Compiladores
Unidad 2: Análisis Léxico

De las Expresiones Regulares a los Autómatas Finitos Deterministas

Manuel Soto Romero

Semestre 2026-1
Facultad de Ciencias UNAM

Esta nota está diseñada a manera de repaso. Si la persona lectora considera que ya domina los conceptos de expresiones regulares, autómatas finitos no deterministas y deterministas, puede saltarse esta lectura sin problema. No obstante, es importante subrayar que los algoritmos y definiciones aquí presentados son indispensables para seguir el curso y participar adecuadamente en sus actividades de evaluación.

Como hemos revisado, el análisis léxico constituye la primera fase de un compilador y se fundamenta en una cadena de equivalencias entre modelos formales. Las expresiones regulares, al proporcionar una forma concisa de especificar lenguajes, se traducen en autómatas finitos no deterministas (con o sin transiciones ε). Estos, a su vez, se transforman en autómatas finitos deterministas mediante la construcción por subconjuntos y finalmente se optimizan a través de algoritmos de minimización. Este recorrido no solo asegura una representación correcta de los lenguajes regulares, sino que también sienta las bases para implementar analizadores léxicos eficientes y prácticos.

1. Expresiones regulares

Las expresiones regulares constituyen el punto de partida en la fase de análisis léxico de un compilador. Con ellas no sólo describimos los lenguajes regulares de manera compacta, sino que también especificamos los patrones de cadenas que, en la práctica, corresponden a los *tokens* de un lenguaje de programación. Estos conceptos ya han sido estudiados previamente en el curso de *Autómatas y Lenguajes Formales*; sin embargo, aquí los retomaremos a manera de repaso, enfatizando su aplicación en la teoría de compiladores. En esta nota emplearemos la notación formal de la teoría de autómatas y lenguajes, en lugar de la notación simplificada de los lenguajes de programación. La razón es doble: primero, porque nos permitirá conectar con la base matemática que garantiza la corrección de los algoritmos; y segundo, porque los procedimientos de traducción hacia autómatas (y posteriormente hacia analizadores léxicos ejecutables) se formulan en este lenguaje formal. En otras palabras, usaremos la notación de teoría de lenguajes formales para *alimentar algoritmos*, no únicamente para describir sintaxis.

Definición 1. (Expresión regular)

*Sea Σ un alfabeto, una **expresión regular** sobre Σ y los conjuntos que ellas denotan son definidos recursivamente como sigue:*

- *$\varnothing$ es una expresión regular y denota el conjunto vacío.*
- *ε es una expresión regular y denota el conjunto $\{\varepsilon\}$.*

- *Para cada a en Σ, a es una expresión regular y denota el conjunto $\{a\}$.*
- *Si r y s son expresiones regulares denotando el lenguaje R y S respectivamente entonces $(r+s), rs, r^\star$ son expresiones regulares que denotan los conjuntos $R \cup S, RS$ y $R^\star$, respectivamente.*

De esta forma, usamos a las expresiones regulares como una simplificación para denotar lenguajes regulares. Recordando que, formalmente hablando, los lenguajes son conjuntos podemos simplificar la definición anterior en la siguiente tabla:

Expresión Regular	Lenguaje
$\varnothing$	$\varnothing$
a	$\{a\}$
ε	$\{\varepsilon\}$
$(r+s)$	$R \cup S$
(rs)	RS
(r^*)	R^*

Los paréntesis sólo son agrupadores. Además, se tiene la siguiente prioridad y asociatividad de operadores:

- La cerradura $\star$ asocia a la izquierda y tiene la mayor prioridad.
- Posteriormente se realiza la concatenación que asocia a la derecha.
- Finalmente se tiene el operador $+$ de alternativa que también asocia por la derecha.

Ejemplo 1

$$0 + 1b^* + a1^*$$

se resuelve en el siguiente orden:

$$(0 + ((1(b^*)) + (a\,(1^*))))$$

Ejemplo 2

Veamos algunos ejemplos de expresiones regulares junto con el lenguaje que reconocen.

- ¿Qué lenguaje reconoce la siguiente expresión regular?

  $$(0+1)^*$$

  Analicemos cada subexpresión:

  * 0 es la expresión regular que denota al lenguaje $\{0\}$
  * 1 es la expresión regular que denota al lenguaje $\{1\}$
  * Entonces $(0+1)$ es el lenguaje $\{0\} \cup \{1\} = \{0, 1\}$
  * De esta forma $(0+1)^* = \{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, ...\}$

Es decir, reconoce el lenguaje de todas las posibles cadenas que se pueden formar como 0 y 1.

$$L_1 = \{w : w \in \Sigma^*\}$$

- ¿Qué lenguaje reconoce la siguiente expresión regular?

$$(a+b)\,c^*$$

Analicemos cada subexpresión:

  * $a = \{a\}$
  * $b = \{b\}$
  * $c = \{c\}$
  * $(a+b) = \{a, b\}$
  * $c^* = \{\varepsilon, c, cc, ccc, cccc, ...\}$
  * $(a+b)\,c^* = \{a, b, ac, bc, acc, bcc, ...\}$

El lenguaje admitido es el de todas las cadenas que comienzan con a o b seguidas de 0 o más símbolos c.

Observación 1

La representación de lenguajes regulares por medio de expresiones regulares no es única. Es posible que haya varias expresiones regulares para el mismo lenguaje. Por ejemplo, $b\,(a+b)^*$ y $b\,(b+a)^*$ representan el mismo lenguaje. Otro ejemplo sería con las expresiones $(a+b)^*$ y $(a^*b^*)^*$.

Ejercicio 1

Indica el lenguaje que reconocen las siguientes expresiones regulares:

1. $(a^*c) + (ab^*)$
2. $(00)^* + (1\,(11)^*)$

Veamos ahora, algunos ejemplos sobre diseño de expresiones regulares.

Ejemplo 3

Encontrar expresiones regulares que representen los siguientes lenguajes, definidos sobre el alfabeto $\Sigma = \{a, b\}$:

1. Lenguaje de todas las cadenas que comienzan con el símbolo b y terminan con el símbolo a.
   - Debemos garantizar que las cadenas inicien con b, lo cual puede ser representado por: b
   - Después del primer símbolo, puede haber cualquier número de as y bs, de forma tal que podemos generar dichas cadenas con la expresión: $(a+b)^*$.
   - Finalmente, la cadena debe terminar con a, con lo cual, la expresión regular más idonea es: a.
   - Concatenando las tres expresiones anteriores tenemos:

Solución: $b\,(a+b)^*\,a$

2. Lenguaje de todas las cadenas que tienen un número par de símbolos.
   - Dado que queremos pares de símbolos necesitamos construir cada uno de éstos, es decir, tanto el primer como el segundo símbolo pueden ser una a o una b y debemos concatenarlos. Esto se puede representar con $(a+b)\,(a+b)$.
   - Con la expresión del punto anterior, estamos construyendo cadenas de longitud 2, pero podemos tener otras longitudes, por ejemplo 0, 2, 4, 6, etc. Dado que ya concatenamos dos símbolos, esto se puede lograr aplicando la estrella de Kleene. Es decir:
     Solución: $((a+b)\,(a+b))^*$

3. Lenguaje de todas las cadenas que tienen un número par de as.
   - Para reconocer cadenas que contengan un número par de as, partamos de un caso fácil, la cadena que tiene 2 de estos símbolos. En este caso, podemos tener alguna de las siguientes formas:
     * Dos símbolos al principio.
     * Dos símbolos al final.
     * Dos símbolos en medio.
     * Dos símbolos en cualquier posición.

     Podríamos considerar todos estos casos y separarlos con una operación de suma, sin embargo, podemos auxiliarnos de la estrella de Kleene para indicar que puede haber cero o más presencias del símbolo b entre las dos a que queremos. Es decir: $b^*ab^*ab^*$.
   - Ahora que tenemos fijado un par de as, podemos generalizarlo como hicimos en el ejemplo anterior con una estrella de Kleene. Es decir: $(b^*ab^*ab^*)^*$.
   - Hasta el punto anterior, pareciera que ya tenemos la expresión regular, sin embargo, recordemos que el cero también es un número par, de forma que una cadena sin as también pertenece al lenguaje. Podemos agregar este paso con una suma, quedando:
     Solución: $(b^*ab^*ab^*)^* + b^*$

Ejemplo 4

Encontrar expresiones regulares que representen los siguientes lenguajes, definidos sobre el alfabeto $\Sigma = \{0, 1\}$

1. Lenguaje de todas las cadenas que tienen exactamente dos ceros.
   - Este es un caso particular de los ejemplos que vimos anteriormente. Dado que queremos exactamente dos ceros, simplemente colocamos los unos correspondientes con su respectiva estrella de Kleene entre los dos símbolos 0:
     Solución: $1^*01^*01^*$

2. Lenguaje de todas las cadenas cuyo penúltimo símbolo, de izquierda a derecha es un 0.
   - Queremos que el penúltimo símbolo sea un 0, de forma que antes de encontrar dicho símbolo, podemos tener cualquier número de ceros y unos, es decir: $(0+1)^*$
   - Posteriormente, aparecerá el símbolo 0, que se reconoce con: 0
   - Y finalmente, después de dichos patrones, podemos tener ya sea otro cero u otro uno, es decir: $(0+1)$
   - Con lo cual, concatenando las 3 expresiones anteriores, tenemos:
     Solución: $(0+1)^*\,0\,(0+1)$

Ejercicio 2

Sea $\Sigma = \{0, 1\}$. Encontrar una expresión regular que represente el lenguaje de todas las cadenas que no tienen dos ceros consecutivos.

Ejercicio 3

Sea $\Sigma = \{a, b, c\}$. Encontrar una expresión regular que represente el lenguaje de todas las cadenas que no contiene la cadena bc.

Ejercicio 4

En el contexto del diseño de un compilador, cada *token* del lenguaje se describe mediante una expresión regular. Consideremos un lenguaje simplificado que incluye identificadores, números enteros y operadores aritméticos básicos. Define expresiones regulares para los siguientes tokens:

1. **Identificador:** una letra seguida de cero o más letras o dígitos.
2. **Número entero:** una secuencia de uno o más dígitos.
3. **Operador aritmético:** cualquiera de los símbolos $+$, $-$, $*$, $/$.

Indica además cómo se construiría el autómata correspondiente al conjunto de expresiones regulares obtenidas, con el fin de reconocer cadenas de entrada y clasificarlas en las categorías léxicas señaladas.

Como paso siguiente en la construcción de un analizador léxico, pasamos de la descripción *declarativa* mediante expresiones regulares a una especificación *operacional* mediante autómatas. En particular, emplearemos AFN-ε porque su construcción es modular y composicional: cada operador de las ER (suma, concatenación y estrella de Kleene) se traduce de manera local a un esquema de autómata que luego conectamos con transiciones ε. Esta traducción (al estilo de Thompson) es el primer eslabón del *pipeline* de compiladores: ER $\Rightarrow$ AFN-ε $\Rightarrow$ AFN $\Rightarrow$ AFD $\Rightarrow$ AFD mínimo, sobre el cual implementaremos finalmente un escáner que produce *tokens* con su categoría léxica y, cuando corresponda, su atributo.

2. Autómatas Finitos No Deterministas con ε-transiciones

Definición 2. (Autómata Finito No Determinita con ε-transiciones)

*Un **autómata finito no determinista con ε-transiciones (AFN-ε)** es un autómata finito no determinista $M = \langle \Sigma, Q, q_0, F, \delta \rangle$ en el que la función de transición está definida como:*

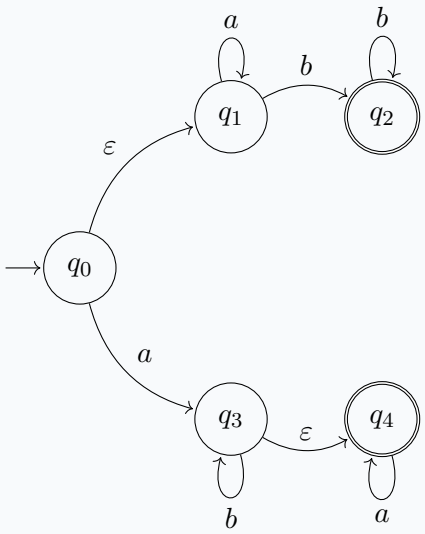
$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$$

*La transición $\delta(q, \varepsilon) = \{p_{i1}, ..., p_{in}\}$, llamada **transición ε** o **transición espontánea,** tiene el siguiente significado computacional: estando en el estado q, el autómata puede cambiar a uno cualquier de los estados $p_{i1}, ..., p_{in}$, independientemente del símbolo leído. Dicho de otra manera, las transiciones ε permiten al autómata cambiar internamente de estado sin consumir el símbolo leído.*

En el diagrama del autómata, las transiciones ε dan lugar a aristas con etiquetas ε. Una cadena de entrada w es aceptada por un AFN-ε si existe por lo menos una trayectoria, desde el estado q_0, cuyas etiquetas son exactamente los símbolos de w, intercalados con cero, uno o más εs.

En los autómatas AFN-ε, al igual que en los AFN, puede haber múltiples camnios para una misma cadena de entrada. Veamos un ejemplo.

Ejemplo 5



Ejemplos de cadenas aceptadas por este autómata son:

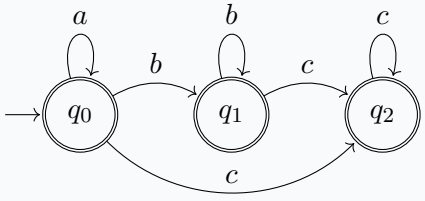
- aab
- $abaa$
- $abbaa$

Los AFN-ε nos dan más libertad en el diseño de autómatas, especialmente cuando tenemos varias concatenaciones.

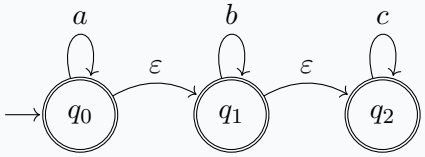
Ejemplo 6

$\Sigma = \{a, b, c\}$. $L = a^*b^*c^*$

AFN que acepta a L:



AFN-ε que acepta a L:

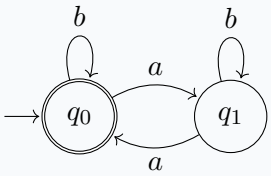


Este autómata es muy similar a la expresión regular $a^*b^*c^*$: las concatenaciones son reemplazadas por transiciones ε.

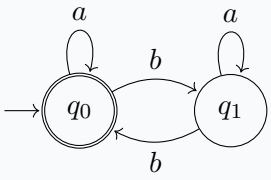
Ejemplo 7

$\Sigma = \{a, b\}$. L =lenguaje de todas las cadenas sobre Σ que tienen un número par de as o un número par de bs.

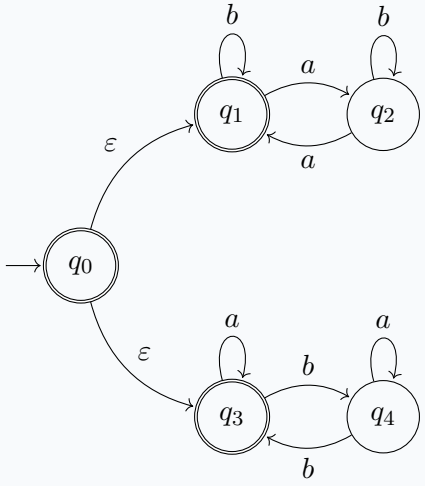
AFD que acepta el lenguaje de las cadenas con un número par de as:



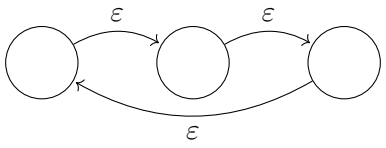
AFD que acepta el lenguaje de las cadenas con un número par de bs:



AFN-ε que acepta el lenguaje de las cadenas con un número par de as o número par de bs:



A diferencia de los AFD y los AFN, en los AFN-ε pueden existir cómputos infinitos, es decir, cómputos que nunca terminan. Esto puede suceder si el autómata ingresa a un estado desde el cual haya varias transiciones ε encadenadas que regresen al mismo estado. Por ejemplo, el siguiente autómata:



Ejercicio 5

Diseñar un AFN-ε que acepte el lenguaje $(ab + b)^* ab^*$, sobre $\Sigma = \{a, b\}$.

Teorema de Kleene

En los cursos de Autómatas y Lenguajes Formales, se muestra la equivalencia computacional entre los AFD, AFN y AFN-ε, lo cual puede ser descrito de la siguiente forma:

$$\text{AFD} \equiv \text{AFN} \equiv \text{AFN-}\varepsilon$$

Esto quiere decir que para cada uno de estos tipos de autómatas se pueden construir otro tipo de autómatas equivalentes. Por lo tanto, estos tres modelos aceptan exactamente los mismos modelos. El Teorema de Kleene establece que tales lenguajes son precisamente los lenguajes regulares.

Teorema de Kleene

Un lenguaje es regular si y sólo si es aceptado por un autómata finito (AFD o AFN o AFN-ε).

Nuevamente, la demostración de este teorema queda fuera de los alcances del curso. Sin embargo, recordemos que la misma demostración nos permite analizar la equivalencia que hay entre las expresiones regulares y los autómatas finitos (en ambos sentidos) por medio de dos partes: La Parte I demuestra que dada una expresión regular R sobre un alfabeto Σ, se puede construir un AFN-ε M tal que el lenguaje aceptado por M sea exactamente el lenguaje representado por R. Por otro lado, la Parte II demuestra que para todo AFN M existe una expresión regular R tal que $L(M) = R$. Para los fines de este curso nos enfocaremos en la Parte I de la demostración desde un enfoque algorítmico.

De esta forma, a partir de cada una de las construcciones para expresiones regulares, podemos construir también autómatas que acepten los lenguajes que representan cada una de éstas. Veamos cada uno de estos casos. Es importante notar que el autómata obtenido será un AFN-ε.

Caso 1 ($\varnothing$)

El siguiente autómata acepta el lenguaje $\varnothing$, es decir el lenguaje representado por la expresión regular $R = \varnothing$.

Autómata

Caso 2 (ε)

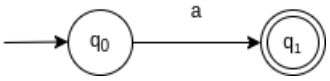
El siguiente autómata acepta el lenguaje $\{\varepsilon\}$, es decir, el lenguaje representado por la expresión regular $R = \varepsilon$.

Autómata

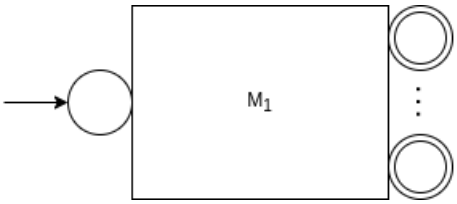
Caso 3 (a)

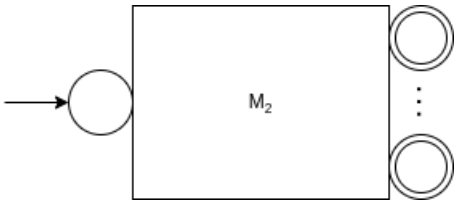
El siguiente autómata acepta el lenguaje $\{a\}$, $a \in \Sigma$, es decir, el lenguaje representado por la expresión regular $R = a$.

Autómata



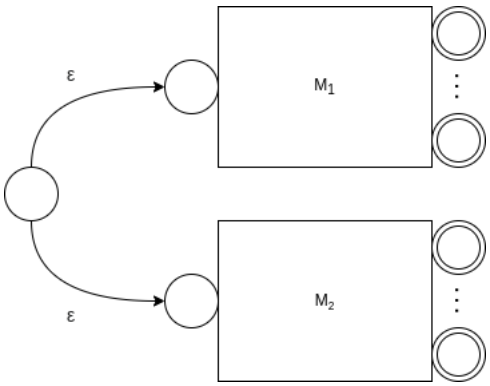
Para los siguientes casos, supóngase que para las expresiones regulares R y S existen AFN-ε M_1 y M_2 tales que $L(M_1) = L(R)$ y $L(M_2) = S$. Esquemáticamente vamos a presentar los autómatas M_1 y M_2 de la siguiente forma:





Caso 4 ($R + S$)

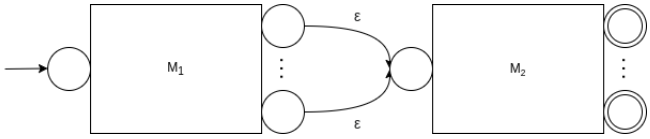
Autómata



Los autómatas M_1 y M_2 se conectan en paralelo y los estados finales del nuevo autómata son los estados de M_1 junto con los de M_2.

Caso 5 (RS)

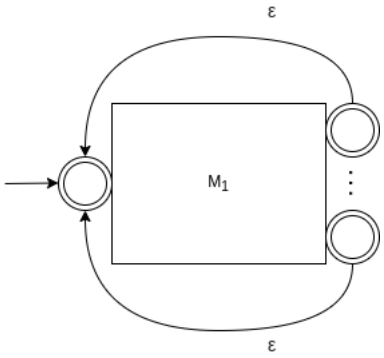
Autómata



Los autómatas M_1 y M_2 se conectan en serie y los estados finales del nuevo autómata son únicamente los estados finales de M_2.

Caso 6 (R^*)

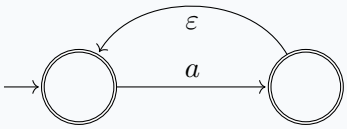
Autómata



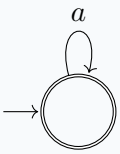
Los estados finales del nuevo autómata son los estados finales de M_1 junto con el estado inicial.

Ejemplo 9

De acuerdo con las construcciones presentadas, a continuación veremos algunos ejemplos, sin embargo antes de empezar, analicemos el siguiente autómata que acepta el lenguaje a^*:

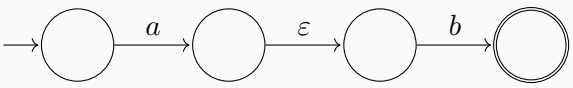


Para simplificar las próximas construcciones, usaremos en su lugar el ciclo de un estado:

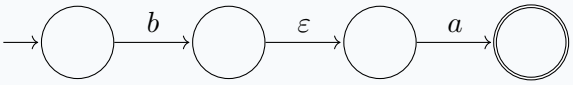


Vamos a utilizar el procedimiento anterior para construir un AFN-ε que acepte el lenguaje $a^*(ab+ba)^* + a(b+a^*)$ sobre el alfabeto $\{a, b\}$.

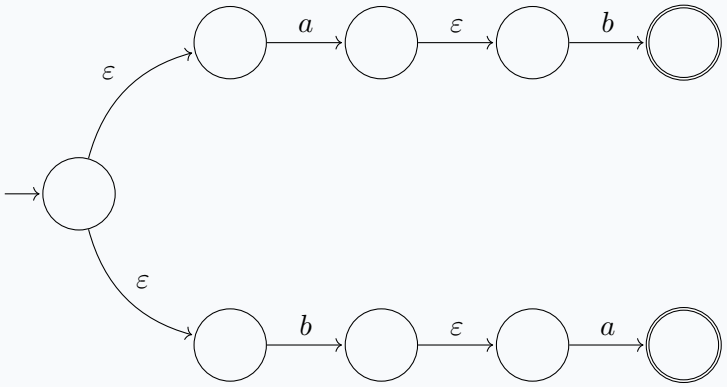
- Autómata que acepta ab:



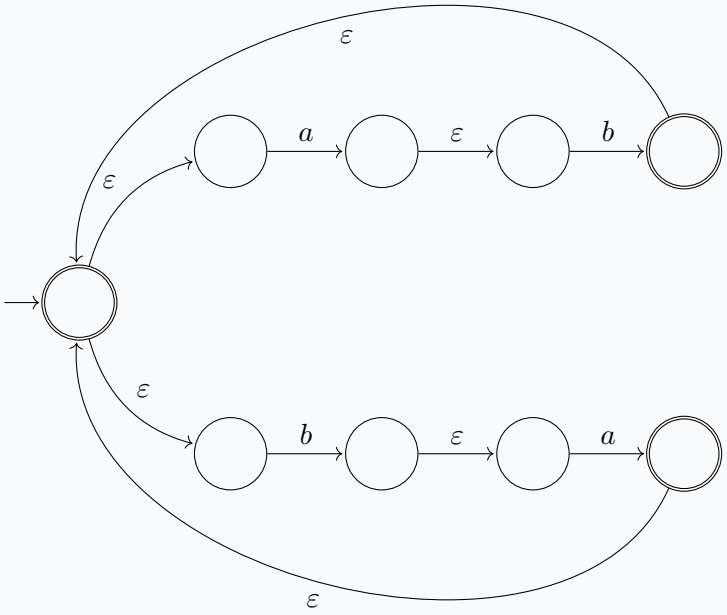
- Autómata que acepta ba:



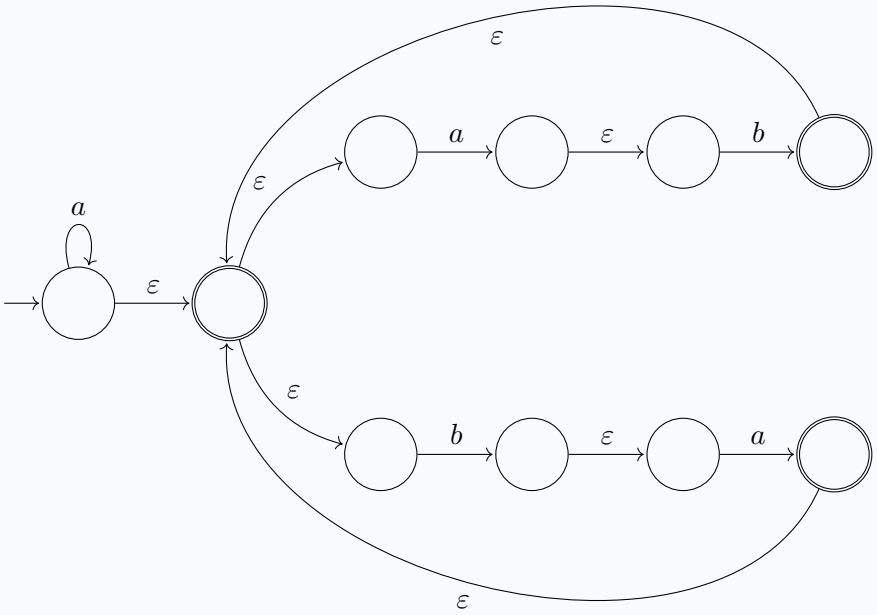
- Autómata que acepta $ab + ba$:



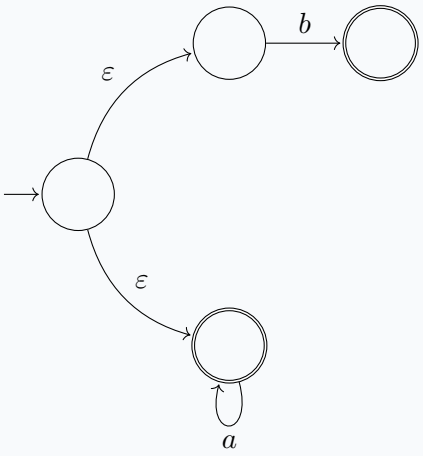
- Autómata que acepta $(ab + ba)^*$:



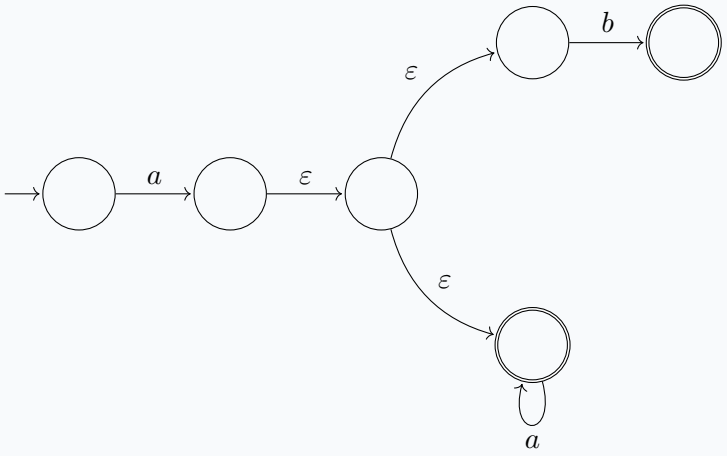
- Autómata que acepta $a^*\,(ab + ba)^*$:



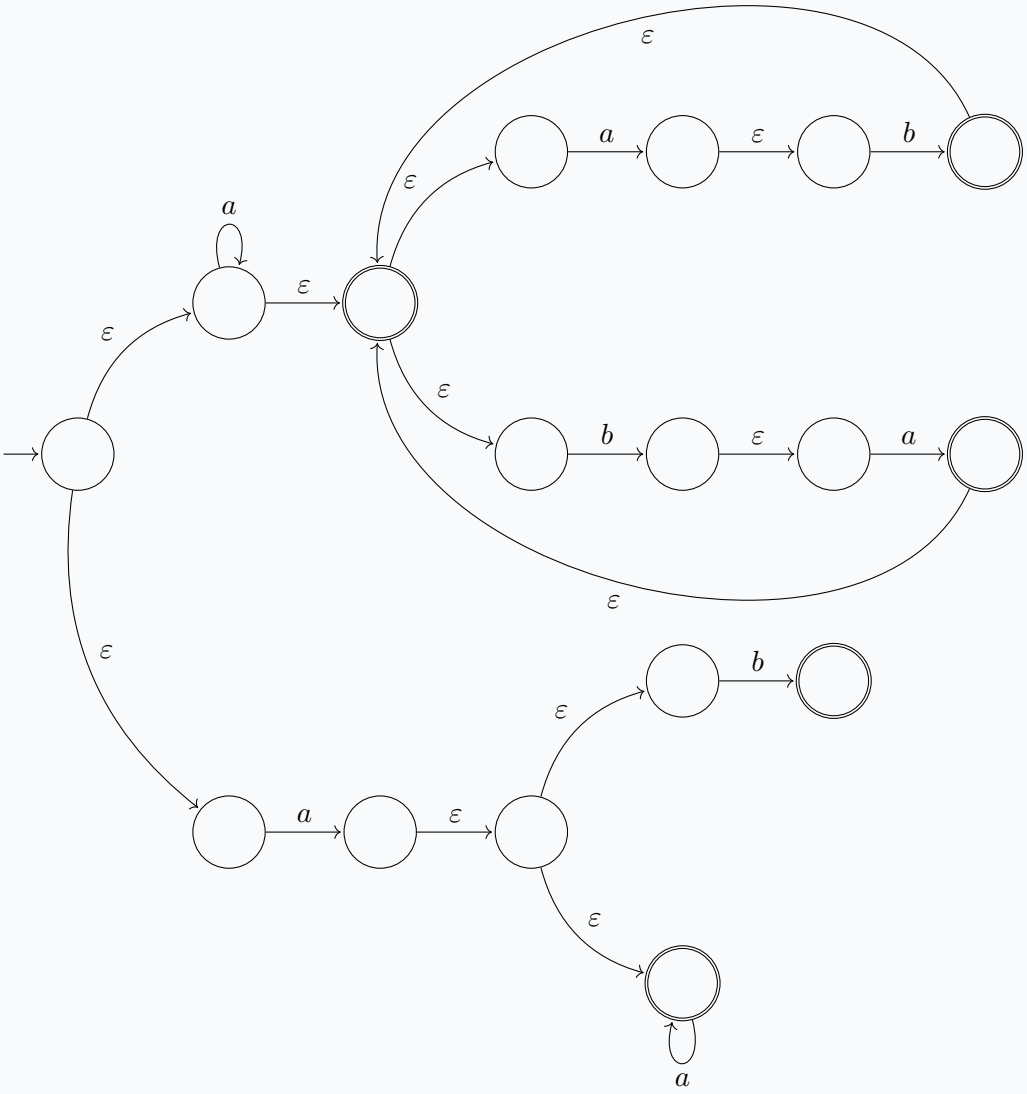
- Autómata que acepta $b + a^*$:



- Autómata que acepta $a\,(b + a^*)$:



- Autómata que acepta $a^* (ab + ba)^* + a (b + a^*)$:



Ejercicio 6

Diseñar autómatas AFN-ε que acepten los siguientes lenguajes sobre el alfabeto $\Sigma = \{a, b, c\}$:

- $a^* (b + ab^* + ab^*a) c^* + (a + b) (a + ac)^*$
- $c^*a (a + ba)^* (abc)^* + c^* (a + cb^*c)$

Ejercicio 7

Retoma las expresiones regulares del **Ejercicio 4** (identificadores, números enteros y operadores aritméticos) y **construye un único AFN-ε** que reconozca la *unión* de las tres categorías léxicas.

Indicaciones.

a) Construye, por separado, el AFN-ε para cada ER (usa los esquemas de suma, concatenación y estrella vistos en la sección).
b) Conecta los tres autómatas en paralelo desde un nuevo estado inicial mediante transiciones ε, y conserva como aceptores los finales de cada subautómata.
c) *Etiqueta* cada estado de aceptación con la categoría léxica correspondiente: ID, NUM, OP.

Ejercicio 8

Este es un **ejercicio de investigación** sobre las políticas que guían a los analizadores léxicos en compiladores reales. En particular, nos interesan dos criterios fundamentales para resolver ambigüedades:

- ***Longest match* (emparejamiento más largo):** si dos o más patrones pueden reconocer un prefijo de la cadena de entrada, se elige siempre aquel que consuma la mayor cantidad de caracteres.
- ***Maximal munch* (consumo máximo):** variante práctica del anterior, usada en compiladores reales: el analizador léxico avanza leyendo la entrada hasta que no es posible extender más la coincidencia con ningún *token* válido, y entonces clasifica el fragmento reconocido.

Actividad. Explica con tus palabras qué diferencias existen entre estos dos criterios y da un ejemplo (real o inventado) en el que se vea qué ocurriría si no se aplicara ninguna de estas reglas en un compilador.

3. Autómatas Finitos No Deterministas

En esta sección mostraremos que el modelo AFN-ε es computacionalmente equivalente al modelo AFN. Es decir, se pueden quitar las transiciones ε y añadir nuevas que las simulen sin alterar el lenguaje aceptado. Comencemos con la siguiente observación.

Observación 2

Un AFN puede ser visto como un AFN-ε en el que, simplemente hay *cero* transiciones ε.

Con esta observación, tenemos el siguiente teorema:

Teorema 1

Dado un AFN-ε M, se puede construir un AFN M' equivalente a M tal que $L(M) = L(M')$.

Como mencionamos al inicio, la demostración del teorema queda fuera de los alcances de este curso, por lo que en su lugar, presentaremos un método de tranformación. Necesitamos primero que nada definir el concepto de *ε-cerradura*.

Definición 3. (ε-cerradura)

*Para un estado q, la **ε-cerradura**, denotada $ECLOSURE(q)$ es el conjunto de estados de un AFN-ε a los que se puede llegar desde q por 0, 1 o más transiciones ε. Nótar además que $ECLOSURE(q) \neq \delta(q, \varepsilon)$. Por definición $q \in ECLOSURE(q)$. De esta forma, la ε-cerradura de un conjunto de estados $\{q_1, ..., q_k\}$ se define por:*

$$ECLOSURE(\{q_1, ..., q_k\}) = ECLOSURE(q_1) \cup ... \cup ECLOSURE(q_k)$$

Además, $ECLOSURE(\varnothing) = \varnothing$.

De esta manera, podemos construir un AFN $M' = \langle \Sigma, Q, q_0, F', \delta' \rangle$ a partir del AFN-ε $M = \langle \Sigma, Q, q_0, F, \delta \rangle$, mediante la siguiente función de transición:

$$\begin{array}{rcl} \delta' : Q \times \Sigma & \rightarrow & \mathcal{P}(Q) \\ (q, a) & \rightarrow & \delta'(q, a) = ECLOSURE(\delta(ECLOSURE(q), a)) \end{array}$$

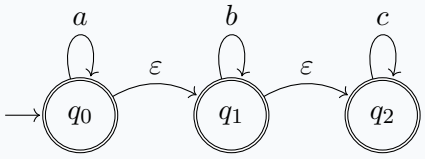
M' simula así las transiciones ε de M teniendo en cuenta todas las posibles trayectorias. F' se define como:

$$F' = \{q \in Q : ECLOSURE(q) \text{ contiene al menos un estado de aceptación}\}$$

Es decir, los estados de aceptación de M' incluyen los estados de aceptación de M y aquellos estados desde los cuales se puede llegar a un estado de aceptación por medio de una o más transiciones ε.

Ejemplo 8

Vamos a ilustrar el método anterior con el autómata del Ejemplo 6 al que llamaremos M:



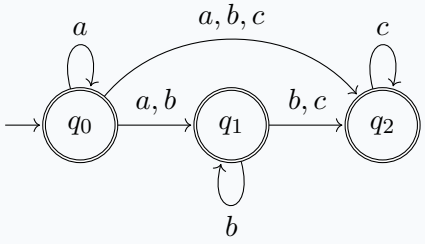
Recordemos que $L(M) = a^*b^*c^*$. Comencemos entonces calculando la ε cerradura de cada estado de nuestro autómata.

$$\begin{aligned} ECLOSURE\,(q_0) &= \{q_0, q_1, q_2\} \\ ECLOSURE\,(q_1) &= \{q_1, q_2\} \\ ECLOSURE\,(q_2) &= \{q_2\} \end{aligned}$$

A partir de aquí construimos las transiciones correspondientes, abreviamos *ECLOSURE como EC*:

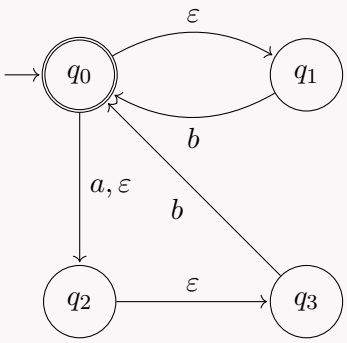
$$\begin{aligned} \delta'\,(q_0, a) &= EC\,(\delta\,(EC\,(q_0), a)) = EC\,(\delta\,(\{q_0, q_1, q_2\}, a)) = EC\,(\{q_0\}) = \{q_0, q_1, q_2\} \\ \delta'\,(q_0, b) &= EC\,(\delta\,(EC\,(q_0), b)) = EC\,(\delta\,(\{q_0, q_1, q_2\}, b)) = EC\,(\{q_1\}) = \{q_1, q_2\} \\ \delta'\,(q_0, c) &= EC\,(\delta\,(EC\,(q_0), c)) = EC\,(\delta\,(\{q_0, q_1, q_2\}, c)) = EC\,(\{q_2\}) = \{q_2\} \\ \delta'\,(q_1, a) &= EC\,(\delta\,(EC\,(q_1), a)) = EC\,(\delta\,(\{q_1, q_2\}, a)) = EC\,(\varnothing) = \varnothing \\ \delta'\,(q_1, b) &= EC\,(\delta\,(EC\,(q_1), b)) = EC\,(\delta\,(\{q_1, q_2\}, b)) = EC\,(\{q_1\}) = \{q_1, q_2\} \\ \delta'\,(q_1, c) &= EC\,(\delta\,(EC\,(q_1), c)) = EC\,(\delta\,(\{q_1, q_2\}, c)) = EC\,(\{q_2\}) = \{q_2\} \\ \delta\,(q_2, a) &= EC\,(\delta\,(EC\,(q_2), a)) = EC\,(\delta\,(\{q_2\}, a)) = EC\,(\varnothing) = \varnothing \\ \delta\,(q_2, b) &= EC\,(\delta\,(EC\,(q_2), b)) = EC\,(\delta\,(\{q_2\}, b)) = EC\,(\varnothing) = \varnothing \\ \delta\,(q_2, c) &= EC\,(\delta\,(EC\,(q_2), c)) = EC\,(\delta\,(\{q_2\}, c)) = EC\,(\{q_2\}) = \{q_2\} \end{aligned}$$

El conjunto de estados finales es $\{q_0, q_1, q_2\}$ dado que las cerraduras de todos los estados contienen estados finales. De esta forma, el autómata AFN queda como:



Ejercicio 9

Construye un AFN equivalente al siguiente AFN-ε:



Ejercicio 10

A partir del **AFN-ε** que construiste en el **Ejercicio 7** (que reconoce la unión de ID, NUM y OP), **construye un AFN equivalente** *eliminando* las transiciones ε mediante la ε-cerradura.

Indicaciones.

a) Calcula $ECLOSURE(q)$ para cada estado q del AFN-ε.

b) Define la nueva función de transición

$$\delta'(q, a) = ECLOSURE\big(\,\delta(ECLOSURE(q), a)\,\big), \quad a \in \Sigma.$$

c) Determina el nuevo conjunto de estados de aceptación

$$F' = \{\, q \in Q \;:\; ECLOSURE(q) \cap F \neq \varnothing \,\}.$$

d) Conserva el mismo alfabeto Σ y estado inicial q_0.

La equivalencia entre los modelos de autómatas nos permite avanzar un paso más en el proceso de construcción de analizadores léxicos. Si bien los AFN (con o sin ε-transiciones) son más fáciles de especificar y resultan convenientes para construirlos a partir de expresiones regulares, no son adecuados para la implementación directa de un escáner, pues su naturaleza no determinista implica explorar múltiples trayectorias en paralelo. Por esta razón, el siguiente paso consiste en transformar un AFN en un Autómata Finito Determinista (AFD), el cual ofrece una máquina más eficiente y práctica para la ejecución real: cada estado tiene una única transición por símbolo, lo que garantiza un comportamiento determinista y lineal en tiempo de ejecución. En la sección que sigue estudiaremos cómo realizar esta transformación.

4. Autómatas Finitos Deterministas

El poder de un autómata se mide en términos de la habilidad para aceptar lenguajes. Recordemos que un AFD es un caso especial de un AFN, de modo que todo lo que un AFD puede hacer, un AFN también lo puede hacer, de hecho para cada AFN existe un AFD que acepta el mismo lenguaje. Además, como seguramente te habrás dado cuenta, es más sencillo construir, especificar y comprender un AFN debido a que son más concisos. En general, los AFNs pueden ser usados para capturar patrones en procesadores de texto, pero optamos por los AFD cuando deseamos encontrar dichos patrones.

Teorema 2

Para cada AFN N, existe un AFD M con $L(N) = L(M)$.

Nuevamente la demostración de este teorema queda fuera de los alcances de este curso, por lo que en su lugar, presentaremos el proceso para transformar un AFN en un AFD por medio de la llamada construcción por subconjuntos:

1. Construir una tabla de transiciones a partir del conjunto potencia, que muestre los estados a donde se puede llegar con cada estado de dichos conjuntos. Al conjunto que tenga como único elemento al estado inicial, lo marcamos como final y a todos los conjunto que contengan al estado final original, los marcamos también como finales.

2. Eliminar estados que no sean alcanzables.

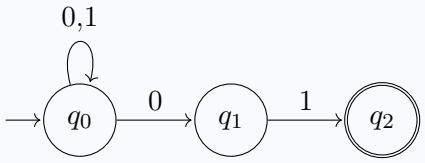
3. Renombrar los estados.

Recordemos que el conjunto potencia se define como el conjunto formado por todos los subconjuntos de un conjunto dado. Por ejemplo, dado el conjunto $A = \{1, 2, 3\}$, entonces

$$\mathcal{P}(A) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

Ejemplo 9

Veamos cómo aplicar este algoritmo con el siguiente autómata



1. Construimos la tabla de transiciones a partir del conjunto potencia

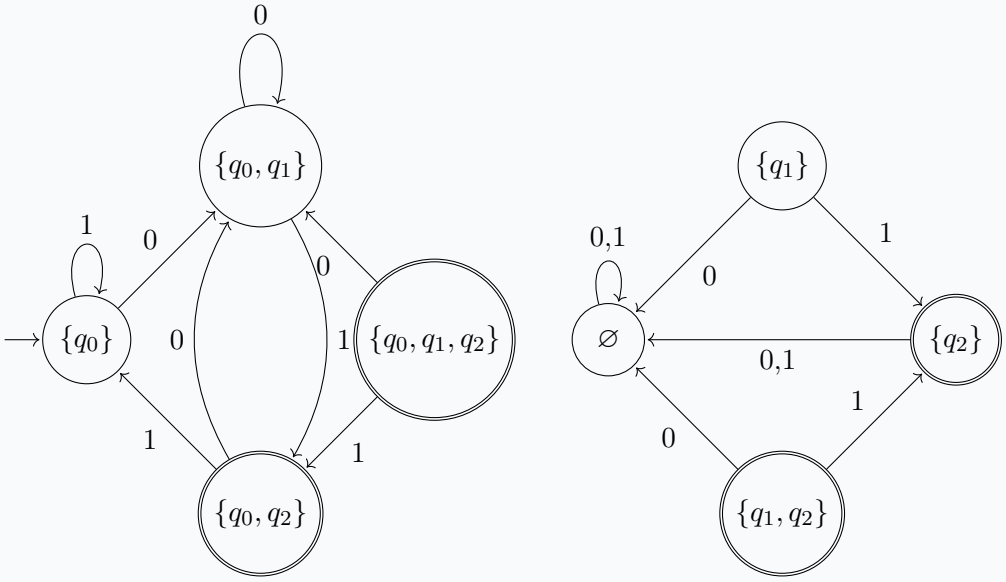
   - Inicialización:

		0	1
	$\varnothing$		
$\rightarrow$	$\{q_0\}$		
	$\{q_1\}$		
F	$\{q_2\}$		
	$\{q_0, q_1\}$		
F	$\{q_0, q_2\}$		
F	$\{q_1, q_2\}$		
F	$\{q_0, q_1, q_2\}$		

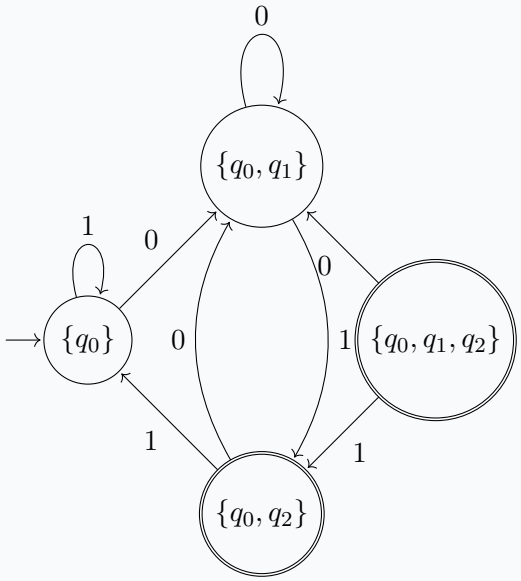
   - Llenamos la tabla con los resultados de transitar desde cada uno de los estados.

		0	1
	$\varnothing$	$\varnothing$	$\varnothing$
$\rightarrow$	$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
	$\{q_1\}$	$\varnothing$	$\{q_2\}$
F	$\{q_2\}$	$\varnothing$	$\varnothing$
	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
F	$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
F	$\{q_1, q_2\}$	$\varnothing$	$\{q_2\}$
F	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

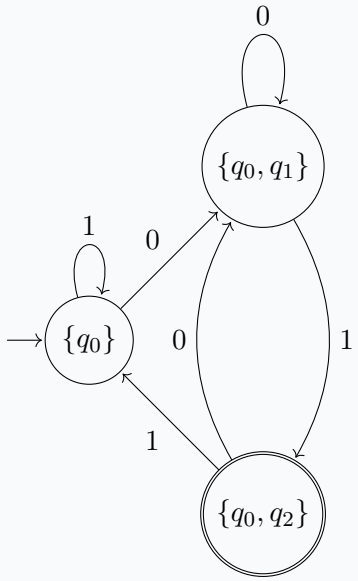
   - Construimos el autómata resultante



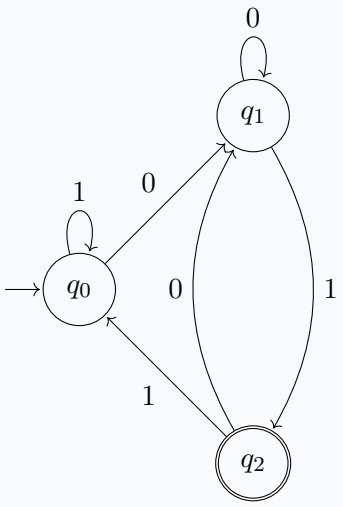
- Notamos que nuestro autómata se encuentra *disconexo*, esto ocurre debido a que tenemos estados inalcanzables. Partiendo desde el estado inicial detectamos estos estados. Claramente el subautómata de la derecha es inalcanzable, por lo tanto lo eliminamos.



  De la misma forma, el estado $\{q_0, q_1, q_2\}$ es inalcanzable pues no hay ningún estado que llegue a él. Lo eliminamos también.



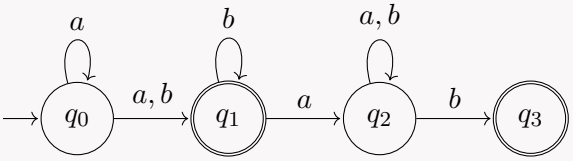
- Finalmente, renombramos los estados.



Notamos que el autómata resultante es un AFD y que además reconoce el mismo lenguaje: cadenas que terminen con 01.

Ejercicio 11

Mediante construcción por subconjuntos, transforma el siguiente AFN a AFD. ¿Qué lenguaje reconoce?



Ejercicio 12

A partir del **AFN** que obtuviste en el **Ejercicio 10** (ya sin transiciones ε), **construye un AFD equivalente** aplicando la **construcción por subconjuntos**.

Indicaciones.

a) Sea $M' = \langle \Sigma, Q, q_0, F', \delta' \rangle$ el AFN del Ejercicio 10. Define el AFD $D = \langle \Sigma, \mathcal{P}(Q), \{q_0\}, F_D, \Delta \rangle$ con:

$$\Delta(S, a) = \bigcup_{q \in S} \delta'(q, a) \quad \text{y} \quad F_D = \{\, S \subseteq Q \mid S \cap F' \neq \varnothing \,\}.$$

b) Considera únicamente los **subconjuntos alcanzables** desde $\{q_0\}$ al iterar Δ sobre los símbolos de Σ.

c) Marca como **estados de aceptación** aquellos subconjuntos que contengan al menos un estado de F'.

d) **Renombra** los estados-conjunto con etiquetas simples (p. ej., $A, B, C, \ldots$) una vez obtenidos.

e) Verifica la equivalencia con M' mostrando algunos ejemplos de cadenas aceptadas y rechazadas en ambos modelos.

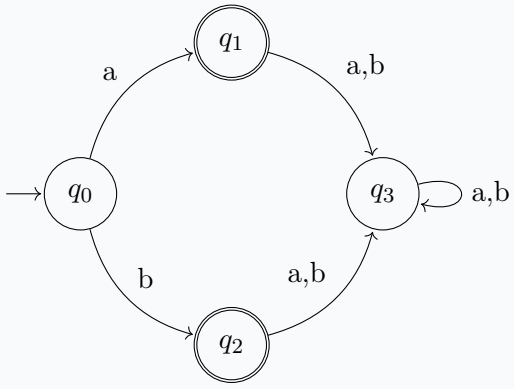
5. Minimización

Al concluir la construcción por subconjuntos, obtenemos un AFD funcional pero no necesariamente óptimo: pueden quedar estados redundantes o estructuras que, aunque correctas, encarecen memoria y saltos en tiempo de ejecución. En un compilador real, donde el analizador léxico se ejecuta sobre cada símbolo de la entrada, estos costos se amplifican. Por ello, el paso natural tras la determinización es la minimización del AFD: colapsar estados indistinguibles preservando el lenguaje reconocido. El resultado es un autómata más pequeño y predecible, que reduce la huella de memoria del *scanner*, simplifica tablas de transición y mejora el rendimiento del ciclo crítico de lectura de *tokens*.

Al diseñar AFD, podemos tener distintas versiones que reconocen el mismo lenguaje. Esto en un principio, no representa ningún problema, sin embargo, podemos llegar a tener casos donde el autómata tenga estados y/o transiciones que en realidad no son del todo esenciales.

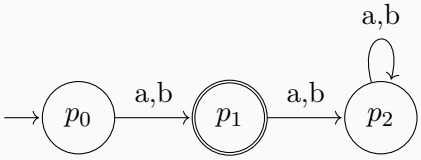
Ejemplo 10

Analicemos el siguiente AFD, ¿qué lenguaje reconoce?



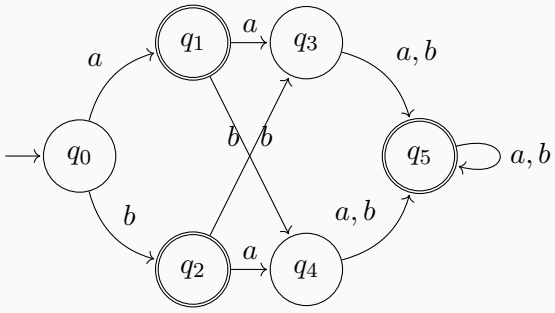
Es fácil ver que el lenguaje admitido por este autómata es $\{a, b\}$, sin embargo, podemos notar a partir del estado q_0, que sin importar si leemos una a o una b, vamos a un estado final, con lo cual, parece ser que tener dos estados no es del todo necesario. De la misma forma, en ambos casos, una vez que llegamos a un estado de aceptación y leer cualquier otro símbolo del alfabeto, llegamos al estado q_3 lo cual reafirma el hecho de que no es necesario tener estado diferentes.

A partir del análisis anterior, podemos modificar el autómata como sigue:



Ejemplo 11

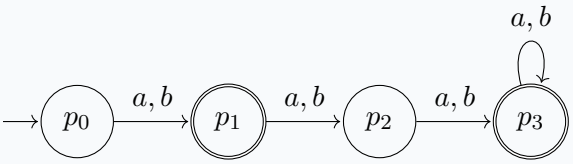
Veamos ahora el siguiente AFD, ¿qué lenguaje reconoce?



Los estados q_1 y q_2 nos hacen notar que, al igual que el autómata anterior, el lenguaje reconocido es $\{a, b\}$, sin embargo, al seguir analizando las transiciones, vemos que también acepta cadenas con una longitud de al menos 3. Es decir, el lenguaje aceptado es:

$$\{a, b\} \cup \{w \in \{a, b\}^* : |w| \text{ es al menos } 3\}$$

Vemos que tanto q_1 como q_2 se pueden colapsar, de la misma forma que en el ejemplo anterior. Por otro lado, los estados q_3 y q_4 son prácticamente idénticos por lo que también podemos colapsarlos, dejando el siguiente AFD:



En ambos ejemplos hemos *colapsado* estados que tienen el mismo comportamiento, es decir, son *equivalentes*. Esta noción de equivalencia y el proceso de decisión sobre si colapsar o no un par de estados, puede realizarse algorítmicamente. Veamos la definición de equivalencia:

Definición 4. (Relación de equivalencia entre estados)

Dado un AFD $\langle Q, \Sigma, \delta, q_0, F \rangle$ *decimos que dos estados* p *y* q *están relacionados si y sólo si*

$$\forall x \in \Sigma^*, \widehat{\delta}(p, x) \in F \leftrightarrow \widehat{\delta}(q, x) \in F$$

Denotamos esto como

$$p \approx q$$

Intuitivamente, esta definición nos indica tres cosas importantes:

1. No importa qué cadena tomemos del alfabeto, si llegamos a un estado final partiendo de p o q es equivalente.

2. No necesariamente p y q deben ser iniciales, puede ser cualquier estado. tampoco deben ser finales.

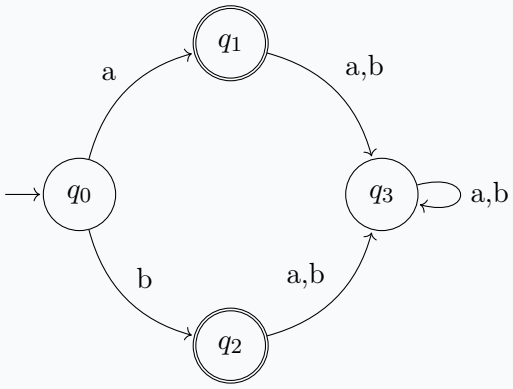
3. Tienen el mismo comportamiento bajo todas las cadenas posibles.

Partiendo de esta definición, se define el *algoritmo de minimización* como sigue:

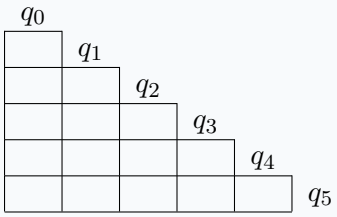
1. Escribir una tabla para cada uno de los pares de estados $\{p, q\}$, inicialmente sin marca alguna.
2. Marcar la celda correspondiente a $\{p, q\}$ si $p \in F$ y $q \notin F$ o viceversa.
3. Repetir lo siguiente hasta que no se presenten cambios: Si existe un par sin marca $\{p, q\}$ tal que $\{\delta(p, a), \delta(q, a)\}$ está marcado por alguna $a \in \Sigma$, entonces, marcamos la celda $\{p, q\}$.
4. Al terminar $p \approx q$ si y sólo si $\{p, q\}$ no tiene marca.

Ejemplo 12

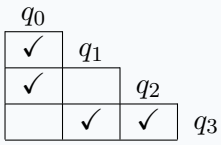
Veamos cómo aplicar el algoritmo con el AFD revisado anteriormente:



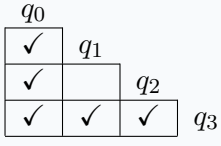
Construimos la tabla para cada par de estados en el autómata.



Para cada celda, marcarla si uno de los estados es final y el otro no.



Ahora revisamos las celdas vacías y analizamos las transiciones con cada uno. Por ejemplo, analicemos $\{q_0, q_3\}$. Tenemos como alfabeto $\{a, b\}$, de forma tal que $\delta(q_0, a) = q_1$, $\delta(q_3, a) = q_3$, $\delta(q_0, b) = q_2, \delta(q_3, b) = q_3$, con lo cual debemos verificar si $\{q_1, q_3\}$ o $\{q_2, q_3\}$ están marcados. Si están marcados, marcamos la celda $\{q_0, q_3\}$. En este caso vemos que $\{q_1, q_3\}$ sí está marcada, por lo tanto marcamos la celda original.



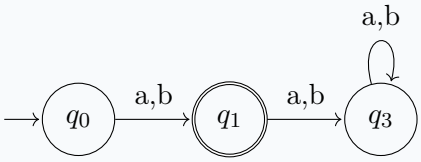
Realizando el mismo análisis vemos que no existe una celda marcada, por lo tanto dejamos dicha celda vacía y dado que el proceso se repite hasta que no haya cambios. Nuestra tabla final queda como

	q_0	q_1	q_2
q_1	✓		
q_2	✓		
q_3	✓	✓	✓

En este caso, nuestra única celda vacía fue $\{q_1, q_2\}$ con lo cual concluimos que

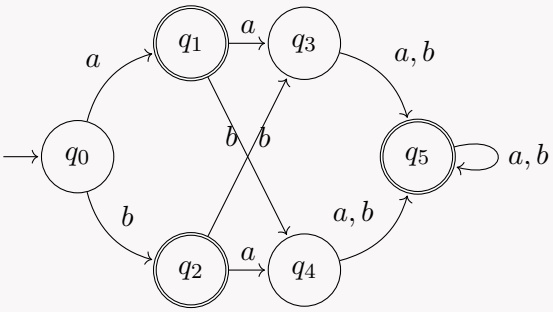
$$q_1 \approx q_2$$

y podemos colapsar entonces dichos estados, por ejemplo, cambiando todos las transiciones que llegaban y salían de q_2 a q_1:



Ejercicio 13

Minimiza el AFD revisado anteriormente, por medio del algoritmo descrito:



Ejercicio 14

Parte del **AFD** obtenido en el **Ejercicio 12** y **constrúyelo mínimo**. Para ello:

1. Elimina estados *inalcanzables* desde el estado inicial, si los hubiera.
2. Determina y colapsa *estados equivalentes* (por tabla de pares distinguibles o por particiones sucesivas).
3. Identifica explícitamente el *estado sumidero* (si existe) y justifica su presencia o ausencia.
4. Renombra los estados del autómata mínimo y presenta su función de transición completa.
5. Compara el número de estados antes y después de minimizar e indica dos ventajas concretas de usar el AFD mínimo en un analizador léxico.

6. Conclusión

En esta nota repasamos el recorrido completo desde las expresiones regulares hasta los autómatas finitos deterministas, enfatizando la importancia de cada transformación intermedia en la construcción de analizadores léxicos. Si bien los AFN y los AFN-ε son más convenientes para la especificación modular, los AFD son los modelos que finalmente permiten implementar analizadores eficientes en compiladores reales.

La siguiente tabla resume las complejidades asintóticas de los algoritmos revisados:

Algoritmo	**Complejidad Asintótica**
Traducción ER $\rightarrow$ AFN-ε (Thompson)	$O(n)$
Eliminación de transiciones ε	$O(n^2)$
Construcción por subconjuntos (AFN $\rightarrow$ AFD)	$O(2^n)$ (peor caso)
Minimización de AFD	$O(n \log n)$
Consumo de una cadena por el AFD	$O(m)$

En conclusión, aunque los algoritmos de construcción pueden tener costos elevados en el peor caso, una vez obtenido el AFD mínimo, el escaneo de cadenas de entrada se realiza en tiempo lineal respecto a su longitud, es decir, en $O(m)$.

Referencias

[1] R. De Castro, *Notas de Clase de Teoría de la Computación: Lenguajes, autómatas, gramáticas*, Apuntes de curso, 2004.

[2] N. S. Hernández, *Notas de Clase para el curso de Autómatas y Lenguajes Formales*, Apuntes de curso, 2020.

[3] J. E. Hopcroft, R. Motwani y J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3.ª ed. Addison Wesley, 2007.

[4] F. Miranda, A. L. Reyes y L. d. C. González, *Notas de Clase para el curso de Autómatas y Lenguajes Formales*, Apuntes de curso, 2020.