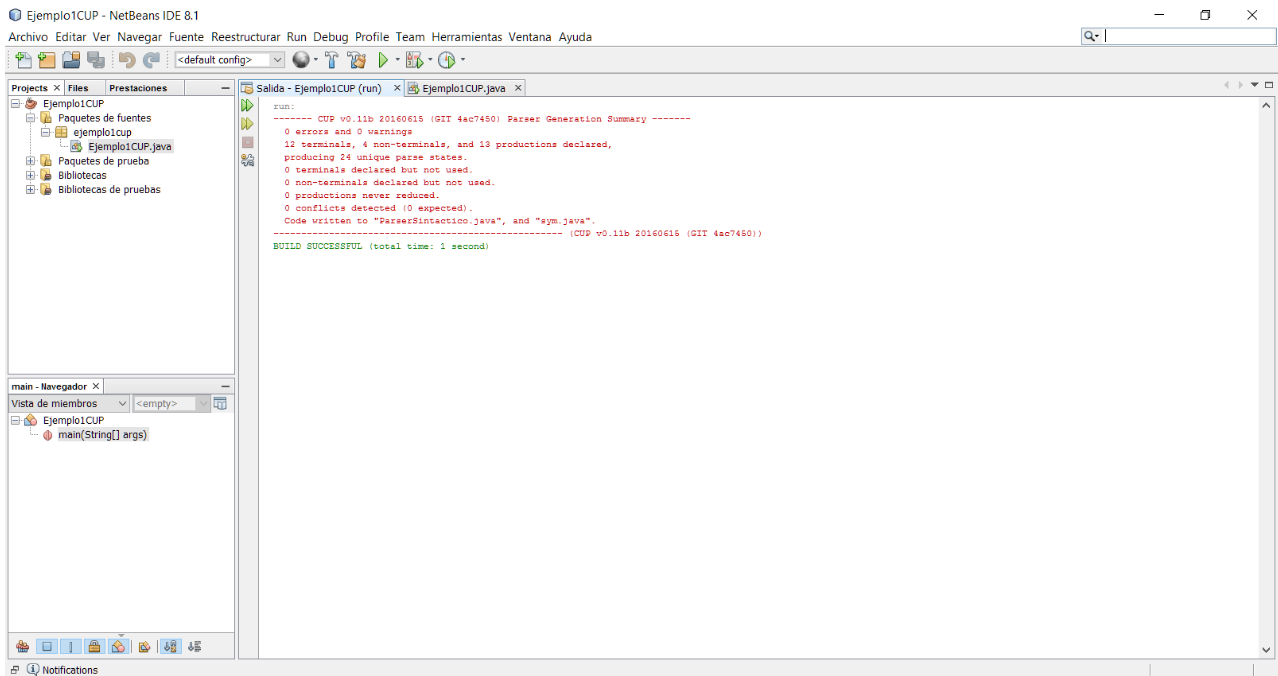


Figura 67. Resultado de la ejecución de CUP sobre el archivo `Parser.cup`.

De esta forma se generan las dos clases anteriormente mencionadas, dentro de la **Figura 68**.

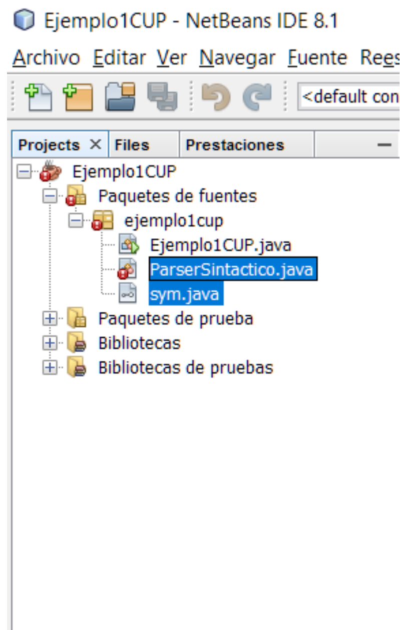


Figura 68. Clases generadas por CUP.

Para este ejemplo, se generará una clase *scanner*, la cual es muy básica para el análisis lexicográfico y de esta forma ir pasando cada *token* hacia el analizador sintáctico. La **Figura 69** muestra el contenido de la clase `scanner`.

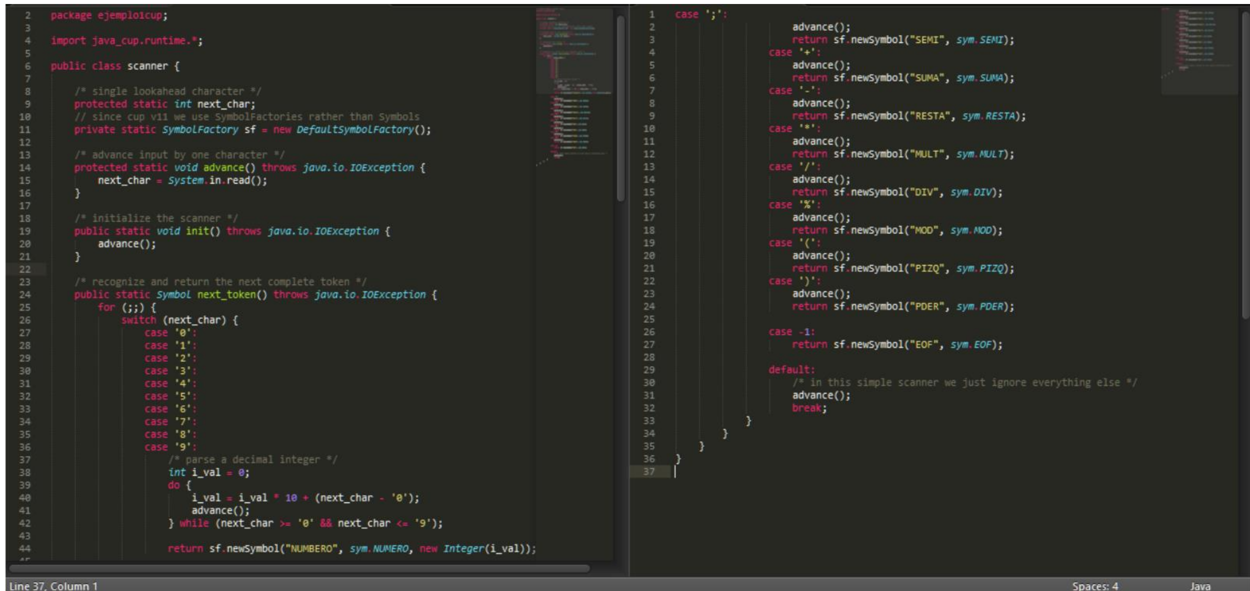


Figura 69. Clase `scanner` para el análisis lexicográfico.

De esta forma y teniendo nuestro *scanner*, comenzara la ejecución, donde se irán introduciendo los valores deseados para poder verificar el funcionamiento del `parser`. La **Figura 70** muestra el resultado de algunos ejemplos que fueron introducidos al `parser`, como es posible apreciarse, es posible introducir operaciones muy básicas a algunas un poco más complejas.

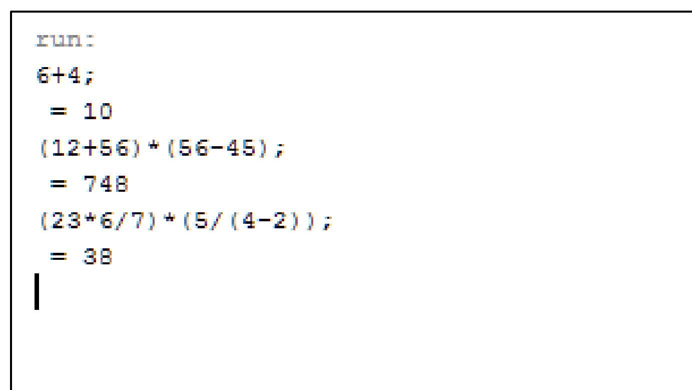


Figura 70. Resultado del `parser` tras introducir diversos elementos.

4.5. INTEGRACIÓN JFLEX Y CUP

Como ya se ha comentado en secciones anteriores, JFLex y CUP cuenta con la característica principal de poder integrarse entre ellos, así mismo el poder integrar sus funcionalidades con algún otro metacompilador, pero por parte de JFLex, principalmente con CUP.

Para continuar con la integración entre ambos metacompiladores, se va a manejar el ejemplo planteado de la calculadora, a modo de establecer las bases de integración y funcionalidad de ambos entornos de una forma práctica y sencilla, intentando abarcar en su mayoría el gran alcance con el que cuentan ambas herramientas. Por ello la gramática es la siguiente:

- Deben permitirse operaciones aritméticas como lo son: suma (+), resta (-), multiplicación (*), división (/) y modulo (%).
- Operaciones entre números naturales.
- Los saltos de línea, tabuladores o saltos de página deberán ser ignorados.
- Agrupación de operaciones por medio de paréntesis.
- La operación finaliza con una punto y coma ";".

Una vez establecidas las reglas a cumplir en la gramática, es necesario definir los archivos necesarios con los que se alimentaran JFLex y CUP, pero sin olvidar la gramática base que ambos utilizaran. La gramática se lista a continuación.

$$exp_list := exp_list\ exp_part \mid exp_part$$

$$exp_part := exp\ ;$$

$$exp := (\ exp\ )$$

$$\mid exp + exp$$

$$\mid exp - exp$$

$$\mid exp * exp$$

$$\mid exp\ /\ exp$$

$$\mid exp\ \%\ exp$$

$$\mid [0-9]$$

Posteriormente a obtener nuestra gramática, se procede a generar el archivo de configuración para JFLex, el cual será nombrado como `AnalizadorLexicografico.flex`. Dentro de la primera sección del archivo, se definirá el paquete al cual estará integrada la clase del analizador lexicográfico y en conjunto, serás integradas dos librerías, `java_cup.runtime` para la lectura de los símbolos y `java.io.Reader` para la lectura de ficheros. La **Figura 71** muestra como quedara la primera sección.

```
/* -------------------------Codigo de Usuario----------------------- */
package ejemplojflexcup;

import java_cup.runtime.*;
import java.io.Reader;
```

Figura 71. Definición de la sección del código del usuario, dentro del archivo de configuración para JFLex.

Continuando con los pasos ya mencionados, se declara la sección de las opciones y declaraciones de JFLex, donde se hará uso de una configuración propia de JFLex que ya fue mencionada con anterioridad, y esta permitirá la integración con CUP. De igual forma se habilitará el conteo de líneas por fila y columna, así como un par de métodos que permitirán la lectura de los símbolos entrantes, ya sea con un valor o sin él, y retornarlos en un objeto del tipo `Symbol`. Por último, serán definidas las macros pertenecientes a los saltos de línea, tabuladores y números naturales. La **Figura 72** muestra el contenido para esta segunda sección.

Para la última sección, se integrarán aquellas reglas y acciones válidas para las expresiones regulares partiendo desde el inicio de cada *token*, para ello se hará uso de `YYINITIAL`. Posterior al encontrarlos, será impreso el *token* encontrado en pantalla. Como se mencionó con anterioridad, los espacios y saltos de línea serán ignorados. Para la parte de errores, simplemente se mostrará en pantalla aquel *token* que sea ilegal. La **Figura 73** muestra la configuración completa de esta última sección.

```
/* ------ Seccion de opciones y declaraciones de JFlex -------------- */
%% //inicio de opciones

/*
    Se cambia el nombre de la clase a AnalizadorLexicografico
*/
%class AnalizadorLexicografico

/*
    Se activa el contador de lineas, yyline
    Se activa el contador de columnas, yycolumn
*/
%line
%column

/*
    Se activa la compatibilidad con Java CUP para analizadores sintacticos
*/
%cup

/*
    El codigo entre %{ %} sera copiado integramente en el analizador generado.
*/
%{
    /*
        Se generara un objeto java_cup.Symbol para guardar el tipo de token encontrado
    */
    private Symbol symbol(int type) {
        return new Symbol(type, yyline, yycolumn);
    }

    /*
        Se generara un objeto java_cup.Symbol para el tipo de token encontrado junto con su valor
    */
    private Symbol symbol(int type, Object value) {
        return new Symbol(type, yyline, yycolumn, value);
    }
%}

/*
    Macros

    Declaracion de expresiones regulares que despues seran usadas en las reglas lexicograficas.
*/

/*
    Salto de linea: \n, \r o \r\n dependiendo del SO
*/
Salto = \r|\n|\r\n

/*
    Espacio en blanco, tabulador \t, salto de linea o avance de pagina \f, normalmente son ignorados
*/
Espacio     = {Salto} | [ \t\f]

/*
    Numero 0, o un digito del 1 al 9 seguido de 0 o mas digitos del 0 al 9
*/
Entero = 0 | [1-9][0-9]*

%% //fin de opciones
```

Figura 72. Definición de macros y opciones de JFLex dentro del archivo `AnalizadorLexicografico.flex`.

```
/* ------------------- Seccion de reglas lexicas ----------------- */

/*
  Esta seccion contiene expresiones regulares y acciones.
  Las acciones son código en Java que se ejecutara cuando se encuentre una entrada valida para la expresion regular correspondiente.
*/

/*
    YYINITIAL es el estado inicial del analizador lexicografico al empezar el escaneo.
    Las expresiones regulares solo serán comparadas si se encuentra en ese estado inicial.
    Es decir, cada vez que se encuentra una coincidencia el scanner vuelve al estado inicial.
    Por lo cual se ignoran estados intermedios.
*/

<YYINITIAL> {

    /*
        Retorna que el token SEMI, declarado en la clase sym, fue encontrado.
    */
    ";" {
        return symbol(sym.SEMI);
    }

    /*
        Retorna que el token OP_SUMA, declarado en la clase sym, fue encontrado.
    */
    "+" {
        System.out.print(" + ");
        return symbol(sym.OP_SUMA);
    }

    /*
        Retorna que el token OP_RESTA, declarado en la clase sym, fue encontrado.
    */
    "-" {
        System.out.print(" - ");
        return symbol(sym.OP_RESTA);
    }

    /*
        Retorna que el token OP_MULT, declarado en la clase sym, fue encontrado.
    */
    "*" {
        System.out.print(" * ");
        return symbol(sym.OP_MULT);
    }

    /*
        Retorna que el token OP_DIV, declarado en la clase sym, fue encontrado.
    */
    "/" {
        System.out.print(" / ");
        return symbol(sym.OP_DIV);
    }

    /*
        Retorna que el token OP_MOD, declarado en la clase sym, fue encontrado.
    */
    "%" {
        System.out.print(" % ");
        return symbol(sym.OP_MOD);
    }

    /*
        Retorna que el token PARIZQ, declarado en la clase sym, fue encontrado.
    */
    "(" {
        System.out.print(" ( ");
        return symbol(sym.PARIZQ);
    }

    /*
        Regresa que el token PARDER, declarado en la clase sym, fue encontrado.
    */
    ")" {
        System.out.print(" ) ");
        return symbol(sym.PARDER);
    }

    /*
        Si se encuentra un Numero, se imprime, se regresa un token del tipo Integer que representa un Numero y el valor que se obtuvo
        de la cadena yytext al convertirla a Numero. yytext es el token encontrado.
    */
    {Numero} {
        System.out.print(yytext());
        return symbol(sym.ENTERO, new Integer(yytext()));
    }

    /*
        No hace nada si encuentra el espacio en blanco
    */
    {Espacio} {
        /* ignora el espacio */
    }
}
/*
    Si el token contenido en la entrada no coincide con ninguna regla entonces se marca un token ilegal
*/
[^] {
    throw new Error("Caracter ilegal <"+yytext()+">");
}
```

Figura 73. Expresiones regulares que detectara JFLex dentro de cada *token*.

Al tener finalizado el archivo de configuración por parte de JFLex, ahora, es necesario configurar el archivo que permitirá a CUP identificar cada *token* y así mismo, realizar las acciones necesarias y deseadas por cada coincidencia encontrada. El archivo será nombrado `AnalizadorSintactico.cup`, dentro de la primera sección, se declara el paquete al cual pertenecerá el analizador sintáctico, así como la importación de las librerías que permitirán la lectura de un fichero e

identificarán los símbolos generados por CUP. La **Figura 74** muestra el contenido de la sección.

```
/* ---------------Seccion de Importacion y Empaquetamiento--------------------*/
package ejemplojflexcup;

/*
    Import the class java_cup.runtime.*
*/
import java_cup.runtime.*;
import java.io.FileReader;
```

Figura 74. Configuración de la importación de librerías y definición de paquetes para el alojamiento de las clases generadas por CUP.

Dentro de la sección de código de usuario, se agregarán tres métodos, dos para el manejo de errores y uno que será la unión de JFLex con CUP permitiendo la ejecución del análisis lexicográfico para posteriormente el análisis sintáctico con CUP. Dentro de los métodos para el manejo de errores, uno de los métodos, informa solamente sobre el error encontrado y posterior a ello, continuar si es posible, el otro método, informa sobre aquellos errores de los cuales al sistema no le es posible recuperarse, donde finaliza la ejecución del análisis. La **Figura 75** muestra el contenido previamente mencionado y como se interconectan ambos metacompiladores.

La sección de procedencias se omitirá, ya que, no es funcional para este ejemplo, de igual forma se definirán aquellos terminales y no-terminales y si poseen algún tipo de objeto o no. La **Figura 76** muestra lo mencionado.

Por último, se declara la sección de la gramática, la cual ejecutara las funciones en específico, según sea el caso que encuentre la gramática y que se cumpla. En la **Figura 77** puede apreciarse la gramática y la acción a ejecutar en cada caso.

```
/* ---------------Seccion de Codigo de Usuario--------------------*/

/*
    Codigo del parser, se copia integramente a la clase final.
    Agregamos el manejo de errores.
*/
parser code {:

    /*
        Reporte de error encontrado.
    */
    public void report_error(String message, Object info) {
        StringBuilder sb = new StringBuilder("Error");
        if (info instanceof java_cup.runtime.Symbol) {
            java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol) info);
            if (s.left >= 0) {
                sb.append(" in line "+(s.left+1));
                if (s.right >= 0) {
                    sb.append(", column "+(s.right+1));
                }
            }
        }
        sb.append(" : "+message);
        System.err.println(sb);
    }

    /*
        Cuando se encuentra un error donde el sistema no puede recuperarse, se lanza un error fatal. Se despliega el mensaje de error y se finaliza la ejecucion.
    */
    public void report_fatal_error(String message, Object info) {
        report_error(message, info);
        System.exit(1);
    }

    /*
        Metodo main para garantizar la ejecucion del analizador lexico y sintactico, ademas que se pase como parametro la tabla de simbolos correspondiente.
    */
    public static void main(String[] args) {
        try {
            AnalizadorSintactico asin = new AnalizadorSintactico(new AnalizadorLexicografico( new FileReader(args[0])));
            Object result = asin.parse().value;
            System.out.println("\n*** Resultados finales ***");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
:};
```

Figura 75. Código insertado por el usuario para la ejecución y manejo de errores.

```
/* ------------Declaracion de Simbolos Terminales y No Terminales---------- */

/*
    Terminales (tokens obtenidos por el analizador lexico).
    Terminales que no tienen un valor son listados primero, los terminales que tienen un valor como los enteros son listados en la segunda o demas lineas.
*/
terminal           SEMI, OP_SUMA, OP_RESTA, OP_MULT, OP_DIV, OP_MOD, PARIZQ, PARDER;
terminal Integer   NUMERO;

/*
    No terminales usados en la seccion gramatical.
    Primero se lista los no terminales que tienen un valor Object y despues se lista los no terminales que tienen un entero.
    Un Object se refiere a que no tienen tipo, pudiendo ser entero o String.
*/
non terminal Object     expr_list, expr_part;
non terminal Integer    expr, factor, termino;


/* ------------Seccion de Predencia y Asociatividad---------- */

/*
    Precedencia de los no terminales, no sirve con simbolos terminales.
    Por eso no la usamos.
    Además indica si se asocia a izquierda o derecha.
*/
```

Figura 76. Declaración de los terminales y los no-terminales dentro del archivo de configuración para CUP.

```
/* ------------------ Seccion de la Gramatica ----------------------- */

/*
    La gramatica de nuestro analizador.

   expr_list ::= expr_list expr_part |
                 expr_part
   expr_part ::= expr ;
   expr      ::= ( expr ) |
                 expr + expr |
                 expr - expr |
                 expr * expr |
                 expr / expr |
                 expr % expr |
                 [0-9]

*/
expr_list ::= expr_list expr_part | expr_part;

expr_part ::= expr : e {: System.out.println(" = " + e); :} SEMI;

expr      ::= expr : e OP_SUMA  factor : f {: RESULT = new Integer(e.intValue() + f.intValue()); :} |
              expr : e OP_RESTA factor : f {: RESULT = new Integer(e.intValue() - f.intValue()); :} |
              factor : f                   {: RESULT = f; :};

factor    ::= factor : f OP_MULT  termino : t {: RESULT = new Integer(f.intValue() * t.intValue()); :} |
              factor : f OP_DIV   termino : t {: RESULT = new Integer(f.intValue() / t.intValue()); :} |
              factor : f OP_MOD   termino : t {: RESULT = new Integer(f.intValue() % t.intValue()); :} |
              termino : t                     {: RESULT = t; :};

termino   ::= PARIZQ expr : e PARDER          {: RESULT = e; :} |
              NUMERO : n                      {: RESULT = n; :};
```

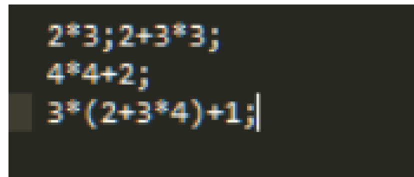
Figura 77. Definición de la gramática y las acciones a ejecutar en cada caso.

Una vez que se tienen ambos archivos para la configuración de los metacompiladores que se están manejando, es necesario generar el proyecto donde serán utilizados e integrar las librerías necesarias para su buen funcionamiento. La **Figura 78** muestra el código para poder llevar a cabo la generación de las clases de los metacompiladores.

```
package ejemplojflexcup;

public class EjemploJFLexCUP {

    public static void main(String[] args) {
        generarClasesMetacompilador();
    }

    public static void generarClasesMetacompilador() {
        String[] alexico = {"AnalizadorLexicografico.flex"};
        String[] asintactico = {"-parser", "AnalizadorSintactico", "AnalizadorSintactico.cup"};
        jflex.Main.main(alexico);
        try {
            java_cup.Main.main(asintactico);
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

Figura 78. Código para la generación de las clases del analizador lexicográfico y sintáctico.

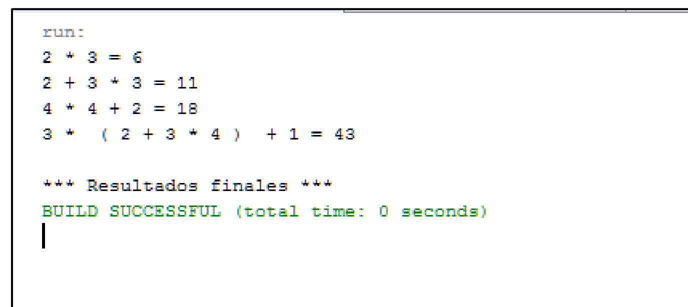
Al tener generadas las clases, es necesario tener un archivo para realizar las debidas pruebas al compilador que se acaba de generar. El archivo `test.txt` que se muestra en la **Figura 79** contiene alguno de los casos que serán introducidos para obtener los debidos resultados.



```
2*3;2+3*3;
4*4+2;
3*(2+3*4)+1;
```

Figura 79. Operaciones que se introducirán al compilador para realizar las pruebas necesarias.

La **Figura 80** muestra los resultados obtenidos tras la ejecución.



```
run:
2 * 3 = 6
2 + 3 * 3 = 11
4 * 4 + 2 = 18
3 *  ( 2 + 3 * 4 )  + 1 = 43

*** Resultados finales ***
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 80. Resultados obtenidos del archivo `test.txt`.

4.6. CASO DE USO JFLEX Y CUP

Como ya se logró observar con anterioridad, la integración de JFLex y CUP llega a ser una herramienta muy potente para la generación de metacompiladores. En este caso se implementarán para la generación de un pequeño compilador perteneciente a un lenguaje de programación. La gramática es sencilla e intuitiva ya que solo permite números enteros y operaciones aritméticas muy básicas, en conjunto del uso de un loop's y bifurcaciones. La gramática a utilizar se muestra en la **Figura 81**.

```
PROGRAM             ::= procedure HEADER BODY
HEADER              ::= IDENTIFIER is
BODY                ::= DECLARATIONS BLOCK
DECLARATIONS        ::= IDENTIFIER : Integer ; |
                        IDENTIFIER : Integer ; DECLARATIONS
BLOCK               ::= begin STATEMENTS end;
STATEMENTS          ::= STATEMENT |
                        STATEMENT STATEMENTS
STATEMENTS          ::= ASSIGMENT_STATEMENT |
                        FOR_STATEMENT       |
                        IF_STATEMENT        |
                        INPUT_STATEMENT     |
                        OUTPUT_STATEMENT    |
                        NULL_STATEMENT
ASSIGMENT_STATEMENT ::= IDENTIFIER := EXPRESSION;
FOR_STATEMENT       ::= for IDENTIFIER in CONSTANT to CONSTANT
                        loop
                            STATEMENTS
                        endloop
IF_STATEMENT        ::= if EXPRESSION then
                            STATEMENTS
                        else
                            STATEMENTS
                        endif
INPUT_STATEMENT     ::= input(IDENTIFIER)
OUTPUT_STATEMENT    ::= output(IDENTIFIER)
NULL_STATEMENT      ::= null
EXPRESSION          ::= TERMN |
                        EXPRESSION ADD_OP TERMN
TERMN               ::= FACTOR |
                        TERMN MULT_OP FACTOR
FACTOR              ::= IDENTIFIER |
                        (EXPRESSION) |
                        CONSTANT
IDENTIFIER          ::= a | b | c | ... | z
ADD_OP              ::= + | -
MULT_OP             ::= * | /
CONSTANT            ::= DIGIT |
                        DIGIT CONSTANT
DIGIT               ::= 0 | 1 | 2 | ... | 9
```

Figura 81. Gramática de la estructura básica de un lenguaje de programación basado en Pascal.

Como es posible de apreciar, la gramática, permite el ingreso y salida de datos, bucles, toma de decisiones en base a un resultado donde si el resultado es diferente a 0 se procederá a ejecutar las sentencias encasilladas dentro de la sección, asignación de valores a variables y, de igual forma, operaciones aritméticas con las mismas variables o entre dígitos. Posterior a comprender la gramática, es necesario establecer su funcionalidad dentro de CUP y JFLex.

Las figuras **Figura 82**, **Figura 83** y **Figura 84** poseen el contenido que dará acción al metacompilador según las reglas establecidas en la gramática. Así mismo,

contendrá el método principal para lograr su ejecución y la integración con las diversas operaciones que serán desarrolladas para facilitar la ejecución de características propias de esta gramática.

```
/* ---------------Seccion de Importacion y Empaquetamiento-------------------*/
package com.uaem.main;
/*
    Import the class java_cup.runtime.*
*/
import java_cup.runtime.*;
import java.io.FileReader;
import com.uaem.util.*;
import com.uaem.classes.*;

/* ---------------Seccion de Codigo de Usuario-------------------*/

/*
    Codigo del parser, se copia integramente a la clase final.
    Agregamos el manejo de errores.
*/
parser code {:

    /*
        Reporte de error encontrado.
    */
    public void report_error(String message, Object info) {
        StringBuilder sb = new StringBuilder("Error");
        if (info instanceof java_cup.runtime.Symbol) {
            java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol) info);
            if (s.left >= 0) {
                sb.append(" in line "+(s.left));
                if (s.right >= 0) {
                    sb.append(", column "+(s.right));
                }
            }
        }
        sb.append(" : "+message);
        System.err.println(sb);
    }

    /*
        Cuando se encuentra un error donde el sistema no puede recuperarse, se lanza un error fatal. Se despliega el mensaje de error y se finaliza la ejecucion.
    */
    public void report_fatal_error(String message, Object info) {
        report_error(message, info);
        System.exit(1);
    }

    /*
        Metodo main para garantizar la ejecucion del analizador lexico y sintactico, ademas que se pase como parametro la tabla de simbolos correspondiente.
    */
    public static void main(String[] args) {
        try {
            AnalizadorSintactico asin = new AnalizadorSintactico(new AnalizadorLexicografico( new FileReader(args[0])));
            Block result = (Block) asin.parse().value;
            result.doAction();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
:};
```

Figura 82. Declaración para el manejo de errores y funciones de ejecución del analizador sintáctico.

```
/* ------------Declaracion de Simbolos Terminales y No Terminales---------- */

/*
    Terminales (tokens obtenidos por el analizador lexico).
    Terminales que no tienen un valor son listados primero, los terminales que tienen un valor como los enteros son listados en la segunda o demas lineas.
*/
terminal Token      PROCEDURE, IS, INTEGER, TWOP, SEMI, BEGIN, END, EQUAL, FOR, IN, TO, LOOP, ENDLOOP,
                    IF, THEN, ELSE, ENDIF, INPUT, OUTPUT, PIZQ, PDER, NULL, OP_ADD, OP_SUB,
                    OP_MULT, OP_DIV, DIGITO, IDENTIFIER;

/*
    No terminales usados en la seccion gramatical.
    Primero se lista los no terminales que tienen un valor Object y despues se lista los no terminales que tienen un entero.
    Un Object se refiere a que no tienen tipo, pudiendo ser entero o String.
*/
non terminal Object        program, header, body, declarations;
non terminal Block         block;
non terminal StatementList statements;
non terminal Statement     statement, assigment_statement, for_statement, if_statement, input_statement, output_statement, null_statement;
non terminal Identifier    identifier;
non terminal Expression    expression;
non terminal Digit         constant;

/* -------------Seccion de Predencia y Asociatividad----------- */

/*
    Precedencia de los no terminales, no sirve con simbolos terminales.
    Por eso no la usamos.
    Además indica si se asocia a izquierda o derecha.
*/
precedence left  OP_ADD, OP_SUB;
precedence left  OP_MULT, OP_DIV;
```

Figura 83. Declaración de terminales y no-terminales y las procedencias de las operaciones a realizar.

```
92   /* ------------------- Seccion de la Gramatica ------------------------ */
93
94   program                  ::= PROCEDURE header body: bd
95                                {:
96                                  RESULT = bd;
97                                :};
98   header                   ::= identifier IS;
99   body                     ::= declarations block: blck
100                               {:
101                                 RESULT = blck;
102                               :};
103  declarations             ::= identifier TWOP INTEGER SEMI |
104                               identifier TWOP INTEGER SEMI declarations;
105  block                    ::= BEGIN statements: stms END SEMI
106                               {:
107                                 RESULT = new Block(stms);
108                               :};
109  statements               ::= statement: stm
110                               {:
111                                 StatementList stmList = new StatementList();
112
113                                 stmList.addElement(stm);
114
115                                 RESULT = stmList;
116                               :}  |
117                               statement: stm statements: stms
118                               {:
119                                 if (stms == null) {
120                                   stms = new StatementList();
121                                 }
122
123                                 stms.addElement(stm);
124
125                                 RESULT = stms;
126                               :};
127  statement                ::= assigment_statement: astm
128                               {:
129                                 RESULT = astm;
130                               :} |
131                               null_statement: nstm
132                               {:
133                                 RESULT = nstm;
134                               :}  |
135                               for_statement: fstm
136                               {:
137                                 RESULT = fstm;
138                               :} |
139                               if_statement: istm
140                               {:
141                                 RESULT = istm;
142                               :} |
143                               input_statement: instm
144                               {:
145                                 RESULT = instm;
146                               :} |
147                               output_statement: ostmn
148                               {:
149                                 RESULT = ostmn;
150                               :};
151  assigment_statement      ::= identifier : id EQUAL: eq expression : ex SEMI
152                               {:
153                                 RESULT = new AssigmentStm(id, ex, eq.getLine(), eq.getColumn());
154                               :};
155  if_statement             ::= IF: ifStm expression: ex THEN statements: stm1 ELSE statements: stm2 ENDIF
156                               {:
157                                 RESULT = new IfStm(ex, stm1, stm2, ifStm.getLine(), ifStm.getColumn());
158                               :};
159  for_statement            ::= FOR: forStm identifier: id IN constant: cons1 TO constant: cons2 LOOP statements: stm ENDLOOP
160                               {:
161                                 RESULT = new ForStm(id, cons1, cons2, stm, forStm.getLine(), forStm.getColumn());
162                               :};
163  input_statement          ::= INPUT PIZQ identifier: id PDER SEMI
164                               {:
165                                 RESULT = new InputStm(id, id.getLine(), id.getColumn());
166                               :};
167  output_statement         ::= OUTPUT PIZQ identifier : id PDER SEMI
168                               {:
169                                 RESULT = new OutputStm(id, id.getLine(), id.getColumn());
170                               :};
171  null_statement           ::= NULL: nullStm SEMI
172                               {:
173                                 RESULT = new NullStm(nullStm.getLine(), nullStm.getColumn());
174                               :};
175  expression               ::= expression: ex1 OP_ADD: add expression: ex2
176                               {:
177                                 RESULT = new ArithmeticOperationExpression(ex1, ex2, OperationType.ADD, add.getLine(), add.getColumn());
178                               :} |
179                               expression: ex1 OP_SUB: sub expression: ex2
180                               {:
181                                 RESULT = new ArithmeticOperationExpression(ex1, ex2, OperationType.SUB, sub.getLine(), sub.getColumn());
182                               :} |
183                               expression: ex1 OP_MULT: mult expression: ex2
184                               {:
185                                 RESULT = new ArithmeticOperationExpression(ex1, ex2, OperationType.MULT, mult.getLine(), mult.getColumn());
186                               :} |
187                               expression: ex1 OP_DIV: div expression: ex2
188                               {:
189                                 RESULT = new ArithmeticOperationExpression(ex1, ex2, OperationType.DIV, div.getLine(), div.getColumn());
190                               :} |
191                               PIZQ  expression: ex  PDER
192                               {:
193                                 RESULT = ex;
194                               :} |
195                               constant: cons
196                               {:
197                                 RESULT = new DigitExp(cons, cons.getLine(), cons.getColumn());
198                               :} |
199                               identifier: id
200                               {:
201                                 RESULT = new IdentifierExp(id, id.getLine(), id.getColumn());
202                               :};
203  constant                 ::= DIGITO: d
204                               {:
205                                 RESULT = new Digit((Integer) d.getValue(), d.getLine(), d.getColumn());
206                               :};
207  identifier               ::= IDENTIFIER : id
208                               {:
209                                 RESULT = new Identifier((String) id.getValue(), id.getLine(), id.getColumn());
210                               :};
211
```

Figura 84. Declaración de la gramática y su funcionalidad en base a la gramática establecida con anterioridad.

Con lo anterior podemos finalizar la definición del archivo perteneciente al analizador sintáctico que se denominara `Sintactico.cup`. Dentro de este archivo fue necesario alterar un poco la gramática para poder agilizar la formación del árbol sintáctico y permitirnos manipular más eficientemente el desarrollo de los componentes que llevaran a cabo la tarea de ejecutar el metacompilador.

La principal alteración que se efectuó a la gramática fue el acortamiento de las derivaciones que permiten las operaciones aritméticas, ya que por medio de los componentes a desarrollar se logró minimizar la generación de derivaciones dentro del metacompilador; con lo anteriormente mencionado, se logra agilizar el análisis de los datos a introducir e implementar algunas de las funciones que nos proporciona JFLex como lo son las procedencias que nos ayudan a evitar ambigüedades dentro del árbol de derivaciones al establecer que camino es necesario seguir al encontrar alguna de las reglas establecidas dentro de las procedencias, tal y como se comentó en segmentos anteriores.

Una vez concluido nuestro analizador sintáctico, es necesario definir el analizador lexicográfico dentro de JFLex. El contenido del analizador lexicográfico puede visualizarse en las figuras **Figura 85**, **Figura 86** y **Figura 87**, donde se define la integración con CUP, las expresiones regulares que darán paso a las palabras reservadas y permitirán definir las operaciones e integrar los resultados con CUP al encontrar estas expresiones dentro del archivo que se denominara `Lexicografico.flex`.

Cómo es posible observar en la **Figura 82**, el analizador lexicográfico, y como se ha venido explicando en las secciones anteriores, retorna un objeto de tipo Scanner, el cual es enviado al analizador sintáctico al finalizar la validación del contenido existente en el archivo de texto que contendrá los casos que nos permitirán realizar las pruebas necesarias.

```
/* ------------------------Codigo de Usuario--------------------- */
package com.uaem.main;

import java_cup.runtime.*;
import java.io.Reader;
import com.uaem.classes.Token;

/* ------ Seccion de opciones y declaraciones de JFlex ------------- */
%% //inicio de opciones

/*
    Se cambia el nombre de la clase a AnalizadorLexicografico
*/
%class AnalizadorLexicografico

/*
    Se activa el contador de lineas, yyline
    Se activa el contador de columnas, yycolumn
*/
%line
%column

/*
    Se activa la compatibilidad con Java CUP para analizadores sintacticos
*/
%cup

/*
    El codigo entre %{ %} sera copiado integramente en el analizador generado.
*/
%{
    /*
        Se generara un objeto java_cup.Symbol para guardar el tipo de token encontrado
    */
    private Symbol symbol(int type) {
        return new Symbol(type, yyline, yycolumn);
    }

    /*
        Se generara un objeto java_cup.Symbol para el tipo de token encontrado junto con su valor
    */
    private Symbol symbol(int type, int yyline, int yycolumn, Object value) {
        return new Symbol(type, yyline, yycolumn, value);
    }
%}

/*
    Macros

    Declaracion de expresiones regulares que despues seran usadas en las reglas lexicograficas.
*/

/*
    Salto de linea: \n, \r o \r\n dependiendo del SO
*/
Salto = \r|\n|\r\n

/*
    Espacio en blanco, tabulador \t, salto de linea o avance de pagina \f, normalmente son ignorados
*/
Espacio = {Salto} | [ \t\f]

/*
    Numero entero
*/
Digito = 0 | [1-9][0-9]*

/*
    Identificador
*/
Identificador = [a-z]*

%% //fin de opciones
```

Figura 85. Definición del alojamiento de las clases que se generarán del analizador lexicográfico y la integración de CUP, en conjunto con la definición de las expresiones regulares que permitirán validar los datos de entrada.

```
/* -------------------- Seccion de reglas lexicas ------------------ */

/*
    Esta seccion contiene expresiones regulares y acciones.
    Las acciones son código en Java que se ejecutara cuando se encuentre una entrada valida para la expresion regular correspondiente.
*/

/*
    YYINITIAL es el estado inicial del analizador lexicografico al empezar el escaneo.
    Las expresiones regulares solo serán comparadas si se encuentra en ese estado inicial.
    Es decir, cada vez que se encuentra una coincidencia el scanner vuelve al estado inicial.
    Por lo cual se ignoran estados intermedios.
*/

<YYINITIAL> {

    /*
        Regresa que el token EQUAL, declarado en la clase sym, fue encontrado.
    */
    ":=" {
        return symbol(sym.EQUAL, yyline + 1, yycolumn + 1, new Token(":=", yyline + 1, yycolumn + 1));
    }

    /*
        Retorna que el token PROCEDURE, declarado en la clase sym, fue encontrado.
    */
    "procedure" {
        return symbol(sym.PROCEDURE, yyline + 1, yycolumn + 1, new Token("procedure", yyline + 1, yycolumn + 1));
    }

    /*
        Retorna que el token IS, declarado en la clase sym, fue encontrado.
    */
    "is" {
        return symbol(sym.IS, yyline + 1, yycolumn + 1, new Token("is", yyline + 1, yycolumn + 1));
    }

    /*
        Retorna que el token INTEGER, declarado en la clase sym, fue encontrado.
    */
    "Integer" {
        return symbol(sym.INTEGER, yyline + 1, yycolumn + 1, new Token("Integer", yyline + 1, yycolumn + 1));
    }

    /*
        Retorna que el token TWOP, declarado en la clase sym, fue encontrado.
    */
    ":" {
        return symbol(sym.TWOP, yyline + 1, yycolumn + 1, new Token(":", yyline + 1, yycolumn + 1));
    }

    /*
        Retorna que el token SEMI, declarado en la clase sym, fue encontrado.
    */
    ";" {
        return symbol(sym.SEMI, yyline + 1, yycolumn + 1, new Token(";", yyline + 1, yycolumn + 1));
    }

    /*
        Retorna que el token BEGIN, declarado en la clase sym, fue encontrado.
    */
    "begin" {
        return symbol(sym.BEGIN, yyline + 1, yycolumn + 1, new Token("begin", yyline + 1, yycolumn + 1));
    }

    /*
        Retorna que el token END, declarado en la clase sym, fue encontrado.
    */
    "end" {
        return symbol(sym.END, yyline + 1, yycolumn + 1, new Token("end", yyline + 1, yycolumn + 1));
    }

    /*
        Retorna que el token FOR, declarado en la clase sym, fue encontrado.
    */
    "for" {
        return symbol(sym.FOR, yyline + 1, yycolumn + 1, new Token("for", yyline + 1, yycolumn + 1));
    }

    /*
        Retorna que el token IN, declarado en la clase sym, fue encontrado.
    */
    "in" {
        return symbol(sym.IN, yyline + 1, yycolumn + 1, new Token("in", yyline + 1, yycolumn + 1));
    }

    /*
        Retorna que el token TO, declarado en la clase sym, fue encontrado.
    */
    "to" {
        return symbol(sym.TO, yyline + 1, yycolumn + 1, new Token("to", yyline + 1, yycolumn + 1));
    }

    /*
        Retorna que el token LOOP, declarado en la clase sym, fue encontrado.
    */
    "loop" {
        return symbol(sym.LOOP, yyline + 1, yycolumn + 1, new Token("loop", yyline + 1, yycolumn + 1));
    }

    /*
        Retorna que el token ENDLOOP, declarado en la clase sym, fue encontrado.
    */
    "endloop" {
        return symbol(sym.ENDLOOP, yyline + 1, yycolumn + 1, new Token("endloop", yyline + 1, yycolumn + 1));
    }

    /*
        Retorna que el token IF, declarado en la clase sym, fue encontrado.
    */
    "if" {
        return symbol(sym.IF, yyline + 1, yycolumn + 1, new Token("if", yyline + 1, yycolumn + 1));
    }
```

Figura 86. Definición de cada uno de los casos que se encontraran a lo largo del documento a analizar. Dentro de estas definiciones, se encuentran principalmente las palabras reservadas de la gramática y las respectivas operaciones que se realizaran al encontrar cada uno de estos casos.

```
    /*
    |   Retorna que el token THEN, declarado en la clase sym, fue encontrado.
    */
    "then" {
    |   return symbol(sym.THEN, yyline + 1, yycolumn + 1, new Token("then", yyline + 1, yycolumn + 1));
    }

    /*
    |   Retorna que el token ELSE, declarado en la clase sym, fue encontrado.
    */
    "else" {
    |   return symbol(sym.ELSE, yyline + 1, yycolumn + 1, new Token("else", yyline + 1, yycolumn + 1));
    }

    /*
    |   Retorna que el token ENDIF, declarado en la clase sym, fue encontrado.
    */
    "endif" {
    |   return symbol(sym.ENDIF, yyline + 1, yycolumn + 1, new Token("endif", yyline + 1, yycolumn + 1));
    }

    /*
    |   Retorna que el token INPUT, declarado en la clase sym, fue encontrado.
    */
    "input" {
    |   return symbol(sym.INPUT, yyline + 1, yycolumn + 1, new Token("input", yyline + 1, yycolumn + 1));
    }

    /*
    |   Retorna que el token OUTPUT, declarado en la clase sym, fue encontrado.
    */
    "output" {
    |   return symbol(sym.OUTPUT, yyline + 1, yycolumn + 1, new Token("output", yyline + 1, yycolumn + 1));
    }

    /*
    |   Retorna que el token PIZQ, declarado en la clase sym, fue encontrado.
    */
    "(" {
    |   return symbol(sym.PIZQ, yyline + 1, yycolumn + 1, new Token("(", yyline + 1, yycolumn + 1));
    }

    /*
    |   Retorna que el token PDER, declarado en la clase sym, fue encontrado.
    */
    ")" {
    |   return symbol(sym.PDER, yyline + 1, yycolumn + 1, new Token(")", yyline + 1, yycolumn + 1));
    }

    /*
    |   Retorna que el token NULL, declarado en la clase sym, fue encontrado.
    */
    "null" {
    |   return symbol(sym.NULL, yyline + 1, yycolumn + 1, new Token("null", yyline + 1, yycolumn + 1));
    }

    /*
    |   Retorna que el token OP_ADD, declarado en la clase sym, fue encontrado.
    */
    "+" {
    |   return symbol(sym.OP_ADD, yyline + 1, yycolumn + 1, new Token("+", yyline + 1, yycolumn + 1));
    }

    /*
    |   Retorna que el token OP_SUB, declarado en la clase sym, fue encontrado.
    */
    "-" {
    |   return symbol(sym.OP_SUB, yyline + 1, yycolumn + 1, new Token("-", yyline + 1, yycolumn + 1));
    }

    /*
    |   Retorna que el token OP_MULT, declarado en la clase sym, fue encontrado.
    */
    "*" {
    |   return symbol(sym.OP_MULT, yyline + 1, yycolumn + 1, new Token("*", yyline + 1, yycolumn + 1));
    }

    /*
    |   Retorna que el token OP_DIV, declarado en la clase sym, fue encontrado.
    */
    "/" {
    |   return symbol(sym.OP_DIV, yyline + 1, yycolumn + 1, new Token("/", yyline + 1, yycolumn + 1));
    }

    /*
    |   Si se encuentra un Numero, se imprime, se regresa un token del tipo Integer que representa un Numero y el valor que se obtuvo
    |   de la cadena yytext al convertirla a Numero. yytext es el token encontrado.
    */
    {Digito} {
    |   return symbol(sym.DIGITO, yyline + 1, yycolumn + 1, new Token(new Integer(Integer.parseInt(yytext())), yyline + 1, yycolumn + 1));
    }

    /*
    |   Retorna que el token IDENTIFIER, declarado en la clase sym, fue encontrado.
    */
    {Identificador} {
    |   return symbol(sym.IDENTIFIER, yyline + 1, yycolumn + 1, new Token(yytext(), yyline + 1, yycolumn + 1));
    }

    /*
    |   No hace nada si encuentra el espacio en blanco
    */
    {Espacio} {
    |   /* ignora el espacio */
    }
}

/*
|   Si el token contenido en la entrada no coincide con ninguna regla entonces se marca un token ilegal
*/
[^] {
|   throw new Error("Caracter ilegal <" + yytext() + ">");
}
```

Figura 87. Continuación de la sección presentada en la **Figura 86**.

Dentro de los módulos que se desarrollaron, se generaron dos clases abstractas que permitirán el manejo adecuado para los objetos presentes con las sentencias y las expresiones de las operaciones que incluyen los identificadores para las asignaciones. Lo anterior con el fin de poder encapsular cada una de las sentencias de forma que se genere internamente una pila de operaciones que permita la ejecución de sus respectivas acciones al finalizar el análisis sintáctico.

La idea de manejar una pila de operaciones fue tomada de la función actual de los compiladores, e intérpretes, pertenecientes a los principales lenguajes de programación, donde se forma una pila de operaciones y se tiene un heap de memoria que almacena las referencias a todas las instancias, o asignaciones, de cada variable existente durante la vida del programa en ejecución. De esta forma, y con la ayuda de JFLex, CUP y Java, fue posible generar un heap de memoria propio, apoyado en las funciones de almacenamiento y manejo de datos que proporciona Java.

El caso de prueba puede observarse en la **Figura 88**, donde existe la declaración de diversas variables sin la necesidad de crear instancias de la variable para poder hacer uso de ella a lo largo de la ejecución del programa, tal y como suele suceder en muchos lenguajes de programación, esto debido a que al tener como base Pascal se trabaja de esta forma, generando automáticamente instancias de las variables dentro del heap de memoria para poder ser utilizadas posteriormente. Dentro de este ejemplo se realizan diversas operaciones, intentando llegar a la mayor complejidad para validar, y verificar, el perfecto funcionamiento del compilador. Se integraron bifurcaciones, bucles, entre otras sentencias anidadas para poder demostrar, y comprobar, que las funciones desarrolladas para el compilador fueron bien establecidas para poder dar la funcionalidad esperada.

Dentro del compilador que se construyó, solo se permiten archivos con la extensión *.ps, una extensión que se decidió establecer para homologar la formalidad del compilador.

```
procedure a is
b: Integer;
begin
    null;

    a := 12;

    b := a;

    c := 7 + 5;
    d := 7 - 5;
    e := 7 * 5;
    f := 7 / 5;

    g := 5 + 6 - 7 * 8 / 9;

    h := (5 + 6 - 7) * (8 / 9);

    i := (5 + 6) - (7 * 8) / 9;

    if a then
        a := 5;
    else
        a := 6;
    endif

    for b in 5 to 10
    loop
        if a then
            a := 5;
        else
            a := 6;
        endif
        output(a);
        output (b);
    endloop

    for b in 5 to 10
    loop
        for c in 0 to 15
        loop
            output (c);
            output (b);
            if i then
                output(i);
            else
                output(h);
            endif
        endloop
    endloop

    input(b);
    output(b);
end;
```

Figura 88. Primer caso de prueba para validar la funcionalidad del metacompilador. Archivo test.ps.

Dentro del compilador desarrollado, se integraron nuevas funcionalidades para poder utilizarlo libremente en cualquier plataforma sin la necesidad de tener un kit de desarrollo o las librerías necesarias de JFLex o CUP ya que están integradas.

La **Figura 89** muestra la ejecución del empaquetado generado del compilador desarrollado. Como es posible apreciar, se integraron funciones muy básicas,

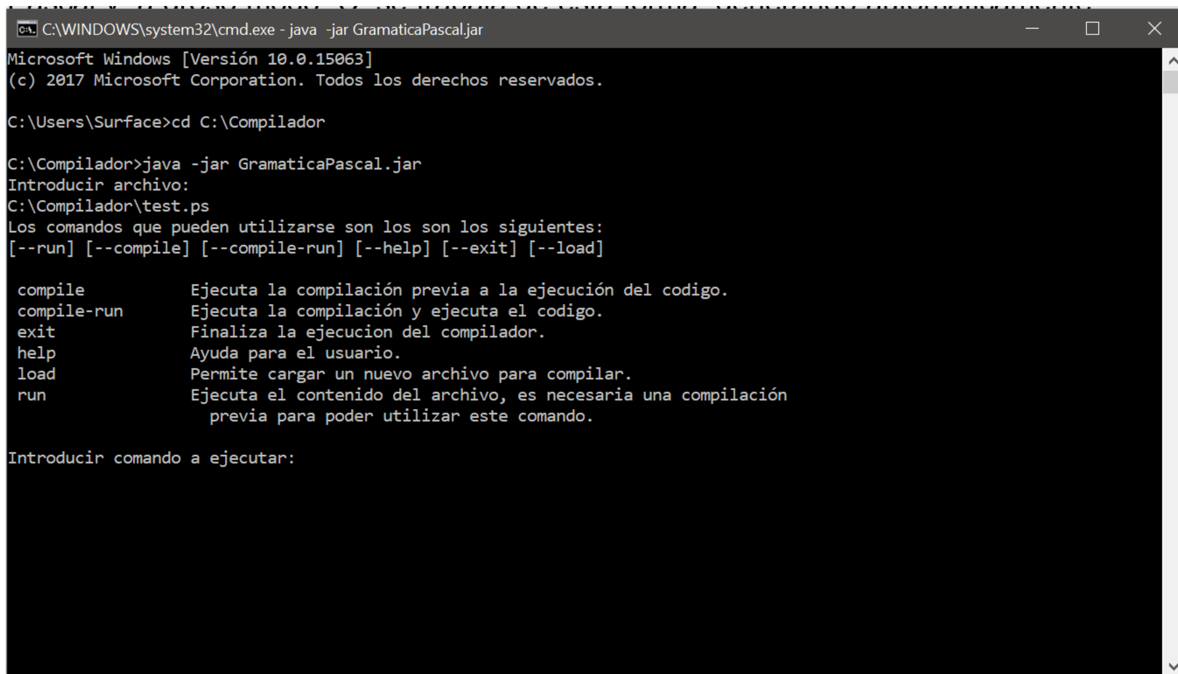


Figura 89. Ejecución del compilador desarrollado.

La **Figura 90**, **Figura 91**, **Figura 92** y **Figura 93** muestran parte del uso del compilador con el archivo de prueba test.ps. Muestran la carga del archivo, la compilación y ejecución por separado y la ayuda que se integró para el usuario.

La **Figura 94** muestra el contenido de un nuevo archivo para pruebas que será cargado al compilador, la **Figura 95** y la **Figura 96** muestran la carga, compilación y ejecución del nuevo archivo de prueba.

```
C:\WINDOWS\system32\cmd.exe - java -jar GramaticaPascal.jar
Microsoft Windows [Versión 10.0.15063]
(c) 2017 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Surface>cd C:\Compilador

C:\Compilador>java -jar GramaticaPascal.jar
Introducir archivo:
C:\Compilador\test.ps
Los comandos que pueden utilizarse son los son los siguientes:
[--run] [--compile] [--compile-run] [--help] [--exit] [--load]

 compile          Ejecuta la compilación previa a la ejecución del codigo.
 compile-run      Ejecuta la compilación y ejecuta el codigo.
 exit             Finaliza la ejecucion del compilador.
 help             Ayuda para el usuario.
 load             Permite cargar un nuevo archivo para compilar.
 run              Ejecuta el contenido del archivo, es necesaria una compilación
                    previa para poder utilizar este comando.

Introducir comando a ejecutar:
--compile
Compilación exitosa.

Introducir comando a ejecutar:
```

Figura 90. Carga y compilación del archivo test.ps dentro del compilador.

```
C:\WINDOWS\system32\cmd.exe - java -jar GramaticaPascal.jar
Microsoft Windows [Versión 10.0.15063]
(c) 2017 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Surface>cd C:\Compilador

C:\Compilador>java -jar GramaticaPascal.jar
Introducir archivo:
C:\Compilador\test.ps
Los comandos que pueden utilizarse son los son los siguientes:
[--run] [--compile] [--compile-run] [--help] [--exit] [--load]

 compile          Ejecuta la compilación previa a la ejecución del codigo.
 compile-run      Ejecuta la compilación y ejecuta el codigo.
 exit             Finaliza la ejecucion del compilador.
 help             Ayuda para el usuario.
 load             Permite cargar un nuevo archivo para compilar.
 run              Ejecuta el contenido del archivo, es necesaria una compilación
                    previa para poder utilizar este comando.

Introducir comando a ejecutar:
--compile
Compilación exitosa.

Introducir comando a ejecutar:
--run
5
5
5
6
5
```

Figura 91. Ejecución del contenido del archivo test.ps.

```
C:\WINDOWS\system32\cmd.exe - java -jar GramaticaPascal.jar
10
5
10
10
5
11
10
5
12
10
5
13
10
5
14
10
5
15
10
5
87
87


Finalizo ejecución.


Introducir comando a ejecutar:
```

Figura 92. Continuación de la ejecución del archivo test.ps.

```
C:\WINDOWS\system32\cmd.exe - java -jar GramaticaPascal.jar
10
5
14
10
5
15
10
5
87
87


Finalizo ejecución.


Introducir comando a ejecutar:
--help
Los comandos que pueden utilizarse son los son los siguientes:
[--run] [--compile] [--compile-run] [--help] [--exit] [--load]

 compile          Ejecuta la compilación previa a la ejecución del codigo.
 compile-run      Ejecuta la compilación y ejecuta el codigo.
 exit             Finaliza la ejecucion del compilador.
 help             Ayuda para el usuario.
 load             Permite cargar un nuevo archivo para compilar.
 run              Ejecuta el contenido del archivo, es necesaria una compilación
                    previa para poder utilizar este comando.

Introducir comando a ejecutar:
```

Figura 93. Muestra de la ayuda que se otorga al usuario dentro del uso del compilador.

```
procedure a is
b: Integer;
begin
    null;

    a := 12;

    b := a;

    c := 7 + 5;
    d := 7 - 5;
    e := 7 * 5;
    f := 7 / 5;

    g := 5 + 6 - 7 * 8 / 9;

    h := (5 + 6 - 7) * (8 / 9);

    i := (5 + 6) - (7 * 8) / 9;

    if a then
        a := 5;
    else
        a := 6;
    endif

    for b in 5 to 10
    loop
        if a then
            a := 5;
        else
            a := 6;
        endif
        output(a);
        output (b);
    endloop
end;
```

Figura 94. Contenido del nuevo archivo a compilar. Archivo test2.ps

```
C:\WINDOWS\system32\cmd.exe - java  -jar GramaticaPascal.jar
C:\Compilador>java -jar GramaticaPascal.jar
Introducir archivo:
C:\Compilador\test.ps
Los comandos que pueden utilizarse son los son los siguientes:
[--run] [--compile] [--compile-run] [--help] [--exit] [--load]

 compile          Ejecuta la compilación previa a la ejecución del codigo.
 compile-run      Ejecuta la compilación y ejecuta el codigo.
 exit             Finaliza la ejecucion del compilador.
 help             Ayuda para el usuario.
 load             Permite cargar un nuevo archivo para compilar.
 run              Ejecuta el contenido del archivo, es necesaria una compilación
                    previa para poder utilizar este comando.

Introducir comando a ejecutar:
--load
Introduzca el nuevo archivo a compilar:
C:\Compilador\test2.ps
Archivo cargado correctamente.


Introducir comando a ejecutar:
```

Figura 95. Carga del archivo test2.ps al compilador.

```
C:\WINDOWS\system32\cmd.exe - java  -jar GramaticaPascal.jar
 compile-run      Ejecuta la compilación y ejecuta el codigo.
 exit             Finaliza la ejecucion del compilador.
 help             Ayuda para el usuario.
 load             Permite cargar un nuevo archivo para compilar.
 run              Ejecuta el contenido del archivo, es necesaria una compilación
                    previa para poder utilizar este comando.

Introducir comando a ejecutar:
--load
Introduzca el nuevo archivo a compilar:
C:\Compilador\test2.ps
Archivo cargado correctamente.


Introducir comando a ejecutar:
--compile-run
5
5
5
6
5
7
5
8
5
9
5
10
Introducir comando a ejecutar:
```

Figura 96. Compilación y ejecución del archivo test2.ps.

CONCLUSIONES

La investigación, y demostración, del uso de la herramienta JFLex resultó satisfactoria, exponiendo en ella los resultados esperados, presentados con ejemplos prácticos para un mayor entendimiento. Además de poder medir y entender la sencillez de su uso y la facilidad de integración a cualquier proyecto.
La herramienta JFLex es un auxiliar para generadores de analizadores lexicográficos, su propósito no es más que ayudar a generar analizadores con expresiones regulares establecidas por el usuario.

CUP es un metacompilador que nos permite generar el análisis sintáctico posterior a un análisis lexicográfico, donde el análisis lexicográfico puede ser ejecutado por un desarrollo propio o por medio de alguna herramienta dedicada a este análisis. CUP es bastante versátil y amigable ya que permite su integración con distintos analizadores léxicos como lo son JFLex, Lex o Flex. Dentro de este documento se demostró e integro con JFLex, permitiendo visualizar la rigidez de ambas herramientas y como su trabajo en conjunto es capaz de generar compiladores con amplias bases lo cual da la oportunidad de construir aplicaciones confiables con rapidez sin la necesidad de estar entrando en tantos detalles como lo implica un compilador convencional.

JFLex y CUP se integran fácilmente a cualquier proyecto Java que se este manejando, son livianos y cada uno esta preparado para trabajar con el otro. De igual forma nos permiten modificar los fuentes generados dando mayor confianza al desarrollador de saber que se está ejecutando o haciendo internamente para que finalmente se pueda ejercer las operaciones deseadas.

BIBLIOGRAFÍA

Baliri, S. (2014). *Teoria de Lenguajes Formales, Una Introduccion para Lingüistica.* Universidad Autonoma de Barcelona.

Campos, A. (1995). *Teoria de Automatas y Lenguajes Formales.* Chile: Pontificia Universidad Catolica de Chile.

Cueva Lovelle, J. M. (2001). *Lenguajes, Gramaticas y Automatas.* Universidad de Oviedo.

Daviana, R. B. (s.f.). *Analizador Lexico.*

Dean, K. (2001). *Teoria de automatas y lenguajes formales.* Madrid: Prentice Hall.

E. Hopcroft, J., Motwani, R., & D. Ullman, J. (2007). *Introduccion a la teoria de automatas, lenguajes y computacion.* Madrid: Pearson Eduacion.

Gerwin, K. (2004). *JFlex: The Fast Lexical Analyser Generatos, User's Manual.*

Ibarra Florencio, N. (2011). *Tipos de Lenguajes Formales.* Mexico: Universidad Nacional Autónoma de México.

Isasi Viñuelas, P., Martinez Fernandez, P., & Borrajo Millan, D. (1997). *Lenguajes, Gramaticas y Automatas: Un enfoque practico.* Madrid: Pearson Educacion.

JFlex, u. g. (2009). *Urquiza Fuentes, Jaime.* Universidad Rey Juan Carlos.

Lewis, H., & Papadimitriou, C. (1998). *Elements of the theory of computation.* Prentice Hall.

Moral, S. (s.f.). *Teoria de Automatas y Lenguajes Formales.* Universidad de Granada.

Navarrete Sanchez, I., Cardenas Viedma, M. A., Sanchez Alvarez, D., Botia Blaya, J. A., Marin Morales, R., & Martinez Bejar, R. (2008). *Teoria de Automatas y Lenguajes Formales.* Universidad de Murcia.

Padilla Beltran, P. G. (2006). *Lenguajes y Algebra de Eventos Regulares.* Mexico: Instituto Politecnico Nacional.

Polanco Fernandez, D. F. (2000). *Evaluacion y Mejora de Un Sistema Automatico de Analisis Sigtanmatico.* Madrid: Universidad Politecnica de Madrid.

Quiroga Rojas, E. A. (2008). *Introduccion a los Automatas y Lenguajes Formales.* Bogota: UNAD, Universidad Nacional Abierta y a Distancia.

Sanchez Dueñas, G., & Valverder Andreu, J. A. (1988). *Compiladores e intérpretes: un enfoque pragmático.* Madrid: Diaz de Santos S.A.

Urquina Fuentes, J. (2008). *Procesadores del Lenguaje.* Universidad Rey Juan Carlos.

Vazquez Palma, J. J. (s.f.). *Automatas y Lenguajes Formales: Conocer, utilizar y diseñar gramaticas de libre contexto.*

Vega Castro, R. A. (2008). *Integracion de JFlex y CUP (Analizador Lexico y Sintactico).*

Vilca Huayta, O. A. (2008). *Expresiones regulares y autómatas finitos.* Chile.

Zeugmann, T. (s.f.). *Theory Of Computation: Further Properties of Context-Free Languages.* Hokkaido University, Laboratory for Algorithmics.