Universidad del Valle de Guatemala
Teoría de la Computación, CC2019
Docentes: Alan Reyes; Tomás Gálvez P.
Semestre 2, 2022

Conversión de AFN a AFD y construcción directa de AFD

Conversión de AFN a AFD

Al momento de reconocer un lenguaje descrito por una expresión regular es deseable la conversión del AFN correspondiente a un AFD debido a que la simulación de un AFD es frecuentemente más fácil de implementar y, en algunos aspectos, es más eficiente. Un AFD es un autómata cuyos estados no tienen transiciones sobre ε ni más de una transición de salida por símbolo del alfabeto. En ocasiones, una conversión de AFN a AFD puede incrementar de forma exponencial la cantidad de estados del autómata, pero la mayoría de casos mantiene más o menos la misma cantidad.

El algoritmos de simulación de un AFD es el siguiente:

```
s = s_0
c = siguiente símbolo
while ( c no sea el final de la cadena ):
    s = move(s, c)
    c = siguiente símbolo
if ( s in F ):
    return "sí"
else:
    return "no"
```

La técnica de conversión de un AFN a un AFD que usaremos se conoce como **construcción de subconjuntos** debido a que la idea principal es que cada estado del AFD corresponde a un subconjunto de estados del AFN en conversión. El estado inicial D del AFD resultante deberá ser construido con la $\varepsilon - closure$ del estado inicial del AFN. Nuestro siguiente estado en el AFD debería ser el conjunto de estados alcanzados en el AFN mediante un símbolo a del alfabeto a partir de cada uno de los estados en D. Sin embargo, recordemos que en la simulación de un AFN cada uno de esos estados alcanzados, al ser visitado, dispara automáticamente sus transiciones ε, por lo que nuestro siguiente estado en el AFD será en realidad $\varepsilon - closure(move(D, a))$.

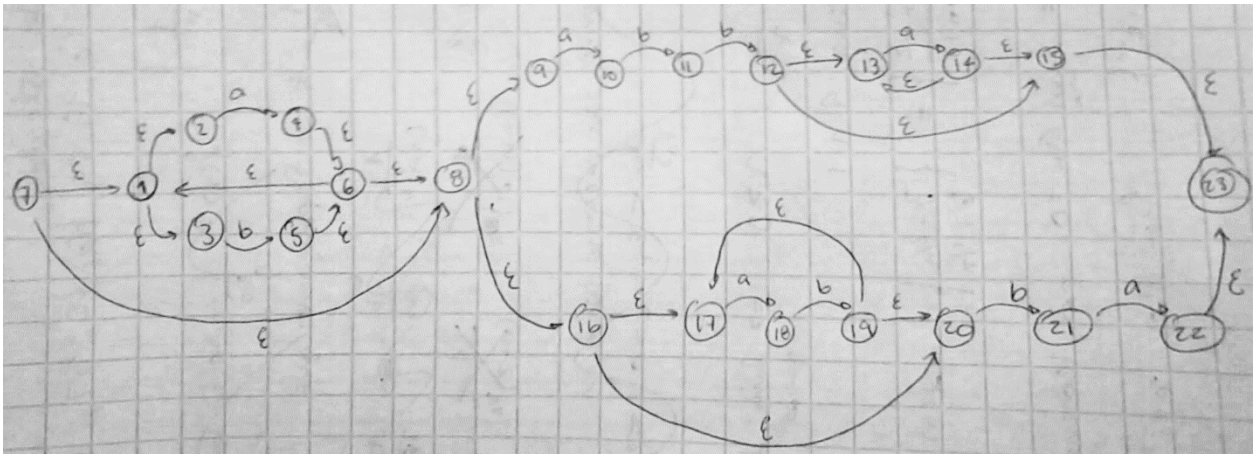
Si llamamos X a este nuevo estado del AFD, entonces en el AFD habrá una transición de D a X con el símbolo a. Esta secuencia de pasos se aplica de forma sucesiva con cada símbolo del alfabeto para obtener todas las transiciones salientes de D, y luego se repite para cada estado nuevo del AFD. Los estados de aceptación del AFD serán aquellos subconjuntos de estados obtenidos en los cuales haya al menos un estado de aceptación del AFN. La construcción de los estados del AFD y su correspondiente función de transición la describimos con el siguiente algoritmo:

```
Dstates = ε − closure(s0)
while ( hay un estado T sin marcar en Dstates ):
    marcar T
    for ( cada símbolo a en el alfabeto ):
        U = ε − closure(move(T, a))
        if ( U ∉ Dstates ):
            agregar U a Dstates sin marcar
        Dtran[T, a] = U
```

Obsérvese que el paso en el `for` recorre cada símbolo que pertenece al alfabeto sobre el cual está construida la expresión regular descrita por el AFN en conversión.

Los estados de un AFN serán designados como **importantes** cuando tengan al menos una transición de salida con un símbolo que no es ε. Dos conjuntos de estados de un AFN pueden ser tratados como el mismo estado resultante de la construcción de subconjuntos si ambos poseen un estado de aceptación y, además, contienen los mismos estados importantes.

Tomemos como ejemplo la siguiente expresión: $(a|b)^*(abba^*|(ab)^*ba)$. El AFN para esta expresión queda, con la construcción de Thompson, así:



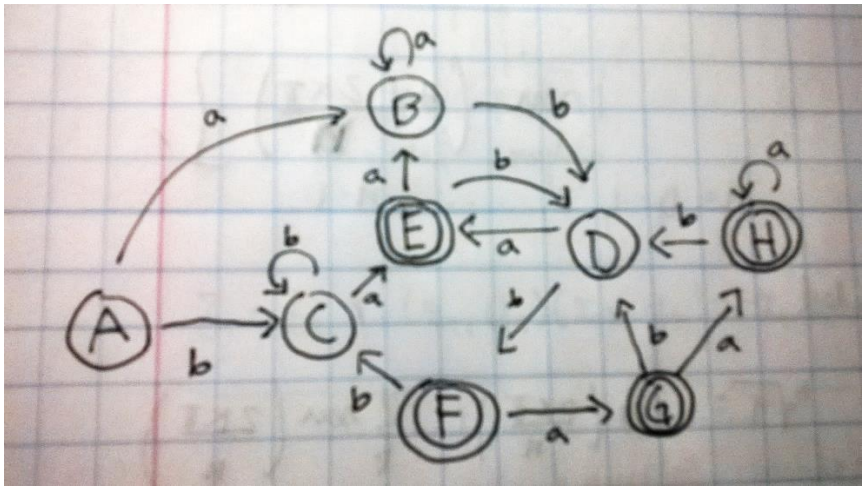
Entonces el procedimiento de construcción de subconjuntos crea el estado inicial del AFD resultante a partir de la $\varepsilon - closure$ del estado inicial, que es 7. A este estado lo llamaremos A, y es conformado por los estados {7,1,2,3,8,9,16,17,20}. Inicialmente A no está "marcado", que sólo es indicar si ya se determinaron sus transiciones de salida o no, entonces lo marcamos y comenzamos a armar sus transiciones de salida.

De acuerdo con el algoritmo, primero vemos la transición con el símbolo a, que se hace con $\varepsilon - closure(move(A, a))$. Esto resulta en el conjunto {4,6,1,2,3,8,9,10,16,17,18,20}. Como este conjunto de estados es diferente al único otro estado que conocíamos hasta el momento, lo llamaremos B y lo ingresamos sin marcar a nuestro conjunto de estados del AFD.

Nótese que debe quedar registrado en la tabla de transiciones del AFD que de A se pasa a B por medio de una transición con el símbolo a. Antes de pasar a B debemos terminar de ver las transiciones de A, por lo que calculamos $\varepsilon - closure(move(A, b))$ resultando en {5,6,1,2,3,8,9,16,17,20,21}. Nuevamente, este estado no es ni A ni B entonces lo llamamos C y lo agregamos a los estados del AFD, sin marcar. También registramos en la tabla de transiciones un salto de A a C por medio de b. Ya terminamos con A; ahora repetimos estos pasos con los estados no marcados. La tabla de transiciones, al final, quedará así:

Estado	Transición con *a*
A={7,1,**2**,3,8,**9**,16,**17**,20}	B={4,6,1,2,3,8,9,10,16,17,18,20}
B={4,6,1,**2**,3,8,**9**,10,16,**17**,18,20}	B={4,6,1,2,3,8,9,10,16,17,18,20}
C={5,6,1,**2**,3,8,**9**,16,**17**,20,**21**}	E={4,6,1,2,3,8,9,10,16,17,18,20,22,23}
D={5,6,1,**2**,3,8,**9**,11,16,**17**,19,20,**21**}	E={4,6,1,2,3,8,9,10,16,17,18,20,22,23}
E={4,6,1,**2**,3,8,**9**,10,16,**17**,18,20,22,23}	B={4,6,1,2,3,8,9,10,16,17,18,20}
F={5,6,1,**2**,3,8,**9**,16,**17**,20,**21**,12,**13**,15,23}	G={4,6,1,2,3,8,9,10,16,17,18,20,22,23,14,13,15}
G={4,6,1,**2**,3,8,**9**,10,16,**17**,18,20,22,23,14,**13**,15}	H={4,6,1,2,3,8,9,10,16,17,18,20,14,13,15,23}
H={4,6,1,**2**,3,8,**9**,10,16,**17**,18,20,14,**13**,15,23}	H={4,6,1,2,3,8,9,10,16,17,18,20,14,13,15,23}
Estado	**Transición con *b***
A={7,1,2,**3**,8,9,16,17,**20**}	C={5,6,1,2,3,8,9,16,17,20,21}
B={4,6,1,2,**3**,8,9,**10**,16,17,**18**,**20**}	D={5,6,1,2,3,8,9,11,16,17,19,20,21}
C={5,6,1,2,**3**,8,9,16,17,**20**,21}	C={5,6,1,2,3,8,9,16,17,20,21}
D={5,6,1,2,**3**,8,9,**11**,16,17,19,**20**,21}	F={5,6,1,2,3,8,9,16,17,20,21,12,13,15,23}
E={4,6,1,2,**3**,8,9,**10**,16,17,**18**,**20**,22,23}	D={5,6,1,2,3,8,9,11,16,17,19,20,21}
F={5,6,1,2,**3**,8,9,16,17,**20**,21,12,13,15,23}	C={5,6,1,2,3,8,9,16,17,20,21}
G={4,6,1,2,**3**,8,9,**10**,16,17,**18**,**20**,22,23,14,13,15}	D={5,6,1,2,3,8,9,11,16,17,19,20,21}
H={4,6,1,2,**3**,8,9,**10**,16,17,**18**,**20**,14,13,15,23}	D={5,6,1,2,3,8,9,11,16,17,19,20,21}

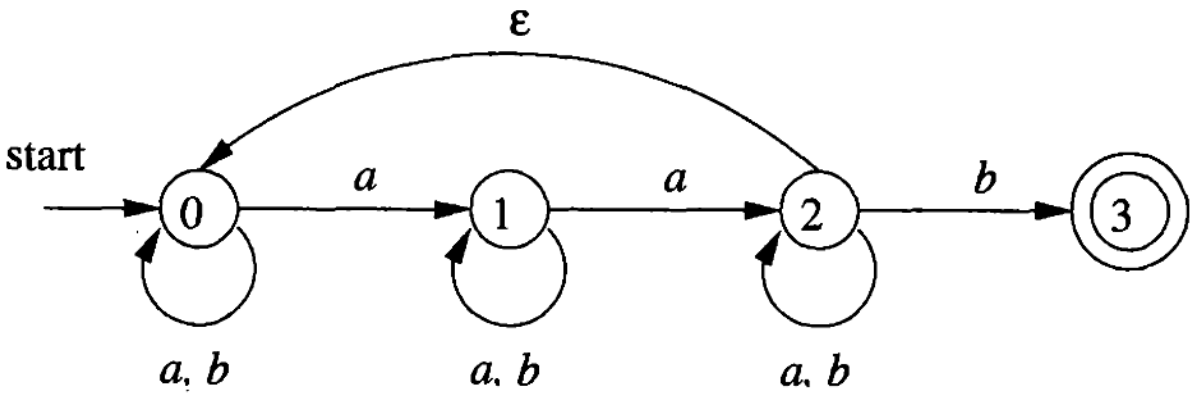
El AFD resultante quedaría así:



Notemos que como los estados G y H contienen ambos un estado de aceptación del AFN y además tienen los mismos estados importantes, podríamos unirlos en un sólo estado en el AFD resultante.

Universidad del Valle de Guatemala
Teoría de la Computación, CC2019
Docentes: Alan Reyes; Tomás Gálvez P.
Semestre 2, 2022

Ejercicio de práctica A: obtenga la expresión regular correspondiente al siguiente AFN usando el algoritmo visto en el tema anterior. Luego, conviértalo a AFD usando construcción de subconjuntos y simúlelo para determinar si acepta la cadena $aabb$.



Construcción directa de AFD

Dado que un estado de aceptación en un AFN construido con Thompson no posee transiciones salientes, determinamos que el símbolo especial # se pone al final de lo que ahora llamaremos una **expresión regular extendida**. Con este símbolo sabemos que hemos alcanzado el final de la cadena a reconocer, y por lo tanto cualquier estado en el AFN que tenga una transición saliente con el símbolo # es un estado de aceptación.

Nótese que los estados importantes, por su definición, corresponden a las subexpresiones fundamentales de la construcción de Thompson donde la transición ocurre con un símbolo del alfabeto. Recordando que estas subexpresiones corresponden a cada ocurrencia de un símbolo del alfabeto en la expresión regular, podemos numerar dichas ocurrencias según su posición en la *regex*. El propósito es etiquetar, con dicha posición, a los estados del AFN cuya transición de salida corresponde a esa ocurrencia del símbolo en la expresión. Esta numeración nos sirve para computar ciertas operaciones, definidas sobre subexpresiones, que nos servirán para construir un AFD a partir de una expresión regular sin necesidad de pasar por la construcción de un AFN.

Definimos las operaciones sobre una subexpresión n cualquiera:

- $\boldsymbol{nullable}(\boldsymbol{n})$: devuelve *true* o *false* dependiendo de si n puede o no producir ε (es decir, tiene a ε como un miembro del lenguaje que describe).

- $\boldsymbol{firstpos}(\boldsymbol{n})$**:** define el conjunto de posiciones donde se ubican los símbolos con los que pueden empezar los *strings* descritos por n. Supongamos que $n = (a|b)^*a$. Numeramos con 1, 2 y 3 cada letra en n. Entonces $firstpos(n) = \{1,2,3\}$ porque hay *strings* descritos por n que pueden empezar con a por culpa de $(a|b)^*$ (posición 1); también los hay que pueden empezar con b por lo mismo (posición 2); y, debido a que $(a|b)$ puede ocurrir 0 veces por la operación *, hay un *string* descrito por n que empieza y termina con la última a (posición 3).

- $\boldsymbol{lastpos(n)}$: como se podrá intuir, define el conjunto de posiciones donde se ubican los símbolos con los que pueden terminar los *strings* descritos por n. Volviendo a tomar $n = (a|b)^*a$, $lastpos(n) = \{3\}$ porque todos los *strings* descritos por n deben terminar con la segunda a.

- $\boldsymbol{followpos(p)}$**:** considérese la posición p. Para cada *string* descrito por la expresión regular n, la cual contiene al símbolo x ubicado en la posición p, obtenemos la posición del símbolo que en ese *string* sigue a x, y la designamos q. $followpos(p)$ describe el conjunto de todas las posiciones q. Siguiendo el ejemplo de $n = (a|b)^*a$ podemos considerar $followpos(2)$. Cualquier *string* producido por n que contenga una b, por la cerradura Kleene, puede ser seguido por b o por a (posiciones 2 y 1, respectivamente). Además, puede ser que b sea el penúltimo símbolo del *string*, entonces como todos los *strings* que n produce terminan con a (posición 3), $followpos(2) = \{1,2,3\}$.

Para facilitar el procesamiento (o pre-procesamiento, dependiendo de nuestro acercamiento) de expresiones regulares es común emplear árboles sintácticos que las describan (la facilidad es la misma que se presenta al *parsear* expresiones aritméticas usando notación *postfix*). En un árbol sintáctico para cualquier expresión regular los operadores corresponden a nodos internos mientras que los operandos (ε y símbolos del alfabeto) corresponden a las hojas. Cada nodo de este árbol sintáctico describe una subexpresión.

Las siguientes reglas describen cómo computar $nullable$ y $firstpos$ para las diferentes posibles subexpresiones en un árbol sintáctico. Las reglas de computación de $lastpos$ son las mismas que las de $firstpos$, pero invirtiendo el orden de los operandos cuando aplique.

Ejercicio de práctica B: complete la tabla deduciendo la computación de cada operación donde sea necesario.

Nodo n	$nullable(n)$	$firstpos(n)$	$lastpos(n)$
Hoja ε	True	$\emptyset$	
Hoja con posición i			
Un or: $c_1\|c_2$		$firstpos(c_1)$ $\cup\, firstpos(c_2)$	
Una concatenación: c_1c_2	$nullable(c_1)$ **and** $nullable(c_2)$		
Una Kleene: c_1^*	True	$firstpos(c_1)$	

Para computar $followpos$ debemos pensar en cómo un símbolo puede suceder a otro en un *string*. Es obvio que esto sucede únicamente en las concatenaciones, y las concatenaciones pueden suceder de forma explícita por medio de un operador de concatenación o de forma implícita por medio de una cerradura. Cuando tenemos un nodo de concatenación, todos los posibles símbolos con los que puede empezar la subexpresión del lado derecho podrían suceder a cada uno de los símbolos con los que puede terminar la expresión del lado izquierdo. Es decir que si c_1 es la expresión del lado izquierdo y c_2 la del lado derecho, $followpos(i)$ incluye a $firstpos(c_2)$ para cada posición i en $lastpos(c_1)$.

Cuando nuestro nodo es una cerradura debemos pensar en que lo que estamos haciendo es concatenar a una subexpresión consigo misma. Por lo tanto, si nuestra subexpresión es n, para todas las posiciones i en $lastpos(n)$, $followpos(i)$ incluye a $firstpos(n)$.

Estas funciones sirven para la construcción de un AFD a partir de una expresión regular sin necesidad de pasar por un AFN ni la construcción de Thompson.

El algoritmo que usa estas funciones se asemeja al de transformación de un AFN en un AFD. La función central del algoritmo es $followpos$, análoga a $move$. Las demás funciones nos sirven exclusivamente para computar $followpos$, a excepción de $firstpos$ que la usamos al principio del algoritmo para obtener nuestro estado inicial. Los estados del AFD son conformados por conjuntos de posiciones en lugar de conjuntos de estados de un AFN.

Lo primero que hacemos es construir el árbol sintáctico para nuestra expresión regular y computar para cada nodo las operaciones $nullable$, $firstpos$, $lastpos$ y $followpos$. Ya que tenemos el resultado procedemos a ejecutar el siguiente algoritmo:

```
Dstates = firstpos(n0) donde n0 es la raíz del árbol T para la expresión extendida

while ( hay un estado S sin marcar en Dstates ):
    marcar S
    for ( cada símbolo a en la regex de entrada ):
        U = ⋃(pos correspondiente a 'a')∈S followpos(p)
        if ( U ∉ Dstates ):
            agregar U a Dstates sin marcar
        Dtran[S, a] = U
```

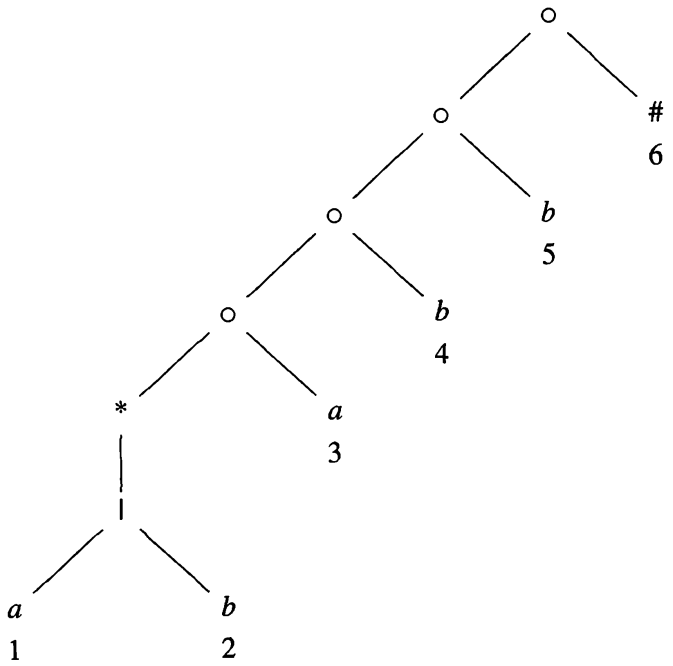
Obsérvese cómo el ciclo `for` de este algoritmo también considera cada símbolo en la expresión, pero su primer paso es unir todos los $followpos$ de las posiciones en la *regex* donde este símbolo ocurre. Esto permite deducir a qué estados llegaríamos con el símbolo en cuestión, ya que se toma en cuenta cualquier ocurrencia del símbolo en la *regex*. Los estados, igual que en la construcción de subconjuntos, serán marcados cuando todas sus transiciones de salida hayan sido encontradas.

Ejercicio de práctica C: construya el árbol sintáctico para la expresión regular del ejercicio de práctica A. Recuerde: extienda la expresión con # al final, pásela a notación *postfix*; y arme el árbol leyendo la expresión en *postfix* de derecha a izquierda, con operadores en nodos internos y operandos en las hojas.

Universidad del Valle de Guatemala
Teoría de la Computación, CC2019
Docentes: Alan Reyes; Tomás Gálvez P.
Semestre 2, 2022

Ejemplo: construcción directa de un AFD a partir de la expresión regular $(a|b)^*abb\#$

El árbol sintáctico es:

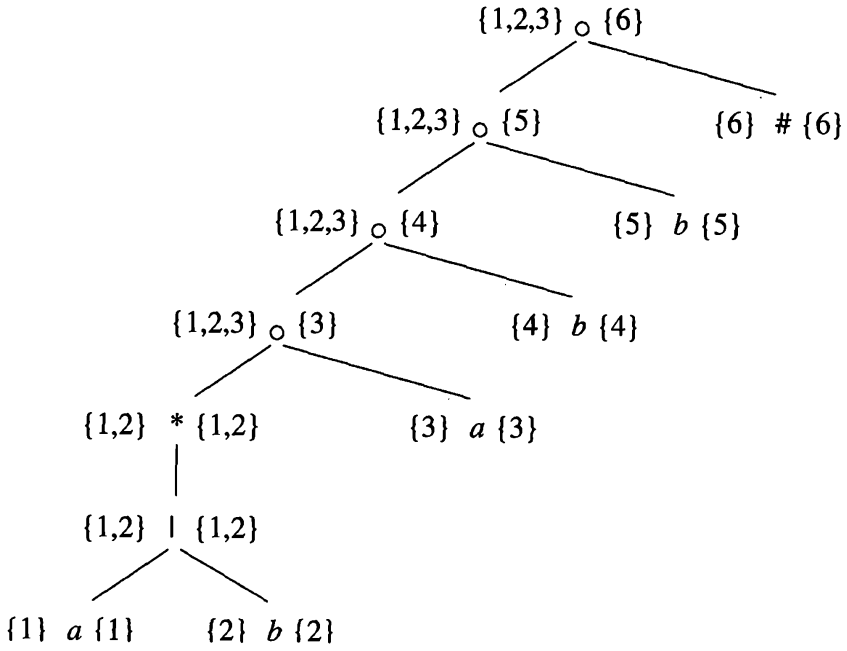


¿Cuáles de los nodos en el árbol son $nullable$? Sabemos que ninguna de las hojas del árbol lo será ya que ninguna es ε. Si vemos todos los nodos internos, los de concatenación tienen siempre a su derecha una hoja y a la izquierda otro nodo de concatenación (a excepción del que está hasta abajo). El nodo de disyunción (el *or*) tiene a ambos lados una hoja. Ninguno de estos nodos, pues, puede producir ε; y por lo tanto ninguno es $nullable$. Sólo queda el nodo de operación Kleene, el cual sabemos que por definición sí es $nullable$.

Calculemos ahora $firstpos$ y $lastpos$. De acuerdo con la tablita, para las hojas el resultado serán ellas mismas. Yendo de abajo para arriba, el $firstpos$ del *or* será la unión entre el $firstpos$ de las hojas con posición 1 y 2... o, sea, $\{1,2\}$. Para $lastpos$ el caso es el mismo porque la unión es conmutativa. El nodo Kleene tiene como $firstpos$ y $lastpos$ a lo que tenga su único hijo en sus $firstpos$ y $lastpos$ (recordemos que Kleene es una operación unaria).

Ahora nos vamos con las concatenaciones. Como el hijo izquierdo de la primera concatenación (de abajo para arriba) es $nullable$, $firstpos$ de ella será la unión del $firstpos$ de sus dos hijos. Entonces el resultado será $\{1,2,3\}$. Para $lastpos$ hacemos lo mismo, pero invertimos los hijos, entonces como el hijo derecho (posición 3) no es $nullable$ el $lastpos$ será sólo $\{3\}$. De allí, para cada concatenación hasta la raíz ninguno de los hijos es $nullable$, entonces el $firstpos$ será igual al $firstpos$ del hijo izquierdo; y el $lastpos$ igual al $lastpos$ del hijo derecho. Por último, para $followpos$ debemos únicamente considerar las concatenaciones y el Kleene (quedamos en que un símbolo sólo puede seguir a otro gracias a una concatenación, y que las concatenaciones ocurren explícitamente o implícitas en las cerraduras). Empezando por el Kleene debemos observar las posiciones en su $lastpos$: $\{1, 2\}$. Entonces $followpos$ de 1 y de 2 debe incluir todo lo que esté en $firstpos$ del Kleene (que es $\{1,2\}$).

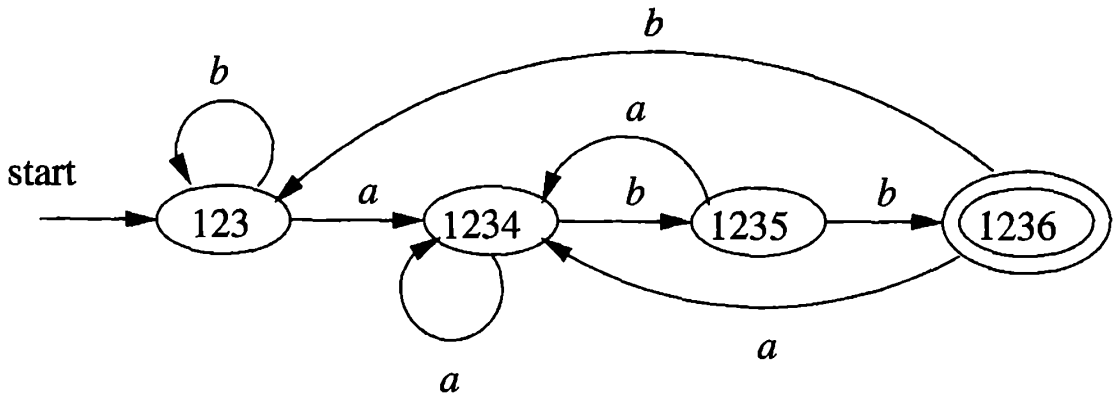
Luego, para las concatenaciones, el $followpos$ de cada posición en el $lastpos$ de su lado izquierdo debe incluir todo el $firstpos$ de su lado derecho. Entonces para la primera concatenación el $lastpos$ del lado izquierdo (del nodo Kleene) es $\{1, 2\}$, y el $firstpos$ del lado derecho es $\{3\}$. Por lo tanto, el $followpos$ de 1 y 2 debe incluir además a $\{3\}$. Las siguientes concatenaciones son más sencillas. El $lastpos$ del lado izquierdo de la segunda concatenación es sólo $\{3\}$, entonces $followpos$ de 3 incluye a todo lo que esté en el $firstpos$ del lado derecho… que sólo es $\{4\}$. El caso se repite para las penúltima y última concatenaciones. El resultado se expresa a continuación. El árbol presenta, para cada nodo, su $firstpos$ a su par izquierda y su $lastpos$ a su par derecha. La tablita muestra el $followpos$ para cada posición.



Nodo n	$followpos(n)$
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	∅

Ahora aplicamos el algoritmo. Nuestro estado inicial A será $firstpos$ de la raíz: $\{1, 2, 3\}$. Marcamos A y calculamos el nuevo estado uniendo los $followpos$ de las posiciones en A que corresponden al símbolo a (1 y 3). Obtenemos el estado $B = \{1, 2, 3, 4\}$. Luego unimos los $followpos$ de las posiciones que corresponden a b, que sólo es 2, y cuyo resultado es el mismo estado A (porque contiene las mismas posiciones). Entonces en nuestra tabla de transiciones A lleva a B por medio de a y a sí mismo por medio de b. Ahora pasamos al siguiente estado sin marcar (que sería el estado B), y repetimos.

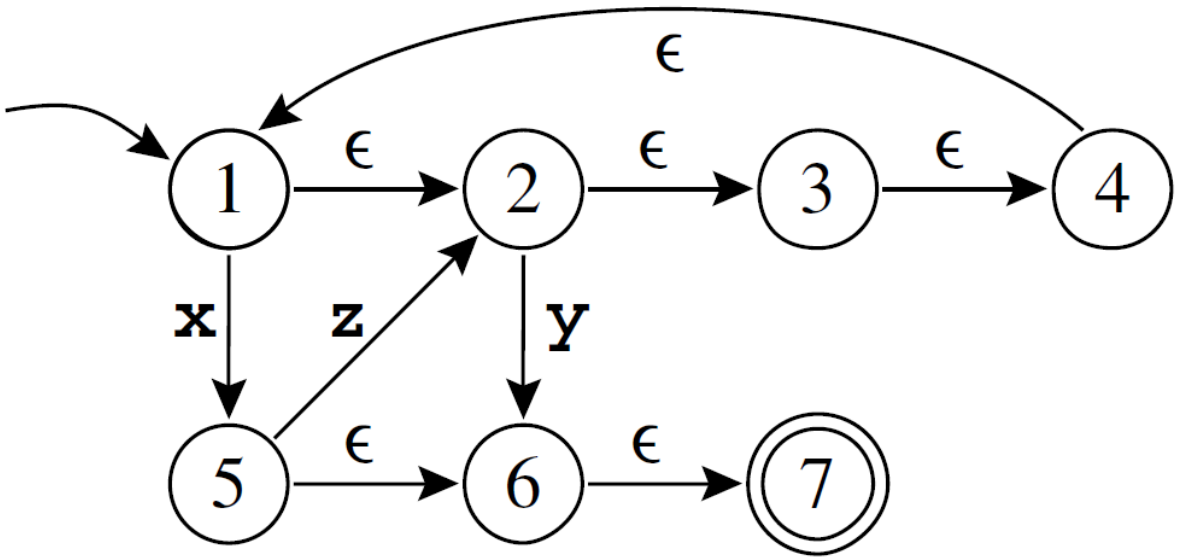
Obtenemos el mismo estado B al transicionar con a, y el estado nuevo $C = \{1, 2, 3, 5\}$ al transicionar con b. Marcamos B y proseguimos con C. Obtenemos el estado B al transicionar con a, y el estado nuevo $D = \{1, 2, 3, 6\}$ al transicionar con b. Marcamos C, tomamos D, y finalmente procesamos sus transiciones con a y con b, las cuales producen los estados B y A respectivamente. El AFD queda así:



¿Por qué es D un estado de aceptación? Porque incluye la posición 6 que corresponde al símbolo #. Como este símbolo está al final de la expresión regular extendida, su presencia en un estado nos indica que para llegar a ese estado debimos haber leído exitosamente el final de la *regex*.

Ejercicio de práctica D: construya el AFD (directamente) para la expresión regular $aa^*|bb^*$.

Ejercicio de práctica E: produzca la expresión regular para el siguiente AFN. Luego, convierta el AFN a AFD. Luego, construya el AFD directamente a partir de la expresión regular obtenida. Comente sobre la comparación entre los AFD's producidos a partir de la conversión y la construcción directa.



Fuentes:

- Aho, A. V., Lam, M. S., Ravi, S., & Ullman, J. D. (2007). *Compilers: Principles, Techniques and Tools.* Pearson.
- Appel, A. (2004). *Modern Compiler Implementation in (Java|C|ML).* Cambridge.
- Stanford University. (2012). *Video Lectures*. Retrieved from Compilers: https://class.coursera.org/compilers-selfservice/lecture/index