

Repaso

Clase 1:

Alfabetos

Cadenas de caracteres

La cadena vacía

Longitud de una cadena

Potencias de un alfabeto

Concatenación de cadenas

Potencia

- Clase 2:

LENGUAJES : Un conjunto de cadenas

Operaciones con Lenguajes:

Unión

Intersección

Concatenación

Potencia

Cerradura de Kleene

Definición de los autómatas finitos determinísticos AFD

- Los diagramas son descripciones útiles de los autómatas finitos porque nos permiten visualizar fácilmente las acciones del algoritmo.
- Sin embargo, es necesario tener una descripción más formal de un autómata finito, y por ello procederemos ahora a proporcionar una definición matemática.
- La mayor parte del tiempo, no necesitaremos una visión tan abstracta como ésta, y describiremos la mayoría de los ejemplos en términos del diagrama.
- Otras descripciones de autómatas finitos también son posibles, particularmente las tablas, y éstas serán útiles para convertir los algoritmos en código de trabajo.

- Un AFD (Autómata Finito Determinista **AFD**) M se compone de:
- Un alfabeto Σ
- Un conjunto de estados S
- Una función de transición $\delta: S_{n-1} \times \Sigma \rightarrow S_n$
- Un estado inicial s_0 que pertenece a S
- Un conjunto de estados de aceptación A que pertenecen a S

El lenguaje aceptado por M , escrito como $L(M)$, se define como el conjunto de cadenas $c_1, c_2, c_3, \dots, c_n$, con cada $c_i \in \Sigma$, tal que existan estados $s_1 = \delta(s_0, c_1)$, $s_2 = \delta(s_1, c_2)$, ..., $s_n = \delta(s_{n-1}, c_n)$ con s_n como un elemento de A

Otro ejemplo de Autómata, es el que reconoce el lenguaje de los números, ya sean estos enteros, con signo o sin él y también números en formato exponencial:

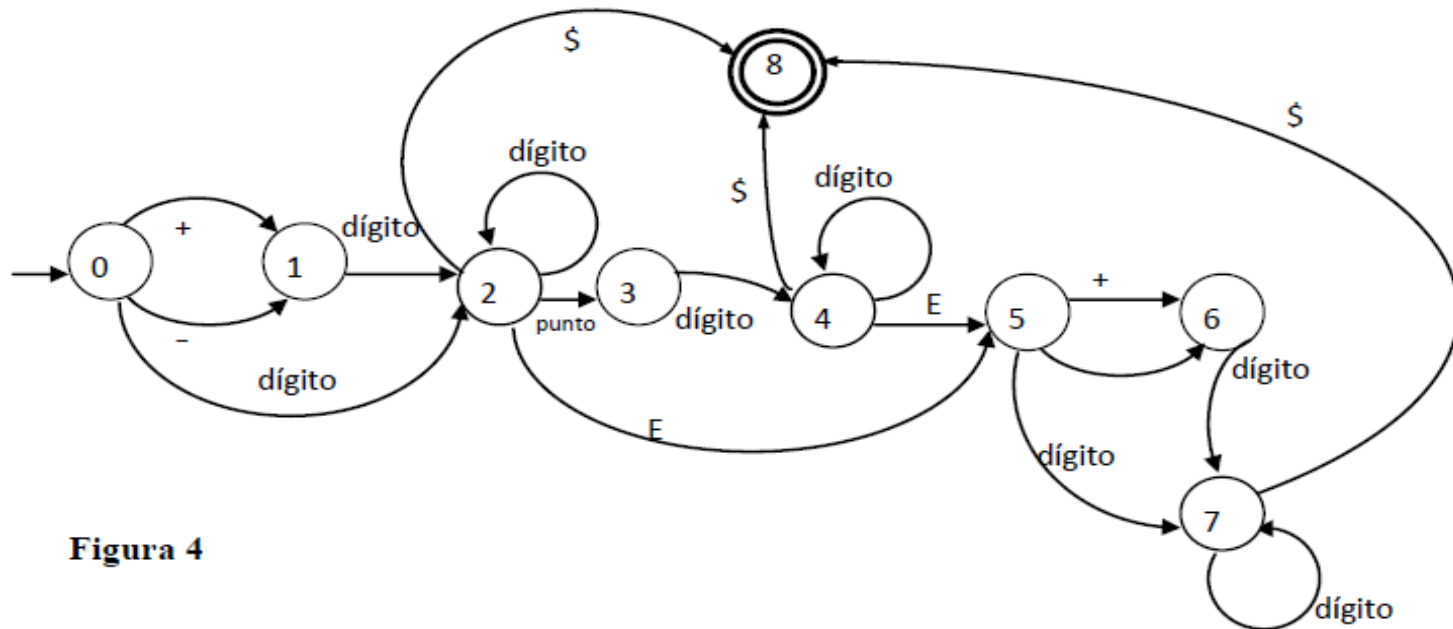


Figura 4

123, +25, -120, 3.14, 12E-3, etc.

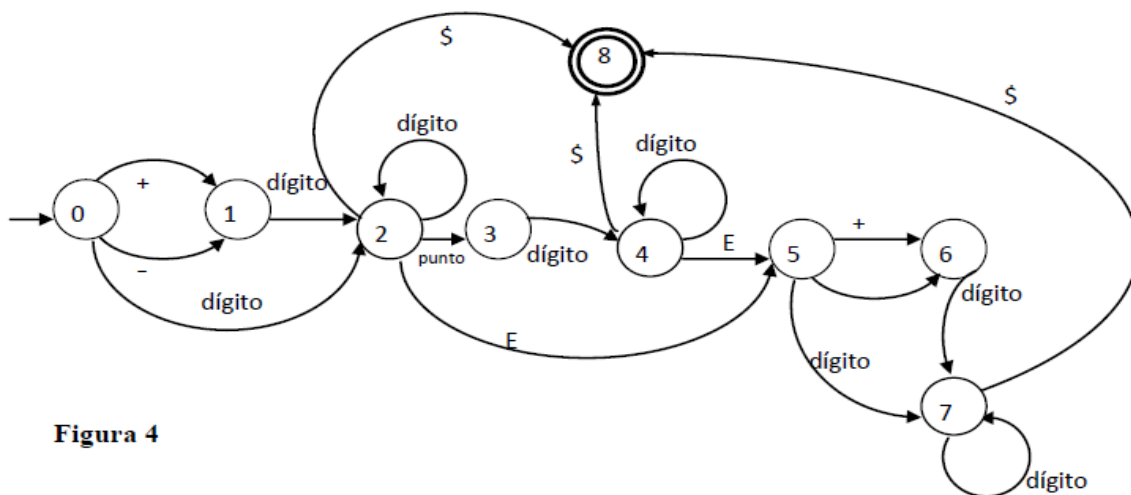


Figura 4

Estados	Alfabeto					
	dígito	+	-	punto	E	\$
0	2	1	1			
1	2					
2	2			3	5	8
3	4					
4	4				5	8
5	7	6	6			
6	7					
7	7					8
8	acepta					

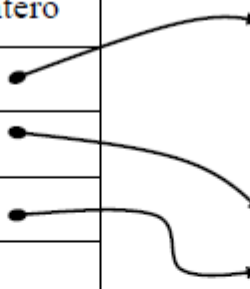
Implementación de Autómatas Finitos en Código

Estados	Alfabeto					
	dígito	+	-	punto	E	\$
0	2	1	1			
1	2					
2	2			3	5	8
3	4					
4	4				5	8
5	7	6	6			
6	7					
7	7					8
8	acepta					

```
estado =0;
while (estado != 8 && estado != error )
{
    ch = siguiente carácter de entrada;
    estado = T[estado][ch];
}
if (estado ==8) aceptar();
else error();
```

- Un analizador típico puede tener unos 100 estados y manejar unos 100 símbolos de entrada, lo que daría lugar a una matriz de transición de 10000 elementos, y a lo sumo con 100 o 200 elementos ocupados.
- Para solucionar este problema, solo se almacenan los elementos no nulos de la matriz en una tabla, y se genera otra tabla que contiene para cada estado (fila) un puntero al primer valor de ese estado, y el número de valores para ese estado:

Estado	Valores	Puntero
0	3	
1	1	
2	4	
3	1	
4	3	
5	3	
6	1	
7	2	
8		



2	dígito
1	+
1	-
2	dígito
2	dígito
3	punto
5	E
8	\$

Estados	Alfabeto					
	dígito	+	-	punto	E	\$
0	2	1	1			
1	2					
2	2			3	5	8
3	4					
4	4				5	8
5	7	6	6			
6	7					
7	7					8
8	acepta					

Estado	Valores	Puntero
0	3	
1	1	
2	4	
3	1	
4	3	
5	3	
6	1	
7	2	
8		

2	dígito
1	+
1	-
2	dígito
2	dígito
3	punto
5	E
8	\$

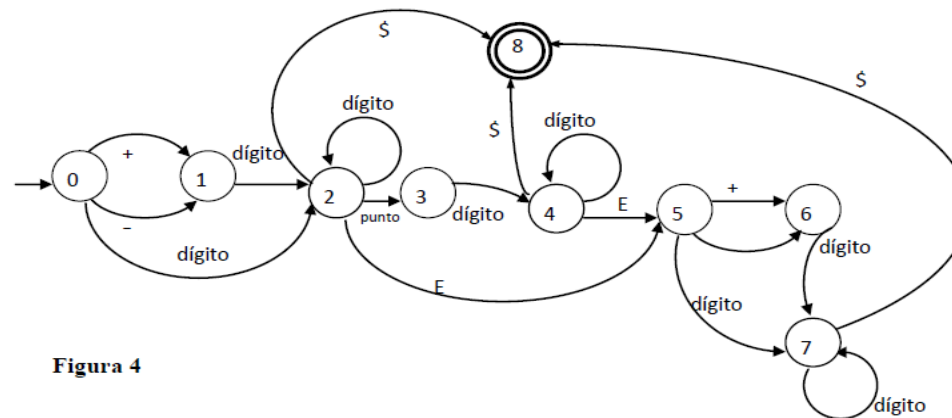


Figura 4

Definición de autómata finito no determinista

Un AFN se representa esencialmente como un AFD:

$$A = (Q, \Sigma, \delta, q_0, F)$$

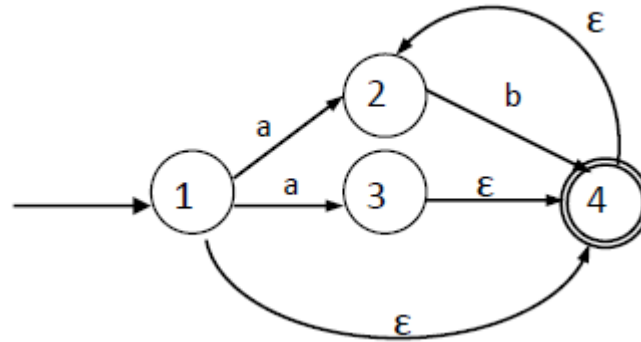
donde:

- 1. Q es un conjunto **finito de estados**.
- 2. Σ es un conjunto **finito de símbolos de entrada**.
- 3. q_0 , un elemento de Q , **es el estado inicial**.
- 4. F , un subconjunto de Q , es el **conjunto de estados finales** (o de aceptación).
- 5. δ , la función de transición, es una función que toma como argumentos un estado de Q y un símbolo de entrada de Σ y devuelve **un subconjunto de Q** .

La única diferencia entre un AFN y un AFD se encuentra en el tipo de valor que devuelve δ : **un conjunto de estados** en el caso de un **AFN** y un **único estado** en el caso de un **AFD**.

Ejemplo de un AFN:

Consideremos el siguiente autómata:



La cadena **abb** puede ser aceptada por cualquiera de las siguientes secuencias de transiciones:

$\rightarrow 1 \xrightarrow{a} 2 \xrightarrow{b} 4 \xrightarrow{\epsilon} 2 \rightarrow 4$

$\rightarrow 1 \xrightarrow{a} 3 \xrightarrow{\epsilon} 4 \xrightarrow{\epsilon} 2 \xrightarrow{b} 4 \xrightarrow{\epsilon} 2 \rightarrow 4$

Los AFN se pueden implementar de maneras similares a los AFD, excepto que como los AFN son no determinísticos, tienen muchas secuencias diferentes de transiciones en potencia que se deben probar.

Un programa que simula un AFN debe almacenar transiciones que todavía no se han probado y realizar el retrosegimiento a ellas en caso de falla.

Los algoritmos que realizan una gran cantidad de retrosegimientos tienden a ser poco eficientes

Se puede resolver el problema de simular un AFN por medio del método que estudiaremos más adelante, el cual convierte un AFN en un AFD.

CLASE 3:

EXPRESIONES REGULARES

Las expresiones regulares representan **patrones** de cadenas de caracteres.

Una expresión regular **r** se encuentra completamente definida mediante el **conjunto de cadenas con las que concuerda**.

Este conjunto se denomina **lenguaje generado por la expresión regular** y se escribe como **L(r)**.

Aquí la palabra **lenguaje** se utiliza sólo para definir "**conjunto de cadenas**" y no tiene una relación específica con un lenguaje de programación. Este lenguaje depende, en primer lugar, del conjunto de caracteres que se encuentra disponible.

Este conjunto de símbolos legales se conoce como alfabeto y por lo general se representa mediante el símbolo griego Σ (sigma).

Una expresión regular **r** también contendrá caracteres del alfabeto, pero esos **caracteres tendrán un significado diferente**: en una expresión regular **todos los símbolos indican patrones**.

Una expresión regular **r** puede contener caracteres que tengan **significados especiales**. Este tipo de caracteres se llaman **metacaracteres o metasímbolos**, y por lo general no pueden ser caracteres legales en el alfabeto, porque no podríamos distinguir su uso como **metacaracteres** de su uso como miembros del alfabeto.

Sin embargo, a menudo no es posible requerir tal exclusión, por lo que se debe utilizar una convención para diferenciar los dos usos posibles de un **metacarácter**

Expresiones regulares básicas

Éstas son los caracteres simples del alfabeto, los cuales se corresponden a sí mismos.

Dado cualquier carácter a del alfabeto Σ , indicamos que la expresión regular a **corresponde al carácter a escribiendo $L(a) = \{a\}$.**

Existen otros dos símbolos que necesitaremos en situaciones especiales. Necesitamos poder indicar una concordancia con la cadena vacía.

Utilizaremos el símbolo ϵ (épsilon) para denotar la cadena vacía, y definiremos el **metasímbolo ϵ (e en negritas) estableciendo que $L(\epsilon) = \{ \epsilon \}$**

También necesitaremos ocasionalmente ser capaces de describir un símbolo que corresponda a la ausencia de cadenas, el conjunto vacío, el cual escribiremos como $\{ \}$

Emplearemos para esto el símbolo ϕ y escribiremos $L\{\phi\} = \{ \}$.

Observe la diferencia entre $\{ \}$ y $\{ \epsilon \}$: el conjunto $\{ \}$ no contiene ninguna cadena, mientras que el conjunto $\{ \epsilon \}$ contiene la cadena simple que no se compone de ningún carácter.

Operaciones de expresiones regulares

- **Selección entre alternativas**

Si **r** y **s** son expresiones regulares, entonces **r | s** es una expresión regular que define cualquier cadena que concuerda con **r** o con **s**.

En términos de lenguajes, el lenguaje de **r | s** es la unión de los lenguajes de **r** y **s**, o $L(r | s) = L(r) \cup L(s)$.

Como un ejemplo simple, considere la expresión regular **a | b**: ésta corresponde tanto al carácter **a** como al carácter **b**,

$$L(a | b) = L(a) \cup L(b) = \{a\} \cup \{b\} = \{a, b\}.$$

- **Concatenación**

La concatenación de dos expresiones regulares **r** y **s** se escribe como **rs**, y corresponde a cualquier cadena que sea la concatenación de dos cadenas, con la primera de ellas correspondiendo a **r** y la segunda correspondiendo a **s**.

Por ejemplo, la expresión regular **ab** corresponde sólo a la cadena **ab**, mientras que la expresión regular **(a | b) c** corresponde a las cadenas **ac** y **bc**.

(Notemos el uso de los paréntesis como **metacaracteres**).

Dados dos conjuntos de cadenas **S1** y **S2**, el conjunto concatenado de cadenas **S1S2** es el conjunto de cadenas de **S1** complementado con todas las cadenas de **S2**.

Por ejemplo, si **S1 = {aa, b}** y **S2 = {a, bb}**, entonces :

$$S1 S2 = \{ aaa, aabb, ba, bbb \}.$$

Ahora la operación de concatenación para expresiones regulares se puede definir como:

$L(rs) = L(r)L(s)$. De esta manera (utilizando el ejemplo anterior):

$$L((a \mid b) c) = L(a \mid b)L(c) = \{a, b\} \{c\} = \{ac, bc\}.$$

- La concatenación también se puede extender a más de dos expresiones regulares:
- $L(r_1 r_2 \dots r_n) = L(r_1)L(r_2) \dots L(r_n)$ = el conjunto de cadenas formado al concatenar todas las cadenas de cada una de las $L(r_1), \dots, L(r_n)$.

- **Repetición**

La operación de repetición de una expresión regular, denominada también en ocasiones cerradura (de Kleene), se escribe r^* , donde r es una expresión regular.

La expresión regular r^* corresponde a cualquier concatenación infinita de cadenas, cada una de las cuales corresponde a r .

Por ejemplo, a^* corresponde a las cadenas ϵ , a , a , aa , aaa ,

Podemos definir la operación de repetición en términos de lenguajes generados definiendo, a su vez, una operación similar $*$ para conjuntos de cadenas. Dado un conjunto S de cadenas

$$s^* = \{ \epsilon \} \cup S \cup SS \cup SSS \cup \dots$$

Ésta es una unión de conjuntos infinita, pero cada uno de sus elementos es una concatenación finita de cadenas de S .

En ocasiones el conjunto S^* se escribe como sigue:

$$S^* = \sum_{n=0}^{\infty} S^n$$

Donde $S^n = S \dots S$ es la concatenación de S por n veces. ($S_0 = \{ \epsilon \}$).

La operación de repetición para expresiones:

$$L(r^*) = L(r)^*$$

Ejemplo: la expresión regular $(a \mid bb)^*$ corresponde a cualquiera de las cadenas siguientes: $\epsilon, a, bb, aa, abb, bba, bbbb, aaa, aabb$

Y así sucesivamente. En términos de lenguajes:

$$L((a \mid bb)^*) = L(a \mid bb)^* = \{a, bb\}^* = \{\epsilon, a, bb, aa, abb, bba, bbbb, aaa, aabb, abba, abbbb, bbaa, \dots\}.$$

- Consideremos el conjunto de cadenas **S** sobre el alfabeto $\Sigma = \{a, b\}$ compuesto de una **b** simple rodeada por el mismo número de **a**:

$$S = \{b, aba, aabaa, aaabaaa, \dots\} = \{a^n b a^n \mid n \neq 0\}$$

Este conjunto **no se puede** describir mediante una expresión regular.

La Única operación de repetición que tenemos es la operación de cerradura *****, la cual permite cualquier número de repeticiones.

De modo que si escribimos la expresión **a^*ba^*** (lo más cercano que podemos obtener en el caso de una expresión regular para **S**), no hay garantía de que el número de **a** antes y después de la **b** será el mismo.

Expresamos esto al decir que **"las expresiones regulares no pueden contar"**.

Precedencia de operaciones y el uso de los paréntesis

- Entre las tres operaciones, se le da al $*$ la precedencia más alta, a la concatenación se le da la precedencia que sigue ya la $|$ se le otorga la precedencia más baja.
- Por ejemplo, $a | bc^*$ se interpreta como $a | (b(c^*))$, mientras que $ab | c^*d$ se interpreta como $(ab) | ((c^*)d)$.
- Cuando deseemos indicar una precedencia diferente, debemos usar paréntesis para hacerlo.
- Por ejemplo $(a | bb)^*$ se interpretaría sin los paréntesis como $a | bb^*$, lo que corresponde a: a, b, bb, bbb, \dots
- Los paréntesis aquí se usan igual que en aritmética, donde $(3 + 4) * 5 = 35$, pero $3 + 4 * 5 = 23$, ya que se supone que $*$ tiene precedencia más alta que $+$.

Nombres para expresiones regulares

A menudo es útil como una forma de **simplificar la notación proporcionar un nombre para una expresión regular larga**, de modo que no tengamos que escribir la expresión misma cada vez que deseemos utilizarla.

Por ejemplo, si deseáramos desarrollar una expresión regular para una secuencia de uno o más dígitos numéricos, entonces escribiríamos $(0|1|2|...|9)$ **$(0|1|2|...|9)^*$** o podríamos escribir

dígito dígito* donde ***dígito = 0|1|2|...|9 es una definición regular del nombre dígito.***

Una expresión regular es una de las siguientes:

- 1. Una expresión regular básica constituida por un solo carácter a , donde a proviene de un alfabeto Σ de caracteres legales;
También lo son los metacaracteres ϵ ; o Φ . En el primer caso, $L(a) = \{a\}$; en el segundo, $L(\epsilon) = \{ \epsilon \}$; en el tercero, $L(\Phi) = \{ \}$.
- 2. Una expresión de la forma $r \mid s$, donde r y s son expresiones regulares. En este caso, $L(r \mid s) = L(r) \cup L(s)$.
- 3. Una expresión de la forma rs , donde r y s son expresiones regulares. En este caso, $L(rs) = L(r)L(s)$.
- 4. Una expresión de la forma r^* , donde r es una expresión regular. En este caso, $L(r^*) = L(r)^*$.
- 5. Una expresión de la forma (r) , donde r es una expresión regular. En este caso, $L((r)) = L(r)$. De este modo, los paréntesis no cambian el lenguaje, sólo se utilizan para ajustar la precedencia de las operaciones.

- **Ejemplo 1:**

Consideremos el alfabeto simple constituido por sólo tres caracteres alfabéticos: $\Sigma = \{ a, b, c \}$. También el conjunto de todas las cadenas en este alfabeto que **contengan exactamente una b**.

Este conjunto es generado por la expresión regular:

$$(a \mid c)^* b (a \mid c)^*$$

Vemos que, aunque **b** aparece en el centro de la expresión regular, la letra **b** no necesita estar en el centro de la cadena que se desea definir.

En realidad, la repetición de **a** o **c** antes y después de la **b** puede presentarse en diferentes números de veces. Por consiguiente, todas las cadenas siguientes están generadas mediante la expresión regular anterior: **b**, **abc**, **abaca**, **baaaac**, **ccbaca**, **ccccccb**.

- **Ejemplo 2:**

Con el mismo alfabeto que antes, considere el conjunto de todas las cadenas que contienen **como máximo una b**.

Una expresión regular para este conjunto se puede obtener utilizando la solución al ejemplo anterior como una **alternativa (definiendo aquellas cadenas con exactamente una b)**

y la expresión regular $(a \mid c)^*$ como la otra **alternativa (definiendo los casos sin b en todo)**. De este modo, tenemos la solución siguiente:

$$(a \mid c)^* \mid (a \mid c)^* b (a \mid c)^*$$

Una solución alternativa sería permitir que **b** o la **cadena vacía** apareciera entre las dos repeticiones de **a** o **c**:

$$(a \mid c)^* (b \mid \epsilon) (a \mid c)^*$$

Este ejemplo plantea un punto importante acerca de las expresiones regulares: el mismo lenguaje se puede generar mediante muchas expresiones regulares diferentes. Por lo general, intentamos encontrar una expresión regular tan simple como sea posible para describir un conjunto de cadenas.

METACARACTERES MÁS COMUNES

- **Metacaracteres delimitadores**

Esta clase de **metacaracteres** nos permite delimitar dónde queremos buscar los patrones de búsqueda. Ellos son:

Metacaracter	Descripción
^	inicio de línea.
\$	fin de línea.
\A	inicio de texto.
\Z	fin de texto.
.	cualquier caracter en la línea.
\b	encuentra límite de palabra.
\B	encuentra distinto a límite de palabra.

- **Metacaracteres clases predefinidas**

Estas son **clases predefinidas** que nos facilitan la utilización de las expresiones regulares. Ellos son:

Metacaracter	Descripción
<code>\w</code>	un caracter alfanumérico (incluye "_").
<code>\W</code>	un caracter no alfanumérico.
<code>\d</code>	un caracter numérico.
<code>\D</code>	un caracter no numérico.
<code>\s</code>	cualquier espacio (lo mismo que [<code>\t\n\r\f</code>]).
<code>\S</code>	un no espacio.

- **Metacaracteres - iteradores**

Cualquier elemento de una expresión regular puede ser seguido por otro tipo de metacaracteres, los *iteradores*.

*Usando estos metacaracteres se puede especificar el **número de ocurrencias** del caracter previo, de un metacaracter o de una subexpresión. Ellos son:*

Metacaracter	Descripción
*	cero o más, similar a {0,}.
+	una o más, similar a {1,}.
?	cero o una, similar a {0,1}.
{n}	exactamente n veces.
{n,}	por lo menos n veces.
{n,m}	por lo menos n pero no más de m veces.
*?	cero o más, similar a {0,}?
+?	una o más, similar a {1,}?

Metacaracter	Descripción
??	cero o una, similar a $\{0,1\}?$.
{n}?	exactamente n veces.
{n,}?	por lo menos n veces.
{n,m}?	por lo menos n pero no más de m veces.

En estos metacaracteres, los dígitos entre llaves de la forma {n,m}, especifican el **mínimo** número de ocurrencias en n y el **máximo** en m.

Se pueden consultar documentación oficial de tutoriales del uso de expresiones regulares en Python en:

<https://docs.python.org/3.5/library/re.html#regular-expression-syntax>

Un resumen por parte de Google Educación:

<https://developers.google.com/edu/python/regular-expressions>

Un par de documentos muy trabajados con ejemplos básicos y avanzados:

http://www.python-course.eu/python3_re.php

<https://pythonspot.com/regular-expressions/>

Ejemplos de Expresiones Regulares:

- IP address `(([2][5][0-5]\.)|([2][0-4][0-9]\.)|([0-1]?[0-9]?[0-9]\.)){3}(([2][5][0-5])|([2][0-4][0-9])|([0-1]?[0-9]?[0-9]))`
- Email `^[^@]+@[^@]+\.[^@]+`
- Date MM/DD/YY `(\d+/\d+/\d+)`
- Integer (positive) `(?<![-.])\b[0-9]+\b(?:!\.[0-9])`
- Integer `[+-]?(?<![\.\-])\b[0-9]+\b(?:!\.[0-9])`
- Float `(?<=>)\d+\.\d+|\d+`
- Hexadecimal `\s-([0-9a-fA-F]+)(?:-)?\s`

Generación de Autómatas a partir de Expresiones Regulares

Estudiaremos un algoritmo para traducir una expresión regular en un AFD.

También existe un algoritmo para traducir un AFD en una expresión regular, de manera que las dos nociones son equivalentes.

Sin embargo, debido a lo compacto de las expresiones regulares, se suele preferir a los AFD como descripciones de lenguajes.

El algoritmo más simple para traducir una expresión regular en un AFD pasa por una construcción intermedia, en la cual se deriva un AFN de la expresión regular, y posteriormente se emplea para construir un AFD equivalente.

Existen algoritmos que pueden traducir una expresión regular de manera directa en un AFD, pero son más complejos y la construcción intermedia también es de cierto interés. Así que nos concentraremos en la descripción de dos algoritmos, uno que traduce una expresión regular en un AFN y el segundo que traduce un AFN en un AFD.

- Construcción de *Thompson*

Utiliza transiciones ϵ para "pegar" las máquinas de cada segmento de una expresión regular con el fin de formar una máquina que corresponde a la expresión completa.

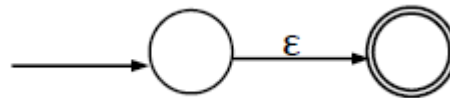
De este modo, la construcción es inductiva, y sigue la estructura de la definición de una expresión regular: mostramos un AFN para cada expresión regular básica y posteriormente mostramos cómo se puede conseguir cada operación de expresión regular al conectar entre sí los AFN de las subexpresiones (suponiendo que éstas ya se han construido).

Expresiones regulares básicas

- Una expresión regular básica es de la forma a, ϵ o Φ , donde a representa una correspondencia con un carácter simple del alfabeto, ϵ representa una coincidencia con la cadena vacía y Φ representa la correspondencia con ninguna cadena.
- Un **AFN** que es equivalente a la expresión regular a (es decir, que acepta precisamente aquellas cadenas en su lenguaje):



De manera similar, un AFN para ϵ es:



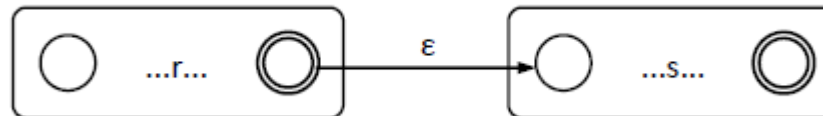
- **Concatenación:**

Para construir un **AFN** para la expresión regular **rs**, donde **r** y **s** son expresiones regulares. Suponemos de manera inductiva que los **AFN correspondientes a r y s ya se construyeron de la forma:**



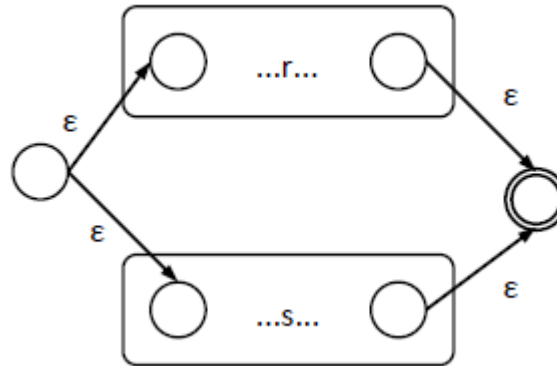
En estos dibujos, el círculo de la izquierda representa el **estado inicial**, y el **doble círculo de la derecha el estado final para los AFN de r y s** respectivamente.

Podemos construir el **AFN** para **rs**, de manera siguiente:



- Selección de alternativas

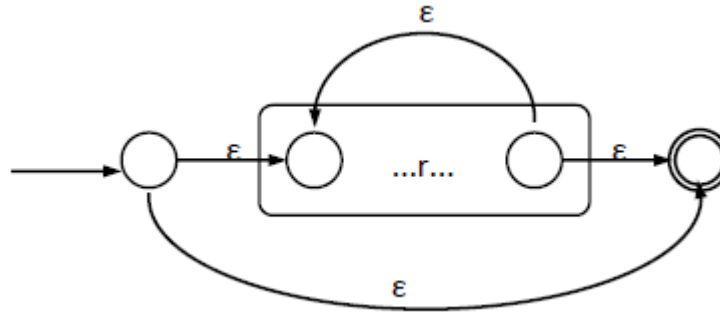
Para la expresión regular **r|s** bajo las mismas suposiciones que antes, podemos hacer:



Se agrega un nuevo estado de inicio y un nuevo estado final y se conectan mediante transiciones ϵ . Esta máquina acepta el lenguaje **L(r|s)**.

- Repetición o cerradura de kleene:

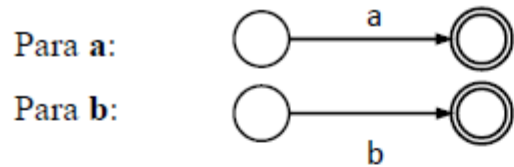
De modo similar podemos construir una máquina para r^* :



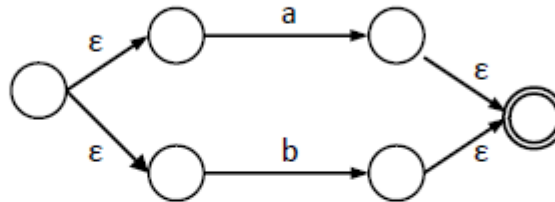
Nuevamente se agregan nuevos estados de inicio y aceptación, la repetición en esta máquina la proporciona la arista ϵ que une el estado final con el inicial de la máquina r .

Para asegurar que ϵ también es aceptada, se traza una transición ϵ entre el estado inicial y final de la máquina r^* .

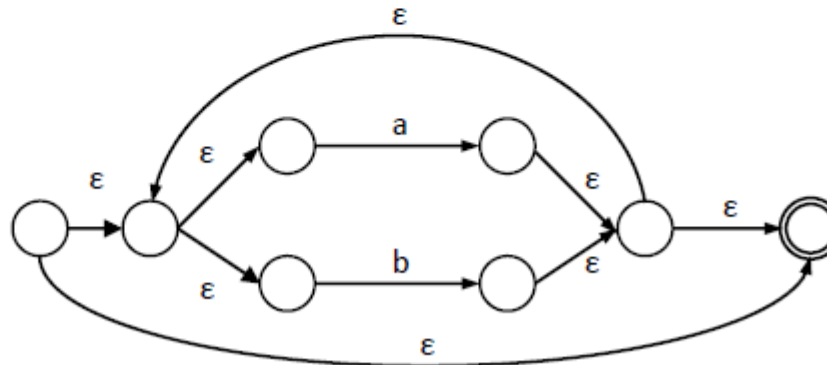
- **Ejemplo :**
- Para la expresión regular: $(a \mid b)^* abb$ se tiene:



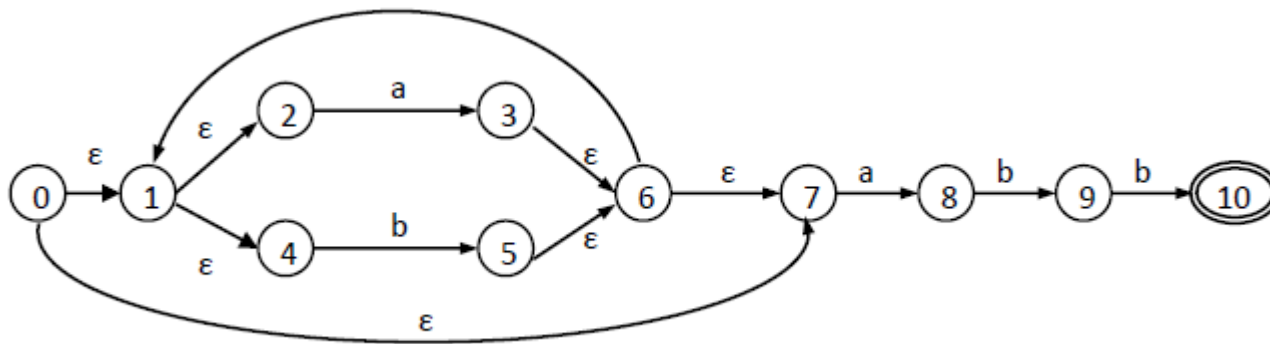
Para **$a \mid b$** :



Para **$(a \mid b)^*$** :



Finalmente para $(a \mid b)^* abb$



Paso de un Autómata Finito No Determinista a uno Determinista

Dado un **AFN** arbitrario, se construirá un **AFD equivalente** (es decir, uno que acepte precisamente las mismas cadenas).

Para hacerlo necesitaremos algún método con el que se eliminen tanto las transiciones **ϵ** como las **transiciones múltiples** de un estado en un carácter de entrada simple.

La eliminación de las transiciones **ϵ** implica el construir **cerraduras ϵ** , las cuales son el conjunto de todos los estados que pueden alcanzar las **transiciones ϵ** desde un estado o estados.

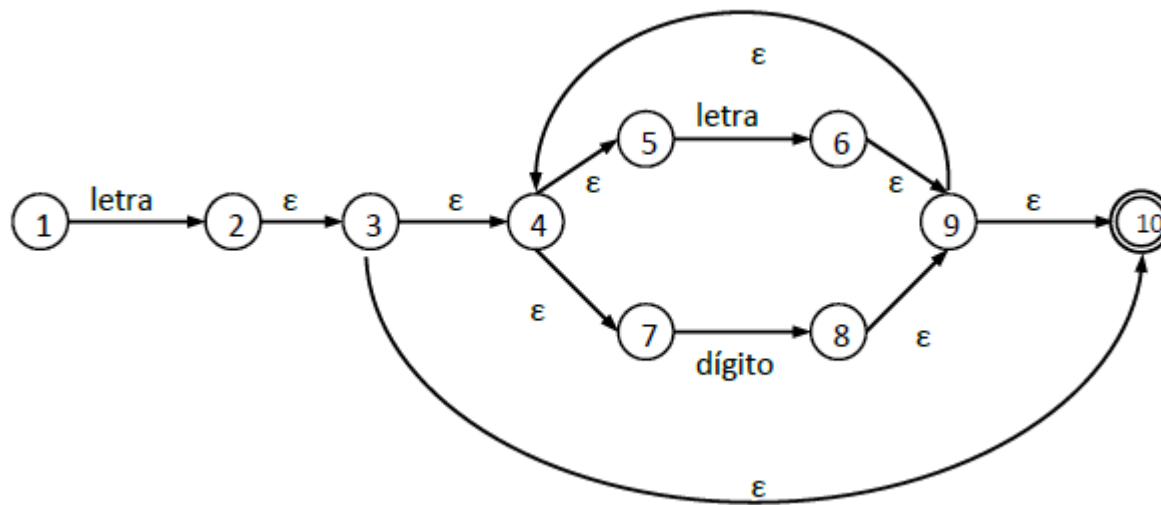
La eliminación de transiciones múltiples en un carácter de entrada simple implica mantenerse al tanto del conjunto de estados que son alcanzables al igualar un carácter simple.

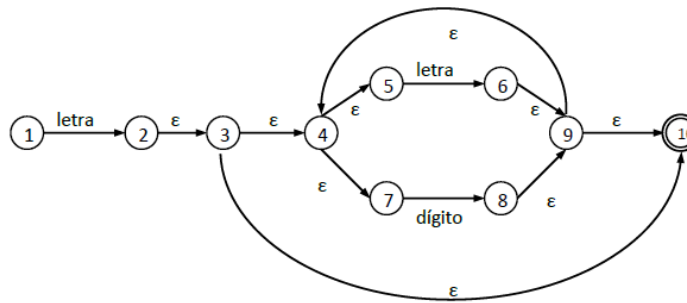
Ambos procesos nos conducen a considerar **conjuntos de estados** en lugar de estados simples.

El **AFD** que construimos va a tener como sus estados los **conjuntos de estados del AFN original**.

Este algoritmo se denomina **construcción de subconjuntos**.

Ejemplo: el autómata construido con el método de Thompson para la expresión regular: **letra (letra | dígito)***





$$\bar{S}_2 = \{2, 3, 4, 5, 7, 10\} = A$$

A será un estado del AFD

$$\tau_{letra}(A, letra) = \{4, 5, 6, 7, 9, 10\} = B$$

B es otro estado del AFD

$$\tau_{dígito}(A, dígito) = \{4, 5, 7, 8, 9, 10\} = C$$

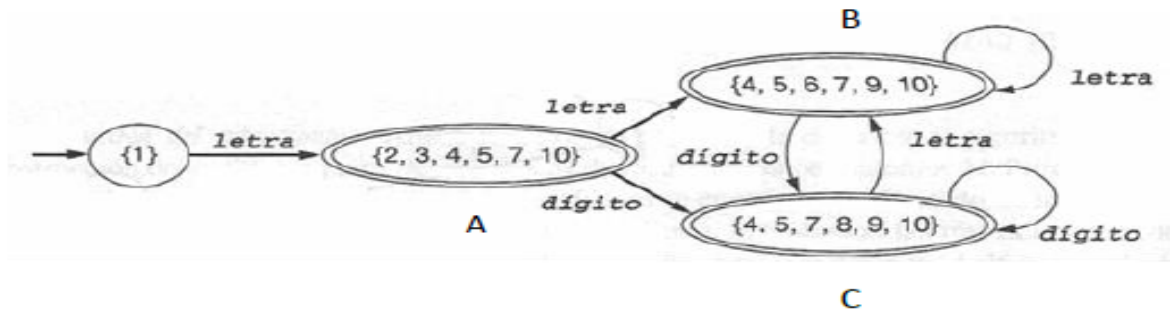
C es otro estado del AFD

Se aplica la función de transición a los nuevos estados:

$$\tau_{letra}(B, letra) = \{4, 5, 6, 7, 9, 10\} = B$$

$$\tau_{dígito}(C, dígito) = \{4, 5, 7, 8, 9, 10\} = C$$

Nótese que los estados A, B y C del AFD contienen el estado final 10 del AFN, por lo tanto serán estados finales del AFD que se muestra a continuación:



- Bibliografía:

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, "Teoría de Autómatas, Lenguajes y Computación". *Pearson Addison Wesley*.
- Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. "Compiladores, Principios, técnicas y herramientas" *Pearson Addison Wesley*.