Práctica 1 Compiladores: Expresiones Regulares

Martín Maximiliano Mora

Adrián Anchuela Villoldo

Alfabeto Utilizado:

`W, X, Y, Z.`

1. **Expresión regular:**

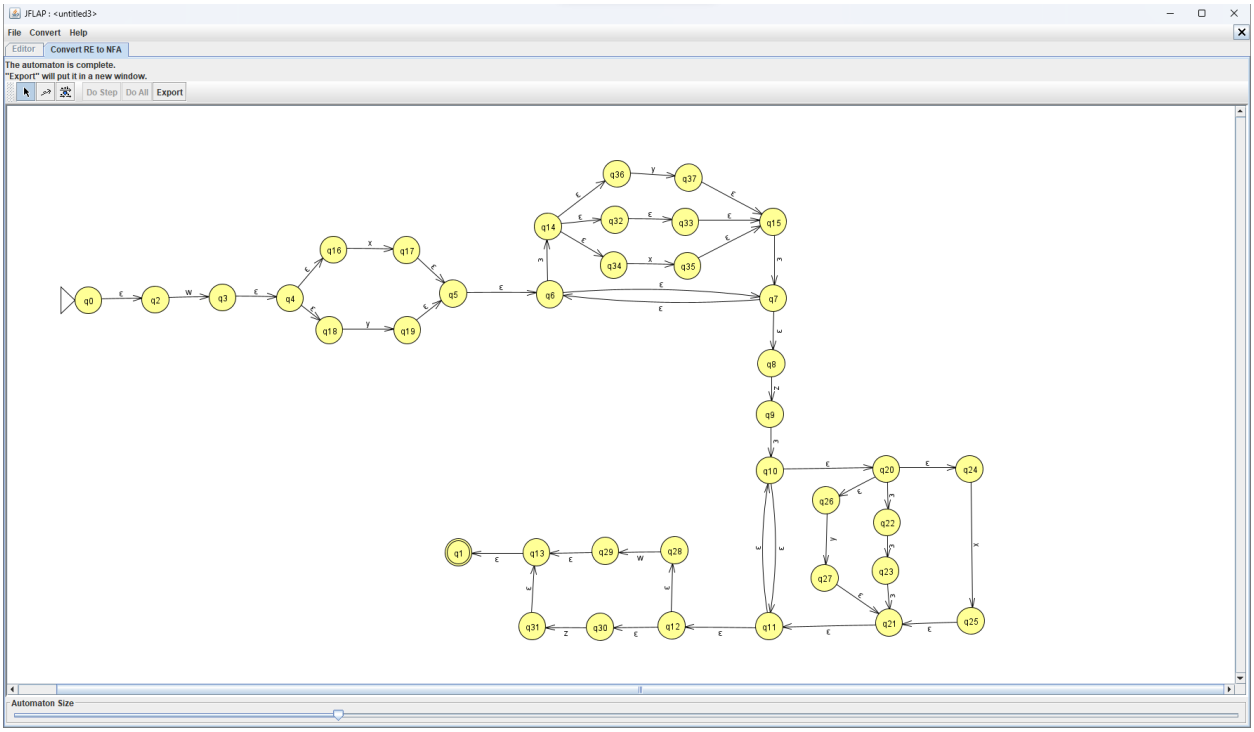
   w(x|y)+z(x|y)*[wz]
   Explicación de la ER:

   1. La cadena comienza con W.
   2. (x|y), esto representa una secuencia de uno o más ‘x’ o ‘y’.
   3. A continuación viene una ‘z’.
   4. Ahora puede venir una secuencia de ‘x’ o ‘y’ o no.
   5. Finalmente la cadena termina con w o con z.

2. **Traducción de la ER a formato JFLAP.**

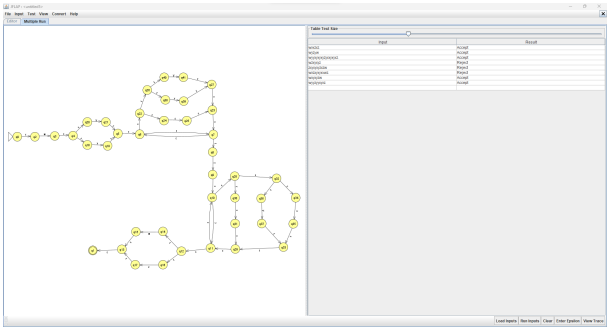
   w(x+y)(!+(x+y))*z(!+(x+y))*(w+z)

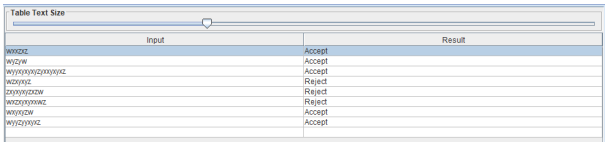
3. **Transformación de la ER a un Autómata Finito No Determinista**



4. **Análisis del Autómata Finito No Determinista para comprobar su validez**

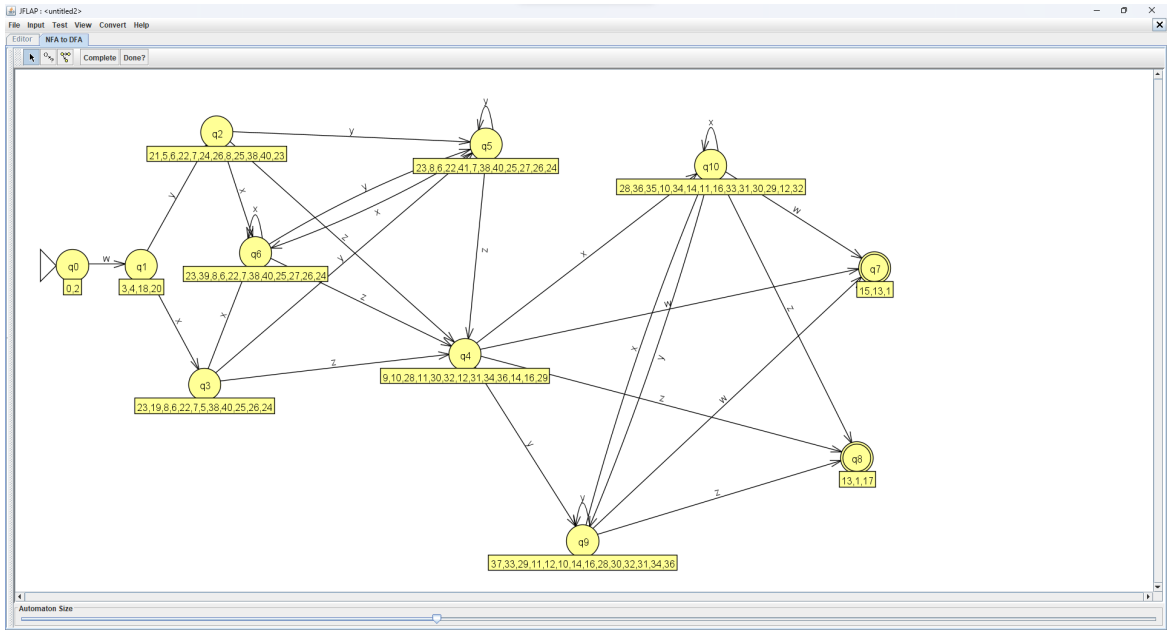
En este punto se han ejecutado varios inputs con distintas cadenas para comprobar el correcto funcionamiento.



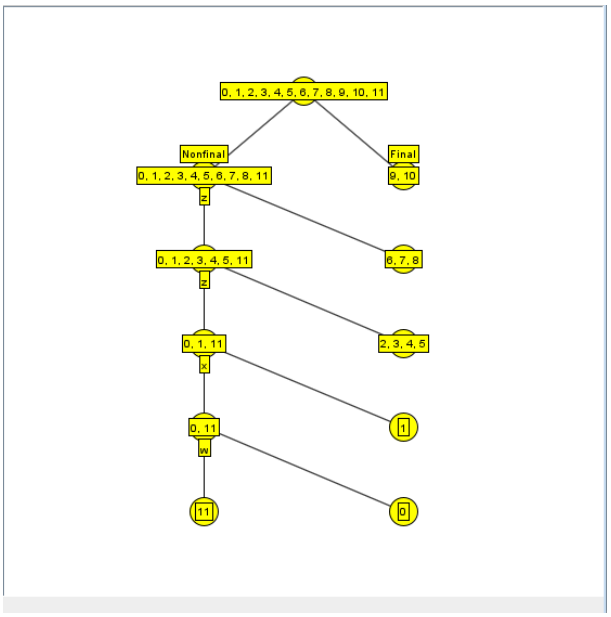


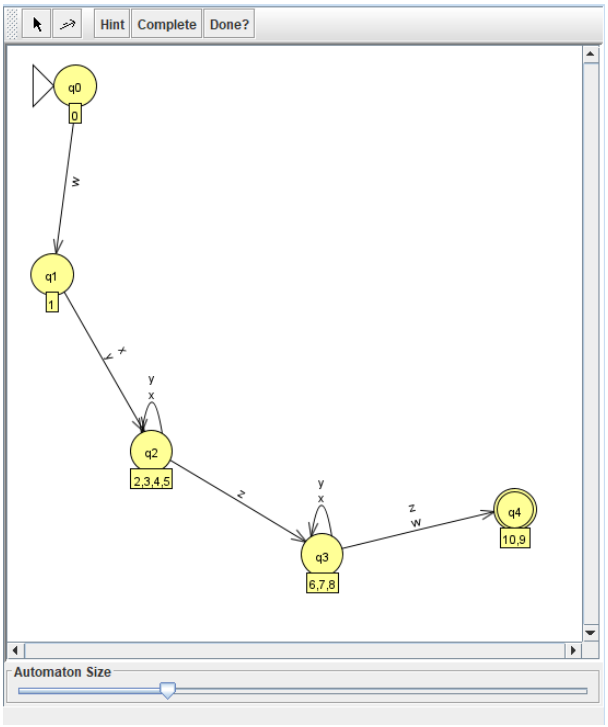
5. **Transformación del AFND a un Autómata Finito Determinista**

En este caso obtenemos dos estados finales que son Q7 y Q8.

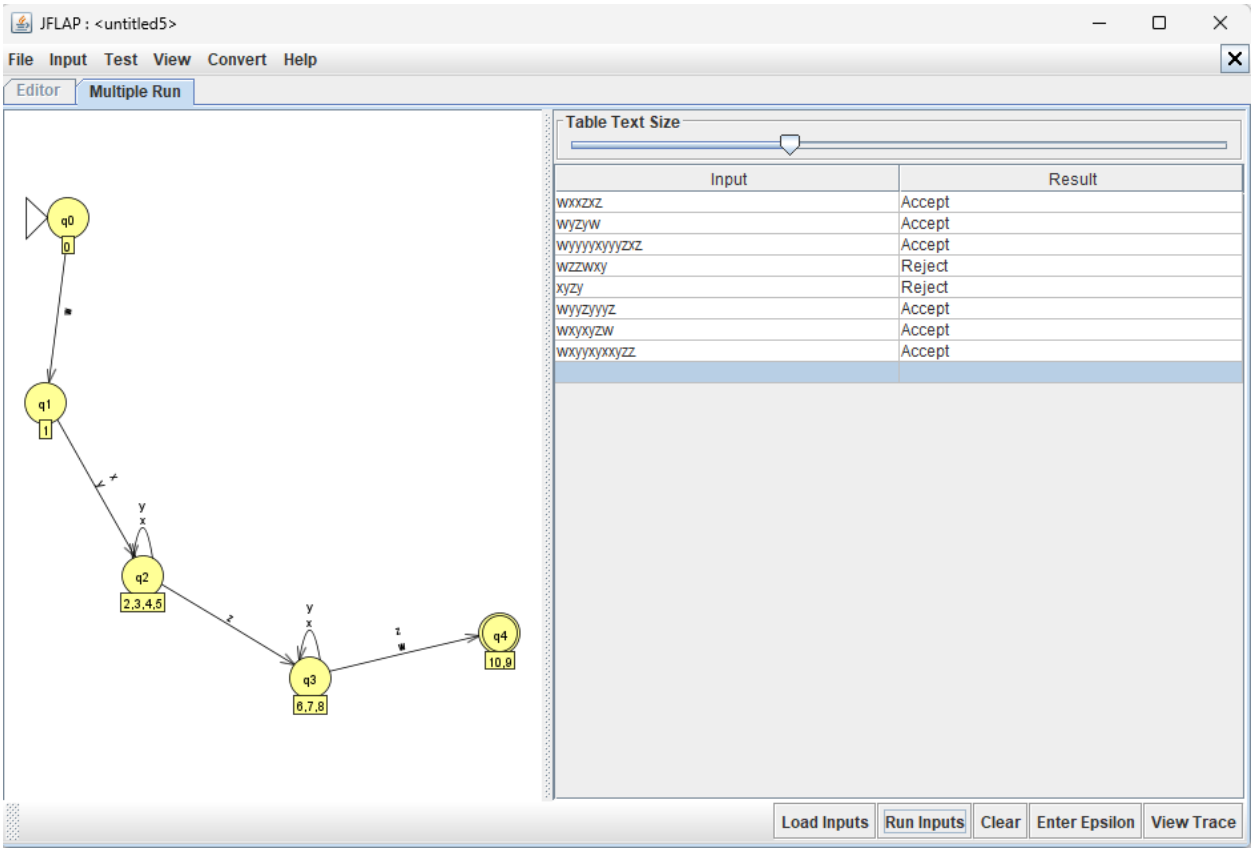


6. **Minimización del AFD.**





7. **Comprobación de validez con cadenas de entrada.**



8. **Transformación del AFD a un matriz de transición de estados para su implementación en una máquina de estados.**

AFD	w	x	y	AFD minimizado	w	x	y
q0I	q1			q0	q1		
q1		q3	q2	q1		q2	q2
q2		q6	q5	q2		q2	q2
q3		q6	q5	q3	q4	q3	q3
q4	q7	q10	q9	q4			
q5		q6	q5	q4			
q6		q6	q5	q4			
q7F							
q8F							
q9	q7	q10	q9	q8			
q10	q7	q10	q9	q8			

Segunda ER

1. **Alfabeto utilizado**

   `W,X,Y,Z`

2. **La expresion a introducir es:**

   a. `(x|y)(w|z)+(y|z)(w)*[xy]` Teoria

   b. (x+y)((w+z)(!+(w+z))*(y+z)(w)*(x+y) JFLAPS

3. **Explicacion:**

   a. `(x|y)`: Esto se traduce en que el primer carácter puede ser `x` o `y`.

   b. `(w|z)+`: Esto indica que los siguientes caracteres pueden ser `w` o `z`, y al menos uno de ellos debe estar presente (debido al signo `+`).

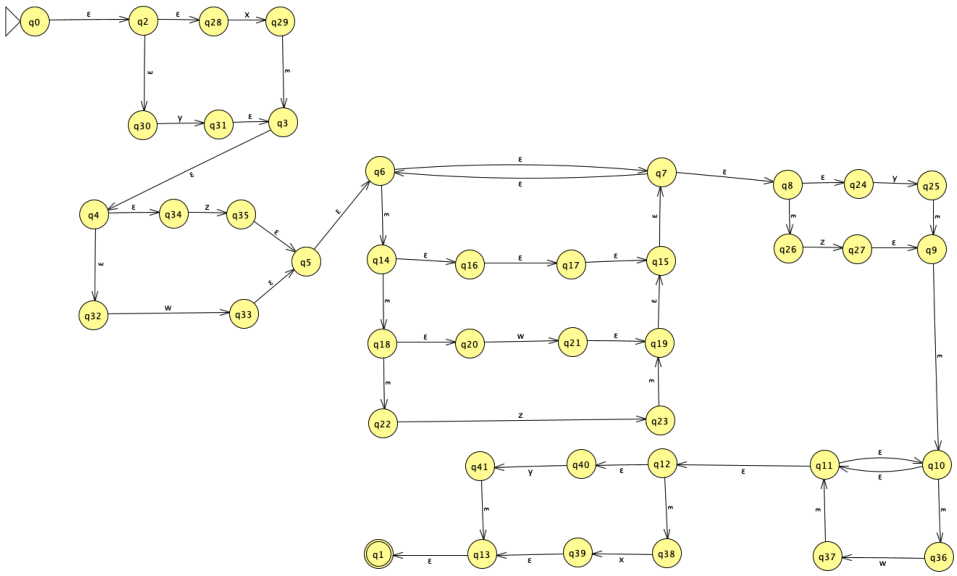
   c. `(y|z)`: Esto se traduce en que después de los caracteres `w` o `z`, debe haber un carácter `y` o `z`.

   d. `(w)*`: Esto indica que después puede haber cero o más caracteres `w`.

   e. `[xy]`: Finalmente, el último carácter debe ser `x` o `y`.

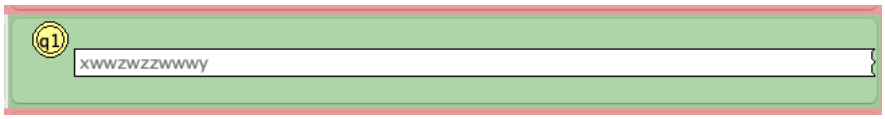
4. **Convert to NFA (automata finito no determinista)**

5. **NFA of the expression**

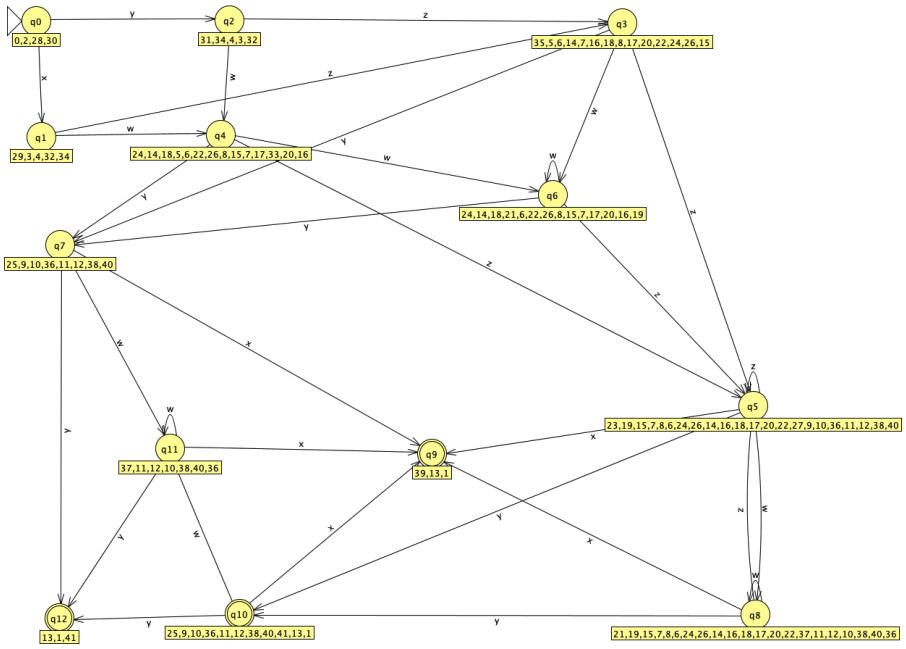


5. **Input Step with Closure**

`xwwzwzzwwwy`

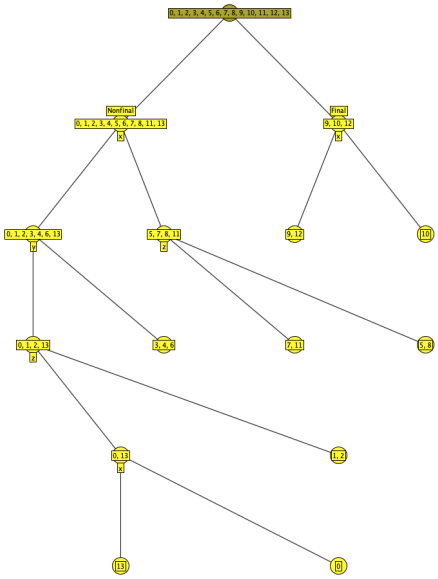


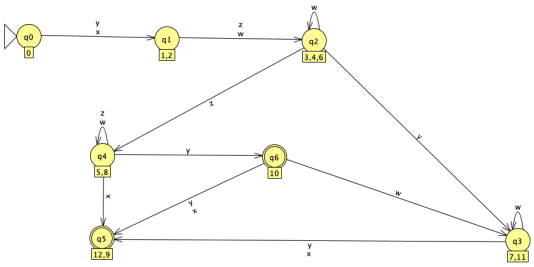
6. **AFD**



Input	Result
xzywy	Accept
zwzwzzwy	Reject
xywwzzww	Reject
yzwywx	Accept
ywwwwzwx	Accept
xzwzwzwzzx	Accept

7. **Minimización del AFD.**





8. **Transformación del AFD a un matriz de transición de estados para su implementación en una máquina de estados.**

AFD	w	x	y	AFD Minimizado	w	x	y
q0		q1	q2	q0		q1	q1
q1	q4			q1	q2		
q2	q4			q2	q2		q3
q3	q6		q7	q3	q3	q5	q5
q4	q6		q7	q4	q4	q5	q6
q5	q8	q9	q10	q5			
q6	q6		q7	q6	q3	q5	q5
q7	q11	q9	q12				
q8	q8	q9	q10	q5			
q9F							
q10F	q11	q9	q12				
q11	q11	q9	q12				
q12							

Implementación de un programa que, dada una matriz de transición de estados, implemente una máquina que permita realizar las siguientes dos operaciones:

1. Dada una cadena de texto de entrada, analizarla para determinar si esa cadena de texto cumple con la ER original.

2. Dar todas las posibles cadenas de texto de entradas válidas, hasta un número máximo determinado configurable (p.ej.100), que no sobrepasen una longitud máxima configurable (p.ej.10 caracteres).

Clase Autómata

```
public class AutomataFD {

    List<Character> alfabeto;
    List<Integer> estados;
    List<Integer> estadosFinales;
    int estadoInicial;

    HashMap<Integer, HashMap<Character, Integer>> matriz;

    public AutomataFD() {

        this.alfabeto = new ArrayList<>();
        this.estados = new ArrayList<>();
        this.estadosFinales = new ArrayList<>();
        this.matriz = new HashMap<>();

    }
```

En primer lugar creamos la clase `AutomataFD`, que representa un autómata finito determinista (AFD). Los componentes de esta clase son:

1. Listas de alfabeto, estados y estados finales:
   - `alfabeto`: Es una lista de caracteres que representa el alfabeto del AFD.
   - `estados`: Es una lista de enteros que contiene los estados del AFD.
   - `estadosFinales`: Es una lista de enteros que almacena los estados finales del AFD.
   - `estadoInicial`: Un entero que indica el estado inicial del AFD.
2. Matriz de transiciones:
   - `matriz`: Es un HashMap anidado que representa la función de transición del AFD. La clave exterior del HashMap es un estado del AFD, y el valor asociado es otro HashMap que mapea caracteres del alfabeto a estados de destino. En otras palabras, `matriz` es una representación de la tabla de transiciones del AFD.
3. Constructor:
   - El constructor de la clase inicializa todas las listas y el HashMap.

```
  public void cargarAlfabeto() {

    alfabeto.add('w');
    alfabeto.add('x');
    alfabeto.add('y');
    alfabeto.add('z');

}
```

Carga el alfabeto que hemos utilizado, formado por los caracteres W X Y Z.

```
public void cargarEstados1() {

        for (int i = 0; i < 6; i++) {

            estados.add(i);

        }

    }

    public void cargarEstados2() {

        for (int i = 0; i < 7; i++) {
```

```
            estados.add(i);

        }
    }
```

Estos métodos cargan los estados transicionales de las matrices dependiendo que Expresión Regular utilicemos.

```
public void establecerEstadoInicial() {

        estadoInicial = estados.get(0);

    }

    public void establecerEstadosFinales1() {

        estadosFinales.add(4);

    }

    public void establecerEstadosFinales2() {

        estadosFinales.add(5);
        estadosFinales.add(6);

    }

    public void inicializarMatriz() {

        for (int i = 0; i < estados.size(); i++) {
            matriz.put(i, new HashMap<>());
        }

    }

    }
```

Establecer estado inicial es un método común para ambos casos ya que el estado inicial siempre es el estado Q0.

Establecer estados finales para cada caso, en el primero el estado final es 4, y en el segundo los estados finales son 5 y 6.

Inicializar la matriz de transiciones del AFD (el HashMap anidado) crea una entrada vacía en el HashMap para cada estado. Esto prepara la matriz de transiciones para definir las transiciones entre los estados

```
public void cargarMatriz1() {

        //Estado 0
        matriz.get(0).put('w', 1);

        //Estado 1
        matriz.get(1).put('x', 2);
        matriz.get(1).put('y', 2);

        //Estado 2
        matriz.get(2).put('x', 2);
        matriz.get(2).put('y', 2);
        matriz.get(2).put('z', 3);

        //Estado 3
        matriz.get(3).put('w', 4);
        matriz.get(3).put('x', 3);
        matriz.get(3).put('y', 3);
        matriz.get(3).put('z', 4);

    }
```

El método `cargarMatriz1()` define las transiciones entre los estados:

- Estado 0:
    - Cuando se lee el carácter 'w', el AFD transita al estado 1.

- Estado 1:
  - Cuando se lee el carácter 'x', el AFD permanece en el estado 1.
  - Cuando se lee el carácter 'y', el AFD transita al estado 2.
- Estado 2:
  - Cuando se lee el carácter 'x', el AFD permanece en el estado 2.
  - Cuando se lee el carácter 'y', el AFD permanece en el estado 2.
  - Cuando se lee el carácter 'z', el AFD transita al estado 3.
- Estado 3:
  - Cuando se lee el carácter 'w', el AFD transita al estado 4.
  - Cuando se lee el carácter 'x', el AFD transita al estado 3.
  - Cuando se lee el carácter 'y', el AFD transita al estado 3.
  - Cuando se lee el carácter 'z', el AFD transita al estado 4.

```
public void cargarMatriz2() {

        //Estado 0
        matriz.get(0).put('x', 1);
        matriz.get(0).put('y', 1);

        //Estado 1
        matriz.get(1).put('w', 2);
        matriz.get(1).put('z', 2);

        //Estado 2
        matriz.get(2).put('w', 2);
        matriz.get(2).put('y', 3);
        matriz.get(2).put('z', 4);

        //Estado 3
        matriz.get(3).put('w', 3);
        matriz.get(3).put('x', 5);
        matriz.get(3).put('y', 5);

        //Estado 4
        matriz.get(4).put('w', 4);
        matriz.get(4).put('x', 5);
        matriz.get(4).put('y', 6);
        matriz.get(4).put('z', 4);

        //Estado 6
        matriz.get(6).put('w', 3);
        matriz.get(6).put('x', 5);
        matriz.get(6).put('y', 5);

    }
```

`cargarMatriz2()`

- Estado 0:
  - Cuando se lee el carácter 'x', el AFD transita al estado 1.
  - Cuando se lee el carácter 'y', el AFD transita al estado 1.
- Estado 1:
  - Cuando se lee el carácter 'w', el AFD transita al estado 2.
  - Cuando se lee el carácter 'z', el AFD transita al estado 2.
- Estado 2:
  - Cuando se lee el carácter 'w', el AFD permanece en el estado 2.

    - Cuando se lee el carácter 'y', el AFD transita al estado 3.
    - Cuando se lee el carácter 'z', el AFD transita al estado 4.
- Estado 3:
    - Cuando se lee el carácter 'w', el AFD transita al estado 3.
    - Cuando se lee el carácter 'x', el AFD transita al estado 5.
    - Cuando se lee el carácter 'y', el AFD transita al estado 5.
- Estado 4:
    - Cuando se lee el carácter 'w', el AFD transita al estado 4.
    - Cuando se lee el carácter 'x', el AFD transita al estado 5.
    - Cuando se lee el carácter 'y', el AFD transita al estado 6.
    - Cuando se lee el carácter 'z', el AFD transita al estado 4.
- Estado 6:
    - Cuando se lee el carácter 'w', el AFD transita al estado 3.
    - Cuando se lee el carácter 'x', el AFD transita al estado 5.
    - Cuando se lee el carácter 'y', el AFD transita al estado 5.

```
public Integer getSiguienteEstado(Integer estado, Character caracter) {

        return matriz.get(estado).get(caracter);

    }

    public boolean isFinal(Integer estado) {

        return estadosFinales.contains(estado);

    }

    public Integer getEstadoInicial() {

        return estadoInicial;

    }
```

1. `getSiguienteEstado(Integer estado, Character caracter)` : Este método toma dos argumentos, un estado actual y un carácter de entrada, y devuelve el siguiente estado al que el AFD debería transicionar si está en el estado actual y lee el carácter dado. La función utiliza la matriz de transiciones previamente definida para buscar el estado de destino apropiado. Si no se define una transición para el estado actual y el carácter dado, este método devolverá `null`.
2. `isFinal(Integer estado)` : Este método toma un estado como argumento y verifica si ese estado es un estado final (es decir, si está en la lista de estados finales previamente definidos). Si el estado pasado como argumento se encuentra en la lista de estados finales, la función devuelve `true`; de lo contrario, devuelve `false`.
3. `getEstadoInicial()` : Este método devuelve el estado inicial del AFD, que es el estado desde el cual comienza a procesar las cadenas de entrada.

```
public void configuracion1() {

        cargarAlfabeto();
        cargarEstados1();
        inicializarMatriz();
        cargarMatriz1();
        establecerEstadoInicial();
        establecerEstadosFinales1();

    }
```

```
    public void configuracion2() {

        cargarAlfabeto();
        cargarEstados2();
        inicializarMatriz();
        cargarMatriz2();
        establecerEstadoInicial();
        establecerEstadosFinales2();

    }
```

Los métodos `configuracion1()` y `configuracion2()` son funciones que configuran la instancia de la clase `AutomataFD` con dos conjuntos diferentes de reglas y transiciones para el autómata finito determinista (AFD). Estos métodos son los que seleccionaremos en la clase Main para trabajar con una u otra ER.

Clase MaquinaEstados

La clase `MaquinaEstados` representa una máquina de estados que utiliza un autómata finito determinista (AFD) definido en la clase `AutomataFD`. Esta máquina de estados se mueve entre estados según los caracteres de entrada y verifica si el estado actual es final.

```
public class MaquinaEstados {

    int estadoActual;
    AutomataFD AFD;

    public MaquinaEstados(AutomataFD AFD) {
        this.AFD = AFD;
    }
```

Constructor `MaquinaEstados(AutomataFD AFD)`: El constructor recibe una instancia de `AutomataFD` como argumento y la almacena en la variable miembro `AFD`. Esto permite que la máquina de estados utilice el AFD configurado previamente para procesar cadenas de entrada.

```
public void inicializar() {

        estadoActual = AFD.getEstadoInicial();

    }

    public void acepta(Character caracter) throws Exception {

        Integer estadoTemp = AFD.getSiguienteEstado(estadoActual, caracter);

        if (estadoTemp != null) {

            estadoActual = estadoTemp;

        } else {

            throw new Exception("Err");

        }

    }

    public boolean isFinal() {

        return AFD.isFinal(estadoActual);

    }
```

1. Método `inicializar()`: Este método inicializa la máquina de estados estableciendo el estado actual como el estado inicial del AFD. Esto se hace llamando al método `getEstadoInicial()` del AFD configurado y almacenando ese estado en la variable

`estadoActual`.

2. Método `acepta(Character caracter)`: Este método toma un carácter como entrada y procesa el carácter para mover la máquina de estados al siguiente estado. Verifica la transición definida en el AFD configurado para el estado actual y el carácter dado llamando al método `getSiguienteEstado(estadoActual, caracter)` del AFD. Si se encuentra una transición válida, actualiza el estado actual de la máquina de estados al nuevo estado. Si no se encuentra una transición válida, lanza una excepción con el mensaje "Err".
3. Método `isFinal()`: Este método verifica si el estado actual de la máquina de estados es un estado final según las reglas definidas en el AFD configurado. Utiliza el método `isFinal(estadoActual)` del AFD para realizar esta verificación. Devuelve `true` si el estado actual es final y `false` en caso contrario.

```
public boolean comprobarCadena(String cadena) {
        inicializar();
        boolean correcto = false;
        int contador = 0;
        try {
            for (int i = 0; i < cadena.length(); i++) {
                Character caracter = cadena.charAt(i);

                acepta(caracter);
                contador++;
                if (isFinal() && cadena.length() == contador) {

                    correcto = true;

                }
            }
        } catch (Exception ex) {
            // System.out.println("La cadena no es valida");

        }
        return correcto;

    }
```

Esta función permite comprobar si una cadena es aceptada por el AFD, siguiendo las reglas y transiciones definidas en el AFD, y devuelve un valor booleano que indica si la cadena es válida.

```
public String generateRandomCadena(Random random, int maxLength) {
        String alphabet = "wxyz"; // Alfabeto de la expresión regular
        int cadenaLength = random.nextInt(maxLength + 1); // Longitud aleatoria hasta el máximo
        StringBuilder sb = new StringBuilder(cadenaLength);

        for (int i = 0; i < cadenaLength; i++) {
            int randomIndex = random.nextInt(alphabet.length());
            char randomChar = alphabet.charAt(randomIndex);
            sb.append(randomChar);
        }

        return sb.toString();
    }
```

Esta función se utiliza para generar una cadena aleatoria con una longitud máxima dada y utilizando un alfabeto específico. Esto puede ser útil para probar o evaluar la capacidad del AFD para reconocer cadenas aleatorias de entrada. La cadena generada se basará en el alfabeto definido y tendrá una longitud aleatoria hasta el valor máximo especificado.

Clase Main

```
public static void main(String[] args) {

        List<String> cadenasValidas = new ArrayList<>();

        AutomataFD AFD = new AutomataFD();
```

```
      //AFD.configuracion1();
        AFD.configuracion2();
        MaquinaEstados ME = new MaquinaEstados(AFD);

        //apartado 1.a
        System.out.println(ME.comprobarCadena("wxzz"));
        //apartado 1.b
        System.out.println(ME.comprobarCadena("xwzy"));
```

1. Se crea una lista llamada `cadenasValidas` que se utilizará para almacenar cadenas válidas según las reglas del autómata finito determinista (AFD).
2. Se crea una instancia de la clase `AutomataFD` llamada `AFD`.
3. Se llama al método `configuracion2()` en el objeto `AFD`, que configura el AFD con un conjunto específico de reglas y transiciones, según las definiciones que proporcionaste anteriormente.
4. Se crea una instancia de la clase `MaquinaEstados` llamada `ME`, que se inicializa con el objeto `AFD`. Esta máquina de estados utilizará las reglas del AFD configurado para procesar cadenas de entrada.

Por último hace las comprobaciones de las cadenas que se le pasan que darán un resultado u otro en función de la configuración seleccionada.

```
//apartado 2
        Random random = new Random();
        int maxLength = 10;
        int validCount = 0;

        while (validCount < 100) {
            String randomCadena = ME.generateRandomCadena(random, maxLength);
            //System.out.println(randomCadena);
            if (ME.comprobarCadena(randomCadena) && !cadenasValidas.contains(randomCadena)) {
                System.out.println("Cadena valida encontrada: " + randomCadena);
                validCount++;
                cadenasValidas.add(randomCadena);
                System.out.println(validCount);

            }
        }
        System.out.println(cadenasValidas);
    }
```

Este código genera cadenas aleatorias, las verifica según las reglas del AFD y las almacena en la lista `cadenasValidas` si son válidas. El bucle continúa hasta que se hayan encontrado 100 cadenas válidas. Las cadenas válidas encontradas se muestran en la consola al final.