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

Expresiones regulares y AFN's

Al compilar código de un programa, el análisis léxico es la fase donde se reconoce el vocabulario de un lenguaje de programación. Tomemos por ejemplo el identificador de una variable. Les podemos dar nombres muy diversos a nuestras variables bajo ciertas limitaciones impuestas por cada lenguaje, aunque comúnmente tienen condiciones como "debe comenzar con una letra" o "no puede contener símbolos especiales". ¿Cómo diferencia un compilador un identificador de otras palabras reservadas? Por medio de instrucciones para reconocer un tipo específico de patrones, llamados **expresiones regulares**.

Antes de tratar con expresiones regulares debemos tener claros varios conceptos:

- **Alfabeto**: conjunto finito de símbolos.
- **Cadena/palabra/oración** en el alfabeto: una secuencia finita de símbolos que pertenecen al alfabeto.
    - **Prefijo**: dada una cadena s, cualquier cadena obtenida al remover uno o más elementos del final de s es un prefijo de esa cadena.
    - **Sufijo**: lo mismo que un prefijo, pero los símbolos se remueven al inicio de s.
    - **Subcadena**: cualquier cadena obtenida de eliminar cualquier prefijo y/o sufijo de s.
    - **Subsecuencia**: cualquier cadena obtenida al remover elementos no necesariamente consecutivos de s.
- **Lenguaje**: cualquier subconjunto del conjunto de todas las posibles cadenas en el alfabeto. Para un alfabeto denotado con el símbolo Σ, el lenguaje que comprende absolutamente todas las posibles cadenas formadas por símbolos del alfabeto se representa como Σ^*.

Nótese que, debido a que el **símbolo nulo $\boldsymbol{\varepsilon}$** (que representa una cadena vacía) forma parte de cualquier cadena, una cadena está dentro del conjunto de sus sufijos, prefijos y subcadenas. Hablamos de un(a) prefijo/sufijo/subcadena propia de una cadena s cuando no es ni ε ni la misma s.

Operaciones definidas sobre lenguajes:

- **Concatenación**: dada la concatenación de cadenas como la conocemos, la concatenación de dos lenguajes es el conjunto de cadenas obtenidas al concatenar cada cadena de un lenguaje con cada cadena del otro lenguaje.
- **Unión**: unir dos lenguajes simplemente produce un conjunto de cadenas que incluyen tanto las cadenas del primer lenguaje como las del segundo.
- **Cerradura**: puede ser **Kleene** o **positiva.**
    - Kleene: conjunto de cadenas obtenido al concatenar un lenguaje consigo mismo cero o más veces.
    - Positiva: conjunto de cadenas obtenido al concatenar un lenguaje consigo mismo una o más veces.

Las expresiones regulares describen lenguajes (llamados **lenguajes regulares**) porque producen cadenas de caracteres con características específicas usando un alfabeto dado. Dadas dos expresiones regulares a y b, los operadores son a^* y a^+ para cerradura Kleene y positiva respectivamente, yuxtaposición (ab) o '$a \cdot b$' para concatenación; y $a|b$ para unión/alternación.

También se incluyen la abreviatura '?' que expresa "cero o una instancia de"; y, en ciertas fuentes, las clases de símbolos con notación $[a_1a_2a_3 \ldots a_n]$ (o $[a_1 - a_z]$ si son elementos en una secuencia predeterminada) que expresa la unión de cada símbolo individual dentro de los corchetes. El orden de precedencia de estas operaciones es el mismo que el orden en el que se han enumerado, con asociatividad a la izquierda y el uso de paréntesis convencional.

Hay dos reglas fundamentales para formar expresiones regulares:

1. ε es una expresión regular que describe al lenguaje que sólo posee a la cadena nula (sí, ε identifica tanto a la cadena nula como al lenguaje que solo tiene a la cadena nula).
2. Para cualquier símbolo individual en un alfabeto determinado existe una expresión regular. Ésta describe al lenguaje que posee una única cadena: la conformada por el símbolo en cuestión.

A partir de estas reglas podemos aplicar las operaciones descritas para obtener cualquier expresión regular sobre un alfabeto dado. Las operaciones, además, cumplen con las siguientes reglas algebraicas:

Regla	Descripción
$r\|s = s\|r$	La unión es conmutativa
$r\|(s\|t) = (r\|s)\|t$	La unión es asociativa
$r(st) = (rs)t$	La concatenación es asociativa
$r(s\|t) = rs\|rt; (s\|t)r = sr\|tr$	La concatenación se distribuye sobre la unión
$\varepsilon r = r\varepsilon = r$	ε es el elemento neutro de la concatenación
$r^* = (r\|\varepsilon)^*$	ε está garantizada como resultado de una cerradura
$r^{**} = r^*$	La cerradura Kleene es idempotente

Los siguientes son ejemplos de expresiones regulares:

- $(0|1)^*0$
- $b^*(abb^*)(a|\varepsilon)$
- $(a|b)^*aa(a|b)^*$

Ejercicio de práctica A: describa los lenguajes representados por cada una de las expresiones regulares anteriores.

Las expresiones regulares se definen sobre un alfabeto, pero podemos hablar de **definiciones regulares**:

$$d_1 \rightarrow r_1$$

...

$$d_n \rightarrow r_n$$

Cada símbolo d_i no pertenece al alfabeto y la expresión regular r_i que lo define está conformada por símbolos del alfabeto y/o cualquier d_j, $1 \leq j < i$. Las definiciones regulares son más que todo una conveniencia de notación que nos permite hacer cosas como presentar una expresión regular que reconozca dígitos, nombrarla y usarla después como símbolo en una expresión regular que reconozca identificadores o números. El siguiente ejemplo muestra una definición regular que describe números, fracciones y potencias:

$$
\begin{aligned}
digit &\rightarrow 0 \mid 1 \mid \cdots \mid 9 \\
digits &\rightarrow digit\ digit^* \\
optionalFraction &\rightarrow .\ digits \mid \epsilon \\
optionalExponent &\rightarrow (\mathtt{E} (+ \mid - \mid \epsilon)\ digits) \mid \epsilon \\
number &\rightarrow digits\ optionalFraction\ optionalExponent
\end{aligned}
$$

Ejercicio de práctica B: indique cuáles de los siguientes números son descritos por la definición regular anterior. Para aquellos que lo sean, desglose sus símbolos e identifique cómo se manifiestan los componentes de la definición.

1. 1.618
2. 3^(-10)
3. 5.3997E+08

Ejercicio de práctica C: elabore una definición regular para los identificadores del lenguaje de programación C. Para referencia, vea este enlace.

En la definición de lenguajes de programación se les llama **lexemas** a las cadenas de símbolos que conforman el código fuente y que son reconocidas con apoyo en expresiones regulares. Todas las expresiones regulares de un lenguaje conforman su **especificación léxica**, y la tarea de un analizador léxico es identificar las expresiones regulares que describen cada lexema en el código fuente**.**

Las especificaciones léxicas suelen listar expresiones regulares en orden de precedencia, y se busca siempre reconocer el lexema más largo posible. El objetivo es hallar coincidencias lo antes posible a lo largo de una lectura, pero distinguiendo cuando un hallazgo forma parte de un reconocimiento más largo para no reportar reconocimientos incorrectos. Por ejemplo, un identificador como *ifyouknowwhatimean* podría ser problema debido a la detección temprana de la palabra reservada *if*. La cadena más larga con la que se pueda coincidir alguna expresión regular recibe en otras fuentes nombres como ***maximal lexeme*** o ***maximal munch***.

En el libro del dragón se describe una técnica de lectura que involucra dos punteros. Uno se coloca al inicio del lexema (llamémosle **puntero de lexema**) a ser leído, y que sólo se actualiza luego de un reconocimiento exitoso. El otro puntero se llama ***forward pointer*** y es el que se encarga de ir revisando símbolo por símbolo hasta lograr un reconocimiento. En ocasiones el *forward pointer* permite cumplir la regla de la cadena más larga realizando ***lookahead***, lo cual significa que lee una cantidad determinada de símbolos más allá de aquel que concluye la coincidencia con algún patrón.

El tema de los *lookahead* es más profundo y comúnmente no se adhiere únicamente a la regla del lexema más largo. Por ejemplo, en el generador de analizadores léxicos Lex hay un operador de *lookahead* que permite buscar un patrón más allá de un lexema reconocido para determinar cómo se interpreta dicho lexema. Para clarificar, vamos el siguiente ejemplo del lenguaje Fortran:

Do 5 i = 1,25 (1)

Do 5 i = 1.25 (2)

En Fortran, *Do* es un ciclo que recibe:

- Una etiqueta o *label* (como las de *assembler*) que le indica hasta qué línea repetir, a partir de la declaración del ciclo.
- Una variable de iteración.
- Un rango de iteración, denotado por dos números separados por una coma.

En Fortran, además, *whitespace* es vilmente ignorado. Entonces, (1) presenta la declaración de un ciclo porque la coma en 1,25 separa los límites inferior y superior de un rango en un ciclo *Do*. Por otro lado, (2) presenta una asignación porque, tomando en cuenta que los espacios en blanco no representan nada, el punto hace que "1.25" se interprete como número con decimales, obligando al compilador a interpretar Do 5 i como Do5i, un identificador. Como el analizador léxico va leyendo caracter por caracter de izquierda a derecha, al completar la lectura del lexema *Do* no podemos saber si esto representa la palabra reservada para el ciclo o parte de un identificador. Necesitamos *lookahead*.

Como el proceso de reconocimiento de lexemas lo haremos caracter por caracter, con cada símbolo leído debemos determinar si hemos reconocido alguna expresión regular de la especificación léxica, si lo leído hasta el momento se parece más a un patrón o a otro; o si definitivamente no coincide con ninguno de los patrones especificados. Esto es expresable como una **máquina de estados**, donde cada símbolo leído nos permite cambiar entre los estados de reconocimiento.

Para facilitar la programación de este proceso de reconocimiento representaremos las expresiones regulares con entes matemáticos llamados **autómatas finitos**, los cuales se visualizan con **diagramas de transición**. Los autómatas finitos poseen:

- Un **conjunto S de estados** de los cuales uno es el **estado inicial** y un subconjunto F son los **estados de aceptación**.
- Un **alfabeto** Σ que no contiene a ε.
- Una ***mappeo* de transición $\boldsymbol{\delta}$** que relaciona parejas {*estado, símbolo de entrada*} a estados (destino): $\big(S \times (\Sigma \cup \{\varepsilon\})\big) \to S$.

En los diagramas de transición, los estados se representan con círculos (y los estados finales con dos círculos concéntricos). Usualmente, cada estado en un diagrama es etiquetado con un número para facilitar la lectura y el trabajo de reconocimiento. También tenemos **aristas** que entran y salen de los estados, representando las **transiciones** (pueden ser bucles que salen de y entran al mismo estado). Las aristas son etiquetadas con los símbolos leídos para representar un cambio de estado con cada símbolo.

Nuestros diagramas de transición permitirán visualizar autómatas finitos, que se separan en **determinísticos** (AFD) y **no-determinísticos** (AFN). La diferencia es que los no-determinísticos permiten usar ε como transición y registrar más de un estado a la vez como el actual (por medio de varias transiciones con el mismo símbolo a partir de un mismo estado); mientras que los determinísticos restringen las transiciones a una única por cada símbolo y no permiten usar ε como transición.

El recorrido sobre un diagrama de transición representa el movimiento del *forward pointer*. Nuestro *forward pointer* comienza a leer un lexema símbolo por símbolo y con cada símbolo cambiamos de estado en el diagrama de transición. Al momento de correr un análisis léxico dispondremos de un diagrama de transición por cada expresión regular definida. Hay diferentes formas de realizar el recorrido de los diagramas:

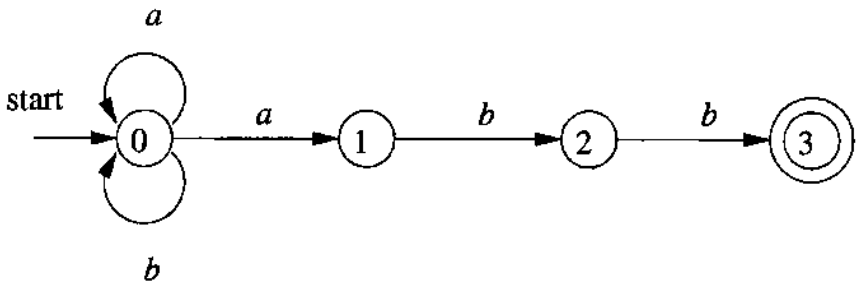
- De forma secuencial: probamos uno y, si falla, el siguiente, y si falla, el siguiente… En esta forma es necesario aplicar la regla de precedencia para las expresiones regulares porque el primer diagrama que no falle determinará el patrón reconocido.
- De forma paralela: todos los diagramas se recorren de forma simultánea y aplicamos la regla del lexema más largo para identificar el patrón reconocido en función del último diagrama sobre el que se alcanza un estado de aceptación.
- Combinar todos los diagramas: formamos un gran diagrama de transición que recibe cada símbolo y va cambiando su estado de forma correspondiente. También aplica la regla del lexema más largo.

Puesto que existe la posibilidad de reconocimientos anidados, podremos determinar que alcanzamos el estado de aceptación definitivo luego de que el *forward pointer* lea un símbolo que no coincide con ningún patrón reconocido hasta el momento.

Como mencionamos anteriormente, un AFN se puede ver como un diagrama de transición que permite varias transiciones desde un mismo estado a otros por medio del mismo símbolo, y donde ε es un símbolo válido para las transiciones. Como los diagramas de transición son, esencialmente, grafos, también podemos representar AFN's con **tablas de transición** donde las filas corresponden a estados y las columnas a los símbolos de entrada. Cada columna marca, con su símbolo de entrada, el estado destino que se alcanza a partir del estado en la fila respectiva.

Un AFN acepta un *string* o cadena de entrada cuando existe algún camino cuyas aristas (transiciones) están marcadas con los símbolos en esa cadena, que inicia en el estado inicial y alcanza algún estado de aceptación. Es importante observar que, aunque una cadena puede describir varios caminos en un AFN, la cadena es aceptada por el AFN si cualquiera de los caminos lleva a un estado de aceptación. Se dice que todas las cadenas aceptadas por un AFN conforman el lenguaje aceptado por ese AFN, que es efectivamente el mismo que su correspondiente expresión regular describe.

A continuación, se presenta un AFN que reconoce el lenguaje descrito por la expresión regular $(a|b)^*abb$ y su correspondiente tabla de transición.



Estado	a	b	ε
0	{0,1}	{0}	∅
1	∅	{2}	∅
2	∅	{3}	∅
3	∅	∅	∅

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

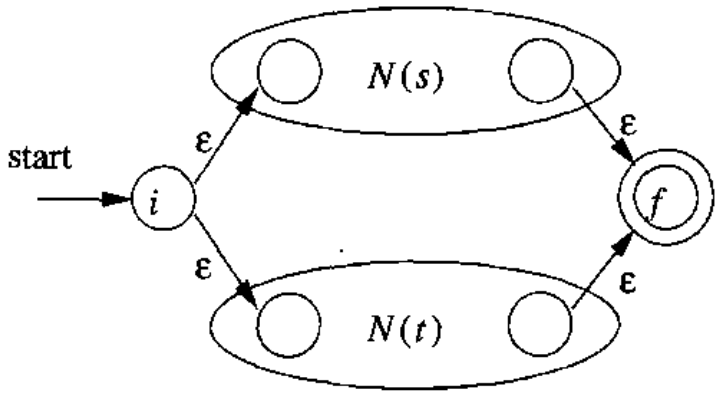
La construcción de un AFN a partir de una expresión regular la implementaremos por medio del **algoritmo de construcción de Thompson**, formalmente conocido como el algoritmo de McNaughton-Yamada-Thompson. Este algoritmo separa una expresión regular en **subexpresiones** que no contienen operadores. Cada subexpresión es transformada en un AFN y luego combinada con los AFN's de otras subexpresiones para formar un AFN más grande que a su vez se puede unir con otros AFN's. El resultado es un AFN que acepta el lenguaje descrito por la expresión regular original.

A sabiendas de que una expresión regular sin operadores es únicamente una de las expresiones regulares fundamentales, los AFN's construidos para cada subexpresión según el algoritmo de Thompson son AFN's con solo un estado inicial y un estado de aceptación alcanzable por medio de ε o un símbolo del alfabeto. Estos AFN's son construidos para cada ocurrencia de dichas subexpresiones, y los estados de estos AFN's deben ser diferentes a los de las demás subexpresiones.

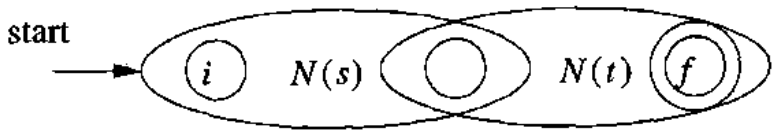
Continuando con el algoritmo de Thompson, dados AFN's $N(s)$ y $N(t)$ para las subexpresiones s y t respectivamente:

- $s|t$ es un AFN cuyo estado inicial se conecta a los estados iniciales de $N(s)$ y $N(t)$ por medio de transiciones ε. Los estados de aceptación de $N(s)$ y $N(t)$ dejan de ser de aceptación y se conectan a un nuevo estado de aceptación por medio de transiciones ε.
- st o $s \cdot t$ es un AFN cuyo estado inicial se conecta al estado inicial de $N(s)$ por medio de una transición ε (o se fusionan en un único estado). El estado de aceptación de $N(s)$ se vuelve uno con el estado inicial de $N(t)$ y el estado de aceptación de $N(t)$ se vuelve el estado de aceptación del AFN de $s \cdot t$.
- s^* es un AFN cuyo estado inicial se une directamente a un nuevo estado de aceptación por medio de una transición ε. También tira una transición ε al estado inicial de $N(s)$, y el estado de aceptación de $N(s)$ deja de ser de aceptación para tirar ahora transiciones ε al nuevo estado de aceptación y a su propio estado inicial.

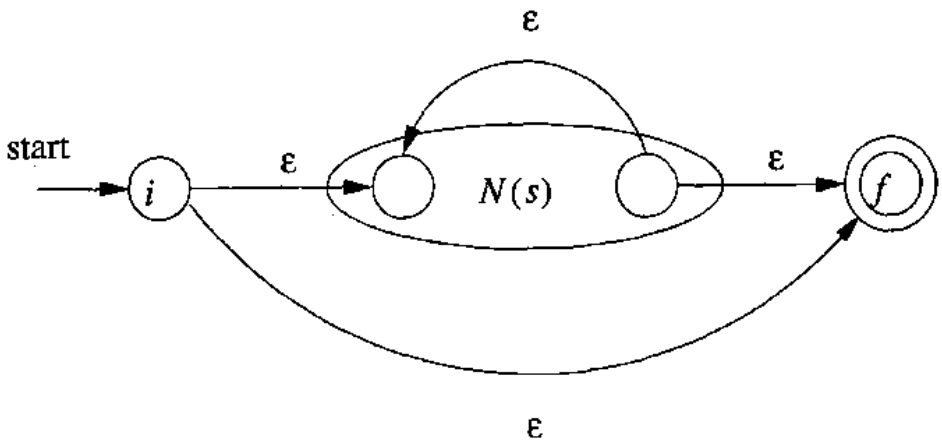
Los siguientes diagramas muestran estos AFN's respectivamente:



"La hamburguesa"



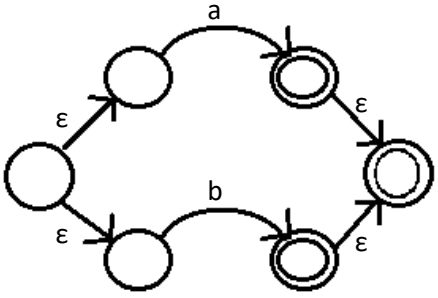
"La cadena"



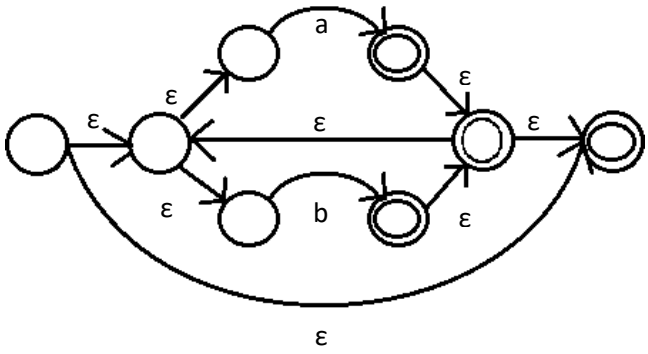
"El barquito"

Es importante notar que los AFN's que se forman con la construcción de Thompson son acumulativos e introducen uno o dos (casi siempre dos) estados nuevos que normalmente son el nuevo estado inicial y el nuevo estado final. Para ejemplo, tomemos nuevamente la expresión regular $(a|b)^*abb$ y construyamos su autómata con del método de Thompson. Consideremos el paréntesis, que consta de las expresiones regulares fundamentales a y b; y a las cuales corresponden los siguientes AFN's:

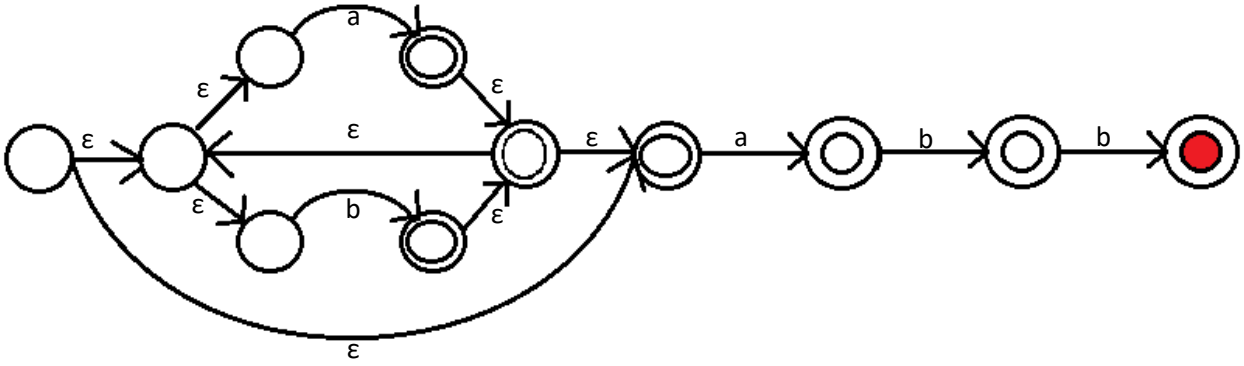
Vemos que estas expresiones regulares se involucran en una operación de unión, a lo que corresponde la siguiente expansión:



Lo siguiente es ver que esta unión está sujeta a una cerradura Kleene:



Finalmente añadimos las concatenaciones del final, y quedamos con nuestro AFN:



Nota: para que sea fácil seguir cómo se adhieren unos autómatas a otros se han dejado marcados los estados de aceptación de cada autómata a lo largo de la construcción. Sin embargo, cada uno de ellos tiene un único estado de aceptación, y en el resultado final este estado se marca con rojo.

Simulación de AFN's

Para **simular** un AFN (es decir, recorrer un AFN siguiendo los caracteres de una cadena) lo que hacemos es enumerar o listar los estados que se alcanzan en cada paso y luego ver a qué nuevos estados nos lleva cada uno de los enumerados con el siguiente símbolo de la cadena. El truco con los AFN's es que, como se permiten transiciones con ε, en cada paso podemos alcanzar varios estados sin necesidad de consumir un símbolo de la cadena siendo leída. Estas acciones son formalizadas con tres operaciones:

- Kleene de ε a partir del estado s ($\boldsymbol{\varepsilon - closure(s)}$): produce el conjunto de estados que se alcanzan a partir del estado s por medio de transiciones ε, incluyendo al mismo s.
- Kleene de ε a partir del conjunto de estados T ($\boldsymbol{\varepsilon - closure(T)}$): $\bigcup_{s \in T} \varepsilon - closure(s)$. Produce el conjunto de todos los estados alcanzables por medio de transiciones ε a partir de cada estado s en T.
- Función de transición $\boldsymbol{move(T, a)}$ de un conjunto de estados T por medio del símbolo a: produce el conjunto de todos los estados alcanzables por medio de una transición con a desde cualquiera de los estados en T.

Obsérvese que, debido a que una transición ε ocurre sin necesidad de un símbolo de entrada, se debe calcular la $\varepsilon - closure$ de cada estado que se visita en el AFN (*i.e.*, por cada estado resultante de cualquier operación) antes de reportar un resultado. El algoritmo de simulación es enunciado a continuación (donde s_0 representa el estado inicial):

```
S = ε − closure(s0)
c = siguiente símbolo
while ( c != fin de cadena):
      S = ε − closure(move(S, c))
      c = siguiente símbolo
if ( S ∩ F != ∅ ):
      return "sí"
else:
      return "no"
```

Lo que nos dice es que como primer paso debemos realizar todas las transiciones ε posibles a partir del estado inicial, para luego consumir un símbolo de la cadena. Con ello entramos a un ciclo que sólo termina cuando se alcanza el final de la cadena, y que consiste en efectuar la transición correspondiente al símbolo consumido en cada estado de los que conforman nuestro conjunto de estados actual. Sobre el conjunto de estados obtenidos hacemos nuevamente las transiciones ε. Obsérvese que, como se debe calcular la $\varepsilon - closure$ de cada estado que se visita, la operación resulta recursiva, pues para cada estado que se visita durante una $\varepsilon - closure$ se dispara otra $\varepsilon - closure$. Esta recursión sólo termina cuando ninguno de los estados alcanzados tiene transiciones salientes con el símbolo ε.

El algoritmo para calcular la $\varepsilon - closure$ de un conjunto de estados T es el siguiente:

```
Meter todos los estados de T a la pila
Inicializar resultado con T
While ( pila no vacía):
      t = pila.pop()
      for ( cada estado u alcanzado desde t por medio de ε ):
            if ( u ∉ resultado ):
                  agregar u a resultado
                  pila.push(u)
return resultado
```

Finalmente, la cadena leída es aceptada si el conjunto de estados alcanzado al final de la simulación contiene al menos un estado de aceptación.

Ejercicio de práctica D: usando el algoritmo de Thompson, construya el AFN para la expresión regular $aa^*|bb^*$ y luego simúlelo para determinar la aceptación de las siguientes cadenas:

1. aaa
2. $abab$
3. $abbb$

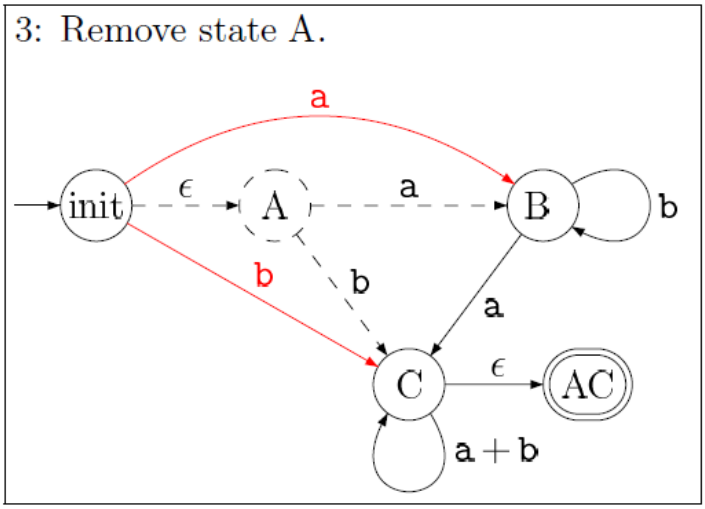
De AFN a expresión regular

La técnica que emplearemos para la construcción de una expresión regular a partir de un autómata usa **GNFA's:** ***generalized NFA's*** o AFN's generalizados. Estos autómatas permiten colocar expresiones regulares en las aristas en vez de solo símbolos, y nos requieren las siguientes condiciones:

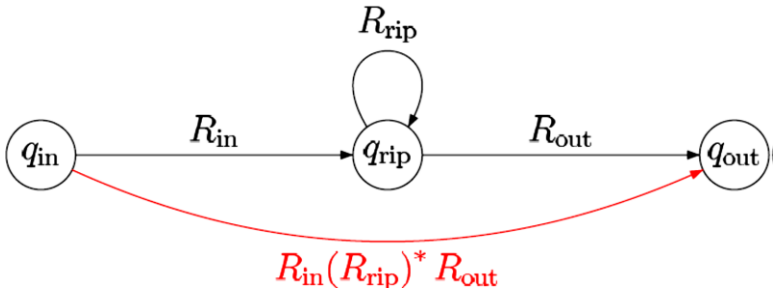
- El estado inicial no tendrá transiciones de entrada y tendrá transiciones de salida a todos los demás estados (si el AFN original no las tiene, serán transiciones sobre el conjunto vacío y, por tanto, intransitables).
- Todos los demás estados tienen transiciones a todos los estados (incluyéndose a sí mismos).
- El estado final es único y solo tiene transiciones de entrada.
- Los estados inicial y final deben ser distintos.

El procedimiento para obtener una expresión regular a partir de un AFN será primero convertirlo a un AFNG y luego ir eliminando estados de éste para obtener, al final, un AFNG con únicamente dos estados (inicial y final) donde la transición es la expresión regular que deseamos. Al eliminar un estado no podemos dejar "colgando" sus transiciones de entrada y salida, por lo que si los conjuntos I y O contienen, respectivamente, las transiciones de entrada y salida, para cada pareja del conjunto $I \times O$ debemos producir una nueva transición que conecte los estados que antes estaban conectados por medio del estado que se eliminó.

Para aclarar, veamos un ejemplo:

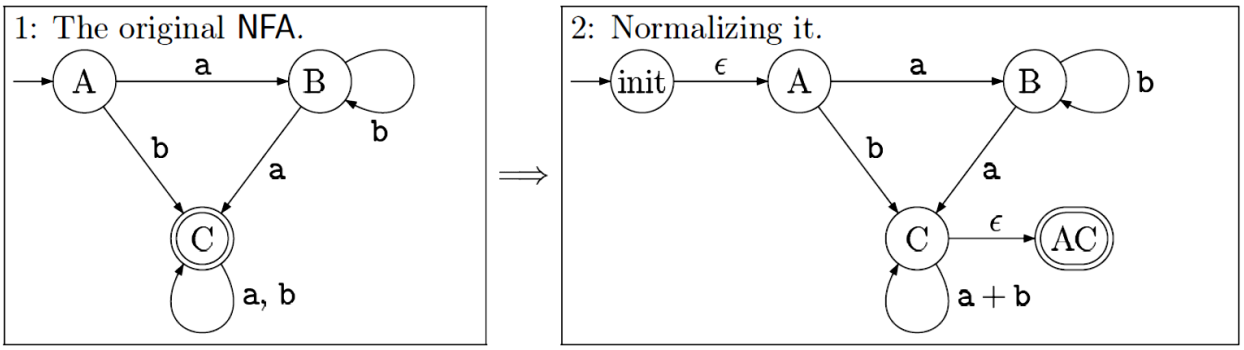


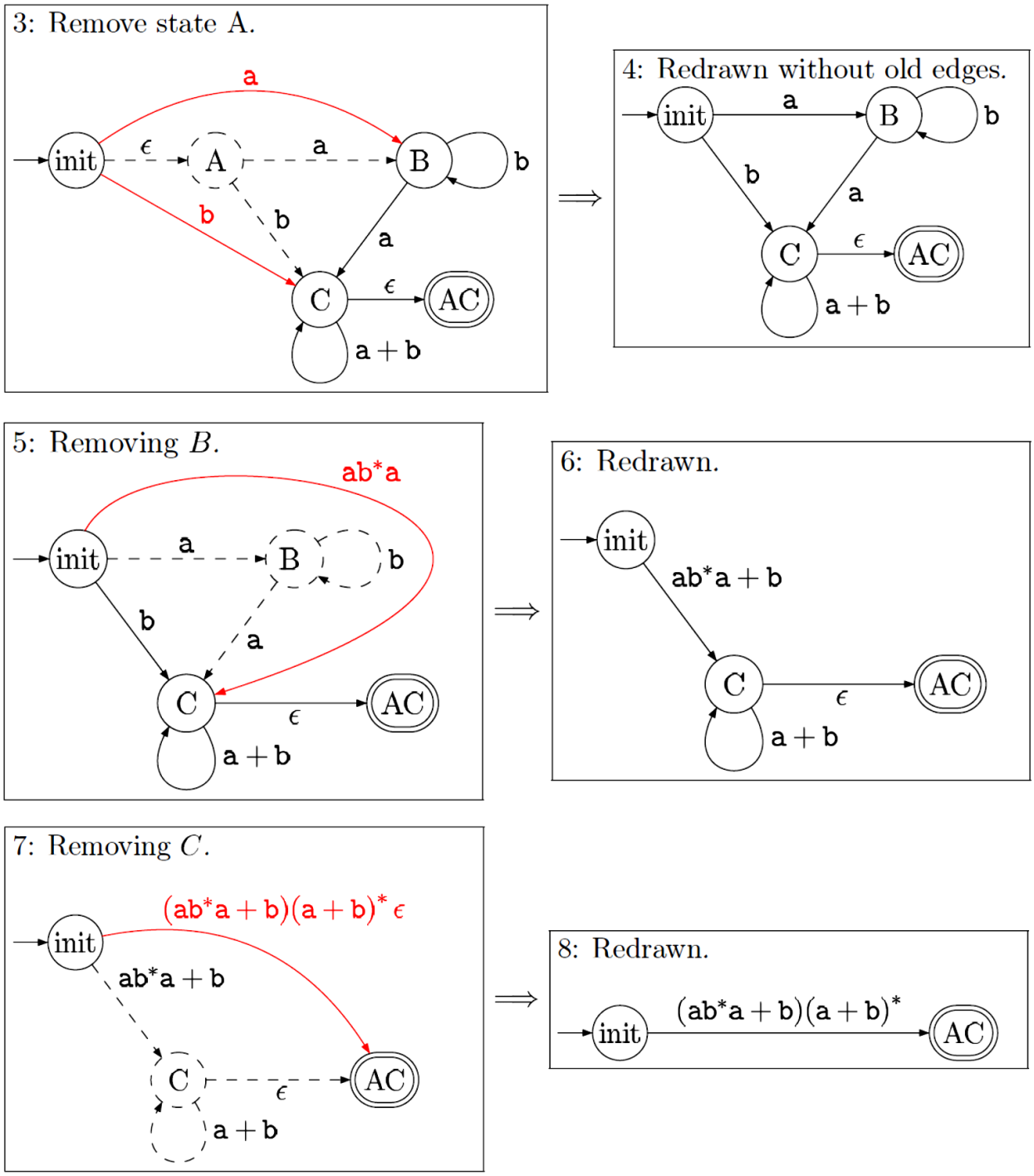
En el AFN anterior, si deseamos eliminar el estado A debemos antes conectar el estado inicial con B y con C para que no quede un autómata con estados desconectados ni transiciones colgando. ¿Cómo determinamos la expresión regular que debe ir en las aristas que conectan el estado inicial con B y con C? O, en general, ¿cómo conectamos los estados que antes usaban de puente al estado eliminado? Observemos la siguiente figura:



Lo que esto nos expresa es que para llegar del estado q_{in} al estado q_{out} debemos pasar por las expresiones regulares R_{in} y R_{out}, en ese orden. Esto es, en realidad, lo mismo que pasar por una expresión concatenada $R_{in} \cdot R_{out}$. Además, sabiendo que en un AFNG todos los estados tienen bucles, siempre debemos incluir la posibilidad de que en el camino de q_{in} a q_{out} nos echemos un colazo por dicho bucle. Claro que podría ser ignorado, pero nada quita la posibilidad de que demos una, dos o mil vueltas por ese bucle antes de alcanzar q_{out}. Por ello concatenamos, entre R_{in} y R_{out}, a la expresión R_{rip} "kleeneada", expresando que en el camino de q_{in} a q_{out} vamos a pasar por R_{in}, posiblemente pasemos por R_{rip} y finalmente pasaremos por R_{out}. La expresión regular resultante, $R_{in} \cdot (R_{rip})^* \cdot R_{out}$, se muestra en la arista roja que salta el estado q_{rip} que será eliminado.

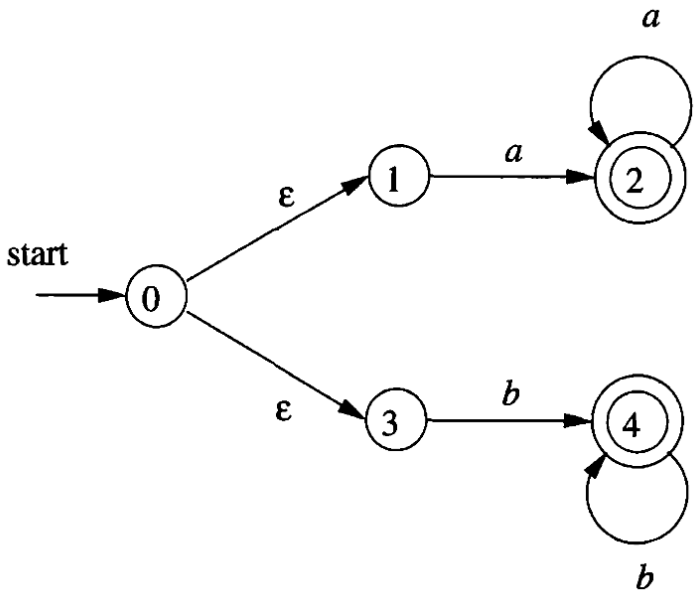
A continuación, debemos notar que si existiera otra forma de llegar de q_{in} a q_{out} podríamos parar con dos o más aristas paralelas de q_{in} a q_{out} que tienen expresiones regulares diferentes. Esto no es un problema pues simplemente indican que cualquiera de los caminos indicados es válido para llegar de q_{in} a q_{out}, y por tanto todas las aristas paralelas entre dos estados en nuestro AFNG serán reemplazadas por una única arista cuya expresión regular une a las expresiones de las aristas paralelas por medio del operador | (*or*). Ahora podemos ver el ejemplo citado anteriormente, pero completo:





La arista en el paso 8 contiene la expresión regular que describe el mismo lenguaje que el autómata del paso 1. Nótese que la normalización en el paso 2 se hace para garantizar que se cumplen las condiciones de un GNFA, aunque las transiciones ∅ no son dibujadas.

Ejercicio de práctica E: convierta el siguiente AFN a expresión regular:



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.
- Har-Peled, S., & Parthasarthy, M. (2009, march 31). *CS 373: Lecture Schedule.* Retrieved from Illinois College of Engineering Course Websites: https://courses.engr.illinois.edu/cs373/sp2009/lectures/lect_18.pdf
- Stanford University. (2012). *Video Lectures.* Retrieved from Compilers: https://class.coursera.org/compilers-selfservice/lecture/index