Universidad de Alcalá

Escuela Politécnica Superior

Grado de Ingeniería en Computadores

Práctica 1: Expresiones Regulares

Compiladores

Alumno: Sergio Sierra Arquero
sergio.sierra@edu.uah.es

Octubre 2023

Índice general

Introducción

1.1. Motivación

Los compiladores desempeñan un papel fundamental en el proceso de traducción de código fuente a código ejecutable, facilitando la comunicación entre humanos y máquinas.

Una de las fases cruciales en este proceso es el **análisis léxico**, que conduce al proceso de transformación de expresiones regulares en autómatas finitos no deterministas (NFA - *Non-deterministic Finite Automaton*) y, de estos, a autómatas finitos deterministas (DFA - *Deterministic Finite Automaton*. Este procedimiento tiene implicaciones significativas en la eficiencia y la comprensión de la lógica subyacente en el comportamiento de los lenguajes formales.

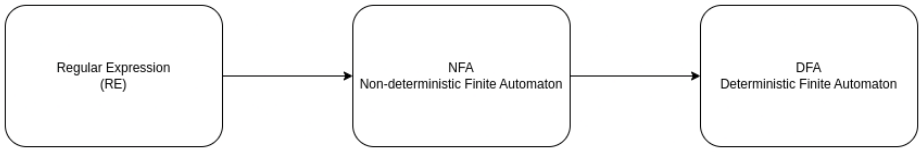


Figura 1.1: Esquema general del proceso.

En este trabajo, se explorará el uso de JFLAP, una potente herramienta para el diseño y simulación de autómatas. Con su ayuda, abordaremos el proceso descrito en la *Figura 1.1*, esencial en la construcción de compiladores eficaces y precisos.

1.2. Planteamiento

1.2.1. Alfabeto y ER's

Para la realización de esta práctica se empleará el siguiente alfabeto que cumple que: $|\Sigma| \geq 3$, tal y como se especifica.

$$\Sigma = \{a, b, c\} \tag{1.1}$$

Se trabajará con el siguiente par de expresiones regulares:

- **RegEx 1**: a?c(ba)*[ac]
- **RegEx 2**: (abc)+bba*

ER	JFLAP
()	()
*	*
+	NA
|	+
[]	NA
?	NA
ϵ	!

Tabla 1.1: Equivalencias ER y JFLAP

Para poder trabajar con la herramienta **JFLAP**, será muy importante tener en cuenta las equivalencias (*Tabla 1.1*) entre los operadores que se emplean usualmente al tratar con expresiones regulares y los que están disponibles en JFLAP.

En el proyecto de Maven que se proporciona junto a esta memoria, se incluye una utilidad para traducir expresiones regulares de modo que estas se puedan utilizar directamente con JFLAP.

com.ssierra.uah.compiladores.pl1.util.Translator

Se observa entonces que, las expresiones a tratar con JFLAP son:

- **RegEx 1 (JFLAP syntax)**: (!+a)c(ba)*(a+c)
- **RegEx 2 (JFLAP syntax)**: (abc)(!+(abc))*bba*

Obtención de Autómatas con JFLAP

A continuación se hallarán las matrices de transición para los autómatas correspondientes a cada una de las expresiones regulares mencionadas en el apartado anterior.

Para ello, se llamará al programa JFLAP mediante el siguiente comando:

```
java -jar path-to-jflap
```

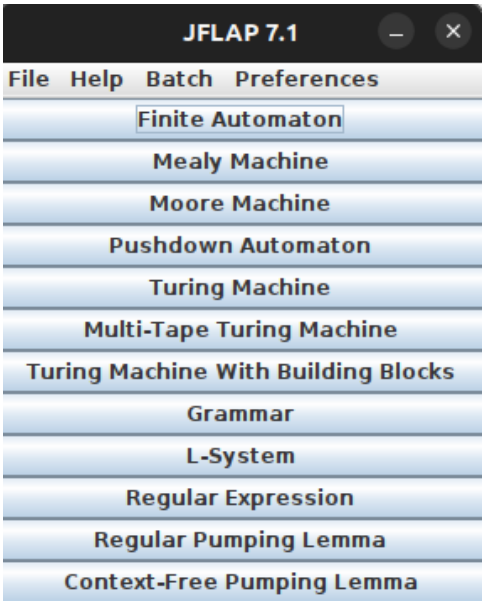


Figura 2.1: Pantalla principal del programa.

2.1. Autómatas finitos no deterministas

2.1.1. Expresión Regular I

Tras introducir la expresión en el formato que acepta JFLAP y resolver todos los pasos del algoritmo, se obtiene el grafo que se muestra en la siguiente figura para el autómata:

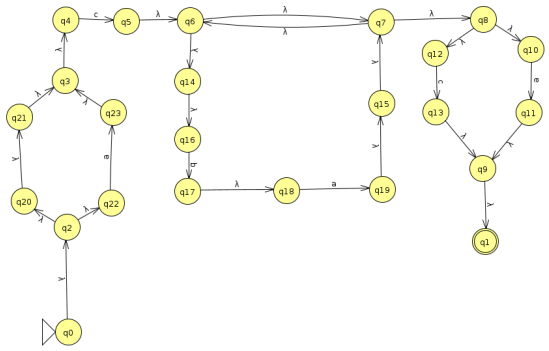


Figura 2.2: Grafo correspondiente al NFA.

2.1.2. Expresión Regular II

Para la segunda expresión regular se obtiene:

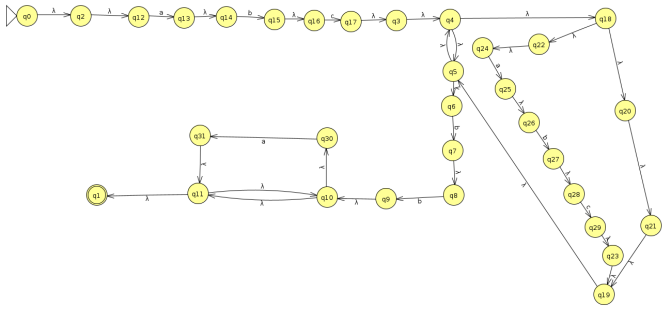
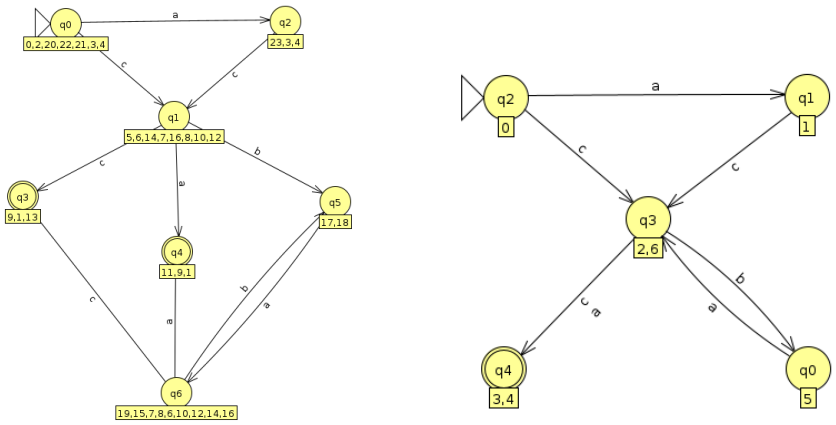


Figura 2.3: Grafo correspondiente al NFA.

2.2. Autómatas finitos deterministas minimizados

2.2.1. Expresión Regular I

El resultado que se obtiene para esta expresión regular es el que se puede apreciar en la siguiente figura (también se muestra el grafo resultante de minimizar el DFA).



(a) DFA (b) DFA Minimizado

Figura 2.4: RegEx I

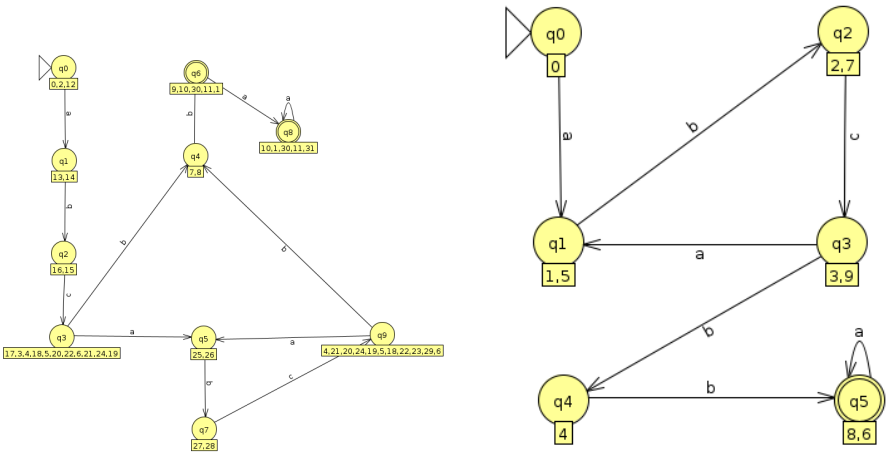
Analizando el grafo obtenido, se puede llegar a la correspondiente tabla/-matriz de transiciones.

Q/Σ	a	b	c
q2 (start)	q1	-	q3
q1	-	-	q3
q3	q4	q0	q4
q0	q3	-	-
q4 (final)	-	-	-

Tabla 2.1: Matriz de transición RegEx I

2.2.2. Expresión Regular II

De manera análoga a como se ha procedido con la expresión anterior, llegamos a las siguientes figuras, donde se muestra el autómata finito determinado y su equivalente minimizado para la segunda expresión regular.



(a) DFA (b) DFA Minimizado

Figura 2.5: RegEx II

Con su matriz de transición como se muestra en la siguiente tabla.

Q/Σ	a	b	c
q0 (start)	q1	-	-
q1	-	q2	-
q2	-	-	q3
q3	q1	q4	-
q4	-	q5	-
q5 (final)	q5	-	-

Tabla 2.2: Matriz de transición RegEx II

Implementación

3.1. Entorno de Desarrollo

La implementación de esta práctica, contenida en este repositorio de GitHub, se ha llevado a cabo en:

- Ubuntu 22.04 LTS
- NetBeans 17
- Java JDK19

El proyecto de NetBeans y código fuente se proporcionan de manera anexa a esta memoria. Si hubiera algún problema con la visualización del mismo, por favor, no duden en hacérmelo saber (sergio.sierra@edu.uah.es).

3.2. Estructura General del Programa

Para implementar los requisitos del enunciado, se emplean principalmente las clases **Automaton** y **State**. Adicionalmente, se proporciona una herramienta para transformar la sintaxis usual para las expresiones regulares en cadenas que puedan ser procesadas por la herramienta JFLAP.

La clase **Automaton** representa el autómata finito determinista minimizado que hemos obtenido anteriormente.

Su constructor toma el alfabeto de entrada, como cadena de caracteres, y la matriz de transición, que ha de separar sus columnas por comas (",") y sus filas por punto y coma (";"). Aquellos estados que den lugar a error o no estén *mapeados* a algún otro, han de representarse con el número entero -1.

3.2.1. Traducción de las matrices de transición

Además de organizarlas como se menciona anteriormente, será necesario renombrar los estados. Esto se hará asignando a cada uno de estos (en orden descendente) un índice $i \in \mathbb{N} + \{0\}$, en orden ascendente, es decir: 0, 1, 2...

- RegEx I (Matriz 3.1): **1,-1,2;-1,-1,2;4,3,4;2,-1,-1;-1,-1,-1**

$$\begin{pmatrix} 1 & -1 & 2 \\ -1 & -1 & 2 \\ 4 & 3 & 4 \\ 2 & -1 & -1 \\ -1 & -1 & -1 \end{pmatrix} \tag{3.1}$$

- RegEx II (Matriz 3.2): **1,-1,-1;-1,2,-1;-1,-1,3;1,4,-1;-1,5,-1;5,-1,-1**

$$\begin{pmatrix} 1 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 3 \\ 1 & 4 & -1 \\ -1 & 5 & -1 \\ 5 & -1 & -1 \end{pmatrix} \tag{3.2}$$

Para proceder con la práctica habrá que inicializar los autómatas con los siguientes códigos (se pueden observar dentro del método *main* del código aportado):

```
Automaton(String inputAlphabet, String stateMatrix, int finalState);
```

3.3. Comprobación de Cadenas

Una vez inicializados ambos autómatas, se comprobará la validez de las cadenas que se exponen a continuación gracias al método *validate()*.

3.3.1. RegEx I - 'acc' y 'cbababababaa'

Al introducir estas dos expresiones en el autómata, se puede observar cómo ambas dos son reconocidas como válidas.

Como se aprecia en la *Figura 3.1*, el método arroja el valor booleano **verdadero** para ambas cadenas.

3.3.2. RegEx II - 'abcbb' y 'abcabcbbaa'

De manera análoga a lo expuesto en la subsección anterior, se probarán los dos mismos valores con el mismo método *validate()*.

A continuación se muestran los resultados de ejecucion para cada uno de los autómatas.

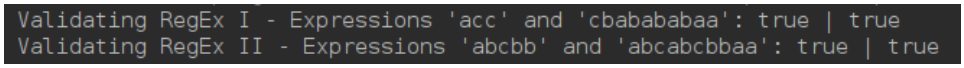


Figura 3.1: Resultado de ejecución.

3.4. Generación de Cadenas Válidas

3.4.1. Descripción del Algoritmo

Para generar las distintas posibles cadenas válidas para cada uno de los autómatas, se ha obtenido, en primer lugar, todas las posibles combinaciones de elementos del alfabeto, es decir, $|\Sigma|^n$ cadenas diferentes. (Donde n es la longitud de las cadenas a evaluar).

Una vez obtenido el conjunto de palabras a evaluar, se irán evaluando mediante la función empleada en el método anterior para así determinar si pertenecen o no al lenguaje definido por el autómata.

Resulta de vital importancia tener en cuenta que la complejidad de este algoritmo aumenta notablemente en función de la longitud del número de caracteres resultantes deseados. Aunque en esta práctica no se ha analizado en profundidad, es muy posible que el comportamiento sea exponencial $O(2^n)$

Si el problema requiere el análisis de cadenas más largas, **resultaría conveniente explorar otro algoritmo**.

3.4.2. Resultados de Ejecución

En la siguiente figura se expone el resultado de la ejecución del algoritmo. En este caso, se van generando todas las posibles cadenas hasta un determinado límite (de longitud de cadena) de manera creciente. Estas se van filtrando con el método de validación.

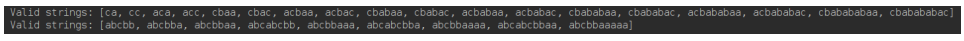


Figura 3.2: Cadenas generadas por los autómatas

Estos mismos resultados se pueden replicar ejecutando el archivo **Main.java** del proyecto.

Conclusiones

A lo largo de este estudio, hemos explorado una serie de pasos fundamentales en el diseño y optimización de autómatas finitos deterministas, utilizando como herramientas clave JFLAP y la programación en Java. Este proceso, esencial en el desarrollo de compiladores robustos y eficientes, ha revelado la importancia de la meticulosidad y la comprensión detallada de las expresiones regulares y sus correspondientes autómatas.

Se observa, a partir de los resultados obtenidos, que el autómata implementado se comporta de acuerdo a lo que se espera según la expresión regular de partida

En el ejemplo proporcionado en la práctica se menciona la obtención de mínimo 100 cadenas válidas para la expresión regular (de diez caracteres cada una). En este caso se obtiene un número de palabras considerablemente menor al esperado según dicho ejemplo. Esto es debido a la naturaleza de las expresiones regulares que hemos estado empleando. La proporcionada, además de ser más extensa, contiene más operadores **XOR** y **cierres**.

Bibliografía y Recursos

- Video Tutorial JFLAPS - Prof. Antonio Moratilla.
- Orientación sobre la estructura de la PL1 - Prof. Antonio Moratilla.
- Equivalencias ER - JFLAP.
- Transparencias de clase.
- Non Deterministic Finite Automaton - Wikipedia
- Automata Theory - Wikipedia
- JFLAP - Duke University | Durham, NC, USA.