Trabajo Práctico Especial

Autómatas, Teoría de Lenguajes y Compiladores 72.39

Grupo G53: AutomaTex

1C - 2024

Tabla de Contenidos

1. Equipo

Nombre	Apellido	Legajo	E-mail
Sol	Rodriguez	63029	**solrodriguez@itba.edu.ar**
Maria Agustina	Sanguinetti	63115	**msanguinetti@itba.edu.ar**
Uriel Ángel	Arias	63504	**uarias@itba.edu.ar**

2. Repositorio

La solución y su documentación serán versionadas en: AutomaTeX.

3. Introducción

En el presente trabajo se describe el desarrollo de un compilador de autómatas finitos, donde se diseña un lenguaje de dominio específico (DSL) destinado a la definición y simulación de autómatas finitos. El objetivo es que, tras compilar un archivo escrito en este lenguaje, se obtenga una representación de los autómatas definidos en formato LaTeX, que incluya tanto el diagrama de estados como la tabla de transiciones correspondiente.

Para la implementación del compilador, se utilizó el lenguaje de programación C junto con las herramientas Flex y Bison, encargadas de realizar el análisis léxico y sintáctico respectivamente. Este informe detalla el proceso de desarrollo, los desafíos encontrados, las soluciones implementadas, entre otros, para la creación del compilador.

4. Modelo Computacional

4.1. Dominio

Se desarrolló un lenguaje de dominio específico (DSL) que permite la creación de autómatas finitos, de manera tal que una vez compilado un archivo escrito en este lenguaje se devuelve una representación equivalente de los autómatas definidos por medio de un documento en formato LaTeX, el cual contiene sus respectivos diagramas de estados y tablas de transiciones.

Permite a los usuarios definir uno o más autómatas, especificando sus estados terminales y no terminales, su estado inicial, las transiciones entre sus estados, y los alfabetos que los mismos reconocerán.

Cabe destacar que si bien se reconoce que se podría manejar la definición de autómatas sin especificar su tipo dadas las equivalencias probadas entre los mismos y la existencia de algoritmos para hacer la conversión de uno a otro, se hace uso de tipos de autómatas para reducir la complejidad del proyecto y aprovechar el uso de errores de compilación para marcar errores conceptuales propios de usuarios principiantes con los conceptos teóricos. Se trata entonces de un aspecto sobre el cual el proyecto podría seguir escalando para hacer que su uso sea menos restrictivo y permita la reutilización de los autómatas ya definidos.

Por otro lado, junto con la definición de autómatas se hace necesario el manejo de conjuntos que bien podrían reutilizarse para autómatas con propiedades similares, por ejemplo, al compartir los mismos estados. Es por esto que se provee de tipos de datos que si bien refieren a las propiedades de los autómatas, como estados, transiciones y un alfabeto, programáticamente se distinguen por usar notación y operaciones binarias básicas de conjuntos, entre las cuales se encuentran la unión, la intersección y la diferencia.

La implementación exitosa de este lenguaje proporcionará una herramienta poderosa para desarrolladores de sistemas basados en autómatas finitos, permitiéndoles modelar y verificar la correctitud de los autómatas definidos de una forma clara y fácil de utilizar para que el usuario no pierda tiempo en recordar la sintaxis de LaTeX.

4.2. Lenguaje

1. Comentarios

```
// Esto es un comentario de una línea

/* Esto es un comentario
que puede contener una o más líneas
*/
```

2. Conjuntos

Los conjuntos serán definidos mediante llaves (“{“, “}”) y sus elementos se deberán separar con comas. No obstante, también se podrá definir un conjunto de un único elemento sin llaves, de manera tal que se asumirá que el conjunto solo tiene un elemento.

3. Definición de un autómata

Creación de un autómata AFD, al cual se lo nombra *Automata1*, con los estados definidos en *states*, el alfabeto indicado en a*lphabet* y las transiciones en *transitions*. El autómata Automata1 aparecerá en el archivo final de LaTeX.

```
// Creamos un autómata AFD
DFA Automata1 [

       // Declaramos los estados
       states: { u, >s, *w, *q },

       // Definimos el alfabeto
       alphabet: {a, b},

       // Declaramos las transiciones
       transitions: {
              |s|-b->|u| ,
              |w|<-b->|u|,      // Equivale a |w|-b->|u| y |u|-b->|w|
              |{s,u}|-a->|w|,   // Equivale a |s|-a->|w| y |u|-a->|w|
              |q|-{a,b}->|w|,
              |q|<-a-|w|
       }
];
```

- **Tipos de autómatas:** pueden ser de tipo DFA para autómatas deterministas, AFN para autómatas no deterministas y AFND para autómatas no deterministas lambda.

- **Estados iniciales y finales:** se utiliza un “>” al inicio de un estado para indicar que es un estado inicial. Para indicar que un estado es final se utiliza “*” *al inicio del estado. En el caso que un estado sea inicial y final simultáneamente se utiliza* “>”.

- **Transiciones:** el lenguaje permite definir transiciones de la forma *|p|-a->|q|*, *|p|<-a-|q|* o *|p|<-a->|q|*, siendo *p* y *q* dos estados o conjuntos de estados y *a* un símbolo o conjunto de símbolos del alfabeto definido para un autómata, donde en el primer caso se define una transición del estado *p* a *q* consumiendo *a*, en el segundo caso se define una transición del estado *q* a *p* consumiendo *a*, y en el último caso se definen ambas transiciones mencionadas anteriormente. Si se quiere hacer referencia a más de 2 estados o símbolos, se podrán listar dichos estados o símbolos del alfabeto entre paréntesis, tal como se ve en el siguiente ejemplo: *|{q0, q1}|-{a,b}->|q2|*, que indica que los estados *q0* y *q1* al consumir *a* o *b* llegan al estado *q2*, es decir, que en una única sentencia se representan 4 transiciones distintas.

- **Transiciones lambda:** se emplea el símbolo @ para representar lambda en aquellas transiciones que pertenezcan a un AFND.

4. Definición de estados, transiciones y símbolos globales

Se pueden crear estados, transiciones y alfabetos independientemente de los autómatas, es decir, como constantes globales. Las mismas pueden ser de tipo *states* para los estados, *transitions* para las transiciones, *alphabet* para el alfabeto y para todas ellas la letra inicial del nombre u identificador deberá estar en mayúscula.

```
states T: t;
transitions Transition:  |s|-a->|{w,q}|;    // = {s-a->w, s-a->q}
alphabet A: a;
```

5. Definición de conjuntos de estados, transiciones y símbolos globales

```
states Q: { *r, o, j };
transitions S1: {|r|-a->|j|, |r|-b->|r|, |j|-{a,b}->|r|};
alphabet MyAlphabet: { a,b };
```

6. Utilización de operaciones de conjuntos

Se pueden emplear operadores para realizar operaciones entre conjuntos del mismo tipo. Los operadores provistos son: el "+" para la unión, el "^" para la intersección y el "-" para la diferencia.

```
states S: ({s,r}^{t,f,r}) + ({u,o} - {o});
      // = ({s,r} ∩ {t,f,r}) ∪ ({u,o} - {o})
```

- Los operadores se pueden usar tanto para constantes, como para conjuntos no vinculados a un nombre o etiqueta. Será válida una expresión de la forma A * B, siendo A y B dos constantes que son del mismo tipo y "*" un operador dentro del conjunto {+, -, ^}, y de la forma A * {*a*, *b*, *c*} donde A es de tipo *alphabet* o *states*, dependiendo de si {a, b, c} son elementos del alfabeto o estados.

- Los operadores se pueden usar tanto para

- Los operadores tienen la misma precedencia. Si se desea indicar cierto orden a la hora de resolver un conjunto de operaciones, se deben usar paréntesis, sino por defecto la lectura se llevará a cabo de izquierda a derecha.

7. Obtención de los estados, alfabeto y transiciones de un autómata

Una vez creado el autómata, se puede acceder a sus estados, transiciones y alfabeto usando el nombre del autómata seguido de un punto y el nombre de dicho conjunto.

```
states Q: {*r, o};

NFA Automata [
	states: { w, >q, Q.final },	// Q.final = {*r}
	alphabet: {a, b},
	transitions: {
		|q|-a->|r| ,
		|w|<-b->|r|
	}
];

alphabet A: Automata.alphabet;		// A = {a, b}
states S: Automata.states;		// S = {*r, w, >q}
states I: Automata.states.initial;	// I = {>q}
```

- Para acceder, por ejemplo, a los estados de un autómata se deberá escribir lo siguiente: *NombreAutomata.states*, donde se tendrá un conjunto formado por 3 subconjuntos, uno de estados regulares, uno con el estado inicial y uno con los estados finales.

- Para acceder a sólo un subconjunto de estados, se debe especificar el requerido de la siguiente forma NombreAutomata.states.tipo donde tipo puede ser reemplazado por las palabras reservadas *initial*, *final* o *regular*.

8. Ejemplos mixtos

Creación de un autómata AFND, al cual se lo nombra *Automata2*, y de uno AFND-Lambda, *Automata3*, con los estados definidos en *states*, el alfabeto indicado en *alphabet* y las transiciones en *transitions*. Se agregan variables y constantes globales del tipo state, transitions y alphabet para definir estados, transiciones y alfabetos respectivamente que puedan ser utilizados por cualquier autómata. A su vez, se agregan operaciones de conjuntos. Los autómatas Automata2 y Automata3 aparecerán en el archivo final de LaTeX.

```
// Definimos valores constantes y variables que podrán ser empleados para crear autómatas
// la letra inicial del nombre de dichas constantes debe estar en mayúscula

states Q: { r, o, j };
transitions R:  |s|-a->|{w,q}|;	// = {s-a->w, s-a->q}
alphabet A: { a,b };
states T: t;			// t = {t}
```

```
// Creamos un autómata AFND
NFA Automata2 [
	states: { Q, >s , *w, *q },

	alphabet: A,

	transitions: {
		|q|-a->|r| ,
		|w|<-b->|r|,
		R
	}
];

// Creamos nuevas constantes para armar el nuevo autómata
// Usamos operadores
states H: { h, o };
transitions S1: {|r|-a->|j|, |r|-b->|r|, |j|-{a,b}->|r|};
transitions S2: {|j|-a->|r|};
transitions S3: S1 - S2;	// ⇔ {|r|-a->|j|, |r|-b->|j|, |j|-b->|r|}
states KO: {k, o, r, j};

// Creamos un autómata AFND-lambda
LNFA Automata3 [
	states: { >s , *w, *q, (KO-T), bin },
	// (KO^T) = ({k, o, r, j} - t)

	alphabet: {a, b, cc, dd},

	transitions: {
		|w|<-@->|r|,	// = w consume lambda para llegar a r y viceversa
		|s|-cc->|{w,q}|,
		|w|-{cc,dd}->|r|, S3
	}
];
```

5. Implementación

5.1. Frontend

En esta sección, se detalla el desarrollo del frontend del compilador para el cual se utilizó Flex y Bison. El frontend es responsable del análisis léxico y sintáctico del código fuente escrito en el lenguaje de dominio definido anteriormente.

En cuanto al analizador léxico, este se encarga de descomponer el código fuente en tokens, en otras palabras, unidades léxicas significativas para el compilador. Para ello se creó el archivo *FlexPatterns.l*, el cual contiene las definiciones de los patrones, como palabras reservadas, operadores y símbolos, que serán reconocidos por el analizador. El archivo *FlexActions.c* define las funciones que ejecutan las acciones correspondientes a los tokens reconocidos. Cada función realiza acciones como registrar el token, ignorar el lexema, o devolver el token correspondiente.

Por otro lado, el archivo *BisonGrammar.y* define la gramática del lenguaje definido. Esta gramática se utiliza con Bison para construir un analizador sintáctico, que convierte el código fuente en una representación interna estructurada, llamada Árbol de Sintaxis Abstracta (AST). En este archivo se distinguen los tokens terminales de los no terminales donde los terminales representan los elementos básicos del lenguaje, como palabras clave, símbolos y operadores, mientras que los no-terminales representan construcciones más complejas. Las reglas que identifican las construcciones válidas del lenguaje se definen en este archivo, donde cada una de ellas está asociada con una acción que se ejecuta cuando se reconoce la regla. Estas acciones son funciones que se encuentran en el archivo *BisonActions.c*.

A continuación se especifican todas las estructuras de datos utilizadas, las cuales son fundamentales para la organización y manipulación de conjuntos, transiciones, estados, símbolos, definiciones, operaciones, entre otras.

- **Definition**

Representa una definición que puede ser un conjunto de estados, un conjunto de símbolos, un conjunto de transiciones o un autómata. Se utiliza *DefinitionSet* para tratar conjuntos de

definiciones.

```
struct Definition {
    union {
        Automata * automata;
        StateSet * stateSet;
        SymbolSet * symbolSet;
        TransitionSet * transitionSet;
    };
    DefinitionType type;
};
```

- **Sets**

Se definen conjuntos de estados, símbolos (alfabeto), transiciones y definiciones respectivamente como *StateSet*, *SymbolSet*, *TransitionSet* y *DefinitionSet*. Para este punto y los puntos siguientes se tomará a modo representativo las estructuras definidas para la manipulación de transiciones.

```
struct TransitionSet {
    TransitionNode * first;
    TransitionNode * tail;
    char * identifier;
    boolean isFromAutomata;
    boolean isBothSidesTransition;
};
```

- **Nodos**

Para múltiples de las estructuras mencionadas anteriormente, se puede ver la definición de nodos. La utilización de nodos permite la creación de listas enlazadas, donde cada nodo contiene información y apunta al siguiente, formando así una secuencia de elementos.

```
struct TransitionNode {
    union {
        Transition * transition;
        TransitionExpression * transitionExpression;
```

```
        TransitionSet * transitionSubset;
    };
    NodeType type;
    TransitionNode * next;
};
```

- **Elementos**

Se definen las estructuras para los siguientes elementos básicos para la construcción de un autómata: *transition*, *symbol* y *state*.

```
struct Transition {
    StateExpression * fromExpression;
    SymbolExpression * symbolExpression;
    StateExpression * toExpression;
};
```

- **Expressions**

Las expresiones pueden ser un conjunto de elementos, un único elemento, o una combinación de expresiones. Para el caso de las transiciones, una expresión puede definirse como un conjunto de transiciones, una única transición, o una combinación de expresiones de transiciones. Las expresiones tienen un tipo, para identificar a cuál de las tres opciones pertenece la expresión (set, elemento o operación) y también para setear el tipo de operador (unión, intersección o diferencia) en caso que sea una operación entre dos expresiones.

```
struct TransitionExpression {
    union {
        TransitionSet * transitionSet;
        struct {
            TransitionExpression * leftExpression;
            TransitionExpression * rightExpression;
        };
        Transition * transition;
    };
    ExpressionType type;
};
```

❖ **Autómata**

La estructura *Automata* incluye un identificador, un tipo de autómata, y expresiones de estados, alfabeto y transiciones.

```
struct Automata {
    char * identifier;
    StateExpression * states;
    StateExpression * finals;
    StateExpression * initials;
    SymbolExpression * alphabet;
    TransitionExpression * transitions;
    AutomataType automataType;
};
```

❖ **Program**

Representa el programa completo que contiene el conjunto de definiciones.

```
struct Program {
    DefinitionSet * definitionSet;
};
```

5.2. Backend

❖ **Automatex.c**

Este componente se encarga del procesamiento de las estructuras que se envían desde el frontend. Al tratarse de funciones que requieren agregar la lógica propia del dominio específico elegido se decidió agrupar en este mismo archivo, aunque podría modularizarse, según las diferentes secciones que se indican con comentarios. De esta forma se cuenta con diversas funciones que se encargan de tomar los nodos del árbol generado desde el frontend y se verifica que se cumpla con las restricciones del dominio, tales como las condiciones de los autómatas según su tipo y las operaciones entre conjuntos. A su vez, también se hacen otras verificaciones más generales como el uso de identificadores que se definieron previamente y el correcto uso de los tipos de datos definidos.

Resulta relevante mencionar que la mayor parte de las funciones que se encargan de estas tareas retornan una estructura de tipo *ComputationResult* que utiliza el tipo *union* para transportar el resultado de procesar autómatas, conjuntos y elementos sueltos, como estados, símbolos del alfabeto y transiciones. Al mismo tiempo, se agregaron algunas variables como *isDefinitionSet*, *isSingleElement* y *definitionType* para indicar que tipo de resultado se obtiene dada la variedad de estructuras que se soporta. Finalmente, la principal variable de esta estructura que se utiliza luego de llamar a estas funciones es la de *success* que permite verificar que se pudo realizar correctamente el procesamiento y con esto seguir avanzando por los nodos del árbol. En conjunto con esta estructura, en cada paso que se realiza al recorrer el árbol se termina modificando el contenido del mismo, dado que, por ejemplo, en los conjuntos que contaban con expresiones para resolver, éstas se reemplazarán por el resultado obtenido a partir de su evaluación, las cuales en general terminan siendo elementos sueltos. Se define la estructura de *ComputationResult* de la siguiente manera:

```
typedef struct {
        boolean succeed;
        union {
            DefinitionSet * definitionSet;
            Automata * automata;
            TransitionSet * transitionSet;
            SymbolSet * symbolSet;
            StateSet * stateSet;
            Transition * transition;
            Symbol * symbol;
            State * state;
        };
        boolean isDefinitionSet;
        boolean isSingleElement;
        DefinitionType type;
} ComputationResult;
```

- **Table.c**

Este componente se encarga de la creación y el manejo de la tabla de símbolos empleada para guardar las constantes definidas. Para la creación de la tabla, se empleó un hashmap que lo

implementa la librería externa *klib*, en particular en el archivo "khash.h"[1], obtenida del repositorio adjuntado en la sección de referencias, item 2. Por lo tanto, Table.c se encarga de inicializar un hashmap y provee funciones para poder acceder y modificar al mismo.

Las entradas de la tabla están formadas por una clave de *char * identifier* y un valor de tipo *Entry*. Dentro de este último se colocan 2 valores, uno que indica el tipo de dato al que apunta, *ValueType type*, y otro contiene un puntero al valor original, *Value value*. Por ende, es necesario recalcar que los valores originales, por ejemplo un set de estados, no son copiados a la tabla, sino que se guarda únicamente un puntero a ellos. Entonces, al buscar un valor en la tabla con la función *getValue()*, se obtiene un puntero que apunta al valor original, por lo que es responsabilidad del cliente, es decir de quien invocó a la función, hacer una copia del mismo de ser necesario.

En cuanto a *ValueType*, éste permite que existan 4 tipos de valores que se pueden ingresar a la tabla: autómata, estados, transiciones y alfabetos. Estos tipos son equivalentes a aquellos empleados en las definiciones. Cabe destacar que dado que el hash se lleva a cabo únicamente mediante el *identifier*, no se permitirá ingresar dos entradas a la tabla con el mismo valor de *identifier*, sin importar si éstas guardan valores de distinto tipo.

Dado que la clase *Entry* no se encuentra disponible para el usuario, *EntryResult* es la clase que retorna los resultados obtenidos de una búsqueda en la tabla con *getValue()*. En ella, se indica si la búsqueda fue exitosa con *found*, y de serlo, en *value* contiene el resultado de la misma. Para las demás funciones como *exist()* o *insert()* se retorna un valor del tipo boolean para indicar si la operación fue exitosa.

- **Generator.c**

Este componente es responsable de generar la salida final del programa. A partir de las estructuras construidas y modificadas en *Automatex*, *Generator* toma estos datos para representar los autómatas ingresados por el usuario en formato Latex, mostrando sus respectivos diagramas de estados y tablas de transiciones[2]. Para ello, se crean dos matrices que son fundamentales para la creación de los mismos: *TransitionMatrixCell* y *AutomataMatrixCell*.

[1] se encuentra descargado junto al resto de archivos de backend, dentro del directorio *domain-specific*.
[2] Ver apéndices I y II.

TransitionMatrixCell almacena las transiciones de estado a estado en función de los símbolos del alfabeto del autómata y se utiliza para armar la tabla de transiciones que luego se mostrará en Latex. Cada celda de la matriz contiene una lista enlazada de nodos de transición (*MatrixNode*) que representan las transiciones válidas desde un estado concreto utilizando un símbolo específico. Por otro lado, *AutomataMatrixCell* se utiliza para crear el diagrama del autómata. A diferencia de la anterior, cada celda de esta matriz contiene una lista enlazada de nodos de símbolos *(SymbolMatrixNode*) que representan los símbolos que se consumen, si es que los hay, para pasar de un estado a otro.

A su vez, para la correcta visualización de los diagramas de estado en Latex, se utilizó la librería *dot2text*[3] y *automata*[4], las cuales permiten graficar autómatas de forma sencilla.

Es importante aclarar que en el caso que el usuario no defina ningún autómata, *Generator* creará una salida en Latex vacía, con únicamente un mensaje indicando que no se definió ningún autómata[5].

5.3. Dificultades encontradas

Durante la realización de este trabajo práctico se encontraron una serie de inconvenientes, algunos técnicos y otros de organización interna del grupo.

En cuanto a los primeros, se destacan el manejo de los estados trampa y la minimización de autómatas los cuales terminaron quedando fuera de las características del compilador por cuestiones de tiempo e implementación de la estructuras previas al desarrollo del backend. Por otro lado, de manera más general, uno de los errores más comunes que se tuvieron fue acceder a posiciones de memoria inválidas como consecuencia del manejo de ciertas estructuras que desde el frontend se pasan para representar la necesidad de un procesamiento y que posteriormente pueden combinarse con otras operaciones que las manipulan. De esta forma, resultó necesario hacer un seguimiento exhaustivo para determinar en qué momento cada una de las expresiones son transformadas a elementos sueltos o conjuntos y con esto hacer un manejo correcto de las mismas.

[3] Ver Referencias, ítem 3.
[4] Ver Referencias, ítem 4.
[5] Ver apéndice III.

Respecto de las dificultades de organización, se puede encontrar que si bien se inició desarrollando la generación de código previa al manejo de símbolos por medio de una tabla, también se priorizó contar con una base lógica general escrita concretamente en código en vez de ir desarrollando de a partes. Esto terminó siendo poco eficiente en parte dado que algunas de las decisiones de implementación inicial no venían acompañadas de un testeo adecuado y por ende, se tuvo que destinar una cantidad de tiempo considerable hacia el final de la entrega en depurar el código, lo cual resultó tedioso y poco práctico.

6. Futuras extensiones

Se considera que el compilador fue desarrollado con la capacidad de adaptar nuevas funcionalidades y de esta forma escalar. Como próxima extensión a desarrollar, se podría permitir al usuario crear no sólo constantes como lo hace ahora, sino también variables de modo que un estado, transición, símbolo o autómata puedan ser modificados luego de ser declarados.

A su vez, se podrían incorporar bucles que permitan realizar iteraciones sobre conjuntos de datos, como estados, transiciones o símbolos, lo cuál sería útil a la hora de realizar tareas repetitivas. También se podrían agregar instrucciones condicionales al estilo if/ else, introduciendo de esta manera la capacidad de realizar evaluaciones condicionales.

Por último, se podría desarrollar una interfaz de usuario que permita a los usuarios interactuar directamente con el compilador, modificar variables, observar estados de autómatas y recibir feedback en tiempo real.

7. Conclusiones

En este informe se detalló el desarrollo de un compilador para autómatas finitos mediante un lenguaje de dominio específico (DSL) diseñado para definir y simular dichos autómatas. El objetivo principal fue generar representaciones en formato LaTeX que incluyen tanto diagramas de estados como tablas de transiciones a partir de archivos compilados en este lenguaje.

Durante el desarrollo, se enfrentaron desafíos técnicos significativos, como la gestión de memoria y la integración de múltiples estructuras de datos para representar de manera precisa las definiciones y operaciones sobre autómatas. A pesar de las dificultades encontradas, el equipo logró superar estos obstáculos y crear un compilador funcional capaz de generar documentos LaTeX a partir de las especificaciones dadas.

Sin lugar a dudas, se puede decir que este proyecto permitió poner en práctica los conceptos aprendidos en clase, no sólo en cuanto el desarrollo de un compilador, sino también en cuanto al aprendizaje teórico de autómatas finitos, al haber aplicado los conceptos teóricos para la correcta creación del compilador de autómatas.

8. Apéndice

Apéndice I: Ejemplo de entrada y salida de un autómata NFA

Entrada:

```
states Q: { r, o, j };
transitions R:  |s|-a->|{w,q}|;
alphabet A: { a,b };

NFA MiPrimerAutomata [
      states: { Q, >s , *w, *q },

      alphabet: A,

      transitions: {
            |q|-a->|r| ,
            |w|<-b->|r|,
            R
      }
];
```

Salida en código Latex:

```latex
\documentclass{article}
\usepackage{tikz}
\usepackage{caption}
\usepackage{geometry}
\usepackage{dot2texi}
\usetikzlibrary{automata}
\geometry{a4paper, margin=1in}

\begin{document}

\begin{figure} [h!]
    \section*{Automata 1: MiPrimerAutomata}
    \centering
    \begin{tikzpicture}
    \begin{dot2tex}[neato,mathmode]
        digraph G {
        edge [lblstyle="auto"];
        d2ttikzedgelabels = true;
        node [shape=circle];
        s [style="initial"];
        r -> w [label="b"];
        s -> w [label="a"];
        s -> q [label="a"];
        w -> r [label="b"];
        q -> r [label="a"];
        w [style="accepting"];
        q [style="accepting"];
        }
```

```
    \end{dot2tex}
    \end{tikzpicture}
    \caption{NFA automata}
    \label{fig:mi_grafo}
\end{figure}

\begin{table} [h!]
    \centering
    \begin{tabular}{|c|c|c|}
        \hline
        $\delta$ & a & b\\
        \hline
        r & & *w \\
        \hline
        o & &  \\
        \hline
        j & &  \\
        \hline
        $\rightarrow$s &*w, *q &  \\
        \hline
        *w & & r \\
        \hline
        *q &r &  \\
        \hline
    \end{tabular}
    \caption{Transitions Table}
    \label{tab:example}
\end{table}
\end{document}
```

Visualización en Latex:

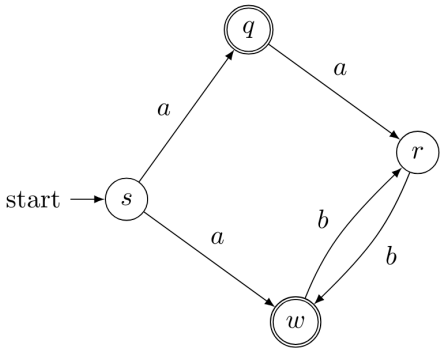

Figure 1: NFA automata

δ	a	b
r		*w
o		
j		
→s	*w, *q	
*w		r
*q	r	

Table 1: Transitions Table

Apéndice II: Ejemplo de entrada y salida de un autómata LNFA

Entrada:

```
states T: t;
transitions S1: {|w|-a->|j|, |r|-b->|r|, |j|-{a,b}->|w|};
transitions S2: {|j|-a->|w|};
transitions S3: S1 - S2;
states KO: {k, o, r, j};

LNFA AutomataLambda [
	states: { >s , *w, *q, (KO-T), bin },

	alphabet: {a, b, cc, dd},

	transitions: {
		|w|<-@->|r|,
		|s|-cc->|{w}|,
		|j|-dd->|{q}|,
		S3
	}
];
```

Salida en código Latex:

```
\documentclass{article}
\usepackage{tikz}
\usepackage{caption}
\usepackage{geometry}
\usepackage{dot2texi}
\usetikzlibrary{automata}
\geometry{a4paper, margin=1in}
\begin{document}
\begin{figure} [h!]
\section*{Automata 1: Automata3}
\centering
\begin{tikzpicture}
\begin{dot2tex}[neato,mathmode]
digraph G {
edge [lblstyle="auto"];
d2ttikzedgelabels = true;
node [shape=circle];
s [style="initial"];
s -> w [label="cc"];
w -> r [label="\lambda"];
w -> j [label="a"];
r -> w [label="\lambda"];
r -> r [label="b"];
j -> w [label="b"];
j -> q [label="dd"];
w [style="accepting"];
q [style="accepting"];
}
```

```
\end{dot2tex}
\end{tikzpicture}
\caption{LNFA automata}
\label{fig:mi_grafo}
\end{figure}
\begin{table} [h!]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
$\delta$ & a & b & cc & dd & $\lambda$\\
\hline
$\rightarrow$s &  &  & *w &  &  \\
\hline
*w & j &  &  &  & r \\
\hline
*q &  &  &  &  &  \\
\hline
k &  &  &  &  &  \\
\hline
o &  &  &  &  &  \\
\hline
r &  & r &  &  & *w \\
\hline
j &  & *w &  & *q &  \\
\hline
bin &  &  &  &  &  \\
\hline
\end{tabular}
\caption{Transitions Table}
\label{tab:example}
\end{table}
\end{document}
```

Visualización en Latex:

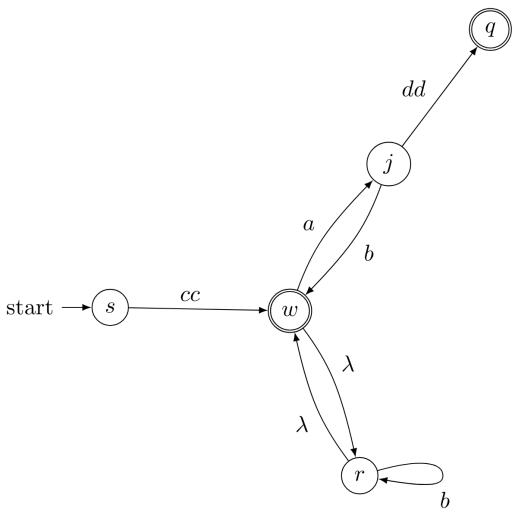

Figure 1: LNFA automata

δ	a	b	cc	dd	λ
$\rightarrow$s			*w		
*w	j				r
*q					
k					
o					
r		r			*w
j		*w		*q	
bin					

Table 1: Transitions Table

Apéndice III: Ejemplo de entrada y salida sin definir de un autómata

Entrada:

```
states Q: { *r, o, j };
transitions S1: {|r|-a->|j|, |r|-b->|r|, |j|-{a,b}->|r|};
alphabet MyAlphabet: { a,b };
```

Salida en código Latex:

```latex
\documentclass{article}
\usepackage{tikz}
\usepackage{caption}
\usepackage{geometry}
\usepackage{dot2texi}
\usetikzlibrary{automata}
\geometry{a4paper, margin=1in}
\begin{document}
\vspace*{\fill}
\begin{center}
\textit{No automata was defined}
\end{center}
\vspace*{\fill}
\end{document}
```

Visualización en Latex:

No automata was defined

9. Referencias

1. **Online Latex Editor:** https://es.overleaf.com/
2. **Libreria de hash map klib:** https://github.com/attractivechaos/klib/tree/master
3. **dot2tex - A Graphviz to LaTeX converter:** https://dot2tex.readthedocs.io/en/latest/
4. **The dot2texi package. Kjell Magne Fauske:**

   https://linorg.usp.br/CTAN/macros/latex/contrib/dot2texi/dot2texi.pdf

10. Bibliografía

1. **Clases teóricas y prácticas de Autómatas, Teoría de Lenguajes y Compiladores**