TEMA 3:

Autómatas Finitos

Autómatas y Lenguajes Formales.
Grado en Informática (2° curso).

Dpto. de Ingeniería de la Información y las Comunicaciones.

UNIVERSIDAD DE MURCIA

Índice general

1. Autómatas finitos deterministas

Un *autómata finito* (AF) es un formalismo matemático que se usa como modelo en distintas fases de especificación, diseño o análisis de sistemas que evolucionan cambiando de un estado a otro como reacción a determinados símbolos que reciben de una fuente de entrada. Dichos símbolos pueden representar caracteres que se leen de una cadena de texto en una aplicación software o eventos que se procesan en secuencia, como pulsaciones de botones en un ascensor, señales discretas que provienen de sensores, eventos de pulsación de teclas o ratón en una interfaz gráfica de usuario, etc.

Hay varios tipos de autómatas finitos, adecuados para diferentes aplicaciones. En esta asignatura dedicamos especial interés a los *autómatas finitos clásicos*, que se usan como *modelo para el diseño de algoritmos de procesamiento de cadenas de patrón regular*. Estos algoritmos resuelven problemas diversos como búsqueda de cadenas en un texto, sustitución o conversión de formato de cadenas y el problema esencial es el *problema de la validación*, que consiste en comprobar si una cadena tiene un formato válido según lo especificado en la descripción de cierto lenguaje. El problema de validación de formato de cadenas es equivalente al problema teórico de *comprobar la pertenencia de una cadena a un lenguaje.*

En esta sección introducimos un tipo de autómata finito llamado **autómata finito determinista** (AFD). Este tipo de autómatas es el más adecuado para la implementación de algoritmos de procesamiento de cadenas, debido a que el proceso es muy eficiente a partir del diseño con estos autómatas, como veremos.

Un ejemplo de aplicación

Ejemplo 1 *Consideramos las cadenas que representan números decimales sin signo, con parte entera y fraccionaria obligatoria y separadas por un punto. Todas esas cadenas forman un lenguaje, que llamamos L_{ef}, donde el alfabeto del lenguaje contiene los símbolos que se usan para escribir estas cadenas y es $V_{ef} = \{.\} \cup V_{dig} = \{., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. El lenguaje puede describirse matemáticamente por comprensión como:*

$$L_{ef} = \{x.y \mid x, y \in V_{dig}^{+}\}$$

El punto decimal en la definición de L_{ef} es un símbolo, no debe confundirse con el operador '$\circ$' de concatenación de cadenas introducido en el tema anterior (que normalmente se omite por brevedad). Supongamos que se requiere escribir un programa donde en un punto determinado se necesita comprobar si una cadena de caracteres w se ajusta al patrón descrito anteriormente (es válida o correcta según la especificación del lenguaje), que equivale a comprobar si $w \in L_{ef}$. Deberíamos pensar en un algoritmo para resolver la parte del programa que se refiere a la **validación de la cadena** *haciendo* **abstracción** *para dejar a un lado en una primera aproximación cuestiones concretas de implementación como:*

- *¿La cadena la introduce el usuario por teclado, hay que leerla de un fichero de texto o hay que extraerla de una estructura de datos?*
- *Una vez que se haya comprobado si la cadena es correcta o incorrecta, ¿hay que imprimir un mensaje? ¿hay que convertir la cadena a su valor numérico? ¿hay que devolver algo?*

*Para centrarnos en el problema de la validación en sí conviene pensar que necesitamos leer la cadena de entrada carácter a carácter de izquierda a derecha y distinguir las situaciones (**estados**) en las que nos podemos encontrar en el procesamiento de la cadena y en cómo se pasa de una situación a otra (**transición**) y cuándo podemos asegurar que la cadena es válida o incorrecta. Los estados distinguibles en este problema serían:*

- *INI: empezamos a recorrer la cadena [INI es un **estado inicial**]*
- *PE: se ha leído un prefijo de la cadena que consiste en uno o más dígitos.*
- *PEPUNTO: se ha leído el prefijo con la parte entera seguida del punto.*
- *OK: se ha leído un prefijo con parte entera y fraccionaria. [OK es un **estado final** y si se ha terminado de leer toda la cadena entonces la cadena se acepta como válida.]*

*Los cambios de estado se producen como reacción a la lectura de cada carácter en el recorrido de la cadena y el proceso de cambio de estado según símbolo leído (transición) se puede describir gráficamente con un **diagrama de transición** como el que se muestra en la Figura 1. Este diagrama representa a un autómata finito determinista que llamamos M_{ef}. Cada nodo del diagrama se corresponde con un estado del autómata y los arcos muestran cómo se cambia de estado al 'leer' el símbolo que etiqueta al arco. En el diagrama se usa la abreviatura 0-9 para indicar, por ejemplo, que hay cambio de estado de INI a PE al leer cualquier dígito entre 0 y 9, en lugar de incluir un arco para cada uno de los símbolos.*

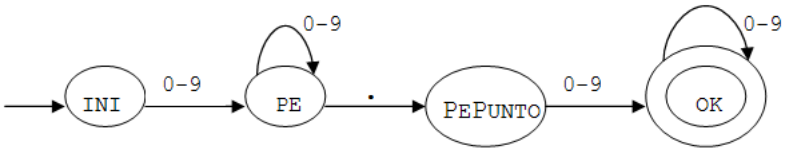


Figura 1: Diagrama de transición del AFD M_{ef}

Una forma sencilla de comprobar si una cadena representa un número de parte entera y fraccionaria (es válida según la definición de L_{ef}) es buscando un camino en el diagrama del autómata que parta del estado inicial y llegue a estado final y en cada arco se lea un símbolo de la cadena. Si existe tal camino la cadena es aceptada como válida y en otro caso es rechazada. Según el autómata M_{ef} la cadena "24.8" es válida por el siguiente camino:

$$INI \xrightarrow{2} PE \xrightarrow{4} PE \xrightarrow{.} PEPUNTO \xrightarrow{8} OK \quad [\text{acaba en estado final} \rightsquigarrow \text{acepta "24.8"}]$$

La cadena "24" es rechazada porque el camino no acaba en estado final (le falta el punto y la parte fraccionaria):

$$INI \xrightarrow{2} PE \xrightarrow{4} PE \qquad [\text{no acaba en estado final} \rightsquigarrow \text{rechaza "24"}]$$

*El autómata M_{ef} está diseñado para aceptar sólo aquellas cadenas que pertenecen al lenguaje L_{ef} y se puede decir que el autómata es un formalismo de **descripción computacional** de este lenguaje porque, a diferencia de la descripción matemática por comprensión, permite que*

pueda diseñarse de manera sencilla un algoritmo para comprobar si una cadena se ajusta al patrón que caracteriza a las cadenas de L_{ef}, como veremos en la sección 1.4. Por tanto, un autómata finito también puede considerarse como un ***modelo de algoritmo de validación*** *de cadenas.*

1.1. Definición formal de AFD

Un *autómata finito determinista M* puede considerarse una *máquina teórica abstracta* (implementable mediante un programa de ordenador o un dispositivo físico) que acepta o rechaza cadenas o secuencias de símbolos de entrada y podemos representarla esquemáticamente como:

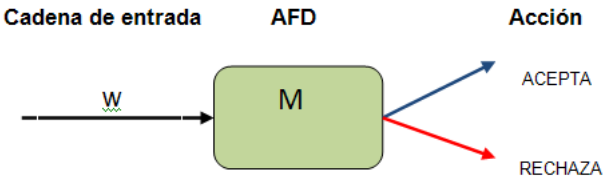


Figura 2: Esquema de caja negra de un AFD

Ahora introducimos la definición matemática formal, comúnmente adoptada en la bibliografía sobre teoría de autómatas y lenguajes formales, para definir de forma escueta y precisa los distintos componentes de un autómata finito determinista, la esencia de lo que forma parte de la caja negra en el esquema anterior.

Definición 1 (AFD) *Un AUTÓMATA FINITO DETERMINISTA (AFD) (deterministic finite automaton DFA) es un modelo de una máquina teórica abstracta que se define matemáticamente mediante una quíntupla* $\boxed{M = (Q, V, \delta, q_0, F)}$ *donde M es el* ***nombre*** *del AFD y los* ***componentes*** *son:*

- Q *es un conjunto finito de* ***estados***
- V *es el* ***alfabeto*** *del autómata*
- q_0 *es el* ***estado inicial***
- $F \subseteq Q$ *es el conjunto de* ***estados finales***
- δ *es la* ***función de transición*** *y se define como:*

$$\delta : Q \times V \longrightarrow Q$$

La **función de transición** δ es el componente esencial del autómata, el que establece cómo se efectúan los cambios de estado y por tanto determina el tipo de cadenas que el autómata es capaz de aceptar. El *dominio* de la función es $Q \times V$ y eso indica que la función se aplica a un

estado y a un símbolo y el *codominio* Q indica que devuelve un estado.
Una **regla de transición** es un caso de la función de transición (es una función finita y por tanto hay finitas reglas), del tipo $\boxed{\delta(q,a)=q'}$, donde $q, q' \in Q, a \in V$.
Se supone que en cada instante de ejecución con una cadena de entrada, el autómata se encuentra en cierto estado y ha leído cierto símbolo de la cadena, que imaginemos quedan almacenados en dos registros de la máquina, que llamamos *estadoActual* y *simboloActual*, respectivamente. Entonces, la regla de transición $\delta(q,a)=q'$ equivale a una instrucción de tipo *if-then*:
SI (*estadoActual*==q y *simboloActual*==a) ENTONCES *estadoActual*:=q'.

El adjetivo **"determinista"** en este tipo de autómatas es debido a que si la condición (*estadoActual*==q y *simboloActual*==a) es cierta entonces en el instante siguiente el estado de la máquina pasa a ser q', que es el estado indicado por la regla de transición $\delta(q,a)=q'$. El estado en el instante siguiente es único y esto implica que sólo hay un cálculo o forma de procesar una cadena de entrada, a diferencia de los *autómatas no deterministas*, donde el cambio de estado se realiza eligiendo entre varias alternativas, dando lugar a distintos cálculos.

Representación compacta de AFDs

En lugar de describir todos los componentes de un AFD M mediante una quíntupla del tipo (Q, V, δ, q_0, F) se puede representar M de forma más compacta de dos maneras: mediante una tabla de transición o mediante un diagrama de transición. La tabla es una representación tabular de la función de transición δ y el diagrama un representación gráfica mediante un grafo dirigido etiquetado. Ambos tienen anotaciones para distinguir el estado inicial y estados finales.

Tabla de transición	- Cada fila corresponde a un estado $q \in Q$ - Cada columna corresponde a un símbolo $a \in V$ - El estado inicial se precede del símbolo $\rightarrow$ - Cada estado final se precede del símbolo # - Para cada regla de transición $\delta(q,a)=q'$ se tiene que la posición de la tabla para la fila de q y columna de a contiene el estado q'
Diagrama de transición	- Los nodos se etiquetan con los nombres de los estados - El estado inicial tiene un arco entrante no etiquetado - Los estados finales están rodeados de un doble círculo - Para cada regla de transición $\delta(q,a)=q'$ se incluye un arco etiquetado con a desde el nodo q hasta el nodo q'

Ejemplo 2 *Supongamos que tenemos el AFD descrito por la quintupla*

$$M = (\{q_0, q_1\}, \{0,1\}, \delta, q_0, \{q_1\})$$

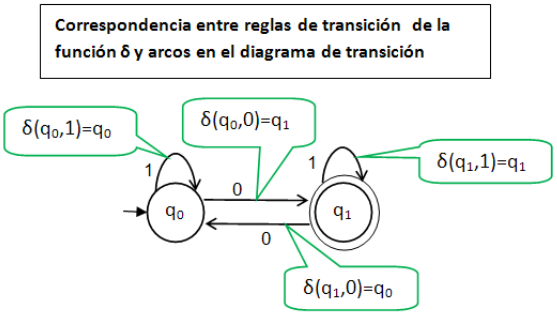
Se tiene que el conjunto de estados es $Q = \{q_0, q_1\}$*, el alfabeto del autómata es* $V = \{0,1\}$ *y el conjunto de estados finales es* $F = \{q_1\}$*. La función de transición* δ *se describe* ***matemáticamente*** *como:*

$$\begin{array}{ll} \delta(q_0,0)=q_1 & \delta(q_0,1)=q_0 \\ \delta(q_1,0)=q_0 & \delta(q_1,1)=q_1 \end{array}$$

El autómata M se representa mediante la ***tabla de transición*** *siguiente:*

δ	0	1
$\rightarrow q_0$	q_1	q_0
$\# q_1$	q_0	q_1

o también gráficamente mediante el ***diagrama de transición:***



Intuitivamente podemos comprobar que el AFD anterior acepta la cadena 101 *porque existe un camino en el diagrama de transición por el que se leen o recorren todos los símbolos de la cadena y acaba en estado final:*

$$q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_1 \quad [q_1 \in F, \text{ acepta } 101]$$

Una cadena como 1010 *es rechazada porque el camino acaba en estado no final:*

$$q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_1 \xrightarrow{0} q_0 \quad [q_0 \notin F, \text{ rechaza } 1010]$$

AFDs completos e incompletos

La función de transición $\delta : Q \times V \longrightarrow Q$ de un AFD puede ser *parcial o total*. Parcial quiere decir que hay algún caso $\delta(q, a)$ no definido en la función de transición. La función es total si está definida para todas las combinaciones de pares de estado y símbolo del alfabeto. Se dice que un **AFD es completo** si su función de transición es total y es un **AFD incompleto** si su función de transición es parcial.

En algunos libros de la bibliografía se considera que los autómatas finitos deterministas deben ser completos. Aquí no ponemos esa restricción y es sencillo transformar un AFD incompleto en uno completo *equivalente* (acepta las mismas cadenas), siguiendo un método que llamamos *completaAFD*.

Lo que se hace es completar el autómata original con un **estado de error o trampa**, donde van a parar todos los arcos correspondientes a los casos no definidos de la función de transición y se incluyen arcos desde el estado trampa a sí mismo con todos los símbolos de alfabeto, de manera que cuando el autómata alcanza ese estado no puede salir de él hasta que termina de leer todos los símbolos de la cadena.

Ejemplo 3 *Consideramos el AFD con alfabeto $V = \{a, b, c\}$ que se muestra en la figura de la página 9 (izquierda). Es incompleto porque los casos $\delta(q_0, b), \delta(q_1, a), \delta(q_1, c)$ de la función*

Algoritmo *completaAFD*

FUNCIÓN completaAFD (M)

ENTRADA: un AFD $M = (Q, V, \delta, q_0, F)$
DEVUELVE: un AFD completo equivalente $M_{new} = (Q_{new}, V, \delta_{new}, q_0, F)$

SI M es completo ENTONCES

DEVUELVE (M); // el autómata de salida es el mismo de entrada

SI-NO

1. $Q_{new} := Q \cup \{q_{err}\}$; // se añade un estado de error q_{err}
2. Se hace que δ_{new} coincida con δ en todos los casos definidos de δ;
3. Completar transiciones:
   - PARA CADA caso $\delta(q, a)$ no definido HACER $\delta_{new}(q, a) := q_{err}$;
   - PARA CADA símbolo $a \in V$ HACER $\delta_{new}(q_{err}, a) := q_{err}$;
4. DEVOLVER (M_{new});

---

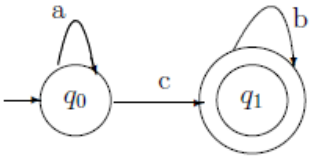
de transición no están definidos. En la tabla de transición tendríamos las posiciones correspondientes a esos casos vacías:

δ	a	b	c
$\rightarrow q_0$	q_0		q_1
$\# \; q_1$		q_1	

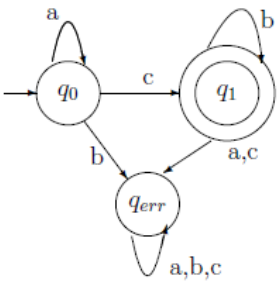
La cadena abc es rechazada porque no se puede terminar de leer la cadena (no se detecta el fin de la entrada) porque el caso $\delta(q_0, b)$ no está definido:

$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} \quad \textit{[rechaza abc]}$$

El diagrama de la derecha en la siguiente figura es el que resulta de completar el AFD:



AFD incompleto



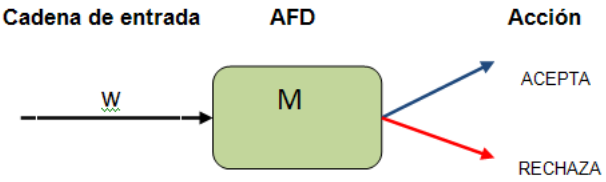
AFD completo equivalente

Nota: En una aplicación real de validación de formato de cadenas, el conjunto de caracteres que pueden aparecer en una cadena normalmente es más amplio que el alfabeto del autómata

que se usa en el diseño del algoritmo para validar las cadenas, pero en el modelo teórico no se incluyen todos los caracteres posibles, sino sólo los símbolos del alfabeto del autómata, con los que se forman las cadenas consideradas válidas. Al completar un AFD sólo se añaden las transiciones no definidas para los símbolos del alfabeto. Ej. la cadena $ac + b$ no es aceptada por AFD anterior porque no hay regla de transición definida con carácter '+', que es como un "símbolo extraño" que no forma parte del alfabeto $\{a, b, c\}$ del autómata y al completar el AFD sólo se añaden las transiciones no definidas con los símbolos de $\{a, b, c\}$.

1.2. Funcionamiento de un AFD. Configuraciones y cálculo

Recordamos el esquema de caja negra de un AFD:



Como máquina teórica, el **funcionamiento de un AFD** es muy simple:

- El *estadoActual* del autómata comienza siendo el estado inicial, apunta al primer símbolo de la cadena de entrada (lee primer símbolo, el de más a la izquierda), que pasa a ser el *simboloActual*.
- Realiza un bucle consistente en "cambiar de estado + avanzar apuntador en la cadena de entrada (lee siguiente símbolo)". El cambio de estado en cada *paso de ejecución o paso de cálculo* del bucle se produce según indica la regla de transición para la combinación de valores de los registros de *estadoActual* (último estado alcanzado) y *símboloActual* (último símbolo leído). El bucle finaliza cuando no es posible cambiar de estado mediante las reglas de transición.
- Si se han leído todos los símbolos de la cadena de entrada (se ha detectado el fin de la entrada) y *estadoActual* es un estado final entonces el autómata ACEPTA la cadena de entrada; en otro caso la RECHAZA.

Nota: El autómata lee la cadena de entrada símbolo a símbolo, de izquierda a derecha (la recorre avanzando el apuntador) sin retroceder y no necesita memoria para almacenar parte de la cadena leída hasta cierto instante. Mediante los estados se recuerda lo esencial de lo que se ha leído de la cadena hasta cierto momento. En teoría los autómatas finitos son máquinas sin salida, salvo devolver *true* (aceptar) o devolver *false* ('rechazar'). En la práctica, los algoritmos de procesamiento de cadenas diseñados a partir de autómatas finitos proporcionan salida o realizan acciones diversas a lo largo del procesamiento, conforme se va cambiando de estado.

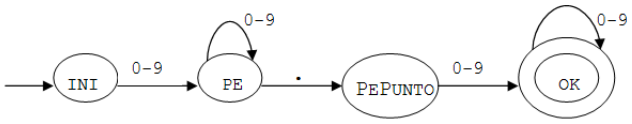
Mediante los conceptos de configuración, paso de cálculo y cálculo en un AFD, que pasamos a definir ahora, se formaliza la condición de aceptación de una cadena, la traza de ejecución del autómata con una cadena de entrada, y se puede determinar el *lenguaje aceptado* o reconocido por un AFD, que consiste en el conjunto de cadenas que el autómata es capaz de aceptar.

Definición 2 (configuración) *Una configuración de un autómata finito en cierto instante (configuración actual) describe el estado actual y la porción de cadena de entrada que le queda por leer o procesar, de la cual el símbolo más a la izquierda es el símbolo actual. Formalmente, una* CONFIGURACIÓN *de* $M = (Q, V, \delta, q_0, F)$ *es un par* (q, x) *con* $q \in Q$ *y* $x \in V^*$.

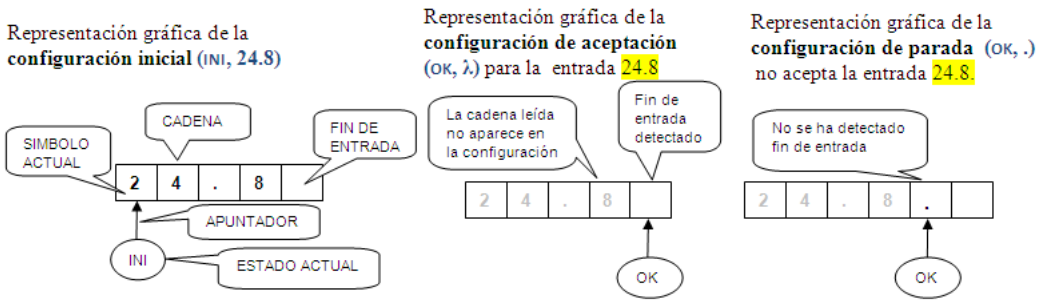
Algunos tipos de configuraciones especiales son:

- **Configuración inicial**: (q_0, w), donde q_0 es el estado inicial y w la cadena de entrada. Es la configuración de la máquina al inicio de la ejecución con la cadena w.
- **Configuración de parada**: cualquier configuración en la que queda el autómata tras finalizar el bucle de ejecución, bien porque se han leído todos los símbolos de la cadena (fin de la entrada detectado), o bien porque no hay transición definida para la combinación de *estadoActual* y *símboloActual* (puede darse en un AFD incompleto).
- **Configuración de aceptación**: (q_F, λ), es una configuración de parada donde q_F es un estado final. La cadena vacía λ indica que no quedan más símbolos por leer (fin de entrada detectado). Una vez alcanzada esta configuración, el autómata acepta la cadena con la que comenzó la ejecución.

Ejemplo 4 *Consideramos de nuevo el autómata* M_{ef}*:*



En este autómata se han dado nombres significativos a los estados INI,PE... *en lugar de la notación teórica por defecto:* $q_0, q_1 \ldots$. *Supongamos que tenemos la cadena de entrada "24.8". La configuración inicial con dicha cadena es* $(\text{INI}, 24.8)$ *y como esta cadena es válida el autómata para en configuración de aceptación* (OK, λ). *Partiendo de esta otra configuración inicial:* $(\text{INI}, 24.8.)$, *el autómata alcanza la configuración de parada* $(\text{OK}, .)$ *después de haber leído el prefijo "24.8" y el autómata para sin llegar a detectar el fin de la entrada, ya que la transición* $\delta(\text{OK}, .)$ *no está definida, por lo que* M_{ef} *rechaza la cadena "24.8.". En los siguientes dibujos vemos la correspondencia entre varias* ***configuraciones en notación matemática*** *y su* ***representación gráfica****:*



Definición 3 (paso de cálculo) *Un* PASO DE CÁLCULO *en un AFD* $M = (Q, V, \delta, q_0, F)$ *consiste en el paso de una configuración a otra por aplicación de una regla de transición.*

Se dice que la configuración (q, az), con $a \in V, z \in V^*$, ***alcanza en un paso de cálculo*** *la configuración (q', z), y se denota:*

$$\boxed{(q, az) \Rightarrow (q', z)}$$

si y sólo si en la función de transición tenemos la regla $\delta(q, a) = q'$.

De no estar definida $\delta(q, a)$ entonces no es posible que el autómata efectúe un paso de cálculo a partir de (q, az), por lo que (q, az) será una configuración de parada.

Ejemplo 5 *En el autómata M_{ef} podemos afirmar que:*

$$(\textsc{ini}, 24.8) \Rightarrow (\textsc{pe}, 4.8)$$

porque por el diagrama de transición comprobamos que tenemos definida la transición $\delta(\textsc{ini}, 2) = \textsc{pe}$. Sin embargo es falso que $(\textsc{ok}, .) \Rightarrow (\textsc{ok}, \lambda)$ porque la transición $\delta(\textsc{ok}, .)$ no está definida.

Definición 4 (cálculo) *Un CÁLCULO en un AFD $M = (Q, V, \delta, q_0, F)$ consiste en el paso de una configuración a otra por aplicación de cero o más reglas de transición. Distinguimos dos casos:*

1. ***Cálculo en uno o más pasos.*** *Se dice que configuración C_0 alcanza la configuración C_n en $n > 0$ pasos, y se denota:*

   $$\boxed{C_0 \Rightarrow^* C_n}$$

   si y sólo si existe cálculo consistente en una secuencia de uno o más pasos de cálculo del tipo:

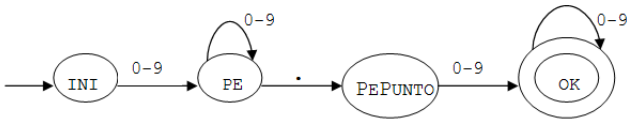
   $$C_0 \Rightarrow C_1 \Rightarrow \ldots \Rightarrow C_n$$

   donde cada C_i es una configuración y en el cálculo se aplican $n > 0$ reglas.

2. ***Cálculo en cero pasos.*** *Por defecto, para cualquier configuración (q, x) se tiene que $\boxed{(q, x) \Rightarrow^* (q, x)}$, lo cual indica que no se aplica ninguna regla de transición.*

Un ***cálculo para una cadena de entrada*** *w (****o traza de ejecución****) es un cálculo que comienza con la configuración inicial (q_0, w) y acaba en configuración de parada.*

Ejemplo 6 *Seguimos con el autómata M_{ef}*



Anteriormente se comprobó de manera informal que la cadena 24.8 es aceptada por el camino del diagrama:

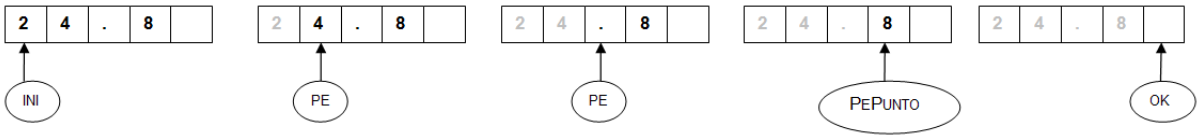
$$\textsc{ini} \xrightarrow{2} \textsc{pe} \xrightarrow{4} \textsc{pe} \xrightarrow{.} \textsc{pepunto} \xrightarrow{8} \textsc{ok}$$

Ahora comprobamos formalmente que se cumple: $(\textsc{ini}, 24.8) \Rightarrow^* (\textsc{ok}, \lambda)$ y para ello mostramos el ***cálculo completo*** *para la cadena 24.8, con la secuencia de pasos de cálculo. Encima del*

símbolo de paso de cálculo $\Rightarrow$*, indicamos además la regla de transición que se aplica en ese paso.*

$$(INI, 24.8) \overset{\delta(INI,2)=PE}{\Rightarrow} (PE, 4.8) \overset{\delta(PE,4)=PE}{\Rightarrow} (PE, .8) \overset{\delta(PE,.)=PEPUNTO}{\Rightarrow}$$
$$(PEPUNTO, 8) \overset{\delta(PEPUNTO,8)=OK}{\Rightarrow} (OK, \lambda) \quad [acepta]$$

Se tiene que el autómata M_{ef} *acepta la cadena* 24.8 *en* 4 *pasos porque se aplican* 4 *reglas de transición, que equivale a ejecutar* 4 *instrucciones elementales. El* ***cálculo o traza de ejecución*** *anterior se puede* ***representar gráficamente*** *mediante una secuencia gráfica de configuraciones:*



El **tiempo de ejecución** de un AFD con una cadena de entrada w se mide por el número de pasos de cálculo para aceptar o rechazar la cadena w o número de reglas de transición que se aplican. Como en cada paso, además de cambiar de estado, se avanza una posición en la cadena, entonces si $|w| = n$ el tiempo de ejecución del AFD con w es de $O(n)$ pasos. Será exactamente n pasos si el AFD es completo, en otro caso puede rechazar una cadena en menos pasos.

1.3. Lenguaje aceptado por un AFD.

Ya hemos comentado cuándo un AFD acepta o rechaza una cadena. El conjunto de todas las cadenas aceptadas por un AFD forma un lenguaje, que se define formalmente como:

Definición 5 (lenguaje aceptado por un AFD) *Dado un autómata finito determinista* $M = (Q, V, \delta, q_0, F)$*, el* LENGUAJE ACEPTADO POR M*, que denotamos* $L(M)$*, se define como:*

$$\boxed{L(M) = \{w \in V^* \mid (q_0, w) \Rightarrow^* (q_F, \lambda), q_F \in F\}}$$

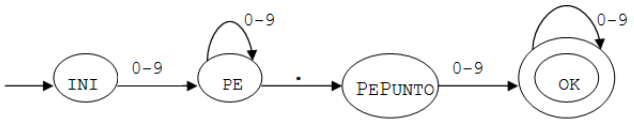
La condición $(q_0, w) \Rightarrow^* (q_F, \lambda), q_F \in F$ significa w es aceptada, porque existe un cálculo que parte de la configuración inicial (q_0, w) y alcanza la configuración de aceptación (q_F, λ), lo cual es equivalente a decir que existe un camino en el diagrama de transición desde q_0 hasta q_F pasando por arcos etiquetados con cada símbolo de w. Un AFD puede aceptar infinitas cadenas pero sólo un único lenguaje.

Como caso especial, **la cadena vacía es aceptada por un AFD si y sólo si el estado inicial es un estado final**, porque a partir de la configuración inicial con (q_0, λ) el único cálculo posible es: $(q_0, \lambda) \Rightarrow^* (q_0, \lambda)$ (cero pasos) y, por tanto, q_0 debe ser final para que $\lambda \in L(M)$.

1.4. Simulación de AFDs

Una vez obtenido el modelo teórico de AFD para aceptar cadenas según la descripción de cierto lenguaje, se puede diseñar un algoritmo de simulación del funcionamiento del AFD, que sirve como *algoritmo de validación* para comprobar si las cadenas que se le pasan como entrada son válidas o incorrectas. En los autómatas deterministas el tiempo de ejecución para cadenas de longitud n es de orden $O(n)$ pasos, por lo que **el algoritmo de simulación es de complejidad O(n)**, que es muy eficiente.

Ejemplo 7 *Como ejemplo de simulación consideramos el autómata* M_{ef}*:*




```
DESCRIPCIÓN: comprueba si una cadena es válida según el lenguaje L_ef.
             Simula el funcionamiento del autómata M_ef.
FUNCION esValida(cadenaEntrada)
        DEVUELVE  true si  cadenaEntrada es aceptada por M_ef y false en otro caso.

INICIO
   estadoActual := INI; //comienza la simulación por el estado inicial
   simboloActual:= nulo; posicion:= 0; // para leer el primer símbolo en posición 0

  MIENTRAS (noFin(cadenaEntrada,posicion))  // bucle de procesamiento
    simboloActual=leeSimbolo(cadenaEntrada,posicion);
    //Aplica regla de transición según diagrama
    SI estado==INI y esDigito(simboloActual) ENTONCES estadoActual:=PE;
    SI-NO SI estado==PE y esDigito(simboloActual) ENTONCES estadoActual:=PE;
    SI-NO SI estado==PE y simboloActual=='.' ENTONCES estadoActual:=PEPUNTO;
    SI-NO SI estado==PEPUNTO y esDigito(simboloActual) ENTONCES estadoActual:=OK;
    SI-NO SI estado==OK y esDigito(simboloActual) ENTONCES estadoActual:=OK;
    SI-NO DEVUELVE(false); //transición no definida, cadena incorrecta, retorna
    posicion := posicion +1; // avanza una posición en la cadena
  FIN-MIENTRAS

  //Comprobación final, la cadena se ha leído por completo
  SI esFinal(estadoActual) ENTONCES DEVUELVE(true); //cadena válida
  SI-NO DEVUELVE(false); //cadena  incorrecta
FIN

//ORDEN DE COMPLEJIDAD: O(n), siendo n la longitud de la cadena de entrada.
```

Figura 3: **Algoritmo de simulación del autómata M_{ef} basado en el diagrama de transición.**

El algoritmo en pseudocódigo de la figura 3 simula el funcionamiento del autómata M_{ef} mediante una **función de validación** (`esValida`) que codifica el flujo de cambio de estado en el diagrama de transición mediante instrucciones IF-THEN-ELSE y sirve para comprobar si una cadena de entrada tiene el formato correcto de número decimal con parte entera y fraccionaria. Se omiten declaraciones de tipos y pseudocódigo de funciones auxiliares.

Se puede diseñar un algoritmo basado en el diagrama de transición para cada autómata concreto, pero también podemos obtener un **algoritmo general de simulación** basado en la tabla de transición, como el mostrado en la Figura 4, que sirve para **cualquier AFD**, pues basta con instanciar las variables que definen los componentes del autómata con los datos de un autómata concreto.

```
DESCRIPCIÓN: simula el funcionamiento de un AFD por acceso a tabla
de transición.

VARIABLES: un autómata M con conjunto de estados Q, alfabeto V,
estado inicial q0,
           conjunto de estados finales F y tabla de transición  tablaTran.
           //tablaTran codifica las reglas de transición del AFD.

FUNCION esValida(cadenaEntrada)
        DEVUELVE  true si  cadenaEntrada es aceptada por M y false en otro caso.
INICIO
   estadoActual := INI; //comienza la simulación por el estado inicial
   simboloActual:= nulo; posicion:= 0; // para leer el primer símbolo en posición 0

  MIENTRAS (noFin(cadenaEntrada,posicion))  // bucle de procesamiento
    simboloActual:=leeSimbolo(cadenaEntrada,posicion);
                //accede al símbolo en la posición actual

    //Aplica una regla de transición para cambiar de estado,
    //consultando tablaTran a través de una llamada a la función transición.
    SI tranDefinida(estadoActual,simboloActual)
        ENTONCES estadoActual:=transicion(estadoActual,simboloActual);
    SI-NO DEVUELVE(false); //transición no definida, cadena incorrecta, retorna
    posicion=posicion +1; // avanza una posición en la cadena
  FIN-MIENTRAS

  //Comprobación final, la cadena se ha leído por completo
  SI esFinal(estadoActual) ENTONCES DEVUELVE(true); //cadena válida
  SI-NO DEVUELVE(false); //cadena  incorrecta
FIN

//ORDEN DE COMPLEJIDAD: O(n), siendo n la longitud de la cadena de entrada.
```

Figura 4: **Algoritmo general de simulación de un AFD**

Se omiten los detalles sobre tipos de datos para los distintos componentes del autómata, tipo de parámetros y el código de funciones auxiliares. Lo llamamos "algoritmo basado en la tabla de transición" porque los cambios de estado se realizan consultando la tabla de transición `tablaTran`, que codifica las reglas de la función de transición del autómata, en lugar de aplicar las transiciones mediante instrucciones IF-THEN-ELSE.

La tabla `tablaTran` puede implementarse mediante un array bidimensional u otra estructura de datos que permita acceso rápido. Hay que considerar la situación en la que haya una regla de transición no definida para cierta combinación de `estadoActual` y `símboloActual`. Para ello, en la función de validación `esValida`, se llama a `tranDefinida(estadoActual,simboloActual)`, que devuelve *true* si la función de transición está definida para la combinación de `estadoActual` y `símboloActual`, o devuelve *false* en otro caso. Se considera la posibilidad de que el símbolo leído sea un símbolo extraño que no pertenece al alfabeto del autómata. Si el AFD es completo y no se puede dar el caso de que la cadena de entrada tenga símbolos extraños entonces no es necesaria esta comprobación mediante la función auxiliar `tranDefinida`.

Por último, se llama a la función `transicion(estadoActual,simboloActual)`, que es la que implementa la función de transición del autómata y se encarga de devolver el estado que se alcanza desde `estadoActual` con `simboloActual`, que está almacenado en la tabla `tablaTran`.

1.5. Análisis de AFDs

Un *problema de análisis* de un AFD M consiste en **deducir el lenguaje que acepta**, describiendo $L(M)$ de forma explícita en lenguaje natural o matemáticamente por comprensión. Para ello se analiza el autómata comprobando todos los caminos que llevan desde el estado inicial a un estado final y el lenguaje que acepta M está formado por la unión de las cadenas que se leen por los distintos caminos.

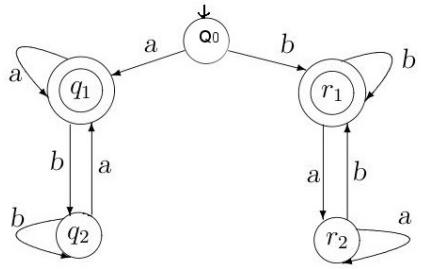
Del análisis de los diferentes caminos se hace una **hipótesis del lenguaje aceptado**, por ejemplo, se describe un lenguaje H y para comprobar que la hipótesis es correcta hay que probar que $\boxed{H = L(M)}$, es decir:

1. Que M "no es demasiado estricto", en el sentido de que acepta todas las cadenas de H. Formalmente significa probar que $H \subseteq L(M)$.
2. Que M "no es demasiado general", en el sentido de que M rechaza cualquier cadena que no pertenece al lenguaje H. Formalmente supone probar $L(M) \subseteq H$.

Una demostración rigurosa de que $H = L(M)$ requiere combinar técnicas de demostración por inducción, por casos y deducción. En algunos ejemplos haremos una justificación menos formal pero convincente de que el lenguaje obtenido es correcto.

Por último comentar que hay autómatas finitos cuyo lenguaje aceptado no admite una descripción concisa e intuitiva. En estos casos se aplica una técnica de análisis obteniendo una expresión algebraica llamada expresión regular, que veremos en el tema siguiente.

Ejemplo 8 *Sea el autómata M_{etm} que se muestra en el siguiente diagrama de transición:*



El alfabeto es $V = \{a, b\}$ porque sólo aparecen esos símbolos en los arcos y el conjunto de estados es $Q = \{Q_0, q_1, q_2, r_1, r_2\}$, de los cuales el inicial es Q_0 y el conjunto de estados finales es $F = \{q_1, r_1\}$. Podemos observar que el autómata es completo porque δ está definida para cada combinación de estado y símbolo. Veamos qué tipo de cadenas acepta este autómata. Analizando los distintos caminos en el diagrama que llevan desde el estado inicial a uno de los estados finales establecemos la HIPÓTESIS *de que este autómata acepta el lenguaje que se describe informalmente como "cadenas de a's y b's que empiezan y terminan con el mismo símbolo" y lo llamamos H. Se supone que las cadenas a, b cumplen esa propiedad. Formalmente H se define como:*

$$H = \{w \in \{a, b\}^* \mid w = a \vee w = b \vee w = aza \vee w = bzb, \text{ donde } z \in \{a, b\}^*\}$$

Justificamos que la hipótesis de que $L(M_{etm}) = H$ es cierta. Para ello seguimos los dos pasos indicados antes:

1. *M_{etm} no es demasiado estricto:* $\boxed{H \subseteq L(M_{etm})}$
   Lo probamos usando un argumento basado en el análisis de caminos del diagrama de transción y en la definición formal de lenguaje aceptado por un AFD, mediante cálculos para los diferentes casos de cadenas en H.

   - *Caso $w = a$ o $w = b$. Estas cadenas claramente son aceptadas por M ya que:*

     $$(Q_0, a) \Rightarrow (q_1, \lambda) \qquad [q_1 \in F, \text{ acepta}]$$
     $$(Q_0, b) \Rightarrow (r_1, \lambda) \qquad [r_1 \in F, \text{ acepta}]$$

   - *Caso $w = aza, z \in \{a, b\}^*$. La cadena de menor longitud para este caso es aa y se tiene que $aa \in L(M)$ porque $(Q_0, aa) \Rightarrow^* (q_1\lambda)$ y $q_1 \in F$, como puede probarse a partir del siguiente cálculo explícito o secuencia de pasos de cálculo:*

     $$(Q_0, aa) \overset{\delta(Q_0,a)=q_1}{\Rightarrow} (q_1, a) \overset{\delta(q_1,a)=q_1}{\Rightarrow} (q_1, \lambda) \qquad [q_1 \in F, \text{ acepta}]$$

     Si la cadena tiene a's/b's entre la 'a' de comienzo y fin de cadena (tipo $aza, |z| > 0$) el autómata la acepta porque cambia al estado q_1 cada vez que lee una a y cambia al estado q_2 cada vez que lee una b. Como el último símbolo de las cadenas de este tipo es una 'a', entonces al leer esa 'a' el autómata acabará en estado final q_1. El estado q_1 'recuerda' que el último símbolo leído fue 'a' y la cadena comenzó por 'a'. Por ejemplo, $abba \in L(M)$ porque $(Q_0, abba) \Rightarrow^* (q_1\lambda)$ y $q_1 \in F$, ya que:

     $$(Q_0, abba) \Rightarrow (q_1, bba) \Rightarrow (q_2, ba) \Rightarrow (q_2, a) \Rightarrow (q_1, \lambda) \qquad [q_1 \in F, \text{ acepta}]$$

- *Caso $w = bzb, z \in \{a,b\}^*$. Este caso es análogo al anterior. La cadena de menor longitud es bb y se tiene que $bb \in L(M)$, como muestra el siguiente cálculo:*

$$(Q_0, bb) \overset{\delta(Q_0,b)=r_1}{\Rightarrow} (r_1, b) \overset{\delta(r_1,b)=r_1}{\Rightarrow} (r_1, \lambda) \qquad [r_1 \in F,\ acepta]$$

Si la cadena tiene a's/b's en medio de la 'b' de comienzo y fin de cadena (tipo $bzb, z \in |z| > 0$) el autómata la acepta porque la última b de la cadena hará que se alcance el estado final r_1.
Por ejemplo, $babb \in L(M)$ y es aceptada en tiempo 3 pasos, ya que:

$$(Q_0, babb) \Rightarrow (r_1, abb) \Rightarrow (r_2, bb) \Rightarrow (r_1, b) \Rightarrow (r_1, \lambda) \qquad [r_1 \in F\ acepta]$$

2. *M_{etm} no es demasiado general:* $\boxed{L(M_{etm}) \subseteq H}$
Veamos que es así mediante un argumento basado en los caminos del diagrama de transición. ¿Es posible que el autómata acepte cadenas que no empiecen y acaben por el mismo símbolo? La respuesta es que no, porque si la cadena empieza por 'a' y termina por 'b' entonces el autómata acaba en estado q_2 tras leer la cadena y si empieza por 'b' y termina por 'a' entonces el autómata acaba en estado r_2 y ninguno de esos estados es final.
Por ejemplo, la cadena abb es rechazada porque:

$$(Q_0, abb) \Rightarrow (q_1, bb) \Rightarrow (q_2, b) \Rightarrow (q_2, \lambda) \qquad [q_2 \notin F,\ rechaza]$$

También comprobamos que λ no se ajusta al patrón de cadenas de H y el autómata no la acepta porque el estado inicial no es un estado final.

1.6. Diseño de AFDs

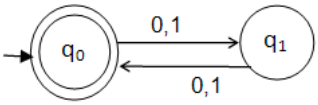
El *problema de diseño* de un AFD consiste en **obtener un autómata finito** que acepte todas las cadenas consideradas válidas en la descripción de cierto lenguaje formal y no otras. Como veremos más adelante, no todos los lenguajes formales pueden ser aceptados por autómatas finitos. El diseño de AFDs es un proceso creativo, como puede serlo la implementación de un programa para resolver determinado problema. No hay método para diseñar un autómata, es algo que se aprende con la experiencia a partir de los conocimientos teóricos. Lo importante es descubrir qué estados se necesitan para decidir si una cadena es válida o no y eso implica distinguir qué es lo esencial que se debe recordar de la parte de la cadena o prefijo leído hasta cierto instante, las situaciones relevantes para el procesamiento.

Toda etapa de diseño debe ir seguida de una etapa de **análisis de corrección del diseño** propuesto. En el caso del diseño de un autómata, una vez obtenido el autómata debe hacerse un análisis del mismo, donde se procede como en el apartado anterior, o de manera informal con un test de prueba que intente cubrir todos los casos de **cadenas válidas** (pertenecen al lenguaje, deben ser aceptadas) e **incorrectas** (no pertenecen al lenguaje, no deben ser aceptadas), para comprobar que el lenguaje aceptado por el autómata coincide con el lenguaje dado en la especificación del problema. De esa manera nos aseguramos que el diseño del autómata es correcto.

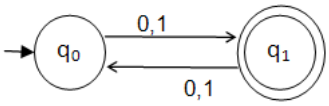
Ejemplo 9 *Con un AFD se pueden hacer comprobaciones sencillas que requieran un "conteo de símbolos". Por ejemplo, queremos comprobar mediante un AFD si una cadena de ceros y unos tiene un número par de dígitos y para ello diseñamos un AFD, que llamamos M_{parbin}, que acepte este tipo de cadenas, que forman un lenguaje que llamamos L_{parbin} y se define como:*

$$L_{parbin} = \{w \in \{0,1\}^* \mid |w| \text{ es par}\}$$

Diseñamos el AFD pensando que necesitamos el estado inicial q_0 para recordar que se ha leído un número par de dígitos binarios y otro estado q_1 para cuando se lee un número impar. La cadena λ es válida según la descripción del lenguaje, por tanto debe ser aceptada por el autómata. Para que el AFD acepte λ se hace que q_0 sea final. El diagrama de transición es:



El autómata es tan sencillo que no cabe duda de que es correcto, esto es, que $L(M_{parbin}) = L_{parbin}$. Si lo que se quiere es comprobar si una cadena de dígitos binarios tiene longitud impar, entonces podemos usar el mismo autómata pero cambiando el estado final:

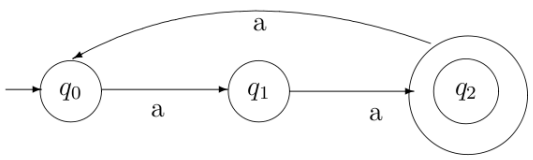


Otro problema más complejo de conteo de símbolos es comprobar si una cadena de a's es de longitud $3n+2, n \geq 0$, o lo que es lo mismo, el resto de dividir la longitud entre 3 es 2. Se trata de diseñar un AFD, que llamamos M_{a3aa} que acepte este tipo de cadenas, que forman el lenguaje:

$$L_{a3aa} = \{w \in \{a\}^* \mid |w| = 3n+2, n \geq 0\} \text{ o equivalentemente: } L_{a3aa} = \{(aaa)^n aa \mid n \geq 0\}$$

Idea para el diseño: *necesitamos estados que sirvan para recordar el resto de la longitud de la cadena leída hasta cierto instante entre 3, puesto que el autómata no realiza operaciones aritméticas y no hay límite en el número de símbolos de la cadena, con lo cual no puede usarse un estado para recordar cada número de símbolos leídos. Sólo hay tres restos posibles y cada uno de ellos se recuerda en un estado:*
Estado q_0: resto 0. Recuerda que se han leído cero $a's$, tres $a's$, seis $a's$, y en general un número de $a's$ múltiplo de 3.
Estado q_1: resto 1. Se han leído una a, cuatro $a's$, siete $a's$, y en general $3n+1$ $a's$.
Estado q_2: resto 2. Se han leído dos $a's$, cinco $a's$, y en general $3n+2$ $a's$.
Hacemos que $F = \{q_2\}$ sea el estado final, así se aceptan sólo cadenas de la forma $(aaa)^n aa$ o de longitud $3n+2$.

Dibujamos el diagrama de transición del autómata M_{a3aa} con esos estados y los arcos necesarios para los cambios de estado.



A partir de la idea del diseño podemos afirmar que ***el autómata es correcto*** *porque* $L(M_{a3aa}) = L_{a3aa}$*, con lo cual el autómata no es ni demasiado estricto ni demasiado general: acepta exactamente las cadenas del tipo* $(aaa)^n aa, n \geq 0$*. Probamos el diseño con un test de prueba de cadenas válidas (pertenecen a* L_{a3aa}*) e incorrectas (no pertenecen a* L_{a3aa}*) para diferentes casos:*

- ***Cadenas válidas:*** *probamos con la de menor longitud y la siguiente en longitud, que son* aa *(caso* $n = 0$ *según el patrón* $(aaa)^n aa$*) y* $aaaaa$ *(caso* $n = 1$*). Estas cadenas son aceptadas por* M_{a3aa} *como muestran los siguientes cálculos:*

  $(q_0, aa) \Rightarrow (q_1, a) \Rightarrow (q_2, \lambda) \quad [q_2 \in F,\ acepta]$
  $(q_0, aaaaa) \Rightarrow (q_1, aaaa) \Rightarrow (q_2, aaa) \Rightarrow (q_0, aa) \Rightarrow (q_1, a) \Rightarrow (q_2, \lambda) \quad [q_2 \in F,\ acepta]$

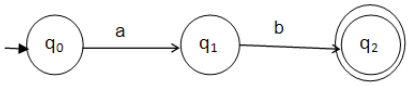
- ***Cadenas incorrectas:*** *rechaza* λ *porque el inicial no es final; probamos con siguientes en longitud de resto 1 y 0, que son* a *y* aaa*, respectivamente:*

  $(q_0, a) \Rightarrow (q_1, \lambda) \quad [q_1 \notin F,\ rechaza]$
  $(q_0, aaa) \Rightarrow (q_1, aa) \Rightarrow (q_2, a) \Rightarrow (q_0, \lambda) \quad [q_0 \notin F,\ rechaza]$

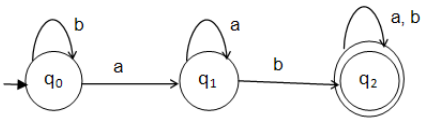
Ejemplo 10 *El siguiente ejemplo es del tipo "comprobar si una cadena contiene cierta subcadena". Consideramos cadenas de símbolos del alfabeto* $V = \{a, b\}$ *y queremos diseñar un AFD para comprobar si una cadena de a's/b's contiene la subcadena* ab*. El autómata debe aceptar exactamente las cadenas del lenguaje*

$$L_{ab} = \{xaby \mid x, y \in \{a, b\}^*\}$$

Idea para el diseño: *incluimos los estados y transiciones necesarias para aceptar la cadena válida de menor longitud, que es* ab*.*



A continuación añadimos los arcos para completar el autómata, con objeto de que se puedan leer cadenas de cualquier longitud, teniendo en cuenta que los estados tienen el siguiente significado:
Estado q_0*: aun no se ha detectado el comienzo de la subcadena* ab*.*
Estado q_1*: se ha leído una* a *que puede ser el comienzo de la subcadena* ab*.*
Estado q_2*: se ha leído la subcadena* ab *y una vez alcanzado este estado la cadena de entrada debe ser aceptada.*
El autómata que acepta el lenguaje L_{ab} *queda como:*



Puede comprobarse que es correcto haciendo un test de prueba cubriendo diferentes casos:

- ***Cadenas válidas:*** *la de menor longitud es* ab *y claramente la acepta; otros casos pueden ser cadenas de longitud mayor que dos que empiecen por* a *como* aab*, o que empiecen por* b *como* bab*, o que terminen por* a*, como* aba*. Para todas estas cadenas existe un camino que acaba en estado final. El cálculo por el que se aceptan se deja como ejercicio.*

- ***Cadenas incorrectas:*** *la de menor longitud es λ y no es aceptada porque el estado inicial no es final. Las siguientes incorrectas de longitud 1 y 2 son: a,b,aa,bb, ba y puede comprobarse que son rechazadas por el autómata.*

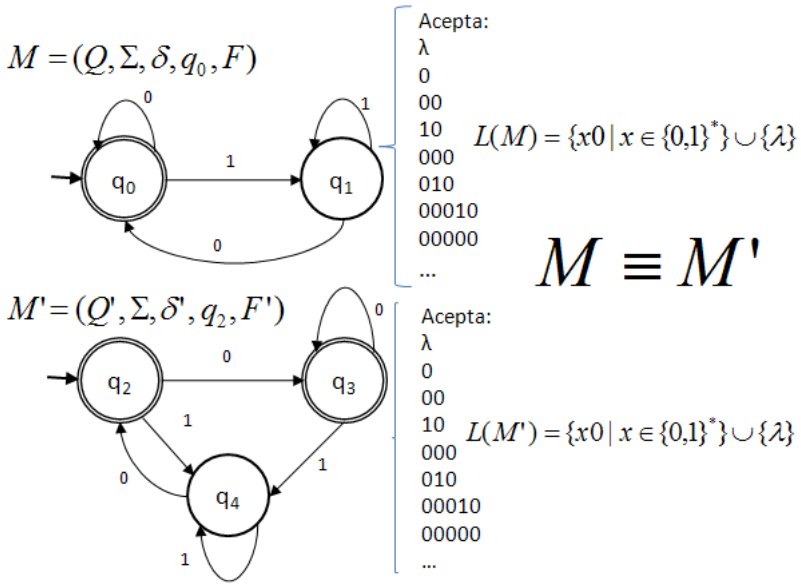
1.7. Minimización de AFDs

En ocasiones un diseño manual rápido de un AFD M o la generación automática de un AFD por aplicación de un método algorítmico puede ocasionar que el autómata tenga "más estados de la cuenta". Eso quiere decir que hay otro AFD M' con menos estados que es equivalente a M, es decir, aceptan las mismas cadenas. La definición formal de equivalencia de autómatas es la siguiente:

Definición 6 (equivalencia de autómatas) *Dos autómatas M y M' se dice que son* EQUIVALENTES *y se denota $M \equiv M'$, si y sólo si aceptan el mismo lenguaje, esto es:*

$$M \equiv M' \overset{def}{\iff} L(M) = L(M')$$

Ejemplo 11 *Los siguientes AFDs son equivalentes:*



Interesa encontrar el AFD equivalente a uno dado con el menor número de estados posible, porque cuanto menos estados tenga un AFD menos espacio ocupa la tabla de transición cuando se simula el AFD mediante un algoritmo y se implementa en un programa. También ocurre que si el autómata se usa para diseñar un dispositivo de control o circuito digital, cuanto menos estados tenga más barato resulta la construcción del hardware. En esta sección vamos a ver una operación, llamada **minimización**, que se aplica a un AFD y produce como resultado un AFD equivalente con el mínimo número de estados posible.

Definición 7 (autómata mínimo) *Un AFD M es* MÍNIMO *si no existe otro AFD completo M' con menos estados y equivalente a M. Más formalmente, se dice que un AFD es mínimo si y sólo si cumple dos condiciones:*

1. *Todos los estados son* ACCESIBLES.

2. *Todas las parejas de estados son* DISTINGUIBLES.

Ej. el autómata M mostrado en el ejemplo 11 es el autómata mínimo equivalente a M'.

Primero definiremos los conceptos de estados accesibles y distinguibles y después veremos un algoritmo que obtiene el autómata mínimo equivalente a cualquier AFD que tome como entrada.

Eliminación de estados inaccesibles

Definición 8 (estado accesible) *Un estado q de un autómata $M = (Q, V, \delta, q_0, F)$ es* ACCESIBLE *si y sólo si existe una cadena $x \in V^*$* tal que $(q_0, x) \Rightarrow^* (q, \lambda)$. *En otro caso el estado es* INACCESIBLE.

La definición formal se traduce en que un estado q es accesible si existe un camino en el diagrama de transición del AFD desde el inicial hasta q. Una primera simplificación que se puede hacer con un AFD es eliminar los estados inaccesibles, pues estos estados no aparecen en ningún cálculo que lleve a aceptar una cadena y por tanto el lenguaje aceptado por el AFD es el mismo si se suprimen estos estados. En la figura 5 se muestra un algoritmo que elimina los estados inaccesibles de un AFD.

FUNCIÓN eliminaInaccesibles(M)

ENTRADA: un AF $M = (Q, V, \delta, q_0, F)$
DEVUELVE: un AF M_{new} sin estados inaccesibles.

1. Obtener el grafo dirigido correspondiente al diagrama de transición de M.

2. Aplicar al grafo un método de recorrido de grafos dirigidos (por ejemplo *primero en profundidad*) para visitar todos los nodos partiendo del estado inicial y siguiendo todos los arcos.

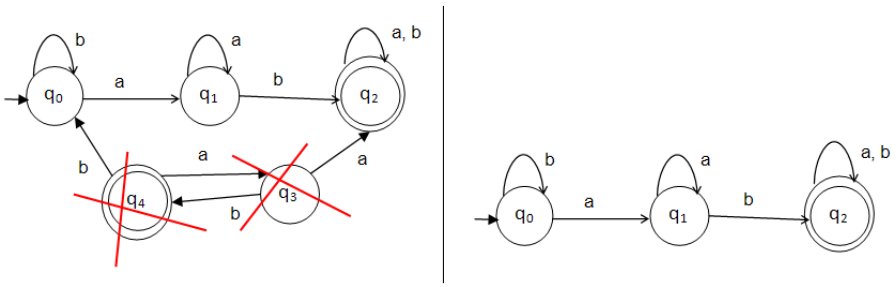
3. Cualquier nodo-estado que no haya sido visitado es inaccesible y se suprime del diagrama, junto con los arcos que parten de ese estado o llegan a ese estado.

4. DEVOLVER (M_{new}), donde los componentes de M_{new} se corresponden con el diagrama de transición modificado.

//ORDEN DE COMPLEJIDAD: $O(n^2)$, donde $n = |Q|$

Figura 5: **Algoritmo *eliminaInaccesibles***

Ejemplo 12 *Los estados q_3 y q_4 en el AFD de la izquierda son inaccesibles. Al eliminarlos se obtiene el AFD de la derecha.*



Estados distinguibles y equivalentes

Definición 9 (estados distinguibles y equivalentes) *Se dice que dos estados q_i, q_j de un AFD $M = (Q, V, \delta, q_0, F)$ son* DISTINGUIBLES *si existe una cadena $x \in V^*$ que los distingue y eso quiere decir que si al leer la cadena x desde el estado q_i y desde el estado q_j, se alcanzan los estados q_i' y q_j', respectivamente, entonces uno de ellos es final y otro no. Expresado de manera más formal, decimos que los estados q_i, q_j son distinguibles si y sólo si cumple la siguiente* ***condición de distinguibilidad:***

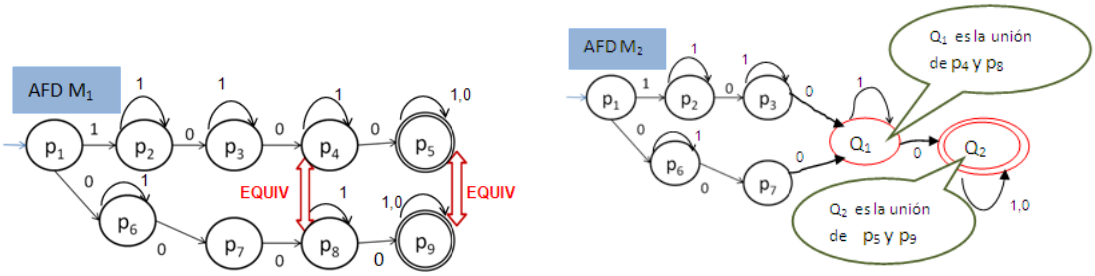
$$\exists\, x \in V^* \text{ tal que SI } (q_i, x) \Rightarrow^* (q_i', \lambda) \text{ y } (q_j, x) \Rightarrow^* (q_j', \lambda)$$
$$\text{ENTONCES } (q_i' \in F \wedge q_j' \notin F) \vee (q_i' \notin F \wedge q_j' \in F)$$

Decimos que dos estados q_i, q_j de un AFD son EQUIVALENTES *si no son distinguibles y se denota* $\boxed{q_i \equiv q_j}$*, es decir, no existe ninguna cadena x que los distinga. Negando la condición de distinguibilidad tenemos que $q_i \equiv q_j$ si y sólo cumplen la siguiente* ***condición de equivalencia****:*

$$\forall\, x \in V^* \text{ SI } (q_i, x) \Rightarrow^* (q_i', \lambda) \text{ y } (q_j, x) \Rightarrow^* (q_j', \lambda)$$
$$\text{ENTONCES } (q_i' \in F \wedge q_j' \in F) \vee (q_i' \notin F \wedge q_j' \notin F)$$

Los estados equivalentes representan situaciones semejantes en el procesamiento de una cadena y por tanto se pueden unir o agrupar en uno único sin que cambie el lenguaje aceptado por el AFD, reduciendo así el número de estados del autómata. Ésta es la idea subyacente en el proceso de minimización de un AFD.

Ejemplo 13 *En el siguiente AFD M_1 nos damos cuenta de forma intuitiva de que hay dos parejas de estados que son equivalentes: $p_4 \equiv p_8$ y $p_5 \equiv p_9$, porque no hay cadena que los distinga, se cumple la condición de equivalencia. Cada pareja de estados equivalentes se une en un único estado y de esa manera el autómata M_2 resultante es equivalente a M_1 y tiene dos estados menos.*



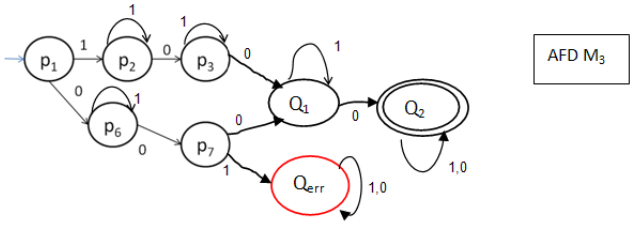
Cuando un AFD es complejo, puede ser difícil descubrir "a ojo" los estados que son equivalentes entre sí. En el algoritmo de minimización que vamos a ver, en lugar de tratar de identificar los estados equivalentes lo que hace es descubrir parejas de estados distinguibles, a partir de que se sabe que otras parejas de estados son distinguibles. Para eso se parte de un AFD completo (si no lo es se completa con un estado de error) y es necesario saber inicialmente qué parejas podemos asegurar que son distinguibles. La idea para **deducir qué parejas de estados son distiguibles en un AFD completo** es la siguiente:

- Regla base: si en una pareja de estados (q_i, q_j) uno de los estados es final y el otro no entonces los estados q_i, q_j son distinguibles.
- Regla deductiva: si tenemos que $\delta(q_i, a) = q'_i$ y $\delta(q_j, a) = q'_j$, para cierto símbolo $a \in V$, y se sabe que los estados q'_i, q'_j son distinguibles entonces se deduce que los estados q_i y q_j también son distinguibles.

La regla deductiva se justifica porque si q'_i, q'_j son distinguibles entonces, por definición, existe una cadena x que los distingue, es decir, se tiene que: $(q'_i, x) \Rightarrow^* (q''_i, \lambda)$ y $(q'_j, x) \Rightarrow^* (q''_j, \lambda)$ donde en la pareja de estados (q''_i, q''_j) uno de ellos es final y el otro no. Entonces la cadena ax distingue a los estados q_i y q_j, como se puede comprobar en los siguientes cálculos:

$$(q_i, ax) \overset{\delta(q_i,a)=q'_i}{\Rightarrow} (q'_i, x) \Rightarrow^* (q''_i, \lambda) \quad \text{y} \quad (q_j, ax) \overset{\delta(q_j,a)=q'_j}{\Rightarrow} (q'_j, x) \Rightarrow^* (q''_j, \lambda)$$

Ejemplo 14 *Partiendo del autómata M_2 del ejemplo anterior, vamos a descubrir que algunas parejas de estados son distinguibles. Para aplicar la regla deductiva es necesario que obtengamos primero un AFD completo M_3 equivalente a M_2, que es el siguiente:*



Siguiendo las reglas para deducir estados distinguibles llegamos a que:

1. *Los estados Q_2 y Q_{err} son distinguibles porque uno es final y otro no y por el mismo motivo Q_1 y Q_2 son distinguibles.*
2. *Como $\delta(Q_1, 0) = Q_2$ y $\delta(Q_{err}, 0) = Q_{err}$ y sabemos (paso 1) que Q_2 y Q_{err} son distinguibles entonces se deduce que Q_1 y Q_{err} son distinguibles.*
3. *Como $\delta(p_3, 0) = Q_1$ y $\delta(Q_{err}, 0) = Q_{err}$ y sabemos (paso 2) que Q_1 y Q_{err} son distinguibles entonces se deduce que p_3 y Q_{err} son distinguibles.*
4. *Tenemos que $\delta(p_3, 0) = Q_1$ y $\delta(p_7, 0) = Q_1$. Como se obtiene el mismo estado no se puede afirmar que p_3 y p_7 sean distinguibles. Sin embargo, considerando las transiciones con el símbolo 1 tenemos: $\delta(p_3, 1) = p_3$ y $\delta(p_7, 1) = Q_{err}$ y sabemos (paso 3) que p_3 y Q_{err} son distinguibles, luego p_3 y p_7 son distinguibles. Si no se hubiera completado el autómata con el estado de error no habríamos llegado a la conclusión de que p_3 y p_7 son distinguibles.*

A partir de la definición de estados distinguibles también se llega a la conclusión de que p_3 y p_7 son distinguibles porque la cadena $x = 100$ los distingue, ya que:

$$(p_3, 100) \Rightarrow (p_3, 00) \Rightarrow (Q_1, 0) \Rightarrow (Q_2, \lambda) \qquad [Q_2 \text{ es final}]$$
$$(p_7, 100) \Rightarrow (Q_{err}, 00) \Rightarrow (Q_{err}, 0) \Rightarrow (Q_{err}, \lambda) \qquad [Q_{err} \text{ no es final}]$$

Analizando el resto de parejas de estados se puede probar que todos los estados de M_3 son distinguibles entre sí, así que el autómata es mínimo.

Algoritmo de marcado de tabla triangular

En la Figura 6 mostramos un algoritmo que considera cada pareja de estados de un AFD y averigua si son distinguibles o equivalentes. Se utiliza una **tabla triangular** T para marcar casillas correspondientes a pares de estados que se descubre que son distinguibles. Usamos la notación $casilla(q_i, q_j)$ para hacer referencia a la entrada de la tabla $T[q_i, q_j]$ cuando $i > j$, ó a $T[q_j, q_i]$ cuando $i < j$. Se usa la tabla triangular para no duplicar información con los pares (q_i, q_j) y (q_j, q_i), pues $q_i \equiv q_j$ si y sólo si $q_j \equiv q_i$. En la tabla sólo se considera una entrada para esos dos pares. Si $casilla(q_i, q_j)$ se llega a marcar entonces los estados q_i, q_j **son distinguibles y en otro caso son equivalentes.**
Cada entrada de T correspondiente a $casilla(q_i, q_j)$, además de tener un variable booleana asociada (marcada=true o false), tiene asociada una **lista de parejas de estados**, a la que nos referiremos como $lista(q_i, q_j)$, donde se guardan las parejas cuya distinguibilidad depende de la del par (q_i, q_j), es decir, que si durante el recorrido de la tabla se llega a marcar la $casilla(q_i, q_j)$ (son distinguibles), de ahí se deduce que todos los pares en espera en $lista(q_i, q_j)$ son distinguibles, por lo que habría que marcar las casillas correspondientes.

El algoritmo primero **elimina los estados inaccesibles** y **completa el autómata** como paso previo. Después se hace un **marcado inicial** de la tabla para señalar las parejas que se sabe inicialmente que son distinguibles: marca cada casilla correspondiente a una pareja donde uno es final y el otro no. Luego realiza un bucle para la deducción de estados distinguibles mediante un proceso de **recorrido de la tabla con marcado recursivo**. La función *marcarRecursivamente*($casilla(q_i, q_j)$) marca la $casilla(q_i, q_j)$ y marca todas las casillas de las parejas en $lista(q_i, q_j)$ y a su vez las casillas de las parejas que aparecen en las listas de cada casilla accedida y así sucesivamente hasta que no se puedan marcar más.
Después del recorrido y marcado de la tabla ya se sabe que las casillas sin marcar corresponden a pares de estados equivalentes y las marcadas a pares de estados distinguibles.

FUNCIÓN marcarTablaTriangular (M)

ENTRADA: un AFD $M = (Q, V, \delta, q_0, F)$ con $Q = \{q_0, \ldots, q_n\}$, $V = \{a_1, \ldots, a_p\}$.
DEVUELVE: una tabla triangular T, con las parejas de estados distinguibles marcadas.

1. $M \leftarrow eliminaInaccesibles(M)$; //modifica M para que no tenga estados inaccesibles.
2. $M \leftarrow completaAFD(M)$; //modifica M para que sea un AFD completo.
3. Construye una tabla T con filas desde q_1 (segundo estado) hasta q_n (último) y columnas desde q_0 (primer estado) hasta q_{n-1} (penúltimo);
4. PARA CADA $casilla(q_i, q_j)$ de T HACER:
5. Asocia una $lista(q_i, q_j)$ de parejas de estados //inicialmente vacía

/* **Marcado inicial de estados distinguibles** */

6. PARA CADA $casilla(q_i, q_j)$ de T HACER:
7. SI en la pareja (q_i, q_j) un estado es final y el otro no
   ENTONCES $marcar(casilla(q_i, q_j))$;

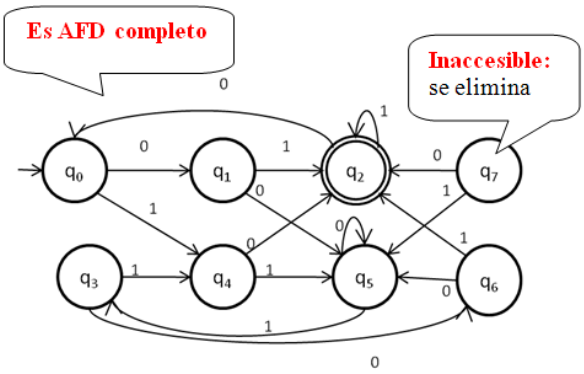
/* **Deducción de estados distinguibles** */

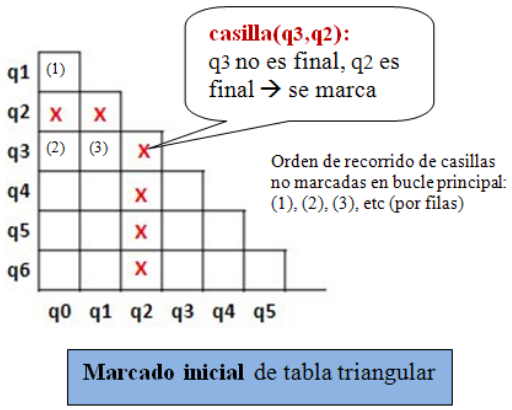
8. PARA CADA $casilla(q_i, q_j)$ de T HACER: //ej., realizar recorrido por filas de la tabla
9. SI $casilla(q_i, q_j)$ no marcada ENTONCES
10. PARA CADA símbolo a_k del alfabeto HACER:
11. $e1 := \delta(q_i, a_k)$; $e2 := \delta(q_j, a_k)$; // obtiene pareja de estados leyendo a_k
12. SI $casilla(e1, e2)$ está marcada ENTONCES
13. *marcarRecursivamente* $(casilla(q_i, q_j))$;
14. BREAK; //no prueba con más símbolos, porque (q_i, q_j) son distinguibles
    // pasa a tratar la siguiente casilla
15. SI-NO SI $(e1 \neq e2)$ y $casilla(e1, e2) \neq casilla(q_i, q_j)$ ENTONCES
16. añade la pareja actual (q_i, q_j) a la lista de pareja obtenida: $lista(e1, e2)$;
17. FIN-PARA CADA
18. FIN-SI
19. FIN-PARA CADA //fin de marcado de tabla triangular
20. DEVUELVE (T);

//ORDEN DE COMPLEJIDAD: $O(n^2)$ (con pasos optimizados), donde $n = |Q|$

Figura 6: **Algoritmo *marcarTablaTriangular***

Ejemplo 15 *En la siguiente figura mostramos un AFD y la aplicación de los primeros pasos del algoritmo de marcado de tabla triangular: eliminar estados inaccesibles, completar el AFD (en este caso es completo), construir la tabla triangular y hacer el marcado inicial de la tabla:*

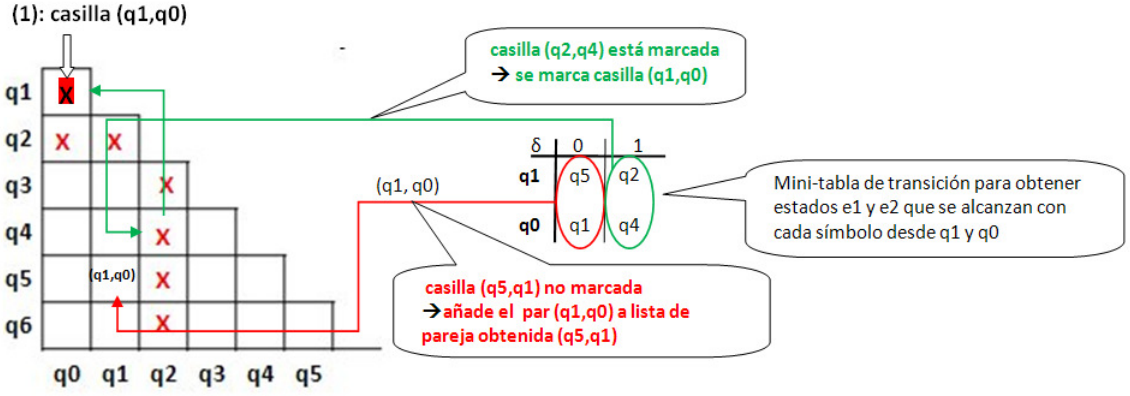




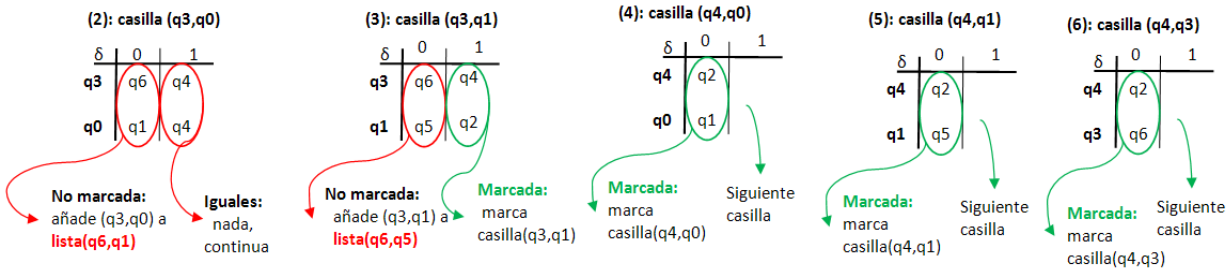
Después hacemos un recorrido por filas de la tabla, visitando cada casilla no marcada. El orden del recorrido podría ser otro, el caso es que se visiten todas las casillas. Aplicamos los pasos del algoritmo a la primera casilla que se visita:

(1): $\underline{casilla(q_1, q_0)}$*:*
▷ *Para el símbolo* 0 *se obtiene que* $e1$ *es* $q_5 = \delta(q_1, 0)$ *y* $e2$ *es* $q_1 = \delta(q_0, 0)$*. Consulta* $casilla(q_5, q_1)$ *y no está marcada* $\rightsquigarrow$ *añade el par* (q_1, q_0) *a* $lista(q_5, q_1)$*.*
▷ *Para el símbolo* 1 *se obtiene* $q_2 = \delta(q_1, 1)$ *y* $q_4 = \delta(q_0, 1)$*. Consulta* $casilla(q_2, q_4)$ *y está marcada* $\rightsquigarrow$ *marca* $casilla(q_1, q_0)$ *[son distinguibles]. El marcado recursivo sólo marca la* $casilla(q_1, q_0)$ *porque la* $lista(q_1, q_0)$ *está vacía* $\rightsquigarrow$ *no tiene pares de estados pendientes de marcar.*

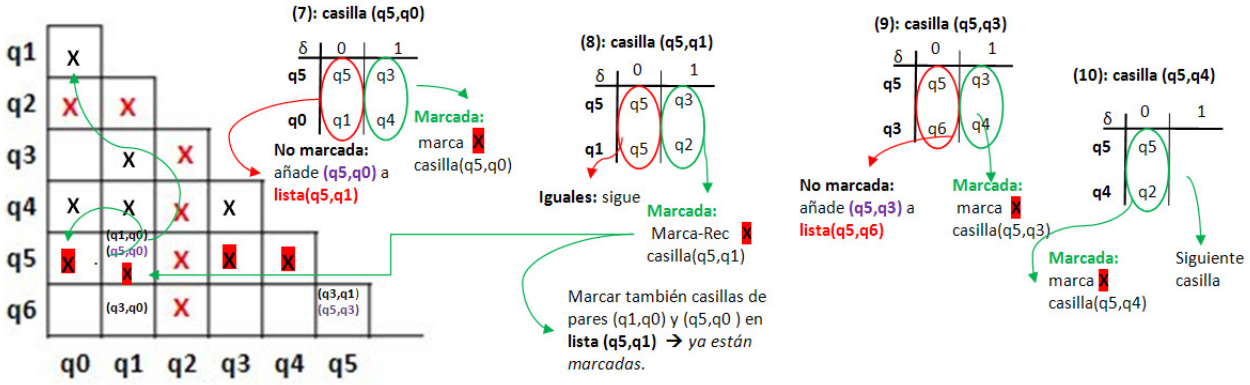
Los pasos anteriores los representamos gráficamente del siguiente modo:



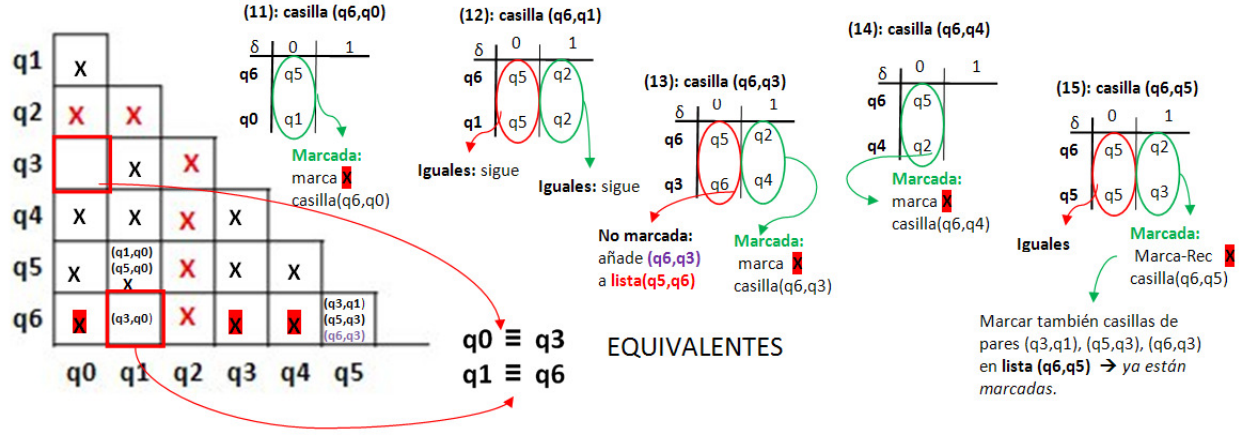
Seguimos con las casillas no marcadas de la fila del estado q_3 *y del estado* q_4*:*



Visitamos las casillas no marcadas de la fila del estado q_5 y mostramos la tabla con apuntes en listas y marcas de pares distinguibles de pasos anteriores y los nuevos de esta iteración:



Finalmente visitamos las casillas de la fila del estado q_6 y la tabla al final de la función de marcado queda con todas las casillas marcadas excepto casilla(q_0, q_3) y casilla(q_6, q_1), lo cual indica que $q_0 \equiv q_3$ y $q_1 \equiv q_6$.



Método de construcción del autómata mínimo

Hasta este punto sabemos cómo averiguar en un AFD qué parejas de estados son equivalentes y cuáles distinguibles. El siguiente paso en la minimización de un AFD consiste en obtener los componentes del AFD mínimo equivalente a partir de la tabla triangular que contiene la información sobre qué estados son equivalentes entre sí.

El concepto de equivalencia de pares de estados define una **RELACIÓN DE EQUIVALENCIA** $\equiv$

en el conjunto Q de estados de un AFD. Como toda relación de equivalencia, la relación $\equiv$ es *reflexiva, simétrica y transitiva*:

$$\begin{aligned}&\forall p \in Q: \quad p \equiv p\\ &\forall p, q \in Q: \quad p \equiv q \Rightarrow q \equiv p\\ &\forall p, q, r \in Q: \quad (p \equiv q \wedge q \equiv r) \Rightarrow p \equiv r\end{aligned}$$

Para reducir el número de estados de un AFD **se unen todos los estados equivalentes entre sí** en un único estado, obteniendo el **conjunto cociente** de Q bajo la relación de equivalencia $\equiv$, que se denota como $\boxed{Q/_{\equiv}}$. Cada elemento de $Q/_{\equiv}$ es un **estado-clase de equivalencia** o subconjunto de estados de Q que son equivalentes entre sí.

Los ***componentes del AFD mínimo***, $M_{min} = (Q_{min}, V, \delta_{min}, q_{0min}, F_{min})$, equivalente a un AFD $M = (Q, V, \delta, q_0, F)$, se obtienen siguiendo los siguientes pasos:

1. **Conjunto de estados del AFD mínimo**: Q_{min} será el conjunto cociente $Q/_{\equiv}$ y se obtiene haciendo un recorrido por la tabla marcada y teniendo en cuenta que la relación de equivalencia de estados es transitiva. Ej. si tenemos que $casilla(q_i, q_j)$ no está marcada y $casilla(q_k, q_j)$ tampoco entonces se deduce que $q_i \equiv q_j \equiv q_k$ y si no hay más estados equivalentes a estos tendríamos el estado-clase $\{q_i, q_j, q_k\}$ en $Q/_{\equiv}$, que puede denotarse también como $[q_i]$, que es la clase de equivalencia de representante q_i, es decir, $[q_i] = \{q_i, q_j, q_k\}$. Nosotros usaremos una notación explícita para un estado-clase, como subconjunto donde se listan todos los estados de la clase de equivalencia, o sea, usamos la notación $\{q_i, q_j, q_k\}$ en lugar de $[q_i]$. Hay que tener en cuenta que el orden de los estados dentro del estado-clase no influye, así que $\{q_i, q_j, q_k\}$ es el mismo estado que $\{q_k, q_j, q_i\}$. Si todas las casillas de la tabla están marcadas entonces el número de estados no se reduce, porque son todos distinguibles y el autómata de entrada ya es mínimo. Si ninguna casilla está marcada es que todos los estados son equivalentes y $Q/_{\equiv}$ sólo tendrá un estado-clase: el subconjunto con todos los estados de Q.

2. **Conjunto de estados finales del AFD mínimo**: F_{min} está formado por aquellos estados-clase de Q_{min} que contiene un estado final del AFD de entrada, es decir:

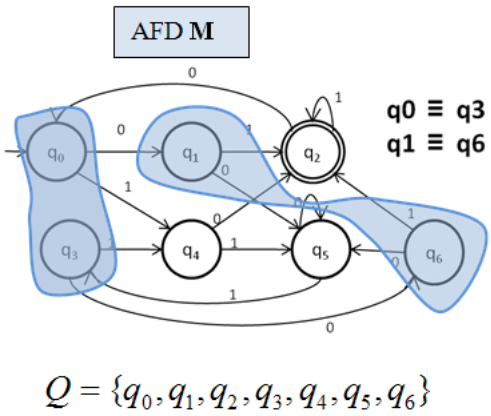
$$F_{min} = \{S \in Q_{min} \mid S \cap F \neq \varnothing\}$$

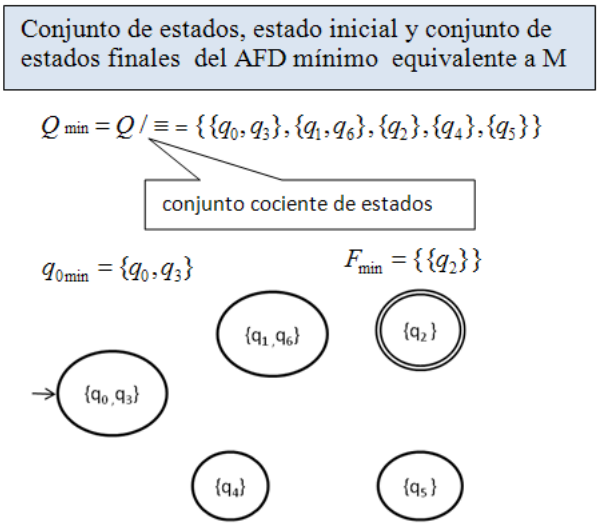
3. **Estado inicial del AFD mínimo**: q_{0min} es el estado-clase de Q_{min} que contiene al estado inicial original.

4. **Función de transición del AFD mínimo**: $\delta_{min} : Q_{min} \times V \longrightarrow Q_{min}$ se define para todo estado-clase S de Q_{min} y todo símbolo a de V de la siguiente forma:
   • Se escoge un estado cualquiera q como representante del estado-clase S.
   • Se obtiene la transición $\delta_{min}(S, a)$, a partir de la regla de transición del AFD de entrada $\delta(q, a)$, como el estado-clase S′ que contiene el estado dado por $\delta(q, a)$, por lo que:

$$\delta_{min}(S, a) = \mathrm{S}' \in Q_{min} \text{ tal que } \delta(q, a) \in \mathrm{S}'$$

Ejemplo 16 *Una vez deducido que $q_0 \equiv q_3$ y $q_1 \equiv q_6$ en el ejemplo 15, ahora se obtiene el conjunto cociente, el estado inicial y los estados finales del autómata mínimo:*



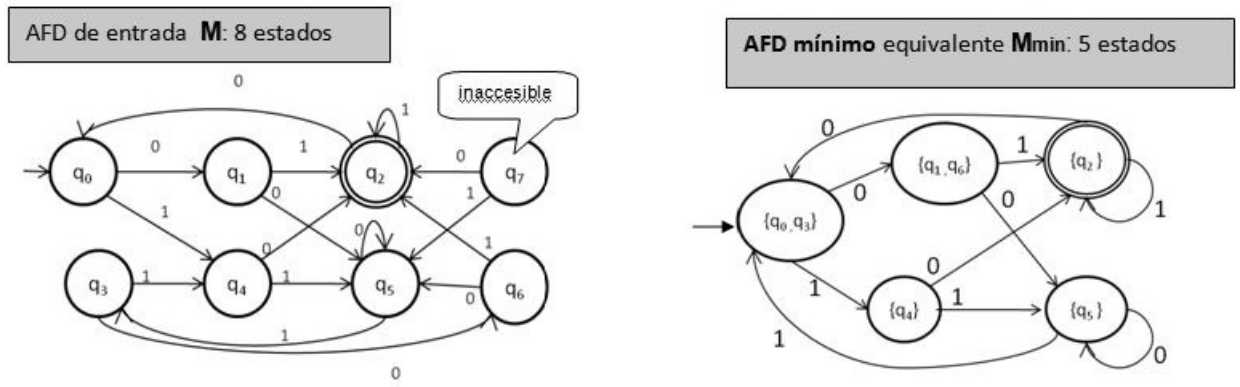


Falta calcular la función de transición δ_{min} a partir de la función δ del AFD original descrita en el diagrama de transición M de este ejemplo (se incluye de nuevo más abajo):

- *Para $S = \{q_0, q_3\}$ (inicial) de Q_{min} escogemos q_0 y definimos nuevas transiciones en δ_{min} con símbolos 0 y 1:*
  - *Puesto que en el AFD original tenemos que $\delta(q_0, 0) = q_1$ y el estado q_1 pertenece al estado-clase $\{q_1, q_6\}$ de Q_{min}, entonces* $\boxed{\delta_{min}(\{q_0, q_3\}, 0) = \{q_1, q_6\}}$
  - *Puesto que $\delta(q_0, 1) = q_4$ y $q_4 \in \{q_4\}$ de Q_{min} entonces* $\boxed{\delta_{min}(\{q_0, q_3\}, 1) = \{q_4\}}$
- *Para $S = \{q_1, q_6\} \in Q_{min}$ escogemos q_1 y calculamos nuevas transiciones:*
  - *Como $\delta(q_1, 0) = q_5$ y $q_5 \in \{q_5\}$ de Q_{min}, entonces* $\boxed{\delta_{min}(\{q_1, q_6\}, 0) = \{q_5\}}$
  - *Como $\delta(q_1, 1) = q_2$ y $q_2 \in \{q_2\}$ de Q_{min} entonces* $\boxed{\delta_{min}(\{q_1, q_6\}, 1) = \{q_2\}}$
- *Para $S = \{q_4\} \in Q_{min}$ escogemos q_4 y calculamos nuevas transiciones:*
  - *Como $\delta(q_4, 0) = q_2$ y $q_2 \in \{q_2\}$ de Q_{min}, entonces* $\boxed{\delta_{min}(\{q_4\}, 0) = \{q_2\}}$
  - *Como $\delta(q_4, 1) = q_5$ y $q_5 \in \{q_5\}$ de Q_{min}, entonces* $\boxed{\delta_{min}(\{q_4\}, 1) = \{q_5\}}$
- *Para $S = \{q_5\} \in Q_{min}$ escogemos q_5 y calculamos nuevas transiciones:*
  - *Como $\delta(q_5, 0) = q_5$ y $q_5 \in \{q_5\}$ de Q_{min}, entonces* $\boxed{\delta_{min}(\{q_5\}, 0) = \{q_5\}}$
  - *Como $\delta(q_5, 1) = q_3$ y $q_3 \in \{q_0, q_3\}$ de Q_{min}, entonces* $\boxed{\delta_{min}(\{q_5\}, 1) = \{q_0, q_3\}}$
- *Para $S = \{q_2\} \in Q_{min}$ escogemos q_2 y calculamos nuevas transiciones:*
  - *Como $\delta(q_2, 0) = q_0$ y $q_0 \in \{q_0, q_3\}$ de Q_{min}, entonces* $\boxed{\delta_{min}(\{q_2\}, 0) = \{q_0, q_3\}}$

- *Como* $\delta(q_2, 1) = q_2$ *y* $q_2 \in \{q_2\}$ *de* Q_{min}*, entonces* $\boxed{\delta_{min}(\{q_2\}, 1) = \{q_2\}}$

Finalmente mostramos el AFD con el que se inició el proceso de aplicación del algoritmo de minimización en el ejemplo 15 y el AFD mínimo equivalente que se obtiene.



En la figura 7 mostramos el **algoritmo de minimización completo**, que usa los métodos de eliminar estados inaccesibles, completar automata, marcado de tabla triangular y finalmente obtiene los componentes del AFD mínimo.

FUNCIÓN minimizaAFD(M)

ENTRADA: un autómata finito determinista $M = (Q, V, \delta, q_0, F)$.
DEVUELVE: $M_{min} = (Q_{min}, V, \delta_{min}, q_{0min}, F_{min})$ tal que
M_{min} es mínimo y $L(M_{min}) = L(M)$ (equivalente a M).

/* **Aplica el algoritmo de marcado de tabla triangular** al autómata M. Este método previamente elimina estados inaccesibles y completa el autómata M */

1. $T \leftarrow marcarTablaTriangular(M)$;

   /* **Obtiene el conjunto de estados del autómata mínimo** */

2. Obtener el conjunto cociente $Q/_{\equiv}$ a partir de la tabla T y HACER $Q_{min} := Q/_{\equiv}$;

   /* **Obtiene el estado inicial del autómata mínimo:** es el estado-clase S que contiene al estado inicial original */

3. $q_{0min} := S \in Q_{min}$ tal que $q_0 \in S$;

   /* **Obtiene el conjunto de estados finales del autómata mínimo** */

4. $F_{min} := \{S \in Q_{min} \mid S \cap F \neq \varnothing\}$; // estado-clase que contienen un final de M

   /* **Obtiene la función de transición del autómata mínimo** */

5. PARA TODO estado-clase S de Q_{min} y todo símbolo a de V HACER

   Se escoge un estado cualquiera $q \in S$;

   Se obtiene la transición:

   $$\delta_{min}(S, a) := \mathrm{S}' \in Q_{min} \text{ tal que } \delta(q, a) \in \mathrm{S}';$$

6. DEVUELVE (M_{min})

//ORDEN DE COMPLEJIDAD: $O(n^2)$, donde $n = |Q|$.

Figura 7: **Algoritmo *minimizaAFD***

Teorema 1 (Unicidad del autómata mínimo) *Dado un autómata finito determinista M, existe un **único AFD mínimo completo** equivalente a M y se puede obtener mediante el algoritmo de minimización anterior.*

No vamos a ver la demostración de este teorema, simplemente decir que hay diversos métodos para calcular el autómata mínimo pero todos ellos obtienen como resultado el mismo autómata, salvo isomorfismos, es decir, sólo se pueden diferenciar en el nombre de los estados, no en las transiciones y eso claramente no afecta al lenguaje aceptado.

1.8. Equivalencia de AFDs

Puesto que sabemos por el teorema de unicidad que el autómata mínimo equivalente a un AFD es único salvo isomorfismos, podemos diseñar un algoritmo, que llamamos $equivalentesAFD$, que comprueba si dos AFDs son equivalentes:

FUNCIÓN **equivalentesAFD**$(\mathbf{M_1}, \mathbf{M_2})$

ENTRADA: dos AFDs M_1 y M_2
DEVUELVE: *true* si $L(M_1) = L(M_2)$ (los AFDs son equivalentes) y *false* en otro caso.

1. $M_{1min} := minimizaAFD(M_1)$; //obtiene AFD mínimo completo equivalente a M_1
2. $M_{2min} := minimizaAFD(M_2)$; //obtiene AFD mínimo completo equivalente a M_2
3. SI M_{1min} y M_{2min} son isomorfos (iguales salvo renombramiento de estados)
   ENTONCES DEVOLVER $(true)$;
   SI-NO DEVOLVER $(false)$;

Para comprobar si dos AFDs son isomorfos se puede aplicar un algoritmo típico de isomorfismo de grafos (no se ve en esta asignatura) a los diagramas de transición de los autómatas. Este algoritmo puede ser costoso. Hay otro método más eficiente que comprueba si dos AFDs son equivalentes en tiempo cuadrático, en función del máximo número de estados de los dos autómatas. Este método lo omitimos, lo que interesa ahora es saber que el problema de comprobar si dos AFDs son equivalentes tiene solución algorítmica.

2. Autómatas finitos no deterministas

Un *autómata finito no determinista* (AFND) (*non deterministic finite automaton NFA*) se define a partir de una estructura matemática $M = (Q, V, \delta, q_0, F)$ donde todos los componentes son como en los autómatas deterministas, excepto la función de transición, que se define ahora:

$$\boxed{\delta : Q \times (V \cup \{\lambda\}) \longrightarrow \mathcal{P}(Q)}$$

donde $\mathcal{P}(Q)$ es el conjunto de las partes de Q, formado por todos los subconjuntos de Q.

El **"no determinismo"** significa que a partir de una configuración es posible que se puedan alcanzar **multiples estados** o **múltiples configuraciones** en el instante siguiente, en un único paso de cálculo, lo cual supone que puede haber diferentes formas de procesar una cadena, o lo que es lo mismo, diferentes cálculos.

2.1. Tipos de reglas de transición en un AFND

Según el dominio y codominio de la función de transición $\delta : Q \times (V \cup \{\lambda\}) \longrightarrow \mathcal{P}(Q)$ podemos distinguir los siguientes tipos de reglas:

- **Regla determinista.**

  ▷ SINTAXIS: es del tipo $\delta(q, a) = \{q'\}$, donde $q, q' \in Q, a \in V$. Es como la de un AFD excepto que cambia la notación por ser el codominio de la función de transición $\mathcal{P}(Q)$ en lugar de Q, por eso se pone el estado q' entre llaves.

  ▷ SEMÁNTICA: si la configuración del autómata en cierto instante es (q, az) entonces al aplicar esta regla se produce el paso de cálculo $(q, az) \Rightarrow (q', z)$.

- **Regla no determinista con lectura de símbolo y múltiples opciones** para cambio de estado.

  ▷ SINTAXIS: una de regla de este tipo es $\delta(q, a) = \{q_1, q_2, q_3\}$, que en el diagrama de transición se refleja con tres arcos que parten del estado q etiquetados con el símbolo a, uno hacia el nodo del estado q_1, otro a q_2 y otro a q_3.

  ▷ SEMÁNTICA: si la configuración es (q, az) con $a \in V, z \in V^*$ entonces por la regla $\delta(q, a) = \{q_1, q_2, q_3\}$ se puede elegir entre cambiar al estado q_1 o a q_2 o a q_3 y en el instante siguiente se tendrían tres configuraciones distintas posibles, como si se realizara un *paso de cálculo paralelo*:

$$(q, az) \begin{cases} \overset{\text{por ser } q_1 \in \delta(q,a)}{\overbrace{\Rightarrow}} (q_1, z) \\ \overset{\text{por ser } q_2 \in \delta(q,a)}{\overbrace{\Rightarrow}} (q_2, z) \\ \overset{\text{por ser } q_3 \in \delta(q,a)}{\overbrace{\Rightarrow}} (q_3, z) \end{cases}$$

- **Regla no determinista sin lectura de símbolo** o **(λ-transición)**.

  ▷ SINTAXIS: puede ser de una opción para cambio de estado, como $\delta(p, \lambda) = \{p'\}$ o con múltiples opciones, como $\delta(p, \lambda) = \{p_3, p_4\}$.

  ▷ SEMÁNTICA: si la configuración es (p, z) entonces al aplicar la regla $\delta(p, \lambda) = \{p'\}$ se produce el paso de cálculo $(p, z) \Rightarrow (p', z)$, donde no se lee símbolo o no se avanza en la entrada. Obsérvese que z puede ser λ, con lo que tendríamos el paso $(p, \lambda) \Rightarrow (p', \lambda)$.

  Aunque una λ-transición sólo tenga una opción para el cambio de estado, este tipo de reglas también **son causa de no determinismo**. Ej. si tenemos definida la regla $\delta(p, \lambda) = \{p'\}$ y también otra regla como $\delta(p, a) = \{p''\}$. entonces a partir de la configuración (p, az) hay dos cálculos posibles en un paso:

$$(p, az) \begin{cases} \overset{\text{por ser } p' \in \delta(p,\lambda)}{\overbrace{\Rightarrow}} (p', az) \\ \overset{\text{por ser } p'' \in \delta(p,a)}{\overbrace{\Rightarrow}} (p'', z) \end{cases}$$

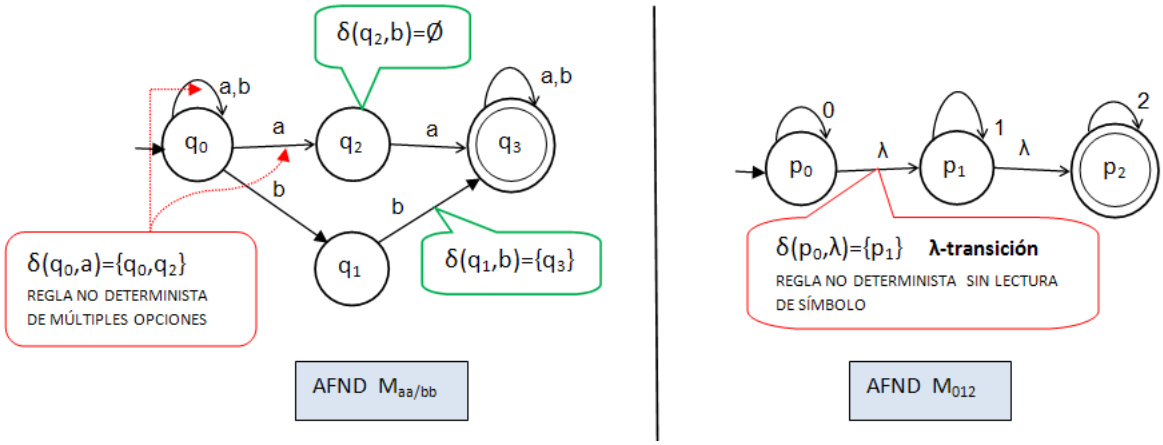
  El no determinismo se hace más evidente si la λ-transición tiene múltiples opciones, como en el caso de la regla $\delta(p, \lambda) = \{p_3, p_4\}$, con la que se puede elegir entre cambiar al estado

p_3 o al p_4, de forma análoga a lo que ocurre al aplicar una regla no determinista con lectura de símbolo y múltiples opciones.

- **Regla vacía** tipo $\delta(q, a) = \varnothing$, o bien, $\delta(q, \lambda) = \varnothing$. La regla $\delta(q, a) = \varnothing$ es como tener indefinido el caso $\delta(q, a)$ en un AFD incompleto, porque el conjunto vacío $\varnothing \in \mathcal{P}(Q)$ no contiene un estado válido de Q.

  $\triangleright$ SEMÁNTICA: si la configuración es (q, az) y $\delta(q, \lambda) = \varnothing$, $\delta(q, a) = \varnothing$ entonces no se puede cambiar de estado, con lo cual (q, az) sería una configuración de parada.

Ejemplo 17 *Los siguientes diagramas de transición corresponden a AFNDs, porque tienen reglas no deterministas. El autómata $M_{aa/bb}$ es un AFND sin λ-transiciones porque no tiene ningún arco etiquetado con λ. El autómata M_{012} tiene λ-transiciones pero no tiene reglas de transición de múltiples opciones.*



2.2. Funcionamiento un AFND. Lenguaje aceptado por un AFND

Un AFDN funciona procesando una cadena de entrada símbolo a símbolo y aplicando reglas de transición que producen cambio de estado y avance opcional en la entrada, hasta que el autómata para cuando no es posible aplicar ninguna transición. El **modo de funcionamiento no determinista** puede interpretarse como que el autómata, como máquina teórica, lleva a cabo un *cálculo paralelo*, de manera que si en cierto instante de ejecución hay varias opciones de cambio de estado, por transiciones con múltiples opciones o λ-transiciones, entonces el cálculo se ramifica generando distintos cálculos secuenciales o hilos que se ejecutan de forma independiente. En el momento que uno de los posibles cálculos (secuenciales) alcanza un estado final después de leer la cadena (acaba en configuración de aceptación) entonces termina la ejecución del autómata y la cadena de entrada se acepta y si ninguno de los cálculos acaba en configuración de aceptación entonces la cadena se rechaza.

Informalmente, un AFND **acepta una cadena** w si existe un camino en el diagrama de transición que parte del estado inicial y llega a un estado final leyendo los símbolos de la cadena, seguidos o pasando por arcos etiquetados con λ antes o después de cada símbolo. Esto equivale a decir, más formalmente, que e**xiste al menos un cálculo para** w **que parte de la configuración inicial y acaba en configuración de aceptación**, es decir, existe un cálculo tal que $(q_0, w) \Rightarrow^* (q_F, \lambda)$, donde q_0 es el estado inicial y q_F es un estado final. Si no existe ningún cálculo que acabe en configuración de aceptación entonces **rechaza la cadena**.

El tiempo de ejecución de un AFND con una cadena válida se mide por el número de pasos del cálculo paralelo, que coincide con el número de pasos del cálculo más corto posible (rama del árbol de cálculo de menor longitud) por el que se acepta la cadena. A diferencia de un AFD, en un AFND que tenga λ-transiciones el número de pasos por el que se acepta una cadena no tiene por qué coincidir con la longitud de la cadena, porque puede haber pasos extra en los que no se lean símbolos. El tiempo de ejecución para una cadena no válida es el número de pasos del cálculo más largo. En cualquier caso, para cadenas de longitud n el tiempo de ejecución teórico es de $O(n)$.

El conjunto de todas las cadenas aceptadas por un AFND forma un lenguaje, que se define formalmente de forma análoga al lenguaje aceptado por un AFD:

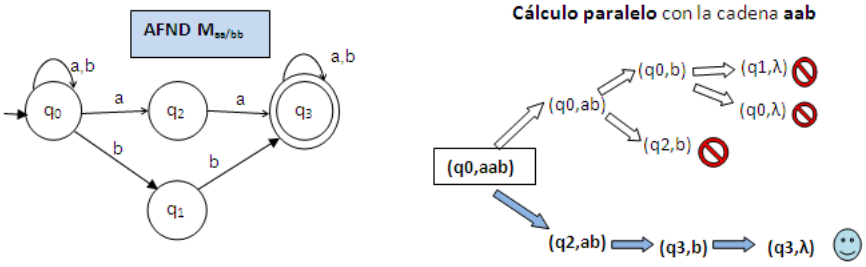
Definición 10 (lenguaje aceptado por un AFND) *Dado un autómata finito no determinista $M = (Q, V, \delta, q_0, F)$, el* LENGUAJE ACEPTADO POR *M, que denotamos $L(M)$, se define como:*

$$L(M) = \{w \in V^* \mid (q_0, w) \Rightarrow^* (q_F, \lambda), q_F \in F\}$$

Se tiene que $\lambda \in L(M)$ si y sólo si, por definición, $(q_0, \lambda) \Rightarrow^* (q_F, \lambda), q_F \in F$. A diferencia de un AFD, en un AFND con λ-transiciones, el cálculo implícito $(q_0, \lambda) \Rightarrow^* (q_F, \lambda)$ puede corresponder a una secuencia de uno o más pasos de cálculo, así que q_0 no es necesariamente igual a q_F y la cadena vacía puede ser aceptada aunque q_0 no sea un estado final.

El proceso para **analizar qué lenguaje acepta** un AFND es similar al de un AFD, hay que comprobar todos los posibles caminos en el diagrama de transición que llevan a un estado final y tratar de identificar el patrón de las cadenas que se leen por cada camino.

Ejemplo 18 *Consideremos de nuevo el AFND $M_{aa/bb}$:*



Se muestra un ***árbol de cálculo paralelo*** *con la cadena de entrada aab y todas las posibles configuraciones en cada paso debido a la aplicación de diferentes opciones de cambio de estado en reglas no deterministas.* ***Cada rama del árbol se corresponde con un cálculo distinto.*** *El cálculo por el que se acepta la cadena aab es el siguiente (se muestran además las opciones de las reglas de transición aplicadas en cada paso):*

$$(q_0, aab) \overset{q_2 \in \delta(q_0,a)}{\Rightarrow} (q_2, ab) \overset{q_3 \in \delta(q_2,a)}{\Rightarrow} (q_3, b) \overset{q_3 \in \delta(q_3,b)}{\Rightarrow} (q_3, \lambda) \quad (\text{ACEPTA EN 3 PASOS})$$

Este cálculo se corresponde con el camino del diagrama:

$$q_0 \xrightarrow{a} q_2 \xrightarrow{a} q_3 \xrightarrow{b} q_3$$

Podemos observar que hay otro camino en el diagrama que no acaba en estado final:

$$q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1$$

y también un camino por el que el autómata para sin detectar fin de cadena de entrada:

$$q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_2 \quad [\text{para porque } \delta(q_2, b) = \varnothing]$$

El caso es que al existir un camino que acaba en estado final la cadena aab es aceptada.

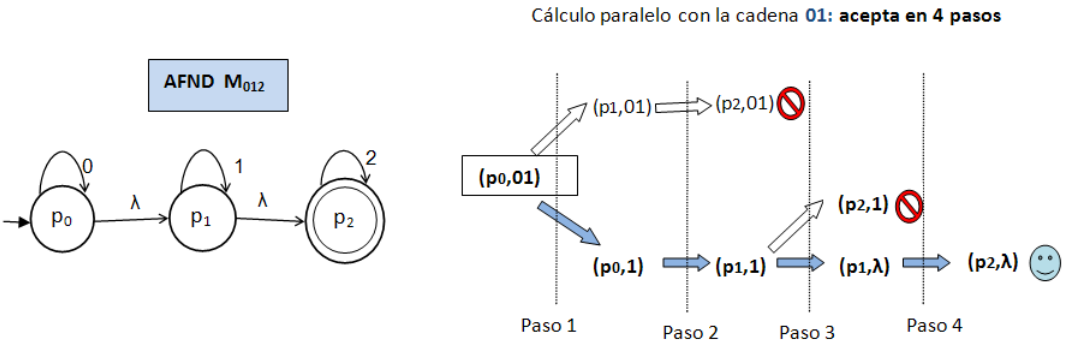
Analicemos cuál es el ***lenguaje aceptado por este autómata****. En el estado q_0 se puede leer cualquier secuencia de a's/b's. Seguidamente hay dos opciones:*
1. Leer la subcadena aa para alcanzar el estado q_3.
2. Leer la subcadena bb para alcanzar el estado q_3.
Una vez que se alcanza q_3 se puede seguir leyendo cualquier secuencia de a's/b's y la cadena de entrada es aceptada.

Deducimos que $M_{aa/bb}$ acepta "cadenas de a's/b's que contienen la subcadena aa o la subcadena bb". Se puede comprobar que el autómata acepta todas las cadenas de este tipo y rechaza cadenas que no son de ese tipo. Entonces puede afirmarse que:

$$L(M_{aa/bb}) = \{w \in \{a,b\}^* \mid w = xaay \vee w = xbby, \text{ donde } x, y \in \{a,b\}^*\}$$

Las cadenas de menor longitud aceptadas son precisamente las propias subcadenas aa y bb, para las que $x = y = \lambda$ en la definición del lenguaje.

Ejemplo 19 *Consideremos ahora el AFND M_{012}:*



Se muestra un ***árbol de cálculo paralelo*** *con la cadena de entrada 01. El cálculo por el que se acepta la cadena 01, de los tres posibles, es:*

$$(p_0, 01) \overset{p_0 \in \delta(p_0,0)}{\Rightarrow} (p_0, 1) \overset{p_1 \in \delta(p_0,\lambda)}{\Rightarrow} (p_1, 1) \overset{p_1 \in \delta(p_1,1)}{\Rightarrow} (p_1, \lambda) \overset{p_2 \in \delta(p_1,\lambda)}{\Rightarrow} (p_2, \lambda)$$

Este cálculo se corresponde con el camino del diagrama:

$$p_0 \xrightarrow{0} p_0 \xrightarrow{\lambda} p_1 \xrightarrow{1} p_1 \xrightarrow{\lambda} p_2$$

Analicemos cuál es el ***lenguaje aceptado*** *por este autómata. En el estado q_0 se puede leer una secuencia opcional de 0's y luego se pasa a q_1 donde se puede leer una secuencia opcional de*

1's y de aquí se pasa al estado final q_2, donde se puede leer otra secuencia opcional de 2's. Así que el estado q_2 "recuerda" que se leyó una secuencia opcional de 0's seguida de una secuencia opcional de 1's, seguida de una secuencia opcional de 2's. Por tanto, M_{012} acepta cadenas del tipo $0^i1^j2^k$, donde $i, j, k \geq 0$, luego:

$$L(M_{012}) = \{0^i1^j2^k \mid i, j, k \geq 0\}$$

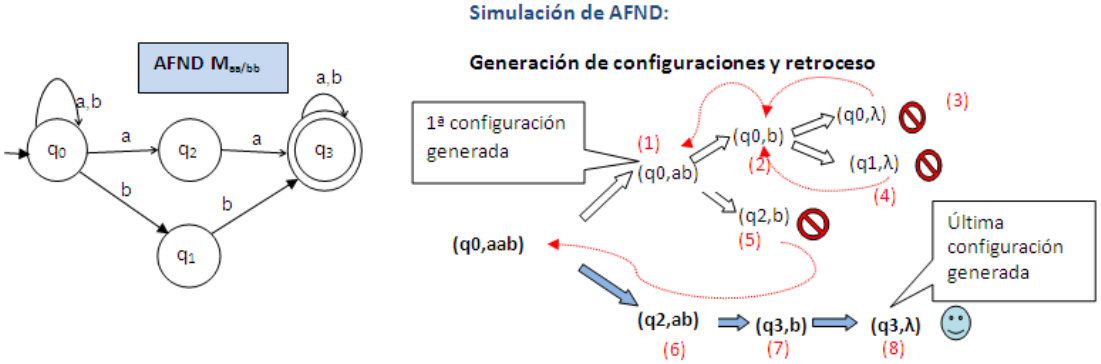
Observamos que $\lambda \in L(M_{012})$ aunque p_0 no es final, porque se acepta por el siguiente cálculo:

$$(p_0, \lambda) \overset{p_1 \in \delta(p_0,\lambda)}{\Rightarrow} (p_1, \lambda) \overset{p_2 \in \delta(p_1,\lambda)}{\Rightarrow} (p_2, \lambda) \quad [p_2 \in F, \text{ acepta } \lambda]$$

2.3. Simulación de un AFND

Hemos comentado que un AFD puede ser simulado por un algoritmo, como el que se muestra en la Figura 4, en $O(n)$, donde n es el número de símbolos de la cadena de entrada. Sin embargo, aunque el tiempo de ejecución teórico de un AFND sea de complejidad $O(n)$, para simularlo mediante un algoritmo que será implementado en una máquina real (un procesador o número constante de procesadores), que no puede implementar un cálculo paralelo con un número no acotado de configuraciones en cada paso, es necesario explorar las distintas ramas del árbol de cálculo paralelo. Esto supone tener que desarrollar un **algoritmo con backtracking**, que recorre el árbol de cálculo en profundidad, eligiendo una de las opciones de la regla de transición en cada paso y si el camino no lleva a aceptar la cadena deshace el cambio de estado y retrocede hasta la configuración anterior para elegir otra opción. Así hasta llegar a un estado final y aceptar la cadena o rechazar por agotar todas las posibilidades. El número de configuraciones que se generan en esta simulación puede ser en el peor de los casos de orden $0(2^n)$, donde n es la longitud de la cadena de entrada, lo cual supone un **tiempo exponencial de simulación**. Esto es debido a que si en cada paso se tiene que elegir, por ejemplo, entre dos opciones distintas por una regla no determinista entonces el número de configuraciones que se pueden generar en el peor caso es de $O(2^n)$: dos en el primer paso, dos en el segundo y así hasta leer el símbolo n-ésimo.

Ejemplo 20 *En el siguiente diagrama mostramos el árbol de cálculo paralelo y las diferentes configuraciones que se generarían (8, en lugar de las 3 necesarias) si se recorre el árbol eligiendo la opción más desfavorable de la regla de transición en cada paso. Las fechas punteadas indican un retroceso a una configuración anterior, donde hay que elegir otra opción.*

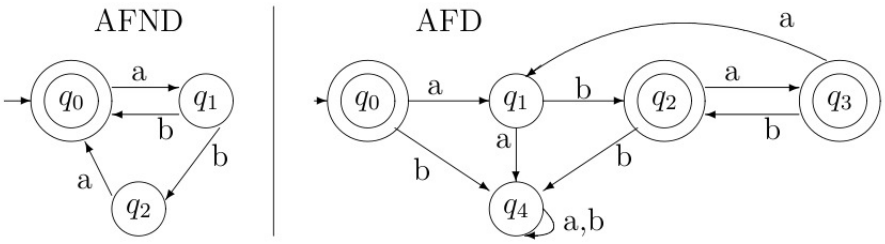


Otra forma de simular la ejecución es por **construcción de subconjuntos de estados**, haciendo que la variable que almacena el estado actual pueda guardar un subconjunto de estados en lugar de un único estado. Por ejemplo, si al comienzo tenemos $estadoActual = q_0$ y el símbolo leído es a y se tiene que actualizar el estado por la transición $\delta(q_0, a) = \{q_0, q_1\}$ entonces se hace $estadoActual := \{q_0, q_1\}$. Esto es como tener activos los estados q_0 y q_1 simultáneamente. Esta es la idea del algoritmo de transformación de un AFND en un AFD que veremos en la sección 4.

2.4. Diseño con AFNDs

El uso de autómatas no deterministas **facilita el diseño** de un autómata para aceptar cadenas de lenguajes complejos.

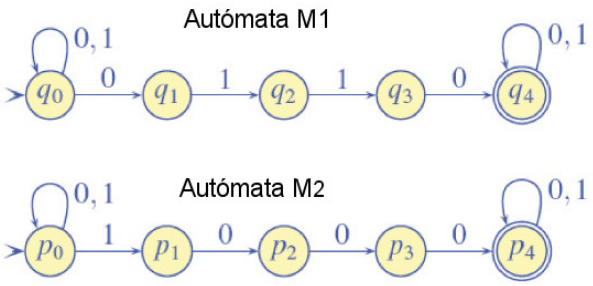
Ejemplo 21 *Si queremos obtener un AF que acepte el lenguaje clausura $\{ab, aba\}^*$, formado por cadenas que resultan de concatenar cero o más veces las subcadenas ab ó aba, es más sencillo obtener un AFND que obtener un AFD directamente.*



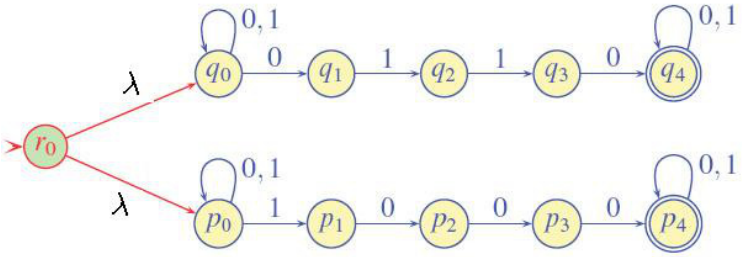
Las λ-transiciones en un AFND son especialmente útiles para combinar de manera sencilla autómatas más simples. En el tema siguiente veremos que son necesarias para convertir automáticamente una expresión regular en un autómata finito.

Ejemplo 22 *Supongamos que tenemos que obtener un AF que sirva para comprobar si una cadena de 0's/1's contiene la subcadena* 0110 *o la subcadena* 1000*. Una manera de resolver el problema es teniendo en cuenta que hay que aceptar cadenas de un lenguaje que consiste en la unión de dos sublenguajes: L_1 formado por las cadenas de $\{0,1\}^*$ que contienen la subcadena* 0110 *y otro L_2 con las cadenas que contienen la subcadena* 1000*. Aunque podría obtenerse un AFD directamente, es bastante más sencillo obtener autómatas no deterministas M_1 y M_2 para cada sublenguaje y luego combinarlos mediante λ-transiciones.*

En la siguiente figura tenemos el autómata M_1 que acepta cadenas con subcadena 0110 *(lenguaje L_1) y el autómata M_2 que acepta cadenas con subcadena* 1000 *(lenguaje L_2). La corrección del diseño de estos autómatas es fácil de comprobar porque los autómatas son sencillos, es cierto que $L(M_1) = L_1$ y $L(M_2) = L_2$.*



Ahora combinamos los dos autómatas para aceptar cadenas de $L_1 \cup L_2$ y obtenemos un autómata M con un nuevo estado inicial r_0 e incluimos arcos λ desde ese estado al inicio de cada autómata:



El autómata así construido ***es correcto*** *para aceptar cadenas de $L_1 \cup L_2$ porque una cadena es aceptada por M si y sólo si es aceptada por M_1 o por M_2, ya que tiene que leerse a través de M_1 o bien a través de M_2 y como $L(M_1) = L_1$ y $L(M_2) = L_2$ entonces tenemos que $L(M) = L(M_1) \cup L(M_1) = L_1 \cup L_2$.*

3. Operaciones REGULARES con autómatas finitos

En esta sección mostramos algoritmos que permiten operar con autómatas finitos (de cualquier tipo) y construyen otro autómata finito que acepta la unión, la concatenación o la clausura (operaciones regulares) de los lenguajes aceptados por los autómatas que se combinan.

Las operaciones con AF pueden servir como **herramienta de diseño** de autómatas complejos que se construyen combinando autómatas más simples y también las vamos a usar para un método que traduce una ER en un AF que acepta las mismas cadenas que describe la expresión regular.

3.1. Algoritmo *unionAF*

DESCRIPCIÓN: combina dos autómatas finitos para obtener otro autómata que acepta la unión de los lenguajes aceptados por los autómatas de entrada.

FUNCIÓN $unionAF\,(M_1, M_2)$.

ENTRADA: dos autómatas finitos $M_1 = (Q_1, V_1, \delta_1, q_{0_1}, F_1)$ y $M_2 = (Q_2, V_2, \delta_2, q_{0_2}, F_2)$.

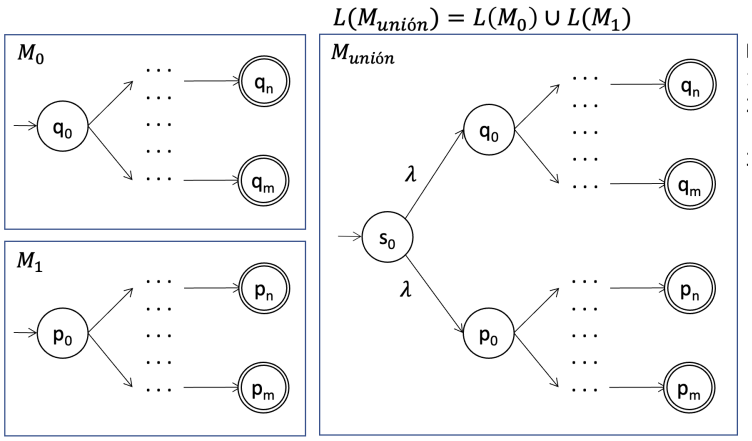


Figura 8: esquema de construcción del autómata unión:

SALIDA: un autómata finito $M_{union} = (Q, V, \delta, q_0, F)$ tal que $L(M_{union}) = L(M_1) \cup L(M_2)$.

1. **Se renombran los estados** de M_2 para que tengan distinto nombre de los de M_1;
2. El **alfabeto** de M_{union} es $V \leftarrow V_1 \cup V_2$;
3. Se considera un estado nuevo q_0 que será el **estado inicial** de M_{union};
4. El **conjunto de estados** de M_{union} es $Q \leftarrow Q_1 \cup Q_2 \cup \{q_0\}$
5. Se obtiene el **diagrama de transición** de M_{union} de la siguiente forma:
   - Se añade el nodo correspondiente al nuevo estado inicial q_0.
   - Se incluyen los estados y transiciones del diagrama de M_1 y los de M_2.
   - Se añade un arco λ desde q_0 al que era inicial de M_1 y otro arco λ desde q_0 al que era inicial de M_2.
6. El **conjunto de estados finales** de M_{union} es $F \leftarrow F_1 \cup F_2$;
7. DEVUELVE (M_{union});

Justificación de que $L(M_{union}) = L(M_1) \cup L(M_2)$

Cualquier camino de q_0 a uno de los estados finales de M_{union} debe pasar forzosamente a través del autómata M_1 y acabar en uno de sus estados finales o pasar por el autómata M_2 y acabar en uno de sus finales. Entonces una cadena w es aceptada por M_{union} (esto es, $w \in L(M_{union})$) si y sólo si es aceptada por M_1 ($w \in L(M_1)$)) o por M_2 ($w \in L(M_1)$)). Luego $L(M_{union}) = L(M_1) \cup L(M_2)$.

3.2. Complementación de un AFD

La operación complementación se aplica a un AFD y obtiene otro AFD. Sea un AFD $M = (Q, V, \delta, q_0, F)$. El **autómata complementario de M** es un AFD, que llamamos $\overline{M}$, y acepta todas aquellas cadenas que no acepta M, es decir:

$$L(\overline{M}) = \overline{L(M)} = V^* - L(M)$$

El autómata $\overline{M}$ se obtiene fácilmente a partir de M aplicando el algoritmo de la figura 9.

FUNCIÓN *complementaAFD*(M)

ENTRADA: un AFD $M = (Q, V, \delta, q_0, F)$.
DEVUELVE: el AFD complementario de M.

1. $M :=$ *completaAFD*(M); /*Primero se completa el autómata de entrada */

   /* Luego se complementa el conjunto de estados finales en M: los estados no finales (incluido el estado de error si se ha añadido en paso 1) pasan a ser el conjunto de estados finales */

2. $F := Q - F$;

   /* El AFD M así modificado es el autómata complementario */

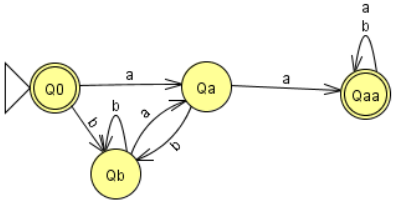
3. DEVUELVE (M)

Figura 9: **Algoritmo *complementaAFD***

Ejemplo 23 *Sea L_{naav} el lenguaje formado por las cadenas de $\{a, b\}^*$ que no son vacías ni contienen la subcadena aa. Queremos obtener un AFD que acepta este tipo de cadenas. Podemos diseñar directamente un AFD para este lenguaje o, si resulta más sencillo pensar en positivo, consideramos el lenguaje L_{aav}, que contiene la cadena vacía o cadenas que contienen la subcadena aa, que por comprensión se define como:*

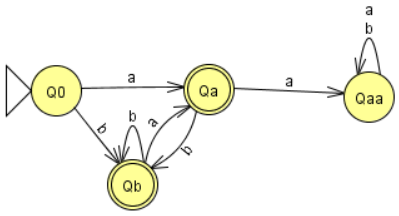
$$L_{aav} = \{xaay \mid x, y \in \{a, b\}^*\} \cup \{\lambda\} = \{w \in \{a, b\}^* \mid w = \lambda \vee w = xaay, \text{ donde } x, y \in \{a, b\}^*\}$$

Un AFD que acepta L_{aav} es el autómata M_{aav} que se muestra a continuación:



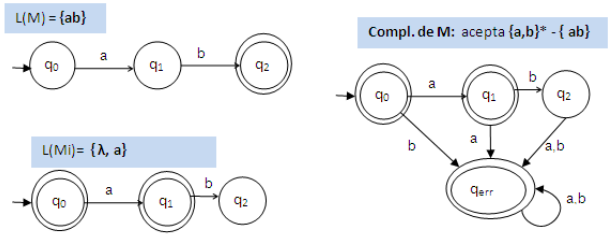
Ahora tenemos en cuenta que $L_{naav} = \overline{L_{aav}} = \{a, b\}^* - L_{naav}$, por lo que podemos obtener un AFD que acepte L_{naav} complementando el AFD M_{aav}. Como el autómata M_{aav} ya está

completo entonces lo único que hay que hacer es complementar los estados finales y obtenemos el autómata complementario $\overline{M_{aav}}$*:*



$\overline{M_{aav}}$ *acepta* L_{naav} *porque* $L(\overline{M_{aav}}) = \overline{L(M_{aav})} = \overline{L_{aav}} = L_{naav}$

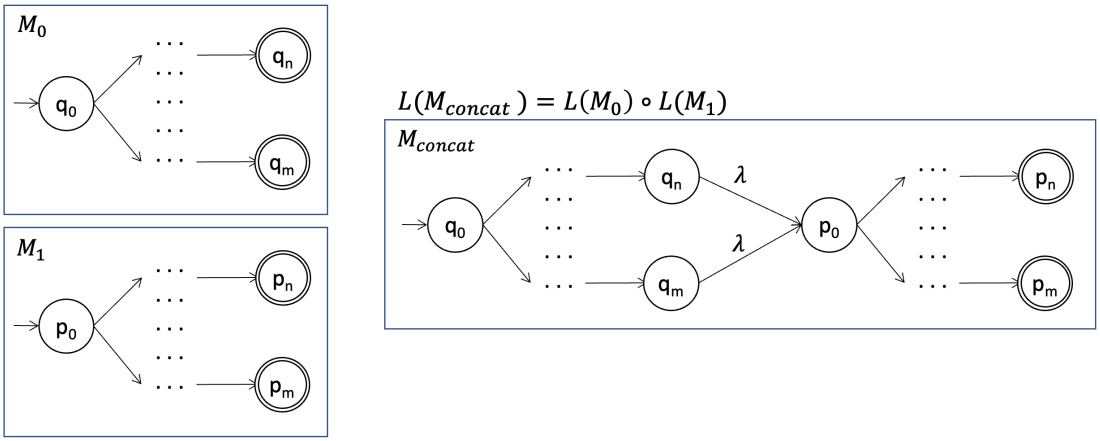
¿Qué pasa si se complementan los estados finales de un AFD incompleto? Supongamos que tenemos un AFD incompleto M y obtenemos otro Mi que sólo se diferencia en que los finales de Mi son los no finales de M. Entonces Mi **no es** el autómata complementario de M, porque no se cumple que $L(Mi) = \overline{L(M)}$, como se puede apreciar en el siguiente ejemplo:



La forma correcta de complementar M es según el algoritmo: primero se completa M y luego se complementan los estados finales. De esa forma si M acepta sólo la cadena ab entonces el autómata complementario acepta todas las cadenas de a's/b's excepto ab, como debe ser.

3.3. Algoritmo *concatenacionAF*

ESQUEMA DE CONSTRUCCIÓN DEL AUTÓMATA CONCATENADO:



Justificación de que $L(M_{concat}) = L(M_1) \circ L(M_2)$

Cualquier camino de q_0 a uno de los estados finales de M_{concat} debe pasar forzosamente a través del autómata M_1 y luego, desde un antiguo final de M_1 seguir por un camino de M_2 que acabe en estado final. Entonces una cadena w es aceptada por M_{concat} si y sólo si es de la forma

DESCRIPCIÓN: combina dos autómatas finitos para obtener otro autómata que acepta la concatenación de los lenguajes aceptados por los autómatas de entrada.
FUNCIÓN *concatenacionAF* (M_1, M_2)
ENTRADA: dos autómatas finitos $M_1 = (Q_1, V_1, \delta_1, q_{0_1}, F_1)$ y $M_2 = (Q_2, V_2, \delta_2, q_{0_2}, F_2)$
SALIDA: un autómata finito $M_{concat} = (Q, V, \delta, q_0, F)$ tal que $L(M_{concat}) = L(M_1) \circ L(M_2)$

1. **Se renombran los estados** de M_2 para que tengan distinto nombre de los de M_1;
2. El **alfabeto** de M_{concat} es $V \leftarrow V_1 \cup V_2$;
3. El **estado inicial** es $q_0 \leftarrow q_{0_1}$ //estado inicial de M_1;
4. El **conjunto de estados** de M_{concat} es $Q \leftarrow Q_1 \cup Q_2$;
5. Se obtiene el **diagrama de transición** de M_{concat} de la siguiente forma:
   - Se incluyen los estados y transiciones del diagrama de M_1 y los de M_2.
   - Se añade un arco λ desde los estados que eran finales en M_1 hasta el que era inicial de M_2.
6. El **conjunto de estados finales** de M_{concat} es $F \leftarrow F_2$;
   //sólo quedan finales los del segundo autómata.
7. DEVUELVE (M_{concat})

$w = z_1 z_2$, de tal manera que z_1 es aceptada por M_1 y z_2 es aceptada por M_2. Eso supone que $L(M_{concat}) = L(M_1) \circ L(M_2)$.

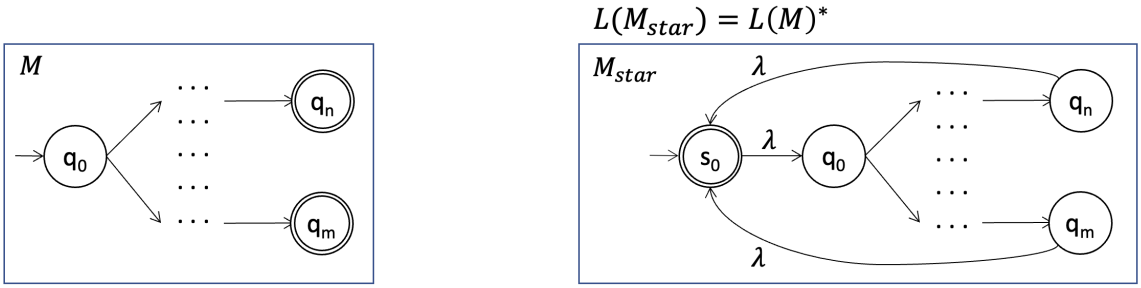
3.4. Algoritmo *clausuraAF*

DESCRIPCIÓN: parte de un AF y obtiene otro que acepta la clausura del lenguaje aceptado por el autómata de entrada.

FUNCIÓN $clausuraAF\,(M_1)$
ENTRADA: un autómata finito $M_1 = (Q_1, V_1, \delta_1, q_{0_1}, F_1)$
SALIDA: un autómata finito $M_{star} = (Q, V, \delta, q_0, F)$ tal que $L(M_{star}) = L(M_1)^*$

1. El **alfabeto** de M_{star} es $V \leftarrow V_1$.
2. Se considera un estado nuevo q_0 que será el **estado inicial** de M_{star}.
3. El **conjunto de estados** de M_{star} es $Q \leftarrow Q_1 \cup \{q_0\}$.
4. Se obtiene el **diagrama de transición** de M_{star} de la siguiente forma:
   - Se incluyen los estados y transiciones del diagrama de M_1.
   - Se añade un arco λ desde el estado q_0 al estado q_{0_1}.
   - Se añade un arco λ desde los estados que eran finales en M_1 hasta el estado inicial.
5. El **conjunto de estados finales** de M_{star} es $F \leftarrow \{q_0\}$
6. DEVUELVE (M_{star});

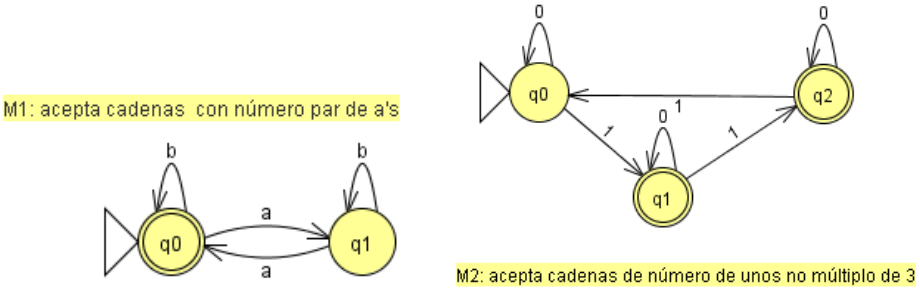
ESQUEMA DE CONSTRUCCIÓN DEL AUTÓMATA CLAUSURA:



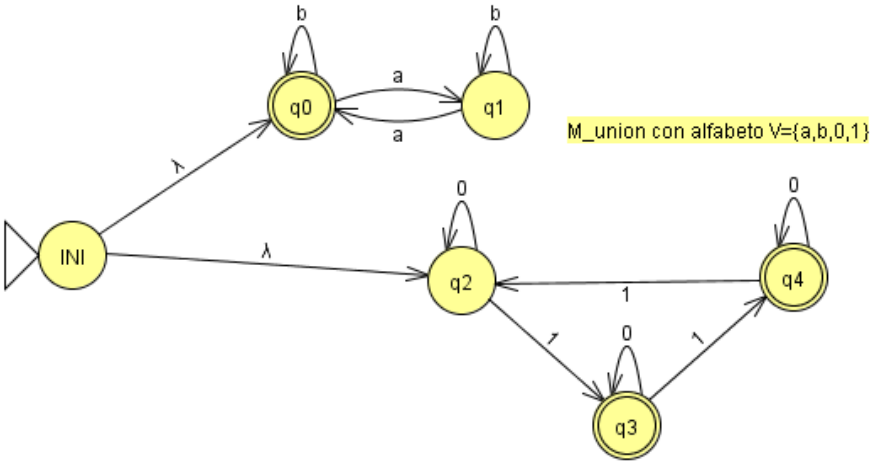
Justificación de que $L(M_{star}) = L(M_1)^*$

Al definir al estado inicial como final, el autómata M_{star} acepta λ. También acepta una cadena w que sea aceptada por M_1 y por las λ-transiciones incluidas desde los finales al inicial, el autómata M_{star} acepta concatenaciones de cadenas aceptadas por M_1. Eso supone que $L(M_{star}) = L(M_1)^*$.

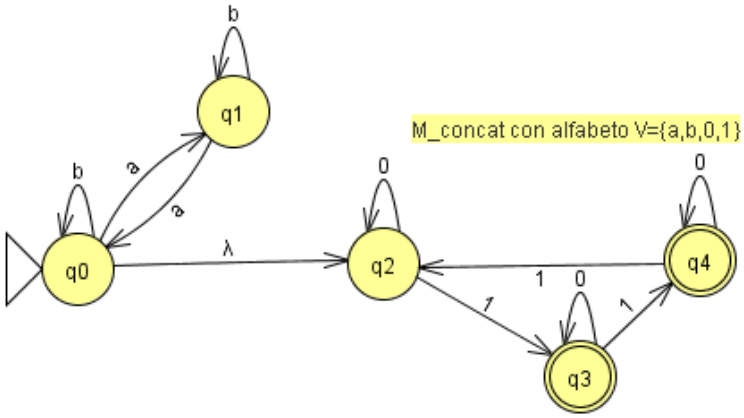
Ejemplo 24 *Sean los autómatas*



Estos autómatas y los siguientes de este ejemplo se encuentran editados en la carpeta ***tema3-ER-jflap*** *de recursos del Aula Virtual. El* **autómata unión** *acepta el lenguaje* $L(M_1) \cup L(M_2)$*, es decir, cadenas de alfabeto* $\{a, b, 0, 1\}$ *que tienen un número par de a's (y sólo símbolos a/b) o que tienen un número de unos que no es múltiplo de 3 (y sólo símbolos 0/1):*

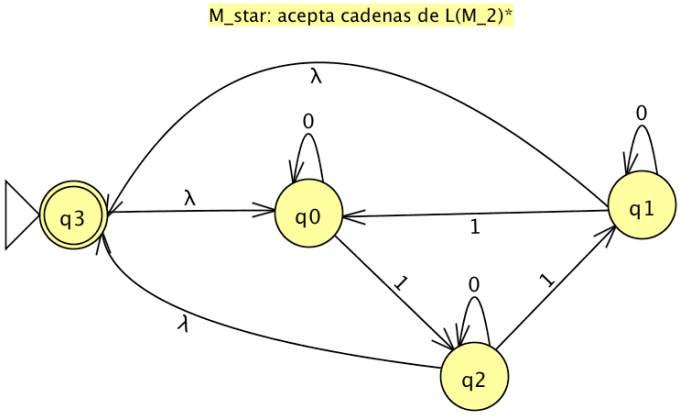


El **autómata concatenado** *acepta la concatenación* $L(M_1) \circ L(M_2)$*, es decir, cadenas de alfabeto* $\{a, b, 0, 1\}$ *de la forma* xz *tal que* x *tiene un número par de a's (y sólo símbolos a/b) y* z *tiene un número de unos que no es múltiplo de 3 (y sólo símbolos 0/1):*



El **autómata clausura** *de* M_2 *acepta* $L(M_2)^*$*, es decir, cadenas de alfabeto* $\{0, 1\}$ *formadas por concatenación repetida de cadenas de 0's/1's que tienen un número de unos que no es múltiplo de 3. Como* λ *pertenece a la clausura de cualquier lenguaje entonces debe ser aceptada por este*

autómata y de hecho la acepta porque el estado inicial es un estado final.

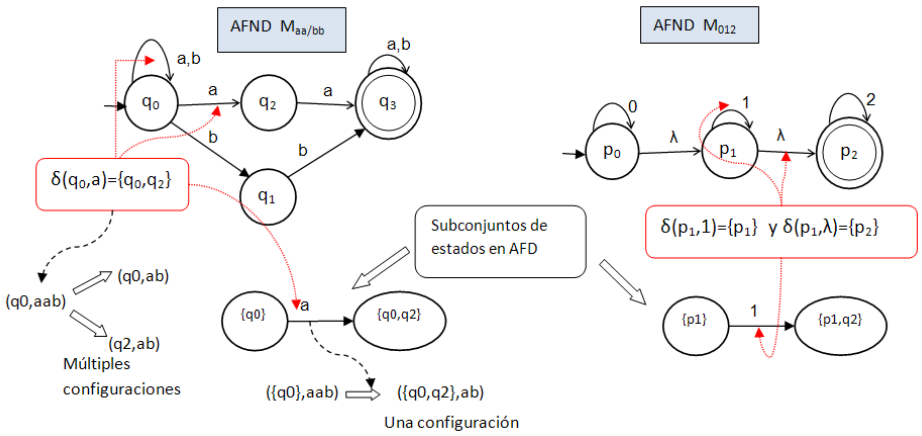


4. Transformación de AFND a AFD

En esta sección mostramos un algoritmo para trasformar un AFND en un AFD equivalente. Este método tiene *interés desde el punto de vista práctico*, porque a efectos de simular el funcionamiento de un autómata finito mediante un algoritmo conviene que el autómata sea determinista por cuestiones de eficiencia.

La **idea** de la transformación de AFND a AFD se basa en un método de **construcción de subconjuntos de estados**. Se trata de obtener un autómata determinista M_D que simule el modo de funcionamiento de un autómata no determinista M, haciendo que **cada estado de M_D sea un subconjunto de estados de M**. Este subconjunto representa los estados de las distintas configuraciones del autómata M en cierto instante de un cálculo paralelo y se hace que las reglas de transición de M_D simulen las diferentes opciones de una transición no determinista. De esa forma lo que sería un árbol de cálculo paralelo con una cadena de entrada se convierte en un único cálculo o traza de ejecución.

Ejemplo 25 *En el autómata $M_{aa/bb}$ desde el estado q_0 leyendo el símbolo a se puede pasar al estado q_0 o al estado q_2, por la regla no determinista $\delta(q_0, a) = \{q_0, q_2\}$, lo cual supone que a partir de una configuración como (q_0, aab) se puedan alcanzar dos configuraciones distintas en un paso de cálculo. Para obtener un AFD equivalente se considera que $\{q_0, q_2\}$ es un único estado en el AFD, por lo que se tiene la regla determinista $\delta_D(\{q_0\}, a) = \{q_0, q_2\}$, que al aplicarla no genera múltiples configuraciones, como se puede apreciar en la siguiente figura:*



Por otra parte, el autómata M_{012} tiene λ-transiciones. Eso supone que cuando parte de un estado como p_1 y lee el símbolo 1, el autómata cambia al estado p_1, pero puede realizar una transición adicional al estado p_2 antes de leer el siguiente símbolo. Por eso se considera que el subconjunto $\{p_1, p_2\}$ es un único estado en el AFD que se obtiene por construcción de subconjuntos.

Antes de ver el algoritmo de transformación de AFND a AFD definimos una función relacionada con las λ-transiciones que será necesario aplicar en el algoritmo.

Definición 11 *La función de **lambda-clausura de un estado**, denotada $LC(q)$, devuelve el conjunto de estados a los que se puede llegar desde q pasando sólo por arcos λ en el diagrama de transición de un AFND. Formalmente,*

$$\boxed{LC(q) = \{q' \mid (q, x) \Rightarrow^* (q', x),\ \textit{donde } x \in V^*\}}$$

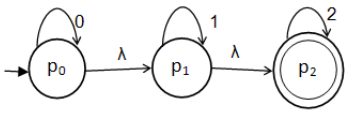
La condición $(q, x) \Rightarrow^* (q', x)$ implica que existe un cálculo donde sólo se aplican λ-transiciones, porque la cadena x no cambia. Se tiene que $\boxed{q \in LC(q)}$ para todo estado q.

*La **lambda-clausura de un conjunto de estados** $S = \{q_{i_1}, q_{i_2}, \ldots, q_{i_k}\}$ devuelve la unión de la lambda-clausura de todos los estados del conjunto:*

$$LC(S) = \bigcup_{q \in S} LC(q) = LC(q_{i_1}) \cup LC(q_{i_2}) \cup \cdots \cup LC(q_{i_k})$$

Se tiene que $\boxed{LC(\varnothing) = \varnothing}$.

Ejemplo 26 *Sea de nuevo el autómata*



Obtenemos: $LC(p_0) = \{p_0, p_1, p_2\}$, $LC(p_1) = \{p_1, p_2\}$, $LC(p_2) = \{p_2\}$

Una vez calculada la lambda-clausura de cada estado se puede calcular la lambda-clausura de cualquier subconjunto de estados del autómata. Por ejemplo:

$$LC(\{p_1, p_2\}) = LC(p_1) \cup LC(p_2) = \{p_1, p_2\} \cup \{p_2\} = \{p_1, p_2\}$$

En la Figura 10 tenemos un algoritmo de transformación de AFND en AFD. Se trata de obtener un autómata determinista M_D que simule el modo de funcionamiento de un AFND de entrada M (es equivalente), siguiendo la idea del método de construcción de subconjuntos. Se va construyendo el **conjunto de estados** Q_D y la **función de transición** δ_D del autómata determinista **de forma incremental**, añadiendo estados-subconjuntos y reglas de transición conforme van apareciendo nuevos estados al calcular las transiciones, en lugar de considerar inicialmente todos los subconjuntos posibles de Q y calcular transiciones, para evitar así que se generen estados inaccesibles.

Al principio sólo se incluye en Q_D el **estado inicial** q_{0D}, que será la lambda-clausura del estado inicial del AFND de entrada: $q_{0_D} := LC(q_0)$, que es el conjunto de estados a los que M puede cambiar antes de leer el primer símbolo de una cadena, como si M pudiera empezar a ejecutarse desde cualquiera de esos estados.
Si el AFND no tiene λ-transiciones entonces $q_{0_D} = LC(q_0) = \{q_0\}$.

Las **reglas de transición** de M_D se definen de tal manera que desde un estado-subconjunto $\{q_{i_1}, q_{i_2}, \ldots, q_{i_k}\}$ de M_D leyendo un símbolo a_j se puedan alcanzar todos los estados a los que se llega en el diagrama de M leyendo a_j desde todos los estados de $\{q_{i_1}, q_{i_2}, \ldots, q_{i_k}\}$ y además a los que se llega después de leer a_j pasando por arcos λ. Por eso hay que usar la función de lambda-clausura para calcular la nueva función de transición δ_D del autómata determinista. Por tanto, si tenemos el subconjunto $\{q_{i_1}, q_{i_2}, \ldots, q_{i_k}\}$ en el nuevo conjunto de estados Q_D y el símbolo a_j del alfabeto, se hace que:

$$\delta_D(\{q_{i_1}, q_{i_2}, \ldots, q_{i_k}\}, a_j) = LC(\ \delta(q_{i_1}, a_j) \cup \delta(q_{i_2}, a_j) \cup \cdots \cup \delta(q_{i_k}, a_j)\ )$$

y el estado dado por $\delta_D(\{q_{i_1}, q_{i_2}, \ldots, q_{i_k}\}, a_j)$ se añade a Q_D si no está ya incluido en Q_D.

El conjunto F_D de **estados finales** de M_D está formado por todos los subconjuntos S de Q_D que contienen un estado final del autómata de entrada M. Por tanto:

$$F_D = \{S \in Q_D \mid (S \cap F) \neq \varnothing\}$$

Orden de complejidad del algoritmo. El número total de subconjuntos del conjunto Q de estados de un AFND viene dado por el cardinal del conjunto $\mathcal{P}(Q)$ de las partes de Q, que es $|\mathcal{P}(Q)| = 2^n$, siendo $n = |Q|$. Todos estos subconjuntos no tienen por qué generase en el algoritmo. Sin embargo, hay casos desfavorables de autómatas no deterministas para los cuales el conjunto de estados generado al obtener el AFD tendría un total de 2^n estados. Por tanto el orden de complejidad del algoritmo de transformación de AFND en AFD en el peor de los casos es **exponencial**, $O(2^n)$, aunque en la práctica se comporta de forma eficiente, porque se obtienen pocos estados al construir Q_D.

FUNCIÓN $AFNDtoAFD(M)$

ENTRADA: un AFND $M = (Q, V, \delta, q_0, F)$

DEVUELVE: un AFD $M_D = (Q_D, V, \delta_D, q_{0_D}, F_D)$ tal que $L(M_D) = L(M)$

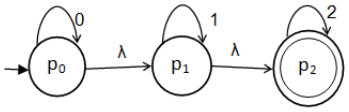
1. $q_{0_D} := LC(q_0)$; // estado inicial de M_D es la lambda-clausura de inicial de M
2. $Q_D := \{q_{0_D}\}$; //al principio sólo el estado inicial en Q_D
3. MIENTRAS haya estados no marcados en Q_D

   /* Un estado S estará marcado si ya se ha calculado $\delta_D(S, a_j)$ para todo $a_j \in V$ */
4. SELECCIONA un estado $S = \{q_{i_1}, q_{i_2}, \ldots, q_{i_k}\}$ no marcado de Q_D;
5. PARA TODO símbolo a_j del alfabeto V
6. $\delta_D(\{q_{i_1}, q_{i_2}, \ldots, q_{i_k}\}, a_j) := LC(\ \delta(q_{i_1}, a_j) \cup \delta(q_{i_2}, a_j) \cup \cdots \cup \delta(q_{i_k}, a_j)\ )$;
7. $Q_D := Q_D \cup \{\ \delta_D(\{q_{i_1}, q_{i_2}, \ldots, q_{i_k}\}, a_j)\ \}$; //añade estado si es nuevo
8. FIN-PARA-TODO
9. MARCA en Q_D el estado $\{q_{i_1}, q_{i_2}, \ldots, q_{i_k}\}$; //ya se han calculado sus transiciones
10. FIN-MIENTRAS
11. $F_D := \{S \in Q_D \mid (S \cap F) \neq \varnothing\}$; //obtiene los estados finales de M_D
12. DEVUELVE (M_D);

// ORDEN DE COMPLEJIDAD: $O(2^n)$ donde $n = |Q|$ (peor caso poco frecuente).

Figura 10: **Algoritmo *AFNDtoAFD* de transformación de AFND a AFD**

Ejemplo 27 *Obtenemos el AFD equivalente a M_{012} que acepta cadenas del tipo $0^i1^j2^k, i, j, k \geq 0$, aplicando el algoritmo AFNDtoAFD paso a paso:*



Primero obtenemos la lambda-clausura de cada estado, para calcular después la lambda-clausura de subconjuntos de estados que se vayan obteniendo:

$$LC(p_0) = \{p_0, p_1, p_2\}, \quad LC(p_1) = \{p_1, p_2\}, \quad LC(p_2) = \{p_2\}$$

▷ *Empezamos incluyendo en Q_D sólo el* ***estado inicial*** *nuevo, que será el conjunto dado por $LC(\{p_0\}) = \{p_0, p_1, p_2\}$. Por tanto, $Q_D = \{\{p_0, p_1, p_2\}\}$.*
Comenzamos a calcular las ***reglas de transición*** *de δ_D, que vamos incluyendo en una tabla de transición:*

δ_D	0	1	2
$\rightarrow \{p_0, p_1, p_2\}$			

▷ *Calculamos δ_D para estado $\{p_0, p_1, p_2\}$ y símbolo 0:*

$\delta_D(\{p_0, p_1, p_2\}, 0) = LC(\delta(p_0, 0) \cup \delta(p_1, 0) \cup \delta(p_2, 0)) = LC(\{p_0\} \cup \varnothing \cup \varnothing) = LC(\{p_0\}) = \{p_0, p_1, p_2\}$

Se obtiene $\{p_0, p_1, p_2\}$ que ya lo tenemos en Q_D, por tanto no se añade a Q_D.

▷ *Para estado $\{p_0, p_1, p_2\}$ y símbolo 1:*

$\delta_D(\{p_0, p_1, p_2\}, 1) = LC(\delta(p_0, 1) \cup \delta(p_1, 1) \cup \delta(p_2, 1)) = LC(\varnothing \cup \{p_1\} \cup \varnothing) = LC(\{p_1\}) = \{p_1, p_2\}$

Hay que añadir el nuevo estado $\{p_1, p_2\}$ a Q_D, que queda $Q_D = \{\{p_0, p_1, p_2\}, \{p_1, p_2\}\}$.

▷ *Para estado $\{p_0, p_1, p_2\}$ y símbolo 2:*

$$\delta_D(\{p_0, p_1, p_2\}, 2) = LC(\varnothing \cup \varnothing \cup \{p_2\}) = LC(\{p_2\}) = \{p_2\}$$

Se añade $\{p_2\}$ y queda $Q_D = \{\{p_0, p_1, p_2\}, \{p_1, p_2\}, \{p_2\}\}$. La tabla de transición hasta este momento es:

δ_D	0	1	2
$\rightarrow \{p_0, p_1, p_2\}$	$\{p_0, p_1, p_2\}$	$\{p_1, p_2\}$	$\{p_2\}$
$\{p_1, p_2\}$			
$\{p_2\}$			

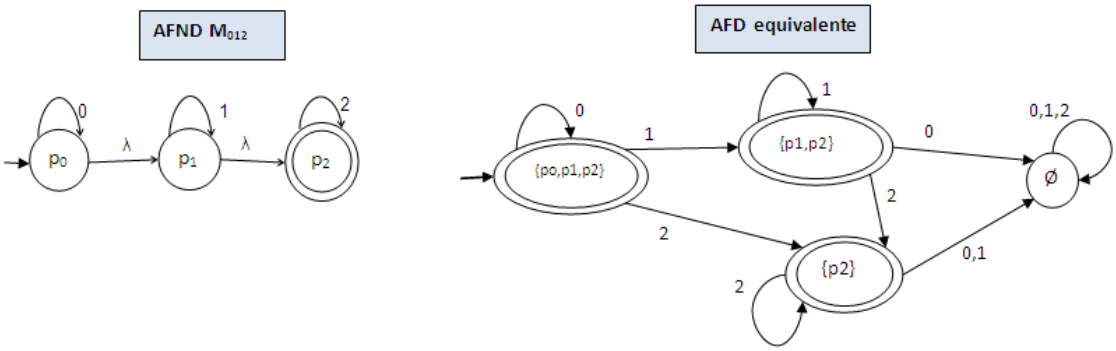
▷ *Calculamos δ_D para los nuevos estados obtenidos con todos los símbolos:*

$\delta_D(\{p_1, p_2\}, 0) = LC(\delta(p_1, 0) \cup \delta(p_2, 0)) = LC(\varnothing) = \varnothing$
$\delta_D(\{p_1, p_2\}, 1) = LC(\delta(p_1, 1) \cup \delta(p_2, 1)) = LC(\{p_1\} \cup \varnothing) = LC(\{p_1\}) = \{p_1, p_2\}$
$\delta_D(\{p_1, p_2\}, 2) = LC(\delta(p_1, 2) \cup \delta(p_2, 2)) = LC(\varnothing \cup \{p_2\}) = LC(\{p_2\}) = \{p_2\}$
$---$
$\delta_D(\{p_2\}, 0) = LC(\delta(p_2, 0)) = \varnothing$
$\delta_D(\{p_2\}, 1) = LC(\delta(p_2, 1)) = \varnothing$
$\delta_D(\{p_2\}, 2) = LC(\delta(p_2, 2)) = LC(\{p_2\}) = \{p_2\}$

Sólo ha aparecido nuevo el conjunto vacío $\varnothing$, que como no contiene estados será $\delta_D(\varnothing, 0) = \delta_D(\varnothing, 1) = \delta_D(\varnothing, 2) = \varnothing$. Este estado vacío actúa como estado de error. Al final queda $Q_D = \{\{p_0, p_1, p_2\}, \{p_1, p_2\}, \{p_2\}, \varnothing\}$ y $F_D = \{\{p_0, p_1, p_2\}, \{p_1, p_2\}, \{p_2\}\}$, porque todos los estados-subconjunto en F_D contienen el estado final p_2 del AFND M_{012}. Ya podemos terminar de rellenar la tabla de transición:

δ_D	0	1	2
$\# \to \{p_0, p_1, p_2\}$	$\{p_0, p_1, p_2\}$	$\{p_1, p_2\}$	$\{p_2\}$
$\#\{p_1, p_2\}$	$\varnothing$	$\{p_1, p_2\}$	$\{p_2\}$
$\#\{p_2\}$	$\varnothing$	$\varnothing$	$\{p_2\}$
$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

Y esta tabla se corresponde con el ***diagrama de transición*** *de la derecha:*



En un ejemplo anterior se comprobó que la cadena λ es aceptada por M_{012} y comprobamos ahora que λ también es aceptada por el AFD porque el estado inicial $\{p_0, p_1, p_2\}$ es un estado final en el AFD.

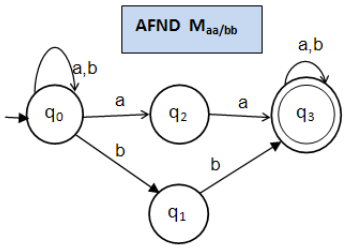
Otra cadena aceptada por ambos autómatas es 01, como demuestran los siguientes cálculos que acaban en configuración de aceptación:

$$(p_0, 01) \overset{p_0 \in \delta(p_0,0)}{\Rightarrow} (p_0, 1) \overset{p_1 \in \delta(p_0,\lambda)}{\Rightarrow} (p_1, 1) \overset{p_1 \in \delta(p_1,1)}{\Rightarrow} (p_1, \lambda) \overset{p_2 \in \delta(p_1,\lambda)}{\Rightarrow} (p_2, \lambda) \quad \textit{[en } M_{012}\textit{]}$$

$$(\{p_0, p_1, p_2\}, 01) \overset{\delta(\{p_0,p_1,p_2\},0)=\{p_0,p_1,p_2\}}{\Rightarrow} (\{p_0, p_1, p_2\}, 1) \overset{\delta(\{p_0,p_1,p_2\},1)=\{p_1,p_2\}}{\Rightarrow} (\{p_1, p_2\}, \lambda) \quad \textit{[en } M_D\textit{]}$$

Como vemos el cálculo es más corto en el AFD (2 pasos) que en AFND (4 pasos), por lo que el AFD es más eficiente.

Ejemplo 28 *En este ejemplo vamos transformar el AFND $M_{aa/bb}$, que no tiene λ-transiciones, en un AFD. Al no tener λ-transiciones se puede simplificar el proceso porque $LC(E) = E$ para todo subconjunto de estados E y es como si no se aplicara la función de lambda-clausura.*



▷ *Empezamos a construir el AFD equivalente incluyendo inicialmente en* Q_D *el estado inicial* $\{q_0\}$ *(el estado inicial del AFD es simplemente el subconjunto que contiene al inicial del AFND porque* $LC(q_0) = \{q_0\}$*).*

▷ *Calculamos* δ_D *para el estado* $\{q_0\}$ *con los símbolos* a *y* b*:*

$$\delta_D(\{q_0\}, a) = LC(\delta(q_0, a)) = LC(\{q_0, q_2\}) = \{q_0, q_2\}$$
$$\delta_D(\{q_0\}, b) = LC(\delta(q_0, b)) = LC(\{q_0, q_1\}) = \{q_0, q_1\}$$

Los estados nuevos que se han generado se añaden a la tabla.

δ_D	a	b
$\rightarrow \{q_0\}$	$\{q_0, q_2\}$	$\{q_0, q_1\}$
$\{q_0, q_2\}$		
$\{q_0, q_1\}$		

▷ *Se calculan las reglas de transición para los nuevos estados:*

$\delta_D(\{q_0, q_2\}, a) = LC(\delta(q_0, a) \cup \delta(q_2, a)) = LC(\{q_0, q_2\} \cup \{q_3\}) = LC(\{q_0, q_2, q_3\}) = \{q_0, q_2, q_3\}$
$\delta_D(\{q_0, q_2\}, b) = LC(\delta(q_0, b) \cup \delta(q_2, b)) = LC(\{q_0, q_1\} \cup \varnothing) = \{q_0, q_1\}$
$- - -$
$\delta_D(\{q_0, q_1\}, a) = LC(\delta(q_0, a) \cup \delta(q_1, a)) = LC(\{q_0, q_2\} \cup \varnothing) = LC(\{q_0, q_2\}) = \{q_0, q_2\}$
$\delta_D(\{q_0, q_1\}, b) = LC(\delta(q_0, b) \cup \delta(q_1, b)) = LC(\{q_0, q_1\} \cup \{q_3\}) = LC(\{q_0, q_1, q_3\}) = \{q_0, q_1, q_3\}$

Se completa la tabla con estas transiciones y se añaden los nuevos estados obtenidos:

δ_D	a	b
$\rightarrow \{q_0\}$	$\{q_0, q_2\}$	$\{q_0, q_1\}$
$\{q_0, q_2\}$	$\{q_0, q_2, q_3\}$	$\{q_0, q_1\}$
$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_3\}$
$\# \{q_0, q_2, q_3\}$		
$\# \{q_0, q_1, q_3\}$		

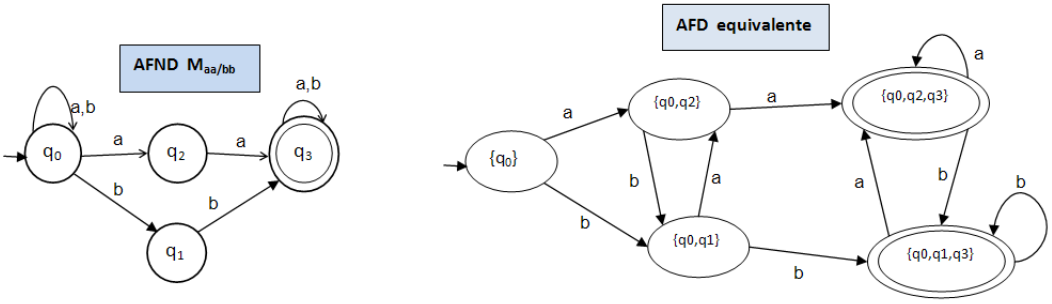
▷ *Se calculan las reglas de transición para los nuevos estados. Lo hacemos directamente, viendo en el diagrama qué estados se pueden alcanzar desde los estados del subconjunto* $\{q_0, q_2, q_3\}$ *leyendo el símbolo* a *y el* b *y lo mismo con* $\{q_0, q_1, q_3\}$*:*

$$\delta_D(\{q_0, q_2, q_3\}, a) = \{q_0, q_2, q_3\}$$
$$\delta_D(\{q_0, q_2, q_3\}, b) = \{q_0, q_1, q_3\}$$
$$- - -$$
$$\delta_D(\{q_0, q_1, q_3\}, a) = \{q_0, q_2, q_3\}$$
$$\delta_D(\{q_0, q_1, q_3\}, b) = \{q_0, q_1, q_3\}$$

Como no hay estados nuevos hemos acabado de calcular la función de transición. El AFD tiene dos estados finales: $\{q_0, q_2, q_3\}$ *y* $\{q_0, q_1, q_3\}$*, porque* q_3 *es final en* $M_{aa/bb}$*. La tabla de transición queda como:*

δ_D	a	b
$\rightarrow \{q_0\}$	$\{q_0, q_2\}$	$\{q_0, q_1\}$
$\{q_0, q_2\}$	$\{q_0, q_2, q_3\}$	$\{q_0, q_1\}$
$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_3\}$
$\# \{q_0, q_2, q_3\}$	$\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_3\}$
$\# \{q_0, q_1, q_3\}$	$\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_3\}$

Y el diagrama de transición es:



El siguiente teorema resume la idea de que los autómatas deterministas y no deterministas son **formalismos equivalentes o igual de potentes**, en el sentido de que aceptan los mismos tipos de lenguajes, que se llaman **lenguajes regulares** y los veremos también en el tema siguiente.

Teorema 2 *Un lenguaje es aceptado por un autómata no determinista si y sólo si es aceptado por un autómata determinista.*

Dem.- La demostración tiene dos partes:

1. ($\Rightarrow$) Hay que probar que si un lenguaje L_r es aceptado por un AFD M_D entonces existe un AFND M_N tal que $L_r = L(M) = L(M_N)$.
   Esta claro que un AFD se puede considerar como un "caso particular" de AFND. Entonces si tenemos un lenguaje L_r que es aceptado por un autómata determinista cualquiera $M_D = (Q_D, V, \delta_D, q_{0D}, F_D)$ podemos obtener fácilmente uno no determinista equivalente $M_N = (Q_N, V, \delta_N, q_{0N}, F_N)$, haciendo que $Q_N = Q_D, q_{0N} = q_{0D}, F_N = F_D$ y simplemente definiendo $\delta_N : Q_N \times V \longrightarrow \mathcal{P}(Q_D)$ de tal forma que:

   $$\delta_N(q, a) = \{\delta_D(q, a)\}, \forall\, q \in Q_N,\ a \in V$$

   Esta transformación para pasar de M_D a M_N sólo afecta a la notación matemática para la función de transición, no al tipo de cadenas que acepta. Por tanto L_r también es aceptado por M_N, como queríamos probar.

2. ($\Leftarrow$) Hay que probar que si un lenguaje L_r es aceptado por un AFND M_N entonces existe un AFD M_D tal que $L_r = L(M_N) = L(M_D)$. Esto es cierto porque para obtener M_D basta aplicar el algoritmo de transformación de AFND a AFD con M_N como entrada (la demostración de la corrección del algoritmo la omitimos), y se cumple la condición $L(M_D) = L(M_N)$. Esta es la parte de la demostración que tiene interés práctico.

Por 1 y 2 se deduce que los autómatas deterministas y no deterministas aceptan los mismos tipos de lenguajes, por tanto son igual de potentes. □

5. Autómatas finitos y expresiones regulares: Teorema de Kleene

Las expresiones regulares y los autómatas finitos están estrechamente relacionados, se consideran **formalismos equivalentes** en el sentido de que ambos tratan con la clase de los lenguajes regulares, más concretamente, en el sentido de que todo lenguaje descrito por una expresión regular puede se aceptado por un autómata finito y viceversa. Sin embargo tienen distinto objetivo, porque una expresión regular es un **formalismo descriptivo**, que permite especificar un lenguaje regular mediante una fórmula algebraica y un autómata finito es un **formalismo operacional** que permite especificar un lenguaje regular mediante una máquina abstracta capaz de distinguir las cadenas del lenguaje (aceptando) de las que no pertenecen al lenguaje (rechazando).

El resultado que muestra la equivalencia o igual potencia expresiva de autómatas finitos y expresiones regulares es un teorema debido a Stephen Kleene:

Teorema de Kleene: *un lenguaje puede ser descrito por una expresión regular si y sólo si puede ser aceptado por un autómata finito.*

El teorema completo se suele dividir en dos partes que afirman:

1. SÍNTESIS: si un lenguaje puede ser descrito por una expresión regular entonces también puede ser aceptado un autómata finito.

2. ANÁLISIS: si un lenguaje puede ser aceptado por autómata finito entonces también puede ser descrito por una expresión regular.

La demostración del teorema de Kleene proporciona algoritmos de traducción entre expresiones regulares y autómatas finitos. El algoritmo de la demostración de la parte de SÍNTESIS es de especial interés práctico y es el que vamos a introducir a continuación.

5.1. De las expresiones regulares a los autómatas finitos

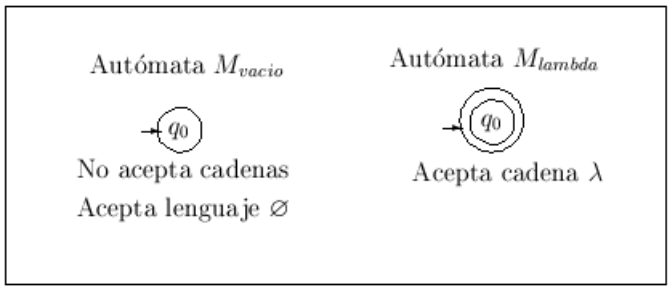
Teorema de Kleene (parte de síntesis)
Si L_r es un lenguaje descrito por una expresión regular R entonces existe un autómata finito M tal que $L_r = L(R) = L(M)$

La demostración de este teorema proporciona un método algorítmico para pasar de una ER a un AF que acepta el mismo lenguaje que describe la expresión regular. Este teorema tiene gran **interés desde el punto de vista práctico**, pues hace posible que las herramientas que permiten al usuario o programador introducir expresiones regulares para describir patrones de cadenas puedan "traducir" la ER en la implementación de un autómata finito cuya tabla de transición se usa en las funciones o métodos que permiten realizar validaciones de formato, búsquedas o sustituciones. Este proceso de traducción, que también se llama *"compilación de una ER"*, se lleva a cabo en un módulo software que se conoce como *regex engine* y este módulo es el que proporciona las funciones para resolver problemas de *regular pattern matching*.

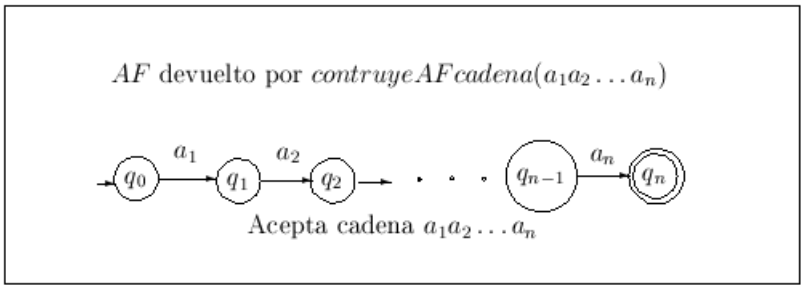
La **idea del algoritmo ERtoAF(R)**, que se muestra más adelante para traducir una ER R en un AF M, se basa en descomponer una ER en subexpresiones, construir autómatas finitos para las subexpresiones y combinar convenientemente estos autómatas hasta conseguir un autómata M para la expresión regular R de partida. El proceso sería análogo al que se sigue para evaluar una expresión aritmética por partes y operar con los resultados parciales para producir el valor final. En este caso, los resultados parciales del proceso de traducción de ER a AF son autómatas finitos para las subexpresiones de la ER y el valor final es el autómata finito para la expresión regular de partida.

Como la sintaxis de una ER la hemos definido de forma recursiva, el algoritmo $ERtoAF(R)$ lo diseñamos recursivo.

El caso base para el que no se producen llamadas recursivas al algoritmo $ERtoAF(R)$ se da cuando R es una **expresión regular sin unión ni clausura**. Entonces R puede ser: $R = \varnothing$, $R = \lambda$ o $R = a_1 a_2 \dots a_n$, donde cada a_i es un símbolo del alfabeto. Los autómatas que se construyen para las dos primeras ER son autómatas constantes:



Para una ER que sólo tiene coincidencia con una cadena, tipo $a_1 a_2 \dots a_n$ se obtiene un autómata que la acepta, como en el siguiente esquema:



El método auxiliar $construyeAFcadena(w)$ se supone que devuelve el autómata (de hecho determinista) que acepta exactamente la cadena w y ninguna más. Es sencillo hacerlo, sólo hay que incluir un estado nuevo por cada símbolo de la cadena w y enlazar los estados mediante arcos etiquetados con los símbolos de la cadena. El estado que se alcanza al leer el último símbolo de la cadena se establece como final.

Para el proceso de traducción de ER a AF se necesita llevar a cabo un **análisis sintáctico (*parsing*)** donde se intenta construir el árbol de análisis para comprobar si la ER es sintácticamente correcta y saber de qué tipo es (cómo se puede dividir en otras más simples), es decir se necesita comprobar:

- si la expresión regular de entrada R es una ER sin operadores de unión ni clausura,
- o si es de la forma $R_1|R_2$ (unión de ER),
- o si es de la forma $R_1 \circ R_2$ (concatenación de ER),
- o si es R_1^* (clausura de ER)

y según el caso se aplica una operación de autómatas u otra. Si la ER no se ajusta a ninguno de esos casos entonces es incorrecta y el algoritmo produce error de sintaxis. Para hacer estas comprobaciones se necesitan técnicas de análisis sintáctico que quedan fuera del alcance de este curso (se verá en la asignatura de Compiladores). Nosotros sólo lo aplicamos de forma manual en los ejemplos teniendo en cuenta la definición recursiva de la sintaxis de ER que habíamos estudiado en el tema anterior.

Algoritmo *ERtoAF*

FUNCIÓN $ERtoAF\,(R)$
ENTRADA: una expresión regular R
SALIDA: un autómata finito M tal que $L(R) = L(M)$

// Casos base

1. SI $R = \varnothing$ ENTONCES DEVUELVE (M_{vacio}); //autómata constante, no acepta cadenas
2. SI $R = \lambda$ ENTONCES DEVUELVE (M_{lambda}); //autómata constante, sólo acepta λ
3. SI R es una cadena de símbolos ENTONCES
   - $M \leftarrow construyeAFcadena(R)$; //obtiene un AF que acepta sólo la cadena en R
   - DEVUELVE (M);

// Casos recursivos

4. SI R es de la forma $R_1 \circ R_2$ ENTONCES
   - $M_1 \leftarrow ERtoAF(R_1)$; //llamada recursiva con subexpresión R_1
   - $M_2 \leftarrow ERtoAF(R_2)$; //llamada recursiva con subexpresión R_2
   - $M \leftarrow concatenacionAF(M_1, M_2)$; //concatenación de autómatas de subexpr.
   - DEVUELVE (M);
5. SI R es de la forma $R_1|R_2$ ENTONCES
   - $M_1 \leftarrow ERtoAF(R_1)$;
   - $M_2 \leftarrow ERtoAF(R_2)$;
   - $M \leftarrow unionAF(M_1, M_2)$; //unión de autómatas para subexpresiones
   - DEVUELVE (M);
6. SI R es de la forma R_1^* ENTONCES
   - $M_1 \leftarrow ERtoAF(R_1)$;
   - $M \leftarrow clausuraAF(M_1)$; //clausura de autómata para subexpresión
   - DEVUELVE (M);
7. EXIT $(syntaxError)$;

// Llegados a este punto se sabe que R es sintácticamente incorrecta y termina con error

//ORDEN DE COMPLEJIDAD: $O(n)$, donde n es el número de caracteres en R

Nota: en algunos libros de la bibliografía se considera que los casos base del algoritmo de traducción de ER a AF coinciden con los casos base de la sintaxis de ER, porque esa es la forma más simple de implementar el proceso de traducción mediante un programa. Por tanto, una expresión regular R del tipo $a_1a_2 \dots a_n$ (en línea 4) se subdivide a su vez en expresiones con un único símbolo a_1, a_2, …, a_n, se construyen autómatas para cada una y se combinan con el método de concatenación de autómatas. Aquí preferimos no subdividir una ER que no contiene uniones ni clausuras por no introducir demasiadas λ-transiciones en el autómata, teniendo en cuenta además que el método se va a aplicar en ejemplos y no se llegará a implementar.

Ejemplo 29 *Vamos a obtener un AFM para la expresión regular* $R = (0|10)^*011$*, de tal manera que* $L(M) = L(R)$*. El árbol de análisis para esta ER lo indicamos en la Figura 11. Junto a cada nodo etiquetado con una subexpresión que tiene que ser convertida a un AF se indica el nombre del autómata que se obtiene según el método (algunos nodos del árbol no generan autómatas). Los autómatas los hemos enumerado por el orden en que se construyen según el método recursivo* $ERtoAF(R)$*.*

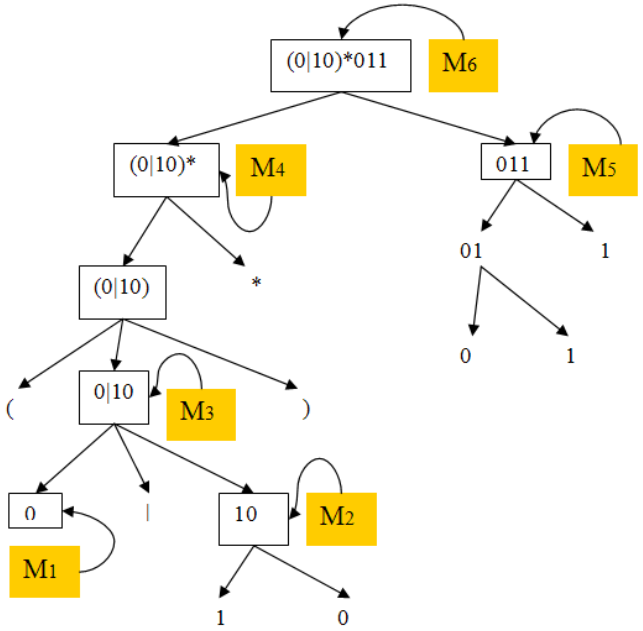
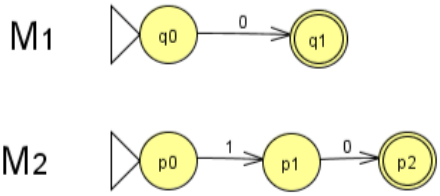


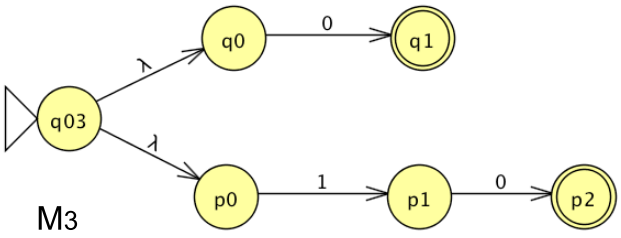
Figura 11: árbol de análisis de $(0|10)^*011$ y autómatas construidos en la traducción

Como $R = (0|10)^*011$ *es el tipo* $R_1 \circ R_2$*, con* $R_1 = (0|10)^*$ *y* $R_2 = 011$*, entonces se originan llamadas recursivas con las subexpresiones concatenadas. Como* $(0|10)^*$ *es un ER que no es del tipo base del algoritmo, entonces esta subexpresión genera otras llamadas y así hasta llegar a las subexpresiones más básicas, que son las que no tienen operadores de alternancia ni clausura.*

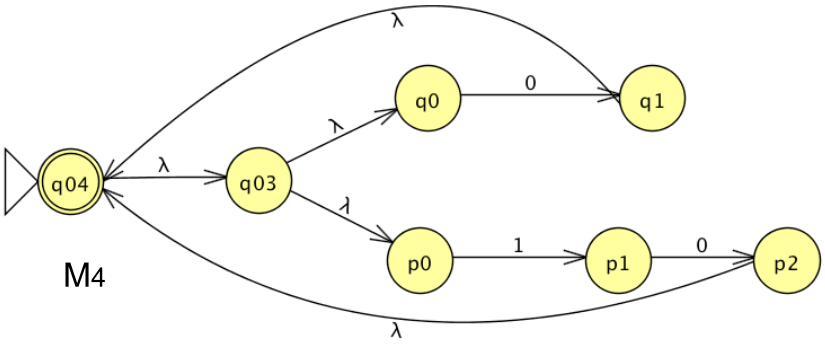
Primero se obtienen los autómatas M_1 *y* M_2 *para* 0 *y* 10 *del primer patrón* $(0|10)^*$*, mediante las llamadas* $construyeAFcadena(0)$ *y* $construyeAFcadena(10)$*. Aquí nombramos de forma diferente los estados de cada autómata para luego combinarlos, aunque los métodos que operan con autómatas se encargarían de renombrar los estados.*



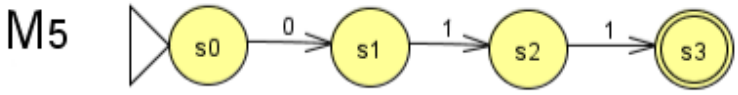
Se construye el autómata M_3 para la expresión regular $(0|10)$ *aplicando* $unionAF(M_1, M_2)$*:*



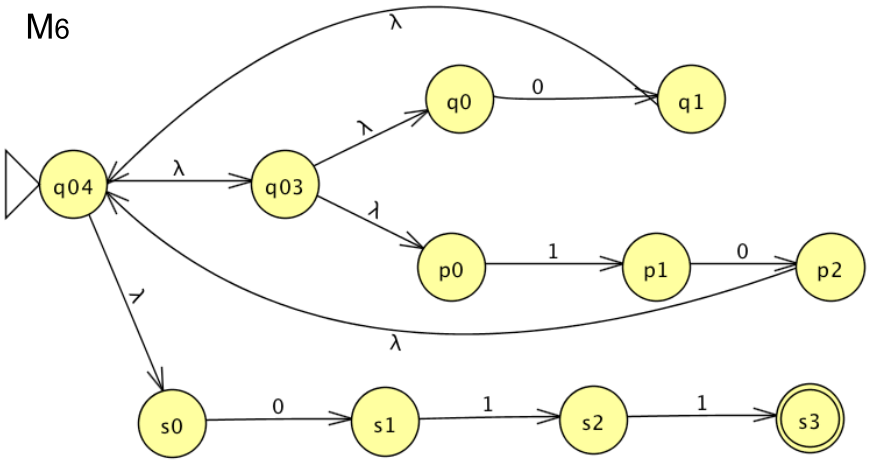
La siguiente subexpresión que se trata es $(0|10)^*$*, que es la clausura de* $(0|10)$*, ya traducida al autómata* M_3*. Entonces se obtiene* M_4 *a partir de* M_3 *aplicando el método* $clausuraAF(M_3)$*:*



Ahora falta el otro patrón concatenado 011 *en que se subdivide la* ER *original* $(0|10)^*011$*. Este patrón es un tipo básico para el que se puede obtener directamente el autómata* M_5 *por el método* $construyeAFcadena(011)$*:*

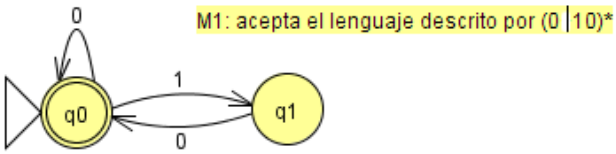


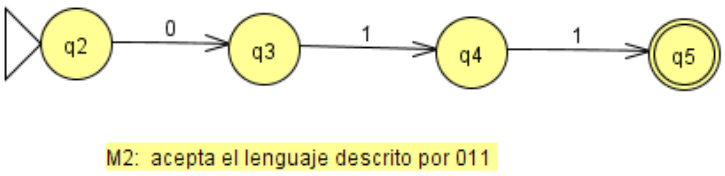
Finalmente se aplica $concatenacionAF(M_4, M_5)$ *y se obtiene* M_6 *que es el autómata final del proceso de traducción. Se cumple que* M_6 *acepta todas y sólo las cadenas descritas por la expresión regular* $(0|10)^*011$*, es decir* $L(M_6) = L((0|10)^*011)$*, como queríamos:*



El método *ERtoAF* es adecuado para ser implementado en una herramienta que requiera convertir expresiones regulares en implementaciones de autómata finitos y es muy eficiente, con **orden de complejidad lineal** $O(n)$, donde n es la longitud de la cadena con el contenido de la expresión regular. Pero para resolver un ejercicio de construir a mano un AF a partir de una ER, puede ser conveniente saltar algunos pasos en la división de la ER y simplificar para no tener que combinar demasiados autómatas parciales. Este "método manual simplificado" debe aplicarse con precaución para asegurar que el autómata que se obtiene es correcto, sobre todo si la expresión regular contiene operadores de clausura. Todo depende de lo complicada que sea la expresión regular.

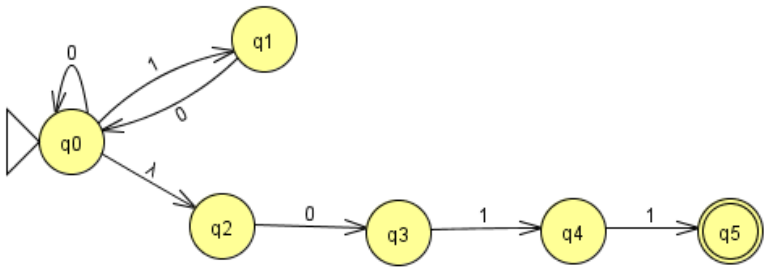
Ejemplo 30 *Podemos simplificar el proceso de construcción del autómata para* $(0|10)^*011$ *considerando que la expresión se subdivide en* $R_1 = (0|10)^*$ *y* $R_2 = 011$ *y manualmente obtenemos los autómatas* M_1 *y* M_2 *que aceptan* $L(R_1)$ *y* $L(R_2)$*, respectivamente evitando incluir* λ*-transiciones en el autómata* M_1*:*





M2: acepta el lenguaje descrito por 011

Ahora se combinan M_1 *y* M_2 *y se aplica el método de concatenación de autómatas, obteniendo el otro autómata que acepta también el lenguaje descrito por* $(0|10)^*011$ *y es más simple que el del ejemplo anterior:*



5.2. De los autómatas finitos a las expresiones regulares

Teorema de Kleene (parte de análisis)
Si L_r es un lenguaje aceptado por un autómata finito M entonces existe una expresión regular R tal que $L_{reg} = L(M) = L(R)$

La demostración de este teorema proporciona un algoritmo para traducir un autómata finito en una expresión regular equivalente (describe el mismo lenguaje que acepta el autómata). Hay distintos métodos para hacer esto; ahora veremos uno de ellos basado en la resolución de un sistema de ecuaciones de expresiones regulares para un autómata finito.

Sistema de ecuaciones de un autómata finito

A partir de un autómata finito (del tipo que sea) vamos a obtener un **sistema de ecuaciones de expresiones regulares**, que al resolverlo proporciona la expresión regular que describe al lenguaje aceptado por el autómata. Sea un autómata finito $M = (Q, V, \delta, q_0, F)$, con estados $Q = \{q_0, \ldots, q_n\}$ y alfabeto $V = a_1, a_2, \ldots, a_k$. En el sistema de ecuaciones de M hay una incógnita asociada a cada estado y los coeficientes y los términos independientes son expresiones regulares. Hay que obtener **una ecuación por cada estado** $q_i \in Q$ de la siguiente manera:

- Si q_i no es final entonces su ecuación tiene la siguiente forma:

  $$q_i = R_0 q_0 \mid R_1 q_1 \mid \cdots \mid R_i q_i \mid \cdots \mid R_n q_n$$

  y si q_i es final entonces $q_i = R_0 q_0 \mid R_1 q_1 \mid \cdots \mid R_i q_i \mid \cdots \mid R_n q_n \mid \lambda$

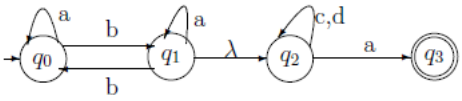
- La parte derecha de la ecuación para q_i es una unión o alternancia de términos (separados con $|$) y cada término de la forma $R_j q_j$ contiene una expresión regular R_j, que es el **coeficiente** de la incógnita-estado q_j. En la segunda ecuación λ es el **término independiente**.

- El coeficiente R_j del término $R_j q_j$ para el estado q_i ("término en q_j") se obtiene a partir del diagrama de transición agrupando con paréntesis y alternado con $|$ las ER de los símbolos o de λ que aparecen en todos los arcos que parten de q_i hasta q_j. Si sólo hay un arco desde q_i a q_j se omiten los paréntesis y R_j sería una ER sin operadores. Si desde el estado q_i no

hay transición hasta q_j entonces el término en q_j sería $\varnothing q_j$, que al ser el coeficiente vacío puede suprimirse este término de la ecuación para q_i.

Ej.Si $Q = \{q_0, q_1, q_2\}$, $V = \{a, b\}$ y tenemos que los arcos que parten desde q_0 hasta q_1 son $q_0 \xrightarrow{a} q_1$ y $q_0 \xrightarrow{\lambda} q_1$ entonces en la ecuación para q_0 incluiríamos el término $(a \mid \lambda)q_1$ en la parte derecha.

Ejemplo 31 *Sea el autómata:*



El sistema de ecuaciones de este autómata es:

$$\begin{aligned} q_0 &= aq_0 \mid bq_1 \\ q_1 &= bq_0 \mid aq_1 \mid \lambda q_2 \\ q_2 &= (c|d)q_2 \mid aq_3 \\ q_3 &= \lambda \end{aligned}$$

Como vemos el autómata tiene 4 estados y se obtiene una ecuación para cada estado. La última ecuación sólo contiene el término independiente λ, porque q_3 es estado final y no hay arcos que parten de q_3.

Método de resolución de sistema de ecuaciones del autómata para obtener la ER

En el sistema de ecuaciones de un AF cada estado q_i del autómata actúa como una incógnita que representa al conjunto de cadenas que se aceptan desde el estado q_i, como si este estado fuera el inicial. Lo que interesa es **resolver la incognita correspondiente al estado inicial**. Eso es lo que hace el algoritmo *AFtoER*: en lugar de resolver completamente el sistema de ecuaciones para un autómata M, sólo lo resuelve para la incógnita que representa al estado inicial del autómata y esa solución será la expresión regular que describe al lenguaje aceptado por el autómata. No obstante, incluimos en el algoritmo una parte opcional que sirve para resolver por completo el sistema. De esa forma, para cada estado q_i se obtiene una ER R_i de manera que:

$$\boxed{(q_i, x) \Rightarrow^* (q_F, \lambda) \text{ (la cadena } x \text{ se acepta desde } q_i) \text{ si y sólo si } x \in L(R_i)}$$

Algoritmo *AFtoER*

FUNCIÓN $AFtoER(M)$.

ENTRADA: un autómata finito $M = (Q, V, \delta, q_0, F)$, con estados $Q = \{q_0, \ldots, q_n\}$.

SALIDA: una expresión regular R_0 tal que $L(R_0) = L(M)$.

1. Obtener el sistema de ecuaciones del autómata M, considerando un orden en el conjunto de estados, por ejemplo, según los índices de estado y dejando el estado inicial en primer lugar: $q_0 < q_1 < \ldots < q_n$.

   / Bucle de transformación de ecuaciones para eliminar incógnitas. Al final obtiene ER solución para q_0 */*

2. FOR $i = n$ (último estado) DOWN-TO 0 (estado inicial).

   / Procesar la ecuación para el estado q_i */*

3. Obtener $\underline{otrosTerminos[q_i]}$:= es la unión (con |) de todos los términos de la parte derecha de la ecuación para q_i, excepto el término en q_i;
   Si sólo aparece el término en q_i entonces $otrosTerminos[q_i] = \varnothing$;

4. Obtener $\boxed{valor[q_i] := coef[q_i]^* \circ otrosTerminos[q_i]}$ donde $coef[q_i]$ es la ER que aparece como coeficiente de q_i y '$\circ$' indica concatenación y '*' clausura.
   Si no hay término en q_i ($coef[q_i] = \varnothing$) entonces $valor[q_i] := otrosTerminos[q_i]$;

5. Dejar ecuación para q_i expresada como $q_i = valor[q_i]$;

   / bucle de sustitución de q_i por valor de q_i en ecuaciones anteriores */*

6. FOR $j = i - 1$ DOWN-TO 0

7. Sustituir en ecuación para q_j la incognita q_i por $valor[q_i]$;

8. Transformar ecuación de q_j aplicando propiedad distributiva para que quede como una unión de términos y agrupar coeficientes del mismo estado con ( ) y uniendo con |.

9. END-FOR-j

10. END-FOR-i

11. DEVUELVE ($R_0 = valor[q_0]$); */* la solución obtenida para el estado inicial es la ER buscada */*

    OPCIONAL: en lugar de devolver solución para estado inicial entra en bucle que obtiene ER-solución para el resto de estados, para los que $valor[q_i]$ aun puede contener incógnitas.

    FOR $i = 0$ TO $n - 1$ */* sustituye incógnitas por soluciones (ER) y resuelve */*

    FOR $j = i + 1$ to n

    sustituir en ecuación para q_j la incógnita q_i por $valor[q_i]$;

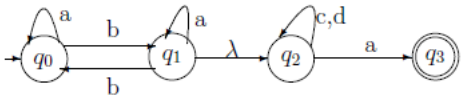
    END-FOR-j

    $valor[q_{i+1}] := coef[q_{i+1}]^* \circ otrosTerminos[q_{i+1}]$; */* obtiene ER-solución para q_{i+1} */*

    END-FOR-i

    DEVUELVE ($R_0 = valor[q_0]$,..., $R_n = valor[q_n]$) */* devuelve ER-solución para cada estado */*

Ejemplo 32 *Vamos a aplicar el método AFtoER al autómata del ejemplo 31 y comprobamos cómo se puede obtener automáticamente una ER para el lenguaje especificado por el autómata. Incluimos pasos adicionales para* ***simplificar y eliminar paréntesis innecesarios****. El sistema de ecuaciones ya se obtuvo anteriormente:*



$$q_0 = aq_0 \mid bq_1$$
$$q_1 = bq_0 \mid aq_1 \mid \lambda q_2$$
$$q_2 = (c|d)q_2 \mid aq_3$$
$$q_3 = \lambda$$

Para $\boxed{i = 3}$ *tenemos que* $otrosTerminos[q_3] = \lambda$ *y como no hay término en* q_3 *en parte derecha se tiene que* $valor[q_3] = otrosTerminos[q_3] = \lambda$. *Se deja la ecución para* q_3 *como* $q_3 = valor[q_3]$ *y se sustituye* q_3 *por su valor en las ecuaciones anteriores, transformando las ecuaciones. Sólo aparece en ecuación para* q_2 *y el sistema queda:*

$$q_3 = \lambda$$
$$q_2 = (c|d)q_2 \mid \underline{a}$$
$$q_1 = bq_0 \mid aq_1 \mid \lambda q_2$$
$$q_0 = aq_0 \mid bq_1$$

Para $\boxed{i = 2}$ *se obtiene* $otrosTerminos[q_2] = a$, $coef[q_2] = (c|d)$ *y*

$$valor[q_2] = coef[q_2]^* \circ otrosTerminos[q_2] = (c|d)^*a$$

Se deja la ecuación para q_2 *como* $q_2 = valor[q_2]$ *y se sustituye* q_2 *por su valor en las ecuaciones anteriores y el sistema queda:*

$$q_3 = \lambda$$
$$q_2 = \underline{(c|d)^*a}$$
$$q_1 = bq_0 \mid aq_1 \mid \underline{\lambda(c|d)^*a} \overset{(simplificando)}{=} \underline{bq_0 \mid aq_1 \mid (c|d)^*a}$$
$$q_0 = aq_0 \mid bq_1$$

Para $\boxed{i = 1}$ *se obtiene* $otrosTerminos[q_1] = (bq_0|(c|d)^*a)$, $coef[q_1] = a$ *y*

$$valor[q_1] = coef[q_1]^* \circ otrosTerminos[q_1] = a^*(bq_0|(c|d)^*a)$$

Se deja $q_1 = valor[q_1]$ *y se sustituye* q_1 *por su valor en la ecuación para* q_0 *y luego se modifica esta ecuación aplicando la propiedad distributiva (con las otras ecuaciones no fue necesario hacerlo):*

$$q_3 = \lambda$$
$$q_2 = (c|d)^*a$$
$$q_1 = a^*(bq_0|(c|d)^*a)$$
$$q_0 = aq_0 \mid \underline{ba^*(bq_0|(c|d)^*a)} \overset{(distrib.)}{=} aq_0 \mid ba^*bq_0 \mid ba^*(c|d)^*a = \overset{(agrupamos\ coef.)}{=} \underline{(a|ba^*b)q_0 \mid ba^*(c|d)^*a}$$

Para $\boxed{i = 0}$ *se obtiene* $otrosTerminos[q_0] = ba^*(c|d)^*a$, $coef[q_0] = (a|ba^*b)$ *y*

$$valor[q_0] = coef[q_0]^* \circ otrosTerminos[q_0] = (a|ba^*b)^*(ba^*(c|d)^*a)$$

Como $j = i - 1 = -1$ se salta el bucle de la línea 6 y por la línea 11 se tiene que la expresión regular R_0 que describe el lenguaje que acepta este autómata es:

$$\boxed{R_0 = valor[q_0] = (a|ba^*b)^*ba^*(c|d)^*a}$$

Ahora vamos a obtener la ER para el resto de estados (parte opcional del algoritmo), partiendo del sistema de ecuaciones ya transformado en pasos anteriores:

$$\begin{aligned} q_0 &= (a|ba^*b)^*ba^*(c|d)^*a \\ q_1 &= a^*(bq_0|(c|d)^*a) \\ q_2 &= (c|d)^*a \\ q_3 &= \lambda \end{aligned}$$

Para $\boxed{i = 0}$ *se sustituye q_0 por su valor en todas las ecuaciones posteriores (desde $j = 1$ hasta $j = 3$):*

$$\begin{aligned} q_0 &= (a|ba^*b)^*ba^*(c|d)^*a \\ q_1 &= a^*(b\,(a|ba^*b)^*ba^*(c|d)^*a \mid (c|d)^*a) \\ q_2 &= (c|d)^*a \\ q_3 &= \lambda \end{aligned}$$

Llegados a este punto nos damos cuenta de que en la parte derecha de las ecuaciones ya no aparecen incógnitas, luego el sistema está resuelto y no es necesario seguir con el proceso de sustitución con las siguientes ecuaciones.

Intuitivamente se aprecia en el diagrama de transiciones del autómata que desde el estado q_3 sólo se acepta λ, por eso la ER solución para ese estado es λ. Por otra parte desde q_2 se aceptan cadenas que consisten en una secuencia opcional de c's/d's seguida de una a, por eso la ER $(c|d)^*a$ obtenida para q_2 es la que describe este tipo de cadenas. Sin embargo no es sencillo "sacar a ojo" la ER que describe a las cadenas que se aceptan desde q_1 ni desde q_0 (las cadenas de $L(M)$). Por tanto con el algoritmo se puede obtener de forma automática la ER que describe al lenguaje $L(M)$ cuando no es evidente intuirla a partir del autómata.

6. Propiedades de cierre de los lenguajes regulares

La clase $\boxed{\mathcal{L}_{reg}}$ es un conjunto que contiene todos los lenguajes regulares y cumple algunas **propiedades de cierre**. Estas propiedades establecen que cuando se aplican ciertas operaciones a lenguajes regulares el resultado es otro lenguaje regular. En ese caso se dice que la clase de lenguajes regulares es **cerrada bajo las operaciones** en cuestión. Las propiedades de cierre aseguran que se pueden obtener autómatas finitos o expresiones regulares para lenguajes regulares complejos a partir de los formalismos obtenidos para otros más simples. Esto ya se ha visto en varios ejemplos. También se usan las propiedades de cierre en resultados teóricos que demuestran que ciertos lenguajes no son regulares (por reducción al absurdo).

Teorema 3 (algunas propiedades de cierre) *La clase $\mathcal{L}_{reg}$ de lenguajes regulares es CERRADA bajo las siguientes operaciones:*

1. *unión*
2. *concatenación*
3. *clausura*
4. *complementación*
5. *intersección*

Dem.- Casos 1, 2 y 3 Para 1 y 2 tenemos que demostrar que si L_1 y L_2 son lenguajes regulares cualesquiera entonces $L_1 \cup L_2$ y $L_1 \circ L_2$ son también lenguajes regulares. En el caso 3 probamos que si L_r es regular entonces L_r^* es regular. Para ello se obtiene una ER para cada caso.
Como se supone que L_1, L_2, L_r son regulares entonces, por definición, existen ER que los describen. Sean entonces R_1, R_2, R las expresiones regulares que describen a los lenguajes L_1, L_2 y L_r, respectivamente. Es decir, partimos de que: $L_1 = L(R_1), L_2 = L(R_2), L_r = L(R)$ y usamos las reglas semánticas de las ER que se estudiaron en el tema anterior:

1. El lenguaje $L_1 \cup L_2$ viene descrito por la expresión regular $R_1|R_2$. En efecto, $L_1 \cup L_2 = L(R_1) \cup L(R_2) = L(R_1|R_2)$. Entonces podemos asegurar que $L_1 \cup L_2$ es un lenguaje regular, como queríamos probar.
2. El lenguaje $L_1 \circ L_2$ es descrito por la expresión regular $R_1 \circ R_2$. En efecto, $L_1 \circ L_2 = L(R_1) \circ L(R_2) = L(R_1 \circ R_2)$. Luego podemos asegurar que $L_1 \circ L_2$ es un lenguaje regular.
3. Al lenguaje L_r^* lo describe la expresión regular R^*, ya que $L_r^* = (L(R))^* = L(R^*)$. Luego es L_r regular.

Caso 4 (cierre bajo complementación). Hay que probar que si L_r un lenguaje regular entonces $\overline{L_r}$ también lo es. Por el teorema de Kleene sabemos que si L_r es regular entonces puede ser descrito por una ER o aceptado por un AF indistintamente. Supongamos que M acepta L_r (el autómata M en general será un $AFND$). El siguiente método algorítmico, cuyos pasos aplican métodos que ya hemos estudiado, obtiene un AF que acepta $\overline{L_r}$:

1. $M_D \leftarrow AFNDtoAFD(M)$ [se obtiene otro equivalente determinista].
2. $\overline{M_D} \leftarrow complementaAFD(M_D)$ [este autómata acepta $\overline{L_r}$]

Caso 5 (cierre bajo intersección). Tenemos que probar que si L_1 y L_2 son lenguajes regulares entonces $L_1 \cap L_2$ también es lenguaje regular y para ello basta encontrar un AF que lo acepte. Supongamos que M_1 y M_2 son los autómatas finitos que aceptan L_1 y L_2 (existen por ser L_1 y L_2 regulares). Observamos lo siguiente, aplicando las leyes de De Morgan y de doble complementación

$$L_1 \cap L_2 = \overline{\overline{L_1 \cap L_2}} = \overline{\overline{L_1} \cup \overline{L_2}}$$

A partir de esta igualdad obtenemos un AF que acepte $L_1 \cap L_2$ aplicando el siguiente método algorítmico:

1. $M_{1D} \leftarrow AFNDtoAFD(M_1)$; $M_{2D} \leftarrow AFNDtoAFD(M_2)$ [se obtienen AFDs].
2. $\overline{M_{1D}} \leftarrow complementaAFD(M_{1D})$ [este autómata acepta $\overline{L_1}$]
3. $\overline{M_{2D}} \leftarrow complementaAFD(M_{2D})$ [este autómata acepta $\overline{L_2}$]
4. $M \leftarrow unionAF(\overline{M_{1D}}, \overline{M_{2D}})$ [este autómata acepta $\overline{L_1} \cup \overline{L_2}$]
5. $M_i \leftarrow complementaAFD(AFNDtoAFD(M))$; [acepta $\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$]

□

7. Limitaciones de los autómatas finitos y las ERs

No todos los lenguajes formales son regulares. Eso significa que hay cadenas con estructura demasiado compleja para que pueda ser descrita por un patrón regular o, por la equivalencia entre expresiones regulares y autómatas finitos, hay cadenas que no pueden ser validadas mediante un autómata finito, porque el autómata no dispone de los recursos necesarios para distinguir entre cadenas válidas e incorrectas cuando las cadenas tiene un formato no regular. Para hacerlo necesitaría usar memoria para guardar parte de la cadena que se procesa o un contador o necesitaría tener un número de estados variable (potencialmente infinito) para que los estados pudieran "recordar" todo lo necesario.

En cualquier caso, con esos cambios para aumentar la potencia de la máquina, ya no tendríamos un modelo de autómata finito, sería otra máquina teórica más potente, como un **autómata con pila** (número finito de estados+ memoria tipo pila, lo vemos en el tema 5) o una **máquina de Turing** (número finito de estados + memoria secuencial, es un modelo simplificado de computador). Que la máquina sea más potente es bueno por una parte, pues permite resolver problemas más complejos, pero por otra parte significa que para distinguir cadenas válidas de cadenas incorrectas se necesitaría en general más tiempo de ejecución, incluso tiempo exponencial.

Se pueden dar argumentos más o menos convincentes de que cierto lenguaje X no es regular, no vamos a ver demostraciones formales. Hay que llegar a la siguiente conclusión:

"Es imposible comprobar que una cadena pertenece al lenguaje X solamente con un recorrido símbolo a símbolo de izquierda a derecha sin retroceder y sin usar ningún tipo de memoria (variables de cadena, contadores, etc) porque ..., luego un autómata finito no puede aceptar el lenguaje X y por eso X no es regular."

Ejemplo 33 *Sea el lenguaje $L_{pban} = \{(^k)^k \mid k \geq 0\}$ de paréntesis balanceados anidados. Sospechamos que no es regular e informalmente lo justificamos pensando que un autómata finito sólo lee la cadena de izquierda a derecha y va cambiando de estado. Pero no puede "recordar" el número de paréntesis de apertura leídos para compararlo con el número de paréntesis de cierre, ya que se necesitaría un número de estados variable (para los distintos valores de k) o algún tipo de memoria para guardar los '(' leídos o contador de número de paréntesis de apertura leídos. Por tanto no es posible diseñar un AF para comprobar si una cadena es del tipo $(^k)^k$. Eso significa*

que el lenguaje L_{pban} no puede ser aceptado por un AF y por tanto no es regular. Si no es posible comprobar el correcto balanceo de cadenas $(^k)^k$ con un AF, tampoco será posible hacerlo en el caso de otras cadenas de paréntesis balanceados donde se anidan y concatenan paréntesis, como ()((()), ni en el caso de cadenas más complejas que incluyen paréntesis balanceados, como las expresiones aritméticas. Por tanto el lenguaje de las expresiones aritméticas no es un lenguaje regular.

También podemos argumentar que con una expresión regular tampoco se pueden describir estas cadenas $(^k)^k$ de forma precisa. Por ejemplo, la expresión regular que más se aproxima es $(^*)^*$ y es demasiado general porque describe a cadenas del tipo $(^k)^k$ y a otras muchas más donde no coincide el número de paréntesis de apertura y cierre. Por ejemplo, $(^*)^*$ tiene coincidencia con la cadena ((())) que no es una cadena de paréntesis balanceados anidados.

Ejemplo 34 *El lenguaje $L_{0i1} = \{w \in \{0,1\}^* \mid ceros(w) = unos(w)\}$ no es regular. En este lenguaje tenemos cadenas de igual número de cero que de unos, que pueden ir mezclados. Para comprobar si una cadena es válida hay que comparar el número de ceros con el número de unos. Se puede argumentar que con un AF no se puede llevar a cabo esta comprobación puesto que tendría que almacenar en memoria parte de la cadena leída o usar un contador. Como ese no es el modo de funcionamiento de un AF podría decirse que el lenguaje L_{0i1} no puede ser aceptado por un AF y por tanto no es regular. En uno de los problemas resueltos se demuestra que este lenguaje no es regular partiendo de que otro lenguaje no es regular y usando las propiedades de cierre de los lenguajes regulares.*

8. Aplicaciones de los Autómatas finitos

Los autómatas finitos no los suele manejar un usuario directamente, ni un programador de aplicaciones de alto nivel, sino un programador de aplicaciones de nivel medio-bajo. De hecho, una herramienta o lenguaje de programación que permite usar expresiones regulares dispone de un *regex engine* o del motor de expresiones regulares, que se encarga de traducir la ER a una implementación de autómata finito cuya tabla de transición se usa para las funciones que permiten el procesamiento de las cadenas.
Los programadores de motores de expresiones regulares sí tienen que tratar a más bajo nivel con técnicas basadas en autómatas finitos. También puede ser necesario el manejo de autómatas finitos por parte de los programadores involucrados en el desarrollo de un compilador o de un sistema operativo, programadores de robots de búsqueda y extracción de información en grandes volúmenes de texto (como en páginas web), diseñadores de circuitos digitales o de dispositivos de control en sistemas de eventos discretos, de protocolos de comunicación, etc.

8.1. Autómatas finitos y eventos discretos

Los *autómatas finitos clásicos* que hemos introducido en este tema están más bien orientados a la validación de formato de cadenas. Estos autómatas forman parte de un grupo más amplio autómatas finitos, que se conocen también como *máquinas de estado finito*, que modelan el comportamiento de **sistemas de eventos discretos**, que son sistemas en los que se distingue

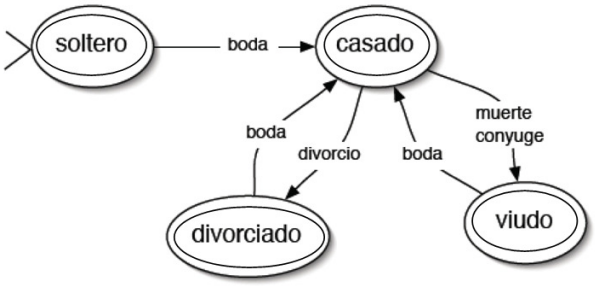
un número finito de estados y evolucionan en el tiempo cambiando de un estado a otro como reacción a determinados *eventos discretos* que se procesan en secuencia. Por defecto se supone que el sistema es *determinista*, es decir, si el sistema está en cierto estado y se recibe o genera un evento entonces el estado en el instante siguiente es único.

Por "eventos discretos" entendemos que son señales o acciones se producen de forma instantánea o no continua en el tiempo (o provienen de señales continuas que se discretizan), que son como los símbolos de un autómata finito clásico. Los sistemas de eventos discretos se pueden modelar a un alto nivel de abstracción mediante autómatas finitos deterministas. El autómata permite simular los cambios de estado en un modelo simplificado del sistema real y sirve para el diseño de dispositivos de control, para hacer verificaciones de algunas propiedades del sistema y en general para realizar un análisis de secuencias de eventos discretos, como verificar si los eventos en la secuencia tienen un orden correcto o comprobar otras propiedades.

Vamos a ver algunos ejemplo simples, que ilustran el modelado de sistemas de eventos discretos con autómatas finitos, sin entrar en aspectos concretos del modelado a más bajo nivel.

Ejemplo 35 *Un ejemplo de sistema de eventos discretos de la vida de una persona está formado por los diferentes estados civiles en que puede estar una persona y los eventos que pueden hacer cambiar el estado civil. Para modelizar el comportamiento de este sistema mediante un autómata hay que identificar los estados, los eventos y las transiciones que regulan la evolución de los estados.*

- *Identificación de los estados: distinguimos entre "soltero", "casado", "divorciado" y "viudo".*
- *Identificación de los eventos: los acontecimientos que producen cambio de estado son "boda", "divorcio" y "muerte de cónyuge".*
- *Diagrama de transición: en la figura de abajo tenemos el diagrama con los estados, eventos y transiciones entre estados civiles de una persona.*



Con este autómata interesa controlar que el orden en la secuencia de eventos es correcto, en el sentido de que es posible un cambio de estado por cada evento que se produce. Observamos que el autómata es incompleto, por lo que una secuencia de eventos no esperada hará que el autómata finalice anormalmente, como si llegara en un estado de error implícito. Ej. la secuencia de eventos "boda divorcio boda boda divorcio" no es correcta porque una persona no se puede casar de nuevo sin haberse divorciado antes, lo que se detecta en el autómata al no tener definida la regla de transición para el estado "casado" y el evento "boda".

En los modelos de eventos discretos se supone que se permanece en los estados un cierto tiempo, pero los eventos son instantáneos. Esto puede ser más o menos realista en este sistema de estados civiles. Por ejemplo, hay bodas que duran una semana, pero desde el punto de vista de la duración de una vida humana este tiempo puede considerarse despreciable. En el caso del evento "divorcio", pudiera ser inadecuado considerarlo como instantáneo, pues hay divorcios que duran años. En este caso, el modelo puede refinarse definiendo un nuevo estado "divorciándose", al que se llega desde "casado" mediante el evento "inicio divorcio".

A continuación mostramos otro ejemplo de **protocolos de comunicación**, relacionado con el envío y recepción de mensajes que producen cambios en el estado de la comunicación.

Ejemplo 36 *El comercio electrónico es una de las actividades esenciales que actualmente pueden hacerse en Internet. Una de las mayores preocupaciones es que las actividades se realicen de manera segura. Vamos a proponer un protocolo de comunicación simple que ayuda a gestionar el "dinero electrónico". El dinero electrónico se maneja mediante un archivo cifrado que un cliente puede utilizar para realizar un pago a una tienda a través de Internet. La tienda solicita al banco del cliente la transferencia del dinero a su cuenta y el banco emite un certificado de que el dinero ha sido ingresado.*

Para poder utilizar dinero electrónico se necesita respetar un protocolo de comunicación u orden en una secuencia de eventos de envío y recepción de mensajes para la manipulación segura del dinero en las distintas etapas de comunicación y para eso diseñaremos un autómata finito. Para simplificar el ejemplo suponemos que sólo tenemos tres participantes en la comunicación: un cliente*, una* tienda *y un* banco *y sólo se maneja un archivo de dinero electrónico. Desde el punto de vista de la tienda se tiene en cuenta el siguiente comportamiento, que define las reglas básicas del protocolo:*

1. *La tienda recibe la orden de pagar de un* cliente *que quiere comprar un producto, junto con dinero electrónico cifrado.*
2. *La tienda recibe la petición de cancelar el pago por parte del cliente. En ese caso se ignora la orden de pago del cliente si aún no ha solicitado al banco que le ingrese el dinero del cliente (se podría modificar para que el cliente pudiera cancelar antes de que el producto haya sido suministrado, se complica).*
3. *Tras recibir el pago del cliente la tienda solicita el dinero al banco. Es decir, envía el archivo de dinero del cliente al banco y solicita la emisión de un nuevo archivo que certifique que el dinero ha sido transferido a la cuenta de la tienda.*
4. *La tienda recibe la orden de transferir del banco, junto con el certificado de transferencia.*
5. *Finalmente, la tienda, tras recibir del banco la transferencia, suministra el producto al cliente y finaliza la operación de venta.*

Modelado del protocolo de comunicación:

Identificamos en primer lugar el conjunto de posibles eventos que pueden cambiar el estado en la comunicación de la tienda con el cliente y el banco:

$$Eventos = \{pagaC, cancelaC, solicitaT, suministraT, transfiereB\}$$

Unos eventos son generados por la tienda por envío de mensaje ("solicitaT", "suministraT") y los otros los genera el cliente ("pagaC", "cancelaC") o el banco ("transfiereB") Ahora podemos modelar la evolución de estados en la comunicación según las reglas del protocolo con un autómata finito como en la figura de abajo, en la que podemos identificar secuencias de eventos que no pueden darse en la comunicación, como por ejemplo la secuencia "pagaC solicitaT suministraT". El estado final indica que el proceso de compra finalizó con éxito.

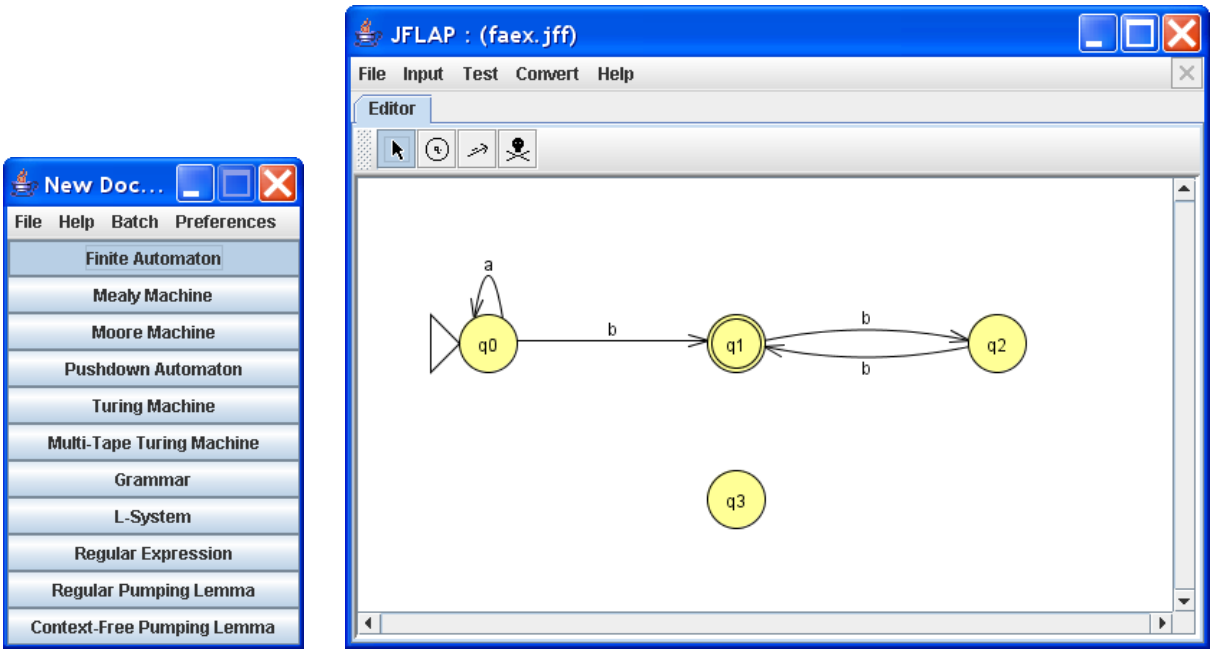
9. Apéndice: introducción a JFLAP

JFLAP (*Java Formal Languages and Automata Package*) es una herramienta didáctica que sirve de ayuda en el aprendizaje de diversos tópicos de Teoría de Autómatas y Lenguajes Formales. Permite resolver ejercicios de forma automática y paso a paso y puede servir para comprobar si uno ha resuelto correctamente ciertos problemas de aplicación de diversos métodos algorítmicos. Ha sido desarrollada por profesores y alumnos de la universidad de Duke (USA). La **web de JFLAP** es:

http://www.jflap.org

Seleccionando *Get JFLAP* se puede descargar la herramienta de forma gratuita tras rellenar un pequeño formulario para las estadísticas de uso. También se puede descargar **jflap.jar** en *Aula Virtual* → *Recursos* → *carpeta JFLAP*). Es una aplicación java que puede ejecutarse tanto en Linux como en Windows.
En http://www.jflap.org/tutorial/ se accede a *JFLAP Tutorial*, donde se explica de forma sencilla el uso de los distintos módulos de la herramienta

En este tema proponemos acceder a *Finite Automaton* del menú principal (ver **video-tutoriales** en *Aula Virtual* → *Recursos* → *carpeta JFLAP*). Seleccionando la opción **Finite Automaton** emerge una nueva ventana para cargar un *AF* ya existente o crear uno nuevo:



Funcionalidad:

- En el *canvas* de *Editor* se puede dibujar el diagrama de transición de un *AF* nuevo o modificar uno previamente cargado.
- Con *Input→ Step by State/Fast Run/Multiple Run* se puede comprobar paso a paso si una cadena es aceptada por el *AF* del editor (también comprobación rápida sin traza o con multiples cadenas). Implementa la simulación del *AF*, tanto si es determinista como si no (ver el video-tutorial **tutorial1-intro-AFD** en *Aula Virtual → Recursos → carpeta JFLAP*).
- Con *Convert→ Add Trap State to DFA* se completa un AFD añadiendo un estado de error.
- Con *Convert→ Minimize DFA* se obtiene el AFD mínimo equivalente. El método de minimización es distinto al que se ve en clase pero el resultado es el mismo porque el autómata mínimo es único (ver el video-tutorial **tutorial2-minimizaAFD**).
- Con *Convert→ Convert to DFA* se puede transformar un autómata no determinista en uno determinista equivalente (ver el video-tutorial **tutorial3-AFNDtoAFD**).
- Con *Convert→ Combine Automata* abre otra ventana añadiendo el diagrama de la ventana activa y el diagrama del autómata abierto en otra ventana, con objeto de que se puedan añadir transiciones para combinar de distintas formas los dos autómatas.
- Con *Test→ Highlight Nondeterminism* se puede comprobar si el autómata es determinista o no.
- Con *Test→ Compare Equivalence* se puede comprobar si dos autómatas son equivalentes (el mostrado en la ventana activa y otro abierto en otra ventana). No es necesario que sean deterministas, internamente los convierte a deterministas y aplica un método que comprueba la equivalencia (ver el video-tutorial **tutorial3-AFNDtoAFD**).

- Con *View→ Apply Ramdom Layout Algorithm→ GEM* se puede mejorar la visibilidad del diagrama, que es útil cuando un *AF* obtenido por transformación de otro tiene muchos estados.

10. Preguntas de evaluación

Aparte de los 29 ejemplos de los apuntes, que refuerzan los conocimientos teóricos, en esta sección tenemos preguntas que sirven también como auto-evaluación de los conocimientos teóricos. Abarcan problemas de razonamiento o demostración, problemas de aplicación algoritmos, métodos o definiciones y preguntas tipo test. Muchas de estas preguntas han aparecido en exámenes de cursos anteriores, aunque eso no significa que las preguntas que puedan aparecer en un examen sean exactamente del tipo de las que se incluyen aquí.

Nota:

- Los problemas de **mayor dificultad** se señalan con **(!)**.
- La etiqueta **JFLAP** delante de un ejercicio indica que es un problema con diagramas de transición editados en un fichero de JFLAP, que se encuentra en el Aula Virtual. En esta carpeta también aparecen algunos autómatas de los ejemplos mostrados en el tema. Conviene consultar los 3 **video-tutoriales** de apoyo a estos problemas disponibles en la carpeta de JFLAP.

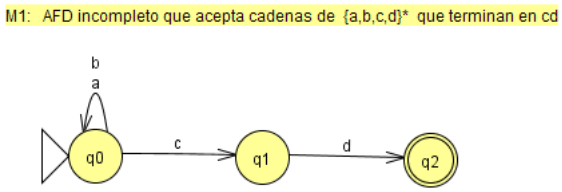
Problemas resueltos

1. (JFLAP) Diseña en JFLAP un AFD que acepte el lenguaje de las cadenas con alfabeto $\{0, 1\}$ que tiene un número impar de ceros.

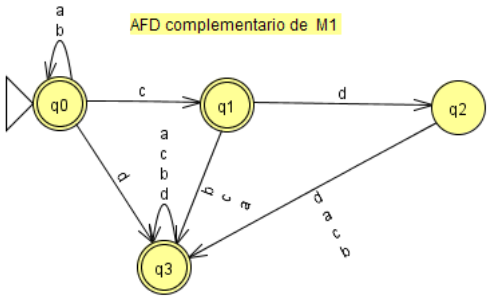
   Solución. La solución está en el video-tutorial de introducción a JFLAP llamado **tutorial1-intro-AFD**.

2. (JFLAP) Diseña directamente un AFD (puede ser incompleto) para aceptar los siguientes lenguajes:

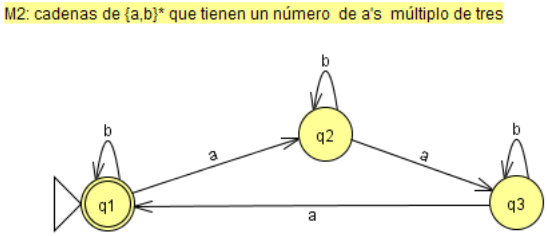
   - $L_1 = \{w \in \{a, b, c, d\}^* \mid w = xcd \wedge x \in \{a, b\}^*\}$.
     Solución. Puede probarse cargando *proResuelto-termina-cd.jff* en JFLAP:

   - $\overline{L_1}$: complementario de L_1 del apartado anterior.
     Solución. En *proResuelto-NO-termina-cd.jff*. Primero hay que completar el autómata M_1 anterior y después se complementan los estados finales.



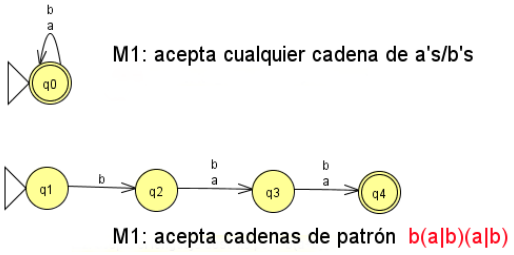
- L_2: cadenas de a's/b's que tienen un número de a's múltiplo de 3
  Solución. Puede probarse cargando *proResuelto-mul3a.jff*



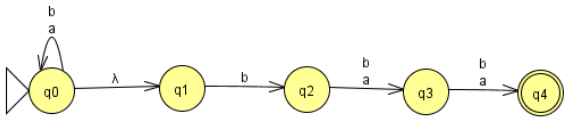
3. Consideremos el lenguaje que consiste en las cadenas con alfabeto $V = \{a, b\}$ que tienen una 'b' en la antepenúltima posición. Obtén la ER que describe a este lenguaje y a partir de ella un autómata finito aplicando el método $ERtoAF$ de manera simplificada, sin más de una λ-transición.

   Solución. una expresión regular que describe este lenguaje es $R = (a|b)^*b(a|b)(a|b)$, porque el patrón obliga a que haya una 'b' en la antepenúltima posición y en penúltima o última una 'a' o una 'b'. En las primeras posiciones puede haber una secuencia opcional de a's/b's (indicado con el patrón inicial $(a|b)^*$).

   A partir de esta ER se puede obtener un AF teniendo en cuenta el método para tal efecto, pero lo vamos a hacer de forma simplificada. En el método $ERtoAF$ la ER se subdivide inicialmente como $R = \underbrace{(a|b)^*b(a|b)}_{R_1}\underbrace{(a|b)}_{R_2}$, siguiendo las reglas de precedencia y asociatividad de operadores. Sin embargo, por la propiedad asociativa de la concatenación de ER podemos igualmente considerar que $R = \underbrace{(a|b)^*}_{R_1}\underbrace{b(a|b)(a|b)}_{R_2}$. No seguimos dividiendo la ER y obtenemos directamente autómatas sin λ-transiciones M_1 y M_2 para aceptar $L(R_1)$ y $L(R_2)$, respectivamente:



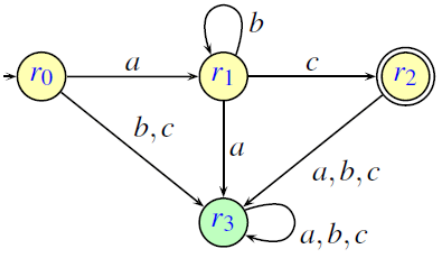
   Ahora se concatenan los autómatas M_1 y M_2 y se obtiene el autómata M que acepta el lenguaje que describe la ER



M: acepta cadenas descritas por la ER (a|b)*b(a|b)(a|b)

En este ejemplo la expresión regular $(a|b)^*b(a|b)(a|b)$ es sencilla, tanto que incluso se podría haber obtenido un $AFND$ sin λ-transiciones directamente, como se hizo en un problema del tema 2.

4. Dado el siguiente AFD M_{abc}, indica cuál es el alfabeto del autómata y deduce el lenguaje que acepta, expresándolo por comprensión. Justifica que el lenguaje obtenido es realmente el lenguaje aceptado por el autómata, analizando los caminos que llevan a estado final e indicando cálculos para diferentes casos de cadenas del lenguaje.



Solución. El **alfabeto** es $V = \{a, b, c\}$ porque sólo aparecen esos símbolos en los arcos. También podemos observar que el autómata es completo porque $\delta(r_i, s)$ está definida para todo $r_i \in Q, s \in V$.

Veamos qué tipo de cadenas acepta el autómata M_{abc}. Después de analizar los caminos que llevan a estado final, deducimos que $L(M_{abc}) = H$, donde

$$H = \{ab^n c \mid n \geq 0\}$$

Justificamos que este lenguaje H es realmente el lenguaje $L(M_{abc})$.

a) M_{abc} no es demasiado estricto: $\boxed{H \subseteq L(M_{abc})}$

Lo probamos usando un argumento basado en la definición formal de lenguaje aceptado por un AFD, mediante cálculos para los diferentes casos de cadenas en H, cuando $n = 0$ y cuando $n > 0$,

- Caso $n = 0$. Puede probarse formalmente que $ab^0c = ac \in H$ es aceptada por M porque $(r_0, ac) \Rightarrow^* (r_2, \lambda)$ y $r_2 \in F$, según se deduce del siguiente cálculo completo:

$$(r_0, ac) \overset{\delta(r_0,a)=r_1}{\Rightarrow} (r_1, c) \overset{\delta(r_1,c)=r_2}{\Rightarrow} (r_2, \lambda)$$

- Caso $n > 0$. La cadena de menor longitud de H con $n > 0$ es abc, que es aceptada porque $(r_0, abc) \Rightarrow^* (r_2, \lambda)$ y $r_2 \in F$, por el siguiente cálculo:

$$(r_0, abc) \overset{\delta(r_0,a)=r_1}{\Rightarrow} (r_1, bc) \overset{\delta(r_1,b)=r_1}{\Rightarrow} (r_1, c) \overset{\delta(r_1,c)=r_2}{\Rightarrow} (r_2, \lambda)$$

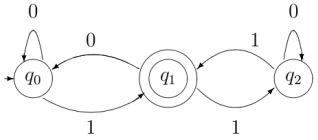
Generalizando para el caso de cadenas con $n > 0$ tenemos que son cadenas con una o más b's, de la forma $abb^{n-1}c$, que igualmente cumplen $(r_0, abb^{n-1}c) \Rightarrow^* (r_2, \lambda)$ porque:

$$(r_0, abb^{n-1}c) \Rightarrow (r_1, bb^{n-1}c) \Rightarrow (r_1, b^{n-1}c) \overbrace{\Rightarrow^*}^{\text{n-1 veces } \delta(r_1,b)=r_1} (r_1, c) \Rightarrow (r_2, \lambda)$$

Obsérvese que en el cálculo anterior se han resumido los n-1 pasos de cálculo para leer desde la segunda hasta la última b (si es que las hay) mediante el cálculo implícito $(r_1, b^{n-1}c) \Rightarrow^* (r_1, c)$, donde se aplica n-1 veces la regla de transición $\delta(r_1, b) = r_1$. Si $n = 1$ entonces $(r_1, b^{n-1}c) = (r_1, c)$ (no se aplica ninguna regla adicional para leer más b's). Informalmente las cadenas del tipo $abb^{n-1}c, n > 0$ son aceptadas porque leyendo 'a' se llega a al estado r_1, desde r_1 se pueden leer todas las b's quedado en r_1 y finalmente con la 'c' se alcanza el estado final r_2.

b) M_{abc} no es demasiado general: $\boxed{L(M_{abc}) \subseteq H}$ Veamos que es así mediante un argumento basado en los caminos del diagrama de transición. ¿Es posible que el autómata acepte cadenas que no siguen el patrón $ab^nc, n \geq 0$? La respuesta es que no, porque sólo hay un estado final y para alcanzar r_2 desde el inicial r_0 forzosamente hay que leer una a y pasar a r_1. En r_1 podemos leer opcionalmente una subcadena de $b's$, y para llegar a r_2 a la fuerza hay que leer una c. Si después de la c hay más símbolos entonces la cadena no es aceptada porque el camino acaba en el estado de error-trampa r_3. Por ejemplo, podemos comprobar que $\lambda \notin H$ y esta cadena es rechazada porque r_0 no es final. Ninguna cadena de longitud uno pertenece a H y el autómata las rechaza. Si la cadena tiene sólo a's, o sólo b's, o sólo c's, o mezcla símbolos o no termina en c tampoco es aceptada. En definitiva, ninguna cadena que no se ajuste al patrón $ab^nc, n \geq 0$ es aceptada por el autómata.

5. Obtén la expresión regular que describe el mismo lenguaje que acepta el siguiente autómata finito:



Solución. como no parece sencillo averiguar "a ojo" de qué expresión regular se trata, podemos aplicar el método algorítmico $AFtoER$ a este autómata finito. Primero se obtiene el sistema de ecuaciones del autómata:

$$\begin{aligned} q_0 &= 0q_0 \mid 1q_1 \\ q_1 &= 0q_0 \mid 1q_2 \mid \lambda \\ q_2 &= 1q_1 \mid 0q_2 \end{aligned}$$

Comenzamos por la última ecuación (para q_2) y obtenemos $valor[q_2] = 0^*1q_1$, se cambia la ecuación para q_2 dejando $valor[q_2]$ en el segundo miembro y se sustituye q_2 por su valor en ecuaciones anteriores (sólo aparece la del estado q_1):

$$\begin{aligned} q_2 &= 0^*1q_1 \\ q_1 &= 0q_0 \mid 10^*1q_1 \mid \lambda \\ q_0 &= 0q_0 \mid 1q_1 \end{aligned}$$

Seguimos con la ecuación anterior, para q_1 y obtenemos $valor[q_1] = (10^*1)^*(0q_0|\lambda)$, se cambia la ecuación para q_1 el valor de q_1 se sustituye en la ecuación anterior, se aplica la propiedad distributiva y se agrupan coeficientes, quedado el sistema:

$$q_2 = 0^*1q_1$$
$$q_1 = (10^*1)^*(0q_0|\lambda)$$
$$q_0 = 0q_0|1(10^*1)^*(0q_0|\lambda) \overset{distrib.}{=} 0q_0|1(10^*1)^*0q_0|1(10^*1)^* \overset{agrupar\ coef.}{=} (0|1(10^*1)^*0)q_0|1(10^*1)^*$$

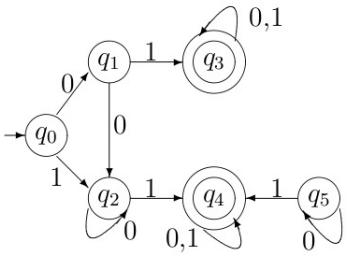
Finalmente para se obtiene $valor[q_0] = (0|1(10^*1)^*0)^*1(10^*1)^*$ y es la ER R_0 tal que $R_0 = L(M)$:

$$\boxed{R_0 = (0|1(10^*1)^*0)^*1(10^*1)^*}$$

6. **(!)** Demuestra que el lenguaje $L_{0i1} = \{w \in \{0,1\}^* \mid ceros(w) = unos(w)\}$ no es regular es partiendo del hecho de que el lenguaje $L_s = \{0^k1^k \mid k \geq 0\}$ no es regular y **usando las propiedades de cierre de los lenguajes regulares**.

   Solución. Partiendo de que L_s no es regular probamos que L_{0i1} no es regular por reducción al absurdo. HIPÓTESIS: supongamos que L_{0i1} es regular. Observamos que $L_s = L_{0i1} \cap L(0^*1^*)$. Como $L(0^*1^*)$ es regular, porque es un lenguaje descrito por una ER, entonces al ser L_{0i1} regular (hipótesis) debería serlo también el lenguaje L_s, porque la clase de lenguajes regulares es cerrada bajo intersección. Pero como hemos partido de que L_s no es regular entonces la hipótesis de que L_{0i1} es regular es falsa. Luego L_{0i1} no es regular.

7. Obtener el AFD mínimo equivalente al siguiente autómata, mostrando el desarrollo de los pasos de aplicación del algoritmo *minimizaAFD*.



   Solución. Lo primero que hacemos es **eliminar el estado inaccesible** q_5. Observamos que el autómata es completo, así que pasamos a construir y marcar la tabla triangular. Primero **marcamos inicialmente en la tabla triangular** las casillas donde un estado es final y el otro no, que son las señaladas con una X recuadrada en la tabla triangular resultante (ver más adelante). Después hacemos un **recorrido por filas de la tabla**, revisando paso a paso las casillas no marcadas y obteniendo las transiciones para el par de estados de la casilla con cada símbolo.

   a) $casilla(q_1, q_0)$:
   ▷ Para símbolo 0 se obtiene $q_2 = \delta(q_1, 0)$ y $q_1 = \delta(q_0, 0)$. Consulta $casilla(q_2, q_1)$ y no está marcada $\rightsquigarrow$ añade el par (q_1, q_0) a $lista(q_2, q_1)$.
   ▷ para símbolo 1 se obtiene $q_3 = \delta(q_1, 1)$ y $q_2 = \delta(q_0, 1)$. Consulta $casilla(q_3, q_2)$ y está marcada $\rightsquigarrow$ marca $casilla(q_1, q_0)$ [son distinguibles].
   Puede indicarse este paso y los siguientes, de forma alternativa, con una mini-tabla

de transición mostrando sólo los estados q_1, q_0 y los símbolos 0 y 1, como en un ejemplo de los apuntes.

b) $\underline{casilla(q_2, q_0)}$:
▷ Para símbolo 0 se obtiene $q_2 = \delta(q_2, 0)$ y $q_1 = \delta(q_0, 0)$. Consulta $casilla(q_2, q_1)$ y no está marcada $\rightsquigarrow$ añade el par (q_2, q_0) a $lista(q_2, q_1)$.
▷ Para símbolo 1 se obtiene $q_4 = \delta(q_2, 1)$ y $q_2 = \delta(q_0, 1)$. Consulta $casilla(q_4, q_2)$ y está marcada $\rightsquigarrow$ marca $casilla(q_2, q_0)$ [son distinguibles].

c) $\underline{casilla(q_2, q_1)}$:
▷ Para símbolo 0 se obtiene $q_2 = \delta(q_2, 0)$ y $q_2 = \delta(q_1, 0)$. Se obtienen estados iguales, prueba siguiente símbolo.
▷ para símbolo 1 se obtiene $q_4 = \delta(q_2, 1)$ y $q_3 = \delta(q_1, 1)$. Consulta $casilla(q_4, q_3)$ y no está marcada $\rightsquigarrow$ añade el par (q_2, q_1) a $lista(q_4, q_3)$.

d) $\underline{casilla(q_4, q_3)}$:
▷ Para símbolo 0 se obtiene $q_4 = \delta(q_4, 0)$ y $q_3 = \delta(q_3, 0)$. Se obtiene la misma pareja, prueba siguiente símbolo.
▷ Para símbolo 1 se obtiene $q_4 = \delta(q_4, 1)$ y $q_3 = \delta(q_3, 1)$. Se obtiene la misma pareja, termina.

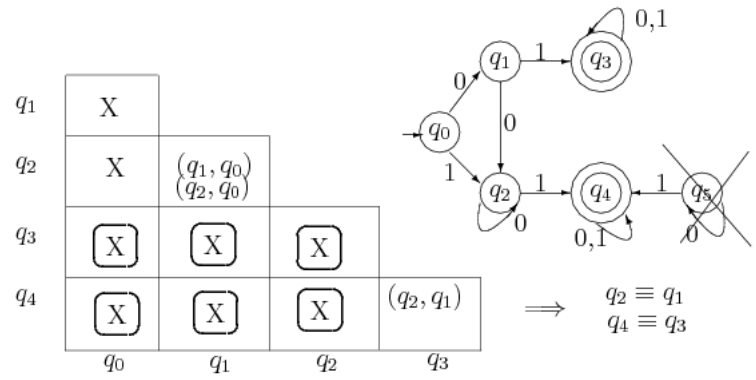


Figura 12: Tabla triangular ya marcada

Al final se deduce que $q_1 \equiv q_2$ y $q_3 \equiv q_4$, porque las casillas correspondientes a esos pares de estados se han quedado sin marcar. A partir de ahí se obtiene el **conjunto cociente** $Q/_{\equiv}$, que será el conjunto de estados del autómata mínimo:

$$Q/_{\equiv} = \{\{q_0\}, \{q_1, q_2\}, \{q_3, q_4\}\}$$

Como vemos se obtiene un conjunto de estados reducido (tiene 3 estados frente a los 5 de Q), y cada estado-clase consiste en la unión de los estados que son equivalentes entre sí. El **estado inicial** del autómata mínimo es $\{q_0\}$ y el conjunto de **estados finales** es: $\{\{q_3, q_4\}\}$. Falta calcular la función de transición δ_{min} a partir de la función δ del AFD original:

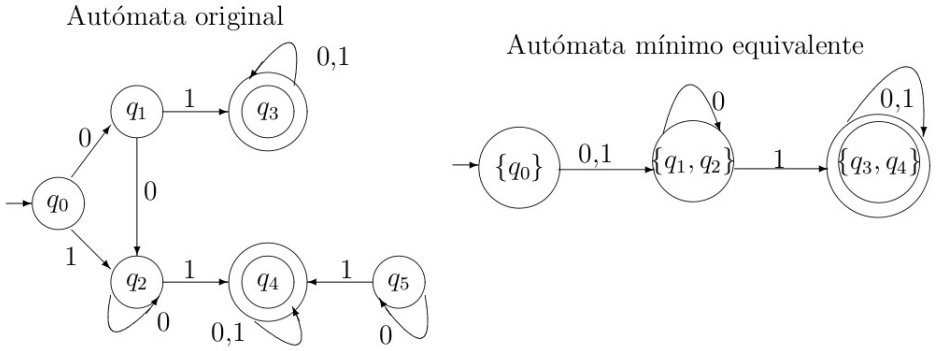
- Para $S = \{q_0\}$ de Q_{min} escogemos q_0 (no hay otra posibilidad) y definimos nuevas transiciones en δ_{min} con símbolos 0 y 1:

• Como en el AFD original tenemos que $\delta(q_0, 0) = q_1$ y el estado q_1 pertenece al estado-clase $\{q_1, q_2\}$ de Q_{min}, entonces $\delta_{min}(\{q_0\}, 0) = \{q_1, q_2\}$

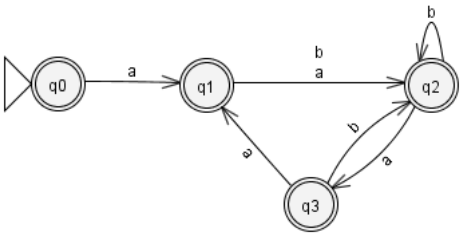
• Como $\delta(q_0, 1) = q_2$ y $q_2 \in \{q_1, q_2\}$ de Q_{min} entonces $\delta_{min}(\{q_0\}, 1) = \{q_1, q_2\}$

- Para $S = \{q_1, q_2\} \in Q_{min}$ escogemos q_1 y calculamos nuevas transiciones:
  • Como $\delta(q_1, 0) = q_2$ y $q_2 \in \{q_1, q_2\}$ de Q_{min}, entonces $\delta_{min}(\{q_1, q_2\}, 0) = \{q_1, q_2\}$
  • Como $\delta(q_1, 1) = q_3$ y $q_3 \in \{q_3, q_4\}$ de Q_{min}, entonces $\delta_{min}(\{q_1, q_2\}, 1) = \{q_3, q_4\}$
- Para $S = \{q_3, q_4\} \in Q_{min}$ escogemos q_3 y calculamos nuevas transiciones:
  • Como $\delta(q_3, 0) = q_3$ y $q_3 \in \{q_3, q_4\}$ de Q_{min}, entonces $\delta_{min}(\{q_3, q_4\}, 0) = \{q_3, q_4\}$
  • Como $\delta(q_3, 1) = q_3$ y $q_3 \in \{q_3, q_4\}$ de Q_{min}, entonces $\delta_{min}(\{q_3, q_4\}, 1) = \{q_3, q_4\}$

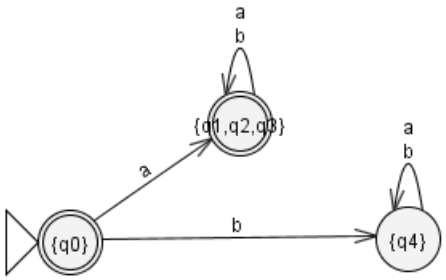
Finalmente mostramos el diagrama de transición del autómata mínimo equivalente obtenido:



8. (JFLAP) Obtener el AFD mínimo equivalente al siguiente autómata:



Solución. Observamos que el AFD es incompleto, por lo que debemos añadir un estado de error, por ejemplo q_4, antes de construir la tabla para marcar parejas de estados distinguibles en el proceso método de marcado de la tabla triangular. El AFD mínimo que se obtiene aplicando el algoritmo de minimización es el siguiente (se omite el desarrollo):

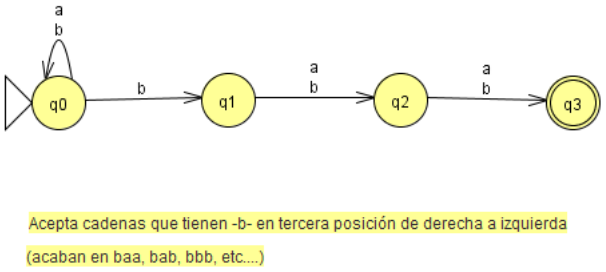


Por los estados que resultan de construir el autómata cociente (mínimo) podemos observar que $q_1 \equiv q_2 \equiv q_3$. Esos tres estados equivalentes entre sí se han agrupado en el estado-clase $\{q_1, q_2, q_3\}$ del AFD mínimo.

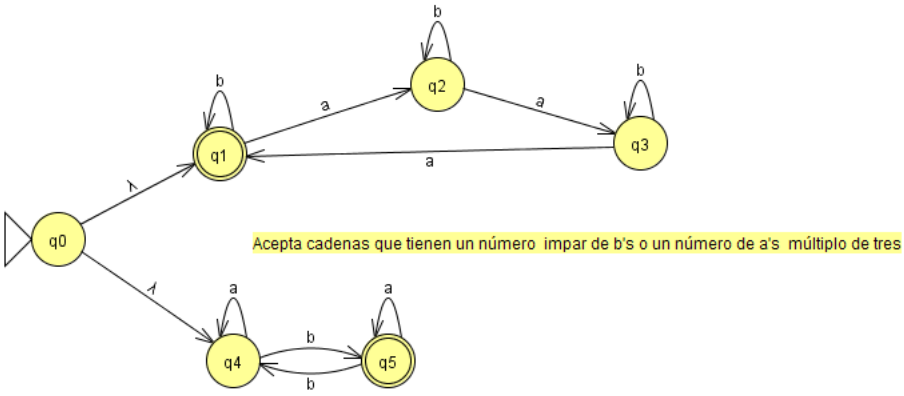
El AFD original está en en el fichero *proResuelto-minimizar.jff* y se puede minimizar automáticamente con JLAP. En el video-tutorial **tutorial2-minimizaAFD** mostramos cómo la solución obtenida a mano aplicando el algoritmo $AFNDtoAFD$ es equivalente a la que se obtiene con JFLAP aplicando otro algoritmo distinto.

9. (JFLAP) Obtén autómatas finitos no deterministas para aceptar los lenguajes indicados.

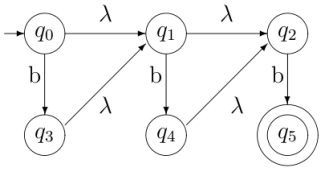
   - L_1: cadenas de $a's/b's$ cuyo tercer símbolo desde la derecha es b
     Solución. En el fichero *proResuelto-b-en-3pos.jff*. El diagrama de transición es:



   - L_2: cadenas de alfabeto $V = \{a, b\}$ que tienen un número impar de $b's$ o un número de $a's$ múltiplo de tres
     Solución. En este caso es conveniente expresar el lenguaje como unión de dos lenguajes más simples, obtener un AF para cada uno y combinar los autómatas obtenidos con λ-transiciones para aceptar el lenguaje completo. En *proResuelto-imparb-mult3a.jff* tenemos la solución:



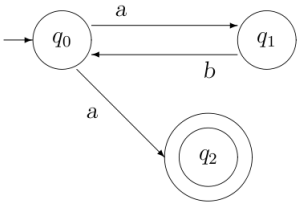
10. Para el siguiente AFND, calcula $LC(q)$ para cada estado q e indica qué lenguaje acepta y justifica brevemente por qué:



Solución. Tenemos que: $LC(q_0) = \{q_0, q_1, q_2\}, LC(q_1) = \{q_1, q_2\}, LC(q_2) = \{q_2\}, LC(q_3) = \{q_3, q_1, q_2\}, LC(q_4) = \{q_4, q_2\}, LC(q_5) = \{q_5\}$.
El AFND acepta el lenguaje finito $\{b, bb, bbb\}$, porque la única forma de llegar al estado final q_5 es leyendo una b, dos $b's$ o tres $b's$. Con las cadena que tienen más de 3 b's no

hay camino que acabe en estado final y no hay más símbolos en el alfabeto aparte de b. Tampoco acepta la cadena vacía porque para llegar a q_5 hay que leer a la fuerza una b.

11. Aplica el método de transformación $AFNDtoAFD$ para obtener un AFD equivalente al siguiente autómata. Muestra el desarrollo de los pasos principales y el diagrama de transición resultante.



Solución. Al no tener λ-transiciones, tenemos en cuenta que $LC(E) = E$, para todo subconjunto de estados E. El **estado inicial** es $\{q_0\}$, porque $LC(q_0) = \{q_0\}$.

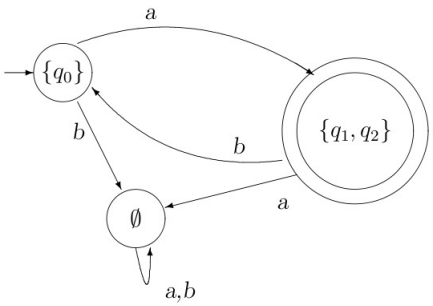
Calculamos δ_D para el estado $\{q_0\}$ con los símbolos a y b:

$$\delta_D(\{q_0\}, a) = LC(\{q_1, q_2\}) = \{q_1, q_2\}$$
$$\delta_D(\{q_0\}, b) = LC(\varnothing) = \varnothing$$

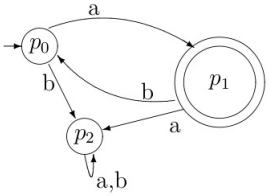
Se calculan las reglas de transición para los nuevos estados:

$$\delta_D(\{q_1, q_2\}, a) = LC(\delta(q_1, a) \cup \delta(q_2, a)) = \varnothing$$
$$\delta_D(\{q_1, q_2\}, b) = LC(\delta(q_1, b) \cup \delta(q_2, b)) = LC(\{q_0\}) = \{q_0\}$$
$$\delta_D(\varnothing, a) = \varnothing; \quad \delta_D(\varnothing, b) = \varnothing$$

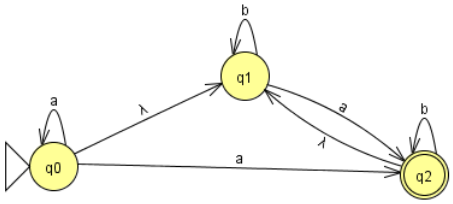
No se han generado estados nuevos, por tanto ya se ha terminado de calcular la **función de transición**. El AFD sólo tiene un **estado final**: $\{q_1, q_2\}$, porque q_2 es final en el AFND. El **diagrama de transición** es:



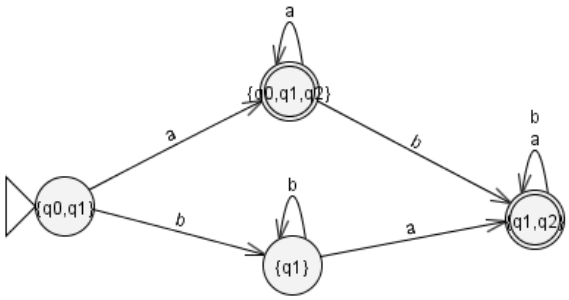
Si se prefiere se pueden renombrar los estados, para que vuelvan a tener nombres comunes en lugar de subconjuntos. El diagrama anterior podría quedar como:



12. (JFLAP) Obtén un AFD equivalente al siguiente autómata:

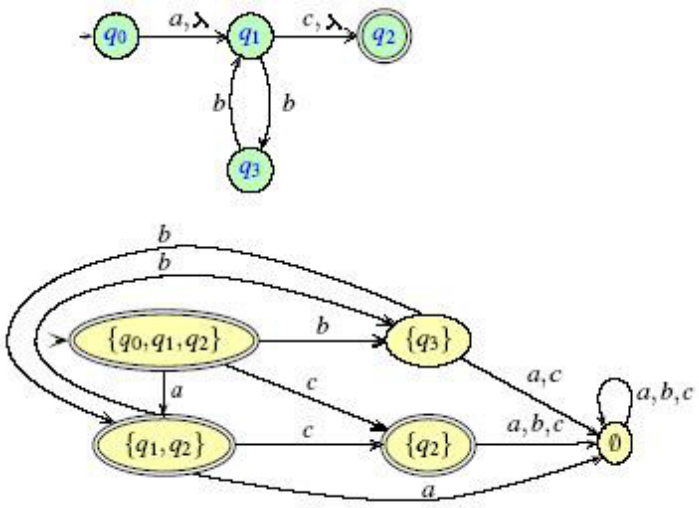


Solución. El autómata determinista que se obtiene es (se omite el desarrollo del algoritmo AFNDtoAFD):

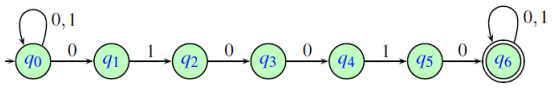


El diagrama del AFND está en el fichero *proResuelto-afndToAFD.jff* y se puede hacer la transformación automáticamente con JFLAP. Para ello se accede a *Convert to DFA*: ver el video-tutorial **tutorial3-AFNDtoAFD**.

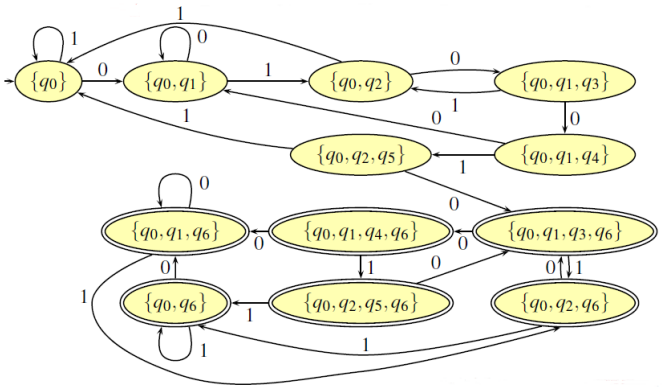
13. Mostramos un AFND con 4 estados y el AFD que se obtiene con el algoritmo $AFNDtoAFD$, que tiene el mismo número de estados más el estado de error (se omite el desarrollo):



14. Obtén un AFD equivalente al siguiente AFND:



Solución. El AFD que se obtiene por el algoritmo de transformación es (se omite el desarrollo):



Problemas propuestos

1. (JFLAP) Construye autómatas finitos para aceptar cadenas descritas por las siguientes expresiones regulares (puede aplicarse el método *ERtoAF* de manera simplificada para evitar demasiadas λ-transiciones):

   a) $(a|ba)^*((bb)^*|aab)$

   b) $(d(ab)^*)^*\, da(ba)^*$

   Nota: los autómatas para cada ER, que resultan de la combinación de AF más simples para subexpresiones regulares, pueden encontrarse en el Aula Virtual. En el video-tutorial ***tutorial4-ERtoAF-jflap*** se obtiene con JFLAP el AF para la ER del apartado a) con un método algo distinto y luego se comprueba que es equivalente a la solución obtenida en el fichero *propuestoERtoAF5-a.jff*. De esta forma se puede usar JFLAP para comprobar si el AF obtenido a mano para la ER del apartado a) es correcto.

2. Consideremos las cadenas del tipo *lista de cadenas*, donde una lista de cadenas consiste en una secuencia opcional de cadenas de a's/b's separadas por ; y encerradas entre corchetes. Algunas listas válidas son "[ab]", "[b;abba;aa]" y también se admite la lista vacía "[]". Las cadenas "[b;bbaa;]", "abab]", "[a;;ab;bbb]" no se consideran listas válidas.

   a) Identifica el alfabeto del lenguaje $LISTAc$ que contiene las cadenas que se ajustan a la especificación anterior.

   b) (JFLAP) Obtén un autómata finito M que sirva para comprobar si una cadena es una lista de cadenas válida.

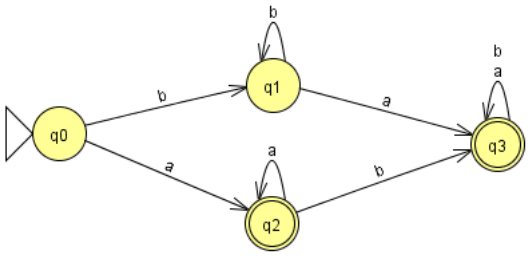
   c) Obtén una expresión regular que describa el lenguaje $LISTAc$ a partir del autómata M obtenido en el apartado anterior, obteniendo el sistema de ecuaciones y aplicando el método *AFtoER*. Indica además cuál es la expresión regular R_i que cumple $x \in L(R_i) \Leftrightarrow [(q_i, x) \Rightarrow^* (q_F, \lambda)]$, siendo x una cadena, q_i cualquier estado del autómata y q_F un estado final.

3. **(!)** Diseña un método algorítmico que, dado un autómata finito de entrada M, construya un autómata que acepta las cadenas de $L(M)^R$, donde este lenguaje es el lenguaje reflejo de $L(M)$ (contiene las cadenas al revés). Con eso quedaría probado que **la clase de lenguajes regulares es cerrada bajo la operación de reflexión**.

4. Diseña directamente AFDs para aceptar los siguientes lenguajes. Para todos ellos realiza un test de prueba de corrección del diseño con cadenas que deban ser aceptadas (válidas/pertenecen al lenguaje) o rechazadas (incorrectas/no pertenecen al lenguaje) que cubran los diferentes casos.

   - L_1: cadenas de a's/b's tal que no contienen la subcadena *bbb*.
   - $L_2 = \{w \in \{a,b\}^* \mid$ ni *aa* ni *bb* es subcadena de $w\}$
   - (**!**) L_3 : cadenas de a's/b's tal que contienen la subcadena *ab* y la subcadena *ba*.
   - L_4: cadenas de 0's/1's que empiezan y terminan por 11. Se supone que $11 \in L_4$.
   - $\overline{L_4}$: lenguaje complementario de L_4

5. (JFLAP) Sea el siguiente AFD:



   a) Analiza el lenguaje que acepta y descríbelo matemáticamente por comprensión.

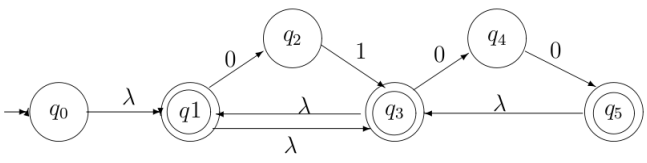
   b) Comprueba si es mínimo y en caso de no serlo obtén el AFD mínimo equivalente aplicando el algoritmo de minimización.

   El diagrama está en el fichero JFLAP *propuesto-Minimizar.jff*. La solución puede hacerse a mano y comprobarla con la que obtiene JFLAP de forma análoga a como se indica en el video-tutorial **tutorial2-minimizaAFD**.

6. Obtén autómatas finitos no deterministas para aceptar los lenguajes indicados.

   - L_1: cadenas de texto de alfabeto ASCII en las que aparecen las palabras "cine" o "web".
   - L_2: cadenas de dígitos de 0 a 9, de forma que el dígito más a la derecha aparece también antes en la cadena. Se supone que no debe aceptar cadenas de longitud menor que 2.
   - $L_3 = \{w \in \{a,b,c\}^* \mid w = ab^ic \vee w = (ca)^j, i \geq 0, j > 0\}$.

7. Dado el siguiente autómata finito M:



   a) Describe la función de transición de M en notación matemática.

b) (**!**) El lenguaje aceptado por M puede expresarse mediante operaciones con los lenguajes simples $\{01\}$ y $\{00\}$. Describe el lenguaje aceptado de esa forma.

c) Indica un cálculo por el que se acepta la cadena 0101 en M y otro por el que no acepta. ¿Cuál es el tiempo de ejecución del autómata M con la cadena 0101?

d) Obtén el AFD mínimo equivalente a este autómata aplicando los algoritmos que sean necesarios. Muestra el desarrollo, no sólo el resultado final.

e) ¿Cuál es el tiempo de ejecución de la cadena 0101 en el autómata mínimo? Justifica la respuesta mostrando un cálculo para esa cadena.

8. Consideremos vehículos automáticos de cuatro marchas. Los cambios de una marcha a otra se producen en función del régimen de revoluciones del automóvil y de la acción del conductor sobre el pedal del acelerador. De esta manera, a medida que el conductor presiona el pedal del acelerador el coche aumenta sus revoluciones y cuando éstas llegan a un límite máximo (MAX) este evento es recibido por el controlador del cambio automático y procede a aumentar la marcha. Por el contrario, si el conductor no presiona el acelerador y las revoluciones disminuyen por debajo de cierto límite mínimo (MIN), se producirá una reducción de la marcha. La excepción se produce cuando el conductor presiona fuertemente (PF) el pedal del acelerador, ya que el cambio automático interpreta que el conductor desea incrementar la velocidad en muy poco tiempo (en una situación de peligro) y procede a reducir la marcha para aumentar la potencia.

   ▷ Se requiere diseñar un autómata finito para modelar este sistema simplificado de cambio automático de marchas. Hay que identificar los estados, eventos (símbolos) y mostrar el diagrama de transición del autómata.

Preguntas tipo test

Para cada pregunta señala la opción correcta. Como ejercicio adicional, intenta justificar por qué es cada opción verdadera o falsa. Una pregunta tipo test también puede enunciarse en un examen como pregunta de razonar la verdad o falsedad de una afirmación.

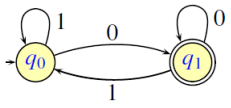
1. Señala la opción <u>falsa</u>. Dado un alfabeto V y un lenguaje L con alfabeto V,

   a) $\overline{L}$ es un lenguaje regular.

   b) Si L es regular entonces $L \circ L^*$ es un lenguaje regular.

   c) Si L es finito entonces es un lenguaje regular.

2. Señala la opción <u>verdadera</u>. Dado el AFD



   a) Acepta el lenguaje de cadenas de $\{0,1\}^*$ que sólo tienen ceros.

   b) Acepta el lenguaje de cadenas que se ajustan al patrón $(10)^n, n > 0$.

   c) Acepta el lenguaje de cadenas de $\{0,1\}^*$ que acaban en cero.

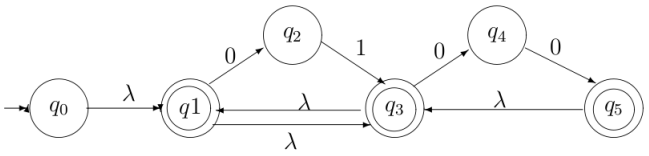
3. Señala la opción verdadera. Considera el método $clausuraAF(M_1)$,

   a) Si se modifica el método haciendo que $F := F_1 \cup \{q_0\}$, en lugar de $F := \{q_0\}$ entonces el autómata resultante sigue aceptando $L(M_1)^*$.

   b) No es posible diseñar otro algoritmo equivalente que no incluya λ-transiciones en el autómata de salida que acepta la clausura de $L(M_1)^*$.

   c) Ninguna de las opciones anteriores es cierta.

4. Señala la opción falsa. Dado el AFND



   a) Si se suprime el arco λ de q_1 a q_3, el autómata resultante es equivalente al original.

   b) Si sólo se deja el estado q_1 como final, el autómata resultante es equivalente al original.

   c) Se cumple que $(q_0, 010001) \Rightarrow^* (q_3, \lambda)$

5. Señala la opción verdadera.

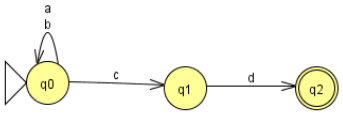
   a) Sea un autómata finito $M = (Q, V, \delta, q_0, F)$. En la expresión de abajo $\Rightarrow^*$ es notación de cálculo en cero o más pasos y $\Rightarrow$ es un símbolo de implicación lógica:

   $$[(q_0, \lambda) \Rightarrow^* (q', \lambda) \wedge q_0 \neq q' \wedge q_0 \notin F] \Rightarrow \lambda \notin L(M)$$

   b) Puede diseñarse un algoritmo que toma como entrada dos autómatas finitos (se supone que de cualquier tipo) y devuelve true si los autómatas son equivalentes o false en caso contrario.

   c) Las dos opciones anteriores son falsas.

6. Señala la opción falsa. Sea el autómata que se muestra a continuación:



   a) Acepta el lenguaje de cadenas que pertenecen al lenguaje universal $\{a, b, c, d\}^*$ y acaban con la subcadena cd.

   b) El autómata es un AFD mínimo, aunque no completo.

   c) El autómata mínimo que acepta $L(M_1) \cup L(M_2)$ no tiene más de 10 estados, siendo M_1 el autómata de esta pregunta y M_2 el autómata de la pregunta tipo test 1.

7. Señala la opción verdadera. Dado un autómata finito M

   a) Si existe un camino en el diagrama por el que se lee la cadena y acaba en estado no final entonces la cadena es rechazada.

   b) El AFD mínimo equivalente a M no puede tener más estados que M.

   c) Las dos opciones anteriores son falsas.