Infraestructura Visual para Teoría de la Computación

por

Ivan Hernández Serrano

Tesis entregada para satisfacer parcialmente los
requisitos para obtener el título de
Licenciado

en

Ciencias de la Computación

en la

FACULTAD DE CIENCIAS
de la
UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Comit de tesis:

M. en C. Elisa Viso Gurovich M. en C. José de Jesús Galaviz Casas.
Lic. Francisco Lorenzo Solsona Cruz
Lic. Manuel Sugawara Muro
M. en I. Maria de Luz Gasca Soto

2002

A mis padres Natalio y Bertha.
A mis hermanos Jorge y Sitlalmina.

Índice General

Índice de Figuras

Índice de Algoritmos

Agradecimientos

Agradezco infinitamente a mi familia por todo el apoyo y cariño que siempre me han brindado, gracias por estar en todo momento conmigo, los quiero. Papá, gracias por ser un ejemplo de trabajo, honestidad, dedicación, responsabilidad, y valentía al enfrentar nuevos retos, te admiro y te quiero. Mami, gracias por ser tan linda con nosotros tus hijos, por formar nuestro carácter, y por la disciplina cuando era necesaria, gracias por ser tan comprensiva y amorosa, gracias por la fuerza que me has dado en los momentos de flaqueza, te quiero mucho. Jorge, hermanito, gracias por ser mi amigo, gracias por el apoyo y por los momentos que hemos pasado juntos, eres un gran ejemplo, estoy orgulloso de ti. Sitlalmina, hermanita, gracias por toda la confianza que me tienes, y por todos los ánimos que siempre me das, sé que siempre puedo contar contigo, te quiero y espero mucho de tí.

Agradezco a Elisa Viso, mi directora de tesis –una de las mejores personas que he conocido, y sin lugar a dudas, la mejor maestra que he tenido– por toda la paciencia que me tuvo durante el desarrollo de este trabajo, por todo el apoyo y la confianza que me ha brindado durante mi paso por la Facultad de Ciencias. Gracias Elisa.

Gracias a mis profesores. Javier García, por tan buenas clases en la carrera –con grandes desmañanadas– y por todo el entusiasmo que contagia en éstas. José Galaviz, clases muy interesantes y amenas, gran persona. Lucy Gasca, por la confianza que siempre me ha tenido.

Gracias a mis amigos desde la infancia. Arturo Gónzales, por ser un amigo incondicional y estar siempre que lo necesito, en las buenas y en las malas, mi estimado ArtClean una de las personas más nobles que conozco. Francisco Paredes, el buen choco, mucho reventón, muchas patadas y muchas *palchaguis*, claro en la juventud. Canek Vázquez, mi politico amigo, siempre puedo contar con sus llamadas –políticas :)– cada dos meses. Gerardo Reza, ¿dónde estás?

Gracias a los amigos que conocí en la carrera. Francisco Solsona, gran amigo, mucha diversión, largas jornadas de trabajo y de fiesta, gracias por toda la confianza y el apoyo que me has brindado Paco, tengo mucho que aprender de tí. Manuel Sugawara, un gran ejemplo de dedicación y trabajo, gracias por el apoyo en todo momento. Julio Barreiro, buen amigo, compañero de clase, de trabajo y de varios reventones, excelente persona y amigo. Karla Ramírez, varias clases juntos, muchas pláticas amenas, bastante confianza y cariño, muy buena amiga. Arturo Vázquez, gran amigo, muchas clases juntos, mucha fiesta y proyectos, siempre confiable e incondicional, el mundo sería mejor si todos le hicieran caso :).

A todos los que se me olvidaron. Mil disculpas y dos mil gracias :)

Capítulo 1

Introducción

La teoría de la computación abarca las propiedades matemáticas fundamentales de las computadoras. Al estudiar este tema buscamos determinar qué puede y qué no puede ser computado, que tan rápido, con cuánta memoria, y con qué modelo matemático. Estos temas tienen una conexión obvia con cuestiones prácticas.

Una de las preguntas que motivan a la teoría de la computación es:

> *¿Cuáles son las capacidades y limitaciones fundamentales de las computadoras?*

Esta pregunta se ha formulado desde hace mucho tiempo por las personas que comenzaban a explorar el significado de la computación. En Autómatas y Lenguajes Formales trabajamos con propiedades matemáticas de modelos de cómputo. Estos modelos juegan un papel muy importante en varias áreas aplicadas de las ciencias de la computación. Al revisar de forma adecuada el material de este curso se establecen bases sólidas para el estudio a fondo de temas fundamentales en teoría de la computación, entre los que se encuentran las teorías de computabilidad y complejidad, áreas que responden desde distintos puntos de vista la pregunta enunciada.

En este trabajo, ilustraremos el curso de Teoría de Computación que se imparte en la Facultad de Ciencias. de la U.N.A.M. a través de software que dé lugar a una relación más cercana de los individuos involucrados en el proceso educativo con el material que se revisa en el curso. Intentaremos dar una herramienta que muestre cada uno de los modelos que se revisan a lo largo de este curso, con el fin de apoyar a los estudiantes en el proceso de construcción del conocimiento, para que con mayor efectividad se pueda realizar el paso paulatino de estructuras y conocimientos simples a otros de mayor complejidad.

Los materiales didácticos visuales, desde los más simples hasta los más complejos, permiten el libre tránsito de la teoría a la práctica y viceversa; facilitan el acceso a la información; permiten superar limitaciones de espacio y tiempo en pro de la calidad y la cantidad de conocimiento; reducen el verbalismo y estimulan otras formas de expresión y aprendizaje.

El material que se revisa en el curso de Teoría de la Computación es particularmente interesante por el carácter constructivo en sus demostraciones. Estas demostraciones nos definen algoritmos, en su mayoría tratables, los cuales nos permitirán tender este puente entre la teoría y su visualización.

Por otro lado, gran parte de los conceptos se pueden manejar a un nivel intuitivo, aunque a veces la intuición se vea traicionada. Es aquí donde la ilustración es importante, pues nos ayuda a educar más la "intuición".

1.1 Ilustrar el curso . . .

A continuación detallamos los puntos que cubrirá este proyecto

1.1.1 Máquinas con un número finito de estados

- Autómatas finitos. Los aspectos que se ilustrarán son:
  - Dada la especificación de un autómata determinístico con un número finito de estados (DFA, por sus siglas en inglés), mostraremos su representación gráfica i.e. su diagrama de transiciones.
  - Recibiremos una cadena y mostraremos las configuraciones por las que la máquina pasa en cada momento de su ejecución.
- Minimización de autómatas finitos. Dada la especificación de un DFA, llevaremos a cabo el proceso de minimización.
- Relación entre distintos modelos de autómatas finitos.
  - Sabemos que

    > Dado un autómata finito determinístico existe un autómata finito no determinístico (NFA, por sus siglas en inglés) equivalente y viceversa, dado un autómata finito no determinístico, existe un autómata finito determinístico equivalente.

    La implicación $\Leftarrow$ es la interesante, así que la ilustraremos. Dada la especificación de un NFA construiremos el DFA equivalente.
- Lenguajes Regulares
- Equivalencia entre gramáticas tipo 3 (regulares) y autómatas finitos (AF). Dado un AF construiremos una gramática tipo 3 y viceversa.

1.1.2 Autómatas de Stack

- Dada la especificación de un Autómata de Stack, mostraremos su representación gráfica, mostrando los símbolos que están dentro del stack.
- Recibiremos una cadena y mostraremos las configuraciones por las que la máquina pasa en cada momento de su ejecución de manera gráfica y en modo texto.

1.1.3 Lenguajes Libres de Contexto

- Simplificar gramáticas libres de contexto (GLC). Dada una GLC obtendremos su versión simplificada, aplicando los algoritmos para eliminar símbolos inalcanzables, producciones unitarias y símbolos nulificables.

- Dada una GLC podremos pasar a Forma Normal de Chomsky y de Greibach.

1.1.4 Máquinas de Turing

- Implementación de una Máquina de Turing.

1.2 Implementación con Java

Usaremos un lenguaje orientado a objetos porque la naturaleza del problema nos sugiere que debemos desarrollar *entes* que se comporten de cierta manera a partir de sus datos; de hecho, que sepan administrar de forma adecuada sus recursos como son los estados y las estructuras de memoria. Esto se traduce en clases que sepan operar sobre sus datos. Así, datos y métodos, describirán el comportamiento de los objetos que definiremos. Se eligío el lenguaje Java porque:

- Es un lenguaje de propósito general, concurrente, basado en clases y orientado a objetos. Diseñado para este paradigma desde el principio, Java mejora características de lenguajes clásicos orientados a objetos como C++ y Smalltalk, y elimina sus características indeseables. Así, Java es un lenguaje de producción, no un lenguaje de investigación, pues su diseño no incluye características nuevas y no probadas.

- Es robusto, gracias a que es un lenguaje fuertemente tipificado; incluye manejo automático del almacenamiento de memoria, usando un recolector de basura para evitar los problemas inherentes a la liberación explícita de memoria, así como un excelente manejo de excepciones.

- Es portátil, pues es totalmente independiente respecto a los distintos tipos de plataformas utilizadas, tanto arquitecturas como sistemas operativos. Esto se logra gracias a que el compilador de Java no traduce a lenguaje de máquina, sino a un lenguaje de pseudo-máquina llamado *byte code* de Java. El *byte code* es el lenguaje de máquina para una máquina virtual de Java (JVM, por sus siglas en inglés). Así, para ejecutar *byte code* de Java basta con tener una JVM instalada en la computadora.

  Para afrontar el problema de la velocidad de ejecución, puesto que el *byte code* es interpretado, existen los compiladores *justo a tiempo* (JIT, por sus siglas en inglés), que traducen *byte code* a código nativo de máquina en tiempo de ejecución.

- Soporta el uso de conceptos avanzados de computación tales como hilos de ejecución (*threads*).

- Ofrece herramientas que promueven buenas formas de programación, como la generación de documentación en HTML acerca de la configuración de las clases, así como una buena organización de bibliotecas basadas en *paquetes*.

- El API de Java que usaremos para construir la interfaz gráfica está implementada sin usar código nativo, específico de una arquitectura y sistema operativo. Por lo que el despliegue será homogéneo en cualquier sistema operativo, sin la necesidad de modificación alguna en el código fuente.

1.3 Descripciones con XML

Usaremos el *lenguaje de marcas extensible* (XML, por sus siglas en inglés), para describir sin ambigüuedades cada uno de los elementos que vayamos presentando y desarrollando.

Seleccionamos XML porque: está basado en texto plano, es decir, no es un formato binario, así que podemos crear y editar cada una de las descripciones usando cualquier editor estándar; tiene una notación consistente, lo cual hace sencillo procesar los datos; define estructuras jerárquicas; y es extendible, es decir, podemos definir nuevos elementos en el lenguaje.

Además, usaremos DTDs para describir los nombres y las estrucuturas, donde, un DTD es una descripción formal en sintaxis declarativa de XML de algún tipo particular de documento[1]. Al usar DTDs podemos decir qué nombres serán usados para los diferentes tipos de elementos, dónde pueden aparecer y cómo se relacionan entre sí todos los elementos. Por ejemplo, si queremos describir una *Lista* que contiene *Artículos*, el DTD completo podría verse como sigue:

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <!ELEMENT Lista  (Artículo)+>
3  <!ELEMENT Artículo (#PCDATA)>
```

Un documento XML siempre comienza con un prólogo, que es la parte del documento que precede a los datos XML. Esta parte puede incluir una declaración de documento. La línea 2 define una lista como un tipo elemento que contiene uno o más artículos; y en la línea 3 se define a los artículos como un tipo de elemento que sólo contienen texto plano. Así, los analizadores sintácticos[2] leen el DTD antes de leer el documento para poder identificar dónde debe de venir cada tipo de elemento y cómo se relaciona con los otros. El ejemplo de arriba nos permitiría crear una lista como la siguiente:

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <!DOCTYPE Lista SYSTEM "lista.dtd">
3  <Lista>
4     <Artículo>Reloj</Artículo>
5     <Artículo>Monitor</Artículo>
6
```

[1] donde un documento será el elemento que queremos describir, i.e. el más alto en la jerarquía
[2] parser

```
7       <Artículo>Cuaderno</Artículo>
8     </Lista>
```

El prólogo del código anterior (líneas 1 y 2), muestra una declaración del tipo de documento que tiene que revisar, este documento es una Lista que debe de ser válido con respecto al DTD especificado en el archivo `lista.dtd` (ver código anterior).

Así, cada que definamos un elemento vamos a dar la descripción sintáctica de éstos mostrando una parte del DTD al que pertenece –en los apéndices se mostrarán todos los DTDs usados–.

Pasamos ahora a describir el trabajo desarrollado.

Capítulo 2

Preparación de la infraestructura de software

En este capítulo revisaremos las estructuras que implementamos, las cuales eran necesarias para los distintos modelos de máquinas y gramáticas. Las máquinas (autómatas) y las gramáticas guardan ambas una relación estrecha con los lenguajes formales. Un lenguaje formal es un conjunto de cadenas donde cada cadena está formada por un número finito de símbolos tomados de un cierto alfabeto. Los autómatas se encargan de *reconocer* si una cadena dada pertenece o no a un cierto lenguaje, mientras que las gramáticas generan cadenas de acuerdo a reglas que describen a un lenguaje dado. De esto, máquinas y gramáticas hacen uso de ciertas estructuras básicas: ambos modelos usan alfabetos y trabajan sobre cadenas, que están compuestas de símbolos. Todas las estructuras básicas, como símbolos, cadenas, alfabetos y demás que vayan apareciendo, serán parte del paquete jaguar.structures. Para empezar, en este paquete se encuentran:

- jaguar.structures.Symbol, los símbolos con los cuales formaremos alfabetos y cadenas.
- jaguar.structures.Alfabeto, alfabeto para modelar los conjuntos de entrada de las máquinas, para los dispositivos especiales de éstas, y para los conjuntos no terminales (N) y terminales (T) de las gramáticas.
- jaguar.structures.Str, para representar cadenas de entrada sobre las cuáles ejecutaremos las máquinas para probar su pertenencia y sobre gramáticas para ver si son generadas por estás, además, para representar producciones en las gramáticas, entre otras cosas.

A continuación describiremos más estructuras generales que definimos para las máquinas y para las gramáticas.

2.1 Infraestructura de Máquinas

Antes de comenzar con la implementación de las distintas máquinas, observemos las características y el comportamiento común de los distintos modelos y definamos una máquina genérica.

Todas las máquinas tienen, al menos, un conjunto de estados Q, un alfabeto de entrada Σ, un conjunto de estados finales F y una función de transición δ. Por ello podemos definir lo que corresponde a una máquina abstracta que define la base para todas las demás máquinas, y la representamos en la clase abstracta Machine, a partir de la cual vamos a implementar y especializar ciertos comportamientos, dependiendo del modelo particular.

Además, en los paquetes jaguar.machine.structures y jaguar.structures definimos e implementamos las estructuras que forman parte de los componentes usados por todos los modelos. Así, agregamos las siguientes clases e interfaces:

- jaguar.structures.State, clase que representa los estados de las máquinas.
- jaguar.structures.StateSet, clase que opera como conjunto de estados. Con esta clase definiremos los conjuntos Q y F, entre otros, de las máquinas.
- jaguar.machine.structures.Delta, clase abstracta que define la estructura básica de una función de transición δ de una máquina.

Cada una de estas estructuras cuenta con un conjunto de operaciones básicas propias de su tipo, así como funciones de comportamiento general como leer y escribir su definición de un archivo. Además, cada una tiene definidas ciertas excepciones (manejo de errores), las cuales residen en los paquetes jaguar.structures.exceptions y jaguar.machine.structures.exceptions.

Del paquete jaguar.machine.structures resalta la clase abstracta Delta, que representa a la función de transición. Ésta define un comportamiento general que debe ser especializado según el tipo de máquina que se quiera implementar.

Ahora, para mostrar gráficamente a las máquinas, en particular su representación como diagrama de transición sobre un lienzo, y siguiendo el paradigma de objetos, tenemos que hacer que ciertas estructuras en jaguar.structures se extiendan para ofrecer estas características gráficas. Estas extensiones se encuentran en jaguar.structures.jstructures. En este paquete se puede encontrar a jaguar.structures.jstructures.JState, clase que representa estados en las interfaces gráficas; extiende a jaguar.structures.State agregando información como: posición en el lienzo, color del estado, forma y dimensiones con las que se muestra el estado actual y se distingue gráficamente a los estados finales e iniciales.

2.1.1 Interfaces gráficas para máquinas

Las similitudes, entre los distintos modelos también nos sugieren que la forma gráfica para representarlos no debería variar mucho. Así, desarrollamos una infraestructura gráfica genérica, con las características más comunes, que puede ser usada por todos los modelos. Esta infraestructura es especializada por cada uno de los modelos. Todo esto lo logramos usando clases abstractas e interfaces de Java.

Para comenzar, tenemos una interfaz JMachine del paquete jaguar.machine, donde sólo definimos constantes y firmas de métodos, los cuales deben ser implementados por los motores de las máquinas que quieran ser mostradas gráficamente, i.e., cada vez que queramos hacer la extensión gráfica de una máquina, ésta debe de cumplir con el contrato

definido por la interfaz. Esto permitirá hacer uso de todas las utilerías gráficas genéricas para las JMachines que implementamos.

Las utilerías gráficas de uso genérico para cualquier máquina que implemente JMachine se encuentran en el paquete jaguar.machine.util.jutil. Entre otras clases tenemos:

- jaguar.machine.util.jutil.JMachineFrame, es una clase abstracta que nos da un marco que contiene un menú genérico que nos permite: cargar definiciones de máquinas, cargar cadenas de archivos para probar la máquina antes cargada, además de construir cadenas de prueba de forma interactiva. Además genera botones de control para la ejecución de las máquinas y contiene un lienzo donde se dibujarán los diagramas. Así, cada que extendemos una máquina y al implementar la interfaz gráfica podemos hacer uso de este marco genérico especializándolo; por ejemplo, para las máquinas de stack agregamos un componente stack y para las máquinas de Turing una cinta sobre la cual se puede mover la cabeza lectora/escritora de la máquina.

- jaguar.machine.util.jutil.JMachineCanvas, esta clase proporciona el lienzo donde la JMachine se muestra con su representación de diagrama. Esta clase también se encarga de detectar todos los eventos que se llevan a cabo sobre el lienzo, como: reacomodar estados cuando son arrastrados[1] con el ratón; cuando uno posiciona el apuntador de ratón se encarga de mostrar un breviario[2] con las transiciones para las cuales está definida dicho estado. Por último, los elementos que se dibujan son los estados y las transiciones, con sus etiquetas respectivas. Esta clase está fuertemente relacionada con jaguar.machine.util.jutil.JDeltaPainter.

- jaguar.machine.util.jutil.JDeltaGraphic, es una interfaz que debe ser implementada por todas las extensiones gráficas de la función de transición δ de cada modelo de máquina gráfica. Proporciona ciertas características para que pueda ser mostrada la representación en diagrama de una máquina. Entre sus principales firmas se encuentra la que regresa el breviario de un estado dado.

- jaguar.machine.util.jutil.JDeltaPainter, esta clase es la que, dada una función de transición δ gráfica que implemente la interfaz anterior JDeltaGraphic, lleva a cabo el proceso de pintar los estados y las flechas que representan las transiciones, así como las etiquetas de estas flechas.

Una de las herramientas más útiles que no interviene propiamente en la implementación de una máquina gráfica es la siguiente clase:

- jaguar.machine.util.jutil.JStrConstructor, es la interfaz gráfica que nos permite construir una cadena que le pasaremos a la máquina para su ejecución. Construye la cadena en base a Σ de la extensión de Machine asociada. El uso de esta interfaz se describe en su momento.

[1] drag & drop
[2] tooltip

2.1.2 Carga de definiciones de máquinas

Una de las operaciones que nos ofrece jaguar.machine.util.jutil.JMachineFrame es la de cargar la definición de una máquina para mostrarla en el lienzo de este marco y poder trabajar con ella. Para llevar a cabo esta tarea, vamos a la opción File → Load Machine... (figura 2.1) ó simplemente oprimiendo la tecla F2, lo cual nos mostrará una interfaz –como se observa en la figura 2.2– para seleccionar el archivo donde tenemos definida la extensión particular de la clase JMachine.

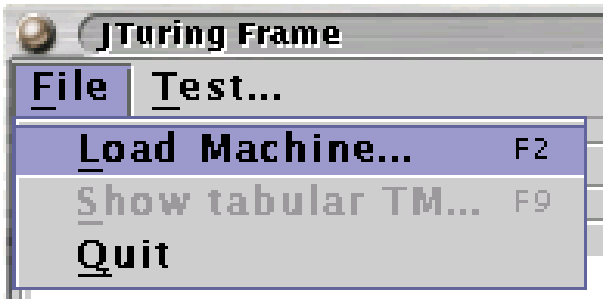

Figura 2.1: Menú para cargar una máquina de archivo

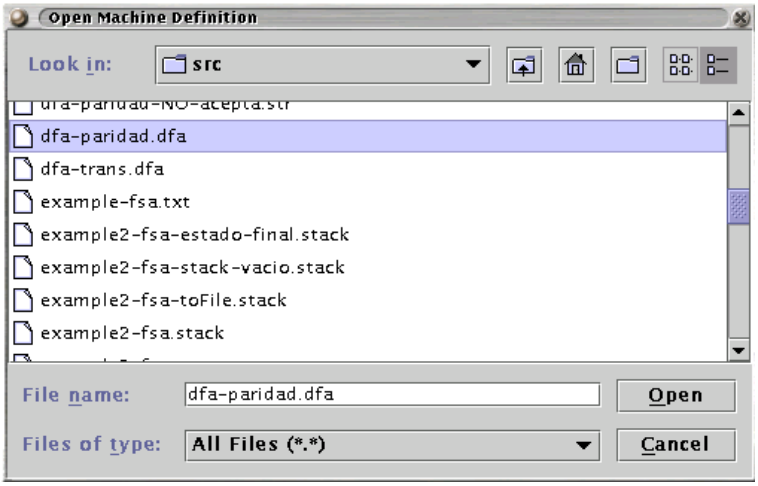

Figura 2.2: Seleccionando la máquina de un archivo

Una vez seleccionada y cargada la máquina particular, se mostrará la representación del diagrama de transiciones de dicha máquina en el lienzo asociado. El campo de configuración actual mostrará el estado inicial (q_0) y la cadena a probar ϵ. La distribución que asumirán los estados en el lienzo se lleva a cabo de la siguiente forma:

- Definimos un círculo de *radio* = min(`lienzo.alto`, `lienzo.ancho`).
- Ahora distribuiremos uniformemente los centros de cada JState alrededor del círculo. Esto es sencillo, pues sólo dividimos 360 entre la cardinalidad de Q y encontramos la

posición de cada estado por medio de coordenadas polares $(x, y) = (r\cos\theta, r\sin\theta)$. Después hacemos una translación al centro del lienzo.

2.1.3 Carga de cadenas de pruebas

Este proceso es completamente análogo al proceso de cargar la definición de una máquina de un archivo que se describe en la sección 2.1.2.

2.1.4 Construcción de una cadena de prueba gráficamente

La construcción de una cadena con la clase jaguar.machine.util.jutil.JStrConstructor se hace de la siguiente manera: una vez que se está en la interfaz gráfica, se selecciona este tipo de constructor con la secuencia de Test... ⟶ Build test String o sólo oprimiendo la tecla F5. Para ello tenemos una interfaz con un componente llamado *Sigma*, el cual nos muestra el alfabeto de entrada; un componente llamado *String built*, el cual nos muestra la cadena que estamos creando; y tres botones: Add Symbol, Backspace y ok, como se observa en la figura 2.3.

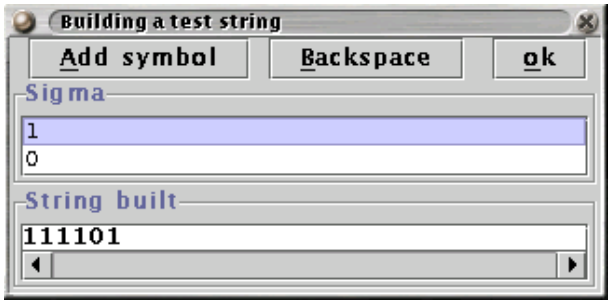

Figura 2.3: Construyendo cadenas gráficamente

El proceso para construir una cadena es el siguiente: tenemos que seleccionar un símbolo de *Sigma*; para agregarlo a la cadena presionamos el botón Add Symbol o hacemos doble *click* sobre dicho símbolo. Con el botón Backspace eliminamos el último símbolo de la cadena. Todos estos cambios se reflejarán en el componente *String built*. Cuando estemos satisfechos con la cadena construida, basta con presionar el botón ok. Si el campo de cadena creada no tiene símbolos y presionamos ok, el constructor creará la cadena vacía mostrándonos un mensaje de advertencia.

La cadena construida será mostrada en el componente *String* de la extensión del JMachineFrame que se haya implementado.

2.2 Descripción de máquinas

Para *programar* una máquina particular procedemos a describirla en un archivo XML, basado en las marcas definidas en su respectivo DTD. Esto nos permite concentrarnos únicamente en el diseño de la Máquina en cuestión.

Las descripciones sintácticas de los componentes principales se encuentran a continuación.

Símbolo. Un símbolo puede ser cualquier secuencia de carácteres dentro de las marcas `<sym> </sym>`. Formalmente:

```
<!ELEMENT sym (#PCDATA)>
```

Por ejemplo:

```
<sym> hola mundo </sym>
```

hay que tener cuidado pues dentro de la marca `<sym>` importan los espacios en blanco. Así, los símbolos: `<sym>foo foo</sym>` y `<sym> foo foo </sym>` son dos símbolos distintos.

Alfabeto. Se define como una colección de símbolos y la marca que usaremos es `<alph>`. Formalmente:

```
<!ELEMENT alph (sym)*>
```

Por ejemplo, el alfabeto { a, b, kkk, bar } estaría representado por:

```
<alph> <sym>a</sym> <sym>b</sym> <sym>kkk</sym> <sym>bar</sym>
                                 </alph>
```

Cadena. Se define como una sucesión de símbolos o la cadena vacía (ϵ), los cuales deben estar dentro de las marcas asociadas `<str> </str>`. Formalmente:

```
<!ELEMENT str (epsilon|(sym)+)>
```

Por ejemplo la cadena "foo" estaría representada por:

```
<str> <sym>f</sym> <sym>o</sym> <sym>o</sym> </str>
```

ϵ. La cadena vacía tiene una forma especial, definida formalmente como:

```
<!ELEMENT epsilon EMPTY>
```

Así, puede ser representada como `<epsilon> </epsilon>` o como `<epsilon/>`, está última no necesita otra marca, se cierra sola y como se definio en cadena, debe estár dentro de las márcas de cadena.

State. Un estado se identificará como una "etiqueta" la cual puede ser cualquier secuencia de carácteres dentro de las marcas `<state> </state>`. Formalmente:

```
<!ELEMENT state (#PCDATA)>
<!ATTLIST state xpos CDATA #IMPLIED>
<!ATTLIST state ypos CDATA #IMPLIED>
```

Por ejemplo:

```
<state>IMPAR</state>
<state xpos=''10'' ypos=''50''>PAR</state>
```

son dos estados, uno con la etiqueta "IMPAR" y el segundo con la etiqueta "PAR" y los atributos xpo y ypos con valores asignados.

Conjunto de estados. Es una colección de estados dentro de las marcas <statesSet> </statesSet>. Formalmente:

```
<!ELEMENT stateSet (state)*>
```

por ejemplo:

- $\{PAR, IMPAR\}$ se representa como:
  <statesSet><state>PAR</state> <state>IMPAR</state></stateSet>
- $\{\}$ se representa como: <statesSet> </stateSet>

Descripción. Es un elemento que contiene texto dentro de las marcas <description> </description>. Formalmente:

```
<!ELEMENT description (#PCDATA)>
```

Esta descripción será para describir el tipo de lenguaje que acepta o genera (según sea el caso) una máquina o gramática dada, y para resaltar hacer comentarios visibles desde las aplicaciones.

2.3 Implementación de convertidores

Además de dar la descripción gráfica de los modelos que iremos revisando, es importante ver la equivalencia de ciertos modelos entre sí. Dada esta equivalencia, podemos pasar de un modelo a otro. El proceso de conversión es hecho por un motor y para mostrar los resultados generados decidimos usar interfaces gráficas homogéneas. Para lograr esto, usamos la siguiente interfaz y clase abstracta de Java:

- jaguar.util.jutil.JConverter, es una interfaz en Java la cual nos compromete a implementar un método llamado doConvertion, entre otras cosas. Así, basta con implementar esta interfaz en los motores de conversión que deseemos hacer para poder usar la interfaz gráfica genérica de conversión.

- jaguar.util.jutil.JConverterFrame, esta clase abstracta tiene tres componentes básicos, como se muestra en la figura 2.4:

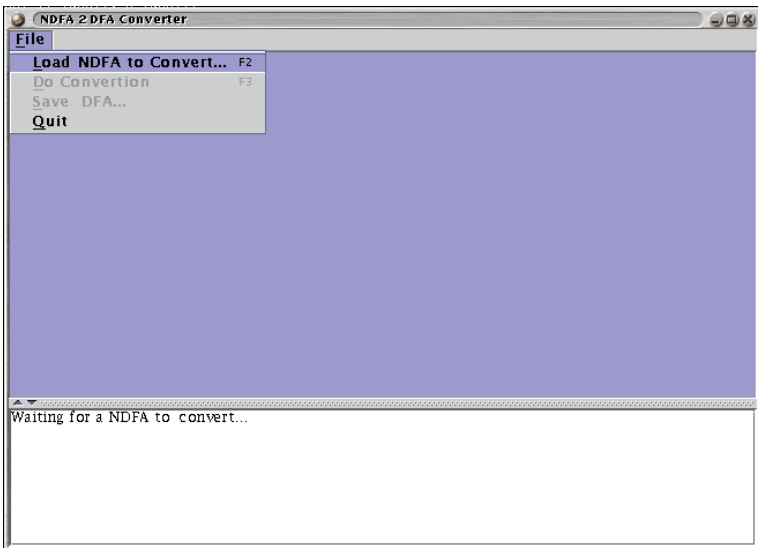

Figura 2.4: Tipo de interfaz que proporciona JConverter

Menú. Este componente nos ofrece un menú con cuatro botones que usaremos en un convertidor, estos son: (Load object to convert), (Do Convertion), (Quit) y en algunos (Save).

Panel de marcos. En este componentes se desplegarán varios marcos sencillos que contendrán los lienzos asociados de los objetos a convertir y sus equivalentes resultantes.

Área de detalles. Ésta nos mostrará cada uno de los pasos que fue realizando el motor de conversión.

Los dos últimos componentes descritos pueden expandirse abarcando la totalidad del marco. De esta forma podemos centrar nuestra atención en la representación gráfica de los objetos en cuestión, o bien, en los pasos que fue realizando el algoritmo de conversión.

Esta clase define una clase interna que especializa a los marcos que despliegan resultados, dotándolos de un menú para guardar el resultado de la conversión.

Ahora, para implementar un convertidor gráfico en particular, lo único que tenemos que hacer es extender esta clase con el JConverter adecuado, así como decir de qué tipo son el objeto original y el objeto que resulta de la conversión, y crear los lienzos de estos dos objetos; recordemos que los lienzos saben mostrar gráficamente a sus objetos asociados. Finalmente, tenemos que llamar a la constructora de la super clase con las cadenas que tienen los nombres de los botones y los breviarios adecuados para cada uno de los convertidores. Por ejemplo, si vamos a extender la interfaz gráfica del convertidor de JNDFA a JDFA, las cadenas que le pasaríamos al convertidor serían: *"Load NDFA to Convert..."*, *"Loads a new NDFA"*, *"Do Convertion"*, *"Convert the loaded NDFA to an equivalent DFA"*.

2.4 Uso de los convertidores

El uso de estas interfaces es muy sencilla e intuitiva. Usaremos como ejemplo el convertidor de NFA a DFA. En los demás convertidores son equivalentes los procesos usados en estos ejemplos.

Primero cargamos la descripción del objeto deseado, según el tipo de convertidor; el objeto se mostrará en un marco contenido en el *Panel de marcos*. Un ejemplo se muestra en la figura 2.5.

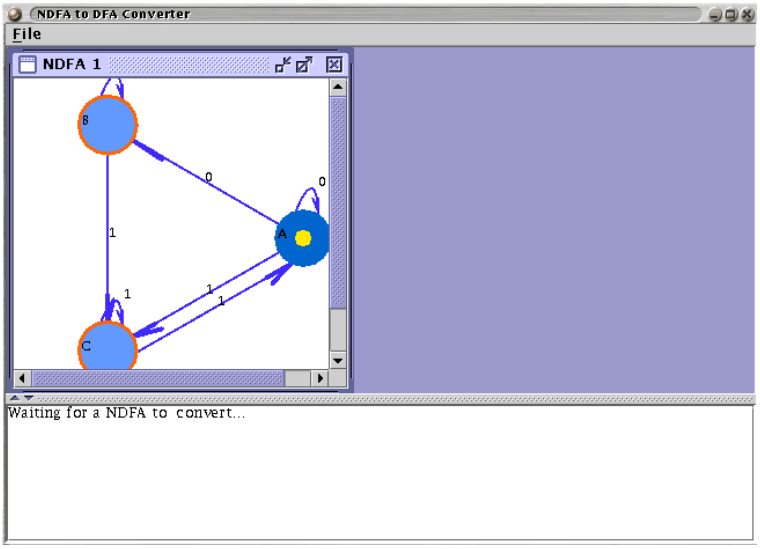

Figura 2.5: Cargando un NFA al convertidor

Como se puede observar, el marco en el que se muestra, en este caso, el JNDFA, es un marco que se puede minimizar, maximizar y eliminar. Además, el título del marco es *NDFA 1*, pues cada vez que cargamos un NFA se le asocia un número que también tendrá el DFA equivalente. Este número se incrementa automáticamente.

Una vez con el NFA cargado podemos llevar a cabo la conversión. Para esto basta con usar los botones de menú File $\longrightarrow$ Do Convertion u oprimir la tecla F3[3]. En cuanto hagamos esto, aparecerá un nuevo marco sencillo con el DFA equivalente en el *Panel de marcos* y, en el *Área de detalles*, se mostrarán todos los pasos realizados por el algoritmo. Esto se muestra en la figura 2.6

[3]La tecla F3 funciona igual para todos los convertidores gráficos.

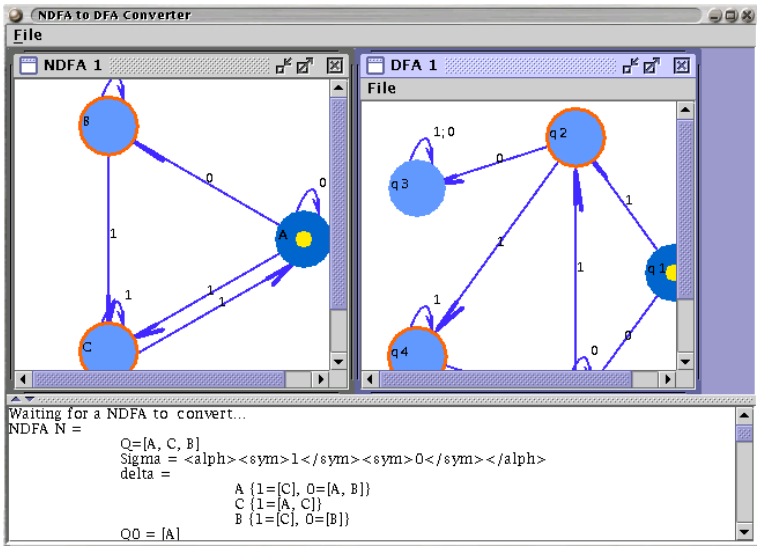

Figura 2.6: Convirtiendo un NDFA a DFA

A partir de este punto podemos hacer varias cosas: guardar el DFA equivalente al NFA definido; cargar la definición de otro NDFA el cual se mostrará en otro marco sencillo con el número en el título incrementado en uno; o revisar los detalles en la ejecución del algoritmo, como se muestran en la figura 2.7.

Para guardar cualquier máquina resultante del proceso de conversión basta con llevar a cabo la secuencia File $\longrightarrow$ Save Machine... o sólo teclear F6 en el marco interno que se quiera guardar.

Estos procesos son análogos en todos los convertidores gráficos que presentemos en adelante.

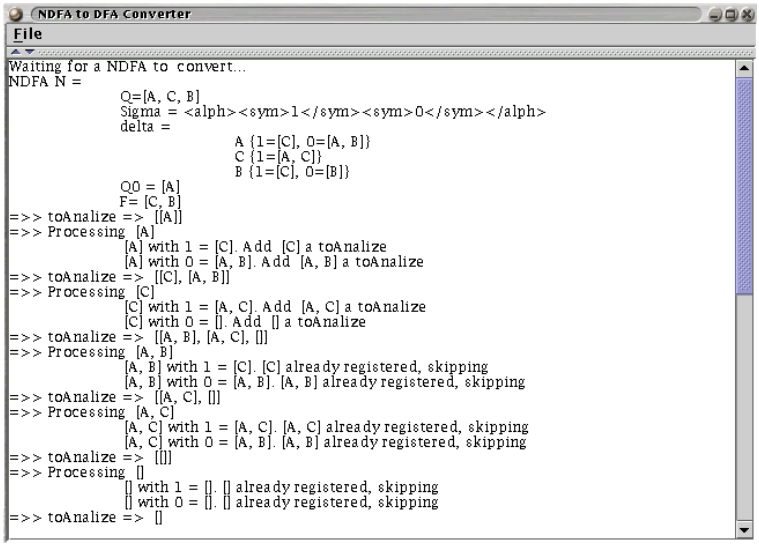

Figura 2.7: Detalles de la conversión *NDFA 1* a *DFA 1*

2.5 De uso global

2.5.1 Formateo de cadenas

Dada la especificación de símbolos y cadenas que damos en la introducción de este capítulo es claro que podemos tener símbolos de más de un carácter de computadora. Por ello es que necesitamos mecanismos para poder identificarlos de manera clara y sin ambigüedades. Estos mecanismos los proporciona la clase jaguar.util.jutil.MachineGrammarStyledDocumment que extiende a javax.swing.text.DefaultStyledDocument. La manera en que formatea cadenas de tipo Str es asignándoles un entorno[4] de distintos colores. Esta clase es utilizada por las máquinas y gramáticas en su representación gráfica.

Pasamos ahora a describir las especializaciones para cada uno de los modelos que se revisan en el curso, extendiendo la infraestructura básica descrita en este capítulo

[4] *background*

Capítulo 3

Autómatas Finitos

Los autómatas finitos son el modelo más sencillo que revisaremos. Son modelos de cómputo con una cantidad de memoria limitada. Aún con esta limitación, estos autómatas son de gran utilidad en ciertas disciplinas de la computación como diseño de sistemas dígitales, comunicaciones y en cualquier situación que implique procesamiento de señales que conlleven información. En particular son de gran utilidad en el diseño de compiladores.

Formalmente, un autómata finito M es un quíntuplo

$$M = (Q, \Sigma, \delta, q_0, F)$$

donde

Q es un conjunto finito de estados,
Σ es el alfabeto de entrada,
$\delta : Q \times \Sigma \longrightarrow Q$ la función de transición,
$q_0 \in Q$ es el estado inicial, y
$F \subseteq Q$ es el conjunto de estados finales

3.1 Implementación del motor

Esta parte consiste de diseñar e implementar un Autómata Finito (DFA) genérico. Para esto definimos un paquete que contiene el *motor* y estructuras que modelan los componentes especifícos de un DFA. La figura 3.1 muestra esta jerarquía.

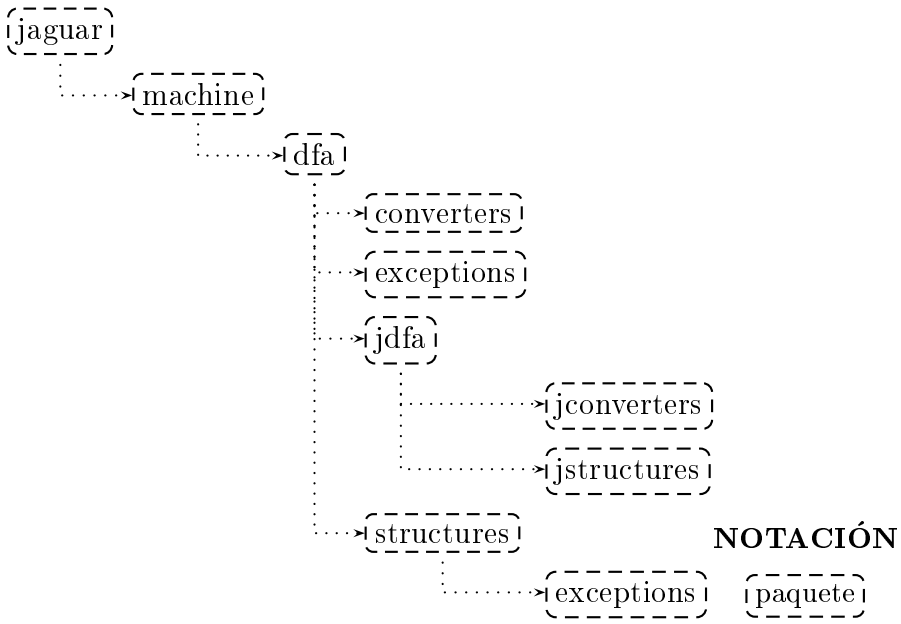

Figura 3.1: Paquetes para los autómatas finitos

Las clases definidas y especializadas son:

- jaguar.machine.dfa.DFA, el motor extiende a la clase Machine, recibe la especificación de un DFA, i.e. un quíntuplo que define su comportamiento, y opcionalmente una cadena sobre la cuál se ejecuta.
- `jaguar.machine.dfa.structures.DfaDelta`, la función de transición

$$\delta : Q \times F \longrightarrow Q$$

  Esta función extiende a la clase Delta.

3.2 Descripción de autómatas finitos

Describamos los elementos que usaremos para la especificación de un DFA:

Transición. Es una tercia dentro de las marcas `<trans> </trans>` donde el primer elemento es un estado, el segundo un símbolo y el tercero un estado. Formalmente:

```
<!ELEMENT trans (state, sym, state)>
```

Por ejemplo:

```
<trans> <state>PAR</state> <sym>1</sym> <state>IMPAR</state> </trans>
```

Función de Transición δ Es una colección de transiciones, $\delta : Q \times \Sigma \longrightarrow Q$ con Q previamente definido lo mismo que Σ. Estas transiciones deben de estar dentro de las marcas `<delta> </delta>`. Formalmente:

```
<!ELEMENT delta (trans)*>
```

Por ejemplo, si $\boldsymbol{\delta}$ consistiera de las transiciones $\boldsymbol{\delta}$(PAR , 0) = PAR y $\boldsymbol{\delta}$(PAR , 1) = IMPAR se escribiría:

```
<delta>
<trans> <state>PAR</state>
        <sym>0</sym>
        <state>PAR</state> </trans>
<trans>
        <state>PAR</state>
        <sym>1</sym>
        <state>IMPAR</state> </trans>
</delta>
```

DFA. Es un quíntuplo $M = (Q, \Sigma, \delta, q_0, F)$ especificado dentro de las marcas <dfa> </dfa> y debe tener, en orden, los siguientes elementos: una descripción del DFA (opcional), un conjunto de estados Q, un alfabeto de entrada Σ, una función de transición δ, un estado inicial q_0, y un conjunto de estados finales F. Formalmente:

```
<!ELEMENT dfa (description?, stateSet, alph, delta, state, stateSet)>
```

Por ejemplo el autómata finito determinístico checador de paridad sería:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE dfa SYSTEM "dfa.dtd">

<!-- Esto es un comentario -->
<!-- Un DFA  M es un quíntuplo (Q,Sigma,delta,q0,F) -->

<dfa>

<description> Este es un autómata que checa si el número de 1s es par
en una cadena sobre {0,1}* </description>

<!-- El conjutno de estados Q -->
<statesSet> <state>PAR</state> <state>IMPAR</state> </stateSet>

<!-- El alfabeto de entrada Sigma -->
<alph> <sym>0</sym> <sym>1</sym> </alph>

<!-- La función de transición delta -->
<delta>
<trans> <state>PAR</state>
        <sym>0</sym>
        <state>PAR</state> </trans>
<trans> <state>PAR</state>
        <sym>1</sym>
        <state>IMPAR</state> </trans>
<trans> <state>IMPAR</state>
```

```
27            <sym>0</sym>
28            <state>IMPAR</state> </trans>
29    <trans> <state>IMPAR</state>
30            <sym>1</sym>
31            <state>PAR</state> </trans>
32    </delta>
33
34    <!-- El estado inicial q0-->
35    <state>PAR</state>
36
37    <!-- El conjunto de estados finales F -->
38    <statesSet> <state>PAR</state> </stateSet>
39  </dfa>
```

Las líneas 1 y 2 muestran el *prolog* del documento, el cual indica que vamos a leer un `dfa`, verificando sea válido con respecto al DTD en `dfa.dtd`. En la línea 7 comenzamos con enunciar la marca para definir el DFA; las líneas 9 y 10 muestran una descripción de está máquina. En la línea 13 definimos Q; en la 16 a Σ; de la 19 a la 32 a δ; en la 35 a q_0; en la 38 a F; y, finalmente, en la línea 39 cerramos la marca que define al autómata.

3.3 Uso del motor

Este motor se puede ejecutar desde la consola, invocando a la clase `DFA` con dos parámetros: el primero, un archivo con una especificación de DFA como la descrita previamente y el segundo un archivo con la especificación de una cadena. Por ejemplo, si la especificación del checador de paridad estuviese en el archivo `dfa-paridad.dfa` y la especificación de una cadena `1010011` en otro llamado `cadena-paridad.str`, teclearíamos:

```
java jaguar.machine.dfa.DFA dfa-paridad.dfa cadena-paridad.str
```

La salida de ejecutar el comando anterior se muestra en la figura 3.2, donde aparece la configuración del autómata finito en cada instante de su ejecución, finalizando con un veredicto de aceptación.

```
d( PAR , 1010011 ) = IMPAR
d( IMPAR , 010011 ) = IMPAR
d( IMPAR , 10011 ) = PAR
d( PAR , 0011 ) = PAR
d( PAR , 011 ) = PAR
d( PAR , 11  ) = IMPAR
d( IMPAR , 1 ) = PAR
PAR

El DFA SI acepta 1010011
```

Figura 3.2: Ejecución del motor con la especificación del DFA Paridad

3.4 Generación de la interfaz gráfica

Ahora, para desarrollar la interfaz gráfica lo que hacemos es extender el motor y algunas estructuras adicionales. La idea es especializar a los componentes básicos con funciones y características adicionales, además de implementar características de las JMachines para mostrarse en una interfaz gráfica. Así, los componentes del DFA con interfaz gráfica JDFA son:

- jaguar.machine.dfa.jdfa.JDFA, es el autómata con interfaz gráfica que extiende a jaguar.machine.dfa.DFA e implementa jaguar.machine.JMachine. Cuenta con un lienzo donde se muestra el DFA con el diagrama de transiciones. La parte más importante del JDFA es que tiene como campo una extensión de la función de transición DfaDelta llamada JDfaDelta. Cada vez que se aplica una transición, esta clase hace una actualización en el lienzo para resaltar visualmente la transición usada y distinguir el estado actual del resto de los estados.

- jaguar.machine.dfa.jdfa.JDfaFrame, el marco que extiende a la clase JMachineFrame. La especialización que hace esta clase es mínima en cuanto a la apariencia, así que es la versión más *pura* de un JMachineFrame.

- jaguar.machine.dfa.jdfa.JDfaCanvas, es aquí donde cada uno de los elementos del JDFA se dibuja y donde se observa el funcionamiento del autómata. Extiende a jaguar.machine.util.jutil.JMachineCanvas. Los elementos que se dibujan son los estados y las transiciones, con sus etiquetas respectivas.

- jaguar.machine.dfa.jdfa.jstructures.JDfaDelta, esta clase extiende a jaguar.machine.dfa.DfaDelta e implementa jaguar.machine.util.jutil.JDeltaGraphic. Se encarga de las transiciones y de establecer la configuración necesaria para que el JDeltaPainter dibuje el diagrama de transiciones en el entorno gráfico del lienzo antes mencionado. Distingue a los estados finales.

3.5 Uso de la interfaz gráfica

Para ejecutar la interfaz gráfica sólo hace falta invocar a JDfaFrame de la siguiente forma:

```
java jaguar.machine.dfa.jdfa.JDfaFrame
```

En seguida aparecerá el JDfaFrame como se muestra en la figura 3.3.

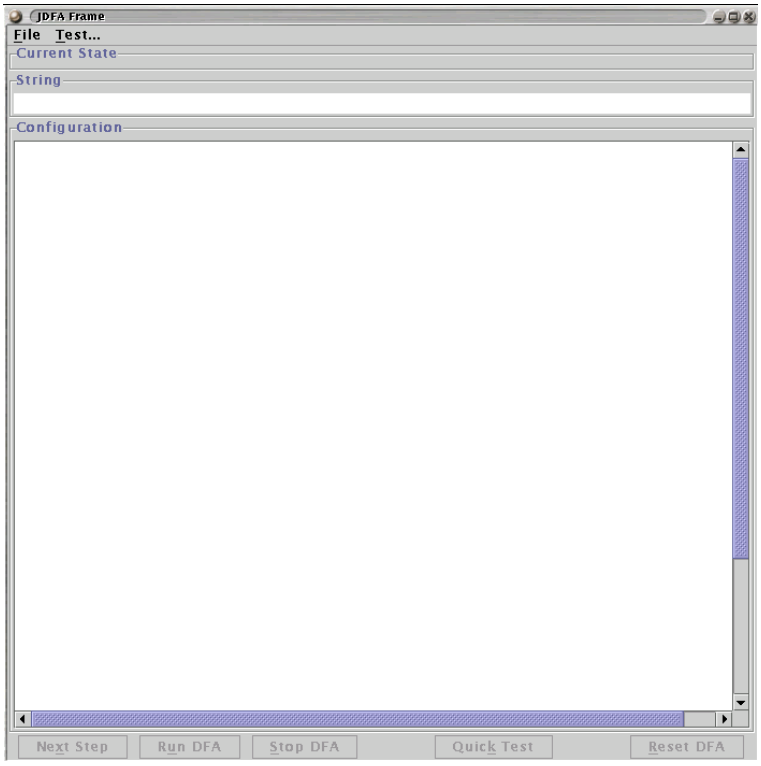

Figura 3.3: JDfaFrame

Podemos observar que tiene dos menús: File y Test... Además, cuenta con cuatro botones para controlar la ejecución con una cadena dada. Estos son:

- Next Step ejecuta una sola transición en el autómata sobre la cadena.
- Run ejecuta el autómata sobre toda la cadena en pequeños intervalos de tiempo.
- Stop detiene una ejecución iniciada con el botón antes descrito.
- Quick Test ejecuta el autómata sobre la cadena de una sola vez, sin mostrar las transiciones. Al final solo muestra el veredicto de pertenencia y lleva a cabo un reset, el cual tiene el mismo efecto que el siguiente botón.
- Reset DFA que restablece las condiciones iniciales de ejecución del autómata sobre la cadena dada.

Finalmente, tiene dos componentes: *Current State* y *String*, los cuales nos van mostrando las configuraciones por las que pasa el DFA (al iniciar no tenemos definido un DFA, por lo que la figura 3.3 no muestra nada en el lienzo).

Una vez que cargamos la descripción de un DFA M que acepta

$$L(M) = \{x001y \mid x, y \in \{0,1\}^*\}$$

de un archivo –como se describió en la sección 2.1.2– y que construimos la cadena 01100010101 con la interfaz gráfica –como se describió en la sección 2.1.4– tenemos el JDfaFrame como muestra la figura 3.4.

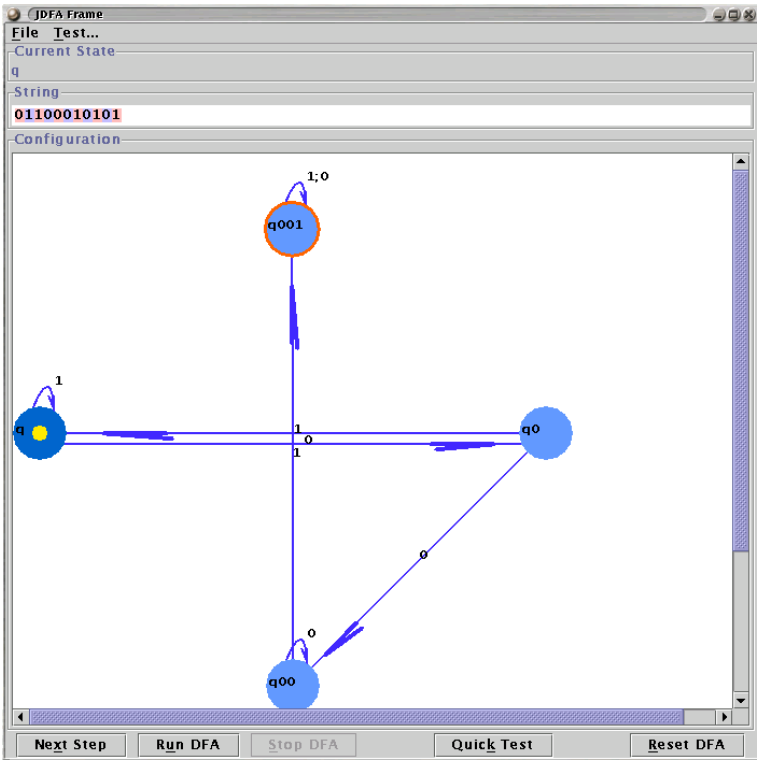

Figura 3.4: El autómata M que acepta $(0+1)^*001(0+1)^*$

Ejecutar el autómata sobre esta cadena resultará en una serie de transiciones que se mostrarán en el lienzo y en los componentes *Current State* y *String*. Al agotarse la cadena de entrada, aparecerá un mensaje de que el autómata acepta (o no) la cadena de entrada, y con esto finaliza la ejecución del autómata.

Los estados pueden ser reacomodados en el lienzo arrastrándolos, para una mejor percepción del autómata y sus transiciones. Por ejemplo, una distribución distinta del autómata M se muestra en la figura 3.5.

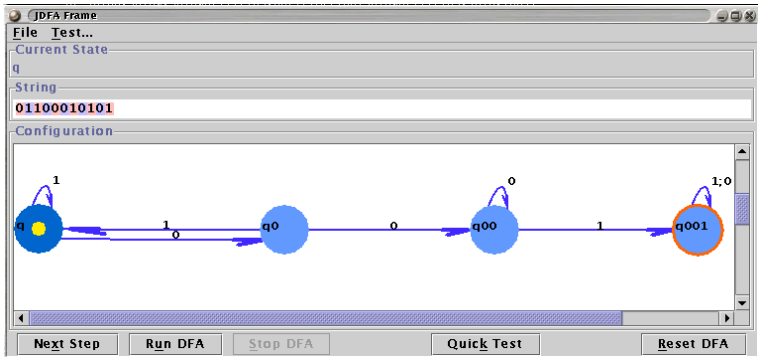

Figura 3.5: El autómata M reacomodado.

3.6 Autómatas finitos no determinísticos

Los autómatas finitos no determinísticos (NFA) son un caso general de los autómatas finitos (DFA). La única diferencia entre éstos es la función de transición, la cual permite que en un estado y al leer un símbolo del alfabeto de entrada, el autómata se transfiera no a un solo estado, sino que lo haga simultáneamente a más de uno: que haya varias posibilidades para el estado sucesor.
Formalmente, un NFA es un quíntuplo

$$N = (Q, \Sigma, \delta, Q0, F)$$

donde

Q	es un conjunto finito de estados,
Σ	es el alfabeto de entrada,
δ	es una función que da la siguiente transición, definida de la siguiente manera: $$Q \times \Sigma \longrightarrow \mathcal{P}(Q)$$
$Q0 \subseteq Q$	es el conjunto de estados iniciales, y
$F \subseteq Q$	es el conjunto de estados finales

3.6.1 Implementación del motor

En el paquete de autómatas finitos (figura 3.1) y sus estructuras, definimos:

- `jaguar.machine.dfa.NDFA`, el motor del NFA, que recibe la especificación de un NFA –i.e. un quíntuplo que define su estructura y comportamiento– y una cadena sobre la cuál se ejecutará.

- `jaguar.machine.dfa.structures.DfaDelta`, la función de transición

$$\delta : Q \times F \longrightarrow \mathcal{P}(Q)$$

3.6.2 Descripción de NFA

Para especificar un NFA por medio de etiquetas sólo tenemos que redefinir un par de elementos ya definidos para DFA:

Transición. Es una tercia dentro de las marcas `<trans> </trans>` donde el primer elemento es un estado, el segundo un símbolo y el tercero un estado. Formalmente:

```
<!ELEMENT trans (state, sym, stateSet)>
```

Por ejemplo, si tuviésemos definida la transición $\delta(P, a) = \{p0, p1, p2\}$ tendríamos dentro de la delta una transición como:

```
<trans> <state>P</state> <sym>a</sym>
        <stateSet> <state>p0</state>
                   <state>p1</state>
                   <state>p2</state>
        </stateSet>
</trans>
```

Función de Transición δ. Es una colección de transiciones, $\boldsymbol{\delta} : Q \times \Sigma \longrightarrow \mathcal{P}(Q)$ con Q previamente definido, lo mismo que Σ. Estas transiciones deben estar dentro de las marcas `<delta> </delta>`. Formalmente:

```
<!ELEMENT delta (trans)*>
```

Por ejemplo, la regla de transición $\boldsymbol{\delta}$(`PAR` , `0`) = { `PAR`, `IMPAR` } se escribiría:

```
<delta>
     <trans> <state>PAR</state> <sym>0</sym>
             <stateSet> <state>PAR</state>
                        <state>IMPAR</state>
             <stateSet>
     </trans>
</delta>
```

NFA. Es un quíntuplo $M = (Q, \Sigma, \delta, q_0, F)$ especificado dentro de las marcas `<ndfa> </ndfa>` y debe tener **en orden** los siguientes elementos: una descripción del NFA (opcional),un conjunto de estados Q, un alfabeto de entrada Σ, una función de transición δ, un conjunto de estados inicial Q_0, y un conjunto de estados finales F. Formalmente:

```
<!ELEMENT ndfa (description?, stateSet, alph, delta, stateSet, stateSet)>
```

Por ejemplo :

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <!DOCTYPE ndfa SYSTEM "ndfa.dtd">
3
4  <!-- Esto es un comentario -->
5
```

```
6    <ndfa>
7
8    <!-- Conjunto de estados Q -->
9     <stateSet>
10         <state>A</state>
11         <state>B</state>
12         <state>C</state>
13    </stateSet>
14
15   <!-- Alfabeto de entrada Sigma -->
16    <alph>
17         <sym>0</sym>
18         <sym>1</sym>
19    </alph>
20
21   <!-- Función de transición delta -->
22    <delta>
23         <trans> <state>A</state> <sym>0</sym>
24                 <stateSet><state>A</state> <state>B</state> </stateSet>
25         </trans>
26         <trans> <state>A</state> <sym>1</sym>
27                 <stateSet><state>C</state>  </stateSet>
28         </trans>
29         <trans> <state>B</state> <sym>0</sym>
30                 <stateSet><state>B</state> </stateSet>
31         </trans>
32         <trans> <state>B</state> <sym>1</sym>
33                 <stateSet><state>C</state> </stateSet>
34         </trans>
35         <trans> <state>C</state> <sym>1</sym>
36                 <stateSet><state>A</state> <state>C</state>  </stateSet>
37         </trans>
38    </delta>
39
40   <!-- Conjunto de estados  iniciales  Q0 -->
41    <stateSet> <state>A</state>  <state>B</state>    </stateSet>
42
43   <!-- Conjunto de estados finales F -->
44    <stateSet>
45            <state>B</state>
46            <state>C</state>
47    </stateSet>
48
49 </ndfa>
```

3.6.3 Uso del motor

Este motor puede ejecutarse desde la consola, invocando a la clase `NDFA` con dos parámetros: el primero, un archivo con una especificación de NFA como la descrita previamente y el segundo un archivo con la especificación de una cadena. Por ejemplo, si tenemos la especificación del autómata no determinístico que se muestra en la tabla 3.1 en el archivo `tabla1.ndfa` y la especificación de una cadena `011011` en otro llamado `cadena-test.str`, ejecutaríamos el programa:

```
java jaguar.machine.dfa.NDFA tabla1.ndfa cadena-test.str
```

Autómata no determinístico			
	Σ		
Q	0	1	F
A	A, B	C	0
B	B	C	1
C	$\emptyset$	A, C	1

Tabla 3.1: Autómata finito no determinístico

La salida de ejecutar el comando anterior se muestra en la figura 3.6, donde aparece la definición del NFA y las configuraciones por las que pasa en cada instante de su ejecución, finalizando con un veredicto de aceptación.

```
M =
        Q=[A, C, B]
        Sigma = <alph><sym>1</sym><sym>0</sym></alph>
        delta =
                A 1=[C], 0=[A, B]
                C 1=[A, C]
                B 1=[C], 0=[B]
        Q0 = [A]
        F= [C, B]
Q0 = [A]
d( A , 011011 ) = [A, B]
d( A , 11011 ) = [C]
d( C , 1011 ) = [A, C]
d( A , 011 ) = [A, B]
d( A , 11 ) = [C]
d( C , 1 ) = [A, C]
A
C

El NDFA SI acepta 011011
```

Figura 3.6: Ejecución del motor con la especificación del NFA de la tabla 3.1

La forma en que se ejecuta el NFA es recorriendo a profundidad el árbol generado por el no determinismo.

3.6.4 Generación de la interfaz gráfica

Al igual que con los autómatas finitos determinísticos, extendemos el motor y las estructuras gráficas de lienzo y marco. Los componentes del NFA con interfaz gráfica JNDFA son:

- jaguar.machine.dfa.jdfa.JNDFA, es el autómata no determinístico con interfaz gráfica que extiende a jaguar.machine.dfa.NDFA e implementa jaguar.machine.JMachine. Cuenta con un lienzo donde se muestra el DFA con la gráfica de transiciones. La parte más importante del JDFA es que tiene como campo una extensión de la función de transición NDfaDelta llamada JNDfaDelta. Cada vez que se aplica una transición, esta clase hace una actualización en el lienzo para resaltar visualmente la transición usada y distinguir el estado actual del resto de los estados.

- jaguar.machine.dfa.jdfa.JNDfaFrame, el marco que extiende a la clase JMachineFrame. Sólo cuenta con dos botones, uno para evaluar la cadena dada y otro para reiniciar la máquina llevándola a su configuración inicial.

- jaguar.machine.dfa.jdfa.JNDfaCanvas, es aquí donde cada uno de los elementos del JDFA se dibuja y donde se observa el funcionamiento del autómata. Extiende a

jaguar.machine.JMachineCanvas. Los elementos que se dibujan son los estados y las transiciones, con sus etiquetas respectivas.

- jaguar.machine.dfa.jdfa.jstructures.JNDfaDelta, esta clase extiende a jaguar.machine.dfa.DfaDelta e implementa jaguar.machine.util.jutil.JDeltaGraphic. Se encarga de las transiciones y de establecer la configuración necesaria para que JNDfaDeltaPainter dibuje el diagrama de transiciones en el entorno gráfico del lienzo antes mencionado. Distingue a los estados finales.

- jaguar.machine.dfa.jdfa.jstructures.JNDfaDeltaPainter, esta clase extiende a jaguar.machine.util.jutil.JDeltaPainter, modificando la función que pinta el diagrama en el lienzo, para pintar correctamente la transición de un estado fuente a varios destinos.

3.6.5 Uso de la interfaz gráfica

Para ejecutar la interfaz gráfica sólo hace falta invocar a JDfaFrame de la siguiente forma:

```
java jaguar.machine.dfa.jdfa.JNDfaFrame
```

En seguida aparecerá el JNDfaFrame y una vez cargada la descripción de una máquina, tendremos un marco como se muestra en la figura 3.7.

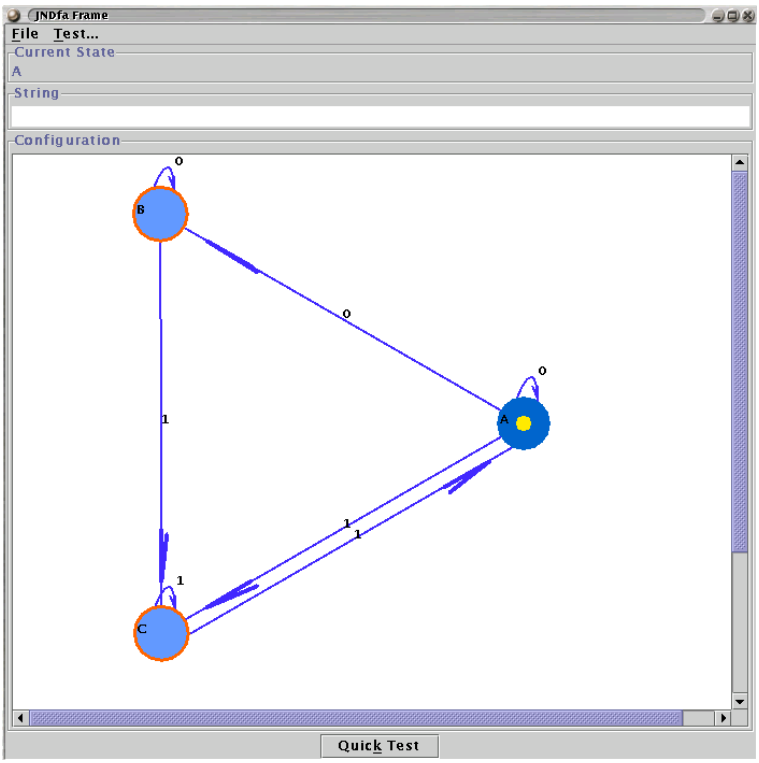

Figura 3.7: Marco para autómatas finitos no determinísticos

El marco es más sencillo que el de los autómatas finitos, pues sólo cuenta con el botón Quick Test que checa la pertenencia de la cadena dada sobre el lenguaje que acepta el autómata.

Capítulo 4

Autómatas con stack

En este capítulo revisaremos un modelo conocido como autómatas de stack. Estos autómatas son parecidos a los autómatas finitos pero tienen un componente adicional que es un *stack*. El stack le proporciona memoria adicional a la de los estados.

Un autómata de stack (SFA), puede escribir símbolos en el stack y leerlos más adelante durante la ejecución. Al escribir un símbolo en el tope[1], se dice que el stack empuja al fondo a los demás símbolos del stack. En cualquier momento de la ejecución se puede leer un símbolo en el tope del stack y adicionalmente ser sustituido por una cadena, que ocupará posiciones consecutivas en el stack.

Esta estructura de stack es valiosa porque nos brinda una cantidad ilimitada de información. Recordemos que con un DFA no es posible reconocer el lenguaje $\{a^n b^n \mid n \geq 0\}$, pero con un SFA esto es posible; más aún, fácil; sólo tenemos que almacenar todas las a que veamos en el stack y sacarlos conforme se procese la secuencia final de bs.
Formalmente, un autómata de stack P es un séptuplo

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

donde

Q	es un conjunto finito de estados,
Σ	es el alfabeto de entrada,
Γ	es el alfabeto del stack,
δ	es una función que da la siguiente transición, definida de la siguiente manera $$\delta : Q \times \Sigma \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma^*)$$
$q_0 \in Q$	es el estado inicial,
Z_0	es el símbolo en el fondo del stack al empezar a funcionar el SFA, y
$F \subseteq Q$	es el conjunto de estados finales

4.1 Implementación del motor

La primera parte consiste en el diseño e implementación de un autómata con stack. Para esto definimos un paquete que contendrá el motor y estructuras que modelan a cada uno de los componentes de un SFA, ver figura 4.1.

[1]La estructura de stack restringe la manera de escribir, de leer y de eliminar elementos de éste. Es una estructura donde el último símbolo que entró será el primero que salga.

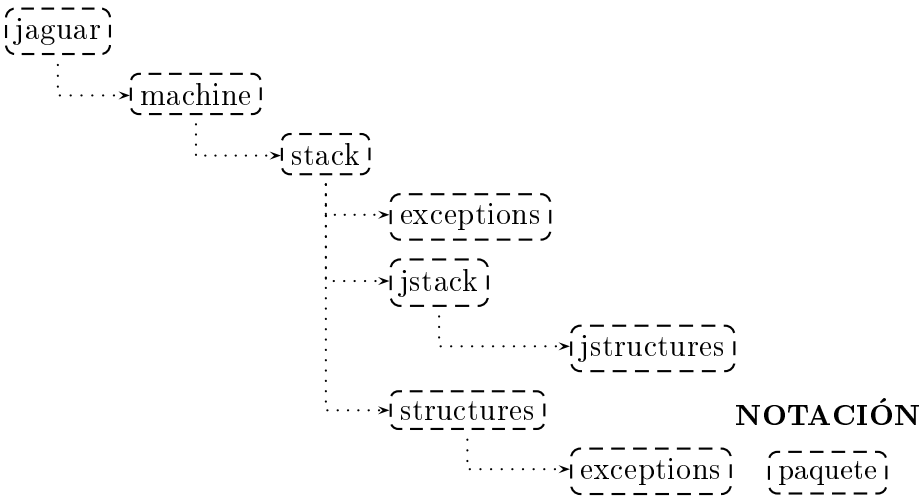

Figura 4.1: Paquetes para el AFS

Las clases definidas son:

- `jaguar.machine.stack.AFS`, el motor, recibe la especificación de un SFA, i.e. un séptuplo que define su comportamiento. El criterio que sigue el SFA para aceptar una cadena se sigue del conjunto de estados finales F. Si este conjunto no tiene elementos, entonces acepta por stack vacío; de otra forma acepta por estado final.

- `jaguar.machine.stack.structures.QxGammaStar`, estructura para almacenar a un par ordenado en $Q \times \Gamma^*$

- `jaguar.machine.stack.structures.QxGammaStarSet`, conjuntos de pares ordenados $Q \times \Gamma^*$

- `jaguar.machine.stack.structures.StackDelta`, la función de transición

$$\delta : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma^*)$$

4.1.1 Descripción de SFA

Para especificar un SFA por medio de marcas sólo tenemos que definir o redefinir los siguientes elementos:

QxGammaStar es un par dentro de las marcas `<QxGammaStar> </QxGammaStar>`, donde el primer elemento es un estado en Q y el segundo es una cadena sobre Γ. Formalmente:

```
<!ELEMENT QxGammaStar (state, str)>
```

QxGammaStarSet es un conjunto dentro de las marcas `<QxGammaStarSet> </QxGammaStarSet>`, donde los elementos son estructuras QxGammaStar, justo como se acaban de definir. Formalmente:

```
<!ELEMENT QxGammaStarSet (QxGammaStar)*>
```

Transición es un cuarteto dentro de las marcas `<trans> </trans>` donde el primer elemento es un estado en Q, el segundo un símbolo sobre Σ, el tercero un símbolo de Γ y por último un conjunto QxGammaStarSet como se acaba de definir. Formalmente:

```
<!ELEMENT trans (state, sym, sym, QxGammaStarSet)>
```

Por ejemplo, si tuviésemos definida la transición

$$\delta(q0, a, Z0) = \{(q0, A)\}$$

tendríamos dentro de delta la transición:

```
<trans> <state>q0</state> <sym>a</sym> <sym>Z0</sym>
    <QxGammaStarSet>
      <QxGammaStar> <state>q0</state> <str><sym>A</sym></str>
      </QxGammaStar>
    </QxGammaStarSet>
</trans>
```

Función de Transición δ es una colección de transiciones,

$$\boldsymbol{\delta} : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma^*)$$

con todos los elementos previamente definidos. Estas transiciones deben estar dentro de las marcas `<delta> </delta>`. Formalmente:

```
<!ELEMENT delta (trans)*>
```

SFA es un séptuplo $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ especificado dentro de las etiquetas `<afs> </afs>` y debe tener, en orden, los siguientes elementos: una descripción del SFA (opcional), un conjunto de estados Q; un alfabeto de entrada Σ; un alfabeto del stack Γ; una función de transición δ; un estado inicial q_0; un símbolo en el fondo del stack al principio de su ejecución Z_0; y un conjunto de estados finales F. Formalmente:

```
<!ELEMENT stack (description?,stateSet, alph, alph, delta, state,
                 sym, stateSet)>
```

Por ejemplo:

```
1   <?xml version="1.0" encoding="iso-8859-1"?>
2   <!DOCTYPE stack SYSTEM "afs.dtd">
3
4    <stack>
5    <!-- El conjunto de estados Q -->
6     <stateSet> <state>q0</state> <state>q1</state> </stateSet>
7
8    <!-- Alfabeto de entrada Sigma -->
9     <alph> <sym>a</sym> <sym>b</sym> </alph>
10
```

```
11      <!-- Alfabeto del stack Gamma -->
12      <alph> <sym>Z0</sym> <sym>A</sym> </alph>
13
14      <!-- Función de transición delta -->
15      <delta>
16        <trans> <state>q0</state> <sym>a</sym> <sym>Z0</sym>
17                <QxGammaStarSet>
18                  <QxGammaStar> <state>q0</state> <str><sym>A</sym></str>
19                  </QxGammaStar>
20                </QxGammaStarSet>
21        </trans>
22
23        <trans> <state>q0</state> <sym>a</sym> <sym>A</sym>
24                <QxGammaStarSet>
25                   <QxGammaStar> <state>q0</state> <str> <sym>A</sym>
26                                                         <sym>A</sym>
27                                                   </str>
28                   </QxGammaStar>
29                </QxGammaStarSet>
30        </trans>
31      </delta>
32
33      <!-- Estado inicial q0 -->
34      <state>q0</state>
35
36      <!-- Símbolo en el fondo del stack, al inicio de la ejecución -->
37      <sym>Z0</sym>
38
39      <!-- El criterio para aceptar es por stack vacío, pues el conjunto
40      de estados finales F es vacío -->
41      <stateSet> </stateSet>
42
43    </stack>
```

4.1.2 Uso del motor

Este motor puede ejecutarse desde la consola, invocando a la clase `AFS` con dos parámetros: el primero, un archivo con una especificación de AFS como la descrita previamente y el segundo un archivo con la especificación de una cadena. Por ejemplo, si tenemos la especificación del autómata que acepta $L = \{a^n b^n | n \geq 1\}$ en el archivo `anbn.xml` y la especificación de una cadena `aaabbb` en otro llamado `cadena-test.xml`, ejecutaríamos el programa:

```
java jaguar.machine.stack.AFS anbn.xml cadena-test.xml
```

La salida de ejecutar el comando anterior se muestra en la figura 4.2. Primero vemos la definición del SFA, después a dos columnas: en la primera la configuración del SFA y en la segunda la regla que se usó para llegar a dicha configuración. Finalmente, despliega

el criterio para aceptar cadenas que éste usará (ver sección 4.1), el estado del stack, el estado de la cadena de entrada y, en base a esto, un veredicto de aceptación.

```
P =
        Q=[q1, q0]
        Sigma = <alph><sym>b</sym><sym>a</sym></alph>
        Gamma = <alph><sym>A</sym><sym>Z0</sym></alph>
        delta =
                 q1
                         b A=[( q1 , <epsilon/> ) ]
                 q0
                         b A=[( q1 , <epsilon/> ) ]
                         a A=[( q0 , AA ) ], Z0=[( q0 , A ) ]
        q0 = q0
        Z0 = Z0
        F= []

Configuraciones
 ( q0 , aaabbb, Z0 )
                        Regla ( q0 , aaabbb , Z0 ) = ( q0 , [( q0 , A ) ] )
 ( q0 , aabbb, A )
                        Regla ( q0 , aabbb , A ) = ( q0 , [( q0 , AA ) ] )
 ( q0 , abbb, AA )
                        Regla ( q0 , abbb , A ) = ( q0 , [( q0 , AA ) ] )
 ( q0 , bbb, AAA )
                        Regla ( q0 , bbb , A ) = ( q1 , [( q1 , <epsilon/> ) ] )
 ( q1 , bb, AA )
                        Regla ( q1 , bb , A ) = ( q1 , [( q1 , <epsilon/> ) ] )
 ( q1 , b, A )
                        Regla ( q1 , b , A ) = ( q1 , [( q1 , <epsilon/> ) ] )
 ( q1 , <epsilon/>,  )

Criterio para aceptar EMPTY_STACK

El estado del Stack es (vacío):
La cadena de entrada es: <epsilon/>

El AFS SI acepta aaabbb
```

Figura 4.2: Ejecución del motor con la especificación del SFA `anbn.xml`

4.2 Generación de la interfaz gráfica

Para desarrollar la interfaz gráfica lo que hacemos es extender el motor y algunas otras estructuras definidas para su funcionamiento, una vez más, especializando y agregando características adicionales para su despliegue gráfico, implementando la interfaz JMachine. Así, los componentes gráficos son:

- jaguar.machine.stack.jstack.JAFS. Extiende jaguar.machine.stack.AFS e implementa la interfaz jaguar.machine.JMachine. Modificando y agregando ciertas funcionalidades del motor para mostrarse en una interfaz gráfica. Cuenta con un lienzo donde se muestra la representación con diagrama de transiciones del SFA.

- jaguar.machine.stack.jstack.JAfsFrame. Es el marco que extiende a la clase jaguar.machine.util.jutil.JMachineFrame, la cual nos ofrece menús genéricos y botones de control para la ejecución. Una característica que se extiende es la de agregar al

marco dos componentes: *stack*, el cual muestra el estado del stack del SFA asociado al marco; y un componente *regla*, que es la regla que se usó para la configuración actual.

- jaguar.machine.stack.jstack.JAfsCanvas. Éste es el lienzo donde los elementos del SFA se dibujan y se observa su funcionamiento, sin considerar al stack, que se especifica en el punto anterior. Extiende a jaguar.machine.util.jutil.JMachineCanvas.

- jaguar.machine.stack.jstack.jstructures.JStackDelta. Es la clase que extiende a jaguar.machine.stack.structures.StackDelta e implementa jaguar.machine.util.jutil.JDeltaPainter, que se encarga de dibujar la representación del SFA en el lienzo.

4.3 Uso de la interfaz gráfica

Para hacer uso de la interfaz gráfica basta con ejecutar el JAfsFrame de la siguiente forma:

```
java jaguar.machine.stack.jstack.JAfsFrame
```

En seguida aparecerá el JAfsFrame como se muestra en la figura 4.3.

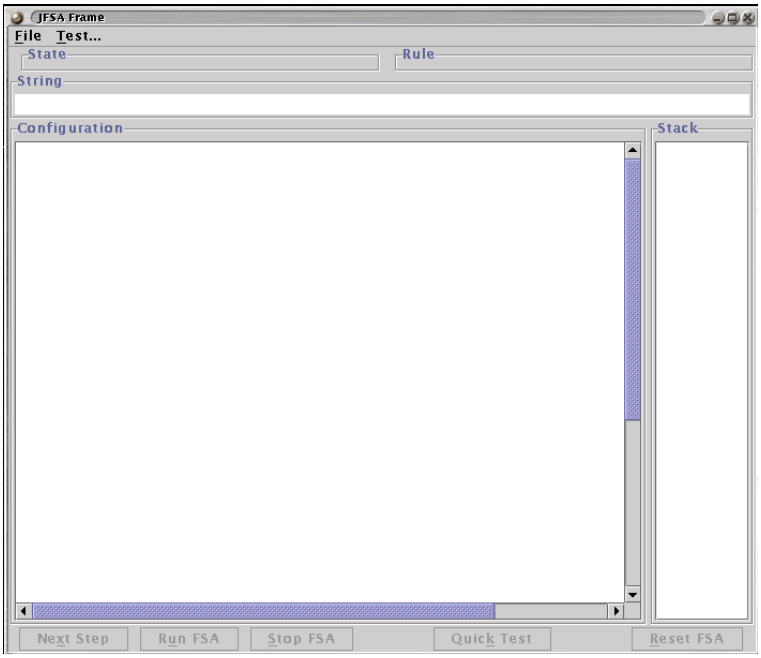

Figura 4.3: JAfsFrame

Además de los componentes ya conocidos: File, Test..., Next Step, Run FSA, Stop FSA, Quick Test, Reset FSA, *Current State* y *String*, tenemos dos nuevos:

- *Rule*, que se refiere a la regla que se usó para llegar a la configuración actual.

- *Stack*, que muestra la configuración actual del stack del SFA asociado, la cual cambia durante la ejecución sobre una cadena.

La figura 4.3 muestra los campos vacíos, pues no hemos cargado la definición de un SFA. Si sólo cargamos la definición de un autómata, la cadena a probar será ϵ. El criterio de aceptación que usará el SFA estará definido por el conjunto F: si este conjunto es vacío el SFA aceptará por stack vacío; si no, aceptará por estado final.

Ahora, una vez que cargamos la descripción del SFA P que acepta

$$L(P) = \{a^n b^n \ | n > 0\}$$

de un archivo como se describió en la sección 2.1.2, y que construimos la cadena *aaabbb* con la interfaz gráfica como se describió en la sección 2.1.4, tenemos el JAfsFrame como se muestra en la figura 4.4.

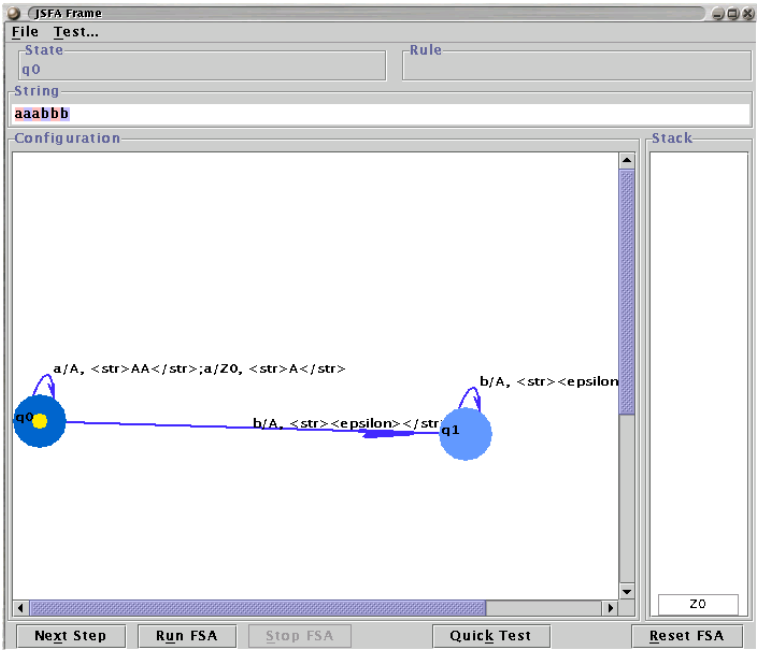

Figura 4.4: El autómata P que acepta $\{a^n b^n \ | n > 0\}$

Ejecutar el autómata sobre esta cadena resultará en una serie de transiciones que se mostrarán en el lienzo y en los componentes *Current State*, *String* y *Stack*, terminando con un mensaje que indica si el autómata acepta la cadena de entrada y el criterio usado.

Capítulo 5

Gramáticas

Hasta ahora la forma en que hemos descrito lenguajes ha sido a través de máquinas aceptadoras con un número finito de estados. Otra forma de describir lenguajes matemáticamente es por medio de gramáticas. Las gramáticas nos ofrecen una descripción generativa de un cierto lenguaje a través de un conjunto de reglas, las cuales aplicadas con un cierto orden generan palabras del lenguaje.
Formalmente, una gramática G es un cuádruplo

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

donde

N es un conjunto finito de símbolos *no terminales*,
T es un conjunto finito de símbolos *terminales*,
P es un conjunto de producciones, y
S es el símbolo inicial tal que $S \in N$

Cada producción en P es una pareja ordenada (α, β) con $\alpha = \gamma A \delta$ en la que β, γ y δ son posiblemente vacías en $(N \cup T)^*$ y $A \in N$ o bien $A = T$. Denotamos a la pareja (α, β) como $\alpha \longrightarrow \beta$

Restringiendo el tipo de producciones que permitimos en una gramática, tenemos una clasificación de gramáticas en cuatro tipos. Esta clasificación es conocida como *clasificación de Chomsky*.

5.1 Implementación de gramáticas

Dada la definición anterior de gramáticas decidimos implementar una gramática genérica, para después extenderla y agregar las restricciones necesarias definidas por la clasificación de Chomsky. Lo anterior lo definimos dentro de un paquete, que se muestra en la figura 5.1.

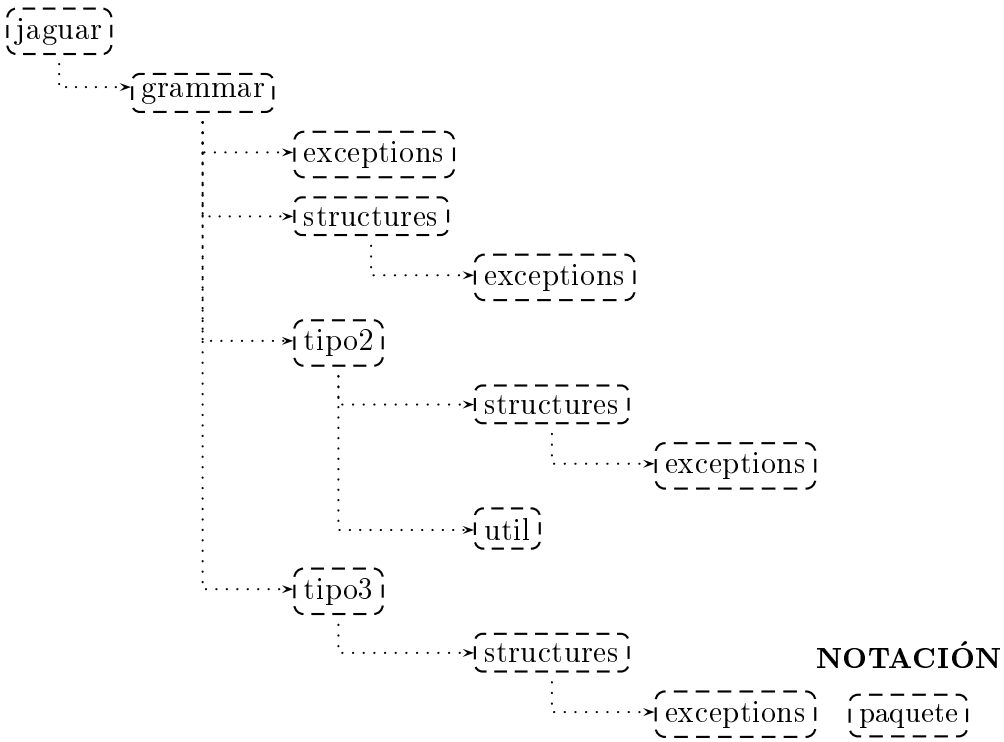

Figura 5.1: Paquetes para las gramáticas

Para definir la gramática más general tenemos las siguientes clases:

- jaguar.grammar.Grammar. Ésta es la clase abstracta, sin restricciones en la forma de las producciones. Tiene dos alfabetos N y T; un conjunto de producciones P y un símbolo inicial S; así como métodos de acceso, de escritura y lectura a partir de la definición de un archivo.
- jaguar.grammar.structures.Production. Ésta es una clase abstracta que define una producción de gramática, i.e. está compuesta de la cadena antecedente y la cadena consecuente.
- jaguar.grammar.structures.ProductionSet. Esta clase es un conjunto de producciones sin restricciones, es abstracta.

Con las tres clases antes mencionadas tenemos una gramática general, sin restricciones. Las tres clases anteriores definen un método *validador*, el cual debemos implementar para cada tipo de gramática. En el método antes mencionado es donde tenemos que hacer explícitas las restricciones de las producciones dado el tipo de gramática a representar.

5.2 Descripción de gramáticas

Como la única diferencia entre las gramáticas que revisaremos es la forma de sus producciones, basta con dar una descripción general de gramáticas. Los elementos usados son:

Antecedente. Usaremos las marcas `<left> </left>`; dentro de estas etiquetas debe haber una cadena como las definimos en la sección 2.2. Formalmente:

```
<!ELEMENT left (str)>
```

Consecuente. Como en el lado derecho podemos tener una cadena sobre $N \cup T$, dependiendo del tipo de gramática sólo definiremos que el consecuente es una cadena sobre el alfabeto dado, entre las marcas `<right> </right>`. Formalmente:

```
<!ELEMENT right (str)>
```

Producción. Es un par ordenado formado por un antecedente seguido de un consecuente entre las marcas `<p> </p>`. Formalmente:

```
<!ELEMENT p (left, right)>
```

Conjunto de Producciones. Es un conjunto de producciones como las descritas anteriormente entre las etiquetas `<productionSet> </productionSet>`. Formalmente:

```
<!ELEMENT productionSet (p)*>
```

Gramática. Es un 4-tuplo entre las etiquetas `<gram> </gram>` con los siguientes elementos en orden: una descripción de la gramática (opcional), un alfabeto de símbolos no terminales N; un alfabeto de símbolos terminales T; un conjunto de producciones como las acabamos de definir y, finalmente, un símbolo inicial $S \in N$. Formalmente:

```
<!ELEMENT gram (description?,alph,alph,productionSet,sym)>
```

Por ejemplo, la gramática G que describe $L(G) = \{w = a^n b^n | n > 0\}$ la describimos:

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <!DOCTYPE gram SYSTEM "grammar.dtd">
3
4   <gram>
5
6   <description>L(G) = \{w = a^{n}b^{n} | n>0\}</description>
7
8   <!-- El alfabeto de no terminales N -->
9    <alph> <sym>A</sym> <sym>S</sym> </alph>
10
11   <!-- El alfabeto de terminales T -->
12   <alph> <sym>a</sym> <sym>b</sym> </alph>
13
14   <!-- El conjunto de producciones P -->
15   <productionSet>
16    <p> <left> <str><sym>P</sym></str> </left>
17        <right> <str><sym>A</sym></str> </right>
18    </p>
19
20    <p> <left> <str><sym>A</sym></str> </left>
21        <right> <str><sym>a</sym>
```

```
22                      <sym>A</sym> <sym>b</sym>
23                  </str>
24              </right>
25          </p>
26
27          <p> <left> <str><sym>A</sym></str> </left>
28              <right> <str><sym>a</sym> <sym>b</sym></str> </right>
29          </p>
30        </productionSet>
31
32        <!-- El símbolo inicial S -->
33        <sym>S</sym>
34      </gram>
```

5.3 Generación de la interfaz gráfica

Desarrollamos una interfaz gráfica genérica para la visualización de las gramáticas en general. Esta interfaz cuenta con las funciones básicas y dependiendo de los temas que podamos desarrollar con cada uno de los tipos de gramáticas especializamos la interfaz. Así, la estructura gráfica está dada por:

- jaguar.grammar.jgrammar.JGrammarFrame. Éste es el marco para las gramáticas; sólo cuenta con un menú básico para cargar definiciones de gramáticas y un área gráfica donde se representa la gramática. Esta clase es abstracta y para extenderla es necesario implementar el método que inicializa la gramática. Esto nos lleva a que cada extensión que se haga tendrá que hacer una verificación implícita al inicializar un tipo de gramática, para cotejar que el tipo de la gramática es correcto.

- jaguar.grammar.jgrammar.JGrammarCanvas. Éste es el lienzo donde mostramos cada uno de los componentes de una gramática dada. Este lienzo es el mismo para cualquier gramática con interfaz gráfica.

5.4 Gramáticas tipo 3

Las producciones de las gramáticas tipo 3 o *regulares* tienen en el consecuente de las producciones a lo más dos símbolos, uno terminal y uno no terminal, o bien, un solo símbolo terminal; excepto por la posible producción $S \longrightarrow \epsilon$, la cual, si está presente en P, entonces S no puede aparecer del lado derecho de una producción.
Si la producción es de la forma

$$A \longrightarrow aB \quad o \quad A \longrightarrow a$$

con $A, B \in N \cup \{S\}$ y $a \in T$, se trata de una producción *lineal a la derecha.*
Si la producción es de la forma

$$A \longrightarrow Ba \quad o \quad A \longrightarrow a$$

se trata de una producción *lineal a la izquierda.*
Las gramáticas tipo 3 sólo pueden ser lineales a la izquierda ó a la derecha.

5.4.1 Implementación

Para implementar gramáticas tipo 3, o en general cualquier tipo, sólo basta con extender las clases abstractas definidas en la sección 5.1.

- jaguar.grammar.tipo3.Gtipo3. Extiende a Grammar implementando el método validador. Además, cuando la gramática es lineal derecha, puede usarse como motor para verificar si dada una cadena, ésta puede ser generada por la gramática.
- jaguar.grammar.tipo3.structures.ProductionT3. Esta clase extiende a Production implementando el método validador, el cual describe cómo debe ser cada una de las producciones, en este caso lineales derechas o izquierdas.
- jaguar.grammar.tipo3.structures.ProductionT3Set. Extiende a la clase ProductionSet implementando el validador de producciones.

5.4.2 Uso del motor

Es importante recalcar que este motor sólo podrá ser usado en gramáticas que sean lineales derechas.

Este motor se puede ejecutar desde la consola, invocando la clase Gtipo3 con dos parámetros: el primero, un archivo con la especificación de una gramática tipo 3 lineal derecha, como la descrita previamente; y el segundo un archivo con la especificación de una cadena. Por ejemplo:

```
java machine/grammar/tipo3/Gtipo3 test/gramatica.gt3 test/gacepta.str
```

La salida de ejecutar el comando anterior se muestra en la figura 5.2, donde aparece la definición de la gramática, la cadena que tiene que generar, y el resultado de intentar generar la cadena dada. En caso de que el veredicto sea que sí la pudo generar, mostrará el orden en que se deben de aplicar las producciones para generar dicha cadena.

```
        Leyendo la gramática de: test/gramatica.gt3
        Leyendo la cadena de: test/gacepta.str

La gramática: G

        N = <alph><sym>S</sym><sym>B</sym></alph>
        T = <alph><sym>1</sym><sym>0</sym></alph>
        S = S
        P = [B --> 1S, S --> 0B, B --> 0, B --> 0B]


        Cadena a generar: 0010000
SI genera la cadena 0010000
La secuencia de producciones es:
[S --> 0B, B --> 0B, B --> 1S, S --> 0B, B --> 0B, B --> 0B, B --> 0]
```

Figura 5.2: Ejecución del motor Gtipo3

5.4.3 Generación de la interfaz gráfica

Para desarrollar la interfaz gráfica sólo hace falta extender a la clase JGrammarFrame. Con este tipo de gramática podemos, dada la descripción de una cadena,validar su pertenencia al lenguaje generado por la gramática. Para esto vemos si a partir de S podemos generar a la cadena. Esto nos lleva a pensar que la extensión de la clase JGrammarFrame debe soportar definiciones de cadenas para ver si las podemos generar. Las clases definidas son:

- jaguar.grammar.jgrammar.JGrammarFrameT3. Agrega el soporte para construir y cargar definiciones de cadenas y verificar si pertenecen al lenguaje generado por una gramática dada.

- jaguar.grammar.jgrammar.JDerivationTreeT3. Una vez que el motor de la gramática tipo 3 asociada al marco determina que una cadena dada es generada por la gramática, esta clase se encarga de generar gráficamente el árbol de derivación de la cadena.

5.4.4 Uso de la interfaz gráfica

Para usar la interfaz gráfica basta con seguir la siguiente secuencia en el centro de control. Al elegir [grammar] → [Type 3 Grammar] aparecerá un marco muy sencillo con un solo componente G. Además, veremos los ya conocidos menús para cargar definiciones de gramáticas (análogo a cargar máquinas, ver sección 2.1.2), así como los relacionados a la construcción y carga de cadenas (ver secciones 2.1.3 y 2.1.4).

En este marco podemos cargar la definición de una gramática, obteniendo un marco como el que se muestra en la figura 5.3

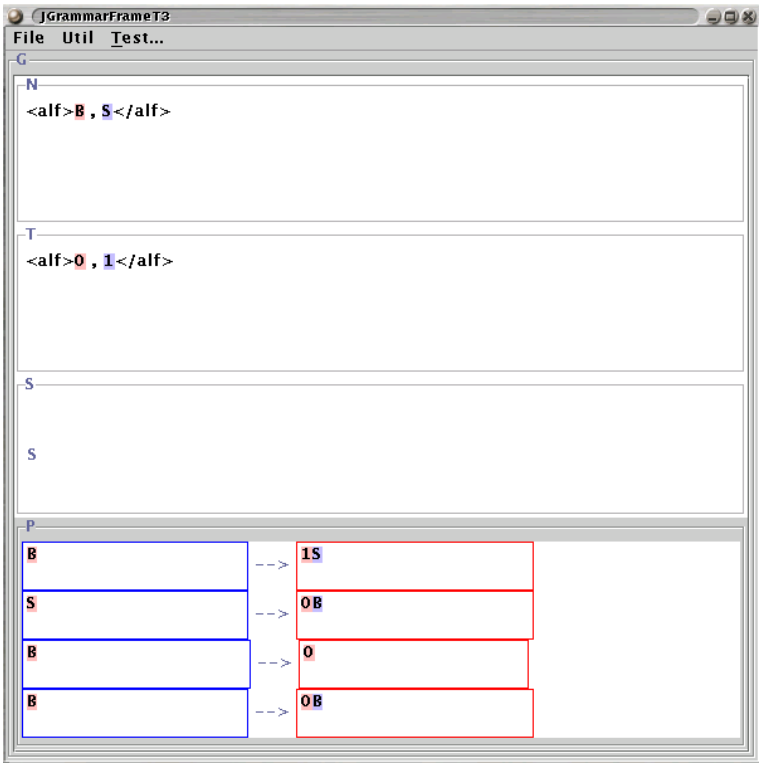

Figura 5.3: Marco para gramáticas tipo 3

Si la definición de la gramática cargada es lineal derecha, como la de la figura 5.3, se habilitarán los menús para cadenas. A partir de ahora puede suceder una de las dos siguiente acciones:

- La cadena es aceptada, en cuyo caso aparecerá un mensaje informándonos que la cadena sí es generada por la gramática – figura 5.4. En cuanto presionemos (ok), podremos observar en una ventana adicional el árbol de derivación que se usó para generar la cadena – figura 5.5.

Figura 5.4: Mensaje de generación exitosa

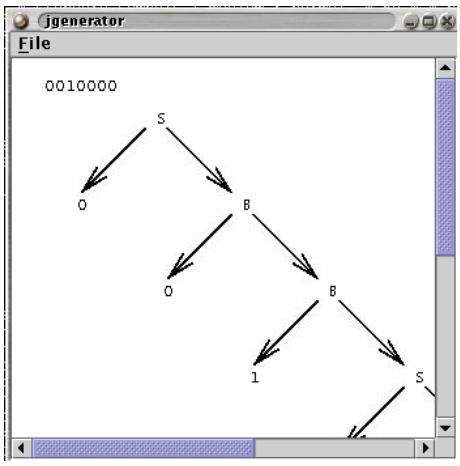

Figura 5.5: Árbol de derivación de una cadena generada

- La cadena no es aceptada; en este caso un mensaje nos lo hará saber como se muestra en la figura 5.6

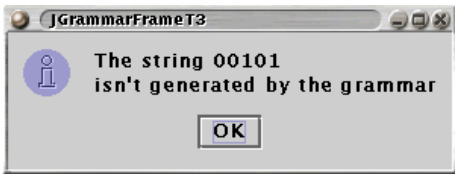

Figura 5.6: Mensaje de generación no exitosa

5.5 Conversión de gramáticas tipo 3 a AF

Podemos convertir gramáticas tipo 3 a autómatas Finitos. En general, el resultado de esta conversión resulta en un autómata finito no determinístico.

5.5.1 Motor de conversión

Dada una gramática tipo 3 $G = (N, T, P, S)$, el motor de conversión definido en la clase jaguar.grammar.tipo3.Gtipo32AF, genera un autómata finito $M = (Q, \Sigma, \delta, q_0, F)$ equivalente, construyendo cada uno de los componentes de M de la siguiente manera:

Σ es la T pues vamos a manejar el mismo lenguaje.

Q Tendrá un estado por cada símbolo en $N \cup \{S\}$, más un nuevo estado Z, tal que el símbolo $Z \notin N \cup \{S\}$ y que fungirá como estado final

$$Q = \{A \in N\} \cup \{S\} \cup \{Z\}$$

q_0 estará representado por el símbolo inicial de G.

$$q_0 = S$$

δ la definimos de la siguiente manera:

- Para las producciones de la forma $A \longrightarrow aB$ definimos $\delta(A, a) \supset B$ entre las transiciones de M.
- Para las producciones de la forma $A \longrightarrow a$ definimos $\delta(A, a) \supset Z$
- Además, para el nuevo símbolo Z, se agregan, para cada símbolo $a \in V$ las siguientes transiciones: $\delta(Z, a) = \varphi$
- φ es un estado pozo, que no acepta, y que tiene sus transiciones definidas de la siguiente manera: $\delta(\varphi, a) = \varphi, \quad \forall a \in \Sigma$
- $$F = \begin{cases} \{q_0, Z\} & \text{Si} \quad S \longrightarrow \varepsilon \in P \\ \{Z\} & \text{Si} \quad S \longrightarrow \varepsilon \notin P \end{cases}$$

La implementación de está construcción, es sencilla pues todos los elementos que conforma a un autómata finito no determinístico ya han sido definidos. A continuación revisaremos la manera de usar el motor.

5.5.2 Uso del motor

```
        Leyendo la gramática de: test/gramm/gramatica-pag86.gt3

Gramática de entrada:
G

        N = <alph><sym>S</sym><sym>B</sym></alph>
        T = <alph><sym>1</sym><sym>0</sym></alph>
        S = S
        P = [B --> 1S, S --> 0B, B --> 0, B --> 0B]

Gtipo32Af.doConvertion
Construyendo un AFN equivalente

     Sigma = <alph><sym>1</sym><sym>0</sym></alph>

     Q =[S, B]

     Nuevo estado final Z =Zf_0

     Q0 =[S]

     Construyendo la delta d =

     Generando estado nuevo pozo = Pozo_0

     El AF equivalente es:

     NDFA N =

        Q=[Zf_0, S, Pozo_0, B]
        Sigma = <alph><sym>1</sym><sym>0</sym></alph>
        delta =
                 Zf_0 <sym>1</sym>=[Pozo_0], <sym>0</sym>=[Pozo_0]
                 S 0=[B]
                 Pozo_0 <sym>1</sym>=[Pozo_0], <sym>0</sym>=[Pozo_0]
                 B 1=[S], 0=[Zf_0, B]
        Q0 = [S]
        F= [Zf_0]
```

Figura 5.7: Ejecución del motor del convertidor Gramática tipo 3 a AF

El motor convertidor puede ejecutarse desde la consola, invocando a la clase jaguar.grammar.tipo3.Gtipo32AF con un parámetro: el archivo donde está definida la gramática tipo 3 de la cual queremos obtener el AFN equivalente. Por ejemplo, si en

el archivo `test/gram/gfoo.gt3` tenemos la gramática a convertir, tendríamos:

```
java jaguar.grammar.tipo3.Gtipo32AF test/gramm/gfoo.gt3
```

La salida del comando anterior se muestra en la figura 5.7.

5.5.3 Interfaz gráfica del convertidor de gramática tipo 3 a AF

Ya que tenemos el motor implementado, sólo hace falta hacer una especialización de éste, la cual debe implementar la interfaz JConverter y extender el marco de la representación gráfica de los convertidores genéricos, el cual deberá saber desplegar gramáticas y autómatas finitos. Todo esto lo hacemos en las clases:

- jaguar.grammar.tipo3.jtipo3.JGtipo32AF. Es la extensión gráfica del motor e implementa la interfaz JConverter.
- jaguar.grammar.tipo3.jtipo3.JGtipo32AFFrame. Extiende a JConverterFrame, usando las etiquetas adecuadas para los menús. Nos ofrece los botones necesarios para el uso del convertidor.

5.5.4 Uso de la interfaz gráfica

El uso de la interfaz gráfica es análoga a la descrita en la sección 2.4.

5.6 Gramáticas tipo 2

Las producciones de las gramáticas tipo 2 o *libres de contexto* son de la forma:

$$A \longrightarrow \alpha$$

con $A \in N \cup \{S\}$, $\alpha \in (N \cup T)^+$, excepto por la posible producción $S \longrightarrow \epsilon$.

5.6.1 Implementación

Para implementar gramáticas tipo 2, o en general cualquier tipo, sólo basta con extender las clases abstractas definidas en la sección 5.1.

- jaguar.grammar.tipo2.Gtipo2. Extiende a Grammar implementando el método validador.
- jaguar.grammar.tipo2.structures.ProductionT2. Esta clase extiende a Production implementando el método validador, el cual describe cómo debe ser cada una de las producciones. En este caso deben tener un único símbolo no terminal de lado izquierdo.
- jaguar.grammar.tipo2.structures.ProductionT2Set. Extiende a la clase ProductionSet implementando el validador de producciones.

5.6.2 Gramáticas tipo 2 *flojas*

Definimos gramáticas tipo 2 que son *flojas* en el sentido de cómo es que validan. Éstas fueron definidas para poder implementar los métodos que simplifican gramáticas tipo 2, aceptando algunas violaciones a la forma de las producciones como las producciones-ε, producciones unitarias, etc.. La forma en que se implementan es redefiniendo su método de validación. La clase donde se encuentra esta extensión es:

- jaguar.grammar.tipo2.Gtipo2Lazy. Extiende a Gtipo2 redefiniendo el método validador.

5.7 Simplificación de gramáticas libres de contexto

El objetivo que perseguimos con la simplificación de gramáticas es la de llevar a una gramática a que cumpla con la especificación de gramática de tipo 2 que dimos anteriomente y, más aún, que las producciones tomen una forma canónica. Esto para facilitar el manejo de las gramáticas libres de contexto y para tener más herramientas para demostrar propiedades de las mismas. La simplificación sigue los procesos descritos en las siguientes secciones.

5.7.1 Eliminar símbolos muertos

Los símbolos muertos son símbolos que no generan cadenas terminales. Para esto tenemos que construir una gramática con un alfabeto de símbolos no-terminales a partir de los cuales siempre podamos generar palabras terminales. Llamaremos a tal conjunto N'. Una vez que tenemos este conjunto podemos crear un conjunto de producciones P' cuyos símbolos están en $N' \cup T$. El Algoritmo 1 encuentra N'.

Algoritmo 1 Construye el conjunto N' para eliminar símbolos muertos

1: $VIEJO_N := \{\}$;
2: $NUEVO_N := \{A \mid A \longrightarrow w$ para alguna $w \in T^*\}$;
3: **while** ($VIEJO_N \neq NUEVO_N$) **do**
4: $\quad VIEJO_N := NUEVO_N$;
5: $\quad NUEVO_N := VIEJO_N \cup \{A \mid A \longrightarrow \alpha$ para alguna $\alpha \in (T \cup VIEJO_N)^*\}$
6: **end while**
7: $N' := NUEVO_N$

5.7.2 Eliminar símbolos inalcanzables

Los símbolos inalcanzables son símbolos a los que no podemos llegar desde el símbolo inicial. Los eliminamos generando dos nuevos conjuntos, el de símbolos no-teminales N' y el de terminales T'. Para construir estos conjuntos lo que tenemos que hacer es primero poner en N' a S. Ahora, si $A \in N'$ y tenemos en P las producciones $A \longrightarrow \alpha_1, A \longrightarrow \alpha_2 \ldots A \longrightarrow \alpha_k$, todos los símbolos terminales que estén en α_j, $1 \leq j \leq k$ se agregan a T', y todas las no-terminales que se encuentren en las α_j son agregadas a N'. Al finalizar, P'

es el conjunto de producciones de P que contienen únicamente símbolos de $N' \cup T'$. El algoritmo 2 corresponde al proceso anterior.

Algoritmo 2 Construye N' y T' para eliminar símbolos inalcanzables

1: $VIEJO_N := \{\}$;
2: $NUEVO_N := \{S\}$;
3: $VIEJO_T := \{\}$;
4: $NUEVO_T := \{\}$;
5: **while** ($VIEJO_N \neq NUEVO_N \vee VIEJO_T \neq NUEVO_T$) **do**
6: $VIEJO_N := NUEVO_N$;
7: $VIEJO_T := NUEVO_T$;
8: $NUEVO_N := VIEJO_N \cup \{\alpha_i \in N \mid A \longrightarrow \alpha_1 \cdots \alpha_k$ para alguna $1 \leq i \leq k$, $A \in VIEJO_N\}$
9: $NUEVO_T := VIEJO_T \cup \{\alpha_i \in T \mid A \longrightarrow \alpha_1 \cdots \alpha_k$ para alguna $1 \leq i \leq k$, $A \in VIEJO_T\}$
10: **end while**
11: $N' := NUEVO_N$
12: $T' := NUEVO_T$

5.7.3 Eliminar producciones-ε

Las producciones-ε son las que tienen la forma $A \longrightarrow \varepsilon$, con $A \in N$. Para eliminarlas, tenemos que identificar los símbolos nulificables[1]. Una vez identificados, reemplazamos cada producción de la forma $B \longrightarrow X_0 X_1 \ldots X_k$ por todas las producciones que resultan de eliminar sucesivamente cada uno de los subconjuntos de símbolos nulificables. Debemos tener cuidado de no eliminar todo el lado derecho de la producción, cosa que puede pasar si todos los símbolos son nulificables.

Para determinar el conjunto de símbolos nulificables usamos el algoritmos 3.

Algoritmo 3 Determina los símbolos nulificables

1: $Nulif_V := \varnothing$
2: $Nulif_N := \{A \in N \mid A \longrightarrow \varepsilon \in P\}$
3: **while** ($Nulif_V \neq Nulif_N$) **do**
4: $Nulif_V := Nulif_N$
5: $Nulif_N := Nulif_N \cup \{B \in N \mid B \longrightarrow \alpha \in P, \alpha = X_1 X_2, \ldots, X_k,\ k \geq 1,$ $X_i \in N \cup \{\varepsilon\}, X_i = \varepsilon$ o bien $X_i \in Nulif_V\}$
6: **end while**

5.7.4 Eliminar producciones unitarias

Las producciones unitarias son aquellas que tienen la forma $A \longrightarrow B$, con $A, B \in N$. Para eliminar este tipo de producciones de la gramática $G = (N, T, P, S)$ –de la cual podemos asumir que no contiene producciones-ε, pues las hemos eliminado en el proceso

[1] Un símbolo A es nulificable si $A \stackrel{*}{\Longrightarrow} \varepsilon$.

anterior– construimos un nuevo conjunto de producciones P' a partir de P, incluyendo en primera instancia todas las producciones de la forma $A \longrightarrow \alpha$, donde $A \longrightarrow \alpha$ es una producción no unitaria de P.

Ahora, detectamos $A \overset{*}{\underset{G}{\Longrightarrow}} B$ construyendo un autómata finito con un estado por cada símbolo no terminal que aparece en una producción unitaria. A continuación marcamos una transición-ε entre dos estados (símbolos no terminales) A y B si la producción unitaria $A \longrightarrow B \in P$; y transiciones etiquetadas con α a un estado final φ desde el estado A si $\mid \alpha \mid > 1$ y la producción $A \longrightarrow \alpha \in P$.

Con el autómata finito construido, para cada estado distinto del final, determinamos la ε-cerradura de ese estado y procedemos a sustituir las producciones unitarias de ese símbolo no terminal con producciones cuyo lado izquierdo es el símbolo del estado y el lado derecho es la etiqueta de cada una de las transiciones desde cualquiera de los estados en su ε-cerradura al estado final φ. Esto lo hacemos, por supuesto, para todos los estados del autómata distintos de φ.

5.7.5 Forma normal de Chomsky

Una gramática libre de contexto está en **Forma Normal de Chomsky** (FNC) si todas su producciones son de alguna de las siguientes formas:

1. $A \longrightarrow BC$ con A, B y $C \in N$, o bien
2. $A \longrightarrow a$ con $a \in T$, o bien
3. Si $\varepsilon \in L(G)$, entonces $S \longrightarrow \varepsilon$ es una producción, y S no aparece del lado derecho de ninguna producción.

Para lograr que las producciones sean de esta forma primero obtendremos una gramática $G' = (N', T, P', S)$, donde las producciones son de alguna de las siguientes formas:

- $A \longrightarrow a$
- $A \longrightarrow X_1 X_2 \ldots X_n$, con $n \geq 2$, $X_i \in N'$

tal que $L(G) = L(G')$. El algoritmo 4 corresponde a esta primera parte, donde $G = (N, T, P, S)$ es una GLC sin producciones unitarias y construimos $G' = (N', T, P', S)$ con las características antes mencionadas.

Algoritmo 4 Primera parte para convertir a FNC

1: Agrega a P' todas las producciones de P de la forma $A \longrightarrow a$, o bien que se encuentren ya en la forma correcta $A \longrightarrow B_1 \ldots B_k$, $B_i \in N$, $1 \leq i \leq k$.

2: Sea $A \longrightarrow X_1 X_2 \ldots X_n$ una producción en P tal que algunas de las X_i's son símbolos terminales. En lugar de esta producción, ponemos en P' las producciones siguientes:

$$\begin{array}{ll} A \longrightarrow Y_1 Y_2 \ldots Y_n & \text{donde } Y_i = X_i \text{ si } X_i \in \text{N} \\ \text{o bien} & Y_i \text{ es un símbolo nuevo en } N' \text{ si } X_i \in T \end{array}$$

3: Por cada Y_i nueva introducida en lugar de alguna $X_i \in T$ en alguna producción, se introduce la producción $Y_i \longrightarrow X_i$.

Ahora, tenemos que convertir las producciones de P' a FNC. Esto lo logramos usando el algoritmo 5.

Algoritmo 5 Segunda parte para convertir a FNC

1: Agrega a P'' todas las producciones de la forma $A \longrightarrow a$ o bien $A \longrightarrow BC$ (que ya están en FNC)

2: Sustituye cada producción de la forma $A \longrightarrow X_1X_2 \ldots X_n$ con $n > 2$, por las producciones de la forma:

$$\begin{aligned} A &\longrightarrow X_1C_2 \\ C_2 &\longrightarrow X_2C_3 \\ &\vdots \\ C_{n-1} &\longrightarrow X_{n-1}X_n \end{aligned}$$

donde cada C_i es un símbolo no-terminal nuevo.

5.7.6 Forma normal de Greibach

Una gramática libre de contexto está en **Forma Normal de Greibach** (FNG) si todas sus producciones son de la forma $A \longrightarrow a\alpha$, con $A \in N$, $a \in T$ y $\alpha \in N^*$.

Para lograr que las producciones sean de la forma anterior, usaremos el algoritmo 6, el cual recibe una GLC en FNC con $N = \{A_1, \ldots, A_n\}$ (donde todos los elementos de N están ordenados y numerados) y cuya salida es una gramática en FNG.

5.7.7 Implementación del motor

Los cuatro procesos de simplificación de gramáticas son implementados en la clase:

- jaguar.grammar.tipo2.util.Simplifier. Esta clase implementa los procesos de eliminación de símbolos muertos, eliminación de símbolos inalcanzables, eliminación de producciones-ε, producciones unitarias, conversión a Forma Normal de Chomksy y de Greibach, de forma independiente. Estos procesos reciben una gramática y regresan otra con la característica deseada, siempre asumiendo que la gramática de entrada cumple con las restricciones pedidas por el algoritmo correspondiente.

5.7.8 Generación de la interfaz gráfica

Ya que contamos con la infraestructuras necesaria para la representación gráfica de gramáticas, sólo necesitamos desarrollar un mecanismo para aplicar los algoritmos de simplificación. Esto lo logramos definiendo:

- jaguar.grammar.tipo2.jtipo2.JSimplifier. Esta clase extiende a Simplifier e implementa a JConverter. Prepara los datos para desplegar gráficamente los resultados.

- jaguar.grammar.tipo2.jtipo2.JSimplifierFrame. Esta clase, dada su naturaleza de convertidor, extiende a jaguar.util.jutil.JConverterFrame y nos proporciona características

Algoritmo 6 Forma Normal de Greibach

PRIMERA PARTE:

Llevar a todas las producciones a que empiecen con un símbolo terminal o con un no-terminal con numeración mayor que la propia

1: **for** $k := 1, \ldots, n$ /**a todas las no-terminales*\/ **do**
2: **for** cada producción de la forma $A_k \longrightarrow A_j\alpha$, $j < k$ y para todas las producciones $A_j \longrightarrow \beta$ **do**
3: agrega producción $A_k \longrightarrow \beta\alpha$
4: quita la producción $A_k \longrightarrow A_j\alpha$
5: **end for**/* *aplicar lema de fusión de producciones* */
6: **for** cada producción $A_k \longrightarrow A_k\alpha$ **do**
7: agrega las producciones $B_k \longrightarrow \alpha$, $B_k \longrightarrow \alpha B_k$;
8: quita la producción $A_k \longrightarrow A_k\alpha$
9: **end for**
10: **for** cada producción $A_k \longrightarrow \beta$ tal que β no empieza con A_k **do**
11: agrega la producción $A_k \longrightarrow \beta B_k$
12: **end for**/* *Eliminar recursividad izquierda* */
13: **end for**/* *de la primera parte* */

SEGUNDA PARTE:

Usando fusión de producciones, hacer que cada producción tenga la FNG.

14: **for** $k := n-1, \ldots, 1$ **do**
15: **for** cada producción de la forma $A_k \longrightarrow A_r\alpha$ con $r > k$ **do**
16: agrega las producciones de la forma $A_k \longrightarrow \gamma\alpha$,
donde $A_r \longrightarrow \gamma$ es una producción.
17: quita la producción $A_k \longrightarrow A_r\alpha$
18: **end for**
19: **end for**/* *Empezar cada una con terminal* */

muy parecidas a los convertidores gráficos que se describen en el capítulo 2.3. Cuentan con un menú; un panel para marcos, donde se desplegarán las gramáticas; y un área de detalles, donde se podrá seguir a detalle los pasos de cada uno de los procesos de simplificación y normalización aplicados.

Implementa una utilería para seleccionar el tipo de simplificación o normalización que se quiere realizar.

El uso de este convertidor es similar al descrito en la sección 2.4. Sólo hay un detalle que resaltar en su funcionamiento y tiene que ver con el orden de las simplificaciones. Veamos un ejemplo. Una vez cargada una gramática a simplificar, vamos a la opción (File) → (Standardize...) ó presionamos la tecla [F3]; aparecerá un marco con la advertencia: "*Los algoritmos serán hechos en el orden en que aparecen listados (de arriba a abajo). Si alguna normalización require de una previa, esta será hecha*"; una vez que presionamos (ok) al mensaje, aparecen las opciones de normalización. En el ejemplo de la figura 5.8 seleccionamos la FNC.

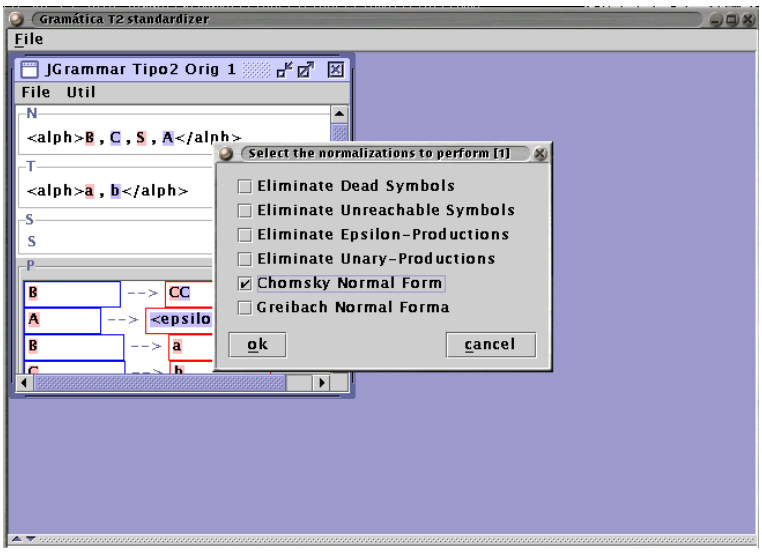

Figura 5.8: Seleccionando FNC

Una vez que presionamos (ok), se realizarán las normalizaciones y aparecerán en marcos en el orden en que fueron hechas, de arriba hacia abajo. Cada uno puede guardarse de manera individual, como se muestra en la figura 5.9.

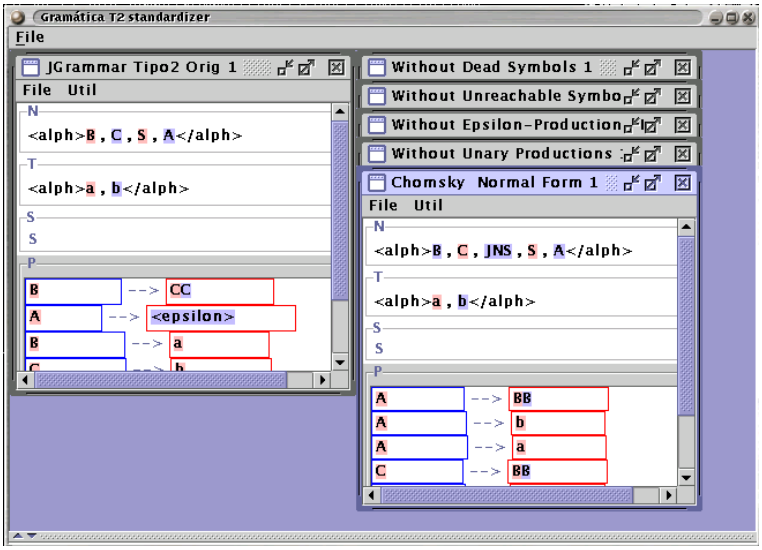

Figura 5.9: Resultado de la normalización

Finalmente, podemos reorganizar los resultados y verlos con distintas organizaciones – figura 5.10. En los ejemplos el área de detalles, donde se siguen los algoritmos, fue minimizada.

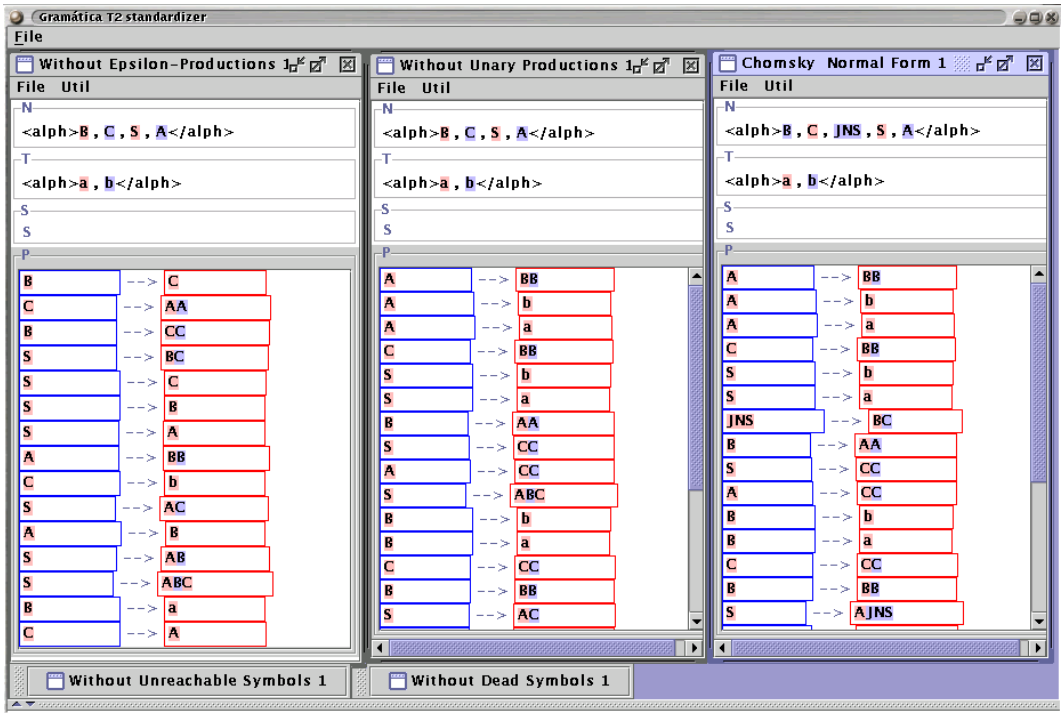

Figura 5.10: Organizando los resultados

Habiendo ya descrito tanto a los aceptadores como a los generadores de lenguajes, pasamos a describir la parte de la aplicación que nos va a permitir pasar de uno a otro.

Capítulo 6

Convertidores

Un convertidor es una aplicación que permite pasar de un modelo de máquina a otro equivalente, y de un modelo de máquina a la gramática que genera el lenguaje aceptado por la máquina. En este capítulo describiremos, la implementación y ejecución de los convertidores contemplados en esta aplicación, de manera similar a como se hizo en la sección 2.4.

6.1 Conversión NFA a DFA

Para convertir un NFA a DFA se utiliza el Algoritmo 7 que se encuentra implementado en la clase jaguar.machine.dfa.converters.Ndfa2Dfa. Toma como argumento el nombre de un archivo donde está la especificación de un NFA y, opcionalmente, el nombre del archivo donde guardará la especificación del DFA equivalente.

6.1.1 Uso del convertidor

El convertidor puede ejecutarse desde la consola, invocando a la clase `Ndfa2Dfa` con al menos un parámetro: el primero, un archivo con una especificación de NFA como la descrita previamente y, opcionalmente, un segundo archivo donde se guardará la especificación del DFA equivalente. Por ejemplo, si tenemos la especificación del NFA que se muestra en la Tabla 3.1 en el archivo `tabla1.ndfa`, una forma de ejecutar el programa sería:

```
java jaguar.machine.dfa.converters.Ndfa2Dfa tabla1.ndfa
```

```
NDFA =>
 M =
        Q=[A, C, B]
        Sigma = <alph><sym>1</sym><sym>0</sym></alph>
        delta =
                 A 1=[C], 0=[A, B]
                 C 1=[A, C]
                 B 1=[C], 0=[B]
        Q0 = [A, B]
        F= [C, B]

DFA =>
  M =
        Q=[q1, q0, q3, q2]
        Sigma = <alph><sym>1</sym><sym>0</sym></alph>
        delta =
                 q1 <sym>1</sym>=[q3], <sym>0</sym>=[q2]
                 q0 <sym>1</sym>=[q1], <sym>0</sym>=[q0]
                 q3 <sym>1</sym>=[q3], <sym>0</sym>=[q0]
                 q2 <sym>1</sym>=[q2], <sym>0</sym>=[q2]
        q0 = q0
        F= []
```

Figura 6.1: Ejecución del motor del convertidor sobre el NFA de la tabla 3.1

La figura 6.1 nos muestra al principio el NFA de entrada y a continuación el DFA equivalente.

Algoritmo 7 Convierte un NFA M a un DFA M'

1: El primer renglón de la tabla de transiciones de M' es el subconjunto formado por los estados iniciales de M.
2: **while** (haya renglones sin todas sus transiciones computadas) **do**
3: Sea S el símbolo que denota a un subconjunto de Q que ha sido incluido como renglón en la tabla, pero que sus transiciones no han sido resueltas. Calcula, para cada símbolo a en Σ, la columna correspondiente, formando un nuevo subconjunto con todos aquellos estados a los que posiblemente se transfiere M bajo a.

$$\delta(S, a) = \{t \mid t \in \delta(s, a) \text{ para } s \in S\}$$

4: **if** (el subconjunto así obtenido no aparece ya como renglón en la tabla) **then**
5: Inclúyelo en la tabla.
6: **end if**
7: **end while**
8: Marca un estado S en Q' como estado final si contiene a algún estado q de Q tal que q es final en M. Si no contiene a ningún estado de F, entonces S no es estado final.

6.1.2 Interfaz gráfica del convertidor

Una vez que contamos con el motor es sencillo generar la interfaz gráfica, como se describió en la sección 2.3. Lo que hacemos es extender la clase Ndfa2Dfa agregando un par de características que describimos a continuación. Las clases para la extensión gráfica son:

- jaguar.machine.dfa.jdfa.jconverters.JNdfa2Dfa. Esta es la extensión antes mencionada del motor. Con esta extensión tenemos la parte algorítmica recibiendo como parámetro un NFA (en realidad le pasamos un JNDFA pues cargamos una descripción desde la interfaz gráfica). Ahora, el resultado del método de conversión de la super clase nos regresa un DFA, pero necesitamos un JDFA para poder mostrarlo gráficamente. Así que redefinimos el método de conversión haciendo una llamada al método de la superclase y haciendo la conversión de todas las estructuras del DFA que nos regresaron, a estructuras de extensión gráfica. Esta clase también implementa la interfaz JConverter para poder usar los extensiones gráficas genéricas de los convertidores.

- jaguar.machine.dfa.jdfa.jconverters.JNdfa2DfaFrame. Extiende a la clase JConverterFrame. Le pasamos las etiquetas adecuadas para el componente *Menú*. Este componente nos ofrece un menú con tres botones: (Load NDFA to Convert...), (Do Convertion) y (Quit). El resultado se muestra en la figura 6.2.

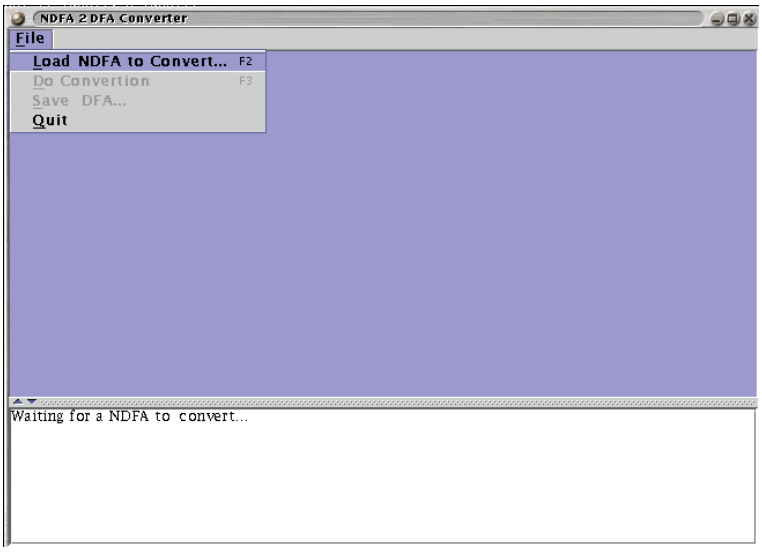

Figura 6.2: Interfaz para el convertidor de NFA a DFA

6.2 Minimización de autómatas finitos

Los problemas asociados a construir y minimizar autómatas finitos surgen de manera constante en problemas de diseño de aplicaciones de software y hardware. Dado que el

costo de implementar un autómata es proporcional al número de estados, al minimizar el tamaño de la máquina en términos del número de estados, se minimiza su costo.

El proceso de minimización consiste en eliminar del autómata los estados inalcanzables e identificar en uno solo a los estados equivalentes . Todo esto lo lleva a cabo el motor de minimización jaguar.machine.dfa.converters.Minimizer.

6.2.1 Interfaz gráfica del minimizador

Ahora, para generar la interfaz gráfica tenemos que extender nuestro motor e implementar algunas interfaces; esto se describe a continuación:

- jaguar.machine.dfa.jdfa.jconverters.JMinimizer. Esta clase extiende a machine.dfa.Minimizer e implementa a la interfaz JConverter. Recibe un JDFA a minimizar, el cual será cargado con los menús.
- jaguar.machine.dfa.jdfa.jconverters.JMinimizerFrame. Extiende a la clase JConverterFrame usando las etiquetas adecuadas, las cuales hacen referencia a cargar un DFA a minimizar, comenzar la minimización y guardar el DFA mínimo resultante.

El uso de este convertidor es equivalente al descrito en la sección 2.4. Esta interfaz nos mostrará el JDFA original y el equivalente minimizado.

6.3 DFA a gramática tipo 3

Finalmente, sabemos de la relación que existe entre los autómatas finitos y las Gramáticas Tipo 3 (ver sección 5.4). Dada esta equivalencia implementamos un convertidor que recibe un DFA y genera una Gramática Tipo 3, Gtipo3. Ambos elementos son desplegados en la interfaz gráfica, cada uno en un marco interno. Para esto primero implementamos el motor de conversión.

6.3.1 Motor de conversión

Dado $M = (Q, \Sigma, \delta, q_0, F)$ un autómata finito determinístico, el motor de conversión, definido en la clase jaguar.machine.dfa.converters.Dfa2Gtipo3, sigue el siguiente proceso. Definimos una gramática tipo 3, $G = (Q, \Sigma, P, q_0)$ de la siguiente manera:

1. $B \longrightarrow aC \in$P si $\delta(B, a) = C$
2. $B \longrightarrow a \in$P si $\delta(B, a) = C$ y $C \in F$

Ahora, si $\varepsilon \in T(M)$ basta agregar la producción $S \longrightarrow \varepsilon$ a P.

El convertidor puede ejecutarse desde la consola, invocando a la clase jaguar.machine.dfa.converters.Dfa2Gtipo3 con un parámetro: el archivo donde está definido el DFA del cual queremos obtener la gramática tipo 3 equivalente. Por ejemplo, si en el archivo `dfa-001.dfa` tenemos el DFA a convertir, tendríamos:

```
java jaguar.machine.dfa.converters.Dfa2Gtipo3 test/dfa/dfa-001.dfa
```

La salida del comando anterior se muestra en la figura 6.3.

```
El DFA de entrada es DFA  M =
        Q=[A, D, C, B]
        Sigma = <alph><sym>1</sym><sym>0</sym></alph>
        delta =
                 A 1=A, 0=B
                 D 1=D, 0=D
                 C 1=D, 0=C
                 B 1=A, 0=C
        q0 = A
        F= [D]

Dfa2Gtipo3.doConvertion

El alfabeto N <alph>ADCB</alph>
El alfabeto T <alph><sym>1</sym><sym>0</sym></alph>
S = A
Generando el conjunto de producciones P

P [<str>D</str> --> 0D, <str>D</str> --> <str>1</str>,
   <str>D</str> --> <str>0</str>, <str>A</str> --> 1A,
   <str>B</str> --> 1A, <str>C</str> --> <str>1</str>,
   <str>C</str> --> 1D, <str>A</str> --> 0B,
   <str>B</str> --> 0C, <str>D</str> --> 1D,
   <str>C</str> --> 0C]

La Gtipo3 resultante es G

        N = <alph>ADCB</alph>
        T = <alph><sym>1</sym><sym>0</sym></alph>
        S = A
        P = [<str>D</str> --> 0D, <str>D</str> --> <str>1</str>,
             <str>D</str> --> <str>0</str>, <str>A</str> --> 1A,
             <str>B</str> --> 1A, <str>C</str> --> <str>1</str>,
             <str>C</str> --> 1D, <str>A</str> --> 0B,
             <str>B</str> --> 0C, <str>D</str> --> 1D, <str>C</str> --> 0C]
```

Figura 6.3: Ejecución del motor del convertidor DFA a gramática tipo 3

6.3.2 Interfaz gráfica del convertidor

Una vez más, como ya tenemos el motor, sólo hace falta implementar la interfaz JConverter y extender el marco que contendrá la representación gráfica del DFA y la de la gramática resultante. Lo anterior lo tenemos en las clases:

- jaguar.machine.dfa.jdfa.jconverters.JDfa2Gtipo3. Es la extensión gráfica del motor e implementa la interfaz JConverter.
- machine.jdfa.jconverters.JDfa2Gtipo3. Extiende a JConverterFrame, usando las etiquetas adecuadas para los menús. Nos ofrece los botones necesarios para el uso del convertidor.

El uso de la interfaz gráfica es análoga a la descrita en la sección 2.4.

Capítulo 7

Máquinas de Turing

En los capítulos anteriores observamos que los lenguajes que reconocen los autómatas finitos están contenidos en los lenguajes que reconocen los autómatas de stack, por lo cual decimos que los autómatas de stack son más poderosos que los primeros. Pero ¿dónde está la diferencia? La diferencia está en el dispositivo de memoria temporal dada por el stack. De lo anterior nos surge la pregunta: si contamos con un autómata con otro tipo de dispositivo ¿podremos reconocer una variedad más grande de lenguajes? La respuesta es afirmativa, y esto lo alcanzamos con la Máquina de Turing.

Aún cuando las Máquinas de Turing son sencillas como modelo, son las que poseen mayor capacidad en relación con la complejidad de lenguajes que pueden reconocer.

Una Máquina de Turing (MT) es un autómata cuyo dispositivo de almacenamiento temporal es una cinta. Esta cinta está dividida en celdas, cada celda puede almacenar sólo un símbolo. Asociada a la cinta está una cabeza de lectura-escritura que puede moverse en ambos sentidos, derecha e izquierda, y puede leer y escribir un símbolo en cada movimiento. Podemos pensar que la cinta es infinita únicamente hacia la derecha, y tiene una celda que podemos considerar la del extremo izquierdo. Acotar la cinta por la izquierda no limita en forma alguna la capacidad de cómputo de la Máquina de Turing. El poderío adicional de la Máquina de Turing se debe a la característica de poder escribir en la misma cinta de donde lee los datos de entrada.

El funcionamiento general de las Máquinas de Turing es como se describe a continuación. Al comenzar a funcionar la MT, las primeras n celdas (las de más a la izquierda), para $n \geq 0$ finita, contienen la entrada o datos de la máquina, que consisten de una cadena de n símbolos, cada uno de ellos de un subconjunto de símbolos, llamados símbolos de entrada, escogidos del conjunto de símbolos de la cinta. El resto de las celdas, un número infinito, contienen un símbolo especial que no forma parte de los símbolos de entrada, digamos "$\Box$" (un blanco). En un movimiento, la MT, dependiendo del símbolo que esté revisando la cabeza lectora y del estado en el que se encuentre el control finito, ejecuta lo siguiente:

1. Cambia de estado.

2. Imprime un símbolo en la celda que está revisando, reemplazando al símbolo que había ahí antes y que fue el que determinó, junto con el estado, el movimiento de la MT, y

3. Mueve su cabeza lectora a la celda contigua izquierda o derecha.

Formalmente, una Máquina de Turing M es un séptuplo

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$$

donde:

Q	es un conjunto finito de estados
Γ	es un conjunto finito de *símbolos de la cinta* permitidos
$\square$	un símbolo de Γ, es el blanco.
$\Sigma \subset \Gamma$	no contiene a $\square$, es el conjunto de *símbolos de entrada*
δ	es una función que da el *siguiente movimiento* y está definida de la siguiente manera: $$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$$ donde $\{L, R\}$ denotan la dirección del movimiento de la cabeza lectora en la cinta, pudiendo ser ésta respectivamente hacia la izquierda (L) o la derecha (R). δ puede no estar definida para algunos argumentos
$q_0 \in Q$	es el *estado inicial*
$F \subseteq Q$	es el conjunto de *estados finales*

Ahora, la ejecución de una Máquina de Turing M se puede seguir a través de descripciones instantáneas, donde una descripción instantánea es una cadena $\alpha_1 q \alpha_2$. En esta cadena, q es el estado en el que se encuentra M, $\alpha_1, \alpha_2 \in \Gamma^*$ es el contenido de la cinta desde el extremo izquierdo hasta el símbolo distinto de $\square$ que se encuentre más a la derecha en la cinta, o el símbolo inmediatamente a la izquierda de la cabeza lectora.

7.1 Implementación del motor

La jerarquía de paquetes que contiene el motor de la máquina de Turing, la interfaz gráfica y todas las estructuras necesarias para la implementación de ambas es la mostrada en la figura 7.1.

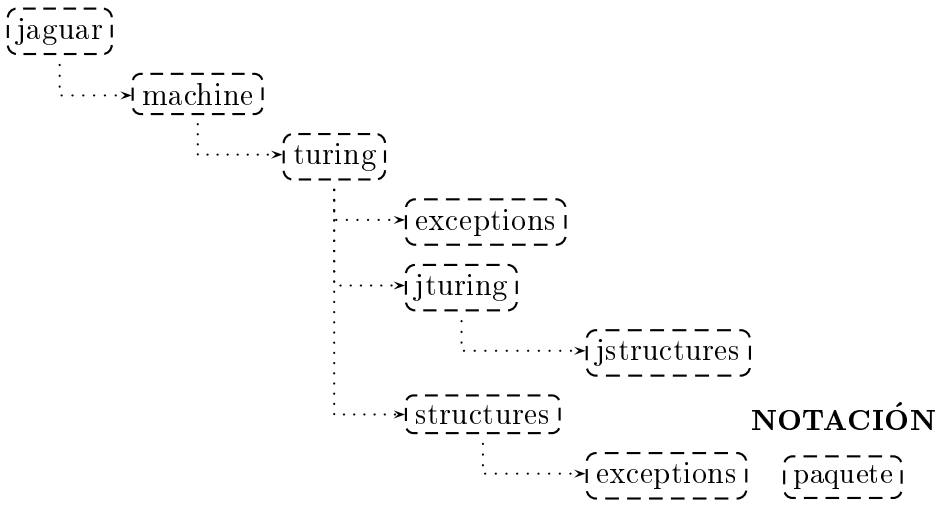

Figura 7.1: Paquetes para la Máquina de Turing

Las clases definidas y especializadas son las siguientes:

- jaguar.machine.turing.Turing. Esta clase es el motor y extiende a jaguar.machine.Machine. Recibe la especificación de una Máquina de Turing, la cual define el comportamiento; adicionalmente recibe el archivo donde se le especifica una cadena, sobre la cual se ejecutará.

- jaguar.machine.turing.structures.QxGammaxDirection. Esta clase representa la estructura $Q \times \Gamma \times \{L, R\}$.

- jaguar.machine.turing.structures.TuringDelta. Esta clase corresponde a la función de transición

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$$

7.1.1 Descripción de máquinas de Turing

Para especificar una Máquina de Turing por medio de etiquetas definimos los siguientes elementos:

Dirección, indica la dirección en la cual se debe de mover la Máquina de Turing en una transición. Puede ser una de las siguientes dos:

- A la izquierda, usaremos la marca `<left/>`
- A la derecha, usaremos la marca `<right/>`

Formalmente:

```
<!ELEMENT left EMPTY>

<!ELEMENT right EMPTY>
```

QxGammaxDirection es una triada ordenada entre las marcas `<QxGammaxDirection></QxGammaxDirection>`, donde el primer elemento es un estado en Q, el segundo es un símbolo en Γ y el tercero una dirección como la especificada antes. Formalmente:

```
<!ELEMENT QxGammaxDirection (state, sym, (left|right))>
```

Transición es una tercia ordenada dentro de los marcas `<trans> </trans>` donde el primer elemento es un estado en Q, el segundo un símbolo sobre Γ y por último un QxGammaxDirection como se acaba de definir. Formalmente:

```
<!ELEMENT trans (state, sym, QxGammaxDirection)>
```

Por ejemplo, si tuviésemos definida la transición

$$\delta(q0, 0) = (q1, X, R)$$

tendríamos dentro de δ una transición como:

```
<trans> <state>q0</state> <sym>0</sym>
        <QxGammaxDirection>
            <state>q1</state> <sym>X</sym> <right/>
        </QxGammaxDirection>
</trans>
```

Función de Transición δ es una colección de transiciones,

$$\boldsymbol{\delta} : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma^*)$$

con todos los elementos previamente definidos. Estas transiciones deben de estar dentro de las marcas `<delta> </delta>`. Formalmente:

```
<!ELEMENT delta (trans)*>
```

Máquina de Turing es un séptuplo $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ especificado dentro de las marcas `<turing> </turing>` y debe de tener en orden los siguientes elementos: una descripción de la Máquina de Turing (opcional), un conjunto de estados Q; un alfabeto de entrada Σ; un alfabeto de la cinta Γ; una función de transición δ; un estado inicial q_0; un símbolo que representa el blanco de la cinta; y un conjunto de estados finales F. Formalmente:

```
<!ELEMENT turing (description?,stateSet,alph,alph,delta,state,sym,stateSet)>
```

Por ejemplo:

```
1 <?xml version='1.0' encoding="iso-8859-1" ?>
2 <!DOCTYPE turing SYSTEM "turing.dtd">
3
4 <turing>
5
6  <description> Reconoce el lenguaje {0^n1^n | n > 1}</description>
```

```
7
8   <!-- El conjunto de estados  Q -->
9   <stateSet>
10    <state>q0</state>
11    <state>q1</state>
12    <state>q2</state>
13    <state>q3</state>
14    <state>q4</state>
15    <state>q5</state>
16  </stateSet>
17
18  <!-- El alfabeto Sigma -->
19  <alph>
20    <sym>0</sym> <sym>1</sym>
21  </alph>
22
23  <!-- El alfabeto Gama -->
24  <alph>
25   <sym>0</sym> <sym>1</sym> <sym>_</sym> <sym>X</sym> <sym>Y</sym>
26  </alph>
27
28  <!-- La función de transición Delta -->
29  <delta>
30   <trans> <state>q0</state> <sym>0</sym>
31           <QxGammaxDirection> <state>q1</state> <sym>X</sym> <right/>
32           </QxGammaxDirection>
33   </trans>
34   <trans> <state>q1</state> <sym>0</sym>
35           <QxGammaxDirection> <state>q1</state> <sym>0</sym> <right/>
36           </QxGammaxDirection>
37   </trans>
38   <trans> <state>q1</state> <sym>1</sym>
39           <QxGammaxDirection> <state>q2</state> <sym>Y</sym> <left/>
40           </QxGammaxDirection>
41   </trans>
42   <trans> <state>q1</state> <sym>Y</sym>
43           <QxGammaxDirection> <state>q1</state> <sym>Y</sym> <right/>
44           </QxGammaxDirection>
45   </trans>
46   <trans> <state>q2</state> <sym>0</sym>
47           <QxGammaxDirection> <state>q4</state> <sym>0</sym> <left/>
48           </QxGammaxDirection>
49   </trans>
50   <trans> <state>q2</state> <sym>X</sym>
51           <QxGammaxDirection> <state>q3</state> <sym>X</sym> <right/>
52           </QxGammaxDirection>
53   </trans>
54   <trans> <state>q2</state> <sym>Y</sym>
```

```
55            <QxGammaxDirection> <state>q2</state> <sym>Y</sym> <left/>
56            </QxGammaxDirection>
57    </trans>
58    <trans> <state>q3</state> <sym>Y</sym>
59            <QxGammaxDirection> <state>q3</state> <sym>Y</sym> <right/>
60            </QxGammaxDirection>
61    </trans>
62    <trans> <state>q3</state> <sym>_</sym>
63            <QxGammaxDirection> <state>q5</state> <sym>Y</sym> <right/>
64            </QxGammaxDirection>
65    </trans>
66    <trans> <state>q4</state> <sym>0</sym>
67            <QxGammaxDirection> <state>q4</state> <sym>0</sym> <left/>
68            </QxGammaxDirection>
69    </trans>
70    <trans> <state>q4</state> <sym>X</sym>
71            <QxGammaxDirection> <state>q0</state> <sym>X</sym> <right/>
72            </QxGammaxDirection>
73    </trans>
74   </delta>
75
76   <!-- El estado inicial -->
77   <state>q0</state>
78
79   <!-- El blanco -->
80   <sym>_</sym>
81
82   <!-- El conjunto de estados finales -->
83   <stateSet>
84     <state>q5</state>
85   </stateSet>
86
87  </turing>
```

7.1.2 Uso del motor

Este motor puede ejecutarse desde la consola, invocando a la clase `jaguar.machine.turing.Turing` con dos parámetros: el primero, un archivo con una especificación de la Máquina de Turing, como la descrita previamente, y el segundo un archivo con la especificación de una cadena. Por ejemplo, si tenemos la especificación de la máquina que acepta $L = \{0^n 1^n | n \geq 1\}$ en el archivo `tm-0n1n.xml` y la especificación de una cadena `000111` en otro llamado `cadena-test.xml`, ejecutaríamos el programa:

```
java machine.turin.Turing tm-0n1n.xml cadena-test.xml
```

La salida de ejecutar el comando anterior se muestra en la figura 7.2. Primero vemos la definición de la Máquina de Turing, después todas las descripciones instantáneas por las que pasa la Máquina de Turing durante la ejecución de la cadena. Finalmente un veredicto de aceptación.

```
TM  M =
        Q=[q1, q0, q5, q4, q3, q2]
        Sigma = <alph><sym>1</sym><sym>0</sym></alph>
        Gamma = <alph><sym>_</sym><sym>1</sym><sym>0</sym><sym>Y</sym><sym>X</sym></alph>
        delta =
          q1 1=( q2 , Y , <left/> ) , 0=( q1 , 0 , <right/> ) , Y=( q1 , Y , <right/> )
          q0 0=( q1 , X , <right/> )
          q4 0=( q4 , 0 , <left/> ) , X=( q0 , X , <right/> )
          q3 _=( q5 , Y , <right/> ) , Y=( q3 , Y , <right/> )
          q2 0=( q4 , 0 , <left/> ) , Y=( q2 , Y , <left/> ) , X=( q3 , X , <right/> )
        q0 = q0
        blanco = _
        F= [q5]
       [q0, 0, 0, 0, 1, 1, 1, _]
    |- [X, q1, 0, 0, 1, 1, 1, _]
    |- [X, 0, q1, 0, 1, 1, 1, _]
    |- [X, 0, 0, q1, 1, 1, 1, _]
    |- [X, 0, q2, 0, Y, 1, 1, _]
    |- [X, q4, 0, 0, Y, 1, 1, _]
    |- [q4, X, 0, 0, Y, 1, 1, _]
    |- [X, q0, 0, 0, Y, 1, 1, _]
    |- [X, X, q1, 0, Y, 1, 1, _]
    |- [X, X, 0, q1, Y, 1, 1, _]
    |- [X, X, 0, Y, q1, 1, 1, _]
    |- [X, X, 0, q2, Y, Y, 1, _]
    |- [X, X, q2, 0, Y, Y, 1, _]
    |- [X, q4, X, 0, Y, Y, 1, _]
    |- [X, X, q0, 0, Y, Y, 1, _]
    |- [X, X, X, q1, Y, Y, 1, _]
    |- [X, X, X, Y, q1, Y, 1, _]
    |- [X, X, X, Y, Y, q1, 1, _]
    |- [X, X, X, Y, q2, Y, Y, _]
    |- [X, X, X, q2, Y, Y, Y, _]
    |- [X, X, q2, X, Y, Y, Y, _]
    |- [X, X, X, q3, Y, Y, Y, _]
    |- [X, X, X, Y, q3, Y, Y, _]
    |- [X, X, X, Y, Y, q3, Y, _]
    |- [X, X, X, Y, Y, Y, q3, _]
    |- [X, X, X, Y, Y, Y, Y, q5]
    |- [X, X, X, Y, Y, Y, Y, q5, _]

   La MT SI acepta 000111
```

Figura 7.2: Ejecución del motor con la especificación del archivo tm-0n1n.xml

7.2 Generación de la interfaz gráfica

Ahora, extenderemos el motor y algunas de las bibliotecas genéricas de las descritas en el capítulo 2. Así, definimos los siguientes componentes:

- machine.turing.jturing.JTuring. Clase que extiende al motor de Máquina de Turing en machine.turing.Turing e implementa a machine.JMachine. Proporciona las operaciones y mecanismos para realizar paso a paso la ejecución sobre una cadena y poder mostrar gráficamente los resultados de cada transición.

- machine.turing.jturing.JTuringFrame. Éste es el marco donde tendremos a todos los elementos gráficos de la Máquina de Turing. Extiende a machine.util.jutil.JMachineFrame la cual proporciona menús genéricos y botones de control para la ejecución. La especialización más importante hecha por esta clase

consiste en agregar al marco genérico el componente *Tape*, el cual simula la configuración de la Máquina de Turing por medio de descripciones instantáneas.

- machine.turing.jturing.JTuringCanvas. Esta clase extiende el lienzo genérico definido en machine.util.jutil.JMachineCanvas. En este componente se muestran gráficamente todos los elementos de la Máquina de Turing.

- machine.turing.jturing.jstructures.JTuringDelta. Extiende a la función de transición machine.turing.structures.TuringDelta, e implementa JDeltaGraphic. Ello le permite hacer uso de machine.util.jutil.JDeltaPainter para mostrar las transiciones definidas, estados finales e iniciales, así como la transición que usamos en cada instante.

7.3 Uso de la interfaz gráfica

Para hacer uso de la interfaz gráfica tenemos que ejecutar el JTuringFrame de la siguiente forma:

```
java machine.turin.jturin.JTuringFrame
```

A continuación se mostrará un JTuringFrame con todos los componentes ya conocidos de la super clase JMachineFrame y con el nuevo componente *Tape* que mostrará las descripciones instantáneas de cada transición.

Cargamos la descripción de una Máquina de Turing M que acepta el lenguaje

$$L(M) = \{0^n 1^n \ | n \geq 1\}$$

de un archivo (ver sección 2.1.2); construimos la cadena 000111 con la interfaz gráfica (ver sección 2.1.4); tendremos entonces el JTuringFrame como se muestra en la figura 7.3.

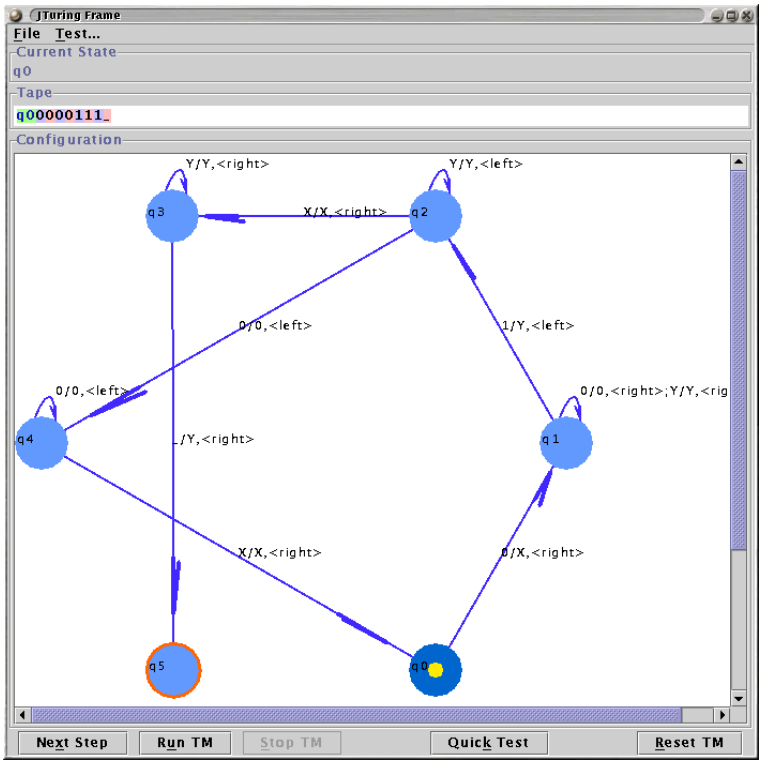

Figura 7.3: JTuringFrame

Ejecutar el autómata sobre esta cadena resultará en una serie de transiciones que se mostrarán en el lienzo y en los componentes *Current State* y *Tape*, terminando con un mensaje de si el autómata acepta la cadena de entrada.

(a) Instante i (b) Instante $i+1$

Figura 7.4: *Tape*: descripciones instantáneas

Resaltemos la representación de las descripciones instantáneas, que son justo como se describen en la introducción de este capítulo. En la figura 7.4.a, podemos observar la configuración del instante i, donde la Máquina de Turing está en el estado q_0, la cabeza lectora está leyendo el símbolo 0. El siguiente corresponde a la δ de la Máquina de Turing que tiene como sus primeros dos componentes a q_0 y 0, y que es:

$$\delta(q_0, 0) = (q1, X, R)$$

El resultado de lo anterior, se observa en la descripción instantánea del instante $i+1$ del componente *Tape*, figuras 7.4.b.

Finalmente, hay diversos modelos de Máquinas de Turing, equivalentes entre sí, que con muy poco trabajo, pueden manejarse como extensión de este modelo básico.

Capítulo 8

Conclusiones y trabajo a futuro

El software desarrollado puede ser usado como material de apoyo para el curso de Teoría de la Computación en la Carrera de Ciencias de la Computación, de la Facultad de Ciencias. Con esta herramienta los alumnos podrán probar a detalle el funcionamiento de sus diseños y seguir paso a paso algoritmos vistos en clase. Asimismo, se espera que el número de ejercicios que se asignen como tareas pueda ser mayor y el tiempo de prueba de sus diseños menor, incrementando así la práctica y aprendizaje, además de fomentar el interés en el material del curso.

El uso de un lenguaje orientado a objetos como Java fue una decisión atinada para la implementación del software desarrollado, pues el diseño de éste está fuertemente basado en patrones generales que de manera directa se convertían en clases abstractas o en interfaces de Java. Una vez definidas estas clases generales, la especialización de cada una de las gramáticas y máquinas era inmediata.

Para la extensión de este proyecto, continuaría implementando varios modelos de máquinas y operaciones sobre gramáticas. En máquinas los recomendados serían autómatas de stack no determinísticos, máquinas de Turing con varias cintas, y máquinas de Turing con varias cabezas lectoras. Para gramáticas recomendaría un convertidor de gramáticas libres de contexto a autómatas de stack, así como la implementación del Lema del Bombeo para gramáticas tipo 2 y tipo 3. Todo lo anterior se puede implementar usando la infraestructura desarrollada en este proyecto, el cual está ampliamente documentado.

Bibliografía

[1] FLANAGAN David. *Java in a Nutshell* O'Reilly. U.S.A. 1997.

[2] GOSLING James, Joy BILL, STEELE Guy. *The Java Language Specification*, Addison-Wesley, 1996.

[3] http://www.java.sun.com

[4] LAMBERT Kenneth, OSBORNE Martin. *JAVA A Framework for Programming and Problem Solving* Brooks/Cole Publishing Company. U.S.A. 1999.

[5] LINZ Peter. *An Introduction to Formal Languages and automata.* Jones and Bartlett Publishers, U.S.A. 2nd Ed, 1997.

[6] SIPSER Michael. *Introduction to the theory of computation* PWS Publishing Company, U.S.A. 1997.

[7] TAYLOR Ralph Gregory. *Models of Computation and Formal Languages* Oxford University Press, U.S.A.1997

[8] VISO Gurovich Elisa. *Teoría de la Computación* Facultad de Ciencias, UNAM. México 2000.

[9] http://www.w3.org/

[10] ECKSTEIN, Robert. *XML : pocket reference* O'Reilly, U.S.A. 1999

Apéndice A

Manual de uso

Todos los programas desarrollados forman parte de un solo paquete, al cual se le dio el nombre en inglés de "Java Automaton and Grammar User Aplication Resources" (jaguar), pues nos proporciona recursos para usar e implementar máquinas y gramáticas. Todo el código fuente queda liberado bajo la licencia GPL de la Free Software Foundation, la cual se incluye en las distribuciones del proyecto.

A.1 Requerimientos

Para ejecutar las aplicaciones se necesita una máquina virtual de Java2 $\geq$ v1.4.0. El proyecto fue compilado y probado en todo momento usando una la máquina virtual de Java2 v1.4.0 que distribuye el proyecto *blackdown* (http://www.blackdown.org). Esta versión de máquina virtual también se puede obtener en http://www.java.sun.com. Los requerimientos mínimos son los que se especifican en los sitios antes mencionados para ejecutar las máquinas virtuales.

A.2 Dónde obtener jaguar

La última versión de jaguar se puede obtener en el sitio web http://sourceforge.net/projects/ijaguar/. Ahí se encuentra el código fuente y los binarios. En este anexo hacemos referencia a la versión x-yz. En las siguientes secciones se muestra cómo compilar los fuentes y cómo ejecutar el comando JCenter, el cual se describirá en su momento.

A.2.1 Versión fuente

Supongamos que el código fuente de la aplicación está en el paquete `jaguar.x-yz.src.tbz2`. Una vez que se obtiene el código fuente del sitio web antes mencionado, se tiene que descomprimir este paquete, obteniendo un directorio `jaguar.x-yz/`. Dentro de él se encontrará el directorio `jaguar`. Una vez en el directorio `jaguar.x-yz/jaguar/` se tiene que usar el comando `make`, el cual se encargará de compilar todos los fuentes. Ahora, desde el directorio `jaguar.x-yz/` se puede invocar al JCenter de la siguiente manera:

```
java jaguar.JCenter
```

A.2.2 Los binarios

Supongamos que la distribución binaria de la aplicación se encuentra en el paquete `jaguar.x-yz.jar`. Una vez que se obtiene este paquete del sitio web antes mencionado, para ejecutar la aplicación se tiene que invocar al JCenter de la siguiente manera:

```
java -jar jaguar.x-yz.jar
```

A.3 Cómo usar el JCenter de jaguar

El paquete jaguar proporciona un centro de control para todos las interfaces gráficas descritas a lo largo de este trabajo. Este centro de control se encuentra en la clase JCenter, el cual se puede invocar de dos formas, dependiendo del tipo de distribución que se obtenga, como se describió en las secciones A.2.1 y A.2.2. Una vez que se invoca a JCenter, tenemos una interfaz gráfica como en la figura A.1.

Figura A.1: JCenter

Es claro el uso de la interfaz: al presionar un botón se mostrará la interfaz gráfica asociada a éste; así no tenemos que recordar las rutas completas de cada uno de los paquetes mostrados en el trabajo, ya que todos pueden ser ejecutados desde el centro de control.

A.4 Descripciones muestra de autómatas y gramáticas

Todas las definiciones de autómatas o gramáticas que se quieran cargar a la aplicación deben de encontrarse en archivos con formato de marcas[1] como se especificó en las secciones anteriores. Estos archivos deberán crearse antes de intentar ejecutar una aplicación. La distribución cuenta con los siguientes archivos que se pueden usar para ejemplificar:

- Autómatas finitos determísticos
  - `dfa-cero-star-uno-cero-plus.xml`. Autómata finito determínistico que acepta el lenguaje $L = \{0^*10^+\}$

[1] tags

  - `dfa-paridad.xml`. Autómata finito determínistico que acepta el lenguaje $L = \{w \in \{0,1\}^* \mid \#1(w) = 2n, n \geq 0\}$
  - `dfa-001.xml`. Autómata finito determínistico que acepta el lenguaje $L = \{x001y \mid x, y \in \{0,1\}^*\}$

- Autómatas finitos no determísticos
  - `ndfa-bar.xml`. Autómata finito no determínistico que acepta el lenguaje $L = \{b^*((ab^*)|a^+)\}$
  - `ndfa-foo.xml`. Autómata finito no determínistico que acepta el lenguaje $L = \{(01^+)^+\}$

- Autómata de stack
  - `sfa-an-bn-final-state.xml`. Autómata de stack que acepta por estado final el lenguaje $L = \{a^n b^n | n \geq 0\}$
  - `sfa-an-bn-empty-stack.xml`. Autómata de stack que acepta por estado final el lenguaje $L = \{a^n b^n | n \geq 0\}$

- Máquinas de Turing
  - `tm-an-bn.xml`. Máquina de Turing que acepta el lenguaje $L = \{a^n b^n \ | n > 0\}$
  - `tm-0-2-radical-n.xml`. Máquina de Turing que acepta el lenguaje $L = \{0^{2^n} \ | n \geq 0\}$

- Gramáticas tipo 3
  - `gram-001.xml`. Gramática tipo 3 que genera el lenguaje $L = \{x001y \mid x, y \in \{0,1\}^*\}$
  - `gram-an-bn.xml`. Gramática tipo 3 que genera el lenguaje $L = \{a^n b^m | n \geq 2, m \geq 3\}$

- Gramáticas tipo 2
  - `gram-prodEpsilon.xml`. Gramática tipo 2 floja con producciones epsilon.
  - `gram-prodUnit.xml`. Gramática tipo 2 floja con producciones unitarias.

Apéndice B

Jerarquía de paquetes

La jerarquía de paquetes de las gramáticas generadoras implementadas se muestra en la figura B.1.

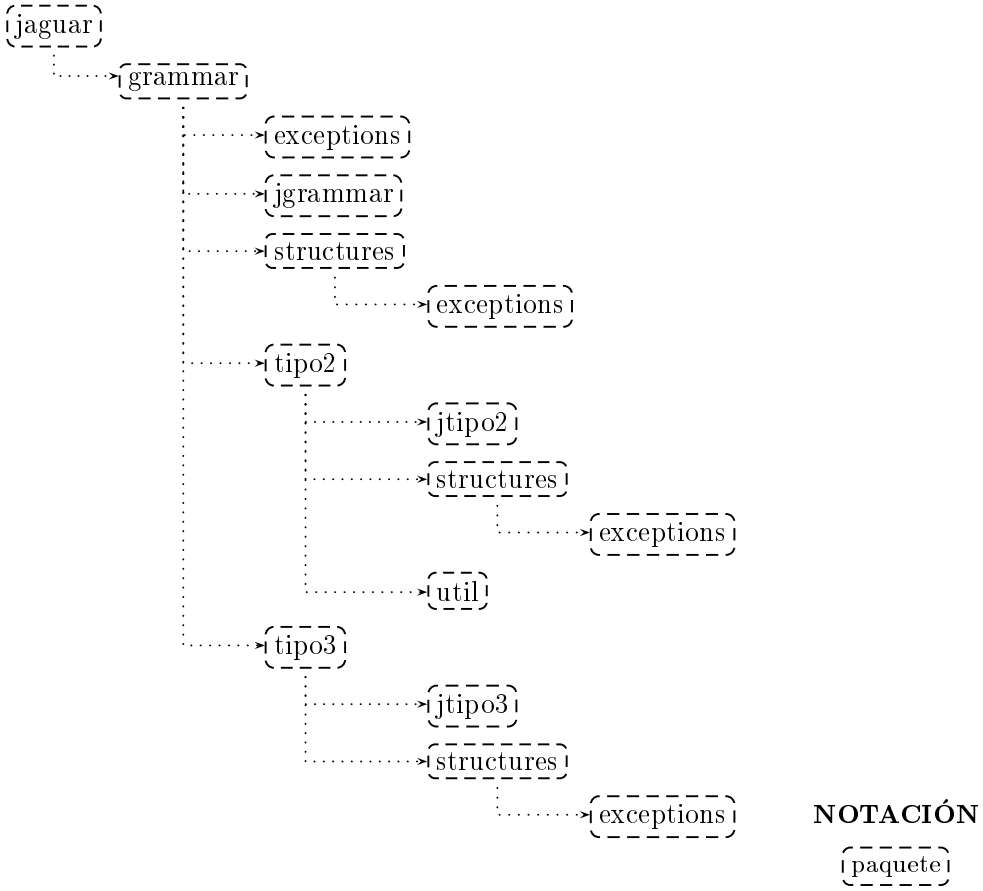

Figura B.1: Jerarquía de gramáticas

La jerarquía de paquetes de las máquinas aceptadoras implementadas se muestra en la figura B.2.

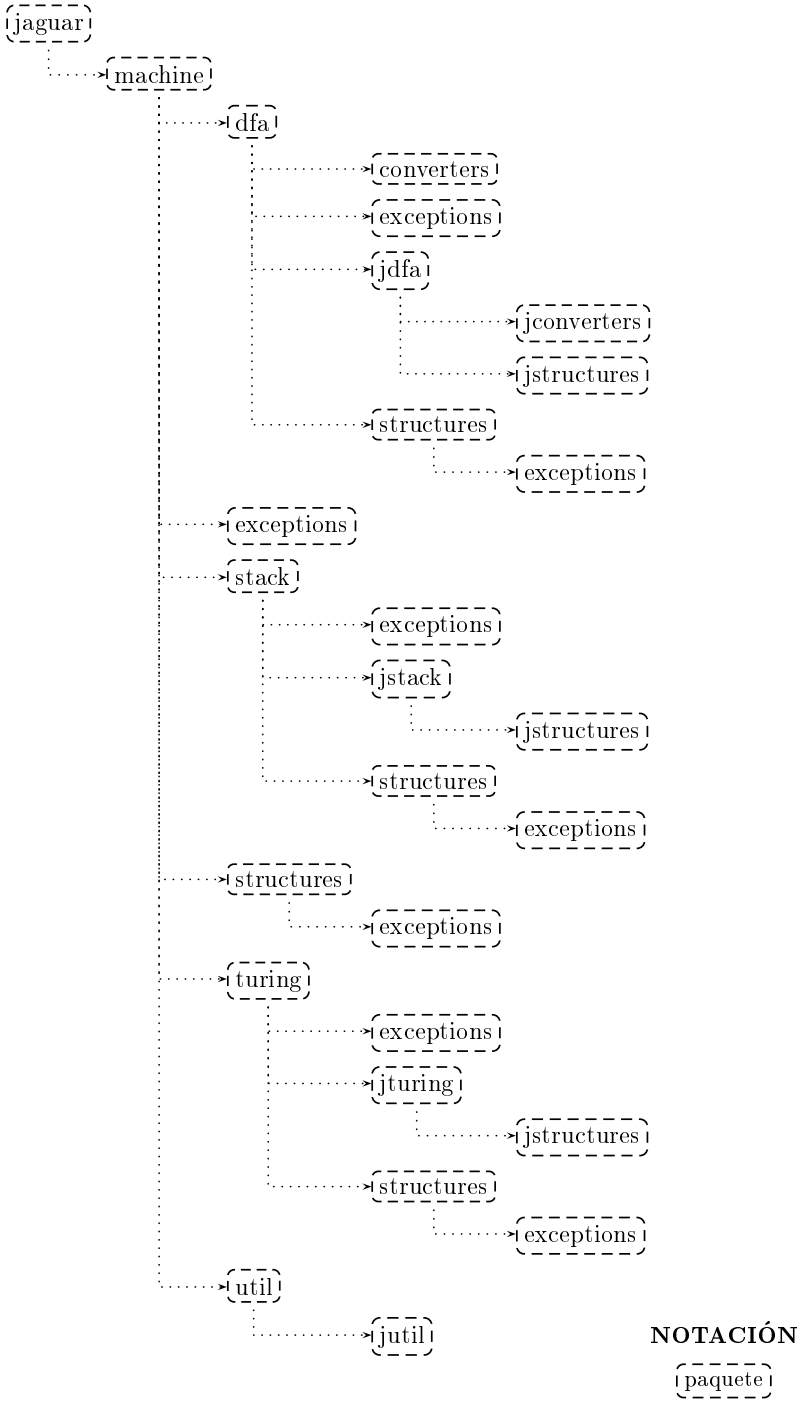

Figura B.2: Jerarquía de máquinas

La jerarquía completa de los paquetes definidos en este trabajo se muestra en la figura B.3.

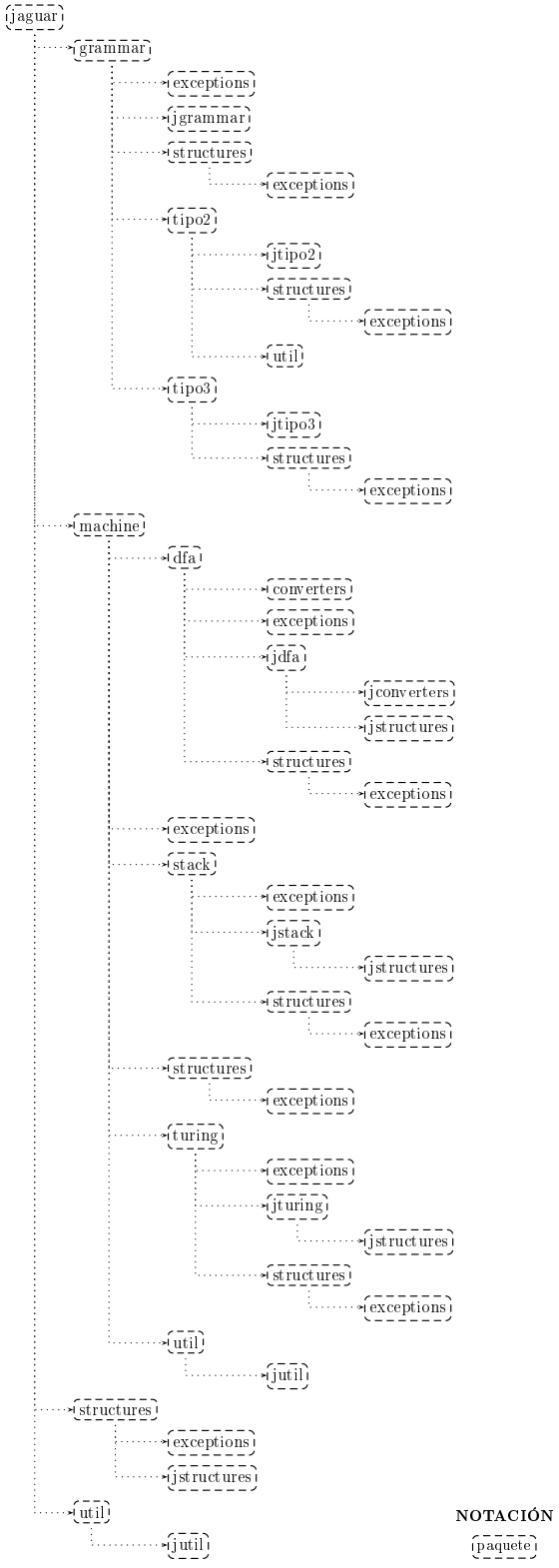

Figura B.3: Jerarquía completa de paquetes

Apéndice C

DTDs

A continuación mostraremos los DTDs para cada uno los modelos de máquinas y gramáticas revisadas en los captítulos anteriores. Todos los archivos están contenidos en el CD-ROM que acompaña este trabajo.

C.1 dfa.dtd

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!ELEMENT sym (#PCDATA)>
<!ELEMENT alph (sym)*>
<!ELEMENT str (epsilon|(sym)+)>
<!ELEMENT epsilon EMPTY>
<!ELEMENT state (#PCDATA)>
<!ATTLIST state xpos CDATA #IMPLIED
                ypos CDATA #IMPLIED>
<!ELEMENT stateSet (state)*>

<!ELEMENT description (#PCDATA)>
<!ELEMENT trans (state, sym, state)>
<!ELEMENT delta (trans)*>
<!ELEMENT dfa (description?, stateSet, alph, delta, state, stateSet)>
```

C.2 ndfa.dtd

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!ELEMENT sym (#PCDATA)>
<!ELEMENT alph (sym)*>
<!ELEMENT str (epsilon|(sym)+)>
<!ELEMENT epsilon EMPTY>
<!ELEMENT state (#PCDATA)>
<!ATTLIST state final (true|false) #IMPLIED
                xpos CDATA #IMPLIED
                ypos CDATA #IMPLIED>
<!ELEMENT stateSet (state)*>
```

```
<!ELEMENT description (#PCDATA)>

<!ELEMENT trans (state, sym, stateSet)>
<!ELEMENT delta (trans)*>
<!ELEMENT ndfa (description?, stateSet, alph, delta, stateSet, stateSet)>
```

C.3 afs.dtd

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!ELEMENT sym (#PCDATA)>
<!ELEMENT alph (sym)*>
<!ELEMENT str (epsilon|(sym)+)>
<!ELEMENT epsilon EMPTY>
<!ELEMENT state (#PCDATA)>
<!ATTLIST state xpos CDATA #IMPLIED
                ypos CDATA #IMPLIED>
<!ELEMENT stateSet (state)*>
<!ELEMENT description (#PCDATA)>

<!ELEMENT QxGammaStar (state, str)>
<!ELEMENT QxGammaStarSet (QxGammaStar)*>
<!ELEMENT trans (state, sym, sym, QxGammaStarSet)>
<!ELEMENT delta (trans)*>
<!ELEMENT stack (description?,stateSet, alph, alph, delta, state, sym, stateSet)>
```

C.4 turing.dtd

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!ELEMENT sym (#PCDATA)>
<!ELEMENT alph (sym)*>
<!ELEMENT str (epsilon?|(sym)+)>
<!ELEMENT epsilon EMPTY>
<!ELEMENT state (#PCDATA)>
<!ATTLIST state xpos CDATA #IMPLIED
                ypos CDATA #IMPLIED>
<!ELEMENT stateSet (state)*>
<!ELEMENT description (#PCDATA)>

<!ELEMENT left EMPTY>
<!ELEMENT right EMPTY>
<!ELEMENT QxGammaxDirection (state, sym, (left|right))>
<!ELEMENT trans (state, sym, QxGammaxDirection)>
<!ELEMENT delta (trans)*>
<!ELEMENT turing (description?,stateSet, alph, alph, delta, state, sym, stateSet)>
```

C.5 grammar.dtd

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!ELEMENT sym (#PCDATA)>
<!ELEMENT alph (sym)*>
<!ELEMENT str (epsilon|(sym)+)>
<!ELEMENT epsilon EMPTY>

<!ELEMENT left (str)>
<!ELEMENT right (str)>
<!ELEMENT p (left, right)>
<!ELEMENT productionSet (p)*>
<!ELEMENT description (#PCDATA)>
<!ELEMENT gram (description?,alph,alph,productionSet,sym)>
```