CONSTRUCCIÓN DE UN ANALIZADOR LÉXICO UTILIZANDO JavaCC y Jflap COMO RECONOCEDORES DE CADENAS[1]

Darwin Ramiro Mercado Polo[2]

Fecha de recepción: 21 de Julio de 2009 / Fecha de aceptación: 30 de septiembre de 2009.

RESUMEN

A los estudiantes de último semestre de Ingeniería de Sistemas y en muchos casos a algunos profesionales del área se les pueden presentar dificultades al momento de hacer traducciones de un lenguaje de programación a otro, esto se debe en la mayoría de los casos a la dificultad en la apropiación de los conceptos lexicográficos debido al contenido abstracto que poseen. Además, la mayoría de los libros no muestran una directa relación entre la teoría y la práctica, propiciando que las personas que estudian estos contenidos pierdan el interés y no profundicen por la falta de una aplicación concreta. Con las herramientas y técnicas presentadas se intenta crear un escenario de estudio más agradable mediante la utilización de JavaCC y Jflap. Estos generadores tienen funciones específicas. Mientras que JavaCC permite implementar en una forma sencilla las expresiones regulares, Jflap facilita crear los autómatas finitos de cada expresión y realizar la evaluación de las cadenas.

Palabras clave

Analizador lexicográfico, generador, traductor, análisis léxico, autómatas finitos, expresiones regulares.

[1] Artículo resultado de la investigación "Herramientas y técnicas para la construcción de un analizador lexicográfico utilizando como generadores JavaCC y Jflap". Grupo de investigación: Ingeniería de Software y Redes. Línea de investigación: Ingeniería de software. Investigador principal: Darwin Mercado Polo.

[2] Ingeniero de Sistemas. Especialista en informática y telemática, Especialista en Estudios Pedagógicos y estudiante de segundo año de doctorado en Ingeniería de Software. Docente medio tiempo del programa de Ingeniería de Sistemas de la Corporación Universitaria de la Costa. Grupo de investigación: Ingeniería de Software y Redes. Línea de Investigación: Ingeniería de software. E-mail: dmercado@cuc.edu.co

CONSTRUCTION OF A LEXICAL ANALYZER USING JavaCC AND Jflap AS RECOGNIZER CHAINS

Darwin Ramiro Mercado Polo

ABSTRACT

The students of last semester of Engineering of Systems and in many cases some professionals of the area them can be presented difficulties at the time of requiring to make translations from a language to another one, this must in most of the cases to the appropriation of the lexicographical concepts due to the abstract content that she has. In addition most of books they do not have a direct relation between the theory and the practice, causing that the people who study these contents lose the interest and they do not deepen by the lack of a concrete application. With the presented tools and techniques it is tried to create a scene of more pleasant study by means of the use of JavaCC and Jflap. These generators have specific functions. Whereas JavaCC allows us to implement in a simple form the regular expressions, Jflap allows to create the finite robots of each expression and to make the evaluation of the chains.

Keywords

Lexicographic analyzer, generator, translator, lexical analysis, finite automata , regular expressions.

1. INTRODUCCIÓN

Dentro del proceso de traducción de un lenguaje a otro siempre se ha considerado que el análisis lexicográfico es el factor más importante porque en esta etapa es donde se recibe el programa fuente y se realiza un rastreo de izquierda a derecha para identificar cada uno de los componentes léxicos; estos son como las palabras de un lenguaje natural, es decir una secuencia de caracteres que representan una unidad de información dentro del programa fuente. De aquí se formula la pregunta ¿Por qué es necesario diseñar técnicas que permitan la construcción de un analizador lexicográfico? Seguro que la mayoría de las personas que en un momento dado han iniciado una traducción siempre tienen dificultad para identificar cada componente ya que requiere un trabajo muy acucioso por el contexto abstracto que maneja. Teniendo en cuenta lo anterior se han construido técnicas utilizando JavaCC y Jflap que faciliten o mejoren este proceso.

En este artículo se presenta información sobre el desarrollo de las técnicas y herramientas para el diseño del analizador lexicográfico, iniciando con una referencia a las diferentes etapas que se deben seguir para la construcción de un traductor. También se precisan los conceptos de traductor y compilador, luego se hace énfasis en la etapa de análisis léxico. Seguidamente se realiza un referente teórico de JavaCC y Jflap, herramientas necesarias para el reconocimiento de los tokens o componentes léxicos. Por último, se presenta un ejemplo breve sobre el uso de JavaCC y Jflap para identificar y reconocer las cadenas.

2. TRADUCTOR

Un traductor es un programa que toma como entrada un texto escrito en un lenguaje, llamado fuente, y da como salida otro texto en un lenguaje denominado objeto [1].

Si el lenguaje fuente es un lenguaje de programación de alto nivel y el objeto es un lenguaje de bajo nivel (ensamblador o código de máquina), a dicho traductor se le denomina compilador. Un ensamblador es un compilador cuyo lenguaje fuente es el lenguaje ensamblador. Un intérprete no genera un programa equivalente, sino que toma una sentencia del programa fuente en un lenguaje de alto nivel y la traduce al código equivalente y al mismo tiempo lo ejecuta.

Una tarea frecuente en las aplicaciones de las computadoras es el desarrollo de programas de interfaces e intérpretes de comandos, que son más pequeños que los compiladores pero utilizan las mismas técnicas. En los traductores, en forma general, se deben seguir una serie de etapas que difieren muy poco con respecto a las etapas de un compilador; sólo en la etapa de síntesis puede haber variaciones porque la traducción final no siempre es hacia un bajo nivel.

2.1. Fases y etapas del compilador

En general todo compilador debe tener una etapa de análisis y una de síntesis. La primera realiza toda la evolución y verificación del código fuente y la síntesis es donde se realiza la traducción final (Ver Figura 1).

En la fase de análisis léxico se revisa o se examina el programa fuente como una cadena de izquierda a derecha y se generan los componentes léxicos o token a partir de una secuencia

de caracteres que tenga un significado válido. Luego, en la fase de análisis sintáctico, recibe todos los componentes léxicos reconocidos en la etapa anterior y los agrupa en forma jerárquica a través de la gramática independiente del contexto (define si está bien escrito). Seguidamente, en la etapa de análisis semántico, el compilador intenta detectar construcciones que tengan la estructura sintáctica correcta pero que no tengan significado para la operación implicada y por último en la generación de código se toma el programa y se crea un código intermedio que generalmente lo traen todos los compiladores. La fase de análisis léxico se constituye en la fase de estudio necesaria para la construcción del analizador lexicográfico en cuestión.

3. ANÁLISIS LÉXICO

El analizador léxico es la primera etapa de un traductor. Su principal función consiste el leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer su análisis. El analizador léxico lee el programa fuente como un archivo de caracteres y lo divide en componentes léxicos [2].

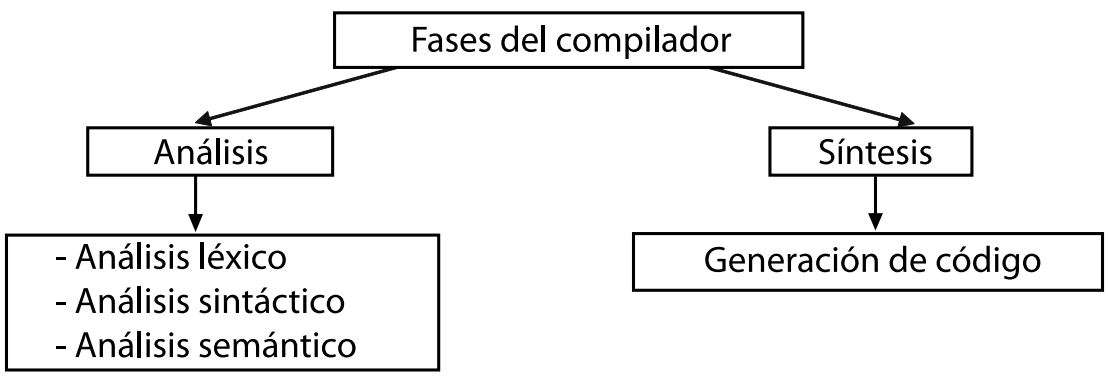
3.1. Funciones del analizador léxico

- Leer el programa fuente como un archivo de caracteres y dividirlo en unidades lógicas denominadas componentes léxicos o tokens.
- Eliminar comentarios o espacios en blanco realizados con las teclas espaciadora, tabuladora o fin de líneas.
- Reportar errores, si existen.

Para este análisis es importante tener en cuenta algunos conceptos como: Componentes léxicos o tokens, patrón o regla, lexema, cadena, lenguajes, alfabetos y un concepto muy importante que es el de "expresiones regulares", a través del cual se simplifica la especificación de un lenguaje regular y sirve para especificar un patrón y por último, los autómatas finitos, los cuales son grafos dirigidos que representan las acciones que tienen lugar al obtener el siguiente componente léxico, además es la estructura que permite representar gráficamente las expresiones regulares.

Los conceptos anteriormente citados, como las expresiones regulares y autómatas finitos, son la parte fundamental para el reconocimiento de tokens o componentes léxicos. El proceso de reconocimiento desde el punto de vista teóri-

Figura 1. Fases del compilador.



co requiere de conceptos muy abstractos que surgen de los lenguajes formales, sin embargo existen generadores como JavaCC y Jflap que facilitan ese reconocimiento o permiten identificar cadenas en una forma práctica sin necesidad de realizar un seguimiento analítico que implica mucho tiempo de análisis y una excelente apropiación del tema.

3.2. Conceptos básicos importantes para el reconocimiento léxico

Componentes léxicos o tokens: Es un caracter o secuencia de caracteres que representan una unidad de información dentro del programa fuente; estos pueden ser: un signo de puntuación, un operador aritmético, una palabra reservada, una constante, etc.

Patrón: Es una regla que describe el conjunto de cadenas de la entrada que corresponden a un componente léxico. Ejemplo: Una variable comienza en letras, seguidas de letras y/o dígitos.

Lexema: Es una secuencia de caracteres del programa fuente que concuerdan con un patrón. Ejemplo: la cadena A123YU concuerda con el patrón anterior (Una variable comienza en letras, seguidas de letras y/o dígitos).

Alfabeto: Es un conjunto no vacío y finito de símbolos. Se denota con el símbolo $\sum$. Ejemplo: $\sum_1 = \{0,1\}$

Cadena: Es una sucesión o secuencia de caracteres tomados a partir de un alfabeto. Ejemplo: 010001 es una cadena que pudo ser tomada del alfabeto.

Lenguaje: Es un conjunto finito o infinito de cadenas construido a partir de los símbolos del alfabeto. Se denota con la letra L. Ejemplo: L_1= {010, 0110, 11100}

Existen dos tipos de lenguajes: Lenguajes regulares y no regulares.

Lenguajes regulares

Si {a} y {∈} son lenguajes regulares básicos. Un lenguaje regular es un lenguaje que puede obtenerse de los lenguajes básicos anteriores a partir de las operaciones de unión, concatenación y cerradura de Kleene [3].

Definición formal:

1. {∈} es un lenguaje regular

2. Si ***a*** pertenece al alfabeto ($\sum$) entonces {***a***} es un lenguaje regular.

3. Si L_1 y L_2 son lenguajes entonces L_1 L_2, L_1 U L_2, L_1*, L_2* son lenguajes regulares.

Expresiones regulares [2]

Simplifica la especificación de un lenguaje regular y sirve para especificar un patrón.

Los operadores que se utilizan son los siguientes:

.	Concatenación
|	Unión
*	Cerradura de Kleene
+	Cerradura positiva
?	Cerradura 0 o 1 caso
(,)	Agrupar

El orden jerárquico es el siguiente:

1 (,)	2.*, +, ?	3. .	4. |

Definición:

1. ∈ es una expresión regular.

2. si **a** pertenece a Σ, entonces **a** es una expresión regular.

3. Si r y s son expresiones regulares entonces su lenguaje regular es respectivamente L(r) y L(s).

a. r es una expresión regular y se representa L (r) = {r}

b. (r | s) es una expresión regular y se representa, L(r) U (s) = {r} U {s} = {r, s}.

c. r. s es una expresión regular y se representa L(r) L(s) = {r}{s} = {rs}

d. r* es una expresión regular y se representa L*(r) = {r}* = {r, rr, rrr, rrrr…}.

Propiedades de la expresión regular:

Sean r, s y t expresiones regulares

1.	r . ∈ = ∈ . r = r	MODULATIVA
2.	(r . s) . t = r . (s . t)	ASOCIATIVA (.)
3.	(r | s) | t = r | (s | t)	ASOCIATIVA (.)
4.	r . (s | t) = r . s | r . t	DISTRIBUTIVA
5.	(r*)* = r*	IDEMPOTENCIA
6.	$r^+ = r . r^*$	
7.	$r^* = \in \mid r^+$	
8.	$r^? = \in \mid r$	

Ejemplo:

Σ = {a, b} ⟶ Alfabeto

$a . b^*$ ⟶ Expresión regular

El lenguaje correspondiente a la expresión regular

a. L(a) . L*(b) cada uno representa su conjunto, entonces,

L(a) . L*(b) = {a} . {b}*, luego se desarrolla la cerradura de Kleene de

{b}* = { ∈, b, bb, bbb, ...} concatenado da como resultado

{a} . {b}*= {a, ab, abb, abbb, abbbb, ...}.

Este lenguaje regular corresponde a la expresión regular $a . b^*$.

A partir de la expresión regular se construye un diagrama de flujo en forma de grafo dirigido, que permite realizar la implementación de las expresiones.

3.3. Autómatas finitos

Son diagramas de transición pero con restricciones. Se dividen en 2:

3.3.1. AFD (Autómatas finitos determinísticos):

Presentan las siguientes características:

- Tienen un solo camino de ejecución para determinar la salida, es decir, de un estado no pueden salir varios caminos con igual componente léxico (Ver Figura 2).
- No pueden tener transiciones ϵ.

3.3.2. AFN (Autómatas finitos no determinísticos):

Presentan las siguientes características:

- Tienen varios caminos de ejecución para determinar la salida.
- Pueden tener transiciones ϵ.

AF = {S, S_0, F, Σ, δ}, donde:

S = Conjunto de estados del autómata.

S_0 = Estado inicial del autómata.

F = Conjuntos de estados de finalización o aceptación.

Σ = Alfabeto o transiciones utilizadas en los autómatas.

δ = Función de transición $\delta: S x \Sigma \longrightarrow S.$

$\delta(S_0, a) \longrightarrow S_1.$

Figura 2. AFD con transición a.

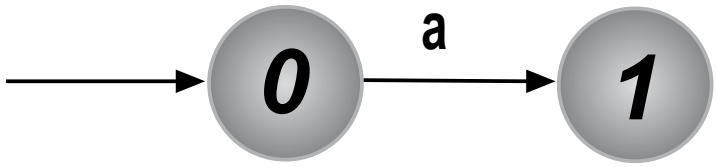


Figura 3. AFN con transiciones a y b.

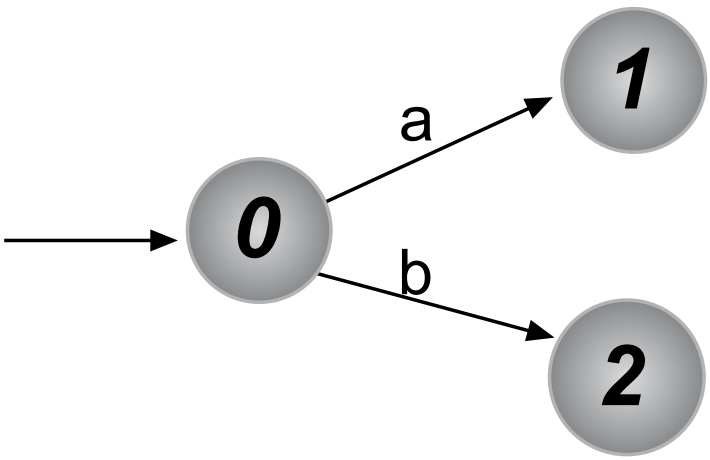
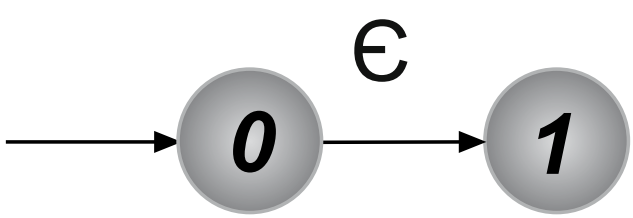


Figura 4. AFN con transición ϵ.



AFD = La imagen siempre será un estado.

AFN = $Sx\Sigma \longrightarrow P_{(S)}$ = Potencias de estados.

$\delta(S_0,a) \longrightarrow S_1,S_2$.

4. JAVACC Y JFLAP

4.1. JavaCC (Java Compiler Compiler – Meta compilador en Java)

Es el principal metacompilador en JavaCC, tanto por sus posibilidades como por su ámbito de difusión. Se trata de una herramienta que facilita la construcción de analizadores léxico y sintáctico por el método de las funciones recursivas, aunque permite una notación muy relajada parecida a la BNF (abreviatura en inglés de Forma de Backus – Naur). De esta manera, los analizadores generados utilizan la técnica descendente a la hora de obtener el árbol sintáctico [5].

En 1996, Sun Microsystems liberó un parser llamado Jack. Los desarrolladores responsables de Jack crearon su propia compañía llamada Metamata y cambiaron el nombre Jack a JavaCC. Metamata se convirtió en WebGain. Después de que WebGain finalizara sus operaciones, JavaCC se trasladó a su ubicación actual*.

Sus principales características son [5]:

JavaCC integra en una misma herramienta al analizador lexicográfico y al sintáctico, y el código que genera es independiente de cualquier biblioteca externa, lo que le confiere una interesante propiedad de independencia respecto al entorno. A grandes rasgos, sus principales características son las siguientes:

• Las especificaciones léxicas y sintácticas se ubican en un solo archivo.

• Admite el uso de estados léxicos y la capacidad de agregar acciones léxicas incluyendo un bloque de código Java para el identificador de un token.

• Incorpora distintos tipos de tokens: normales (TOKEN), especiales (SPECIAL_TOKEN), espaciadores (SKIP) y de continuación (MORE). Ello facilita trabajar con especificaciones más claras, a la vez que permite una mejor gestión de los mensajes de error y advertencia por parte de JavaCC en tiempo de metacompilación.

• La especificación léxica puede definir tokens de manera tal que no se diferencien las mayúsculas de las minúsculas bien a nivel global, bien en un patrón concreto.

También es considerado altamente eficiente, lo que lo hace apto para entornos profesionales y lo ha convertido en uno de los metacompiladores más extendidos (quizás el que más, por encima de JFlex/Cup).

La estructura básica de un programa en JavaCC es como se aprecia en la Figura 5.

4.2. Jflap

Es un software utilizado para la experimentación con lenguajes de libre contexto, que maneja conceptos como Máquinas de Turing, análisis sintáctico y gramáticas LL, sobre todo expresiones regulares y autómatas finitos; además de la construcción de estas y las pruebas léxicas de expresiones.

Jflap permite interactuar con las pruebas de construcción de una forma a otra, como la conversión de un NFA (Autómata Finito No Determinístico) a un DFA (Autómata Finito Determinístico) a un estado mínimo de DFA a una expresión regular o gramática; así como también la conversión de expresión de regular a autómata finito y viceversa [6].

* Consultado en http://es.wikipedia.org/wiki/JavaCC

5. PROPUESTA

El proceso de análisis léxico parte inicialmente de un reconocimiento de los elementos o símbolos que forman parte del lenguaje fuente. Estos elementos o símbolos se definen dentro de un conjunto denominado Alfabeto. A partir de estos alfabetos se pueden construir notaciones para especificar patrones o para simplificar las especificaciones de un lenguaje regular. Una vez establecidas las expresiones regulares se pueden construir los autómatas finitos que no son más que grafos dirigidos que permiten reconocer cadenas y a partir de estos autómatas se pueden rechazar o aceptar las cadenas siendo este el objetivo primordial del análisis léxico.

El resultado de esta investigación se centra básicamente en ofrecer técnicas y herramientas para el reconocimiento de un componente léxico y su representación mediante los autómatas finitos. Es así como se han creado una guías mediante videos donde el estudiante puede hacer el seguimiento de los pasos que debe tener en cuenta para dicho reconocimiento.

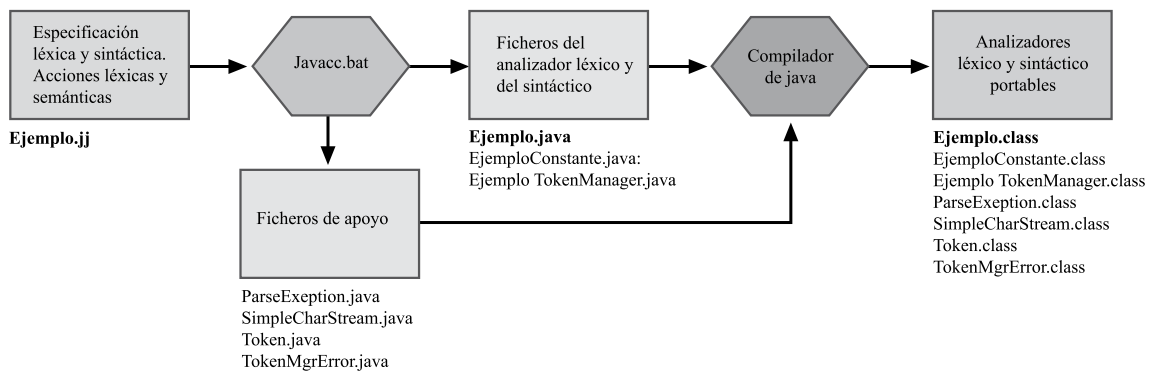
A continuación se muestra cómo se utiliza JavaCC y Jflap.

Inicialmente el usuario debe construir su expresión regular o definición regular. Si se desea sólo recocer cadenas aceptadas por el lenguaje regular se puede utilizar JavaCC, por ejemplo si la expresión regular que se quiere evaluar cumple con el siguiente patrón: Una variable que empieza solo en letra o guión bajo y puede ir seguida de letras, dígitos o guiones bajos donde los guiones bajos no pueden ir seguidos. La expresión correspondiente a dicho patrón es *(Letra ? | _)((letra | dig) _?)**

Se realiza la generación de la codificación entre los parsers (PARSER_BEGIN y PARSER_END); esta codificación es en lenguaje Java, por lo tanto toda la codificación que se realice entre estos parsers es omitida por el compilador de JavaCC y todos los errores generados en este código no son vistos hasta que las clases generadas por JavaCC sean compiladas por el compilador de Java.

Nuestra clase se llama Variable y dentro de los parsers se coloca un método "main" para que la clase principal sea ejecutable. En la principal se crea una instancia de la clase y se le llama parser, usando un constructor que tiene como parámetro un java.io.InputStream. En este caso usamos

Figura 5. Estructura de un programa en JavaCC[10]



el FileInputStream.

El código se verá de la siguiente manera:

```
options {
 LOOKAHEAD = 1;
 CHOICE_AMBIGUITY_CHECK = 2;
 OTHER_AMBIGUITY_CHECK = 1;
 STATIC = true;
 DEBUG_PARSER = false;
 DEBUG_LOOKAHEAD = false;
 DEBUG_TOKEN_MANAGER = false;
 ERROR_REPORTING = true;
 JAVA_UNICODE_ESCAPE = false;
 UNICODE_INPUT = false;
 IGNORE_CASE = false;
 USER_TOKEN_MANAGER = false;
 USER_CHAR_STREAM = false;
 BUILD_PARSER = true;
 BUILD_TOKEN_MANAGER = true;
 SANITY_CHECK = true;
 FORCE_LA_CHECK = false;
} /* estas son las opciones que ofrece Java por defecto.
PARSER_BEGIN (Variable)
public class variable {
 /** Main entry point. */
 public static void main(String args[]) throws ParseException, java.io.FileNotFoundException {
Variable parser = new Variable(new java.io.FileInputStream("prueba.txt"));
parser.Input();
}
}
PARSER_END(Variable)
SKIP :
{
 " "
| "\t"
| "\n"
| "\r"
}
TOKEN :
{
 < #letra: ["a"-"z","A"-"Z"]>
| < #dig: ["0"-"9"] >
}
TOKEN :
{
 < Variable: (<letra> "_"? | "_") (( <letra> | <dig>) "_"? )* >
}
/** Top level production. */
void Input() :
{}
{
 ( <Variable> )+ <EOF>
 { System.out.println("La evaluacion de las variables fue exitosa"); }
}
```

Si el objetivo no sólo es reconocer las cadenas sino también obtener los autómatas que representa dicha expresión es necesario utilizar Jflap.

Convertiremos una Expresión Regular a un Autómata Finito no determinístico y evaluaremos múltiples cadenas para ver si son rechazadas o no. Escribimos la siguiente expresión que estando en papel sería: b?(ab | a)* y que se debe escribir de la siguiente manera: (b+!)(ab+a)* (Ver Figura 6).

Para convertir a NFA (Autómata Finito No determinístico) se utiliza Convert To NFA como se aprecia en la Figura 7.

Luego se hace click en Do Step varias veces para completar la conversión. Terminada la conversión queda como en la Figura 8.

El Autómata Finito resultante es no determinístico, el cual es obtenido a través del método de Thompson.

Por último exportamos y procedemos a evaluar las cadenas.

Ahora exportamos y procedemos a evaluar las cadenas que sean aceptadas o rechazadas.

Ya estando en la ventana que nos lleva a la exportación (Ver Figura 9), nos dirigimos a Input / Multiple Run:

Aparecerá un cuadro como el que se observa en la Figura 10.

Luego nos vamos a Run Inputs (Ver Figura 11).

Aparece si fue aceptado o no, con la siguiente notación: Aceptada: Accept y rechazada: Reject.

Figura 6. Entrada de la expresión regular

Figura 7. Conversión a NFA.

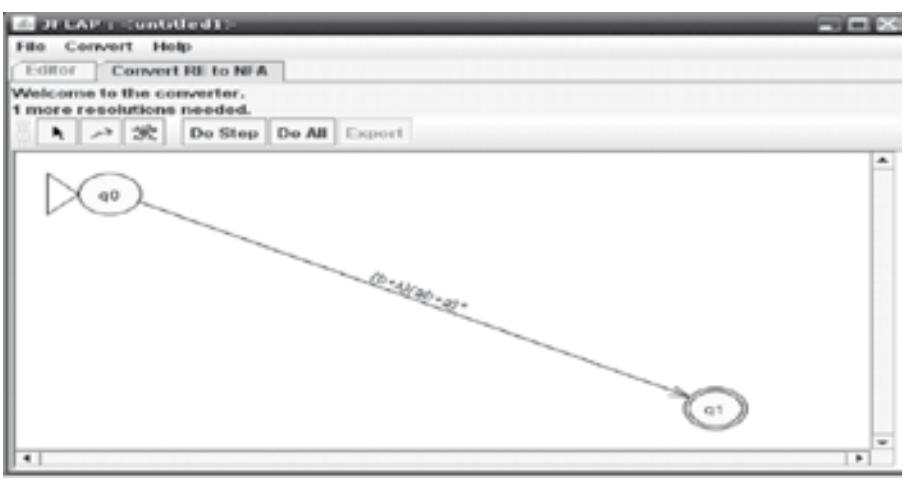


Figura 8. Conversión terminada a NFA.

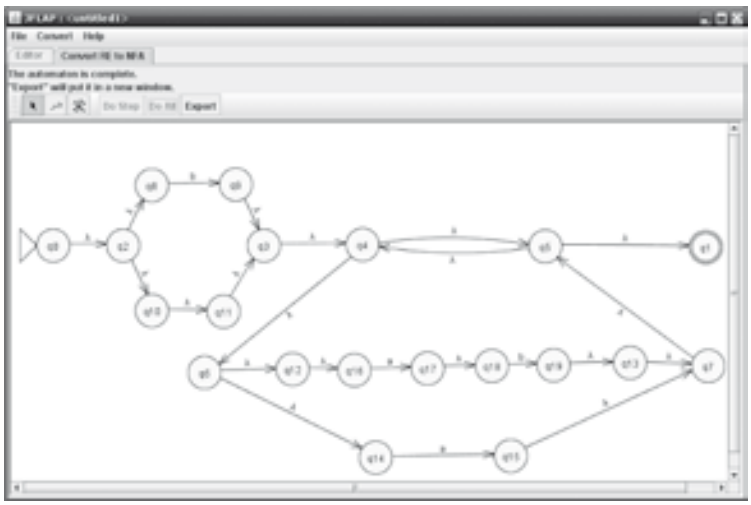


Figura 9. Ventana de exportación.

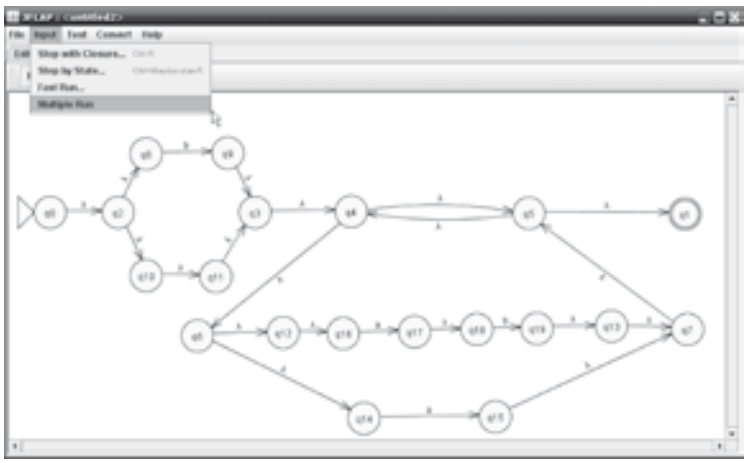


Figura 10. Ventana de evaluación de cadenas.

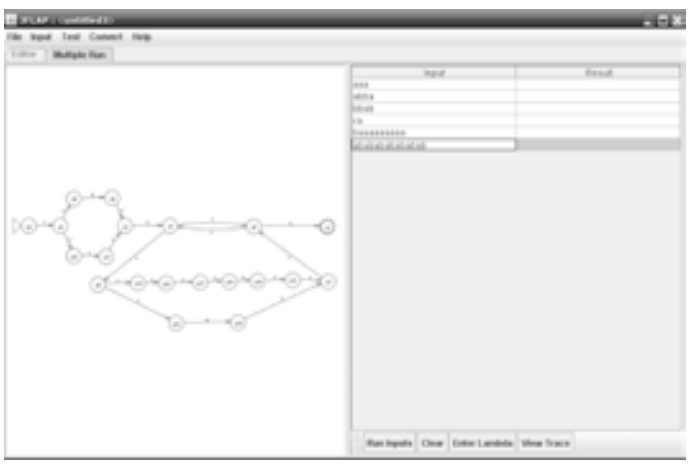
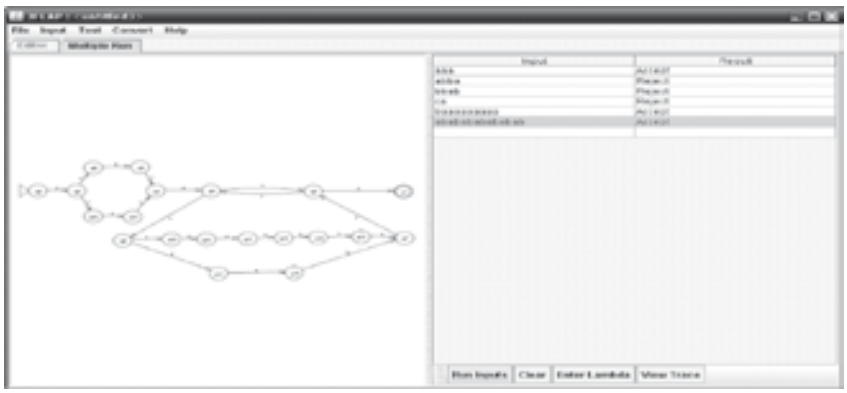


Figura 11. Ventana de aceptación o rechazo de cadenas.



6. CONCLUSIONES

A través de la aplicación de JavaCC y Jflap se puede concluir que será un gran aporte desde el punto de vista académico porque los docentes utilizarían estas técnicas para profundizar y mejorar su proceso enseñanza – aprendizaje para que los estudiantes puedan confrontar todo el contexto teórico abstracto con la representación práctica hecha directamente con JavaCC y Jflap. También será un aporte importante desde el punto de vista científico porque a través de estas técnicas y la relación teórico – práctica planteada se pueden sentar las bases para realizar traducciones léxicas de un lenguaje de programación a otro con menos trabajo y con un margen de error muy pequeño.

Las técnicas propuestas reducirán el proceso de reconocimiento léxico ya que siempre se ha considerado muy gravoso y muy complejo evaluar y reconocer las cadenas de un lenguaje de programación, sobre todo si tienen relacionado un patrón o reglas que definen sus características. Entre estas cadenas se pueden citar las variables o identificadores y las constantes.

El proceso normal en un reconocimiento léxico es:

1. Inicialmente se identifica el patrón.
2. Se diseña la expresión regular que especifique ese patrón.
3. Se construye el autómata finito con su respectiva matriz de transición.
4. Se realiza un programa a partir de la matriz obtenida que permita evaluar las cadenas y reconocer cuáles son aceptadas y cuáles no.

Mediante las técnicas anteriormente mencionadas se mejorará el proceso de reconocimiento de cadenas sobre todo a partir de los pasos 3 y 4 porque Jflap facilita la construcción de los autómatas para su posterior evaluación y JavaCC facilita el reconocimiento de las cadenas, las cuales están representadas en la expresión regular.

BIBLIOGRAFÍA

[1] A.V. Aho, R. Sethi y J.D. Ullman. Compiladores: Principios, técnicas y herramientas. Wilmington, Delaware: Addison-Wesley Iberoamericana S. A., 1990. pp. 1 - 5.

[2] L.C. Kenneth. Construcción de Compiladores. Principios y práctica. México: Thomson editores, 2004. pp. 31.

[3] J. Martín. Lenguajes formales y teoría de la computación. Editorial McGraw-Hill, 2004. p. 85.

[4] D.K. Rodrigo. Teoría de la Computación: Lenguajes, Autómatas y Gramáticas. Colombia: Unilibros, 2004. pp. 17 - 20.

[5] S. Gálvez y M.M. Mora (2004, Mar). Compiladores: Traductores y compiladores con Lex/yacc y JavaCC. pp. 127, 131 y 134. Disponible: http://books.google.com.co/books?id=F3lWLs1iTAMC&printsec=frontcover#PPA131,M1

[6] S.H. Rodger y T.W. Finley (2006, agosto). Tutorial de Jflap. Disponible: http://www.jflap.org/tutorial/

REFERENCIAS BIBLIOGRÁFICAS

B. Teufel, S. Schmidt y T. Teufel. Compiladores: Conceptos fundamentales. Addison-Wesley Iberoamericana, 1993. pp. 10 - 15.

G. Sánchez y J.A. Valverde. Compiladores e intérpretes: un enfoque pragmático. Ediciones Díaz de Santos. 1989, pp. 4 - 8.