Universidad Nacional de Asunción

Facultad Politécnica

Ingeniería Informática
Décimo Semestre

Diseño de Compiladores
Prof. Sergio Aranda

Proyecto:

"Máster en análisis Léxico"

Generación de Automatas Finitos para Expresiones Regulares

Autores:

Fernando Mancía
Cristhian Parra

Octubre 2008

Índice de contenido

1 Descripción del Proyecto

Master en Análisis Léxico, o *AfGen* como nos gustar llamarle a nosotros, es el proyecto de fin de Curso de la asignatura "Diseño de Compiladores", de la carrera Ingeniería Informática de la Facultad Politecnica de la Universidad Nacional de Asunción.

Se trata de un ***convertidor de expresiones regulares*** a su Automatas Finitos equivalentes (no-determinístico, determinístico, determinístico mínimo).

El objetivo final de afgen es proveer una interfaz agradable para trabajar con expresiones regulares, observar los automatas generados, validar cadenas de entradas para verificar su pertenencia al lenguaje L generado por las expresiones regulares y simular procesos de validación.

1.1 Autores:

El proyecto fue desarrollado por:

- Cristhian D. Parra T.
- Fernando M. Mancia Z.

Ambos son estudiantes de la carrera Ingeniería Informática de la Facultad Politécnica de la Universidad Nacional de Asunción.

2 Diseño

El proyecto fue diseñado como un ***Traductor Dirigido por la Sintaxis*** que recibe como entrada una expresión regular, y produce un grafo dirigido que representa el ***Automata Finito no Determinístico***.

Para lograr este diseño, se definió la sintaxis de las expresiones regulares y un conjunto de reglas y acciones semánticas asociadas al mismo que permiten aplicar construcciones de ***Thompson*** en cada etapa del análisis sintáctico de la entrada.

Las siguientes secciones definen la sintaxis utilizada, las optimizaciones realizadas a la sintaxis para permitir la implementación de un analizador predictivo, y las reglas semánticas asociadas con las construcciones de thompson.

2.1 Definición de la Gramática para expresiones regulares

La gramática utilizada está Basada en el BNF de expresiones regulares del lenguaje Perl.

```
1 RE            => RE "|" resimple | resimple
2 resimple      => resimple rebasico | rebasico
3 rebasico      => list op
4 op            => * | + | ? | ϵ
5 list          => grupo | leng
6 grupo         => "(" RE ")"
7 leng          => [alfabeto del lenguaje]
```

Tabla 1: *BNF básico de expresiones regulares*

2.1.1 Eliminación de la recursividad por la izquierda

Existe recursividad por la izquierda en las producciones (1) y (2). Luego de eliminar dicha recursividad, tenemos el siguiente BNF.

```
1 RE            => resimple A
2 A             => "|" resimple A | Є
3 resimple      => rebasico B
4 B             => rebasico B | Є
5 rebasico      => list op
6 op            => * | + | ? | Є
7 list          => grupo | leng
8 grupo         => "(" RE ")"
9 leng          => [alfabeto del lenguaje]
```

Tabla 2: *BNF mejorado sin recursividad por la izquierda.*

Nótese que ni en el BNF original ni en el mejorado se han encontrado problemas de factorización.

2.1.2 BNF + Reglas Semánticas

Se describen a continuación el conjunto combinado de Reglas de sintaxis y reglas semánticas básicas que guiaron la implementación del traductor.

1	RE	=> resimple **A**	`if (A.G != null) {`
			`  RE.G = thompson_or(resimple.G,A.G);`
			`} else {`
			`  RE.G = resimple.G;`
			`}`

2	A	=> "|" resimple **A**	`A.G = thompson_concat(resimple.G,A1.G);` `Especial = '|'`
3	A	=> **Є**	`A.G = null;` `Especial = ''`

4	resimple	=> rebasico **B**	`if ( B.G != null) {`
			`  resimple.G = thompson_concat(rebasico.G,B.G);`
			`} else {`
			`  resimple.G = rebasico.G;`
			`}`

5	B	=> rebasico **B**	`B.G = thompson_concat(rebasico.G,B1.G);`
6	B	=> **Є**	`B.G = null;`

7	rebasico	=> list op	`case op.x {`
			`  '*': rebasico.G = thompson_kleene(list.G);`
			`  '+': aux = thompson_kleene(list.G);`
			`  rebasico.G =thompson_concat(list.G,aux);`
			`  '?': rebasico.G = thompson_or(list.G,vacio);`
			`  '' :`
			`}`

8	op	=> *	`op.x = '*'`
9	op	=> +	`op.x = '+'`
10	op	=> ?	`op.x = '?'`
11	op	=> **Є**	`op.x = ''`
12	list	=> grupo	`list.G = grupo.G;`
13	list	=> leng	`list.G = leng.G;`
14	grupo	=> "(" RE ")"	`grupo.G = RE.G;`
15	leng	=> **"simbolo"** *** simbolo pertenece a [alfabeto del lenguaje]**	`leng.G = thompson_simbolo(simbolo);`

2.2 Análisis Léxico y Sintáctico

El análisis léxico implementado es un sencillo consumidor de caracteres cuya única verificación es controlar que el carácter a consumir pertenezca al alfabeto, o sea algunas de las operaciones soportadas.

El análisis léxico se facilita en gran medida porque solo se procesan Tokens de un solo carácter.

El análisis sintáctico es predictivo recursivo y está basado en el traductor simple orientado a la sintaxis presentado en el libro de Aho [1]. Consiste en la implementación de lo siguiente:

- Una función **Match()** para consumir los simbolos de la entrada
- Una función por cada producción del BNF.

2.3 Construcciones de Thompson utilizadas.

Para generar el AFN, se utilizaron las construcciones tradicionales de thompson con dos pequeñas simplificaciones. Las simplificaciones están relacionadas con el operador de **cerradura positiva (+)** y el operador **cero o una instancia (?).** Las siguientes imágenes muestran las versiones simplificadas utilizadas en esta implementación:

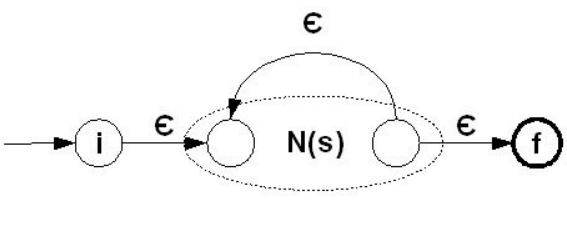

Figura 1:*Cerradura Positiva simplificada*

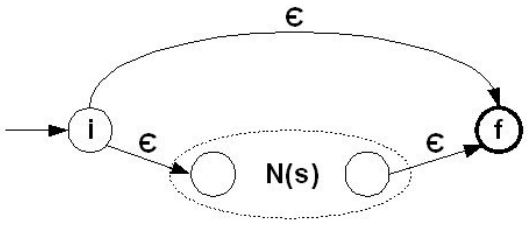

Figura 2: *cero o una instancia*

La razón por la cual se optó por estas simplificaciones es que permite simplificar el tratamiento de los subrafos que se van generando evitando la necesidad de replicarlos, lo cual sería necesario si una operación N(s)+ se implementara como N(s)N(s)*.

Además de estas dos operaciones, se soportan las siguientes:

- Cerradura de Kleene (*)
- Disyunción (|)
- Agrupación por paréntesis ().

2.4 Generación de los Automatas Determinísticos

Para la generación de los automatas finitos determinísticos, se utilizan dos algoritmos:

- Algoritmo de construcción de Subconjuntos para la conversión del AFN al AFD
- Algoritmo de minimización para producir un AFD mínimo a partir del AFD inicial.
    - Este algoritmo se complementa con una etapa preliminar de eliminación de estados inalcanzables y otra de eliminación de lo sestados muertes al terminar el algoritmo.

Ambos algoritmos citados se describen en el Capítulo 3 de [1].

2.5 Procesos de validación y Simulación.

Los procesos de validación implementados corresponden a los presentados en el Capítulo 3 de [1]. A estos procesos, se le agregó la construcción en línea del camino completo de simulación, que luego es utilizado para dirigir la simulación visual del sistema.

2.6 Graficación

Para la graficación de los automatas se optó por un esquema de uso de herramientas y librerías externas. Los componentes externos utilizados son:

- GraphViz [2]
- jGraph [3].

La primera es un conjunto de herramientas para describir y dibujar grafos a partir de una sintaxis de definición de los mismos.

La segunda es una API completa para manipulación de grafos desde Java.

3 Implementación

En esta sección se detallan brevemente las características y puntos resaltantes de la implementación, desde los detalles del lenguaje en sí hasta la arquitectura y estructura usada para el proyecto.

3.1 Plataforma

Fue desarrollado utilizando el lenguaje de programación Java. El compilador usado fue el JDK1.6.0_07 de Sun Microsystems así como también el intérprete. El entorno de desarrollo fue Netbeans 6.1.

Para la interfaz gráfica se utilizó el paquete Swing y awt del jdk de Sun. También para este fin se utilizó las herramientas y código proveidas por el IDE (© NetBeans).

El software fue probado exitosamente en sistemas operativos Linux (Fedora Red Hat 9) y Windows (© Windows XP) lo cual lo hace un sistema multiplataformas.

3.2 Librerias Externas Usadas

Para el dibujo del automata fueron utilzadas 2 librerías, una es el GraphViz (www.graphviz.org) de uso totalmente libre y gratuito y otro es el jgraph(www.jgraph.com), utilizando licencia académica.

Para la manipulación del archivo de configuración (xml) utilizamos una librería llamada simple. (simple.sourceforge.net) Esta provee abstracción para el procesamiento de archivos xml.

3.3 Descripción de los paquetes

3.3.1 afgenjava

En este paquete se encuentran las clases más importantes que definen y operan sobre el Automata. Estas clases son la base del automata y son Estado, Enlace, ListaEstados, ListaEnlaces, TipoAutomata, etc.

También están las clases que implementan los diferentes algoritmos sobre Automatas, como el de Thompson, el de Subconjuntos, Minimización, el de simulación.

3.3.2 app

En este paquete se encuentran todas las clases desarrolladas para la interfaz gráfica y que dan soporte a las mismas.

3.3.3 exceptions

Aquí se encuentran las excepciones que utilizamos dentro del programa para detectar con más precisión los tipos de problemas que puedan surgir en tiempo de ejecución. Las excepciones clasificamos como: Errores Lexicos, Errores Sintácticos y Errores producidos en el automata.

3.3.4 graphviz

Se encuentra la clase necesaria para la utilización del graphviz.

3.3.5 traductor

Soporte para un traductor dirigido por la sintaxis. En este paquete se encuentran clases encargadas de implementar los procedimientos necesarios para llevar a cabo el proceso de traducción. El traductor está basado en el BNF definido en la sección de Diseño de este documento.

3.4 Problemas Conocidos

La siguiente es una lista de algunos Bugs conocidos y no resueltos que han quedado para versiones futuras de la aplicación.

- Los path al directorio del graphviz y de imágenes temporales, y las expresiones regulares, alfabetos o cadenas de validación no pueden tener caracteres especiales como acentos o al ñ por problema no resueltos en la codificación interna utilizada.
- La ventana de Configuraciones se cuelga al intentar ingresar a particiones distintas de la C: en plataformas Windows XP.
- Simulación con jGraph no repinta en tiempo de ejecución los nodos.
- No se ha implementado la simulación con el AFN.

4 Funcionamiento

El programa provee una interfaz gráfica con las siguientes herramientas:

- Editor de expresiones regulares.
- Editor de Alfabetos, con alfabetos precargados.
- Editor de configuraciones de la aplicación con archivos de configuración en Xml.
- Procesos de generación de AFN, AFD y AFDMín integrados en un solo proceso.
- Presentación Visual de Tablas de transición.
- Editor de cadenas de prueba para validación. La validación se puede hacer con cualquiera de los automatas generados.
- Visualizador de Automatas con controles de simulación paso a paso.

La siguiente sección presenta algunos screenshots de la aplicación.

4.1 Screenshots

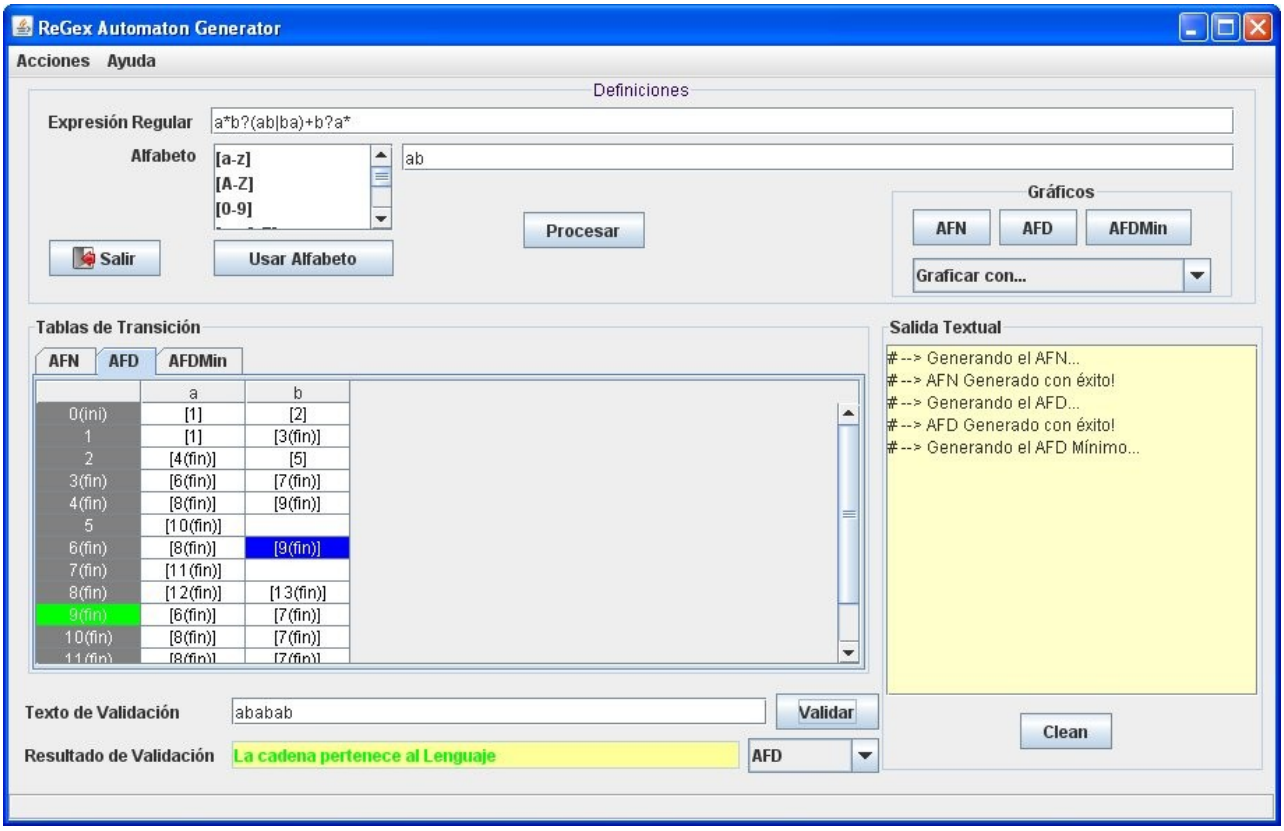

Figura 3:*Vista de la ventana principal.*

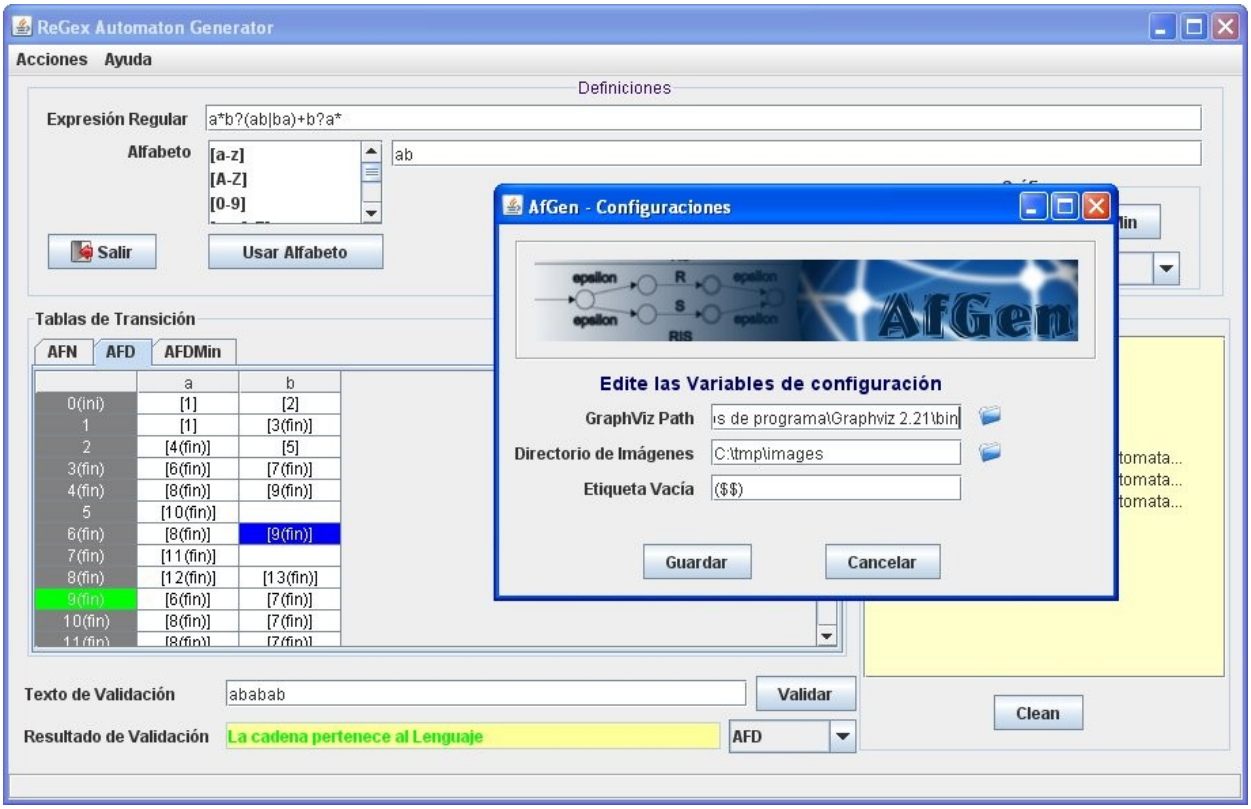

Figura 4:*Ventana de Configuraciones*

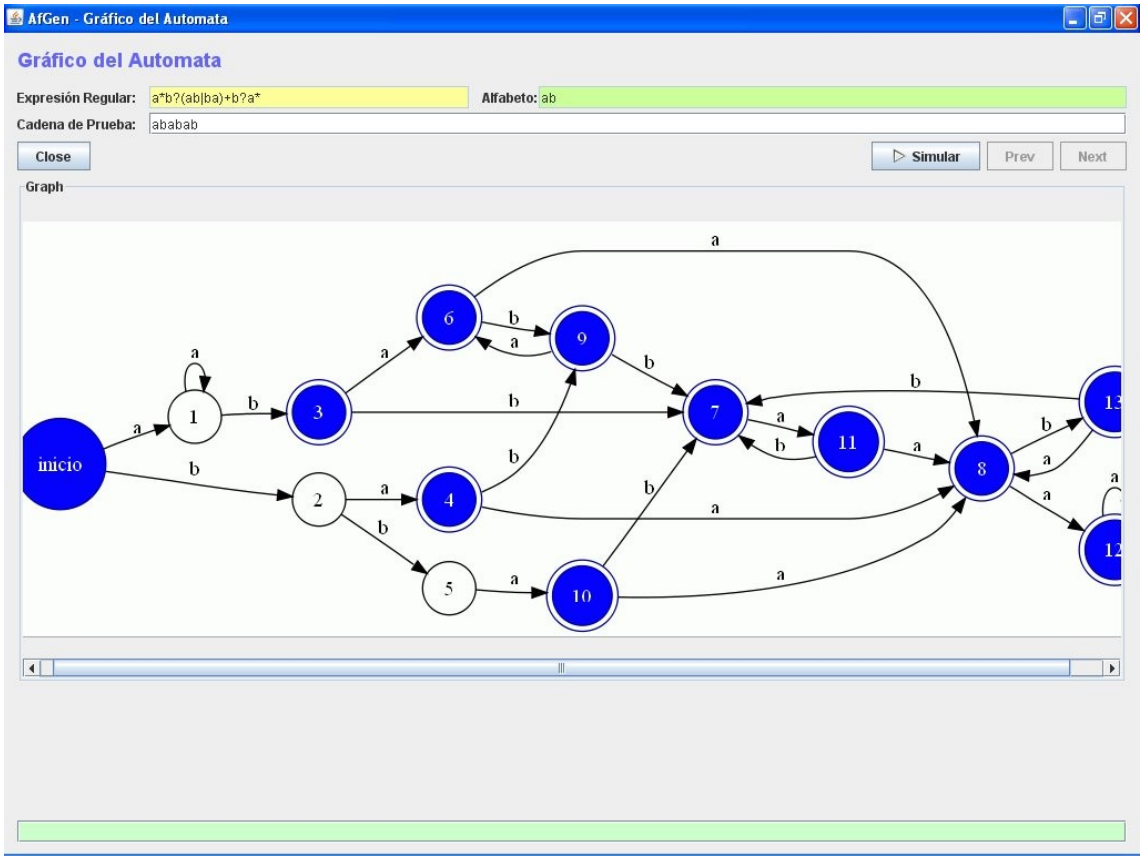

Figura 5:*Vista de la ventana de gráficos y simulación en el modo GraphViz*

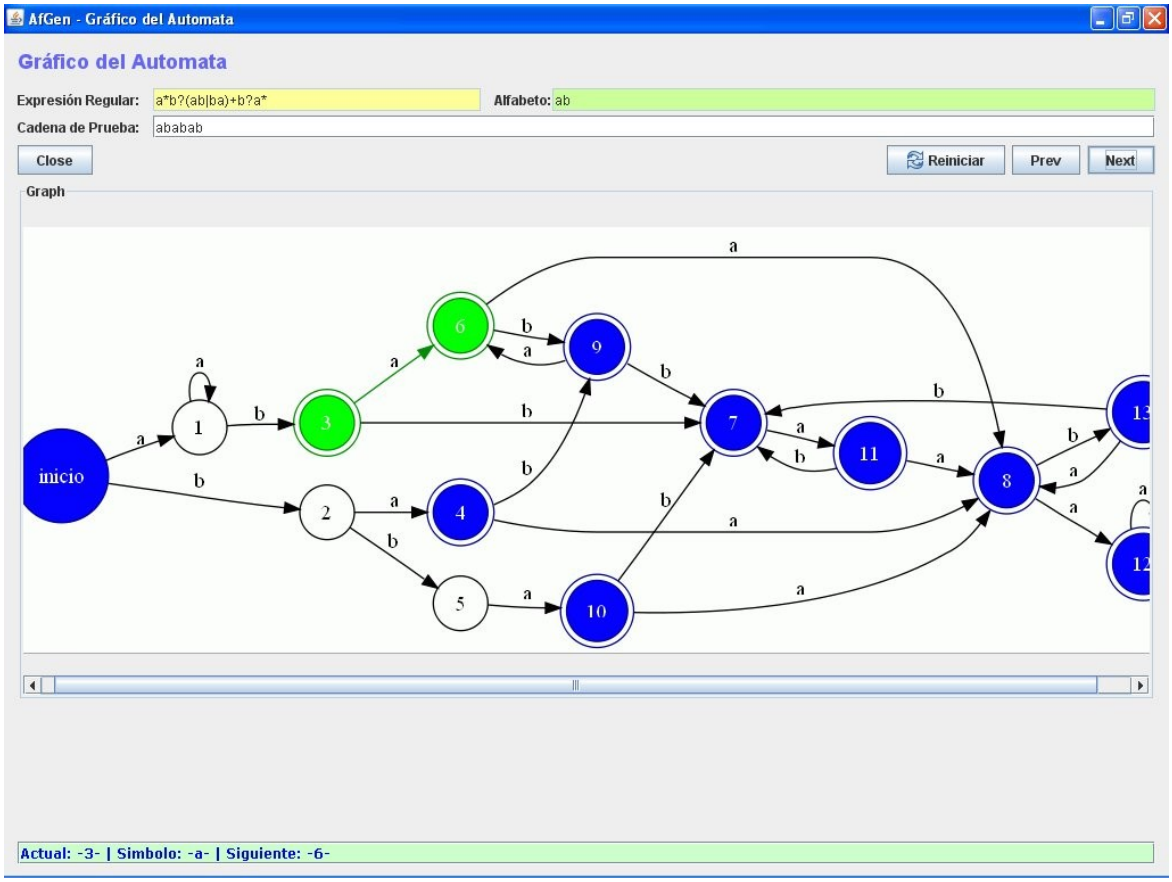

Figura 6:*Vista de la ventana de gráficos y simulación en el modo GraphViz en medio de un proceso de simulación*

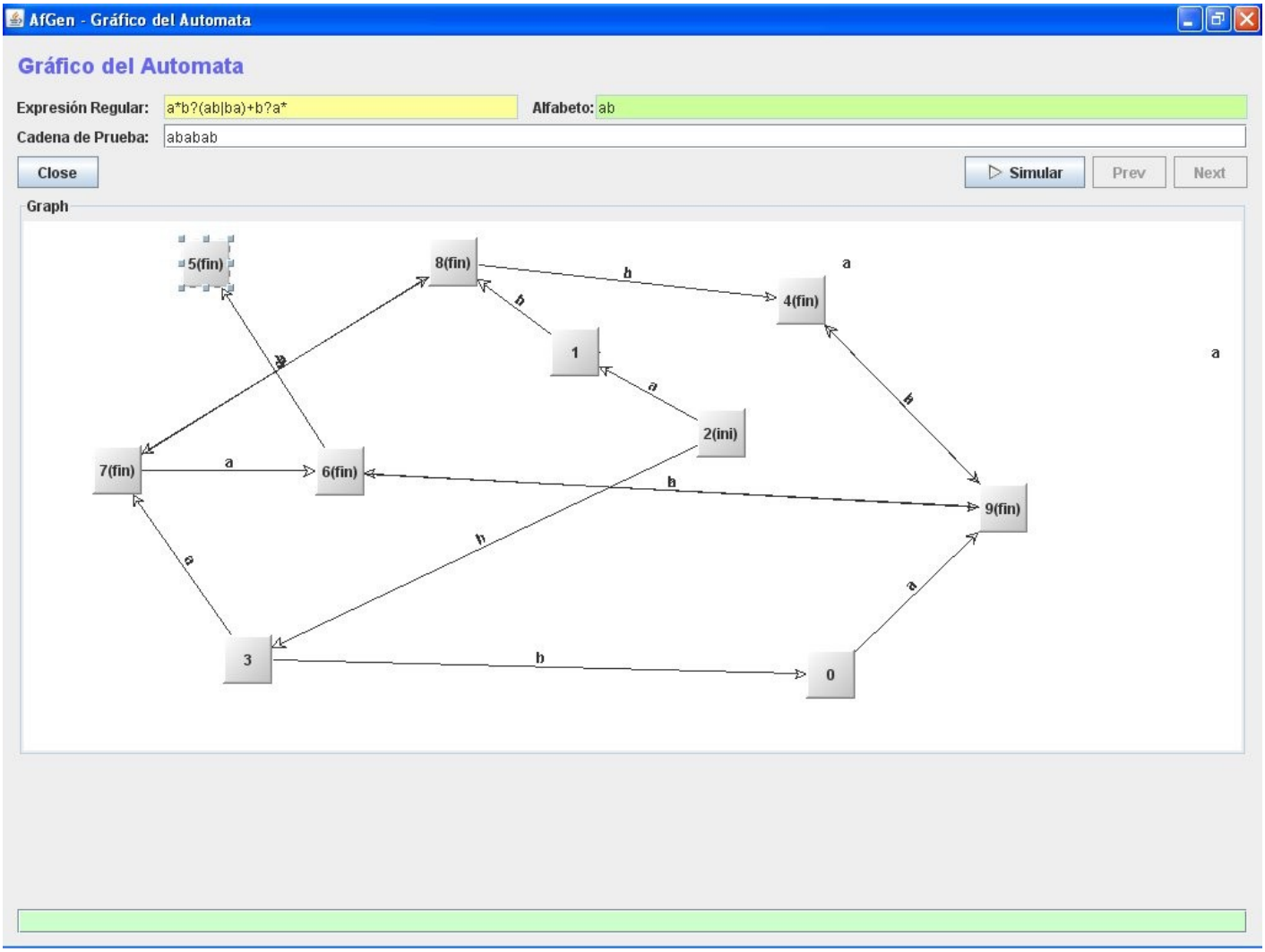

Figura 7:*Vista de la ventana de gráficos y simulación en el modo JGraph*

5 Referencias

1. Compiladores. Principios, técnicas y herramientas. Segunda Edición. Alfred V. Aho et. al.
2. GrapViz Visualization Software. http://www.graphviz.org/
3. Java Graph Visualization and Layout. http://www.jgraph.com/