PRÁCTICA 1: EXPRESIONES REGULARES

PROCESADORES DEL LENGUAJE

DIEGO DÍAZ VIDAL
GRADO EN INGENIERÍA INFORMÁTICA
CURSO 2025-2026

Contenido

Introducción 3
Objetivos 3
Tecnologías utilizadas 3
Planteamiento 3
Expresiones regulares y casos de estudio 3
Caso (a): Identificador 4
Enunciado 4
ER propuesta 4
ER en JFLAP 4
Ejemplos 8
Caso (b): Número par de aes 8
Enunciado 8
ER propuesta 8
ER en JFLAP 9
Ejemplos 12
Caso (c): Números float/int 12
Enunciado 12
ER propuesta 12
ER en JFLAP 13
Ejemplos 17
Caso (d): Operaciones aritméticas 17
Enunciado 17
ER propuesta 17
ER en JFLAP 18
Ejemplos 19
Revisión del Código 20
Main.java 20
AFD.java 24
MaquinaEstados.java 28
Ejecución 30

Introducción

El presente documento tiene como finalidad describir, explicar y justificar el desarrollo de la Práctica 1 de la asignatura, relacionada con expresiones regulares y autómatas finitos.

Esta práctica incluye la implementación de un sistema en Java que evalúa cadenas de texto según expresiones regulares, así como los pasos necesarios para transformar dichas expresiones en autómatas finitos deterministas.

Objetivos

Los objetivos que se tratan de conseguir con la realización de esta práctica son los siguientes:

- Aprender a representar expresiones regulares en JFLAP.
- Transformar expresiones regulares en AFND y luego en AFD.
- Minimizar los AFD para optimizar la evaluación de cadenas.
- Implementar una máquina de estados en Java que permita comprobar cadenas de entrada.
- Generar y validar casos de prueba para cada expresión regular.

Tecnologías utilizadas

Las tecnologías y herramientas empleadas para la realización de la práctica son las siguientes:

- Java 22 para la implementación del sistema.
- JFLAP para la creación, transformación y análisis de autómatas.
- Visual Studio Code como entorno de desarrollo.

Planteamiento

El planteamiento se ha dividido en varias fases:

1. Definición de las expresiones regulares según los casos propuestos en el enunciado.
2. Representación en JFLAP y transformación a AFND y AFD.
3. Creación de la matriz de transición de estados para el AFD simplificado.
4. Implementación en Java de la máquina de estados.
5. Pruebas con cadenas de entrada y validación de resultados.
6. Preparación de un juego de pruebas y documentación de decisiones de diseño.

Expresiones regulares y casos de estudio

A continuación, se detalla cada caso solicitado, con su ER, explicación y ejemplos de cadenas.

Caso (a): Identificador

Enunciado

Identificador en un lenguaje de programación. Estos pueden estar formados por le tras y números, pero siempre empiezan por letra. (e.g. 'var', 'var2', 'COUNTER', etc.).

ER propuesta

La expresión regular propuesta es la siguiente:

```
[L][L|D]*
```

Donde `[L]` representa una letra (mayúscula o minúscula) y `[L|D]*` indica que después de la primera letra puede haber cero o más letras o dígitos.

Esta ER asegura que el identificador siempre comienza por una letra y puede contener cualquier combinación de letras y dígitos. A nivel de implementación, el mapper `letterDigitMapper` transforma cada letra en `L` y cada dígito en `D`, permitiendo que el AFD trabaje sobre esos tokens.

ER en JFLAP

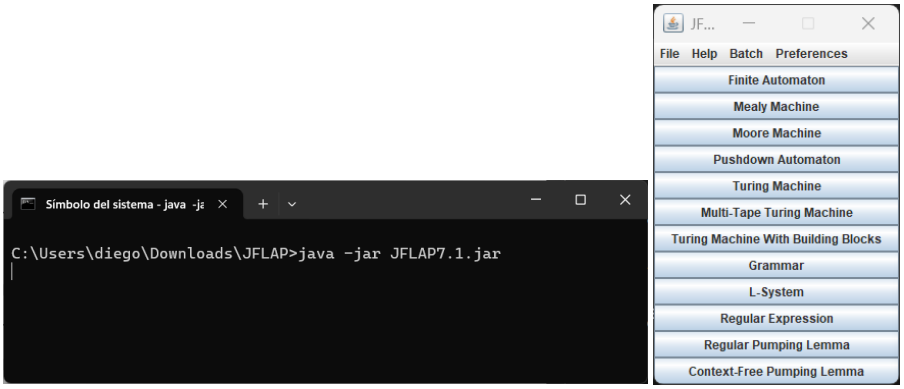
La traducción de la ER anterior a formato JFLAP es:

```
L(L+D)*
```

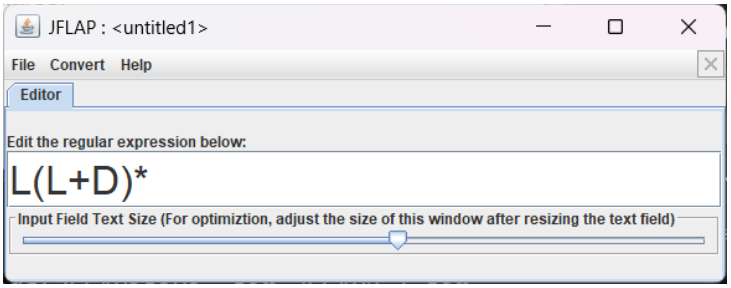
A continuación, se muestra el paso a paso de ER → AFND → AFD → AFDM.

Al ser este el primer caso, la explicación será más detallada. Para el resto de casos se excluirán detalles como "Se pulsa en el botón Export" o "Se ejecuta JFLAP".

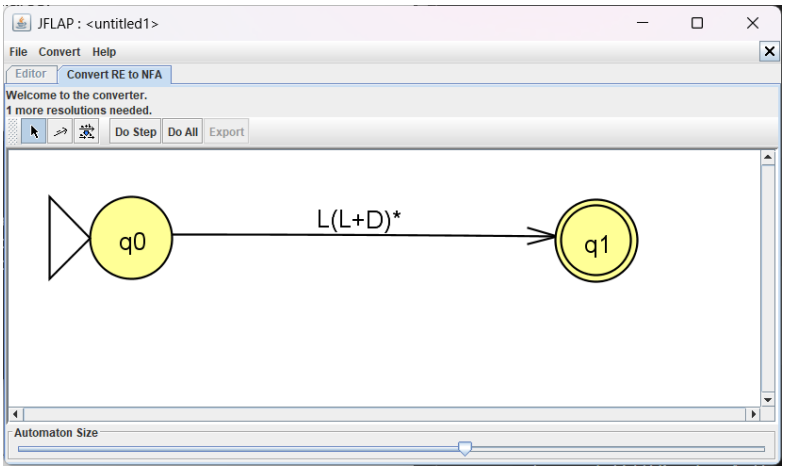
1. Ejecutar JFLAP.



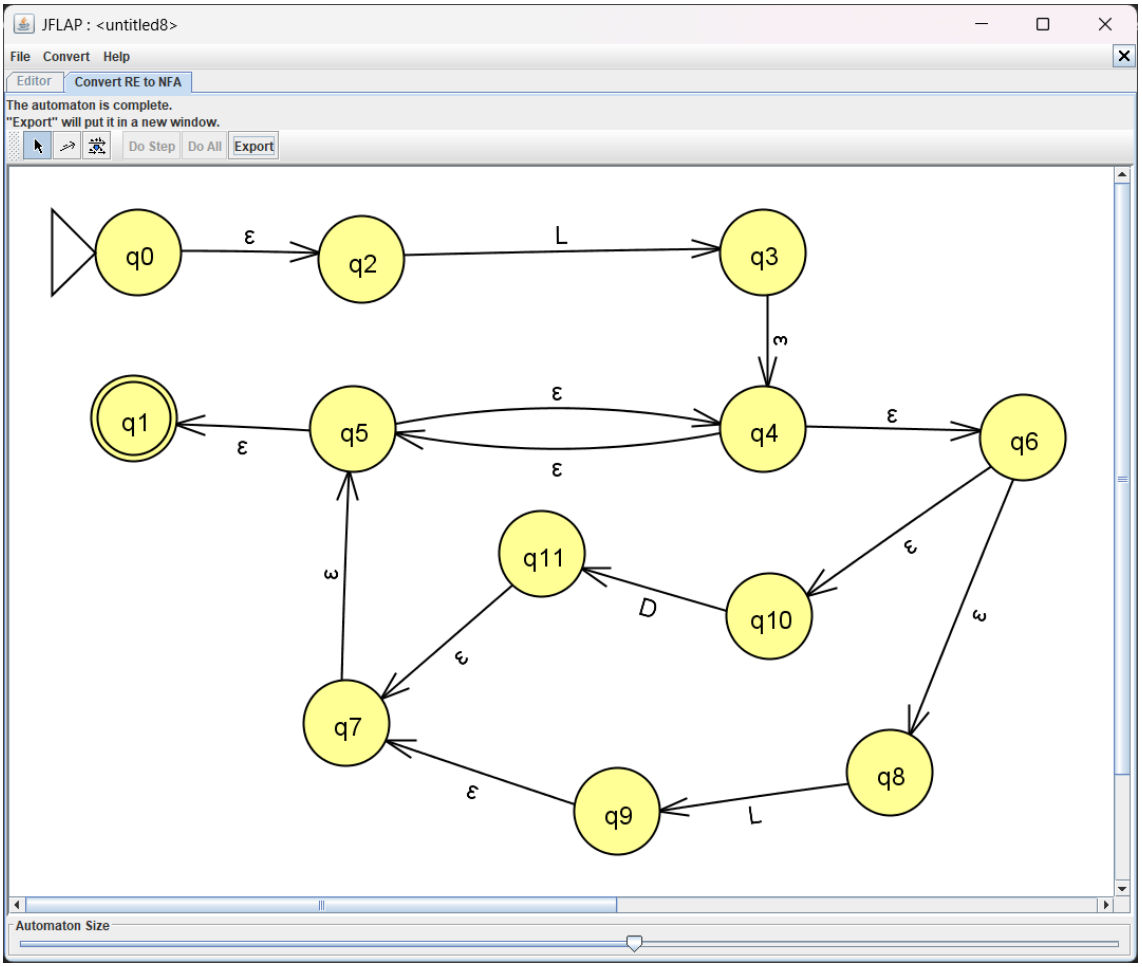
2. Introducir la ER (seleccionamos "Regular Expression" en el menú).



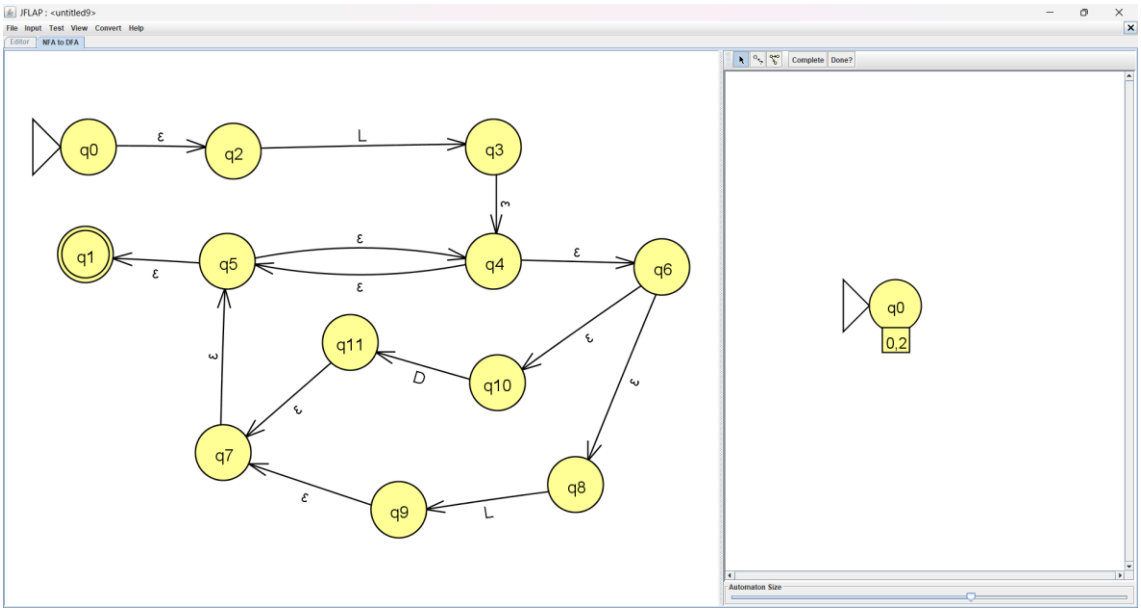
3. Creamos el AFND: Convert → Convert to NFA.



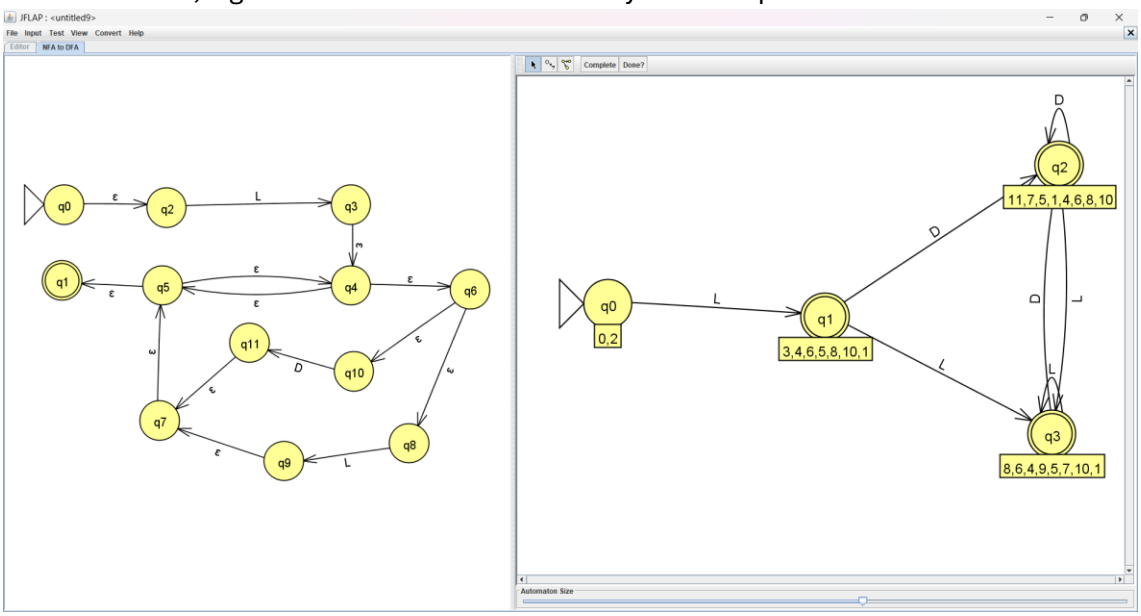
4. Desarrollamos el autómata separando la ER en sus elementos indivisibles: pulsamos en "Do Step" hasta haberlo reducido al máximo.



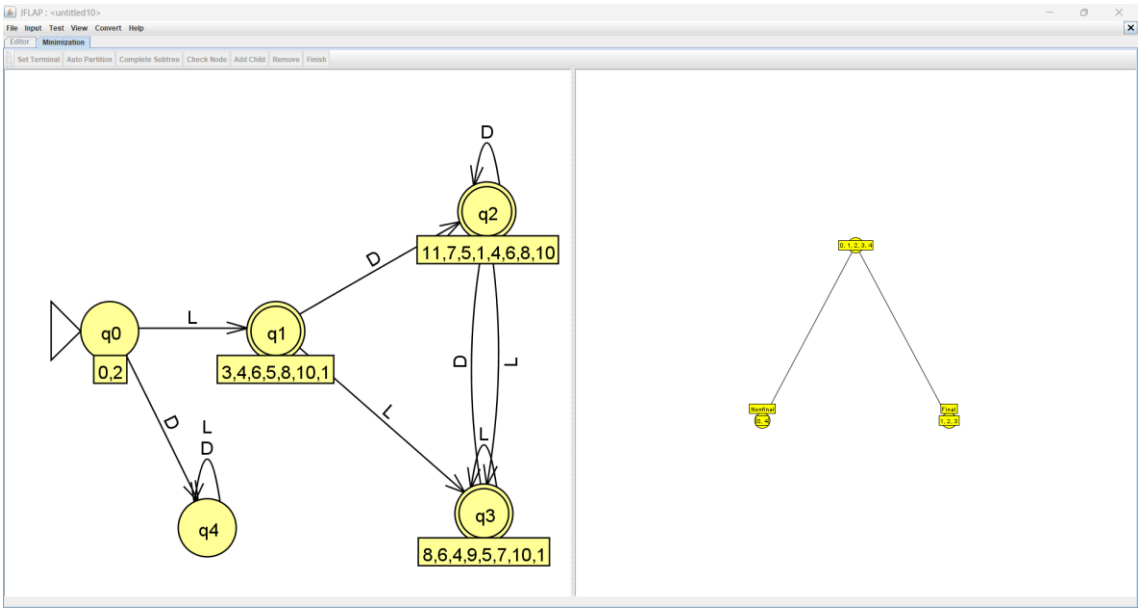
5. Una vez tenemos el AFND, lo transformamos a AFD: Export → Convert → Convert to DFA.
    a. Empezamos por el Cierre de Épsilon, el cual contiene únicamente los estados 0 y 2.



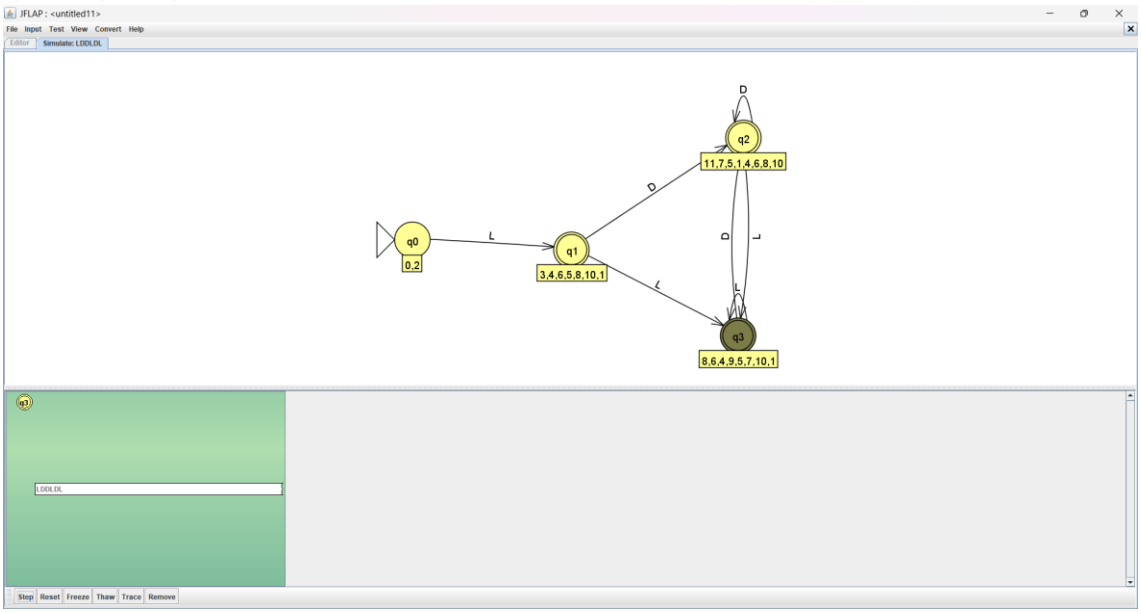
b. Pulsamos en “Complete” para completar el AFD. Este autómata es, evidentemente, significativamente más reducido y sencillo que el FND.



6. El siguiente paso es minimizar el AFD. Para ello, pulsamos en “Done” para llevarlo a una nueva ventana, y posteriormente en “Convert → Minimize DFA”.

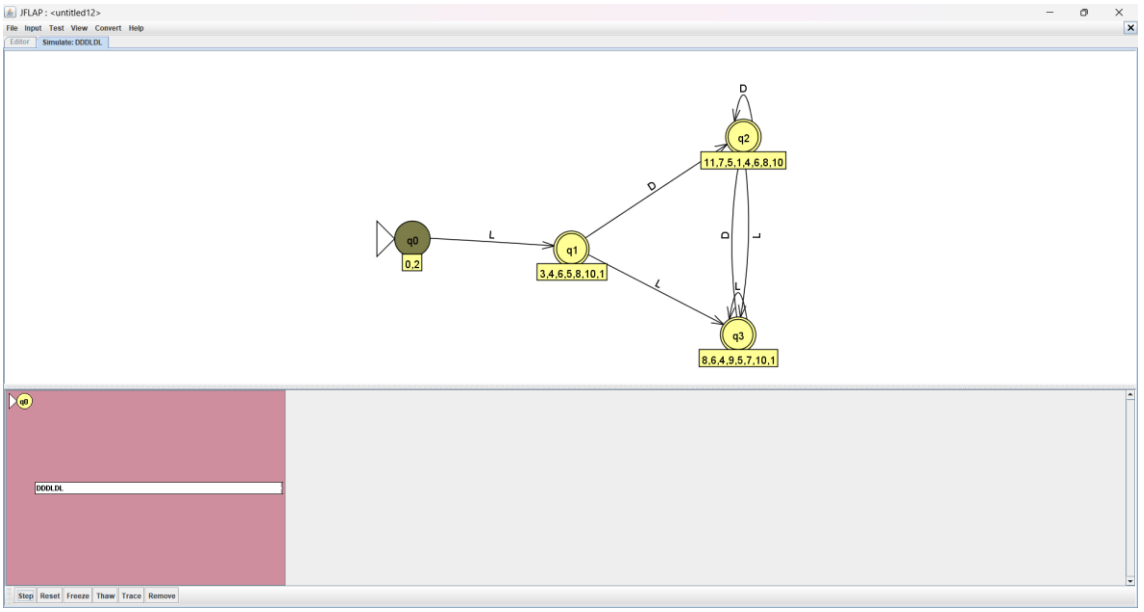


7. Finalmente, comprobamos el autómata introduciéndole las siguientes cadenas:
   a. "LDDLDL", la cual corresponde a un identificador con varias letras y dígitos que empieza por letra.



   Consumida la cadena acaba en un estado de aceptación → ACEPTADA.

   b. "DDDLDL", la cual corresponde a un identificador con varias letras y dígitos que empieza por un dígito.



Error al tratar de consumir el primer carácter: tiene que empezar por letra.

Ejemplos

A continuación se presentan los ejemplos de cadenas propuestas junto con los resultados obtenidos al evaluarlas:

Cadena	Resultado
Var	ACEPTADA
Var2	ACEPTADA
COUNTER	ACEPTADA
2var	RECHAZADA
v@r	RECHAZADA

Caso (b): Número par de aes

Enunciado

Cadenas con el alfabeto [a,b] formadas por un número par de a's. (e.g. 'baba').

ER propuesta

La expresión regular propuesta es la siguiente:

(b*ab*a)*

Esta expresión regular se traduce en: "acepto cero o más **b** antes de una **a**, y luego acepto una **a** seguida de una **b**, y finalmente una **a**". Todo ello puede aparecer cero o más veces (acepto la cadena vacía).

A nivel de implementación, el AFD mantiene dos estados:

- Estado 0: número par de **a** (estado final).
- Estado 1: número impar de **a** (estado no final).

Cada `a` alterna entre los estados, mientras que cada b mantiene el estado actual. Se utiliza `identityMapper` porque no se requiere transformar los caracteres.

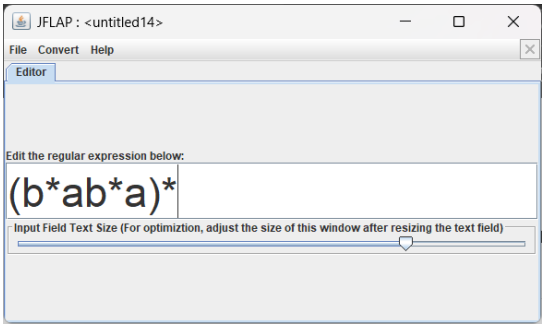
ER en JFLAP

La traducción de la ER anterior a formato JFLAP es:

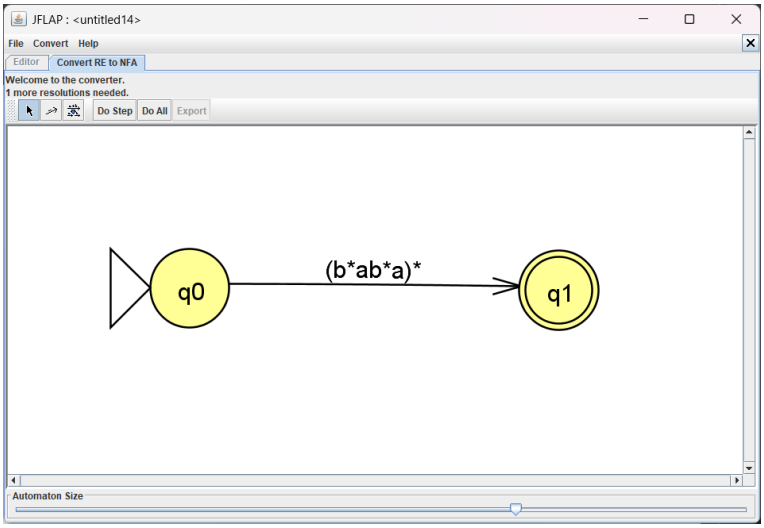
(b* a b* a)*

A continuación, se muestra el paso a paso de ER → AFND → AFD → AFDM.

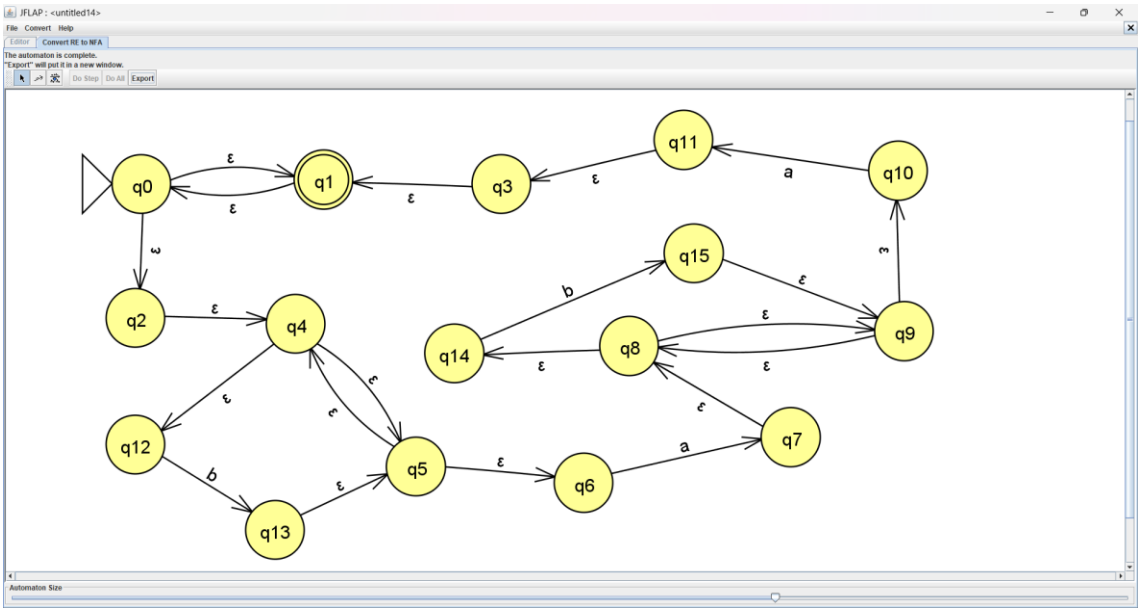
1. Se introduce la ER.



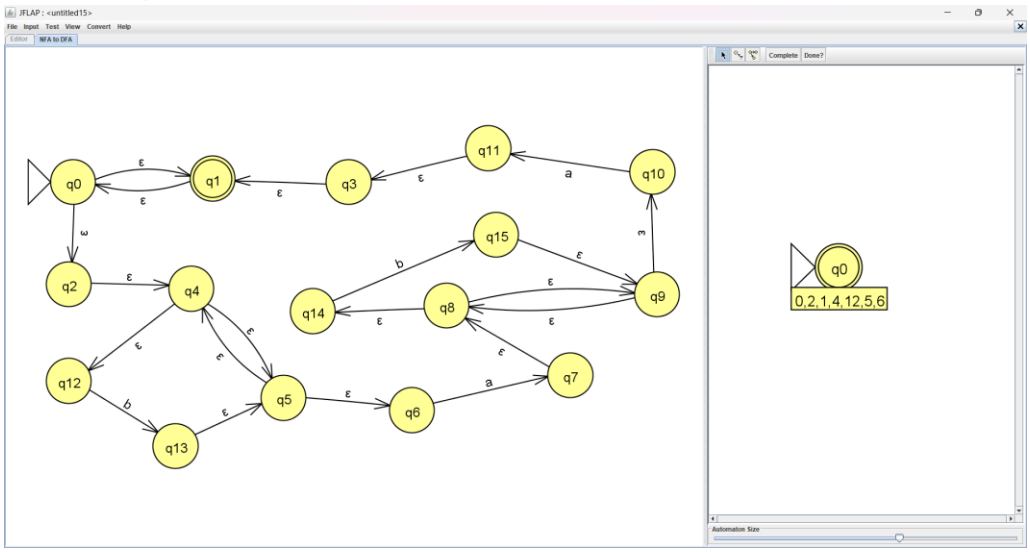
2. Se crea el AFND.



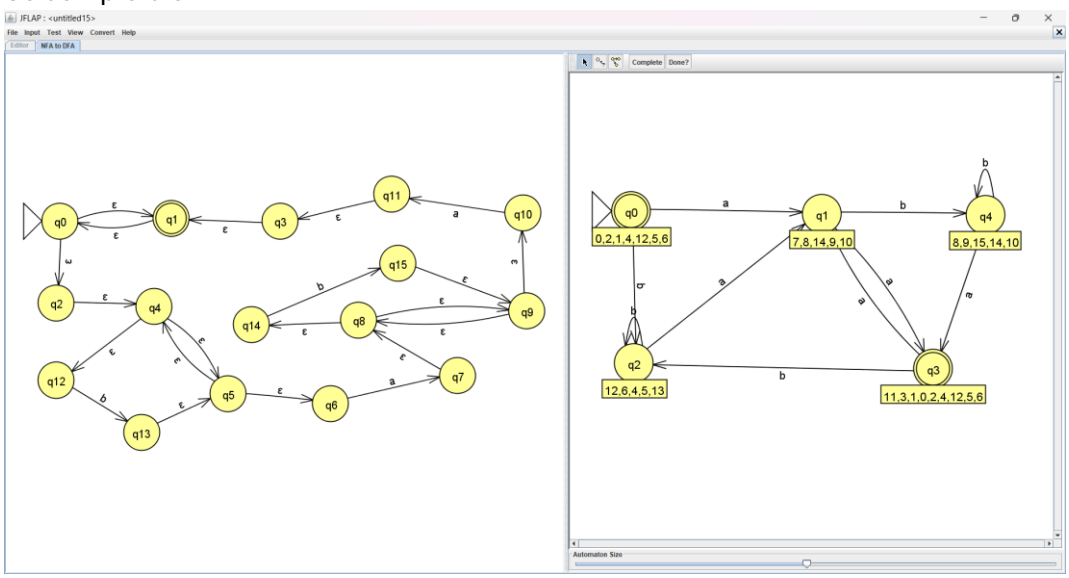
3. Se desarrolla el autómata separando la ER en sus elementos indivisibles.



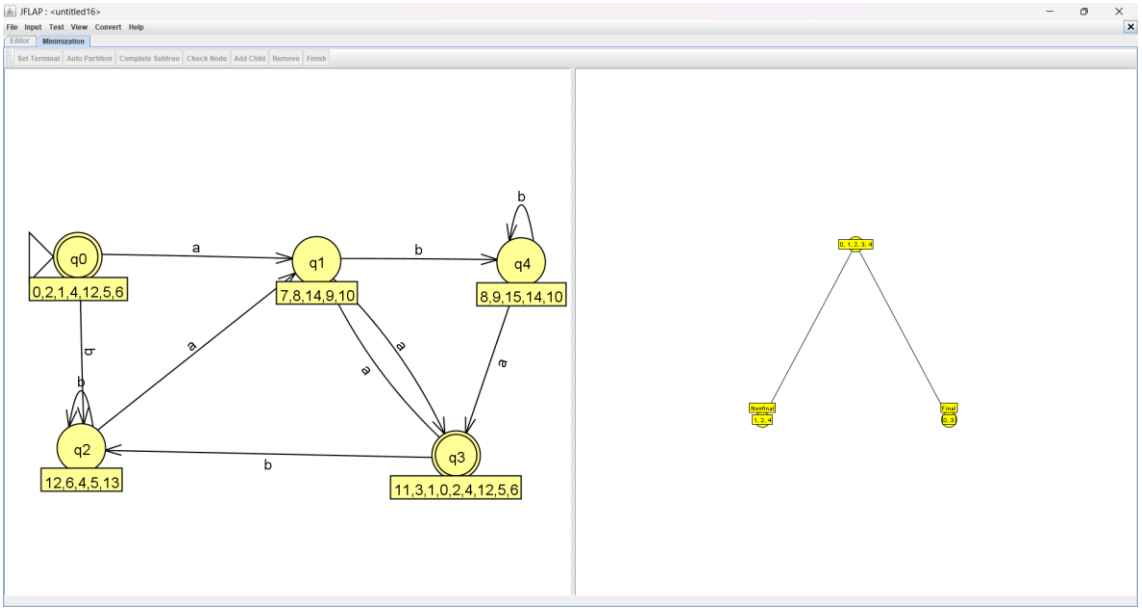
4. Se transforma el AFND en AFD.
   a. Cierre de Épsilon:



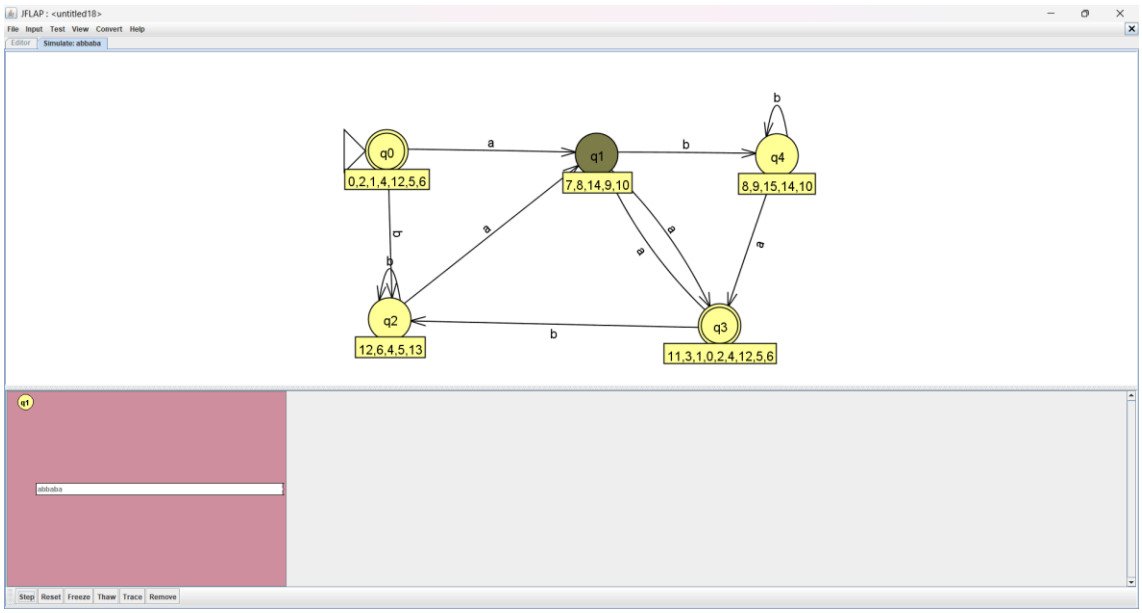
   b. Se completa el AFD.



5. Se minimiza el AFD.

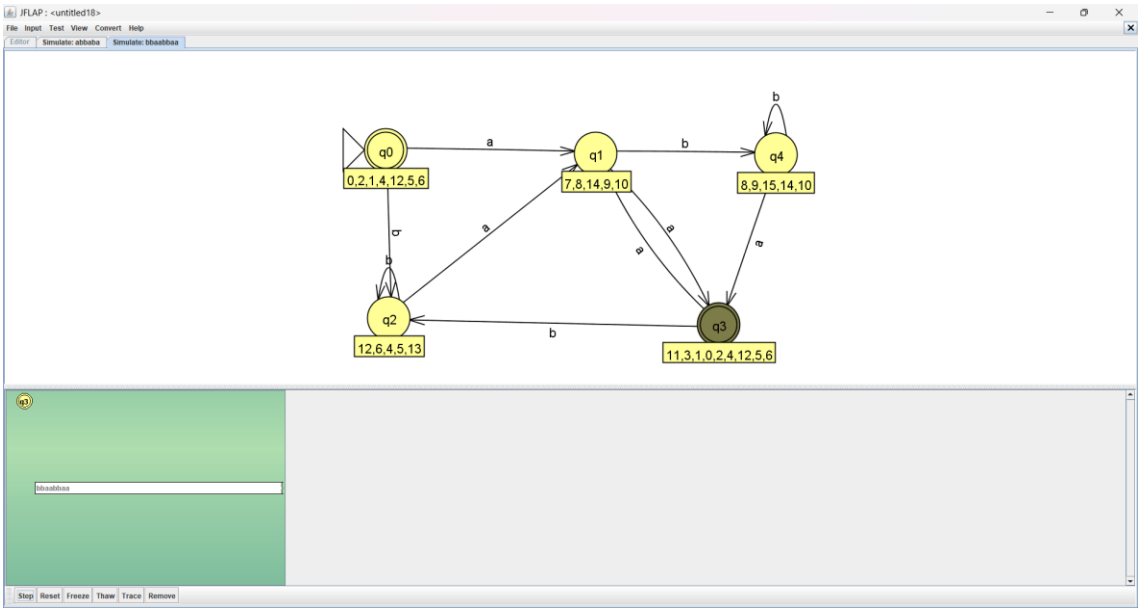


6. Se comprueba el AFD introduciendo las siguientes cadenas.
   a. “abbaba”: tiene 3 aes, debería ser rechazada.



   Consumida la cadena no acaba en un estado de aceptación → RECHAZADA.

   b. “bbaabbaa”: tiene 4 aes, debería ser aceptada.



Consumida la cadena, acaba en un estado de aceptación → ACEPTADA.

Ejemplos

A continuación se presentan los ejemplos de cadenas propuestas junto con los resultados obtenidos al evaluarlas:

Cadena	Resultado
“	ACEPTADA
baba	ACEPTADA
a	RECHAZADA
aa	ACEPTADA
ababa	RECHAZADA

Caso (c): Números float/int

Enunciado

Números en coma flotante (es decir, floats en cualquier lenguaje). (e.g. ’3.5’, ’33.02’, ’25’).

ER propuesta

La expresión regular propuesta es la siguiente:

```
D+(\.D+)?
```

Donde **D+** indica uno o más dígitos antes del punto decimal y **(\.D+)?** indica un punto seguido de uno o más dígitos. Se ha tomado la decisión de rechazar entradas que cumplan con el formato “punto digito”. Por ejemplo, “.5” será rechazada (aunque en algunos países esto se lee como “cero punto cinco”).

A nivel de implementación, el AFD tiene los siguientes estados:

- 0: inicio.
- 1: entero (aceptación).

- 2: tras punto, esperando dígito.
- 3: fracción (aceptación).

Se utiliza **floatMapper** para transformar los dígitos a **D** y el punto a **.**. El AFD acepta números enteros y decimales con, al menos, un dígito antes y después del punto.

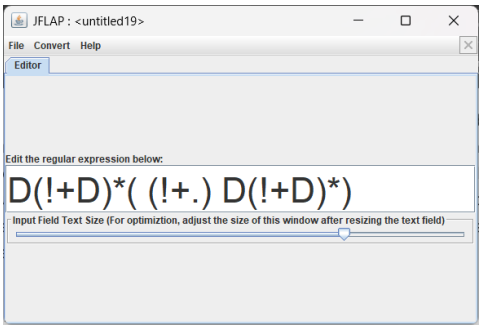
ER en JFLAP

La traducción de la ER anterior a formato JFLAP es:

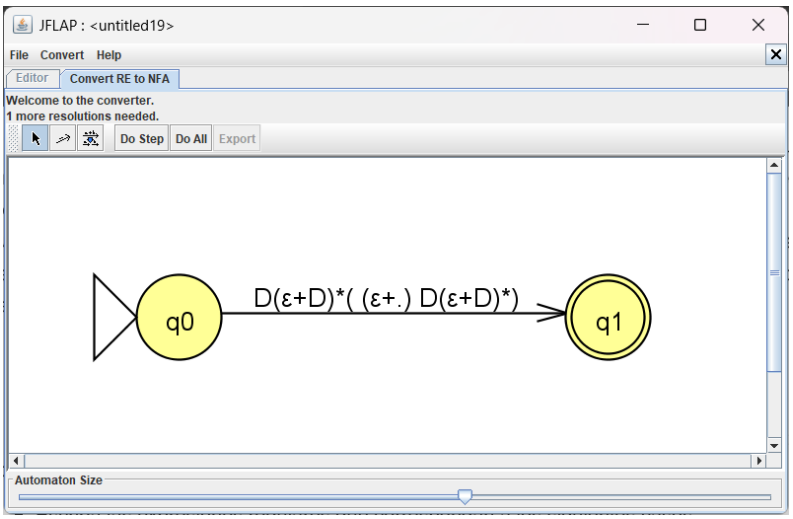
D(!+D)*((!+.)D(!+D)*)

A continuación, se muestra el paso a paso de ER → AFND → AFD → AFDM.

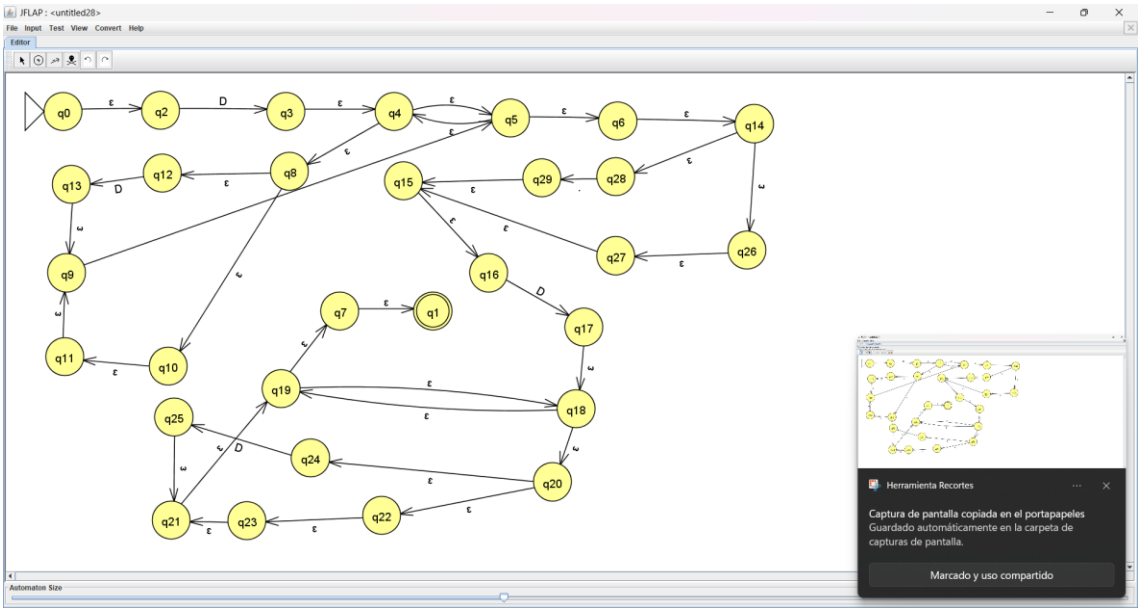
1. Se introduce la ER.



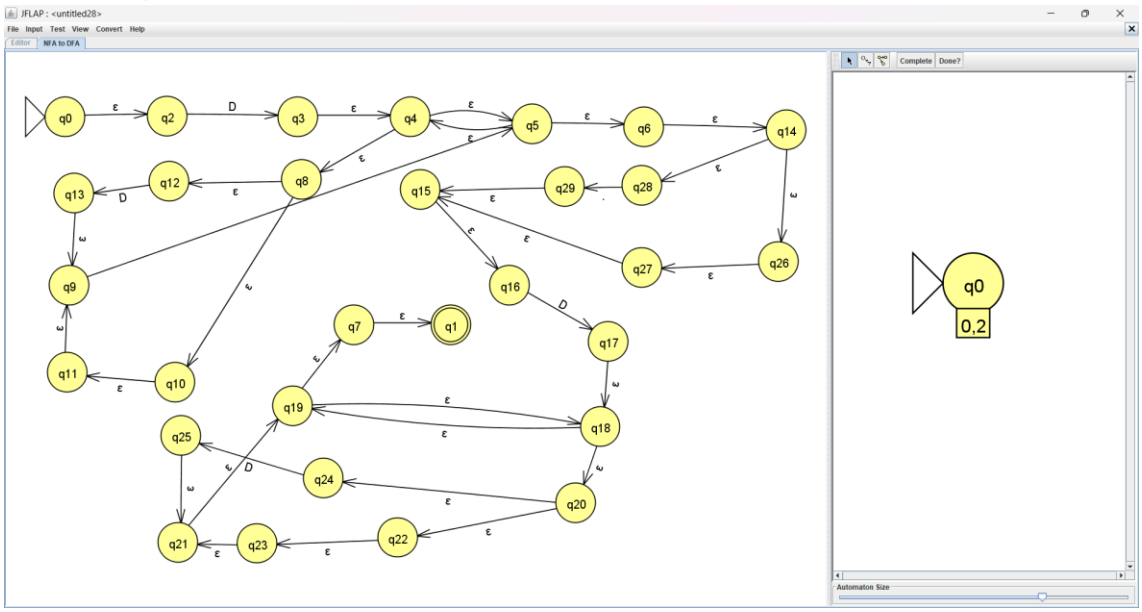
2. Se crea el AFND.



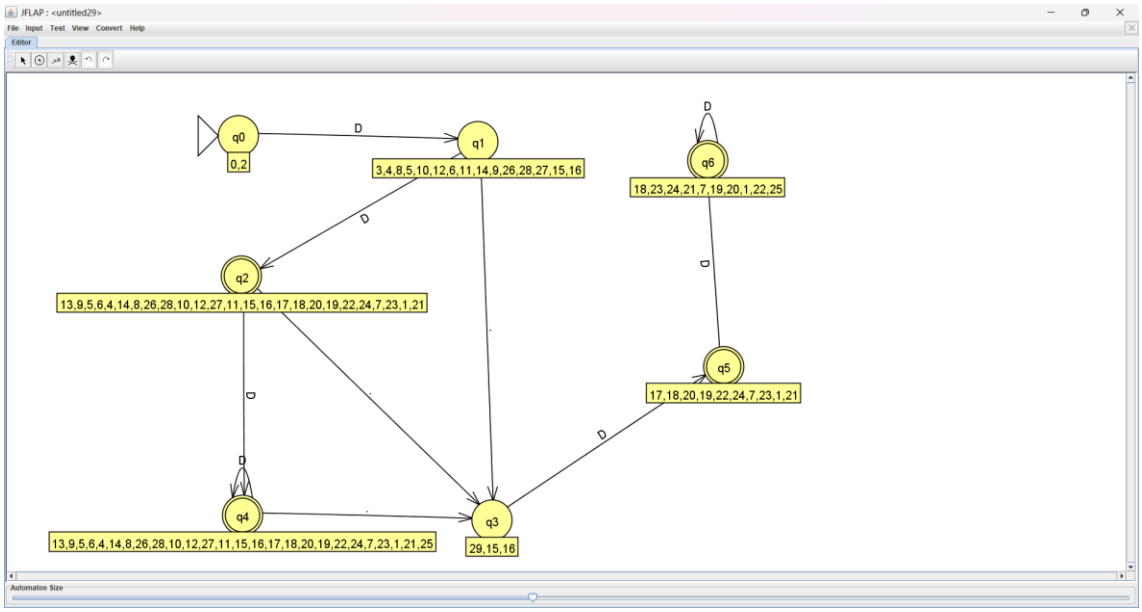
3. Se desarrolla el autómata separando la ER en sus elementos indivisibles.



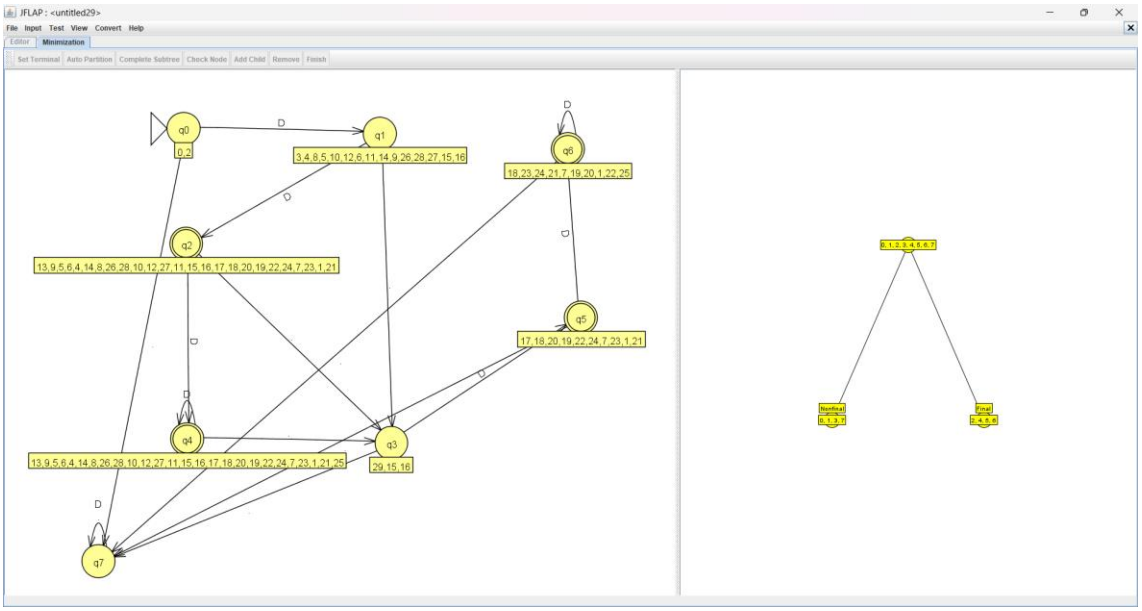
4. Se transforma el AFND en AFD.
    a. Cierre de Épsilon:



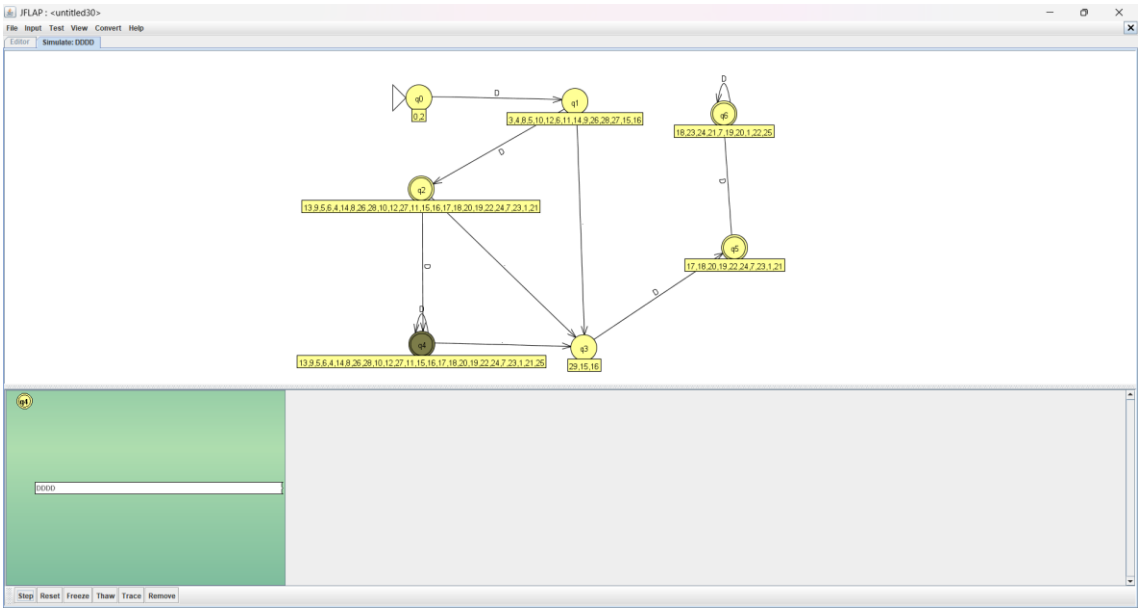
    b. Se completa el AFD.



5. Se minimiza el AFD.

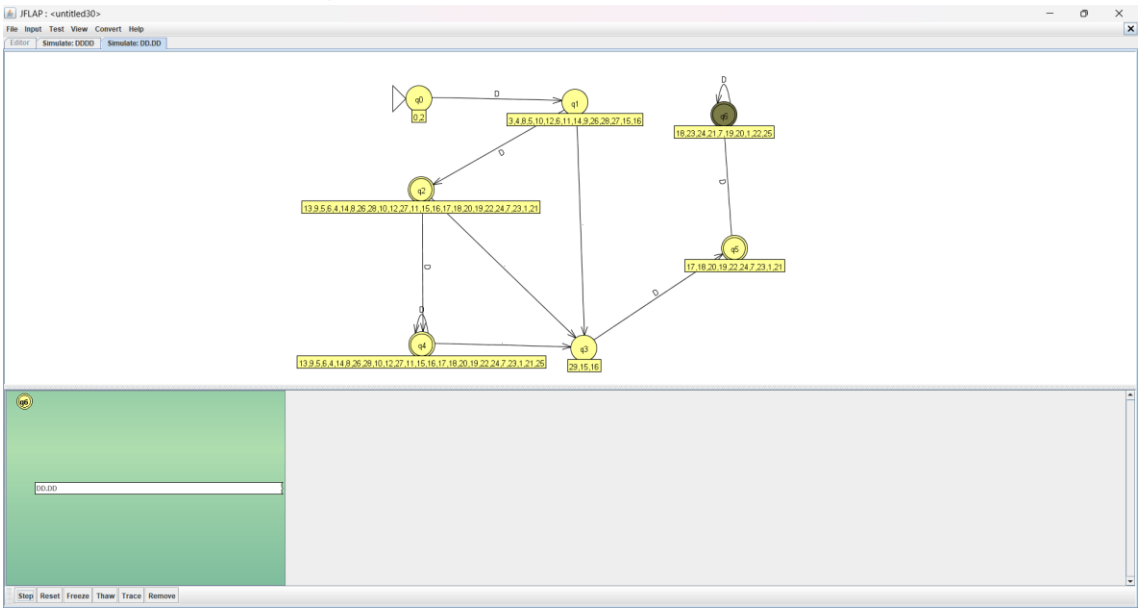


6. Se comprueba el AFD introduciendo las siguientes cadenas.
    a. “DDDD”: debería ser aceptada.



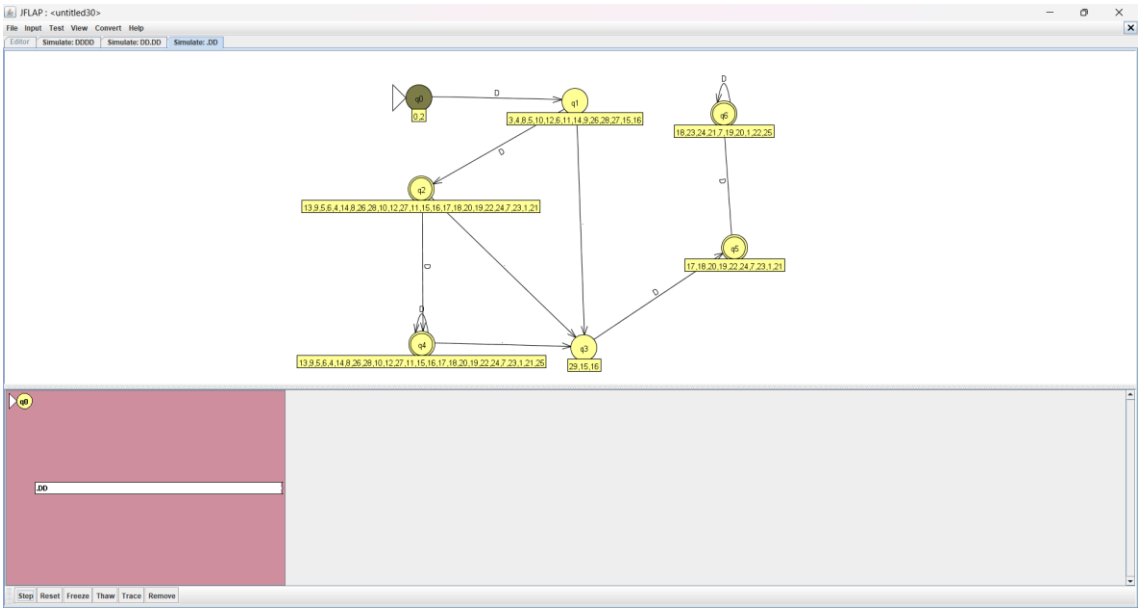
Consumida la cadena acaba en un estado de aceptación → ACEPTADA.

b. “DD.DD”: debería ser aceptada.



Consumida la cadena, acaba en un estado de aceptación → ACEPTADA.

c. “.DD”: debería ser rechazada.



Error al tratar de consumir el primer carácter.

Ejemplos

A continuación se presentan los ejemplos de cadenas propuestas junto con los resultados obtenidos al evaluarlas:

Cadena	Resultado
25	ACEPTADA
3.5	ACEPTADA
3.	RECHAZADA
.5	RECHAZADA
33.02	ACEPTADA
abc	RECHAZADA
12.0.3	RECHAZADA

Caso (d): Operaciones aritméticas

Enunciado

Operaciones aritméticas de suma y resta entre variables y/o floats. (e.g. '3- var + 2.5 + var2').

ER propuesta

La expresión regular propuesta es la siguiente:

```
(L|D+(\.D+)?)([+-](L|D+(\.D+)?))*
```

Por supuesto, esta no es la única implementación posible, pero es la que he realizado yo.

Los estados del AFD son:

- 0: inicio/esperando operando.
- 1: identificador.
- 2: número entero.

- 3: tras un punto.
- 4: fracción.
- 5: operador leído.

A nivel de implementación, se reutiliza `letterDigitMapper` para mapear letras y dígitos, mientras que `+` y `–` forman parte del alfabeto del AFD. Esto permite que la máquina evalúe cadenas complejas de operaciones.

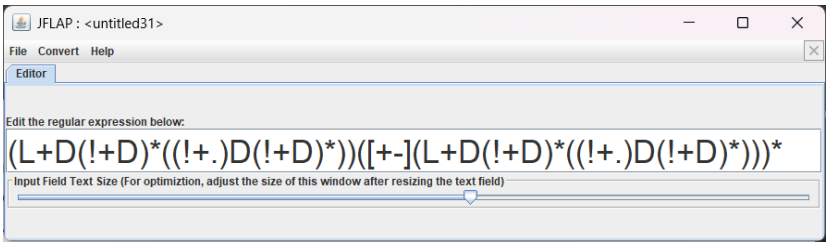
ER en JFLAP

La traducción de la ER anterior a formato JFLAP es:

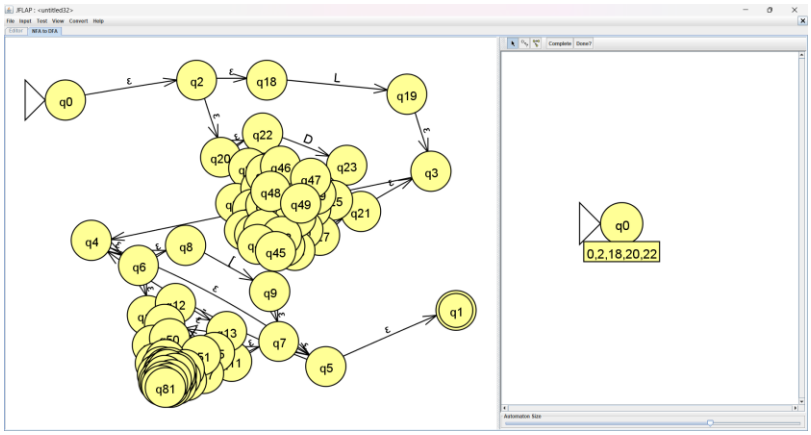
```
(L+D(!+D)*((!+.)D(!+D)*))([+-](L+D(!+D)*((!+.)D(!+D)*)))*
```

A continuación, se muestra el paso a paso de ER → AFND → AFD → AFDM.

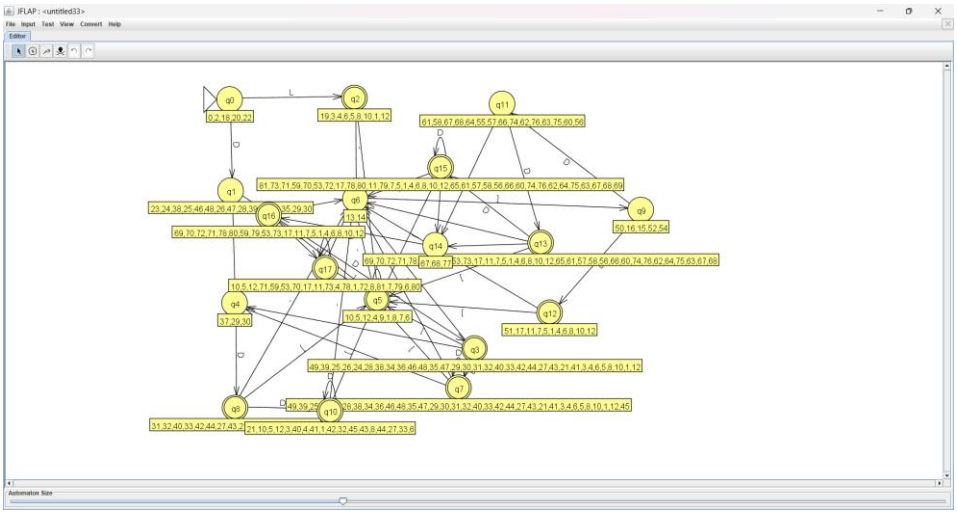
1. Se introduce la ER.



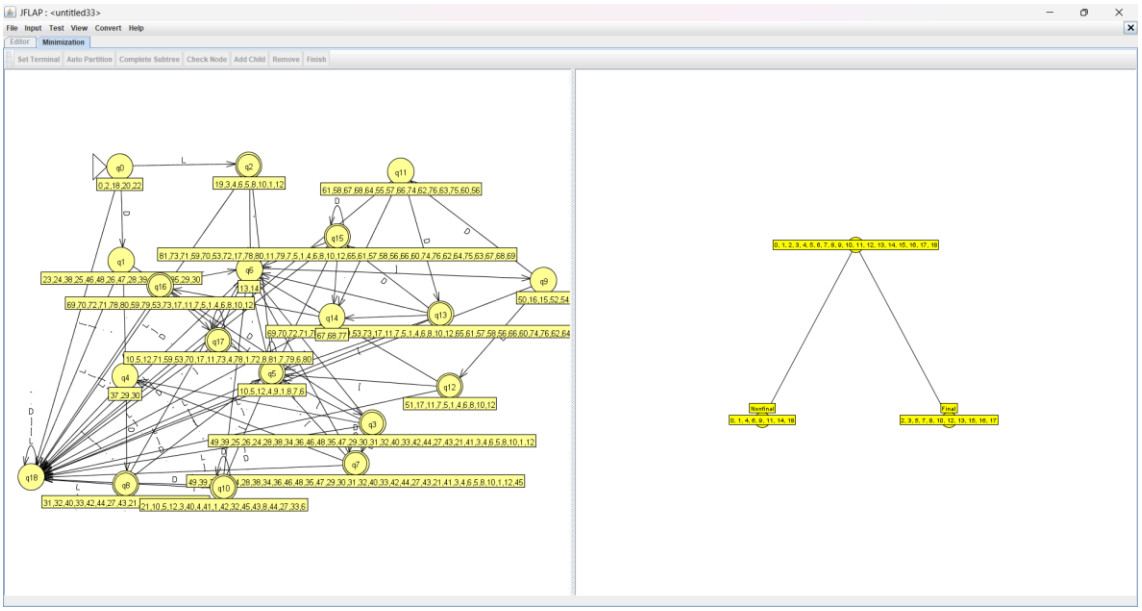
2. Se crea el AFND.
   El AFND tiene demasiados estados.
3. Se transforma el AFND en AFD.
   a. Cierre de Épsilon:



   b. Se completa el AFD.



4. Se minimiza el AFD.



Ejemplos

A continuación se presentan los ejemplos de cadenas propuestas junto con los resultados obtenidos al evaluarlas:

Cadena	Resultado
3-var+2.5+var2	ACEPTADA
var+3.5-2	ACEPTADA
2+2	ACEPTADA
3..5	RECHAZADA
+var	RECHAZADA
var+	RECHAZADA

Revisión del Código

A continuación se describen las clases que forman el código del programa y se explican las decisiones de diseño.

Main.java

Clase principal que actúa como punto de entrada y capa de configuración. Construye los autómatas concretos para los cuatro casos de estudio, ejecuta pruebas y ofrece un modo interactivo para experimentar.

Define una función constructora para cada caso, cada una devolviendo un autómata configurado según su especificación.

Usa mapeadores de caracteres según el contexto del autómata, centraliza las pruebas e incluya una pequeña interfaz de consola para realizar pruebas manuales.

```
// src/Main.java
import java.util.*;

public class Main {

    // Mapper helper: convierte caracteres reales a símbolos del AFD
    private static final java.util.function.Function<Character,
Character> identityMapper = c -> c;

    // Mapper que transforma letras -> 'L', dígitos -> 'D', resto
identity
    private static final java.util.function.Function<Character,
Character> letterDigitMapper = ch -> {
        if (Character.isLetter(ch)) return 'L';
        if (Character.isDigit(ch)) return 'D';
        return ch;
    };

    // Mapper que transforma dígitos -> 'D', '.' -> '.', resto identity
    private static final java.util.function.Function<Character,
Character> floatMapper = ch -> {
        if (Character.isDigit(ch)) return 'D';
        if (ch == '.') return '.';
        return ch;
    };

    /** Construye el AFD del caso (a): identificador (token 'L' y 'D').
*/
    private static AFD construirIdentificador() {
        AFD afd = new AFD();
        afd.cargarAlfabeto('L'); afd.cargarAlfabeto('D');
afd.cargarAlfabeto('O');
```

```
        afd.addEstadosConsecutivos(0,1);
        afd.inicializarMatriz();
        afd.establecerQi(0);
        afd.establecerQf(1);

        afd.cargarTransicion(0, 'L', 1);
        afd.cargarTransicion(1, 'L', 1);
        afd.cargarTransicion(1, 'D', 1);

        return afd;
    }

    /** Construye el AFD del caso (b): par de 'a' en {a,b}. */
    private static AFD construirParA() {
        AFD afd = new AFD();
        afd.cargarAlfabeto('a'); afd.cargarAlfabeto('b');

        afd.addEstadosConsecutivos(0,1);
        afd.inicializarMatriz();
        afd.establecerQi(0);
        afd.establecerQf(0);

        afd.cargarTransicion(0, 'a', 1);
        afd.cargarTransicion(0, 'b', 0);
        afd.cargarTransicion(1, 'a', 0);
        afd.cargarTransicion(1, 'b', 1);

        return afd;
    }

    /** Construye el AFD del caso (c): float/int con tokens 'D' y '.'
*/
    private static AFD construirFloat() {
        AFD afd = new AFD();
        afd.cargarAlfabeto('D'); afd.cargarAlfabeto('.');
afd.cargarAlfabeto('O');

        afd.addEstadosConsecutivos(0,3);
        afd.inicializarMatriz();
        afd.establecerQi(0);
        afd.establecerQf(1);
        afd.establecerQf(3);

        afd.cargarTransicion(0, 'D', 1);
        afd.cargarTransicion(1, 'D', 1);
        afd.cargarTransicion(1, '.', 2);
        afd.cargarTransicion(2, 'D', 3);
        afd.cargarTransicion(3, 'D', 3);
```

```
        return afd;
    }

    /** Construye el AFD del caso (d): operaciones aritméticas entre
variables y floats */
    private static AFD construirOperaciones() {
        AFD afd = new AFD();
        afd.cargarAlfabeto('L'); afd.cargarAlfabeto('D');
afd.cargarAlfabeto('.');
        afd.cargarAlfabeto('+'); afd.cargarAlfabeto('-');
afd.cargarAlfabeto('O');

        afd.addEstadosConsecutivos(0,5);
        afd.inicializarMatriz();
        afd.establecerQi(0);
        afd.establecerQf(1);
        afd.establecerQf(2);
        afd.establecerQf(4);

        // Inicio -> identificador o número
        afd.cargarTransicion(0, 'L', 1);
        afd.cargarTransicion(0, 'D', 2);

        // Identificador -> identificador o operador
        afd.cargarTransicion(1, 'L', 1);
        afd.cargarTransicion(1, 'D', 1);
        afd.cargarTransicion(1, '+', 5);
        afd.cargarTransicion(1, '-', 5);

        // Entero -> entero, operador o punto
        afd.cargarTransicion(2, 'D', 2);
        afd.cargarTransicion(2, '+', 5);
        afd.cargarTransicion(2, '-', 5);
        afd.cargarTransicion(2, '.', 3);

        // Punto -> dígito (fracción)
        afd.cargarTransicion(3, 'D', 4);

        // Fracción -> dígito o operador
        afd.cargarTransicion(4, 'D', 4);
        afd.cargarTransicion(4, '+', 5);
        afd.cargarTransicion(4, '-', 5);

        // Operador -> inicio de nuevo operando
        afd.cargarTransicion(5, 'L', 1);
        afd.cargarTransicion(5, 'D', 2);

        return afd;
    }
```

```
    /** Demostración de un AFD con un conjunto de pruebas. */
    private static void demoAFD(AFD afd,
java.util.function.Function<Character, Character> mapper, String[]
pruebas) {
        MaquinaEstados maquina = new MaquinaEstados(afd, mapper);
        System.out.println("=== Probando AFD ===");
        System.out.println(afd);
        for (String s : pruebas) {
            boolean ok = maquina.compruebaCadena(s);
            System.out.printf("Input: %-20s -> %s\n", "\"" + s + "\"",
ok ? "ACEPTADA" : "RECHAZADA");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        AFD afdIdent = construirIdentificador();
        AFD afdParA = construirParA();
        AFD afdFloat = construirFloat();
        AFD afdOper = construirOperaciones();

        String[] pruebasIdent = {"var", "var2", "COUNTER", "2var",
"v@r"};
        String[] pruebasParA = {"", "baba", "a", "aa", "ababa"};
        String[] pruebasFloat = {"25", "3.5", "3.", ".5", "33.02",
"abc", "12.0.3"};
        String[] pruebasOper = {"3-var+2.5+var2", "var+3.5-2", "2+2",
"3..5", "+var", "var+"};

        System.out.println("=== CASO (a) Identificador ===");
        demoAFD(afdIdent, letterDigitMapper, pruebasIdent);

        System.out.println("=== CASO (b) Par de 'a' ===");
        demoAFD(afdParA, identityMapper, pruebasParA);

        System.out.println("=== CASO (c) Float/Int ===");
        demoAFD(afdFloat, floatMapper, pruebasFloat);

        System.out.println("=== CASO (d) Operaciones aritméticas ===");
        demoAFD(afdOper, letterDigitMapper, pruebasOper);

        // Interactivo
        try (Scanner sc = new Scanner(System.in)) {
            System.out.println("Prueba interactiva rápida. Pulsa ENTER
para salir.");
            AFD current = afdIdent;
            java.util.function.Function<Character, Character>
currentMapper = letterDigitMapper;
```

```
            MaquinaEstados machine = new MaquinaEstados(current,
currentMapper);
            while (true) {
                System.out.println("\nElige AFD: 1=identificador,
2=parA, 3=float, 4=operaciones, ENTER=shell exit");
                String sel = sc.nextLine().trim();
                if (sel.isEmpty()) break;
                if (sel.equals("1")) { current = afdIdent;
currentMapper = letterDigitMapper; }
                else if (sel.equals("2")) { current = afdParA;
currentMapper = identityMapper; }
                else if (sel.equals("3")) { current = afdFloat;
currentMapper = floatMapper; }
                else if (sel.equals("4")) { current = afdOper;
currentMapper = letterDigitMapper; }
                else { System.out.println("Opción inválida"); continue;
}

                machine = new MaquinaEstados(current, currentMapper);
                System.out.print("Introduce cadena a comprobar (ENTER
para volver al menú): ");
                String s = sc.nextLine();
                if (s.isEmpty()) continue;
                boolean accept = machine.compruebaCadena(s);
                System.out.println(accept ? "ACEPTADA" : "RECHAZADA");
            }
        }

        System.out.println("Demo finalizada.");
    }
}
```

AFD.java

Esta clase representa la estructura de un autómata finito determinista genérico, independiente de cualquier caso particular. En cuanto al diseño, mantiene colecciones internas para el alfabeto, estados, estado inicial, estados finales y una matriz de transición representada como **`Map<Integer, Map<Character, Integer>>`**.

Usa estructuras dinámicas para una mayor flexibilidad al definir o modificar autómatas e incluye métodos para cargar símbolos y estados, inicializar la matriz de transiciones, definir las relaciones entre estados y consultar transiciones.

La representación textual (**`toString()`**) permite inspeccionar el autómata fácilmente para depuración.

```
// src/AFD.java
```

```java
import java.util.*;

public class AFD {
    private final List<Character> alfabeto; // lista de símbolos del
alfabeto
    private final List<Integer> estados;    // lista de estados (ints)
    private int estadoInicial;
    private final Set<Integer> estadosFinales;
    private final Map<Integer, Map<Character, Integer>> matriz;     //
la "matriz" dinámica

    public AFD() {
        this.alfabeto = new ArrayList<>();
        this.estados = new ArrayList<>();
        this.estadoInicial = 0;
        this.estadosFinales = new HashSet<>();
        this.matriz = new HashMap<>();
    }

    // ----- Métodos para cargar estructura -----

    /** Añade un símbolo al alfabeto (si no existe). */
    public void cargarAlfabeto(Character c) {
        if (!alfabeto.contains(c)) alfabeto.add(c);
    }

    /** Añade una lista de símbolos al alfabeto. */
    public void cargarAlfabeto(Collection<Character> cs) {
        for (Character c : cs) cargarAlfabeto(c);
    }

    /** Añade un estado (por ejemplo 0..n-1). */
    public void addEstado(int q) {
        if (!estados.contains(q)) {
            estados.add(q);
        }
    }

    /** Añade varios estados consecutivos del from (inclusive) al to
(inclusive). */
    public void addEstadosConsecutivos(int from, int to) {
        for (int q = from; q <= to; q++) addEstado(q);
    }

    /** Establece el estado inicial. */
    public void establecerQi(int q) {
        if (!estados.contains(q)) addEstado(q);
        this.estadoInicial = q;
    }
```

```
    /** Marca un estado como final. */
    public void establecerQf(int q) {
        if (!estados.contains(q)) addEstado(q);
        estadosFinales.add(q);
    }

    /** Elimina un estado final (si existía). */
    public void quitarQf(int q) {
        estadosFinales.remove(q);
    }

    // ----- Inicialización y carga de la "matriz" -----

    /**
     * Inicializa la matriz creando por cada estado una fila (mapa
vacío).
     * Se debe llamar a este método después de crear estados y alfabeto
(o cada vez
     * que se añadan estados nuevos).
     */
    public void inicializarMatriz() {
        for (Integer q : estados) {
            if (!matriz.containsKey(q)) {
                matriz.put(q, new HashMap<>()); // fila vacía
            }
        }
    }

    /**
     * Rellena la transición desde estado 'origen' con el símbolo
'simbolo' hacia 'destino'.
     * Si la fila no existe, la crea (por eso es robusto).
     */
    public void cargarTransicion(int origen, Character simbolo, Integer
destino) {
        if (!estados.contains(origen)) addEstado(origen);
        if (!estados.contains(destino)) addEstado(destino);
        cargarAlfabeto(simbolo);
        matriz.computeIfAbsent(origen, k -> new HashMap<>())
              .put(simbolo, destino);
    }

    /**
     * Obtiene la transición: para estadoActual y caracter devuelve el
estado destino
     * o null si no existe.
     */
```

```
    public Integer getSiguienteEstado(int estadoActual, Character
caracter) {
        Map<Character, Integer> fila = matriz.get(estadoActual);
        if (fila == null) return null;
        return fila.get(caracter); // puede ser null si no existe
transición
    }

    // ----- Consultas -----

    public boolean esFinal(int estado) {
        return estadosFinales.contains(estado);
    }

    public int getEstadoInicial() {
        return estadoInicial;
    }

    public List<Character> getAlfabeto() {
        return Collections.unmodifiableList(alfabeto);
    }

    public List<Integer> getEstados() {
        return Collections.unmodifiableList(estados);
    }

    public Set<Integer> getEstadosFinales() {
        return Collections.unmodifiableSet(estadosFinales);
    }

    public Map<Integer, Map<Character, Integer>> getMatriz() {
        return Collections.unmodifiableMap(matriz);
    }

    // ----- Representación para depuración -----
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("AFD:\n");
        sb.append("Alfabeto: ").append(alfabeto).append("\n");
        sb.append("Estados: ").append(estados).append("\n");
        sb.append("EstadoInicial:
").append(estadoInicial).append("\n");
        sb.append("EstadosFinales:
").append(estadosFinales).append("\n");
        sb.append("Matriz(transiciones):\n");
        for (Integer q : estados) {
            sb.append(" q").append(q).append(" -> ");
            Map<Character, Integer> fila = matriz.get(q);
```

```
            if (fila == null) { sb.append("{}\n"); continue; }
            sb.append(fila.toString()).append("\n");
        }
        return sb.toString();
    }
}
```

MaquinaEstados.java

Este script encapsula la lógica de ejecución o simulación del autómata, controlando el estado actual durante la lectura de una cadena.

Recibe un autómata (AFD) y una función mapper que transforma los caracteres reales en símbolos del alfabeto del AFD (por ejemplo, letras en “L” o dígitos en “D”). Este patrón permite reutilizar un mismo AFD con distintos alfabetos o entradas.

Contiene métodos que permite procesar símbolos y actualizar estados, evaluar cadenas completas o consultar si el estado actual es de aceptación o final.

```
// src/MaquinaEstados.java
import java.util.function.Function;

public class MaquinaEstados {
    private final AFD afd;
    private int estadoActual;
    private final Function<Character, Character> mapper; // transforma
input -> símbolo del AFD

    /** Excepción lanzada cuando no existe transición. */
    public static class TransicionNoDefinidaException extends Exception
{
        public TransicionNoDefinidaException(String msg) { super(msg);
}
    }

    /**
     * Constructor.
     * @param afd El autómata (matriz, alfabeto, etc.)
     * @param mapper Función que mapea el carácter real a un símbolo
del alfabeto del AFD.
     */
    public MaquinaEstados(AFD afd, Function<Character, Character>
mapper) {
        this.afd = afd;
        this.mapper = mapper != null ? mapper : (c -> c);
        this.estadoActual = afd.getEstadoInicial();
    }
```

```
    /** Inicializa la máquina (estadoActual := estado inicial del AFD).
*/
    public void inicializar() {
        this.estadoActual = afd.getEstadoInicial();
    }

    /**
     * Consume un carácter: aplica el mapper para obtener el símbolo
del AFD
     * y actualiza el estadoActual, o lanza
TransicionNoDefinidaException si no existe.
     */
    public void acepta(char caracter) throws
TransicionNoDefinidaException {
        char simbolo = mapper.apply(caracter);
        Integer siguiente = afd.getSiguienteEstado(estadoActual,
simbolo);
        if (siguiente == null) {
            throw new TransicionNoDefinidaException(
                "No hay transición desde el estado " + estadoActual + "
con símbolo '" + simbolo + "' (entrada '" + caracter + "')");
        }
        this.estadoActual = siguiente;
    }

    /** Devuelve true si el estado actual es final. */
    public boolean esFinal() {
        return afd.esFinal(estadoActual);
    }

    /**
     * Recorre la cadena y devuelve true si la AFD acepta la cadena
completa.
     * Si en cualquier momento no existe transición para un símbolo,
devuelve false.
     *
     */
    public boolean compruebaCadena(String input) {
        inicializar();
        for (int i = 0; i < input.length(); i++) {
            char c = input.charAt(i);
            try {
                acepta(c);
            } catch (TransicionNoDefinidaException ex) {
                // transición no definida => rechazo
                return false;
            }
        }
```

```
        return esFinal();
    }

    /** Devuelve el estado actual (útil para debugging). */
    public int getEstadoActual() {
        return estadoActual;
    }
}
```

Ejecución

A continuación se muestra el resultado de la ejecución del programa: