PROCESADORES DEL LENGUAJE

César Martín Guijarro

Universidad de Alcalá de Henares

ÍNDICE

CAPÍTULO I: Introducción de las expresiones regulares

Este capítulo trata del proceso de introducción de expresiones regulares al proyecto JAVA. Para ello, se usará como apoyo el programa JFLAP.

En primer lugar, se debe convertir una expresión regular normal a una expresión regular aceptada por JFLAP.

Por ejemplo, si se tiene la expresión *a+(bc*[m-q])+* su expresión equivalente en JFLAP sería la siguiente: *a(!+a)*(bc*(m+n+o+p+q))(!+(bc*(m+n+o+p+q)))**

En JFLAP se obtendrá el siguiente AFND:

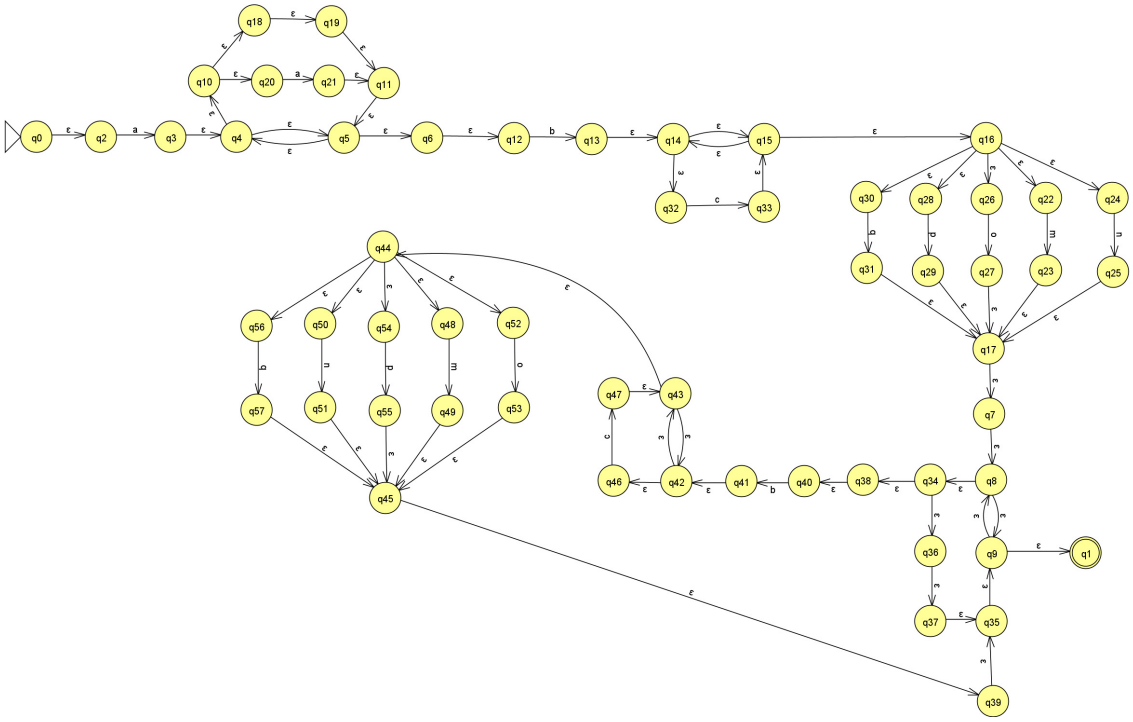

Figura 1.1 – AFND de la expresión de ejemplo

A continuación, se debe convertir el AFND obtenido a un AFD, cuyo resultado será:

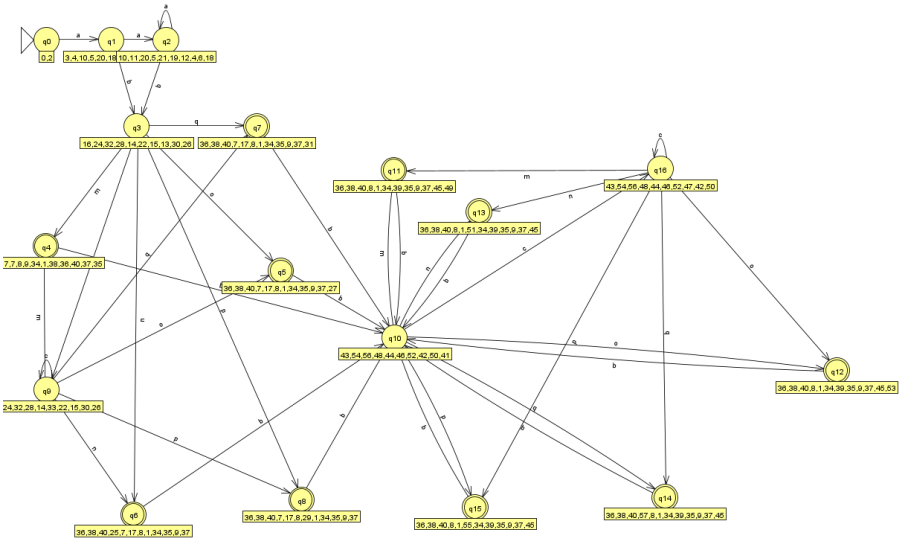

Figura 1.2 – AFD de la expresión de ejemplo

Dicho AFD podrá traducirse a la siguiente matriz, la cual tendrá una fácil implementación en JAVA:

	a	b	c	m	n	o	p	q
Q_0	Q_1							
Q_1	Q_2	Q_3						
Q_2	Q_2	Q_3						
Q_3			Q_9	Q_4	Q_6	Q_5	Q_8	Q_7
Q_4		Q_{10}						
Q_5		Q_{10}						
Q_6		Q_{10}						
Q_7		Q_{10}						
Q_8		Q_{10}						
Q_9			Q_9	Q_4	Q_6	Q_5	Q_8	Q_7
Q_{10}			Q_{16}	Q_{11}	Q_{13}	Q_{12}	Q_{15}	Q_{14}
Q_{11}		Q_{10}						
Q_{12}		Q_{10}						
Q_{13}		Q_{10}						
Q_{14}		Q_{10}						
Q_{15}		Q_{10}						
Q_{16}			Q_{16}	Q_{11}	Q_{13}	Q_{12}	Q_{15}	Q_{14}

Figura 1.3 – Matriz de la expresión del ejemplo

CAPÍTULO II: Implementación AFD y MDE en JAVA

A continuación, se abordará la manera en la que se ha implementado la clase AFD y MDE en Java, hablando de sus atributos, métodos y funcionamiento.

- AFD: El AFD se ha implementado utilizando un HashMap<Entero, HashMap<Carácter,Entero>>. El primero de estos hashmaps representa la selección de una fila de la matriz, y el segundo, la selección de una celda de una fila según el carácter. A parte, tiene un atributo que determina cuál es el estado inicial, así como una lista de los estados finales. Por otro lado, consta de otro atributo que contiene el alfabeto del AFD.
  El AFD está implementado en *"AutomataFinitoDeterminista.java"*.

- MDE: tiene dos atributos: el AFD y el estado actual. Utilizando el método *"aceptar()"* se puede hacer un salto de un estado a otro según el estado actual y el carácter que llegue. Por otro lado, el método *"buscarToken(cadena)"* busca cuál es el token con el lexema más largo que puede ser reconocido por la MDE. Para ello, utilizará dos Strings: *cadenaTMP* (que se inicializa como vacía e irá agregando un carácter cada vez que se acepte el mismo con el método *aceptar()*. Los caracteres que se pasan al método *aceptar()* son los que en ese momento estén en la posición inicial de la cadena recibida como argumento) y *cadenaFinal* (que se inicia vacía y que se actualiza con el valor de *cadenaTMP* cada vez que se pasa por un estado final. Haciendo que siempre en esta cadena, el valor que haya sea equivalente al lexema más largo reconocido por la MDE. Si la MDE no reconoce ningún lexema, esta cadena se quedará vacía)
  Al finalizar el método, se retorna el valor de *cadenaFinal*. El método termina cuando la cadena pasada como argumento se queda vacía.
  La MDE está implementada en *"MaquinaDeEstados.java"*.

CAPÍTULO III: Implementación Analizador Léxico

Pese a que en el enunciado lo que se pide es simplemente comprobar si una cadena dada es aceptada o no por una ER, en este proyecto he querido ir más allá e implementar un Analizador Léxico completo en Java, el cual acepta un número *n* MDE las cuales analizarán una misma cadena y extraerán todos los tokens contenidos en la misma.

En "AnalizadorLexico.java" se encuentra el método "*analizar()*" que tiene como parámetros una lista de AFD, con los cuales se instanciarán unas MDE que analizarán de manera independiente el argumento "cadena" usando el método *MDE.buscarToken();* y finalmente una lista de nombres de cada MDE la cual indica qué nombre se mostrará en cada token según que MDE reconozca el lexema más largo. Tras haber analizado la cadena todas las MDE, se determina cuál de estas ha reconocido el lexema más largo, y se devuelve el token que corresponda. Finalmente, se repite todo el proceso analizando ahora desde el punto en el que terminaba el lexema más largo que se reconoció anteriormente. Si ninguna MDE reconoce nada, se devolverá un token de error, cuyo lexema será el primer carácter de la cadena. Este proceso continuará hasta que la cadena quede vacía.

CAPÍTULO IV: Ejecución del ejercicio 3

En este capítulo se describirá la ejecución del ejercicio 3, ya que este, al ser el más complejo, es el que mejor dará a entender el funcionamiento del programa.

En este ejercicio el analizador léxico nos dirá de qué tipo (dentro de los tipos de datos de PSEInt) son los lexemas que encuentre.

En primer lugar, hablaremos de las ER utilizadas y sus peculiaridades:

-Lógicos: debe poder aceptar *verdadero* y *falso* independientemente de que alguno de los caracteres esté en mayúsculas. Ejemplo: *fALSo.*

-Cadenas: aceptará (prácticamente) cualquier conjunto de carácteres ASCII rodeado de comillas dobles, simples o una mezcla de estas. Ejemplo: *“”, ‘Hola”.*

-Numérico: acepta cualquier valor entero o real introducido y las absurdas posibles entradas que admite PSEInt. Ejemplo: *2, +3.5, -., .8*

- EJEMPLO DE EJECUCIÓN REAL:

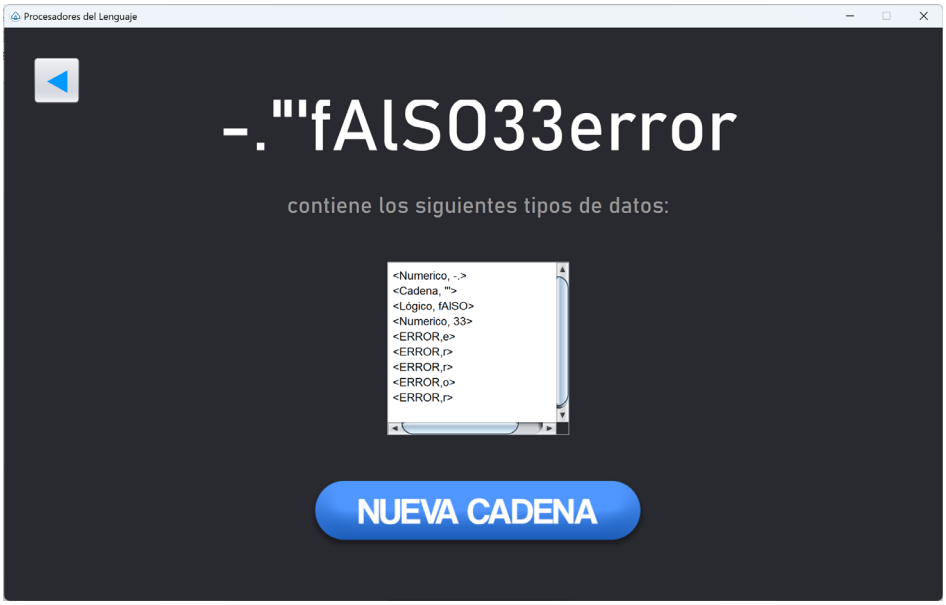

Figura 4.1 – Ventana Ejercicio 3

En este caso, se llamará al analizador léxico pasándole los AFD asociados a las ER de Lógico, Cadena y Numérico.

Por lo explicado anteriormente, la MDE asociada a Numérico, detectará el token *<Numérico, -.>*. Después, la MDE asociada a Cadena, detectará el token *<Cadena, “’>*... hasta que se da el último token de error *<ERROR, r>*; momento en el cual, la cadena quedaría vacía en el método *analizar()* y, por ende, se daría por finalizado el trabajo del analizador léxico.

CAPÍTULO V: Información adicional

Para concluir, se mencionarán ciertos factores a tener en cuenta sobre el resto de ejercicios para entender mejor su funcionamiento y predecir qué salidas se deberían obtener, según las entradas introducidas.

- Ejercicio 1: En el ejercicio 1, la expresión regular utilizada es la que se muestra en el capítulo 1.

- Ejercicio 2: En el ejercicio 2, "AUT0" reconoce lexemas del tipo "si", "AUT1", del tipo "sino" y "AUT2" del tipo "mientras".

- A la hora de realizar el proyecto se ha buscado crear un **código limpio enfocado en la "universalidad".** Esto puede ser visto en la manera en la que todos los ejercicios llaman a un mismo método del analizador léxico para llevar a cabo sus análisis, así como la manera en la que se ha facilitado la implementación y los cambios de las Máquinas de Estado que hay en cada ejercicio, gracias a la inclusión de métodos como *AFD.leerCSV()* que importa automáticamente el AFD a partir de su matriz en formato .csv; y principalmente gracias a la manera en la que **sólo editando o añadiendo una ruta .csv** en cualquiera de los ejercicios **podemos cambiar o añadir nuevas MDE para la búsqueda de tokens** por parte del Analizador Léxico.