Teoría de Autómatas

Introducción

La teoría de autómatas es el estudio de dispositivos de cálculo abstractos, es decir, de las "máquinas". Antes de que existieran las computadoras, en la década de los años treinta, A. Turing estudió una máquina abstracta que tenía todas las capacidades de las computadoras de hoy día, al menos en lo que respecta a lo que podían calcular. El objetivo de Turing era describir de forma precisa los límites entre lo que una máquina de cálculo podía y no podía hacer; estas conclusiones no sólo se aplican a las máquinas abstractas de Turing, sino a todas las máquinas reales actuales.

En las décadas de los años cuarenta y cincuenta, una serie de investigadores estudiaron las máquinas más simples, las cuales todavía hoy denominamos "autómatas finitos". Originalmente, estos autómatas se propusieron para modelar el funcionamiento del cerebro y, posteriormente, resultaron extremadamente útiles para muchos otros propósitos.

En 1969, S. Cook amplió el estudio realizado por Turing sobre lo que se podía y no se podía calcular. Cook fue capaz de separar aquellos problemas que se podían resolver de forma eficiente mediante computadora de aquellos problemas que, en principio, pueden resolverse, pero que en la práctica consumen tanto tiempo que las computadoras resultan inútiles para todo excepto para casos muy simples del problema. Este último tipo de problemas se denominan "**insolubles**" o "**NP-difíciles** ".

Es extremadamente improbable que incluso la mejora de carácter exponencial en la velocidad de cálculo que el hardware de computadora ha experimentado ("Ley de Moore") tenga un impacto significativo sobre nuestra capacidad para resolver casos complejos de problemas insolubles.

Todos estos desarrollos teóricos afectan directamente a lo que los expertos en computadoras hacen.

Algunos de los conceptos, como el de autómata finito y determinados tipos de gramáticas formales, se emplean en el diseño y la construcción de importantes clases de software.

Otros conceptos, como la máquina de Turing, nos ayudan a comprender lo que podemos esperar de nuestro software. En particular, la teoría de los problemas intratables nos permite deducir si podremos enfrentarnos a un problema y escribir un programa para resolverlo.

Introducción a los autómatas finitos

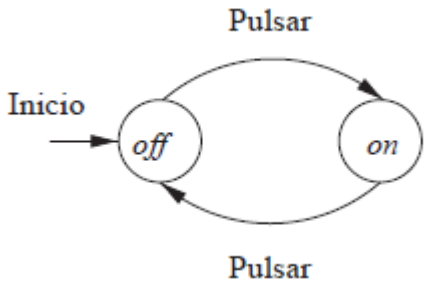
Los autómatas finitos constituyen un modelo útil para muchos tipos de hardware y software, algunos de los tipos más importantes:

1. Software para diseñar y probar el comportamiento de circuitos digitales.
2. El "analizador léxico" de un compilador típico, es decir, el componente del compilador que separa el texto de entrada en unidades lógicas (tokens), tal como identificadores, palabras clave y signos de puntuación.
3. Software para explorar cuerpos de texto largos, como colecciones de páginas web, o para determinar el número de apariciones de palabras, frases u otros patrones.
4. Software para verificar sistemas de todo tipo que tengan un número finito de estados diferentes, tales como protocolos de comunicaciones o protocolos para el intercambio seguro de información.

Aunque se verá una definición precisa de los distintos tipos de autómatas, comenzaremos esta introducción informal con un boceto de lo que es y lo que hace un autómata.

Ejemplo:

Quizá el autómata finito no trivial más simple sea un interruptor de apagado/encendido (posiciones on/off). El dispositivo **recuerda** si está en el estado encendido ("on") o en el estado apagado ("off"), y permite al usuario pulsar un botón cuyo efecto es diferente dependiendo del estado del interruptor. Es decir, si el interruptor está en el estado off, entonces al pulsar el botón cambia al estado on, y si el interruptor está en el estado on, al pulsar el mismo botón pasa al estado off.



Conceptos fundamentales de la teoría de autómatas

Los conseptos más importantes empleados en la teoría de autómatas incluyen el de "**alfabeto**" (un conjunto de símbolos),"**cadenas de caracteres**" (una lista de símbolos de un alfabeto) y "**lenguaje**" (un conjunto de cadenas de caracteres de un mismo alfabeto).

Alfabetos

Un alfabeto es un conjunto de símbolos **finito y no vacío**. Convencionalmente, utilizamos el símbolo $\sum$ para designar un alfabeto. Entre los alfabetos más comunes se incluyen los siguientes:

1. $\sum = \{0,1\}$, el alfabeto binario.
2. $\sum = \{a,b, \ldots, z\}$, el conjunto de todas las letras minúsculas.
3. $\sum = \{0,1, \ldots,9\}$, el conjunto de los dígitos del 0 al 9.
4. El conjunto de todos los caracteres ASCII o el conjunto de todos los caracteres ASCII imprimibles.

Cadenas de caracteres

Una cadena de caracteres (que también se denomina en ocasiones palabra) es una secuencia finita de símbolos seleccionados de algún alfabeto. Por ejemplo, 01101 es una cadena del alfabeto binario $\sum = \{0,1\}$. La cadena 111 es otra cadena de dicho alfabeto.

La cadena vacía

La cadena vacía es aquella cadena que presenta cero apariciones de símbolos. Esta cadena, designada por **ε**, es una cadena que puede construirse en cualquier alfabeto.

Longitud de una cadena

Suele ser útil clasificar las cadenas por su longitud, es decir, el número de posiciones ocupadas por símbolos dentro de la cadena. Por ejemplo, 01101 tiene una longitud de 5.
La notación estándar para indicar la longitud de una cadena w es |w|. Por ejemplo, |011| = 3

Potencias de un alfabeto

Si $\sum$ es un alfabeto, podemos expresar el conjunto de todas las cadenas de una determinada longitud de dicho alfabeto utilizando una notación exponencial. Definimos $\sum^k$ para que sea

el conjunto de las cadenas de longitud k, tales que cada uno de los símbolos de las mismas pertenece a $\sum$

Observe que $\sum^0 = \{ \varepsilon \}$, independientemente de cuál sea el alfabeto $\sum$ es decir, ε es la única cadena cuya longitud es 0.

Si $\sum = \{0,1\}$, entonces

$\sum^1 = \{0,1\}$
$\sum^2 = \{00,01,10,11\}$
$\sum^3 = \{000,001,010,011,100,101,110,111\}$, etc.

El conjunto de todas las cadenas de un alfabeto $\sum$ se designa mediante $\sum^*$. Por ejemplo:

$\{0,1\}^* = \{ \varepsilon ,0,1,00,01,10,11,000, \ldots\}$

Expresado de otra forma:
$\sum^* = \sum^0 \cup \sum^1 \cup \sum^2 \cup \cdots$

En ocasiones, desearemos excluir la cadena vacía del conjunto de cadenas. El conjunto de cadenas no vacías del alfabeto $\sum$ se designa como $\sum^+$. Por tanto, dos equivalencias apropiadas son:

$\sum^+ = \sum^1 \cup \sum^2 \cup \cdots$

$\sum^* = \sum^+ \cup \{\varepsilon\}$.

Concatenación de cadenas

Sean x e y dos cadenas. Entonces, xy denota la concatenación de x e y, es decir, la cadena formada por una copia de x seguida de una copia de y. Dicho de manera más precisa, si x es la cadena compuesta por i símbolos
$x = a_1a_2 \cdots a_i$ e y es la cadena compuesta por j símbolos $y = b_1b_2 \cdots b_j$, entonces xy es la cadena de longitud
i + j: $xy = a_1a_2 \cdots a_ib_1b_2 \cdots b_j$.

Ejemplo

Sean x = 01101 e y = 110. Entonces xy = 01101110 e yx = 11001101. Para cualquier cadena w, tenemos las ecuaciones ε w = w ε = w. Es decir, ε es el elemento neutro de la concatenación, dado que su concatenación con cualquier cadena proporciona la misma cadena.

En consecuencia podemos definir la Potencia como:

Potencia:

$$w^n = \begin{cases} \varepsilon & n = 0 \\ w w^{n-1} & n > 0 \end{cases}$$

Si w = 122 sobre el alfabeto Σ = { 1, 2 }

$w^0 = \varepsilon$

w^1=122

w^2=122122

w^3=122122122

Lenguajes

Un conjunto de cadenas, todas ellas seleccionadas de un $\sum^*$, donde $\sum$ es un determinado alfabeto se denomina lenguaje.

Si $\sum$ es un alfabeto y L $\subseteq$ $\sum^*$, entonces L es un lenguaje de $\sum$

Un ejemplo es el lenguaje C, o cualquier otro lenguaje de programación, donde los programas correctos son un subconjunto de las posibles cadenas que pueden formarse a partir del alfabeto del lenguaje. Este alfabeto es un subconjunto de los caracteres ASCII. El alfabeto en concreto puede diferir ligeramente entre diferentes lenguajes de programación, aunque generalmente incluye las letras mayúsculas y minúsculas, los dígitos, los caracteres de puntuación y los símbolos matemáticos.

Sin embargo, existen también otros muchos lenguajes

Algunos ejemplos son los siguientes:

1. El lenguaje de todas las cadenas que constan de n ceros seguidos de n unos para cualquier $n \geq 0$:
1. $\{\sum, 01, 0011, 000111, \ldots\}$.
2. El conjunto de cadenas formadas por el mismo número de ceros que de unos:
3. $\{\sum, 01, 10, 0011, 0101, 1001, \ldots\}$
4. El conjunto de números binarios cuyo valor es un número primo:
5. $\{10, 11, 101, 111, 1011, \ldots\}$
5. $\sum^*$ es un lenguaje para cualquier alfabeto $\sum$.
6. ϕ, el lenguaje vacío, es un lenguaje de cualquier alfabeto.
7. $\{ \varepsilon \}$, el lenguaje que consta sólo de la cadena vacía, también es un lenguaje de cualquier alfabeto.

$\phi \neq \{ \varepsilon \}$, el primero no contiene ninguna cadena y el segundo sólo tiene una cadena.

La única restricción importante sobre lo que puede ser un lenguaje es que todos los alfabetos son finitos. De este modo, los lenguajes, aunque pueden tener un número infinito de cadenas, están restringidos a que dichas cadenas estén formadas por los símbolos que definen un alfabeto finito y prefijado.

Operaciones con Lenguajes

Se pueden definir las siguientes operaciones con lenguajes:

Sean A y B dos lenguajes tenemos:

Concatenación:

$$A.B = \{ w.x \mid w \in A \text{ y } x \in B \}$$

Si A es un lenguaje sobre Σ_1 y B es un lenguaje sobre Σ_2

AB será un lenguaje sobre $\Sigma_1 \cup \Sigma_2$

Unión:

$$A \cup B = \{ x \mid x \in A \text{ o } x \in B \}$$

Intersección:

$A \cap B = \{ x \mid x \in A \text{ y } x \in B \text{ simultáneamente} \}$

Cerradura de Kleene:

Es el lenguaje compuesto por todas las cadena sobre un alfabeto dado Σ

Ejemplo: Σ = {1}

A* = { ε, 1, 11, 111, 1111, …. }

A* es infinito

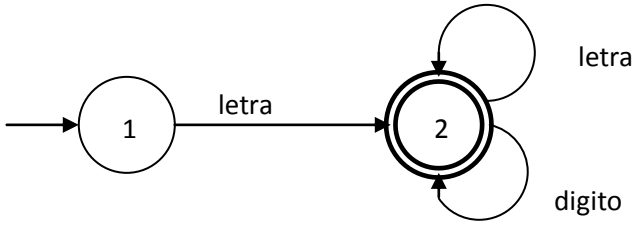
$$A^* = \sum_{0}^{\infty} A^n$$

Autómatas Finitos

Los autómatas finitos, o máquinas de estados finitos, son una manera matemática para describir clases particulares de algoritmos (o "máquinas"). En particular, los autómatas finitos se pueden utilizar para describir el proceso de reconocimiento de patrones en cadenas de entrada, y de este modo se pueden utilizar para construir analizadores léxicos. Por supuesto, también existe una fuerte relación entre los autómatas finitos y las expresiones regulares, y veremos cómo construir un autómata finito a partir de una expresión regular. Sin embargo, antes de comenzar nuestro estudio de los autómatas finitos de manera apropiada, consideraremos un ejemplo explicativo. El patrón para identificadores como se define comúnmente en los lenguajes de programación está dado por la siguiente definición regular (supondremos que letra y dígito ya se definieron):

identificador = letra(letra | dígito)*

Esto representa una cadena que comienza con una letra y continúa con cualquier secuencia de letras y/o dígitos. El proceso de reconocer una cadena así se puede describir mediante el diagrama de la siguiente figura:



En este diagrama los círculos numerados 1 y 2 representan estados, que son localidades en el proceso de reconocimiento que registran cuánto del patrón ya se ha visto. Las líneas con flechas representan transiciones que registran un cambio de un estado a otro en una coincidencia del carácter o caracteres mediante los cuales son etiquetados. En el diagrama de muestra, el estado 1 es el estado de inicio, o el estado en el que comienza el proceso de reconocimiento. Por convención, el estado de inicio se indica dibujando una línea con flechas sin etiqueta que proviene de "de ninguna parte". El estado 2 representa el punto en el cual se ha igualado una sola letra (lo que se indica mediante la transición del estado 1 al estado 2 etiquetada con letra). Una vez en el estado 2, cualquier número de letras y/o dígitos se puede ver, y una coincidencia de éstos nos regresa al estado 2. Los estados que representan el fin del proceso de reconocimiento, en los cuales podemos declarar un éxito, se denominan estados de aceptación, y se indican dibujando un borde con línea doble alrededor del estado en el diagrama. Puede haber más de uno de éstos. En el diagrama de muestra el estado 2 es un estado de aceptación, lo cual indica que, después que cede una letra, cualquier secuencia de letras y dígitos subsiguiente (incluyendo la ausencia de todas) representa un identificador legal.

Veamos otro ejemplo de un problema real, cuya solución emplea autómatas finitos que desempeñan un importante papel.

Vamos a investigar protocolos que ayudan a gestionar el “dinero electrónico” (los archivos que un cliente puede utilizar para realizar pagos por bienes a través de Internet, y que el vendedor puede recibir con la seguridad de que el “dinero” es real). El vendedor debe estar seguro de que el archivo no ha sido falsificado y de que el cliente no se ha quedado con una copia del mismo para enviárselo más de una vez.

La cuestión de la falsificación del archivo es algo que un banco debe asegurar mediante una política criptográfica. Es decir, un tercer jugador, el banco, tiene que emitir y cifrar los

archivos de “dinero”, de manera que la falsificación no constituya un problema. Sin embargo, el banco desempeña una segunda tarea también importante: tiene que mantener una base de datos de todas las transacciones válidas que se hayan realizado, de modo que sea posible verificar al vendedor que el archivo que ha recibido representa dinero real que ha sido ingresado en su cuenta.

Para poder utilizar dinero electrónico, se necesitan protocolos que permitan la manipulación del dinero en las distintas formas que los usuarios desean. Dado que los sistemas monetarios siempre invitan al fraude, tenemos que verificar la política que adoptemos independientemente de cómo se emplee el dinero. Es decir, tenemos que demostrar que las únicas cosas que pueden ocurrir son las cosas que queremos que ocurran (cosas que no permitan a un usuario poco escrupuloso robar a otros o “fabricar” su propio dinero electrónico).

Veamos un ejemplo muy simple de un (pobre) protocolo de dinero electrónico, modelado mediante un autómata finito y vamos a mostrar cómo pueden utilizarse las construcciones sobre autómatas para verificar los protocolos (o, como en este caso, para descubrir que el protocolo tiene un error).

Tenemos tres participantes: el cliente, la tienda y el banco. Para simplificar, suponemos que sólo existe un archivo de “dinero electrónico”. El cliente puede decidir transferir este archivo a la tienda, la cual lo reenviará al banco (es decir, solicita al banco que emita un nuevo archivo que refleje que el dinero pertenece a la tienda en lugar de al cliente) y suministra los bienes al cliente. Además, el cliente tiene la opción de cancelar el archivo;
es decir, el cliente puede pedir al banco que devuelva el dinero a su cuenta, anulando la posibilidad de gastarlo.

La interacción entre los tres participantes se limita por tanto a cinco sucesos:

1. El cliente decide pagar. Es decir, el cliente envía el dinero a la tienda.
2. El cliente decide cancelar el pago. El dinero se envía al banco con un mensaje que indica que el dinero se ha añadido a la cuenta bancaria del cliente.
3. La tienda suministra los bienes al cliente.

4. La tienda libra el dinero. Es decir, el dinero se envía al banco con la solicitud de que su valor se asigne a la cuenta de la tienda.
5. El banco transfiere el dinero creando un nuevo archivo de dinero electrónico cifrado y se lo envía a la tienda.

El protocolo:

Los tres participantes tienen que diseñar sus comportamientos muy cuidadosamente, o de lo contrario pueden producirse errores. En nuestro ejemplo, podemos hacer la suposición de que no es posible confiar en que el cliente actúe de manera responsable. En concreto, el cliente puede intentar copiar el archivo de dinero y emplearlo para pagar varias veces, o puede pagar y luego cancelar el pago, obteniendo así los bienes "gratuitamente".

El banco debe comportarse de manera responsable. En concreto, debe garantizar que dos tiendas no puedan liberar el mismo archivo de dinero y no debe permitir que el mismo dinero sea a la vez cancelado por el cliente y liberado por una tienda. La tienda también tiene que ser cuidadosa. En concreto, no debe suministrar los bienes hasta asegurarse de que ha recibido dinero electrónico válido.

Los protocolos de este tipo pueden representarse mediante un autómata finito. Cada estado representa una situación en la que puede encontrarse uno de los participantes. Es decir, el estado "recuerda" qué sucesos importantes han ocurrido y qué sucesos todavía no han tenido lugar. Las transiciones entre estados se producen cuando tiene lugar uno de los cinco sucesos descritos anteriormente. Supondremos que estos sucesos son "externos" al autómata que representa a los tres participantes, aunque cada participante sea responsable de iniciar uno o más de los sucesos. Lo importante en este problema es qué secuencias pueden ocurrir y no quién las inicia.

La Figura 1 representa los tres participantes mediante autómatas. En este diagrama, sólo se muestran los sucesos que afectan a un participante. Por ejemplo, la acción pagar sólo afecta al cliente y a la tienda. El banco no sabe que el cliente ha enviado el dinero a la tienda; sólo descubre este hecho cuando la tienda ejecuta la acción librar.

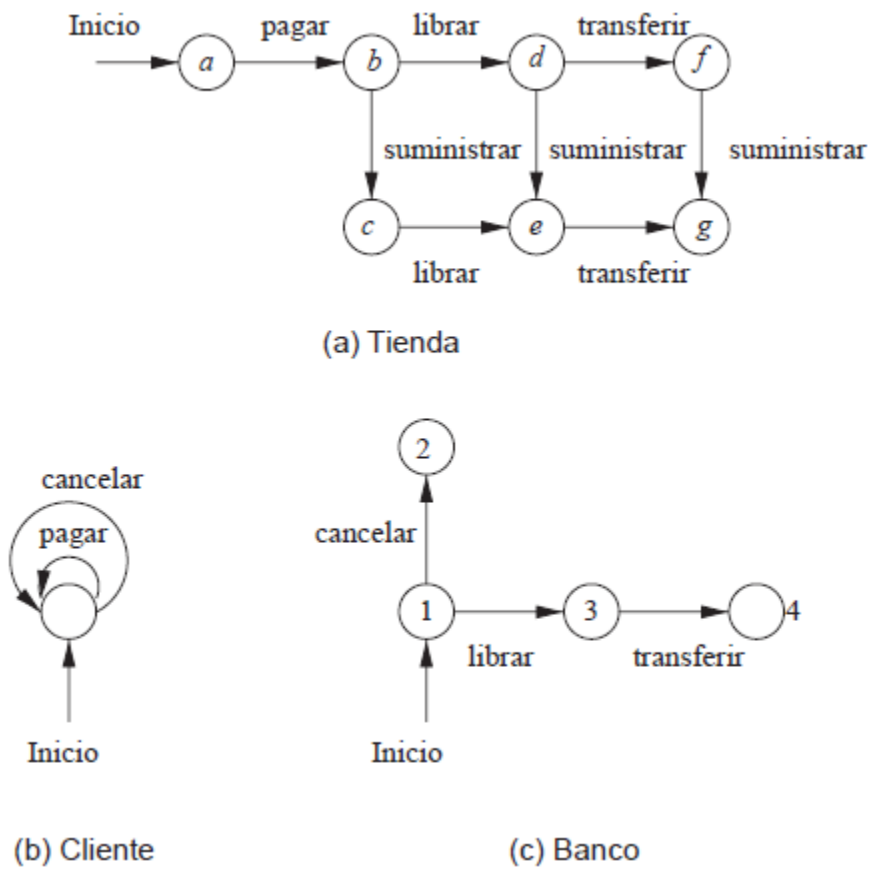


Figura 1.

Examinemos en primer lugar el autómata (c) correspondiente al banco. El estado inicial es el estado 1, que representa la situación en la que el banco ha emitido el archivo de dinero electrónico en cuestión pero no se ha solicitado que sea liberado o cancelado. Si el cliente envía al banco una solicitud cancelar, entonces el banco devuelve el dinero a la cuenta del cliente y pasa al estado 2. Este estado representa la situación en la que el pago ha sido cancelado. El banco, puesto que es responsable, no saldrá del estado 2 una vez que ha entrado en él, ya que no debe permitir que el mismo pago sea cancelado otra vez o gastado por el cliente.

Alternativamente, cuando se encuentra en el estado 1, el banco puede recibir una solicitud de librar procedente de la tienda. En ese caso, pasa al estado 3 y envía rápidamente a la tienda un mensaje transferir, con un nuevo archivo de dinero electrónico que ahora

pertenece a la tienda. Después de enviar el mensaje transferir, el banco pasa al estado 4. En dicho estado, no aceptará solicitudes cancelar ni librar ni realizará ninguna otra acción referente a este archivo de dinero electrónico en concreto.

Consideremos ahora la Figura 1(a), el autómata que representa las acciones de la tienda. Mientras que el banco siempre hace lo correcto, el sistema de la tienda tiene algunos defectos. Imaginemos que las operaciones de suministro y financieras se hacen mediante procesos separados, de modo que existe la oportunidad para que la acción de suministro se lleve a cabo antes, después o durante la de libramiento del dinero electrónico. Esta política permite a la tienda llegar a una situación en la que ya haya suministrado los bienes y se descubra que el dinero no es válido.

Observemos por último el autómata correspondiente al cliente, la Figura 1(b). Este autómata sólo tiene un estado, el cual refleja el hecho de que el cliente "puede hacer cualquier cosa". El cliente puede realizar las acciones pagar y cancelar cualquier número de veces, en cualquier orden y después de cada acción permanece en su único estado.

Los autómatas finitos, o máquinas de estados finitos, son una manera matemática para describir clases particulares de algoritmos (o "máquinas"). En particular, los autómatas finitos se pueden utilizar para describir el proceso de reconocimiento de patrones en cadenas de entrada, y de este modo se pueden utilizar para construir analizadores léxicos. Por supuesto, también existe una fuerte relación entre los autómatas finitos y las expresiones regulares, y veremos cómo construir un autómata finito a partir de una expresión regular.

Definición de los autómatas finitos determinísticos

Los diagramas como el que analizamos son descripciones útiles de los autómatas finitos porque nos permiten visualizar fácilmente las acciones del algoritmo. Sin embargo, en ocasiones 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 obstante, no necesitaremos una visión tan abstracta como ésta, y describiremos la mayoría de los ejemplos en términos del diagrama solo. Otras descripciones de autómatas

finitos también son posibles, particularmente las tablas, y éstas serán útiles para convertir los algoritmos en código de trabajo.

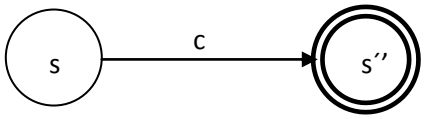
Definición

Un AFD (Autómata Finito Determinista) M se compone de:

- Un alfabeto Σ
- Un conjunto de estados S
- Una función de transición ∂: $S \times \Sigma \rightarrow S$
- Un estadio inicial $S_0 \in S$
- Un conjunto de estados de aceptación $A \in S$

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

Para una transición del estado s al estado s' etiquetada con c, se tendrá un diagrama del siguiente aspecto:



El doble círculo del estado s', indica que este es un estado de aceptación.
Un AFD para reconocer el lenguaje de los identificadores se verá como:

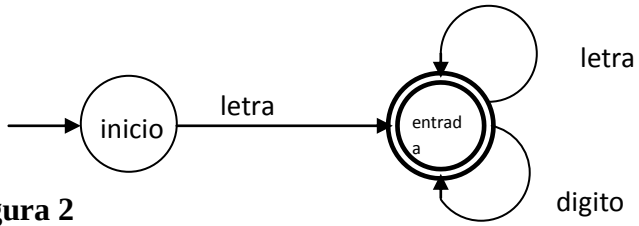


Figura 2

Debemos destacar que en este caso hemos usado nombres para los estados, en lugar de números, lo que no contradice a la definición dada. Otra cosa a tener en cuenta es que no se han etiquetado las aristas con símbolos del alfabeto, sino que se han etiquetado con nombres que representan conjuntos de símbolos:

letra → [a-zA-Z] digito → [0-9]

Tal como se los ha definido, un AFD debe tener una transición ∂ (s,c) para cada estado s y para cada símbolo c de del alfabeto Σ, sin embargo en el diagrama de la figura 2, en el estado inicio, está definida la transición ∂ (inicio, c), solo si c es una letra. Las transiciones que no se han definido, representan errores para el lenguaje que reconoce el autómata, y en general no se dibujan en los diagramas de estado, pero deben ser tenidas en cuenta al momento de la programación del autómata.

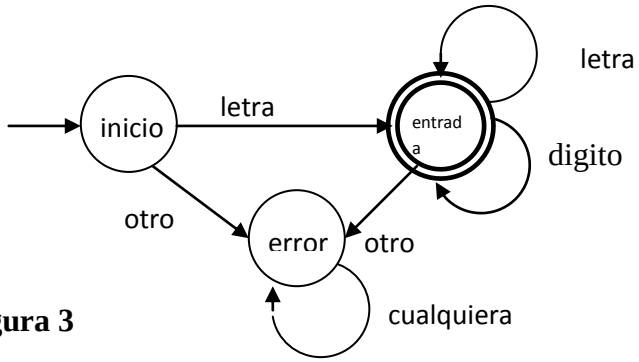


Figura 3

En esta figura, se agrega el estado error y se etiquetan las transiciones de error como *otro.* Por convención, *otro* representa cualquier carácter que no aparezca en ninguna otra transición desde el estado donde se origina. En el estado de inicio, *otro* es:

otro → ~ letra

Y en el estado entrada _id, *otro* es: **otro → ~ [letra | dígito]**

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:

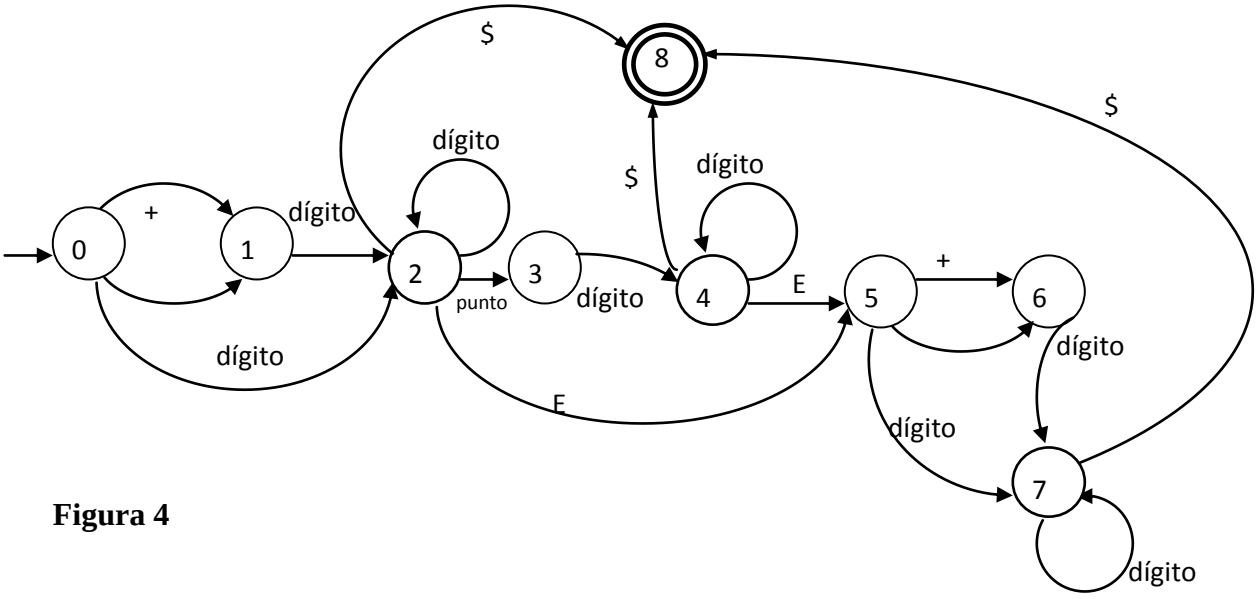


Figura 4

Las transiciones etiquetadas con **"$"** equivalen a *otro* carácter distinto al del resto de transiciones que salen del estado. Se han numerado los estados, siendo el estado inicial el "0", y el de aceptación o final el "8".

Algunos ejemplos de los números reconocidos por este autómata son:

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

Especificar un AFD utilizando una quíntupla con una descripción detallada de la función de transición ∂ resulta bastante tedioso y complicado de leer.

Hay disponibles notaciones más cómodas para describir los autómatas como ser una Tabla de Transiciones.

Del diagrama de la figura 4, se puede obtener la siguiente ***Tabla de Transiciones:***

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					

Un Autómata acepta o reconoce una cadena, sí y solo si existe un camino desde el estado inicial a alguno de los estados de aceptación, de forma que las ***etiquetas*** de la aristas del camino de aceptación deletreen la cadena.

Estudiamos los AFD como una manera de representar algoritmos que aceptan cadenas de caracteres de acuerdo con un patrón. Existe una fuerte relación entre una expresión regular para un patrón y un AFD que acepta cadenas de acuerdo con el patrón. Advertimos que el diagrama de un AFD no representa todo lo que necesita un AFD, sino que sólo proporciona un esbozo de su funcionamiento. En realidad, vimos que la definición matemática implica que un AFD debe tener una transición para cada estado y símbolo de $\sum$, y que aquellas transiciones que dan errores como resultado por lo regular se dejan fuera del diagrama para el AFD. Pero incluso la definición matemática no describe todos los aspectos del comportamiento de un algoritmo de AFD. Por ejemplo, no especifica lo que ocurre cuando se presenta un error. Tampoco especifica la acción que tomará un programa al alcanzar un estado de aceptación, o incluso cuando iguale un carácter durante una transición.

Una acción típica que ocurre cuando se hace una transición es mover el carácter de la cadena de entrada a una cadena que acumula los caracteres hasta convertirse en una cadena perteneciente al lenguaje reconocido por el autómata. Una acción típica cuando se alcanza

un estado de aceptación es devolver la cadena que se acaba de reconocer, junto con cualquier atributo asociado. Una acción típica cuando se alcanza un estado de error es retroceder hacia la entrada (retroseguimiento) o generar un error.

Nuestro ejemplo original de una cadena de identificador muestra gran parte del comportamiento que deseamos describir aquí. Si regresamos al autómata de la figura 2,

esta figura no muestra el comportamiento que queremos por varias razones. En primer lugar, el estado de error no es en realidad un error en absoluto, pero representa el hecho de que un identificador no va a ser reconocido (si venimos desde el estado de inicio), o bien, que se ha detectado un delimitador y ahora deberíamos aceptar la cadena. Supongamos por el momento (lo que de hecho es el comportamiento correcto) que existen otras transiciones representando las transiciones sin letra desde el estado de inicio. Entonces podemos indicar que se ha detectado un delimitador desde el estado de entrada, y que debería aceptarse la cadena identificador mediante el diagrama de la figura 5:

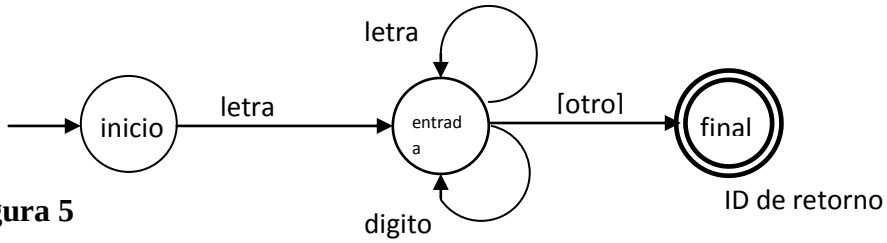


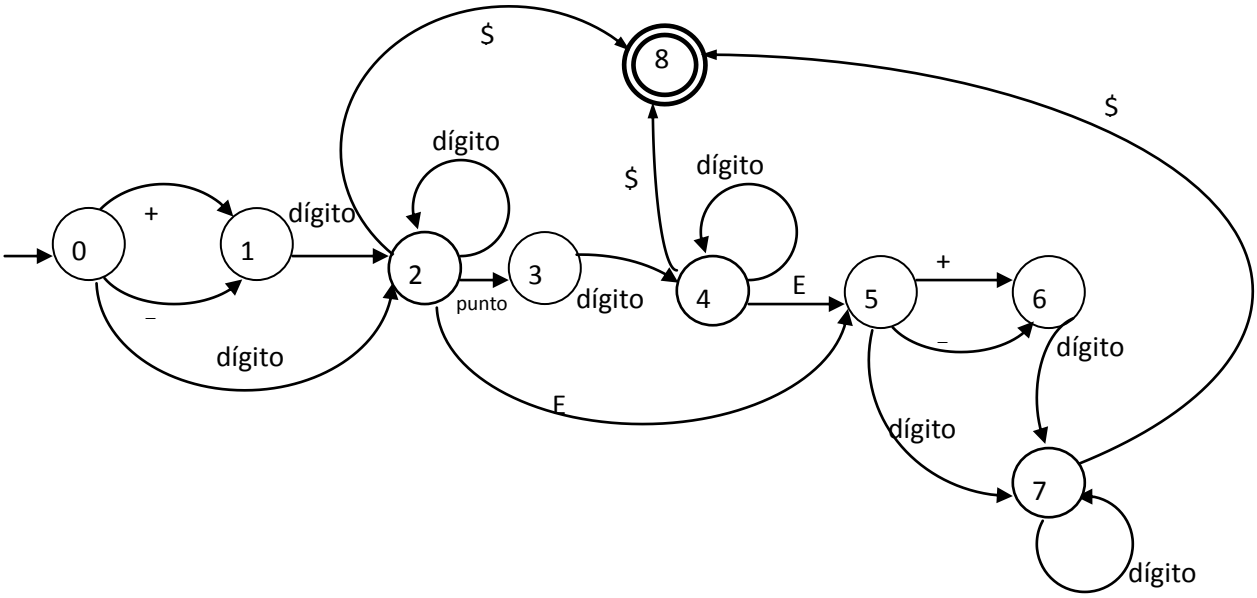
Figura 5

En el diagrama encerramos la transición *otro* entre corchetes para indicar que el carácter delimitador, se debería considerar como búsqueda hacia delante, es decir, que debería ser devuelto a la cadena de entrada y no consumido. Además, el estado de error se ha convertido en el estado de aceptación en este diagrama y no hay transiciones fuera del estado de aceptación. Esto es lo que queremos, puesto que el autómata debería reconocer una cadena a la vez y comenzar de nuevo en su estado de inicio después de reconocer cada una.

Implementación de Autómatas Finitos en Código

Existen diversas maneras de implementar en código un AFD, una de ellas es mediante la tabla de transiciones o Matriz de Transiciones que se vio anteriormente.

Recordemos el autómata para números de la figura 4



De este diagrama se construyó la tabla:

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					

Esta tabla representa perfectamente al autómata, obsérvese además que la mayoría de sus celdas están vacías y representan transiciones que no se muestran en el diagrama (es decir representan transiciones a estados de error, u otros procesamientos)

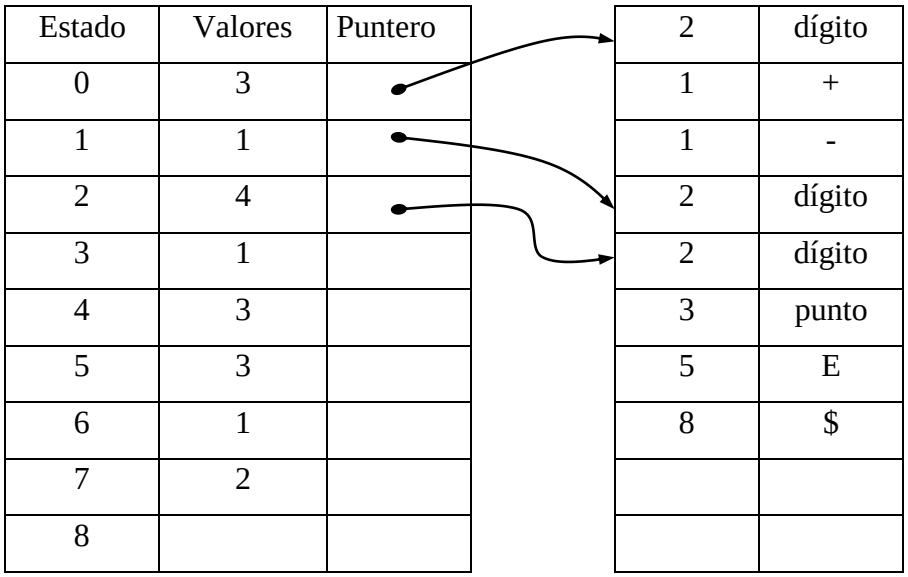
En base a esta tabla podemos escribir del código en una forma que implementará cualquier AFD, dadas las entradas y estructuras de datos apropiadas. El siguiente esquema de código en "C" supone que las transiciones se mantienen en un arreglo de transición T indexado por estados y carácter de entrada:

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

Los métodos algorítmicos como los que acabamos de describir se denominan controlados por tabla porque emplean tablas para dirigir el progreso del algoritmo. Los métodos controlados por tabla tienen ciertas ventajas: el tamaño del código se reduce, el mismo código funcionará para muchos problemas diferentes y el código es más fácil de modificar (mantener). La desventaja es que las tablas pueden volverse muy grandes y provocar un importante aumento en el espacio utilizado por el programa. En realidad, gran parte del espacio en los arreglos que acabamos de describir se desperdicia. Por lo tanto, los métodos controlados por tabla a menudo dependen de métodos de compresión de tablas, tales como representaciones de arreglos dispersos, aunque por lo regular existe una penalización en tiempo que se debe pagar por dicha compresión, ya que la búsqueda en tablas se vuelve más lenta. Como los autómatas deben ser eficientes, rara vez se utilizan estos métodos para ellos, aunque se pueden emplear en programas generadores de analizadores léxicos tales como Lex.

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		

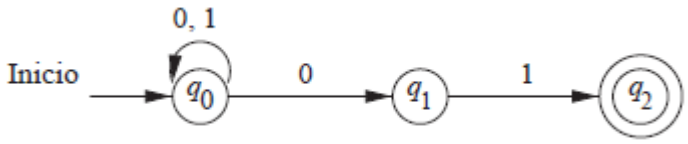
2	dígito
1	+
1	-
2	dígito
2	dígito
3	punto
5	E
8	$

Este es un método para compactar la tabla, el ejemplo solo muestra los punteros para los estados 0, 1 y 2, con el resto se procede de un modo similar.

Autómatas Finitos No Deterministas

Un autómata finito "no determinista" (AFN) tiene la capacidad de estar en varios estados a la vez. Esta capacidad a menudo se expresa como la posibilidad de que el autómata "conjeture" algo acerca de su entrada. Por ejemplo, cuando el autómata se utiliza para buscar determinadas secuencias de caracteres (por ejemplo, palabras clave) dentro de una cadena de texto larga, resulta útil "conjeturar" que estamos al principio de una de estas cadenas y utilizar una secuencia de estados únicamente para comprobar la aparición de la cadena, carácter por carácter.

Al igual que el AFD, un AFN tiene un conjunto finito de estados, un conjunto finito de símbolos de entrada, un estado inicial y un conjunto de estados de aceptación. También dispone de una función de transición, que denominaremos normalmente ∂ . La diferencia entre los AFD y los AFN se encuentra en el tipo de función ∂ . En los AFN, ∂ es una función que toma un estado y símbolos de entrada como argumentos (al igual que la función de transición del AFD), pero devuelve un conjunto de cero, uno o más estados (en lugar de devolver exactamente 1)



La figura muestra un AFN que acepta todas las cadenas que terminan en 01.

Definición de autómata finito no determinista

A continuación presentamos las nociones formales asociadas con los autómatas finitos no deterministas e indicamos las diferencias entre los AFD y AFN. Un AFN se representa esencialmente como un AFD:

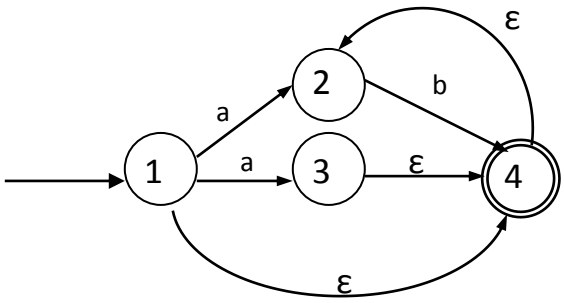
A = (Q, Σ, ∂,q0,F)

donde:

1. Q es un conjunto finito de estados.
2. Σ es un conjunto finito de símbolos de entrada.
3. q0, un elemento de Q, es el estado inicial.
4. F, un subconjunto de Q, es el conjunto de estados finales (o de aceptación).
5. ∂, 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. Observe que la única diferencia entre un AFN y un AFD se encuentra en el tipo de valor que devuelve ∂ : un conjunto de estados en el caso de un AFN y un único estado en el caso de un AFD.

Ejempo de un AFN:

Consideremos el siguiente autómata:



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

a b ε b
→ 1 → 2 → 4 → 2 → 4

a ε ε b ε b
→ 1 → 3 → 4 → 2 → 4 → 2 → 4

Por último, advertimos que 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. Por consiguiente, un programa que simula un AFN debe almacenar transiciones que todavía no se han probado y realizar el retroseguimiento a ellas en caso de falla. Esto es muy similar a los algoritmos que intentan encontrar trayectorias en gráficas dirigidas, sólo que la cadena de entrada guía la búsqueda.
Como los algoritmos que realizan una gran cantidad de retroseguimientos tienden a ser poco eficientes, y un autómata debe ser tan eficiente como sea posible, no describiremos tales algoritmos. En su lugar, se puede resolver el problema de simular un ADN por medio del método que estudiaremos más adelante, el cual convierte un AFN en un AFD.

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. En general, estaremos hablando del conjunto de caracteres ASCII o de algún subconjunto del mismo. En ocasiones el conjunto será más general que el conjunto de caracteres ASCII, en cuyo caso los elementos del conjunto se describirán como símbolos. 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. Distinguiremos el uso de un carácter como patrón escribiendo todo los patrones en negritas. De este modo, **a** es el carácter a usado como patrón.

Por último, 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. En muchas situaciones esto se realiza mediante el uso de un carácter de escape que "desactiva" el significado especial de un metacarácter. Unos caracteres de escape comunes son la diagonal inversa y las comillas. Advierta que los caracteres de escape, si también son caracteres legales en el alfabeto, son por sí mismos metacaracteres.

Definición de expresiones regulares

Ahora estamos en posición de describir el significado de las expresiones regulares al establecer cuáles lenguajes genera cada patrón. Haremos esto en varias etapas.

Comenzaremos por describir el conjunto de expresiones regulares básicas, las cuales se componen de símbolos individuales. Continuaremos con la descripción de las operaciones que generan nuevas expresiones regulares a partir de las ya existentes. Esto es similar a la manera en que se construyen las expresiones aritméticas: las expresiones aritméticas básicas son los números, tales como 43 y 2.5. Entonces las operaciones aritméticas, como la suma y la multiplicación, se pueden utilizar para formar nuevas expresiones a partir de las existentes, como en el caso de 43 * 2.5 y 43 * 2.5 + 1.4. El grupo de expresiones regulares que describiremos aquí es mínimo, ya que sólo contienen los metasímbolos y las operaciones esenciales. Después consideraremos extensiones a este conjunto mínimo.

Expresiones regulares básicas:

Éstas son precisamente 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, es decir, la cadena que no contiene ningún carácter. Utilizaremos el símbolo ε (épsilon) para denotar la cadena vacía, y definiremos el metasímbolo **ε** (e en negritas) estableciendo que L(**ε**) = { ε } .También necesitaremos ocasionalmente ser capaces de describir un símbolo que corresponda a la ausencia de cadenas, es decir, cuyo lenguaje sea el conjunto vacío, el cual escribiremos como { } .Emplearemos para esto el símbolo **ϕ** y escribiremos L{**ϕ**}= { }. Observe la diferencia entre { } y { ε }: el conjunto { } no contiene ninguna cadena, mientras que el conjunto { ε } contiene la cadena simple que no se compone de ningún carácter.

Operaciones de expresiones regulares

Existen tres operaciones básicas en las expresiones regulares: 1) **selección entre alternativas**, la cual se indica mediante el metacarácter **|** (barra vertical); 2) **concatenación**, que se indica mediante yuxtaposición (sin un metacarácter), y 3**) repetición o "cerradura",** la cual se indica mediante el metacarácter *****. Analizaremos cada una por

turno, proporcionando la construcción del conjunto correspondiente para los lenguajes de cadenas concordantes.

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**) ∪ L(**s**). Como un ejemplo simple, considere la expresión regular **a | b**: ésta corresponde tanto al carácter a como al carácter b, es decir, L(**a | b**) = L(**a**) ∪ L(**b**) = {a} ∪ {b} = {a, b}.

Como segundo ejemplo, la expresión regular **a | ε** corresponde tanto al carácter simple a como a la cadena vacía (que no está compuesta por ningún carácter). En otras palabras, L(**a | ε**) = { a, ε }.

La selección se puede extender a más de una alternativa, de manera que, por ejemplo,

L(**a | b | c | d**) = { a, b, c, d}. En ocasiones también escribiremos largas secuencias de selecciones con puntos, como en **a | b | ... | z**, que corresponde a cualquiera de las letras minúsculas de la a a la z.

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. (El uso de los paréntesis como metacaracteres en esta expresión regular se explicará en breve.).

Podemos describir el efecto de la concatenación en términos de lenguajes generados al definir la concatenación de dos conjuntos de cadenas. Dados dos conjuntos de cadenas S_1 y S_2, el conjunto concatenado de cadenas S_1S_2 es el conjunto de cadenas de S_1 complementado con todas las cadenas de S_2. Por ejemplo, si S_1 = {aa, b} y S_2 = {a, bb}, entonces S_1 S_2 = { aaa, aabb, ba, bbb}.

Ahora la operación de concatenación para .expresiones regulares se puede definir como sigue: L(**rs**) = L(**r**)L(**s**). De esta manera (utilizando nuestro ejemplo anterior),

L(**(a | b) c**) = L**(a | b)**L(**c**) = {a, b} {c} = {ac, bc}.

La concatenación también se puede extender a más de dos expresiones regulares:

L($\mathbf{r_1}$ $\mathbf{r_2}$ **...**$\mathbf{r_n}$) = L($\mathbf{r_1}$)L($\mathbf{r_2}$) ...L($\mathbf{r_n}$) = el conjunto de cadenas formado al concatenar todas las cadenas de cada una de las L($\mathbf{r_1}$), ..., L($\mathbf{r_n}$).

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 a, a, aa, aaa, (Concuerda con ε porque ε es la concatenación de ninguna cadena concordante con a.) 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, sea

s* = { ε } ∪ S ∪ SS ∪ SSS ∪...

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...S es la concatenación de S por n veces. (S^0 = { ε }).

Ahora podemos definir la operación de repetición para expresiones regulares como sigue:

L(**r***) = L(r)*

Considere como ejemplo la expresión regular **(a | bb) *** . Esta expresión regular corresponde a cualquiera de las cadenas siguientes: ε, a, bb, aa, abb, bba, bbbb, aaa, aabb y

así sucesivamente. En términos de lenguajes, L(**(a | bb) ***) = L(a | bb)* = { a, bb} * = { ε, a, bb, aa, abb, bba, bbbb, aaa, aabb, abba, abbbb, bbaa, ...}.

Precedencia de operaciones y el uso de los paréntesis

La descripción precedente no toma en cuenta la cuestión de la precedencia de las operaciones de **elección, concatenación y repetición**. Por ejemplo, dada la expresión regular **a | b *** , ¿ deberíamos interpretar esto como **(a | b) *** o como **a | (b*)** ? (Existe una diferencia importante, puesto que L(**(a | b) ***) = { ε, a, b, aa, ab, ba, bb, ...}, mientras que L(**a | (b*)**) = { ε, a, b, bb, bbb, .. .}.) La convención estándar es que la repetición debería tener mayor precedencia, por lo tanto, la segunda interpretación es la correcta. En realidad, 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. De este modo, 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. Ésta es la razón por la que tuvimos que escribir **(a | b) c** para indicar que la operación de elección debería tener mayor precedencia que la concatenación, ya que de otro modo **a | bc** se interpretaría como si correspondiera tanto a **a** como a **bc**.

De manera similar, **(a | bb) *** se interpretaría sin los paréntesis como **a | bb*** , lo que corresponde a a, b, bb, bbb, ...

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**.

El uso de una definición regular es muy conveniente, pero introduce la complicación agregada de que el nombre mismo se convierta en un metasímbolo y se deba encontrar un significado para distinguirlo de la concatenación de sus caracteres. En nuestro caso hicimos esa distinción al utilizar letra cursiva para el nombre. Advierta que no se debe emplear el nombre del término en su propia definición (es decir, de manera recursiva): debemos poder eliminar nombres reemplazándolos sucesivamente con las expresiones regulares para las que se establecieron. Antes de considerar una serie de ejemplos para elaborar nuestra definición de expresiones regulares, reuniremos todas las piezas de la definición de una expresión regular.

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; el metacarácter **ε**; o el metacarácter **Φ**. En el primer caso, L(**a**) = {a}; en el segundo, L(**ε**) = { ε }; en el tercero, L(**Φ**) = { }.
2. Una expresión de la forma **r | s**, donde **r** y **s** son expresiones regulares. En este caso, L(**r | s**) = L(**r**) ∪ 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.

Advertimos que, en esta definición, la precedencia de las operaciones en (2), (3) y (4) está en el orden inverso de su enumeración; es decir, | tiene precedencia más baja que la concatenación, y ésta tiene una precedencia más baja que el asterisco * .También advertimos que esta definición proporciona un significado de metacarácter a los seis símbolos **Φ, ε, |,*, (,)** .

En lo que resta de esta sección consideraremos una serie de ejemplos diseñados para explicar con más detalles la definición que acabamos de dar. Éstos son algo artificiales, ya que por lo general no aparecen como descripciones de token en un lenguaje de programación.

En los ejemplos siguientes por lo regular se incluye una descripción en idioma coloquial de las cadenas que se harán corresponder, y la tarea es traducir la descripción a una expresión regular. Esta situación, en la que un manual de lenguaje contiene descripciones de los tokens, es la que con más frecuencia enfrentan quienes escriben compiladores. En ocasiones puede ser necesario invertir la dirección, es decir, desplazarse de una expresión regular hacia una descripción en lenguaje coloquial, de modo que también incluiremos algunos ejercicios de esta clase.

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 | c)* b (a | c)*

Advierta 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 | c) *** como la otra alternativa (definiendo los casos sin b en todo). De este modo, tenemos la solución siguiente:

(a | c)* | (a | c)* b (a | c)*

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

(a | c)* (b | ε) (a | 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, aunque nunca intentaremos demostrar que encontramos, de hecho, la "más simple": la más breve por ejemplo. Existen dos razones para esto. La primera es que raramente se presenta en situaciones prácticas, donde por lo regular existe una solución estándar "más simple". La segunda es que cuando estudiemos métodos para reconocer expresiones regulares, los algoritmos tendrán que poder simplificar el proceso de reconocimiento sin molestarse en simplificar primero la expresión regular.

Ejemplo 3: Consideremos el conjunto de cadenas S sobre el alfabeto Σ = { a, b} compuesto de una b simple rodeada por el mismo número de a:

$S = \{b, aba, aabaa, aaabaaa, ...\} = \{a_n\ b\ a_n \mid n \neq 0\}$

Este conjunto no se puede describir mediante una expresión regular. La razón es que 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".**

Sin embargo, para proporcionar una demostración matemática de este hecho, requeriríamos utilizar un famoso teorema acerca de las expresiones regulares conocido como lema de la bombeo (pumping lemma), que se estudia en la teoría de autómatas y que aquí solo lo vamos a enunciar, sin demostrar.

El lema de bombeo para lenguajes regulares (pumping lemma)

Sea L un lenguaje regular. Existe entonces una constante n (que depende de L) tal que para toda cadena *w* perteneciente a L con $|w| \geq n$, podemos descomponer *w* en tres cadenas, *w* = **xyz**, tales que:

1. $y \neq \varepsilon$.
2. $|xy| \leq n$.
3. Para todo $k \geq 0$, la cadena $xy^k z$ también pertenece a L.

Es decir, siempre podemos hallar una cadena no vacía y no demasiado alejada del principio de *w* que pueda "bombearse"; es decir, si se repite y cualquier número de veces, o se borra (el caso en que k = 0), la cadena resultante también pertenece al lenguaje L.
(la demostración de este lema puede encontrarse en la bibliografía citada).

Es evidente que no todos los conjuntos de cadenas que podemos describir en términos simples se pueden generar mediante expresiones regulares. Por consiguiente, un conjunto de cadenas que es el lenguaje para una expresión regular se distingue de otros conjuntos al denominarlo **conjunto regular**. De cuando en cuando aparecen conjuntos no regulares como cadenas en lenguajes de programación que necesitan ser reconocidos por un analizador léxico, los cuales por lo regular son abortados cuando surgen.

Ejemplo 4: Consideremos las cadenas en el alfabeto $\Sigma = \{ a, b, c\}$ que no contienen dos b consecutivas. De modo que, entre cualesquiera dos b, debe haber por la menos una **a** o una **c**. Construiremos una expresión regular para este conjunto en varias etapas. En primer lugar, podemos obligar a que una **a** o **c** se presenten después de cualquier **b** al escribir:

(b (a | c))*

Podemos combinar esto con la expresión **(a | c) *** , la cual define cadenas que no tiene **b**, y escribimos:

(a | c)* | (b (a | c))*)*

advirtiendo que **(r* | s*)*** corresponde con las mismas cadenas que **(r | s)***

((a | c) | (b (a | c)))*

o:

(a | c | ba | bc)*

Ésta todavía no es la respuesta correcta. El lenguaje generado por esta expresión regular tiene, en realidad, la propiedad que buscamos, a saber, que no haya dos **b** consecutivas (pero aún no es lo bastante correcta). En ocasiones podremos demostrar tales aseveraciones, así que esbozaremos una demostración de que todas las cadenas en L(**(a | c | ba | bc)***) no contienen dos b consecutivas. La demostración es por inducción sobre la longitud de la cadena (es decir, el número de caracteres en la cadena).
Es evidente que esto es verdadero para todas las cadenas de longitud 0, 1 o 2: estas cadenas son precisamente las cadenas: **ε, a, c, aa, ac, ca, cc, ba, bc**.

Ahora supongamos que es verdadero para todas las cadenas en el lenguaje con una longitud de i < n, y sea s una cadena en el lenguaje con longitud n > 2. Entonces, s contiene más de una de las cadenas no-ε anteriormente enumeradas, de modo que S = S_1S_2, donde S_1 y S_2

también se encuentran en el lenguaje y no son **ε**. Por lo tanto, mediante la hipótesis de inducción, tanto S1 como S2 no tienen dos b consecutivas. Por consiguiente, la única manera de que s misma pudiera tener dos **b** consecutivas sería que S_1 finalizara con una b y que S_2 comenzara con una **b**. Pero esto es imposible, porque ninguna cadena en el lenguaje puede finalizar con una **b**.

Este último hecho que utilizamos en el esbozo de la demostración (o sea, que ninguna cadena generada mediante la expresión regular anterior puede finalizar con una **b**) también muestra por qué nuestra solución todavía no es bastante correcta: no genera las cadenas **b, ab y cb**, que no contienen dos **b** consecutivas. Arreglaremos esto agregando una **b** opcional rezagada de la manera siguiente:

(a | c | ba | bc)* (b | ε)

Advierta que la imagen especular de esta expresión regular también genera el lenguaje

(b | ε) (a | c | ba | bc)*

También podríamos generar este mismo lenguaje escribiendo

(not b | b not b)* (b | ε) donde **not b** = **a | c.**

Éste es un ejemplo del uso de un nombre para una subexpesión. De hecho, esta solución es preferible en los casos en que el alfabeto es grande, puesto que la definición de **not b** se puede ajustar para incluir todos caracteres excepto b, sin complicar la expresión original.

Extensión para las expresiones regulares

Pueden considerarse las siguientes extensiones para las expresiones regulares:

Una o más repeticiones

Dada una expresión regular **r**, la repetición de **r** se describe utilizando la operación de cerradura estándar, que se escribe **r*** . Esto permite que **r** se repita 0 o más veces.

Una situación típica que surge es la necesidad de una o más repeticiones en lugar de ninguna, la que garantiza que aparece por lo menos una cadena correspondiente a **r**, y no permite la cadena vacía **ε**.

Un ejemplo es el de un número natural, donde queremos una secuencia de dígitos, pero deseamos que por la menos aparezca uno. Por ejemplo, si deseamos definir números binarios, podríamos escribir **(0 | 1) *** , pero esto también coincidirá con la cadena vacía, la cual no es un número. Por supuesto, podríamos escribir:

(0 | 1) (0 | 1)*

pero esta situación se presenta con tanta frecuencia que se desarrolló para ella una notación relativamente estándar en la que se utiliza **+** en lugar de *: **r+,** que indica una o más repeticiones de **r**.

De este modo, nuestra expresión regular anterior para números binarios puede escribirse ahora como $\mathbf{(0 \mid 1)^+}$

Cualquier carácter

Una situación común es la necesidad de generar cualquier carácter en el alfabeto. Sin una operación especial esto requiere que todo carácter en el alfabeto sea enumerado en una

alternativa. Un metacaracter típico que se utiliza para expresar una concordancia de cualquier carácter es el punto **"."**, el cual no requiere que el alfabeto se escriba realmente en forma extendida. Con este metacaracter podemos escribir una expresión regular para todas las cadenas que contengan al menos una **b** como se muestra a continuación:

.*b.*

Un intervalo de caracteres

A menudo necesitamos escribir un intervalo de caracteres, como el de todas las letras minúsculas o el de todos los dígitos.

Hasta ahora hemos hecho esto utilizando la notación **a | b |. ..| z** para las letras minúsculas o **0 | 1 | ...| 9** para los dígitos. Una alternativa es tener una notación especial para esta situación, y una que es común es la de emplear corchetes y un guión, como en **[a- z]** para las letras minúsculas y **[0- 9]** para los dígitos.

Esto también se puede emplear para alternativas individuales, de modo que **a | b | c** puede escribirse como **[abc]** .

También se pueden incluir los intervalos múltiples, de manera que **[a-z A-Z]** representa todas las letras minúsculas y mayúsculas.

Esta notación general se conoce como clases de caracteres. Advierta que esta notación puede depender del orden subyacente del conjunto de caracteres. Por ejemplo, al escribir **[A-Z]** se supone que los caracteres B, C, y los demás vienen entre los caracteres A y Z (una suposición razonable) y que sólo los caracteres en mayúsculas están entre A y Z (algo que es verdadero para el conjunto de caracteres ASCII). Sin embargo, al escribir **[A-z]** no se definirán los mismos caracteres que para **[A-Za-z]** , incluso en el caso del conjunto de caracteres ASCII.

Cualquier carácter que no esté en un conjunto dado

Como hemos visto, a menudo es de utilidad poder excluir un carácter simple del conjunto de caracteres por generar. Esto se puede conseguir al diseñar un metacarácter para indicar la operación de negación **("not")** o complementaria sobre un conjunto de alternativas. Por ejemplo, un carácter estándar que representa la negación en lógica es la **"tilde" ~,** y

podríamos escribir una expresión regular para un carácter en el alfabeto que no sea "a" como **~a** y un carácter que no sea a, ni b, ni c, como

~ (a | b | c)

Una alternativa para esta notación se emplea en Lex, donde el carácter **"^"** se utiliza en conjunto con las clases de caracteres que acabamos de describir para la formación de complementos. Por ejemplo, cualquier carácter que no sea a se escribe como **[^ a]** , mientras que cualquier carácter que no sea a, ni b ni c se escribe como **[^abc]**

Subexpresiones opcionales

Por último, un suceso que se presenta comúnmente es el de cadenas que contienen partes opcionales que pueden o no aparecer en cualquier cadena en particular. Por ejemplo, un número puede o no tener un signo inicial, tal como + o -.

Podemos emplear alternativas para expresar esto como en las definiciones regulares:

natural = [0-9]$^{+}$

naturalconSigno = natural | + natural | -natural

Esto se puede convertir rápidamente en algo voluminoso, e introduciremos el metacarácter de signo de interrogación **r?** para indicar que las cadenas que coincidan con **r** son opcionales (o que están presentes 0 o 1 copias de **r**). De este modo, el ejemplo del signo inicial se convierte en:

natural = [0-9]$^{+}$

naturalconSigno = (+|-)? natural

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
\w	un caracter alfanumérico (incluye "_").
\W	un caracter no alfanumérico.
\d	un caracter numérico.
\D	un caracter no numérico.

Metacaracter	Descripción
\s	cualquier espacio (lo mismo que [\t\n\r\f]).
\S	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 Phyton en:

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

Un resumen por parte de Google Eduactión:

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/

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

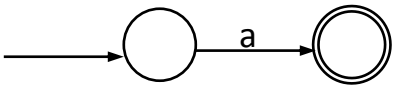
directa en un AFD, pero son más complejos y la construcción intermedia también es de cierto interés. Así que nos concentraremos en la descripción de dos algoritmos, uno que traduce una expresión regular en un AFN y el segundo que traduce un AFN en un AFD.

La construcción que describiremos se conoce como la construcción de ***Thompson***, en honor a su inventor. 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).

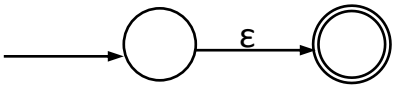
Expresiones regulares básicas

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:

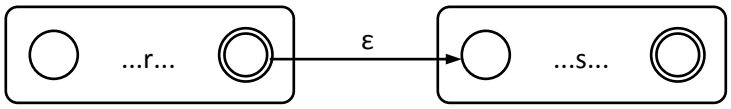


Concatenación:

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:

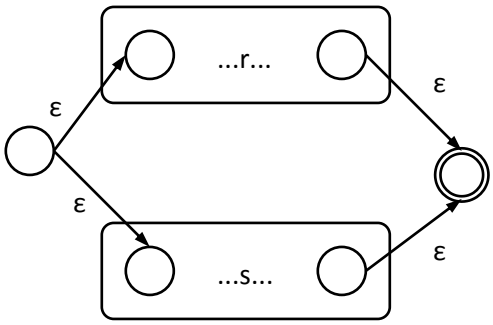


Conectamos el estado de aceptación de la máquina **r** al estado de inicio de la máquina s mediante una transición **ε**.

La nueva máquina tiene como estado de inicio el de **r** y de aceptación el de **s**, esta máquina acepta el lenguaje L(**rs**) = L(**r**)L(**s**) de modo de corresponder a la expresión regular **rs**.

Selección de alternativas

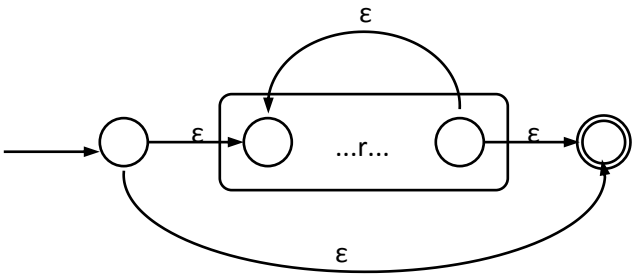
Para la expresión regular ***r*|*s*** bajo las mismas suposiciones que antes, podemos hacer:



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***:



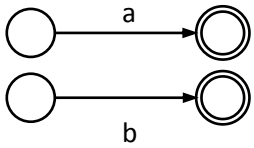
Nuevamente se agregan nuevos estados de inicio y aceptación, la repetición en esta máquina la proporciona la arista ε que une el estado final con el inicial de la máquina **r**.
Para asegurar que ε también es aceptada, se traza una transición ε entre el estado inicial y final de la máquina **r*.**

Ejemplo 1:

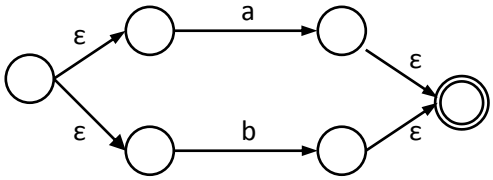
Para la expresión regular: **(a | b)* abb** se tiene:

Para **a**:

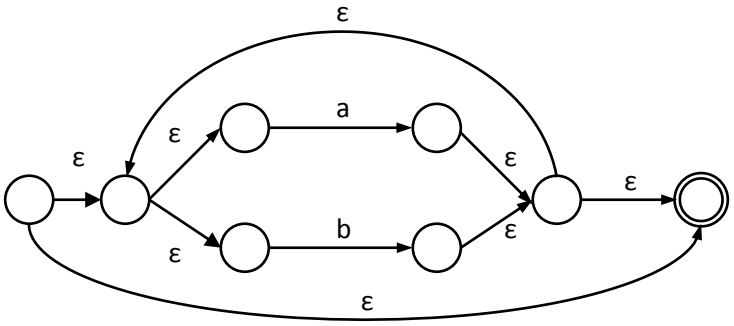
Para **b**:



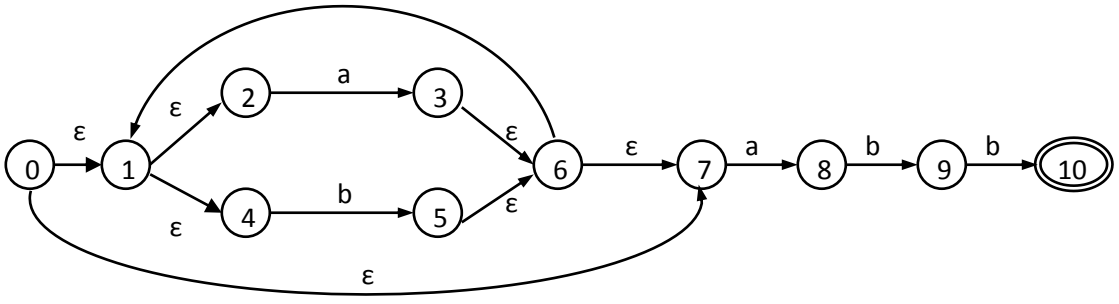
Para **a | b** :



Para **(a | b)*** :



Finalmente para **(a | 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.

Cerraduras ε de un conjunto de estados

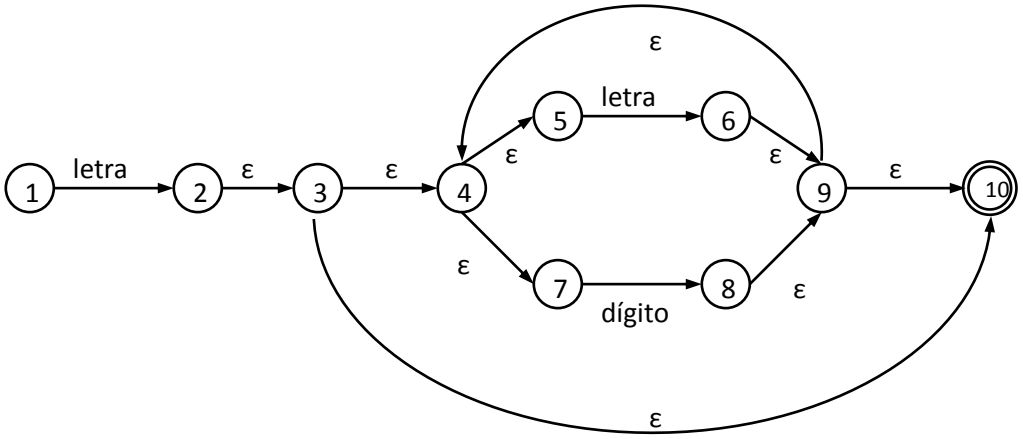
Definimos la **cerradura ε** de un estado simple s como el conjunto de estados alcanzables por una serie de cero o más transiciones ε, y escribimos este conjunto como $\bar{S}$. La cerradura a de un estado siempre contiene al estado mismo, y será un estado del nuevo AFD

Tra**nsiciones desde un conjunto de estados**

Definimos la función de transición de un conjunto de estados $\overline{S_1}$ como la función τ_i que produce la transición del conjunto $\overline{S_1}$ a un conjunto de estados $\overline{S_2}$ para cada símbolo i del alfabeto $\sum \quad \tau_i(S_1, i) = S_2$

Los conjuntos de estados del AFN que incluyen al estado final, serán estados finales del AFD a construir.

Veamos por 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

τ_{letra}(A , $letra$) = {4, 5, 6, 7, 9, 10} = B — B es otro estado del AFD

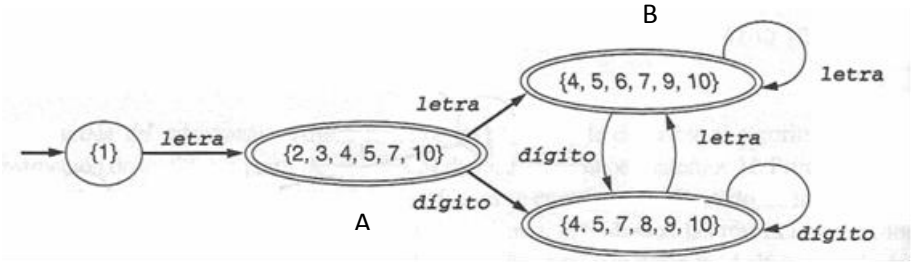
τ_{digito}(A , $digito$) = {4, 5, 7, 8, 9, 10} = C — C es otro estado del AFD

Se aplica la función de transición a los nuevos estados:

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

τ_{digito}(C , $digito$) = {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:



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

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*.