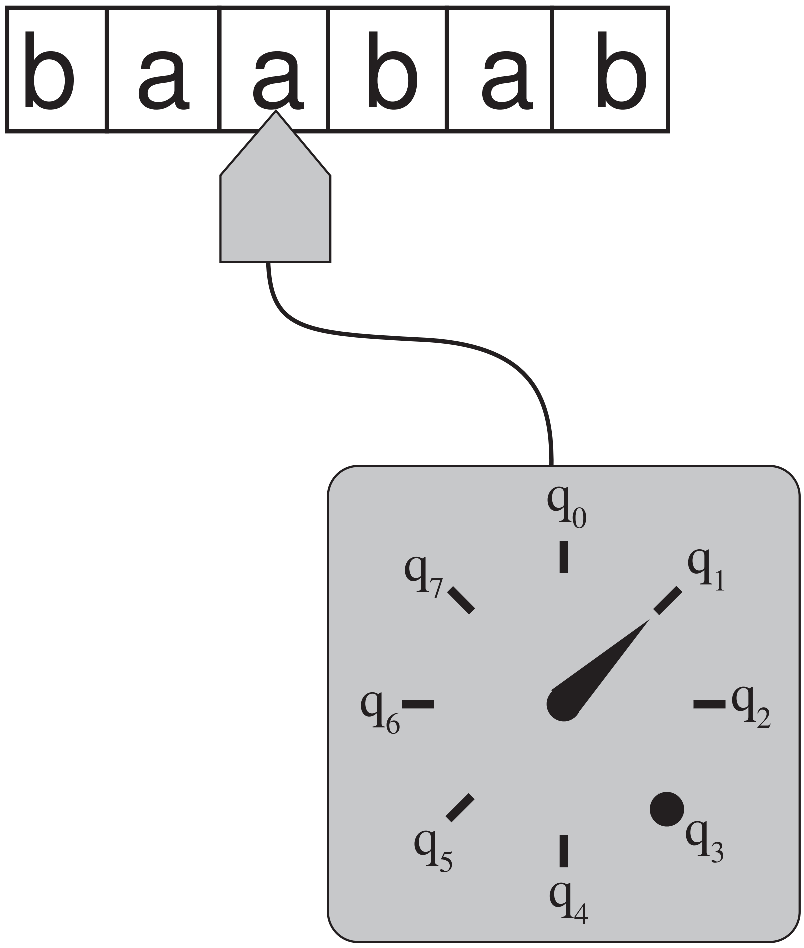
LENGUAJES FORMALES Y AUTOMATAS

Incluye Redes de Petri

Dr. Ramón Brena Pinero

Centro de Inteligencia Artificial
Instituto Tecnológico y de Estudios Superiores
de Monterrey Campus Monterrey

Enero de 1997


b
a
a
b
a
b
q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7

Prefacio

En años reciente se ha visto la aparición de un buen número de textos en el tema de Lenguajes Formales y Autómatas (Ver al final referencias [5], [3], [10], [4], [2], etc.). Por una parte, esto indica la importancia y riqueza que el tema tiene; por otra, ante tal variedad de oferta todo nuevo libro en el área requiere una justificación, un aporte con respecto a lo existente.

Este texto se sitúa en una generación de textos que tratan de poner el estudio de los lenguajes formales y aotómatas al alcance de estudiantes que no necesariamente son avezados matemáticos buscando establecer nuevos teoremas, sino que buscan una iniciación a estos temas, que además les sirva como un ejercicio en el arte de formalizar, en particular en nociones relacionadas con la computación. Entre estos textos "accesibles", encontramos, por ejemplo, a [10]. Estos nuevos textos han reemplazado en muchas universidades a los "clásicos" [3] y aún [5] -que ya era más accesible-, y han permitido que la teoría de la computación se estudie a nivel profesional en carreras relacionadas con computación y matemáticas.

El presente libro es resultado de una experiencia de impartir el curso de Teoría de la Computación por más de 10 semestres en el ITESM, Monterrey. Durante este lapso, aunque ciertamente se fué enriqueciendo el contenido técnico, el principal refinamiento consistió en ir detectando cuidadosamente las dificultades principales a las que se enfrentaban los estudiantes, para poder estructurar y presentar el material de forma que aquellos estuvieran en condiciones de "digerirlo" de manera constructiva. Aqui el énfasis no es tanto en hacer el curso "mas fácil" para el estudiante, sino en darle los elementos para ser más eficaz como aprendiz. La teoría educativa que sustenta esta forma de trabajo es la del "andamiaje", que, como el nombre sugiere, provee al estudiante una estructura de apoyo para que *él mismo* vaya reconstruyendo en su cabeza la red de conceptos y relaciones que caracterizan al área.

Este texto es aplicable tanto al nivel de maestría en computación o equivalente, como a clases de nivel profesional (licenciaturas, ingenierías). De hecho en el ITESM se ha aplicado en ambos niveles. Las secciones cuyo nivel es propiamente de maestría son identificadas por medio de un signo "$*$".

En breve, los puntos que caracterizan a este libro, y que en cierta medida lo hacen particular, son:

- La presentación didáctica ha sido -en nuestra opinión- más pulida que en la mayoría de textos en Teoría de la Computación.
- Se cubre el tema de las *Redes de Petri*, tema íntimamente relacionado con los autómatas, y que ha adquirido una gran importancia en aplicaciones de manufactura, Sistemas Operativos y otras áreas.
- Los algoritmos no se presentan en forma de "seudocódigo", es decir, usando estructuras de control de lenguajes imperativos (p.ej. *while*, *for*, etc.), sino que se trata de dar una

interpretación intuitiva de los resultados intermedios obtenidos por los algoritmos. En efecto, opinamos que el código y seudocódigo están bien para ser entendidos por un compilador, pero no por un humano.

- ¡El texto está en español en el original! (es decir, no se trata de una traducción de un texto en inglés). Las traducciones son muchas veces desventajosas respecto al original.

El interés de este curso es tanto teórico como práctico. En lo que respecta a la teoría, se trata de un curso extraordinariamente formativo, una oportunidad de ejercitar la rigurosidad matemática y la precisión de pensamiento.

Adicionalmente, el interés teórico de los temas que cubriremos viene del hecho de que escudriñaremos los límites extremos de varios modelos de máquinas abstractas, y nos interesaremos por saber si cierto problema puede o no ser resuelto por un tipo de máquina dado. Es muy diferente sospechar que cierta máquina no es suficientemente poderosa para llevar a cabo una tarea, que demostrarlo rigurosamente.

El interés práctico del curso viene del hecho de que los modelos de máquinas que veremos son efectivamente aplicables a la práctica, desde el diseño de controladores lógicos programables (PLC), usados ampliamente en la industria, hasta verificación de protocolos de comunicación entre componentes digitales.

La estructura de estos apuntes es la siguiente: en los primeros capítulos (2-4) veremos la clase más simple de lenguajes, los *lenguajes regulares*, junto con las máquinas abstractas que les corresponden –los *Autómatas Finitos*–, y al mismo tiempo introduciremos una metodología de análisis de las máquinas abstractas y de los lenguajes, metodología que volveremos a utilizar en las siguientes secciones del curso, para otros tipos de lenguajes y de máquinas. Este estudio se complementa en el capítulo 5, con un tipo de autómatas, las *Redes de Petri*, que se relacionan con los autómatas finitos, y que tienen una gran aplicación en la industria.

En los capítulos 6 y 7 veremos los *lenguajes libres de contexto* y los *Autómatas de Pila*.

Finalmente, a partir del capítulo 8 estudiaremos el tipo de máquinas más poderoso, las *Máquinas de Turing*, que son en cierta forma el límite teórico de lo que es posible de hacer con máquinas procesadoras de información.

Agradezco la colaboración de Leonardo Garrido en la recopilación de ejercicios. También agradezco al Comité del Fondo de Apoyo a Proyectos en Didáctica su apoyo financiero. Finalmente doy las gracias a muchos alumnos que ayudaron a depurar el escrito mientras sirvió como apuntes de la materia.

Índice General

Capítulo 1

Preliminares

En esta parte repasaremos brevemente algunas nociones y notaciones que serán necesarias para comprender adecuadamente el resto del material de este libro. Debe, sin embargo, quedar claro que este repaso queda fuera del área de autómatas y lenguajes formales. Por otra parte, no es nuestra intención hacer una introducción para un lector que no tenga ninguna base en matemática, especialmente en teoría de conjuntos, sino que únicamente haremos un *repaso*, ayudando al lector a detectar sus puntos débiles, además de recordar nociones que pueden no estar frescas. Un objetivo adicional del presente capítulo es uniformizar la notación, que varía bastante de libro a libro. Para los lectores que requieran una introducción más exhaustiva a la teoría de conjuntos y temas afines, recomendamos textos como [8].

1.1 Conjuntos

El fundamento más importante para el estudio de los lenguajes y autómatas es la *Teoría de Conjuntos*. En efecto, siempre que hablemos de "formalizar" una noción, estaremos diciendo en realidad "expresar en términos de la Teoría de Conjuntos".

La idea de un conjunto como una colección de individuos u objetos no es, para un verdadero matemático, suficientemente precisa, y se parece a la noción de clase; sin embargo, para nuestros propósitos es suficiente.

Un conjunto que vamos a utilizar con frecuencia es el de los números naturales, $\{1, 2, 3, \ldots\}$, denotado por $\aleph$.

Los conjuntos pueden expresarse de dos maneras básicamente:

- En *extensión*, lo cual quiere decir que citamos explícitamente cada uno de sus elementos, como en el conjunto $\{1, 3, 5\}$ que contiene exactamente los números 1, 3 y 5.

- En *intención*, dando una descripción precisa de los elementos que forman parte del conjunto, en vez de citarlos explícitamente. Por ejemplo, el conjunto del punto anterior puede ser visto como $\{i \in \aleph | impar(i), i < 6\}$, donde se supone que los números impares cumplen la condición $impar(i)$.

Representamos a los conjuntos con letras mayúsculas, como en $A = \{2, 4\}$. Los conjuntos pueden contener conjuntos como elementos, como en $B = \{\{a\}.\{b, c\}\}$. El conjunto sin elementos (vacío) se representa por $\emptyset$ o bien por $\{\}$.

La notación $a \in B$ significa que a es *elemento* o está contenido en el conjunto B; por ejemplo, $\{2, 3\} \in \{1, \{2, 3\}, 4\}$. Para indicar que a no está en B se escribe $a \notin B$.

Dos conjuntos A y B son *iguales*, $A = B$, si y sólo si tienen los mismos elementos, esto es, $x \in A$ ssi $x \in B$. [1] Por ejemplo, $\{1, \{2, 3\}\} = \{\{3, 2\}, 1\}$; vemos que en los conjuntos el orden de los elementos es irrelevante.

Similarmente, $A \subseteq B$ significa que el conjunto A está "contenido" en el conjunto B, o más técnicamente, que A es *subconjunto* de B. En otras palabras, $A \subseteq B$ ssi $x \in A$ implica $x \in B$. Obsérvese que de acuerdo con esta definición, $A \subseteq A$ para cualquier conjunto A.

Frecuentemente, para probar que $A \subseteq B$ suponemos un elemento x que está contenido en A, y probamos que $x \in B$. Por ejemplo, tratamos de probar que si $A \subseteq B$ y $B \subseteq C$ entonces $A \subseteq C$. Sea x un elemento de A. De acuerdo con la definición de subconjunto, esto implica que $x \in B$. Ahora bien, como $x \in B$, dado que $B \subseteq C$, tenemos que $x \in C$. Resumiendo, $x \in A$ implica $x \in C$, esto es, $A \subseteq C$, QED. [2]

Para indicar que un subconjunto contiene menos elementos que otro, es decir, que es un *subconjunto propio* de éste, se escribe $A \subset B$.

Claramente, $A = B$ ssi $A \subseteq B$ y $B \subseteq A$. Para probar que dos conjuntos A y B son iguales muchas veces es conveniente dividir la prueba en dos partes: primero probar $A \subseteq B$ y luego $B \subseteq A$.

Sean A y B conjuntos. Se definen las siguientes operaciones con los conjuntos:

Unión de conjuntos, denotada por $A \cup B$, que contiene los elementos del conjunto A y los del conjunto B, es decir, $A \cup B = \{x | x \in A \text{ } o \text{ } x \in B\}$. Por ejemplo, $\{1, 2, 3\} \cup \{3, 4\} = \{1, 2, 3, 4, 5\}$.

Intersección de conjuntos, escrita $A \cap B$, que contiene los elementos que pertenecen simultáneamente al conjunto A y al conjunto B, es decir, $A \cap B = \{x | x \in A \text{ } y \text{ } x \in B\}$. Por ejemplo, $\{1, 2, 3\} \cap \{3, 4\} = \{3\}$.

[1] "ssi" se lee "si y sólo si".
[2] "QED" significa en latín "lo cual debía demostrarse".

Diferencia de conjuntos, $A - B$, que contiene los elementos de A que no están en B, esto es, $A - B = \{x | x \in A \ y \ x \notin B\}$. Por ejemplo, $\{1, 2, 3\} - \{3, 4\} = \{1, 2\}$.

Complemento de un conjunto, es un caso particular de la diferencia, cuando el primer conjunto es considerado como el "universo" que contiene todos los elementos posibles. Sea U un universo, entonces el complemento del conjunto A, denotada por A^c contiene los elementos del universo que no están en A. Por ejemplo, si el universo son los números naturales $\{1, 2, 3, \ldots\}$, complemento de los números pares son los números nones: $\{2, 4, 6, \ldots\}^c = \{1, 3, 5, \ldots\}$. Claramente $A \cup A^c = U$, para todo conjunto A; además, $A \cap A^c = \emptyset$.

Potencia de un conjunto A, denotada como 2^A, contiene como elementos a todos los subconjuntos de A, esto es, $2^A = \{x | x \subseteq A\}$. En otras palabras, 2^A es un conjunto de conjuntos. Por ejemplo, $2^{\{1,2,3\}} = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$ La notación "2^A" recuerda que el tamaño del conjunto potencia de A es 2 elevado a la potencia del tamaño de A, esto es, $|2^A| = 2^{|A|}$.

Producto Cartesiano de dos conjuntos, $A \times B$, es el conjunto de pares ordenados (a, b) tales que $a \in A$ y $b \in B$. Por ejemplo,

$$\{1, 2\} \times \{3, 4, 5\} = \{(1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5)\}$$

.

Se llama *relación* a todo subconjunto de un producto cartesiano; por ejemplo la relación "$\leq$" contiene los pares de números naturales tales que el primer componente es menor o igual al segundo, esto es, $\leq = \{(1, 1), (1, 2), (1, 3), (2, 3), \ldots\}$. Un caso particular de las relaciones son las *funciones*, que son relaciones en que no hay dos pares ordenados que tengan el mismo primer componente. Escribimos $f : A \to B$ para indicar que si $(a, b) \in f$ entonces $a \in A$ y $b \in B$; decimos que A es el *dominio* de la función y B es el *codominio*. Una función $f : A \to B$ puede verse como un *mapeo* que relaciona cada elemento del dominio A con un elemento del codominio B. Por ejemplo, la función $cuadrado : \aleph \to \aleph$ relaciona cada número natural con su cuadrado, es decir, $cuadrado = \{(1, 1), (2, 4), (3, 9), \ldots\}$.

1.1.1 Conjuntos infinitos

Además de los conjuntos "finitos" –esto es, con un número de elementos determinado– también puede haber conjuntos infinitos, cuyo tamaño no puede expresarse con un número; un ejemplo es el conjunto de los números naturales $\{1, 2, 3, \ldots\}$. Aún a estos conjuntos pueden aplicarse todas las operaciones antes descritas.

Sin embargo, la comparación de tamaños de conjuntos infinitos no es tan simple como en el caso de los conjuntos finitos, pues no se puede expresar su tamaño como un número. En estos casos se aplica lo que se conoce como "el principio del palomar", que sirve para comprobar si dos conjuntos tienen o no el mismo tamaño. Supóngase que se quiere comprobar si en un palomar la cantidad de palomas con que se cuenta es mayor, menor o igual a la cantidad de lugares disponibles en el palomar. Una manera simple de verificarlo es asignar a cada una de las palomas un sitio disponible, y si es posible hacerlo para todas las palomas, se sabe que no hay más palomas que lugares. Similarmente se puede ver si no hay más lugares que palomas. Así verificamos que el conjunto de palomas tiene el mismo tamaño que el de lugares disponibles.

Esta simple idea puede aplicarse para comparar el tamaño de conjuntos infinitos. Así se puede verificar, por ejemplo, que el conjunto de los pares tiene el mismo tamaño que el de los naturales, un resultado difícil de aceptar intuitivamente. En efecto, sean $\aleph$ y $\mathcal{P}$ los naturales y los pares, respectivamente. Es fácil ver que $|\mathcal{P}| \leq |\aleph|$, pero es mucho menos evidente que $|\aleph| \leq |\mathcal{P}|$, cosa que vamos a mostrar usando el principio del palomar. A cada número natural le debemos poder asignar un número par distinto; esto se puede hacer de muchas maneras, pero una muy simple consiste en asignar a cada número el doble de sí mismo; por ejemplo, al 7 le asignamos el par 14, etc. Como esto se puede hacer para todos los números, y no va a haber dos números que compartan el mismo par, concluímos que no hay más números naturales que pares.

Definición.- Un conjunto infinito es *contable* cuando sus elementos pueden ponerse "en una fila", o dicho de una manera mas técnica, cuando sus elementos pueden ponerse en correspondencia uno a uno con los números naturales. En otras palabras, los conjuntos contables tienen el mismo tamaño que el conjunto de los números naturales.

Otro ejemplo de conjunto infinito contable es el conjunto de pares de números, esto es,

$$\aleph \times \aleph = \{(1,1),(2,1),(1,2),(1,3),(2,2),(3,1),(4,1),\ldots\}$$

(La prueba de que es contable se deja como ejercicio).

Aunque resulte increíble, hay conjuntos infinitos "más grandes" que los conjuntos contables, en el sentido de que no van a alcanzar los elementos del conjunto contable para asignar uno a cada elemento del conjunto "grande". A estos conjuntos se les llama *incontables*.

Un ejemplo de conjunto incontable es $2^{\aleph}$, esto es, el conjunto potencia de los naturales; el llamado "teorema de Kantor" establece este hecho.

La prueba del teorema de Kantor es muy simple y se basa en empezar suponiendo que $2^{\aleph}$ sí es contable, y se llega a una contradicción, concluyendo entonces que $2^{\aleph}$ en realidad es incontable.

En efecto, si $2^{\aleph}$ es contable, sus elementos pueden ser puestos en una sucesión como sigue:

$$2^{\aleph} = \{S_1, S_2, S_3, \ldots\}$$

Supóngase ahora el conjunto $D = \{n \in \aleph | n \notin S_n\}$, que está formado por aquellos números n que no aparecen en el conjunto S_n que les corresponde. Como por hipótesis todos los subconjuntos de los naturales fueron puestos en la sucesión $S_1, S_2, \ldots$, tenemos que el conjunto

D, –que está formado de naturales– debe hallarse en dicha sucesión, es decir, debe ser igual a S_k para una cierta k. Ahora nos preguntamos si k aparece o no en el conjunto D:

- Si la respuesta es afirmativa, entonces, por la definición de D, tenemos que $k \notin S_k$, lo que es una contradicción;
- Si la respuesta es negativa, entonces, por la definición de D, $k \in S_k$, lo que también es una contradicción.

Concluímos que $2^{\aleph}$ es incontable.

Aún dentro de los conjuntos incontables hay unos conjuntos "mas grandes" que otros. En efecto, se sabe que para todo conjunto infinito A, se tiene que $|A| < |2^A|$, por lo que hay toda una jerarquía de "infinitos":

$$|\aleph| < |2^{\aleph}| < |2^{2^{\aleph}}| < \dots$$

1.2 Pruebas por inducción

Una forma de prueba que utilizaremos repetidamente en este texto es la prueba *por inducción*. Sirve para probar que una cierta propiedad es válida para todos los elementos de un conjunto infinito contable. Su estructura se entenderá mejor a partir de un ejemplo:

Supongamos que queremos probar que todo número natural es menor que el doble de sí mismo, esto es, $n < 2n,\ n \in \aleph$. Lo hacemos en dos pasos:

(base) Primero comprobamos que para el caso del 1 se cumple, pues $1 < 2$.

(inducción) Ahora, suponiendo que para un número i la propiedad se cumple, esto es, $i < 2i$, debemos comprobar que también se cumple para el siguiente número, esto es, $i+1 < 2(i+1)$. En efecto, si $i < 2i$, entonces $i+1 < 2i+1$, pero $2i+1 < 2i+2 = 2(i+1)$, por lo que $i+1 < 2(i+1)$, como debíamos probar.

Una vez que tenemos la "base" y la "inducción" podemos estar seguros de que todo número cumple la propiedad, pues el 1 la cumple (por la base), el 2 la cumple, dado que el 1 la cumple y la inducción; el 3 la cumple, dado que el 2 la cumple y la inducción, y así en adelante.

Las pruebas por inducción no siempre son para probar propiedades de los números naturales, y existen variantes, como tener varias "bases". No entraremos aqui en detalles, postergando su estudio para las secciones donde se le utiliza directamente.

Parte I

Lenguajes regulares y sus máquinas

Capítulo 2

Lenguajes Regulares

Uno de los primeros conceptos que necesitamos es el de Lenguaje. Para llegar a este concepto es necesario definir antes otras nociones más elementales. Para todas las definiciones utilizaremos extensivamente la teoría elemental de conjuntos.

2.1 Alfabeto, cadena de caracteres

Un alfabeto es un conjunto de símbolos. Los símbolos pueden ser cualesquiera, como w, 9, #, etc., pero nosotros vamos a utilizar las letras a,b,c, etc. Así, el alfabeto del idioma espanol, $E = \{a, b, c, \ldots, z\}$, es sólo uno de tantos alfabetos posibles. En general utilizaremos el símbolo Σ para denotar un alfabeto.

Con los elementos de un alfabeto es posible formar secuencias o cadenas de caracteres, tales como $mxzxptlk$, $balks$, r, etc. Desde el punto de vista de los conjuntos, una cadena de caracteres puede ser vista como una función de los números naturales al alfabeto:

$$f : N \rightarrow \Sigma$$

Es decir, una cadena es un conjunto de pares ordenados, donde el segundo elemento de cada par es uno de los caracteres de la cadena, y el primer elemento denota la posición de dicho caracter en la cadena; por ejemplo, la cadena "*perro*" se representaría como $\{(1, p), (2, e), (3, r), (4, r), (5, o)\}$. Por simplicidad usaremos la notación habitual para las cadenas de caracteres, pero debe entenderse que ésta es sólo una abreviatura de la notación formal que hemos descrito. Las cadenas de caracteres son llamadas también palabras.

Un caso particular de cadena es la cadena vacía, ε, la cual no tiene ninguna letra.

La longitud de una palabra es la cantidad de letras que contiene, contando las repeticiones; se denota por $|w|$ para una palabra w. Formalmente, $|w|$ es igual a la cardinalidad de w, tomada como conjunto, por la notación arriba descrita.

Una *ocurrencia* de un caracter en una palabra es la posición en que dicho caracter aparece en la palabra. Un mismo caracter puede tener varias ocurrencias en una sóla palabra. Por ejemplo, las ocurrencias de la letra r en la palabra *perro* son $\{3,4\}$. En la representación formal de las palabras, las ocurrencias de un caracter k en una palabra w están en el conjunto $\{n|(n,k) \in w\}$.

Cuando escribimos varias palabras o caracteres uno a continuación de otro, se supone que forman una sóla palabra (se concatenan). Por ejemplo, si $w = abra$ y $v = cada$, entonces $wvbra$ es la palabra $abracadabra$.

Ejemplo.- Definir formalmente la concatenación. *Solución.-* Considérense, para ser específicos, las palabras "saka" y "yama", que concatenadas dan "sakayama"[1]. Representadas como parejas ordenadas, estas palabras son, respectivamente, $\{(1,s),(2,a),(3,c),(4,a)\}$ y $\{(1,y),(2,a),(3,m),(4,a)\}$, mientras que su concatenación es: $\{(1,s),(2,a),(3,c),(4,a),(5,y),(6,a),(7,m),(8,a)\}$. Obsérvese que la segunda palabra aparece "desplazada" 4 caracteres en la concatenación. Esta observación sugiere que la concatenación es la unión de las palabras que se concatenan, pero incrementando el índice[2] tanto como sea el tamaño de la primera palabra. Formalizando, esto da:

$$uv = u \cup \{(i+|u|,j)|(i,j) \in v\}$$

Ejemplo.- Probar que la longitud de la concatenación de dos palabras es la suma de las longitudes de cada una, es decir:

$$|uv| = |u| + |v|$$

Solución.- Aplicando la definición de la concatenación, tenemos:

$$|uv| = |u \cup \{(i+|u|,j)|(i,j) \in v\}|$$

Ahora bien, el tamaño de una unión de conjuntos, si éstos son disjuntos, [3] es igual a la suma de sus tamaños, esto es:

$$|A \cup B| = |A| + |B|$$

si $A \cap B = \emptyset$ Para que este resultado sea aplicable a nuestro problema, tenemos que garantizar que los conjuntos u y $\{(i+|u|,j)|(i,j) \in v\}$ no tienen elementos en común, lo cual dejamos como ejercicio al lector. Entonces tenemos:

$$|u \cup \{(i+|u|,j)|(i,j) \in v\}| = |u| + |\{(i+|u|,j)|(i,j) \in v\}$$

[1] Esto quiere decir "encendedor" en japonés.

[2] Llamamos índice al primer elemento i de los pares (i,σ)

[3] Dos conjuntos son disjuntos si no tienen ningún elemento en común

Finalmente, observamos que el conjunto $\{(i+|u|, j)|(i,j) \in v\}$ tiene el mismo tamaño que v [4], por lo que finalmente tenemos $|uv| = |u| + |v|$ QED.

Una palabra v es *subcadena* de otra w cuando existen cadenas x, y - posiblemente vacías- tales que $xvy = w$. [5]

El conjunto de todas las palabras que se pueden formar con un alfabeto Σ es denotado por Σ^*. Por ejemplo, si $\Sigma = \{a, b\}$, $\Sigma^* = \{a, aa, aaa, aaaa, \ldots, b, bb, bbb, bbbb, \ldots, ab, aba, abb, \ldots\}$. El conjunto Σ^* es infinito, pero enumerable. [6]

2.2 Lenguajes, operaciones con lenguajes

Un *lenguaje* es simplemente un conjunto de palabras. Así, $\{abracadabra\}$ es un lenguaje (de una sóla palabra), $\{ali,\ baba,\ y,\ sus,\ cuarenta,\ ladrones\}$ es otro, Σ^* es otro, etc. [7] Puesto que los lenguajes son conjuntos, podemos efectuar con ellos todas las operaciones de los conjuntos (unión, intersección, diferencia). Definiremos además la operación de concatenación de lenguajes, escrita como $L_1 \bullet L_2$, como una extensión de la concatenación de palabras: $L_1 \bullet L_2 = \{w | w = xy, x \in L_1, y \in L_2\}$ [8]

2.3 Representación finita de lenguajes

Muchos lenguajes son infinitos (por ejemplo Σ^*). Por lo tanto, no podemos representarlos en extensión -esto es, listando cada uno de sus elementos. Es por esto que nos propondremos representar los lenguajes en intención -esto es, dando las características que deben cumplir sus elementos.

Trataremos por lo tanto de representar un lenguaje mediante una expresión finita compuesta de símbolos (variables, paréntesis, etc.) que forman parte de un cierto alfabeto. Por ejemplo, el conjunto de todas las palabras formadas por a's y b's, que es el conjunto infinito $\{\varepsilon, a, b, ab, ba, aaa, aab, \ldots\}$, puede ser representado mediante la cadena de caracteres "$\{a, b\}*$", que es una palabra formada por caracteres del alfabeto $\{$"a","b","$\{$","$\}$","$*$", "," $\}$. Como vemos en este ejemplo, una cadena de caracteres de 6 caracteres puede representar todo un lenguaje infinito.

En la figura 2.1 se ilustra el mapeo que pretendemos entre los lenguajes, que son elementos

[4]Ejercicio: probar este hecho.

[5]Ejercicio: defina la noción de subcadena usando la notación de conjuntos para las palabras.

[6]Ejercicio: probar la enumerabilidad de Σ^*; ayuda: utilizar el orden del diccionario

[7]Ejercicio: explique la diferencia -si la hay- entre un lenguaje vacío y uno que contiene sólo la palabra vacía.

[8]Ejercicio: probar que la concatenación de lenguajes es asociativa pero no conmutativa.

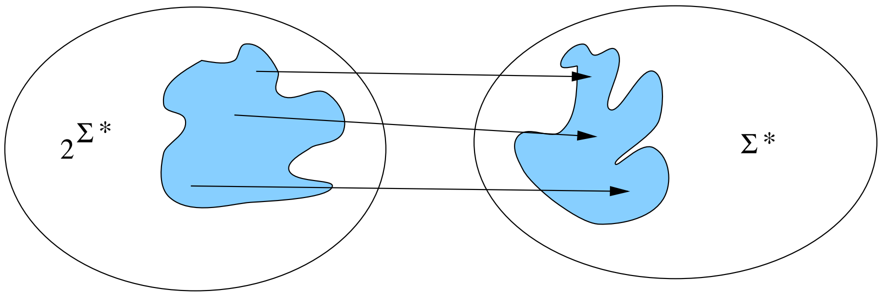


Figura 2.1: Representaciones de lenguajes

del conjunto 2^{Σ^*}, y las cadenas de caracteres que los representan, que son elementos de Σ^*. Desde luego, quisiéramos que una cadena de caracteres no pudiera representar a más de un lenguaje, pues de otro modo no sabríamos a cuál de ellos representa. En cambio, es aceptable que un lenguaje tenga varios representantes.

En vista del éxito obtenido, quisiéramos tener, para cada lenguaje posible, ya sea finito o infinito, un representante que fuera una de estas cadenas finitas de caracteres. Existe sin embargo un problema: para poder hacer lo anterior se necesitaría que hubiera tantos representantes (cadenas de caracteres) como lenguajes representados. Ahora bien, aunque parezca sorprendente, hay mas lenguajes posibles que cadenas de caracteres para representarlos! Esto se debe a que la cantidad de lenguajes posibles es incontable, mientras que las representaciones de dichos lenguajes son contables.

Vamos a probar el siguiente

Teorema.- El conjunto de los lenguajes en un alfabeto Σ finito es incontable.

Nos apoyaremos en el célebre teorema de Kantor, que establece que el conjunto potencia de los números naturales, 2^N, es incontable. En efecto, observamos que el conjunto de todos los lenguajes, que es 2^{Σ^*}, tiene el mismo tamaño que 2^N, pues N y Σ^* son del mismo tamaño, que es lo mismo que decir que Σ^* es contable, lo cual es sencillo de probar [9] Así podemos concluir que, como 2^N es incontable, 2^{Σ^*} también lo es QED.

Se sabe que los conjuntos incontables son propiamente mas "grandes" que los contables, en el sentido de que un conjunto contable no puede ser puesto en correspondencia uno a uno con uno incontable, pero sí con subconjuntos de éste. Así resulta que la cantidad de lenguajes a representar es mayor que la cantidad de cadenas de caracteres que pudieran ser representaciones de aquellos. La conclusión es que no todos los lenguajes pueden ser representados en forma finita.

[9] Para probar que los elementos de Σ^* se pueden poner en una sóla fila, basta con proponer un orden, que puede ser poner primero todas las palabras de longitud 0, luego las de longitud 1, luego las de longitud 2, etc.; para diferenciar el orden de varias palabras de una misma longitud se puede usar el orden del diccionario ("lexicográfico").

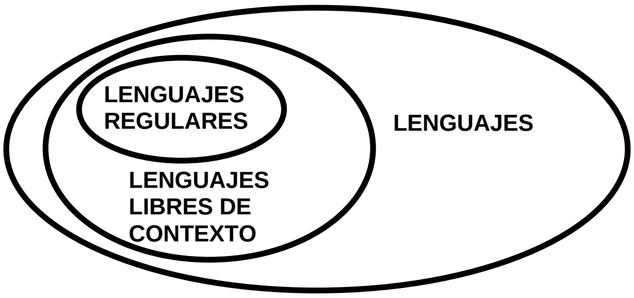


Figura 2.2: Los LR en la jerarquía de Chomsky

En vista de esta desafortunada conclusión, nos tendremos que conformar con representar sólamente algunos lenguajes. Las clases de lenguajes que representaremos son principalmente los lenguajes regulares y los lenguajes libres de contexto, que veremos en las siguientes secciones.

2.4 Lenguajes regulares

En esta sección estudiaremos la más simple de las clases de lenguajes cubiertas en este curso, la de los lenguajes *regulares*, que es al mismo tiempo la de mayor utilidad práctica.

Los Lenguajes Regulares (LR) son incluídos en otras clases más amplias de lenguajes, como los "libres de contexto", los "sensitivos al contexto", los "recursivamente enumerables", etc., que veremos más adelante. Los "libres de contexto" son a su vez incluidos en los "sensitivos al contexto", los cuales son a su vez incluidos en los "recusivamente enumerables", formando así una jerarquía de lenguajes llamada *jerarquía de Chomksy* en honor a N. Chomsky, quien la propuso. Dicha jerarquía se ilustra en la figura 2.2, donde se aprecia que los LR son la más pequeña de las clases de lenguajes; no es de extrañar que es también la más simple.

2.4.1 Conjuntos regulares

Es posible denotar ciertas clases de lenguajes mediante expresiones que siguen la notación de conjuntos. Para definir la noción de conjuntos regulares necesitamos la definición siguiente:

Si L es un lenguaje, L^*, llamado "cerradura de Kleene de L", es el más pequeño conjunto que contiene: [10]

- La palabra vacía, ε

[10] Esta definición es congruente con la notación Σ^* que se utilizó para definir el conjunto de todas las palabras sobre un alfabeto, pues de hecho Σ^* es la cerradura de Kleene del alfabeto.

- El conjunto L
- Todas las palabras formadas por la concatenación de otros miembros de L^*

Por ejemplo, si $L = \{abra, cadabra\}$, $L^* = \{\varepsilon, abra, abraabra, abracadabra, cadabraabra, \ldots\}$ [11]

Definición.- Un lenguaje L es regular si y sólo si se cumple alguna de las condiciones siguientes:

- L es finito;
- L es la unión o concatenación de lenguajes regulares, $L = R_1 \cup R_2$ o $L = R_1 R_2$.
- L es la cerradura de Kleene de algún lenguaje regular, $L = R^*$.

Esta definición nos permite construir expresiones en la notación de conjuntos que representan lenguajes regulares. (Ver ejemplo que sigue)

Ejemplo.- Sea el lenguaje L de palabras formadas por a y b, pero que empiezan con a, como aab, ab, a, $abaa$, etc. Probar que este lenguaje es regular, y dar una expresión de conjuntos que lo represente.

Solución.- El alfabeto es $\Sigma = \{a, b\}$. El lenguaje L puede ser visto como la concatenación de una a con cadenas cualesquiera de a y b; ahora bien, éstas últimas son los elementos de $\{a, b\}^*$, mientras que el lenguaje que sólo contiene la palabra a es $\{a\}$. Ambos lenguajes son regulares. [12] Entonces su concatenación es $\{a\}\{a, b\}^*$, que también es regular.

2.5 Expresiones regulares

La notación de conjuntos nos permite describir los lenguajes regulares, pero nosotros quisiéramos una notación en que las representaciones de los lenguajes fueran simplemente texto (cadenas de caracteres). Así las representaciones de los lenguajes regulares serían simplemente palabras de un lenguaje (el de las representaciones correctamente formadas). Con estas ideas vamos a definir un lenguaje, el de las expresiones regulares, en que cada palabra va a denotar un lenguaje regular.

Definición.- Sea Σ un alfabeto. Las expresiones regulares (ER) sobre Σ son cadenas en el alfabeto $\Sigma \cup$ {"$\wedge$", "+", "$\bullet$", "*", "(", ")", "Φ"} que cumplen con lo siguiente:

[11] Debe quedar claro que la descripción de L^* en este ejemplo no es formal, pues los "..." dejan abierta la puerta a muchas imprecisiones.

[12] En efecto, $\{a\}$ es finito, por lo tanto regular, mientras que $\{a, b\}^*$ es la cerradura de $\{a, b\}$, que es regular por ser finito.

1. "$\wedge$" y "Φ" $\in ER$
2. Si $\sigma \in \Sigma$, entonces $\sigma \in ER$.
3. Si $R_1, R_2 \in ER$, entonces "("R_1"+"R_2")" $\in ER$, "("R_1"$\bullet$"R_2")" $\in ER$, "("R_1")*" $\in ER$.

Las comillas " " enfatizan el hecho de que estamos definiendo cadenas de texto, no expresiones matemáticas [13]

Es la misma diferencia que hay entre el caracter ASCII "0", que se puede teclear en una terminal, y el número 0, que significa que se cuenta un conjunto sin ningún elemento.

Ejemplos.- Son ER en $\{a, b, c\}$ las siguientes: "a", "$((a + b))^*$", "$((a \bullet b) \bullet c)$". No son ER: "ab", "$((a \bullet b(c)^*)$".

2.5.1 Significado de las ER

El significado de una ER es el lenguaje que ella representa. A continuación definiremos la correpondencia entre la representación (una ER) y el lenguaje representado.

Definición.- El significado de una ER es una función $\mathcal{L} : ER \to 2^{\Sigma^*}$ definida de la manera siguiente:

1. $\mathcal{L}$("Φ") $= \emptyset$ (el conjunto vacío)
2. $\mathcal{L}$("$\wedge$") $= \{\varepsilon\}$
3. $\mathcal{L}$("σ") $= \{\sigma\}, \sigma \in \Sigma$.
4. $\mathcal{L}$("("R"$\bullet$"S")") $= \mathcal{L}(R)\mathcal{L}(S), R, S \in ER$
5. $\mathcal{L}$("("R"+"S")") $= \mathcal{L}(R) \cup \mathcal{L}(S), R, S \in ER$
6. $\mathcal{L}$("("R")*") $= \mathcal{L}(R)^*, R \in ER$

Ejemplo.- El significado de la ER "$(((a + b))^* \bullet a)$" se calcula de la manera siguiente:

$\mathcal{L}$("$(((a+b))^* \bullet a)$") $= \mathcal{L}$("$((a+b))^*$")$\mathcal{L}$("a") $= \mathcal{L}$("$(a+b)$")$^*\{a\} = (\mathcal{L}$("a") $\cup \mathcal{L}$("b")$)^*\{a\}$ $= (\{a\} \cup \{b\})^*\{a\} = \{a, b\}^*\{a\}$.

Este es el lenguaje de las palabras sobre $\{a, b\}$ que terminan en a.

[13] Este último es el caso de las expresiones de conjuntos para describir los conjuntos regulares.

Con objeto de hacer la notación menos pesada, vamos a simplificar las ER de la manera siguiente:

- Omitiremos las comillas " ".
- Se eliminan los paréntesis innecesarios. Se supone una precedencia de operadores en el orden siguiente: primero "$*$", luego "$\bullet$" y finalmente "$+$". Además se supone que los operadores "$\bullet$" y "$+$" son asociativos.
- Eventualmente omitiremos el operador "$\bullet$", suponiendo que éste se encuentra implícito entre dos subexpresiones contiguas.

Ejemplos.- a, $(a+b)^*$, abc, ac^* son tomados como "a", "$((a+b))^*$", "$((a \bullet b) \bullet c)$" y "$(a \bullet (c)^*)$", respectivamente.

Ejemplo.- Encontrar una expresión regular para el lenguaje en $\{a,b\}^*$ en el que inmediatamente antes de toda b aparece una a.

Solución.- Una posible ER es $(a+ab)^*$

Una solución aceptable para este tipo de problemas debe cumplir dos características:

1. *Corrección.-* Las palabras que represente la ER propuesta deben satisfacer la descripción del problema (por ejemplo, para el problema del ejemplo, la solución $a^*(a+b)^*$ no es adecuada porque representa algunas palabras, como abb, que no satisfacen la condición de que toda b esté inmediatamente precedida por una a;
2. *Completez.-* La ER propuesta debe representar *todas* las palabras que satisfagan la condición. Así, para el problema del ejemplo, la solución $(ab)^*$ no es adecuada porque hay palabras tales como aab, pertenecientes al lenguaje, que no son representadas por dicha ER.

Al tratar de encontrar un ER para un lenguaje dado, mientras más complejo sea el lenguaje es obvio que resulta mas difícil encontrar por pura intuición dicha ER. En estos casos puede ser conveniente trabajar en forma metódica. Una técnica que funciona en muchos casos consiste en determinar primero la *estructura* de la ER, dejando unos "huecos" pendientes para resolverse luego. Estos huecos, que llamaremos *contextos*, son también lenguajes para los que habrá que encontrar una ER.

Ejemplo.- Obtener una ER para el lenguaje en el alfabeto $\{a,b,c\}$ en que las palabras contienen exactamente una vez dos b contiguas. Por ejemplo, las palabras $aabb$, $babba$,pertenecen al lenguaje, pero no $aaba$, $abbba$ ni $bbabb$.

Para resolver este problema, expresamos primero la estructura de la ER de la manera siguiente:

$$< contexto_1 > \ bb \ < contexto_2 >$$

Podemos ver que en esta expresión aparecen directamente las bb que deben estar en la ER, rodeadas por otras dos ER, que son $< contexto_1 >$ y $< contexto_2$. Ahora el problema es determinar qué ER corresponden a $< contexto_1 >$ y $< contexto_2 >$, lo cual es un *subproblema* del problema original. El lenguaje de $< contexto_1 >$ comprende a las palabras en que toda b está seguida de una a o una c. Esta ER es fácil de obtener directamente, y es $(ba+bc+a+c)^*$. Similarmente se puede obtener la expresión para $< contexto_2 >$, que es $(a+c+ab+cb)^*$, por lo que finalmente la ER del problema es:

$$(ba + bc + a + c)^* bb(a + c + ab + cb)^*$$

2.5.2 Equivalencias de Expresiones Regulares

Las expresiones regulares no representan en forma única a un lenguaje -esto es, la función $L : ER \rightarrow 2^{\Sigma^*}$ descrita arriba no es inyectiva. Esto quiere decir que puede haber varias ER para un mismo lenguaje, lo cual desde luego no es conveniente, pues al ver dos ER distintas no podemos aún estar seguros de que representan dos lenguajes distintos. Por ejemplo, las ER $(a+b)^*$ y $(a^*b^*)^*$ representan el mismo lenguaje.[14].

En esta situación puede ser útil poder verificar que dos ER representan el mismo lenguaje. Esto puede ser hecho por medio de las ecuaciones de equivalencia entre las ER, que son expresiones de la forma $ER_1 = ER_2$ que quieren decir que el lenguaje de ER_1 es el mismo que el de ER_2.

A continuación damos una lista de las principales equivalencias de ER, agrupadas en 9 grupos:

1. $R + S = S + R, \quad (R+S)+T = R+(S+T), \quad R + \Phi = \Phi + R = R, \quad R + R = R$
2. $R \bullet \wedge = \wedge \bullet R = R, \quad R \bullet \Phi = \Phi \bullet R = \Phi, \quad (R \bullet S) \bullet T = R \bullet (S \bullet T)$
3. $R \bullet (S+T) = R \bullet S + R \bullet T, \quad (S+T) \bullet R = S \bullet R + T \bullet R$
4. $R^* = R^* \bullet R^* = (R^*)^* = (\wedge + R)^*, \quad \Phi^* = \wedge^* = \varepsilon$
5. $R^* = \wedge + RR^*$
6. $(R+S)^* = (R^* + S^*)^* = (R^*S^*)^* = (R^*S)^*R^* = R^*(SR^*)^* \neq R^* + S^*$
7. $R^*R = RR^*, \quad R(SR)^* = (RS)^*R$

[14]Este ejemplo es una prueba formal de que las ER no representan en forma única los lenguajes

8. $(R^*S)^* = \wedge + (R+S)^*S, \quad (RS^*)^* = \wedge + R(R+S)^*$
9. $R = SR + T$ ssi $R = S^*T, \quad R = RS + T$ ssi $R = TS^*$

La prueba de varias de estas equivalencias sigue un mismo esquema, que vamos a ejemplificar demostrando $R(SR)^* = (RS)^*R$ (grupo 7). Esta equivalencia se puede probar en dos partes: $R(SR)^* \subseteq (RS)^*R$ y $(RS)^*R \subseteq R(SR)^*$.

1a. parte.- Sea $x \in R(SR)^*$. Entonces x es de la forma $x = r_0 s_1 r_1 s_2 r_2 \dots s_n r_n$. Pero esta misma palabra puede agruparse de otra manera: $x = (r_0 s_1)(r_1 s_2) \dots (r_{n-1} s_n) r_n$. Por lo tanto, $x \in (RS)^*R$.

2a. parte.- Se prueba similarmente. QED.

Las equivalencias de estos 9 grupos pueden usarse para verificar que dos ER denotan el mismo lenguaje, pero sin tener que hacer en ningún momento referencia al lenguaje representado. La técnica a usar para verificar que $P = Q$, donde $P, Q \in ER$, es formar una serie de equivalencias $P = R_1 = R_2 = \dots = R_n = Q$, usando las equivalencias dadas arriba para hacer reemplazamientos.

Ejemplo: Probar $(ab + a)^*a = a(ba + a)^*$ usando equivalencias.

Solución:

$(ab + a)^*a = (a + ab)^*a$ -por (1);

$= (a^*ab)^*a^*a$ -por (6);

$= ([a^*a]b)^*[a^*a]$ -agrupamos términos;

$= [a^*a](b[a^*a])^*$ -por (7);

$= aa^*(baa^*)^*$ - aplicando (7) a los términos entre corchetes;

$= [a][a^*(baa^*)^*]$ -agrupando;

$= a(a + ba)^*$ - por (6);

$= a(ba + a)^*$ - por (1).

2.6 Ejercicios

1. Encontrar Expresiones Regulares que representen los siguientes lenguajes: *Ejercicios.-* Encontrar Expresiones Regulares que representen los siguientes lenguajes:

(a) El conjunto de las palabras en $\{a, b\}$ tales que toda a está precedida por alguna b, como por ejemplo "ε", "b", "bba", "$babaa$", etc. [15]

(b) Conjunto de palabras en $\{0, 1\}$ terminadas en 00.

(c) Conjunto de palabras en $\{0, 1\}$ que contengan tres ceros consecutivos, como "0001000", "000001", etc.

(d) Conjunto de palabras en $\{a, b\}$ que no contienen dos b consecutivas, como "$ababab$", "$aaaa$", etc.

(e) Conjunto de palabras en $\{0, 1\}$ con a lo más un par de ceros consecutivos y a lo más un par de unos consecutivos.

(f) Conjunto de palabras en $\{0, 1\}$ tales que no hay ningún par de ceros consecutivos después [16] de un par de unos consecutivos, como "0000110", "0001000", etc.

(g) El lenguaje $\{101, 1110\}$.

(h) El lenguaje $\{w \in \Sigma^* | w = a^n b a^k, n, k \geq 0\}$

(i) El lenguaje sobre $\{a, b\}$ en que todas las palabras son de longitud impar.

(j) Conjunto de cadenas en $\{a, b\}$ que contienen un número impar de b.

(k) Conjunto de cadenas en $\{a, b\}$ que no contienen ni aa ni bb.

(l) Lenguaje de las palabras en $\{a, b\}$ que no contienen la subcadena "$abaab$".

(m) $\{w \in \{a, b, c\} \mid |w| \neq 3\}$

(n) $\{w \in \{a, b, c\} \mid w \neq \alpha abc \beta\}$

(o) Lenguaje en $\{a, b\}$: $\{w \mid Long(w)$ es par , $|occur(a, w)|$ es par$\}$

(p) El lenguaje sobre $\{a, b\}$ en que las palabras contienen la subcadena "baaab" o bien la subcadena "abbba".

(q) El lenguaje sobre $\{0, 1\}$ en que las palabras no vacías empiezan o terminan en cero.

(r) El lenguaje en $\{0, 1\}$ en que las palabras contienen más de tres ceros.

(s) El lenguaje en $\{0, 1\}$ en que las palabras contienen un número de ceros distinto de 3, por ejemplo "010", "1111", "00100,", etc.

(t) El lenguaje sobre $\{a, b\}$ en que las palabras pueden contener pares de a's consecutivas, pero no grupos de 3 a's consecutivas; por ejemplo, "baabaab", pero no "baaaab".

(u) El lenguaje en $\{0, 1\}$ en que las palabras no contienen un cero exactamente (pero pueden contener dos, tres, etc.), como "1111", "1010", etc.

(v) El lenguaje en $\{a, b\}$ en que toda "b" tiene a su izquierda y a su derecha una "a" (no necesariamente junto), y además el número de "b" es impar.

[15] La b que precede a la a no necesita estar inmediatamente antes.

[16] En cualquier posición a la derecha

2. Demostrar la siguiente equivalencia por identidades de Exresiones Regulares:
   $(ab^*)^*a = a + a(a+b)^*a$

3. Probar las equivalencias:

   (a) $R \bullet \phi = \phi \bullet R = \phi$, para una ER R

   (b) $\phi^* = \wedge$

4. Convertir la ER $a^*ab + b(a + \wedge)$ en notación "fácil" a ER estricta.

5. Pruebar formalmente si los siguientes pares de ER son equivalentes, usando equivalencias o bien encontrando un contraejemplo:

   (a) $a^* + b^*$ y $(a+b)^*$

   (b) a^* y $(aa^*)^*a^*$

6. Demuestrar que $a^* + b^*$ no es equivalente a $(a+b)^*$

7. Suponer que añadimos a las ER un operador, "$-$" que significa que $R_1 - R_2$ representa las palabras representadas por R_1 pero no por R_2. Curiosamente, el operador "$-$" no aumenta el poder de las ER, en el sentido de que para toda ER con operador "$-$" hay una ER equivalente sin "$-$".

   (a) Definir formalmente el significado del operador "$-$", usando el mapeo $\mathcal{L}(ER)$ como se hizo con los demás operadores de las ER.

   (b) Usando este operador, proponer una ER para el lenguaje en $\{a, b\}$ donde las palabras no contienen la subcadena "$abaab$" ni la subcadena "$bbbab$".

8. Suponer que añadimos a las ER un operador de intersección, "&" que significa que $R_1 \& R_2$ representa las palabras representadas simultáneamente por R_1 y por R_2. El operador "&" tampoco aumenta el poder de las ER.

   (a) Define formalmente el significado del operador "&", usando el mapeo $\mathcal{L}(ER)$ como se hizo con los demás operadores de las ER.

   (b) Usando este operador, proponer una ER para el lenguaje en $\{a, b\}$ donde las palabras contienen la subcadena "$abaab$" y un número impar de b's.

Capítulo 3

Autómatas finitos

El término máquina evoca algo hecho en metal, usualmente ruidoso y grasoso, que ejecuta tareas repetitivas que requieren de mucha fuerza / velocidad / precisión. Ejemplos de estas máquinas son las embotelladoras automáticas de refrescos. Su diseño requiere de conocimientos en mecánica, resistencia de materiales, y hasta dinámica de fluídos. Al diseñar tal máquina, el plano en que se le dibuja hace abstracción de algunos detalles presentes en la máquina real, tales como el color con que se pinta, o las imperfecciones en la soldadura.

El plano de diseño mecánico de una máquina es una abstracción de ésta, que es útil para representar su forma física. Sin embargo, hay otro enfoque con que se puede modelizar la máquina embotelladora: cómo funciona, en el sentido de saber qué secuencia de operaciones ejecuta. Así, la parte que introduce el líquido pasa por un ciclo repetitivo en que primero introduce un tubo en la botella, luego descarga el líquido, y finalmente sale el tubo para permitir la colocación de la cápsula (corcholata). El orden en que se efectúa este ciclo es crucial, pues si se descarga el líquido antes de haber introducido el tubo en la botella, el resultado no será satisfactorio.

La modelización de una máquina en tanto que secuencias o ciclos de acciones, se aproxima más al enfoque que adoptaremos en este curso. Las máquinas que estudiaremos son abstracciones matemáticas que capturan sólamente el aspecto referente a las secuencias de eventos que ocurren, sin tomar en cuenta ni la forma de la máquina ni sus dimensiones, ni tampoco si efectúa movimientos rectos o curvos, etc.

En esta parte estudiaremos las máquinas abstractas más simples, los autómatas finitos, las cuales están en relación con los lenguajes regulares, como veremos a continuación.

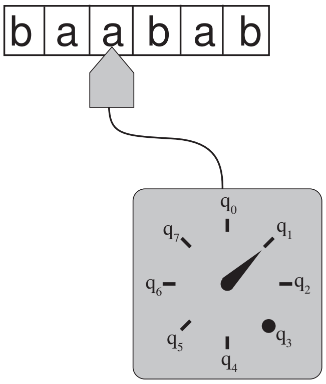


Figura 3.1: Componentes de una máquina abstracta

3.1 Máquinas abstractas

Primeramente estableceremos un marco conceptual que nos permita cernir lo que son las máquinas abstractas para nuestros propósitos. Vamos a suponer que nuestras máquinas pueden ser visualizadas con los siguientes componentes: (ver figura 3.1)

- Una cinta de entrada;
- Una cabeza lectora;
- Un control.

La cabeza lectora se coloca en los segmentos de cinta que contienen los caracteres que componen la palabra de entrada, y al colocarse sobre un caracter lo "lee" y manda esta información al control; tambien puede recorrerse un lugar a la derecha. El control (indicado por una carátula de reloj en la figura) le indica a la cabeza lectora cuándo debe recorrerse a la derecha. Se supone que hay manera de saber cuando se acaba la entrada (por ejemplo, al llegar al blanco). La "aguja" del control puede estar cambiando de posición, y hay algunas posiciones llamadas *finales* (como la indicada por un punto, q_3) que son consideradas especiales, por que permiten determinar si una palabra es aceptada o rechazada, como veremos más adelante.

3.2 Máquinas de estados finitos

Las llamadas *máquinas de estados finitos* se caracterizan por que la cantidad de "posiciones" que puede tomar la "aguja" del control es fija –y desde luego finita. Las "posiciones" del control son llamadas *estados*; de ahí el nombre de estas máquinas. En una máquina de estados finitos la cabeza lectora siempre se desplaza a la derecha después de leer un caracter (no puede ni quedarse quieta ni moverse a la izquierda). Además, con cada caracter leído, el control cambia de estado; decimos que efectúa una *transición*.

Cuando el autómata finito acaba de leer la cinta de entrada, pueden ocurrir dos cosas: ya sea que se encuentre en un estado final, en cuyo caso la palabra es *aceptada*, o bien que se encuentre en un estado cualquiera que no sea final, caso en el que la entrada no es aceptada (es *rechazada*).

Los estados son el único medio de que disponen estas máquinas para recordar los eventos que ocurren (por ejemplo, qué caracteres se han leído hasta el momento); esto quiere decir que son máquinas de memoria limitada. En última instancia, las computadoras digitales son máquinas de memoria limitada, aunque la cantidad de estados posibles que puede tener su memoria puede ser enorme.

3.2.1 Definición formal de autómatas finitos determinísticos

Una máquina de estados finitos M es un quíntuplo $(K, \Sigma, \delta, s, F)$, donde:

- K es un conjunto de identificadores (símbolos) de estados;
- Σ es el alfabeto de entrada;
- $s \in K$ es el estado inicial;
- $F \subseteq K$ es un conjunto de estados finales;
- $\delta : K \times \Sigma \to K$ es la función de transición, que a partir de un estado y un símbolo del alfabeto obtiene un nuevo estado.[1]

La función de transición indica a qué estado se va a pasar sabiendo cuál es el estado actual y el caracter que se está leyendo. Es importante notar que δ es una función y no simplemente una relación; esto implica que para un estado y un símbolo del alfabeto dados, habrá un y sólo un estado siguiente. Esta característica, que permite saber siempre cuál será el siguiente estado, se llama *determinismo*. La definición dada arriba corresponde a los *autómatas finitos determinísticos* [2]

Una vez que hemos formalizado los elementos básicos de un autómata, falta describir su funcionamiento desde el punto de vista de los cálculos que se pueden efectuar con ellos, en particular las palabras que aceptan o rechazan.

[1] que puede ser el mismo en el que se encontraba.
[2] después veremos otros autómatas finitos no determinísticos.

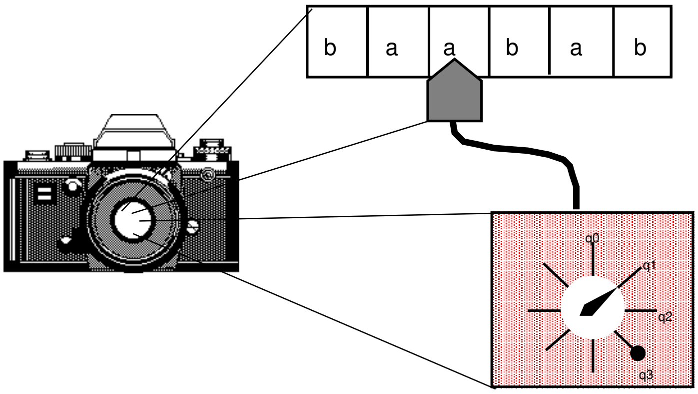


Figura 3.2: La configuración es como una fotografía de la situación de un autómata en medio de un cálculo

Configuraciones $(*)$

El *funcionamiento* dinámico de los AF lo vamos a definir de manera análoga a como se simula el movimiento en el cine, es decir, mediante una sucesión de fotografías. Así, la operación de un AF se describirá en términos de la sucesión de situaciones por las que pasa mientras analiza una palabra de entrada.

El equivalente en los AF de lo que es una fotografía en el cine es la noción de *configuración*, como se ilustra en la figura 3.2. La idea básica es la de describir completamente la situación en que se encuentra la máquina en un momento dado, incluyendo el contenido de la cinta, la cabeza lectora y el control.

Las informaciones relevantes para resumir la situación de la máquina en un instante son:

1. El contenido de la cinta,
2. la posición de la cabeza lectora,
3. el estado en que se encuentra el control.

Una configuración sería entonces un elemento de $\Sigma^* \times N \times K$, donde el primer elemento es el contenido de la cinta, el segundo describe la posición de la cabeza, y el tercero es el estado.

Sólo nos interesará incluir en las configuraciones aquellas informaciones que tengan relevancia en cuanto a la aceptación de la palabra al final de su análisis. Así, por ejemplo, es evidente que, como la cabeza lectora no puede echar marcha atrás, los caracteres por los que ya pasó no afectarán más el funcionamiento de la máquina. Por lo tanto, es suficiente con considerar lo que falta por leer de la palabra de entrada, en vez de la palabra completa.

Esta solución tiene la ventaja de que entonces no es necesario representar la posición de la cabeza, pues ésta se encuentra siempre al inicio de lo que falta por leer.

Entonces una configuración será un elemento de $K \times \Sigma^*$. Por ejemplo, la configuración correspondiente a la figura 3.1 sería: $(q_1, abab)$.

Para hacer las configuraciones más legibles, vamos a utilizar dobles corchetes en vez de paréntesis, como en $[[q_1, abab]]$.

Cálculos en máquinas finitas $(*)$

Vamos a definir una relación entre configuraciones, $C_1 \vdash_M C_2$, que significa que de la configuración C_1 la máquina M puede pasar en un paso a la configuración C_2. Definimos formalmente esta noción:

Definición.- $[[q_1, w_1]] \vdash_M [[q_2, w_2]]$ si y sólo si $w_1 = \sigma w_2$ para un $\sigma \in \Sigma$, y existe una transición en M tal que $\delta(q_1, \sigma) = q_2$. ($\sigma$ es el caracter que se leyó).

La cerradura *reflexiva* y *transitiva* de la relación $\vdash_M$ es denotada por $\vdash_M^*$. [3] Así, la expresión $C_1 \vdash_M^* C_2$ indica que de la configuración C_1 se puede pasar a C_2 en algún número de pasos (que puede ser cero, si $C_1 = C_2$). Ahora ya tenemos los conceptos necesarios para definir cuando una palabra es aceptada.

Definición.- Una palabra $w \in \Sigma^*$ es *aceptada* por una máquina $M = (K, \Sigma, \delta, s, F)$ ssi existe un estado $q \in F$ tal que $[[s, w]] \vdash_M^* [[q, \varepsilon]]$. Nótese que no basta con que se llegue a un estado final q, sino que además ya no deben quedar caracteres por leer (lo que falta por leer es la palabra vacía).

El concepto de *lenguaje aceptado* es una simple extensión de aquél de palabra aceptada:

Definición.- El lenguaje aceptado por una máquina M es el conjunto de palabras aceptadas por dicha máquina.

Ejemplo.- Sea el autómata finito $M = (K, \Sigma, \delta, q_0, F)$, donde:

$K = \{q_0, q_1, q_2\}$

$\Sigma = \{a, b\}$

$F = \{q_0, q_2\}$

y la función δ aparece tabulada como sigue:

[3] La cerradura reflexiva y transitiva de una relación R, denotada R^*, es la menor extensión de R, es decir, $R \cup \Delta$, tal que $R \cup \Delta$ es reflexiva y transitiva aunque inicialmente R no lo haya sido. En otras palabras, a R se le agregan todos los pares ordenados que sean necesarios hasta que se vuelva transitiva y reflexiva.

$\mathbf{q}$	σ	$\delta(\mathbf{q},\sigma)$
q_0	a	q_1
q_0	b	q_2
q_1	a	q_1
q_1	b	q_1
q_2	a	q_0
q_2	b	q_2

Probar que esta máquina acepta la palabra *babba*.

Solución.- Hay que encontrar una serie de configuraciones tales que se pueda pasar de una a otra por medio de la relación $\vdash_M$. Una posibilidad es la siguiente:

$$[[q_0, babba]] \vdash_M [[q_2, abba]] \vdash_M [[q_0, bba]]$$
$$\vdash_M [[q_2, ba]] \vdash_M [[q_2, a]] \vdash_M [[q_0, \varepsilon]].$$

Como $q_0 \in F$, la palabra es aceptada.

Definición.- Un *cálculo* en una máquina M es una secuencia de configuraciones C_1,C_2,...,C_n, tales que $C_i \vdash C_{i+1}$. Generalmente escribimos los cálculos como $C_1 \vdash_M C_2 \vdash_M \ldots \vdash_M C_n$.

Teorema.- Dados una palabra $w \in \Sigma^*$ y una máquina $M = (K, \Sigma, \delta, s, F)$, sólo hay un cálculo $[[s, w]] \vdash_M \ldots \vdash_M [[q, \varepsilon]]$.

Prueba.- (por contradicción): Sean dos cálculos distintos:

$$[[s, w]] \vdash_M \ldots \vdash_M [[p, \sigma w']] \vdash_M [[r, w']] \vdash_M \ldots [[q_r, \varepsilon]]$$

$$[[s, w]] \vdash_M \ldots \vdash_M [[p, \sigma w']] \vdash_M [[s, w']] \vdash_M \ldots [[q_s, \varepsilon]]$$

y sean $[[r, w']]$ y $[[s, w']]$ las primeras configuraciones distintas en los dos cálculos. [4] Esto implica que $\delta(p, \sigma) = r$ y también $\delta(p, \sigma) = s$, y como δ es función, se sigue que $r = s$, lo que contradice la hipótesis. QED.

3.2.2 Notación gráfica

La representación tabular de la función δ no es muy cómoda para su comprensión intuitiva. Una alternativa conveniente es la de representar las transiciones por medio de grafos dirigidos, llamados *diagramas de estados*, donde las flechas que representan una transición van de un estado a otro, y tienen como etiqueta el símbolo que ocasiona la transición; un ejemplo de AF se muestra en la figura 3.3.

[4] Es decir, los cálculos son iguales hasta cierto punto, que en el peor caso es la configuración inicial $[[s, w]]$.

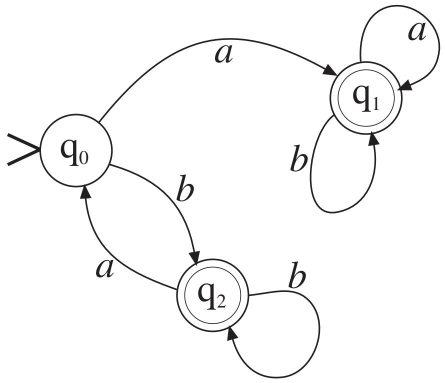


Figura 3.3: Notación gráfica

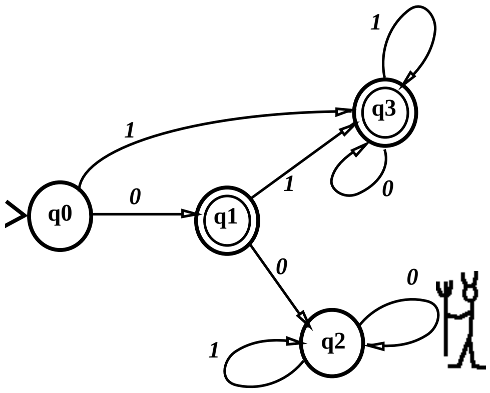


Figura 3.4: AF para palabras que no empiezan en "00"

El estado inicial del autómata se indica con una marca ">", y los estados finales aparecen con doble círculo.

Puesto que δ es una función, es evidente que de cada nodo de cualquier diagrama de estados deben partir exactamente $|\Sigma|$ flechas con etiquetas distintas (tantas como elementos haya en el alfabeto); de otra manera no se están respetando las normas de un autómata finito determinístico.

Sobre la figura del diagrama de estados de un autómata es muy simple ver si acepta o no una palabra dada; basta con ir siguiendo las flechas según el símbolo de entrada, y verificar si al final se llegó o no a un estado final. En otras palabras, una palabra es aceptada ssi existe una *trayectoria* en el diagrama de estados, que une el estado inicial con un estado final, y tal que la concatenación de las etiquetas de las flechas es igual a la palabra de entrada.

Ejemplo.- Para el AF de la figura 3.3 la trayectoria seguida para la palabra ab consiste en la secuencia de estados: q_0, q_1, q_1

Ejemplo.- Diseñar un AF que acepte exactamente el lenguaje en el alfabeto $\{0, 1\}$ en que las palabras no comienzan con 00. *Solución.*- Para emprender el diseño en forma metódica,

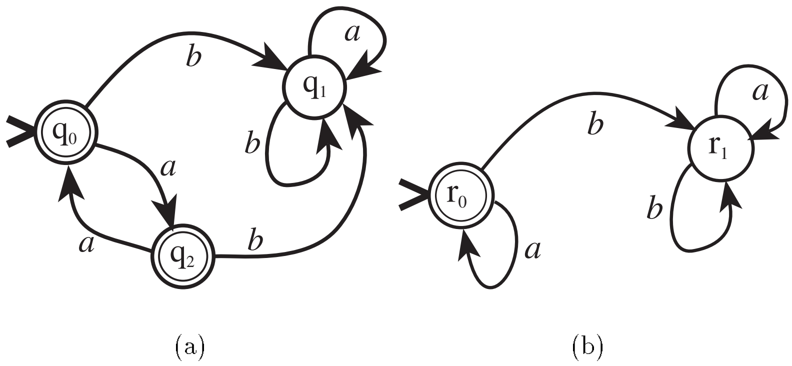


Figura 3.5: Autómatas equivalentes

comenzamos por formar un "esqueleto" en que, a partir de un estado inicial q_0 pasamos con un 0 a q_1 y de ahí a q_2, que debe ser un estado no final en el que caen todas las palabras que empiecen con 00 (ver figura 3.4). Claramente tanto q_0 como q_1 deben ser estados finales. Ahora hay que completar el AF, agregando las transiciones que falten. A partir de q_0, si llega un 1 habrá que ir a un estado final en el que se permanezca en adelante; agregamos al AF un estado final q_3 y la transición de q_0 a q_3 con 1. El estado q_3 tiene transiciones hacia sí mismo con 0 y con 1. Finalmente, al estado q_1 le falta su transición con 1, que obviamente dirigimos hacia q_3, con lo que el AF queda terminado.

Ejercicios.- Diseñar máquinas de estados finitos para los ejercicios de la página 21. Para cada inciso dibuje un grafo para la función de transición, describa formalmente el autómata y desarrolle el cálculo para una palabra aceptada y para otra rechazada.

3.3 Equivalencia de autómatas finitos.

Decimos que los distintos autómatas que aceptan ese mismo lenguaje son equivalentes:

Definición.- Dos autómatas M_1 y M_2 son *equivalentes*, $M_1 \approx M_2$, cuando aceptan exactamente el mismo lenguaje.

Pero, ¿puede haber de hecho varios AF distintos[5] que acepten un mismo lenguaje? La respuesta es afirmativa, y una prueba consiste en exhibir un ejemplo.

Por ejemplo, los autómatas (a) y (b) de la figura 3.5 aceptan ambos el lenguaje a^*.

En vista de esta situación, dados dos AF distintos existe la posibilidad de que sean equivalentes. Pero ¿cómo saberlo?

[5]¿Qué se quiere decir por "distintos"? ¿Si dos AF sólo difieren en los nombres de los estados se considerarán distintos?

De acuerdo con ls definición anterior, la demostración de equivalencia de dos autómatas se convierte en la demostración de igualdad de los lenguajes que aceptan. Sin embargo, demostrar la igualdad de dos lenguajes puede complicarse si se trata de lenguajes infinitos. Es por esto que se prefieren otros métodos para probar la equivalencia de autómatas.

Recuérdese que en el caso de las ER se hizo uso de un grupo de *equivalencias*, para probar, mediante remplazamientos de iguales por iguales, que dos ER son equivalentes. Sin embargo, dicho método es en la práctica muy difícil de usar, pues en general no se sabe qué equivalencia puede ser útil aplicar.

El método que aquí propondremos para los AF se basa en el siguiente teorema:

Teorema de Moore.- Existe un algoritmo para decidir si dos autómatas finitos son equivalentes o no lo son.

El algoritmo mencionado en el teorema de Moore consiste en la construcción de una tabla de comparación de autómatas. Esta tabla permite convertir el problema de la comparación de los lenguajes aceptados en un problema de comparación de estados de los autómatas.

Definición.- Decimos que dos estados q y q' son *compatibles* si ambos son finales o ninguno de los dos es final. En caso contrario, son estados *incompatibles*.

La idea del algoritmo de comparación de AFD_1 y AFD_2 consiste en averiguar si existe alguna secuencia de caracteres w tal que siguiéndola simultáneamente en AFD_1 y AFD_2 se llega a estados incompatibles. Si dicha secuencia no existe, entonces los autómatas son equivalentes.

El único problema con esta idea estriba en que hay que garantizar que sean cubiertas todas las posibles cadenas de caracteres w, las cuales son infinitas en general. Por ello se ideó una estructura, la *tabla de comparación*, que permite explorar exhaustivamente todas las posibilidades de incompatibilidad de estados, en un tiempo finito. La tabla de comparación de autómatas consta de $|\Sigma|+1$ columnas, y se construye de la manera siguiente, para dos autómatas $M=(K,\Sigma,\delta,s,F)$ y $M'=(K',\Sigma',\delta',s',F')$:

1. Inicialmente se anota en la columna 0 un par ordenado (s,s') que contiene los estados iniciales de M y M' respectivamente;

2. Si en la columna 0 hay un par (r,r'), en las columnas 1 a $|\Sigma|$ se inscriben los pares (r_σ,r'_σ), con $r_\sigma=\delta(r,\sigma)$, $r'_\sigma=\delta(r',\sigma)$.

3. Cada par (r_σ,r'_σ) generado en el punto 2 que no aparezca en la columna 0, se anota ahí.

4. Si aparece en la columna 0 un par (r,r') de estados incompatibles, se interrumpe la construcción de la tabla, concluyendo que los dos autómatas no son equivalentes. En caso contrario se continúa a partir del paso 2.

5. Si no aparecen nuevos pares (r_σ, r'_σ) que no estén ya en la columna 0, se termina el proceso, concluyendo que los dos autómatas son equivalentes.

Ejemplo.- Sean los autómatas M y M' los de la figuras 3.5(a) y (b) respectivamente. La tabla de comparación es como sigue:

(q, q')	a	b
(q_0, r_0)	(q_2, r_0)	(q_1, r_1)
(q_2, r_0)	(q_0, r_0)	(q_1, r_1)
(q_1, r_1)	(q_1, r_1)	(q_1, r_1)

Se concluye que M y M' son equivalentes.

Para probar que este método constituye un algoritmo de decisión para verificar la equivalencia de dos autómatas, hay que mostrar los puntos siguientes:

1. La construcción de la tabla termina siempre (no se “cicla”)
2. Si en la tabla aparecen pares de estados incompatibles (uno final y el otro no final), entonces los lenguajes aceptados por los autómatas son efectivamente distintos.
3. Si se comparan dos autómatas que no son equivalentes, entonces en la tabla aparecerán estados incompatibles.

El punto 1 se prueba fácilmente porque, la columna 1 siendo un subconjunto de $K \times K'$, que es finito, dicha columna no puede extenderse indefinidamente.

Para probar el punto 2 basta con trazar los pasos que llevaron a generar un par de estados incompatibles, (r, r'), $r \in F$, $r' \notin F'$. Si concatenamos los caracteres de entrada s del paso 2 del algoritmo para cada uno de esos pasos, obtendremos una palabra w. Si aplicamos w como entrada al autómata M llegaremos al estado r, es decir, w será aceptada. En cambio, si aplicamos la misma w a M', llegaremos al estado r', que no es final, por lo que w no será aceptada. Esto muestra que los lenguajes aceptados por M y por M' difieren en al menos una palabra, w.

En cuanto al punto 3, si los lenguajes $\mathcal{L}(M)$ y $\mathcal{L}(M')$ son diferentes, entonces existe al menos una palabra, sea w, tal que es aceptada por uno y rechazada por el otro. En consecuencia, siguiendo la palabra w sobre la tabla, caracter por caracter, debemos llegar a un par incompatible. [6]

[6] Ejercicio: ¿Cómo se está seguro de que es posible seguir w sobre la tabla, caracter por caracter? ¿No podría “atorarse” el proceso?.

Por otra parte, el punto 3 implica que si no hay pares incompatibles en la tabla, entonces los lenguajes son idénticos. En efecto, por propiedades de la lógica elemental, al negar la conclusión de 3 se obtiene la negación de su premisa. QED. [7]

3.4 Simplificación de Autómatas finitos

Es posible aplicar la comparación de autómatas para simplificar los autómatas finitos. Recuérdese que en el caso de las Expresiones Regulares la simplificación podía hacerse aplicando equivalencias, pero este método resulta bastante difícil e ineficaz, pues no es evidente qué equivalencia hay que aplicar, y después de aplicarla todavía no está uno seguro si se avanzó hacia el resultado deseado.

En el caso de los AFD, vamos a entender por simplificación la reducción en el número de estados, pero aceptando el mismo lenguaje que antes de la simplificación. Más aún, llamaremos *minimización* a la obtención de un autómata con el menor número posible de estados.

Como un primer ejemplo, considérense los AFD de las figuras 3.5 (a) y (b). En el AFD de (a), los estados q_0 y q_2 son en cierto modo redundantes, porque mientras se estén recibiendo as, el AFD continúa en q_0 o en q_2, y cuando se recibe una b se pasa a q_1. Se puede pensar entonces en eliminar uno de ellos, por ejemplo q_2, y obtener el autómata de la figura 3.5(b), que tiene un estado menos.

Esta idea de "estados redundantes" se formaliza en lo que sigue:

Definición (∗).- Dos estados son *equivalentes*, $q_1 \approx q_2$, ssi intercambiar uno por otro en cualquier configuración no altera la aceptación o rechazo de toda palabra.

Formalmente escribimos: Dos estados p y q son equivalentes si cuando $[[s, uv]] \vdash_M^* [[q, v]] \vdash_M^* [[r, \varepsilon]]$ y $[[p, v]] \vdash_M^* [[t, \varepsilon]]$ entonces r y t son estados compatibles.

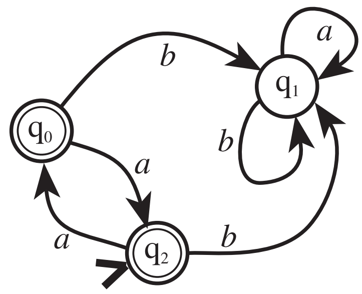
Esta definición quiere decir que, si $p \approx q$, al cambiar q por p en la configuración, la palabra va a ser aceptada (se acaba en el estado final t) si y sólo si de todos modos iba a ser aceptada sin cambiar p por q (se acaba en el estado final r).

El único problema con esta definición es que, para verificar si dos estados dados p y q son equivalentes, habría que examinar, para cada palabra posible de entrada, si intercambiarlos en las configuraciones altera o no la aceptación de esa palabra. Esto es evidentemente imposible para un lenguaje infinito. La definición nos dice qué son los estados equivalentes, pero no cómo saber si dos estados son equivalentes. Este aspecto es resuelto por el siguiente lema:

[7] *Ejercicio.-* Desarrollar un programa en el lenguaje *Scheme* para efectuar la comparación de dos autómatas mediante la construcción de la tabla de comparación.

Lema: Dado un AFD $M = (K, \Sigma, \delta, q, F)$ y dos estados $q_1, q_2 \in K$, $q_1 \approx q_2$ ssi $(K, \Sigma, \delta, q_1, F) \approx (K, \Sigma, \delta, q_2, F)$. La prueba es un ejercicio (difícil) para el lector.

Es decir, para saber si dos estados q_1 y q_2 son equivalentes, se les pone a ambos como estado inicial de sendos autómatas M_1 y M_2, y se procede a comparar dichos autómatas. Si éstos últimos son equivalentes, quiere decir que los estados q_1 y q_2 son equivalentes. Por ejemplo, para el autómata de la figura 3.5(a), para verificar si $q_0 \approx q_2$, habría que comparar dicho AFD con el de la figura siguiente:



Una vez que se sabe que dos estados son equivalentes, se puede pensar en eliminar uno de ellos, para evitar redundancias y hacer más eficiente al AFD. Sin embargo, la eliminación de un estado en el AFD plantea el problema de qué hacer con las flechas que conectan al estado eliminado con el resto del autómata. Esta cuestión se resuelve con los siguientes criterios:

1. Las flechas que salen del estado eliminado son eliminadas;
2. Las flechas que llegan al estado eliminado son redirigidas hacia su estado equivalente.

Por ejemplo, en el autómata de la figura 3.5(a), si verificamos que q_0 y q_2 son equivalentes, y pensamos eliminar q_2, hay que redirigir la flecha que va de q_0 a q_2 para que vaya al mismo q_0 (se vuelve un ciclo). Así se llega al autómata de la figura 3.5(b).

La eliminación de estados redundantes de un AFD es suficiente para simplificarlo al mínimo. Este resultado –que de ninguna manera es evidente por sí mismo– nos permite definir un algoritmo para minimizar los AFD.

Vamos a llamar $\mu(M)$ el AFD que resulta de eliminar los estados redundantes (que son equivalentes a otros).

Teorema.- Al eliminar los estados redundantes de un AFD M, se llega a un AFD mínimo $\mu(M)$, que no contiene estados equivalentes, y que es único para el lenguaje $\mathcal{L}(M) = \mathcal{L}(\mu(M))$.

El algoritmo que permite obtener el AFD mínimo es como sigue:

Repetir:

Para cada par (q_1, q_2) de estados del autómata:

1. Correr el algoritmo de equivalencia con $(K, \Sigma, \delta, q_1, F)$ y $(K, \Sigma, \delta, q_2, F)$
2. Si $q_1 \approx q_2$, eliminar q_2 y volver a empezar;

hasta que no haya estados que eliminar.

Este algoritmo conlleva un alto costo computacional, puesto que hay que confrontar todos los estados contra todos para saber si son equivalentes, y esto implica correr el algoritmo de equivalencia de autómatas para cada par de estados. Varias optimizaciones son posibles, por ejemplo, no comparar mas que los estados compatibles (si dos estados no son compatibles, menos van a ser equivalentes). Otros algoritmos de minimización de autómatas pueden encontrarse en la literatura.

Con este algoritmo de minimización se puede también comparar dos AFD M_1 y M_2: $\mathcal{L}(M_1) = \mathcal{L}(M_2)$ ssi $\mu(M_1) = \mu(M_2)$; dos AFD son equivalentes ssi sus AFD mínimos son iguales. Aquí la igualdad $\mu(M_1) = \mu(M_2)$ se entiende en cuanto a la estructura de los AFD, pero los nombres de los estados pueden ser diferentes. Esta idea puede ser precisada con la idea de *isomorfismo* que se encuentra definida en los libros de álgebra elemental, y que en el caso de los AFD se puede expresar como sigue:

Definición.- Dos AFD $(K, \Sigma, \delta, q_1, F)$ y $(K', \Sigma, \delta', q_1', F')$ son *iguales* (isomorfos) ssi existe un mapeo uno a uno $h : K \to K'$ tal que:

- $\delta(r, \sigma) = s$, $r, s \in K$, ssi $\delta'(h(r), \sigma) = h(s)$
- $h(q_1) = q_1'$
- $q \in F$ ssi $h(q) \in F'$

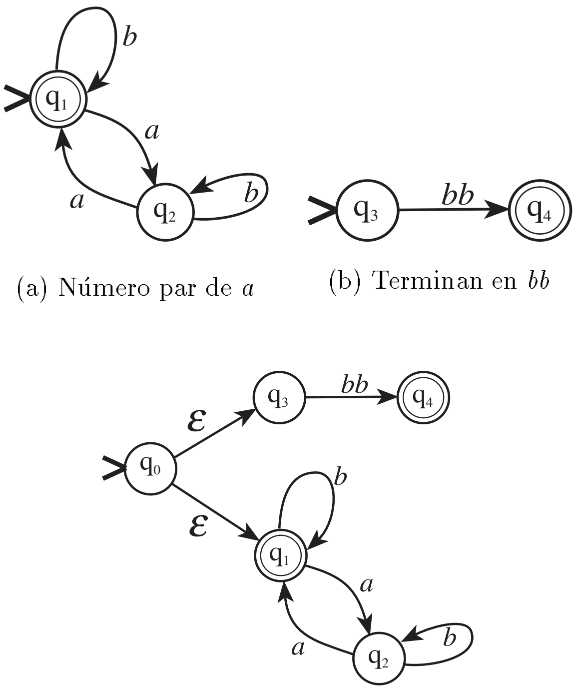
Desde luego, minimizar AFD para compararlos no es buena idea desde el punto de vista de la eficiencia, puesto que para minimizar cada AFD hay que efectuar muchas comparaciones de autómatas, al menos con el algoritmo que hemos visto.

El hecho de que haya un autómata finito único para cada lenguaje aceptable por algún AFD, es de gran trascendencia, puesto que dicho AFD mínimo es de alguna manera "el representante" del lenguaje que acepta; este hecho contrasta con la situación que encontrábamos en las expresiones regulares, en las que no hay una ER que sea el "mejor representante" de un lenguaje regular.

3.5 Autómatas finitos no deterministas

Una extensión a los autómatas finitos deterministas es la de permitir que de cada nodo del diagrama de estados salga un número de flechas mayor o menor que $|\Sigma|$. Así, se puede permitir que falte la flecha correspondiente a alguno de los símbolos del alfabeto, o bien que haya varias flechas que salgan de un sólo nodo con la misma etiqueta. A estos autómatas finitos se les llama *no determinísticos* o *no deterministas*, por razones que luego veremos.

Al retirar algunas de las restricciones que tienen los autómatas finitos determinísticos, su diseño para un lenguaje dado puede volverse más simple. Por ejemplo, para diseñar un autómata que acepte las palabras sobre $\{a, b\}$ que tengan un número par de a (figura 3.6-a) o que terminen en bb (figura 3.6-b), simplemente "juntamos" los autómatas correspondientes a cada uno de estos lenguajes (figura 3.6-c).



(a) Número par de a (b) Terminan en bb

(c) Combinación de los dos

Figura 3.6: Combinación de AFNs

Resolver este mismo problema con autómatas finitos determinísticos es bastante menos simple.

Hasta aquí sólo vemos ventajas de los AFN sobre los AFD. Sin embargo, en los autómatas no determinísticos se presenta el problema de saber qué hacer cuando, estando en un nodo n, y habiendo un símbolo de entrada a, no existe ninguna flecha que salga de n con etiqueta

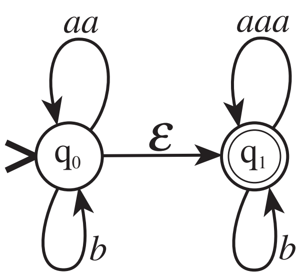


Figura 3.7: AFN que acepta $(aa+b)^*(aaa+b)^*$

a. También hay que saber qué camino tomar cuando hay varias flechas que parten de n con etiqueta a.

Estas diferencias con los AFD se deben reflejar en la definición formal de los AFN, como se hace en seguida.

Definición.- Un autómata finito no determinista es un quíntuplo $(K, \Sigma, \Delta, s, F)$ donde K, Σ, s y F tienen el mismo significado que para el caso de los autómatas determinísticos, y Δ, llamado la *relación de transición*, es un subconjunto finito de $K \times \Sigma^* \times K$.

El punto esencial es que Δ es una relación, no una función. Obsérvese también que el segundo elemento de la relación de transición es una palabra, no un caracter del alfabeto. Esto significa que cada tripleta $(q_1, w, q_2) \in \Delta$, llamada *transición*, y representada como una flecha de etiqueta w en el diagrama de estados, permite pasar de q_1 a q_2 "gastando" en la entrada una subcadena w. [8]

Vamos a definir la noción de palabra aceptada en términos de la representación gráfica de los autómatas no determinísticos.

Definición.- Una palabra w es aceptada por un autómata no determinístico ssi existe una trayectoria en su diagrama de estados, que parte del estado inicial y llega a un estado final, tal que la concatenación de las etiquetas de las flechas es igual a w. [9]

Ejemplo.- Considérese el lenguaje denotado por la expresión regular $(aa+b)^*(aaa+b)^*$. Obtener un autómata no determinístico que acepte dicho lenguaje, y probar que acepta la cadena $aabaaaab$, mientras que no acepta la cadena $aaabaaaab$.

Solución.- Se propone el autómata de la figura 3.7. Este autómata contiene dos estados, q_0 y q_1, que corresponden respectivamente a las subexpresiones $(aa+b)^*$ y $(aaa+b)^*$. Se pasa de q_0 a q_1 mediante una transición con la palabra vacía, o sea que se puede cambiar de estado sin consumir ningún caracter de la entrada. Una vez que se pasó de q_0 a q_1 ya no hay manera de regresar a q_0. En esto consiste el no determinismo: estando en q_0 uno puede

[8] Nótese que w puede ser la palabra vacía.

[9] *Ejercicio.*- Expresar la definición de palabra aceptada en términos de la noción de configuración.

decidir quedarse en q_0 o bien cambiarse a q_1. [10]

La cadena $aabaaaaab$ se puede dividir como $(aa \bullet b \bullet aa) \bullet (aaa \bullet b)$, y en esta expresión, el grupo de caracteres en los paréntesis a la izquierda $(aabaa)$ puede ser generada estando en el estado q_0 del autómata, mientras que el grupo a la derecha $(aaab)$ puede ser generado en el estado q_1. Así probamos que la cadena $aabaaaaab$ sí es aceptada por el autómata. Probar que una cadena no es aceptada por un autómata no determinístico es más difícil, pues hay que mostrar que no existe ninguna trayectoria que satisfaga los requisitos; la cantidad de trayectorias posibles puede ser muy grande como para examinar una por una. En este ejemplo en particular es posible ver que la cadena $aaabaaaaab$ no es aceptada por el autómata, pues el primer grupo de aaa obliga al autómata a estar en q_1, y a partir de q_1 ya sólo se pueden admitir cadenas que se puedan dividir en grupos de aaa, lo cual no es el caso de $aaaaa$. [11]

3.5.1 El método de los conjuntos de estados

Teorema.- Es posible determinar si un AFN dado acepta una palabra w o no la acepta.

Una prueba de este teorema podría consistir en proponer un método específico para decidir la aceptación de una palabra por un AFN.

En el caso de los AFD, es trivial probar si aceptan o no una palabra dada w, pues es suficiente con verificar si con w se llega a un estado final al seguir la (única) trayectoria en el grafo del AFD. En cambio en los AFN, al poder existir varios caminos posibles, hay la posibilidad de elegir uno erróneo; más aún, si encontramos para una palabra w una trayectoria que llegue a un estado no final, esto no basta para probar que la palabra no es aceptada, pues siempre queda la duda si simplemente la trayectoria fué mal elegida.

Una idea para solucionar este problema sería la de explorar *simultáneamente todas las posibles trayectorias* para la palabra dada, y si en ninguna de ellas se llega a un estado final, entonces la palabra no es aceptada; en caso contrario (en al menos una de las trayectorias se llega a un estado final) la palabra es aceptada.

Una transformación inofensiva

Sin embargo, una dificutad para explorar simultáneamente varias trayectorias es que algunas flechas del autómata tienen como etiquetas palabras de varias letras, y así una de las trayectorias puede "consumir" en una transición más letras que otra. Este problema se aprecia en el AFN de la figura 3.7, al hacer una transición a partir del estado inicial con la entrada "b".

[10] *Ejercicio.-* Para fines de comparación, trate de hacer directamente un autómata finito determinístico para el lenguaje $(aa + b)^*(aaa + b)^*$.

[11] *Ejercicio.-* Proponer una definición formal de palabra aceptada por un autómata no determinístico utilizando las nociones de configuración y de paso de una configuración a otra (cálculo).

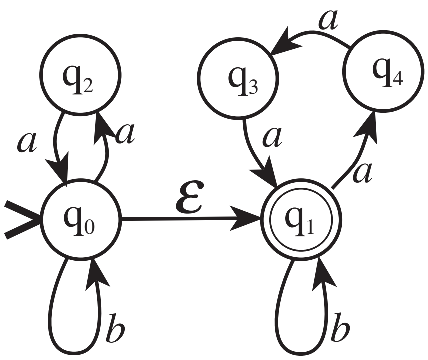


Figura 3.8: AFN transformado

Una solución a esta dificultad es normalizar a 1 la longitud de las palabras que aparecen en las flechas. Esto puede hacerse intercalando $|w|-1$ estados intermedios en cada flecha con etiqueta w. Así, por ejemplo, de la transición (q_1, aaa, q_1) de la figura 3.7, se generan las transiciones siguientes: (q_1, a, q_2), (q_2, a, q_3), (q_3, a, q_1), donde los estados q_2 y q_3 son estados nuevos generados para hacer esta transformación.

Con esta transformación se puede pasar de un AFN cualquiera M a un AFN M' equivalente cuyas transiciones tienen a lo más un caracter. Esta transformación es "inofensiva" en el sentido de que no altera el lenguaje aceptado por el AFN. [12]

En el autómata transformado ya es posible explorar simultáneamente todas las trayectorias, tomando nota de todos los estados en los que puede encontrarse el AFN en cada momento. Por ejemplo, para el AFN de la figura 3.7 se tiene el AFN transformado de la figura 3.8.

En este AFN se puede verificar si acepta o no la palabra $baaaaab$. Para ello, vamos llevando registro de los conjuntos de estados en los que podría encontrarse el AFN. Inicialmente, podría encontrarse en el estado inicial q_0, pero sin "gastar" ningun caracter podría estar también en el estado q_1, o sea que el proceso arranca con el conjunto de estados $\{q_0, q_1\}$. Al consumirse el primer caracter, b, se puede pasar de q_0 a q_0 o bien a q_1 (pasando por el ε), mientras que del q_1 sólo se puede pasar a q_1. Entonces, el conjunto de estados en que se puede estar al consumir la b es $\{q_0, q_1\}$. Y así en adelante. La tabla siguiente resume los conjuntos de estados por los que se va pasando para este ejemplo:

[12] *Ejercicio.*- Probar que esta transformación preserva la equivalencia

Entrada	*Estados*
	$\{q_0, q_1\}$
b	$\{q_0, q_1\}$
a	$\{q_2, q_4\}$
a	$\{q_0, q_1, q_3\}$
a	$\{q_1, q_2, q_4\}$
a	$\{q_0, q_1, q_3, q_4\}$
a	$\{q_1, q_2, q_3, q_4\}$
b	$\{q_1\}$

Puesto que el último conjunto de estados $\{q_1\}$ incluye a un estado final, se concluye que la palabra es aceptada.

Formalizando, si queremos definir el conjunto de estados a los que se puede llegar por medio del símbolo σ a partir de un conjunto $Q \subseteq K$, la idea más inmediata sería definir una función $TRANSICION$ de la manera siguiente:

$$TRANSICION(Q, \sigma) = \{q \in K | \exists p \in Q, (p, \sigma, q) \in \Delta\}$$

Sin embargo, esta definición no toma en cuenta el hecho de que a veces es posible tener transiciones que no gastan ningún caracter -aquellas marcadas con ε. Por lo tanto, hay que modificar la definición anterior.

Vamos a definir una función auxiliar *cerr*-$\varepsilon(q)$ que es el conjunto de estados a los que se puede llegar desde el estado q pasando por transiciones vacías. La función *cerr*-$\varepsilon(q)$, llamada *cerradura al vacío*, se puede definir como sigue:

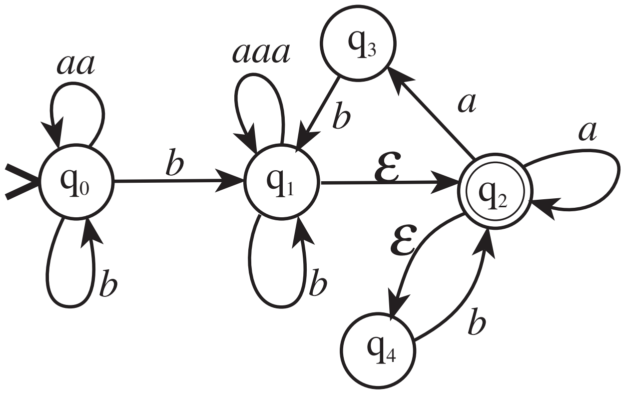
Definición.- La cerradura al vacío *cerr*-$\varepsilon(q)$ de un estado q es el más pequeño conjunto que contiene:

1. Al estado q;
2. Todo estado r tal que existe una transición $(p, \varepsilon, r) \in \Delta$, con $p \in$ *cerr*-$\varepsilon(q)$.

Es fácil extender la definición de cerradura al vacío de un estado para definir la cerradura al vacío de un conjunto de estados:

Definición.- La cerradura al vacío de un conjunto de estados *CERR*-$\varepsilon(\{q_1, \ldots, q_n\})$ es igual a *cerr*-$\varepsilon(q_1) \cup, \ldots, \cup$*cerr*-$\varepsilon(q_n)$.

Ejemplo.- Sea el AFN de la figura siguiente:



Entonces $cerr\text{-}\varepsilon(q_1) = \{q_1, q_2, q_4\}$, y $CERR\text{-}\varepsilon(\{q_1, q_3\}) = \{q_1, q_2, q_3, q_4\}$.

Ahora sí vamos a poder escribir una versión modificada de la función $TRANSICION$ que tome en cuenta las transiciones vacías:

$$TRANSICION-\varepsilon(Q, \sigma) = \{q \in K \mid \exists p, r \in Q, (p, \sigma, r) \in \Delta, q \in cerr-\varepsilon(r)\}$$

O en forma equivalente:

$$TRANSICION-\varepsilon(Q, \sigma) = CERR-\varepsilon(TRANSICION(Q, \sigma))$$

3.6 Equivalencia de AFD Y AFN

Los autómatas finitos determinísticos (AFD) son un subconjunto propio de los no determinísticos (AFN), y son de hecho una extensión de éstos. [13] Podría entonces pensarse que los AFN son "más poderosos" que los AFD, en el sentido de que habría algunos lenguajes aceptados por algún AFN para los cuales no habría ningún AFD que los acepte. Sin embargo, en realidad no sucede así.

Teorema.- Para todo AFN N, existe algún AFD D tal que $\mathcal{L}(N) = \mathcal{L}(D)$.

Este resultado, sorprendente, pero muy útil, puede probarse en forma constructiva, proponiendo para un AFN cómo construir un AFD que sea equivalente.

El método que usaremos para pasar de un AFN a un AFD se basa en la misma idea que se utilizó para comprobar si un AFN acepta o no una palabra, es decir, considerar el conjunto de estados en los que podría encontrarse el AFN al haber consumido una cierta entrada. (Ver sección 3.5.1).

[13] *Ejercicio.*- Justificar este hecho.

La única diferencia es que ahora necesitamos examinar los conjuntos de estados que se forman ante todas las posibles entradas, y no sólamente para una palabra en particular. Para poder ser exhaustivos, necesitamos organizar las entradas posibles de manera sistemática.

Vamos a describir inicialmente el método sobre un ejemplo. Considérese el problema de transformar a AFD el AFN de la figura 3.8. Vamos a considerar el conjunto de estados del AFN en los que podría encontrarse éste en cada momento. El conjunto inicial de estados estará formado por los estados del AFN de la figura 3.8 en los que se pudiera estar antes de consumir el primer caracter, esto es, q_0 y q_1. Dicho conjunto aparece en la figura 3.9(a). A partir de ahí, tras recibir un caracter a, el AFN pudiera encontrarse ya sea en q_2 o en q_4, los cuales incluímos en un nuevo conjunto de estados, al que se llega con una transición con a, como se ilustra en la figura 3.9(b); similarmente, a partir del conjunto inicial de estados $\{q_0, q_1\}$ con la letra b llegamos al mismo conjunto $\{q_0, q_1\}$, lo que se traduce en un "lazo" a sí mismo en la figura 3.9(b). Con este mismo procedimiento, se siguen formando los conjuntos de estados; por ejemplo, a partir de $\{q_2, q_4\}$, con una a se pasa a $\{q_3, q_0, q_1\}$. Continuando así, al final se llega al diagrama de la figura 3.9(c). Si nos alejamos del dibujo de manera que no observemos que son conjuntos de estados, sino que vemos los círculos como *estados*, nos daremos cuenta de que ¡hemos construído un AFD! Unicamente falta determinar cuáles de los nuevos estados son finales y cuales no. Obviamente, si uno de los conjuntos de estados contiene un estado final del antiguo AFN, esto muestra que *es posible que en ese punto el AFN hubiera aceptado la palabra de entrada, si ésta se terminara.* Por lo tanto, los estados finales del nuevo autómata serán aquellos conjuntos de estados que contengan algún estado final. Así, en el AFD de la figura 3.9(d) marcamos los estados finales; además borramos los estados del antiguo AFN de cada uno de los círculos, y bautizamos cada conjunto de estados como un estado.

Ahora bien, se supone que el AFD que hemos construido acepta el mismo lenguaje que el AFN original. Para garantizar la infalibilidad del procedimiento descrito falta aún justificar los siguientes puntos:

1. El procedimiento de construcción del AFD termina siempre
2. El grafo es un AFD
3. El AFD así construído acepta el mismo lenguaje que el AFN original.

La construcción de este grafo tiene que acabarse en algún momento, porque la cantidad de nodos está limitada a un máximo de $2^{|K|}$, donde K son los estados del AFN (¿Porqué?).

El segundo punto se justifica dando la definición completa del AFD: $(K_D, \Sigma, \delta_D, s_D, F_D)$, donde:

- Cada elemento de K_D es uno de los conjuntos de estados que aparecen en el grafo;

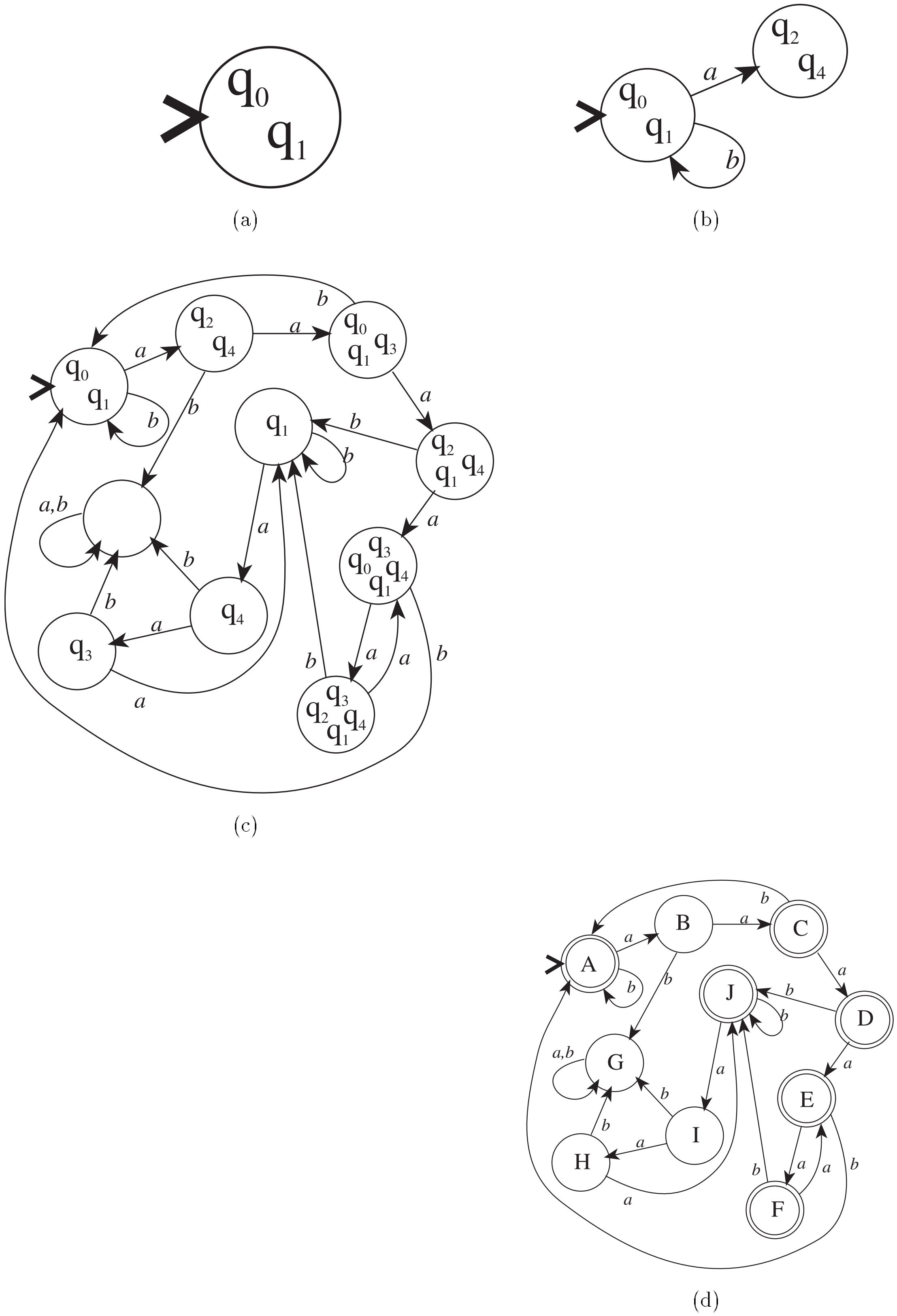


Figura 3.9: Transformación de AFN a AFD

- El alfabeto Σ es el mismo que el del AFN original;
- Hay una tripleta (p, σ, q) en δ_D y sólo una por cada flecha con etiqueta σ que va del conjunto de estados p al conjunto q ;
- El estado inicial s_D del AFD es igual a $cerr\text{-}\varepsilon(s)$, donde s es el estado inicial del AFN;
- F_D es el conjunto de conjuntos de estados tales que en ellos aparece al menos un estado final del AFN.

Finalmente, queda pendiente probar que el AFD (que llamaremos D) acepta el mismo lenguaje que el AFN original $N = (K, \Sigma, \Delta, s, F)$. [14] Esta prueba se puede dividir en dos partes:

$\mathcal{L}(N) \subseteq \mathcal{L}(D)$. Si una palabra $w = \sigma_0\sigma_1 \dots \sigma_n, \sigma_i \in \Sigma \cup \{\varepsilon\}$, es aceptada por N, entonces existe una secuencia estados $q_0, q_1, \dots, q_n$, por los que pasa N en el cálculo:

$$[[q_0, \sigma_0\sigma_1 \dots \sigma_n]] \vdash [[q_1, \sigma_1 \dots \sigma_n]] \vdash \dots \vdash [[q_n, \sigma_n]] \vdash [[q_{n+1}, \varepsilon]]$$

Esta misma secuencia de estados puede seguirse en D, de la manera siguiente (vamos a denotar con Q mayúsculas los "estados" de D):

Iniciamos el recorrido de N en q_0 –su estado inicial– y el recorrido de D en $cerr\text{-}\varepsilon(q_0)$, que es el estado inicial de D. Hay dos posibilidades:

1. Si en N estamos en un estado $q \in K$ –que aparece en $Q \in K_D$ – y se presenta una transición vacía de q a q', en D vamos a permanecer en Q, que va a contener tanto a q como a q'.
2. Si en N estamos en un estado $q \in K$ que aparece en $Q \in K_D$, y de q pasamos a q_σ con el caracter σ, entonces en D pasamos a $Q_\sigma = TRANSICION(Q_D, \sigma)$, que va a ser un "estado" de D que va a contener a q_σ (¿Porqué?).

Siguiendo este procedimiento, cuando la palabra de entrada se acaba al final del cálculo, en una configuración $[[q_f, \varepsilon]]$, $q_f \in K$, habremos llegado en D a un estado $Q_f \in K_D$ que debe contener a q_f, y que por ello es estado final, aceptando así la palabra de entrada.

$\mathcal{L}(D) \subseteq \mathcal{L}(N)$. Se puede seguir el procedimiento inverso al del punto anterior para reconstuir, a partir de un cálculo que acepta w en D, la secuencia de estados necesaria en N para aceptar w. Los detalles se dejan como ejercicio. QED

Ejercicio.- Hacer un programa computacional que transforme un AFN a AFD construyendo la tabla de transición.

[14] Se supone que N ya sufrió la "transformación inofensiva".

3.7 Autómatas finitos con salida

Hasta donde hemos visto, la única tarea que han ejecutado los autómatas finitos es la de aceptar o rechazar una palabra, determinando así si pertenece o no a un lenguaje. Sin embargo, es posible definirlos de manera tal que produzcan una salida diferente de "si" o "no". Por ejemplo, en el contexto de una máquina controlada por un autómata, puede haber distintas señales de salida que correspondan a los comandos enviados a la máquina para dirigir su acción. En los compiladores, el analizador lexicográfico es un autómata finito con salida, que recibe como entrada el texto del programa y manda como salida los elementos lexicográficos reconocidos ("tokens"). Hay dos formas de definir a los autómatas con salida, dependiendo de si la salida se hace depender de las transiciones o bien del estado en que se encuentra el autómata. En el primer caso, se trata de los autómatas de *Mealy*, y en el segundo, de los autómatas de *Moore*.

3.7.1 Máquinas de Moore

Definición.- Una máquina de Moore es un séxtuplo $(K, \Sigma, \Gamma, \delta, \lambda, q_0)$, en donde K, Σ y δ son como en los AFD, y q_0 es el estado inicial; además tenemos a Γ es el alfabeto de salida, y λ, que es una función de K a Γ^*, que obtiene la salida asociada a cada estado.

La salida de una máquina de Moore M ante una entrada $a_1 \dots a_n$ es la concatenación de $\lambda(q_0)\ \lambda(q_1) \dots \lambda(q_n)$, donde $q_i = \delta(q_{i-1}, a_i), a_i \in \Sigma, i \in \aleph$.

Ejemplo.- La siguiente máquina de Moore recibe secuencias de 0 y 1, y cambia su salida (0 a 1 o 1 a 0) cada vez que recibe un 0 a la entrada:

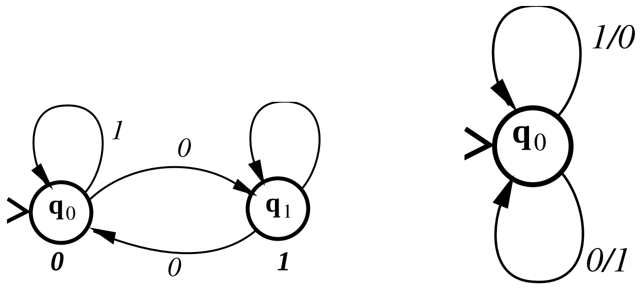
$K = q_0, q_1$, $\Sigma = \Gamma = \{0, 1\}$, $\lambda(q_0) = 0$, $\lambda(q_1) = 1$, y δ está tabulada como:

q	$\delta(q,0)$	$\delta(q,1)$
q_0	q_1	q_0
q_1	q_0	q_1

Las máquinas de Moore se representan gráficamente como cualquier AFD, al que se añade, a cada estado, el símbolo de salida asociado. Por ejemplo, el autómata tabulado arriba se representa en la figura 3.10(a).

3.7.2 Máquinas de Mealy

Definición.- Una máquina de Mealy es un séxtuplo $(K, \Sigma, \Gamma, \delta, \lambda, q_0)$, en el que todos los componentes tienen el mismo significado que arriba, a excepción de λ, que es una función $\lambda : K \times \Sigma \to \Gamma^*$.



(a) Moore (b) Mealy

Figura 3.10: Autómatas de Moore y Mealy

La salida de una máquina de Mealy ante una entrada $a_1 \ldots a_n$ es $\lambda(q_0, a_1)\ \lambda(q_1, a_2)\ \ldots\ \lambda(q_{n-1}, a_n)$, donde $q_i = \delta(q_{i-1}, a_i)$.

Obsérvese que, a diferencia de las máquinas de Moore, en las máquinas de Mealy la salida depende de la entrada, además de los estados. Podemos imaginar que asociamos la salida a las transiciones, más que a los estados.

Ejemplo.- La siguiente máquina de Mealy invierte el flujo de entrada de 0 a 1 y viceversa:

$K = \{q_0\}, \Sigma = \{0, 1\}, \delta(q_0) = q_0$, y $\lambda(q_0, 1) = 0$, $\lambda(q_0, 0) = 1$.

En la representación gráfica de las máquinas de Mealy las etiquetas de las flechas son de la forma a/s, donde a es un caracter de entrada y s es la salida. Por ejemplo, el diagrama para el inversor de Mealy se presenta en la figura 3.10(b).

3.7.3 Equivalencia de las máquinas de Moore y Mealy

Aunque muchas veces, para un mismo problema, la máquina de Mealy es más simple que la correspondiente de Moore, ambas clases de máquinas son equivalentes. Si despreciamos la salida de las máquinas de Moore antes de recibir el primer caracter (o sea, con entrada ε), es posible encontrar, para una máquina de Moore dada, su equivalente de Mealy, en el sentido de que producen la misma salida, y viceversa.

La transformación de una máquina de Moore en Mealy es trivial, pues hacemos $\lambda_{Mealy}(q, a) = \lambda_{Moore}(\delta_{Moore}(q, a))$, es decir, simplemente obtenemos qué salida producirá una transición de Mealy viendo la salida del estado al que lleva dicha transición en Moore.

La transformación de una máquina de Mealy en Moore es más complicada, pues en general hay que crear estados adicionales; remitimos al alumno a la referencia [Hopcroft, Ullman 79].

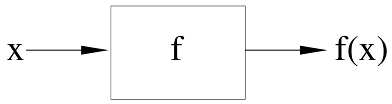
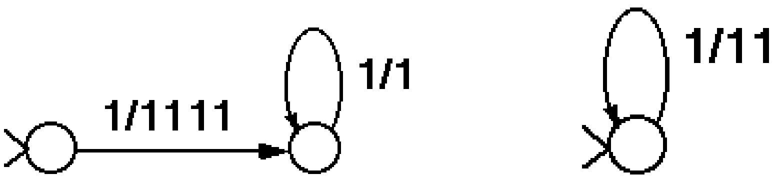


Figura 3.11: Función como “caja negra”



(a) $f(x) = x + 3$ (b) Función $f(x) = 2x$

Figura 3.12: Funciones aritméticas en Mealy

3.7.4 Cálculo de funciones en AF

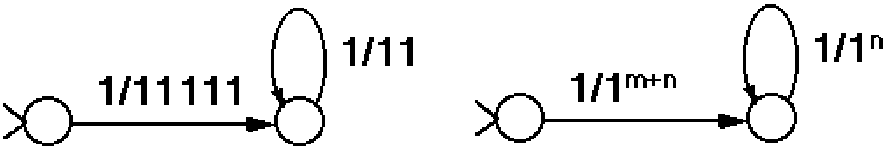
Ya que las máquinas de Mealy y de Moore pueden producir una salida de caracteres dada una entrada, es natural aplicar dichas máquinas al cálculo de *funciones*, donde la función es vista como una forma de relacionar una *entrada*, que es una palabra de un cierto alfabeto Σ, con una *salida*, que es otra palabra formada por caracteres del alfabeto de salida Γ. Podemos así ver una función como una “caja negra”, como se ilustra en la figura 3.11, que a partir del argumento x entrega un resultado $f(x)$.

Ejemplo.- Representamos los números naturales en el sistema unario, es decir, 3 es 111, 5 es 11111, etc. Queremos una máquina de Mealy que calcule la función $f(x) = x + 3$. Esta máquina está ilustrada en la figura 3.12(a). En efecto, al recibirse el primer caracter, en la salida se entregan cuatro caracteres; en lo subsecuente por cada caracter en la entrada se entregan cuatro en la salida, hasta que se acabe la antrada. Debe quedar claro que los tres caracteres que le saca de ventaja la salida al primer caracter de entrada se conservan hasta el final del la entrada; de este modo, la salida tiene siempre tres caracteres más que la entrada, y en consecuencia, si la entrada es x, la salida será $x + 3$.

Sería interesante ver si los AF pueden calcular funciones aritméticas más complejas que la simple suma de una constante. Por ejemplo, ¿se podrá multiplicar la entrada en unario por una constante?

La respuesta es sí. El AF de la figura 3.12(b) entrega una salida que es la entrada multiplicada por dos. Aún más, el AF de la figura 3.13(a) calcula la función $f(x) = 2x + 3$.

Estos resultados pueden ser generalizados para mostrar que una máquina de Mealy puede calcular cualquier función lineal. En efecto, el esquema de AF de la figura 3.13(b) muestra cómo calcular una función $f(x) = nx + m$.



(a) $f(x) = 2x + 3$ (b) $f(x) = nx + m$

Figura 3.13: Funciones lineales en Mealy

Cerca del final de este texto veremos que un AF no puede calcular funciones mucho más complejas que las que hemos visto; ni siquiera pueden calcular la función $f(x) = x^2$.

Formalización del cálculo de funciones

Decimos que una máquina M calcula una función $f : \Sigma^* \to \Sigma^*$ si dada una entrada $x \in \Sigma^*$ la concatenación de los caracteres que entrega a la salida es $y \in \Sigma^*$, donde $y = f(x)$.

La definición anterior puede ser formalizada en términos de las configuraciones y del paso de una configuración a otra. En efecto, la "concatenación de los caracteres a la salida" puede ser tomada en cuenta en la configuración, añadiendo a ésta un argumento adicional en el que se vaya "acumulando" la salida entregada. Esto nos lleva a una definición modificada de configuración.

Definición.- Una configuración de una máquina de Mealy $(K, \Sigma, \Gamma, \delta, \lambda, s)$ es una tercia $[[q, \alpha, \beta]] \in K \times \Sigma^* \times \Gamma^*$, donde q es el estado en que se encuentra el AF, α es lo que resta por leer de la palabra, y β es lo que se lleva acumulado a la salida.

De este modo el funcionamiento del autómata que permite concatenar caracteres a la salida se define de una manera muy simple, utilizando la relación del paso de una configuración a otra, escrita "$\vdash$", como sigue:

Definición.- $[[p, \sigma u, v]] \vdash [[q, u, v\xi]]$ si $q = \delta(q, \sigma)$ y $\xi = \lambda(q, \sigma)$.

Por ejemplo, dado el AF de Mealy de la figura 3.10(b), tenemos que $[[q_0, 101, 0]] \vdash [[q_0, 01, 00]]$.

Utilizando la cerradura transitiva y reflexiva de la relación "$\vdash$", que se denota por "$\vdash^*$", podemos definir formalmente la noción de función calculada:

Definición.- Una máquina $M = (K, \Sigma, \Gamma, \delta, \lambda, s)$ calcula una función $f : \Sigma^* \to \Sigma^*$ si dada una entrada $x \in \Sigma^*$, se tiene:

$$[[s, x, \varepsilon]] \vdash^* [[q, \varepsilon, y]]$$

donde $q \in K$, siempre que $y = f(x)$.

Por ejemplo, para el AF de Mealy de la figura 3.10(b), se pasa de una configuración inicial $[[q_0, 1101, \varepsilon]]$ a una configuración final $[[q_0, \varepsilon, 0010]]$ en cuatro pasos, lo que quiere decir que la función que calcula –sea f– es tal que $f(1101) = 0010$.

3.8 Ejercicios

1. Demostrar la siguiente equivalencia $(ab^*)^*a = a + a(a+b)^*a$ por equivalencia de autómatas.

2. Dados dos autómatas finitos AF_1 y AF_2 ¿Cómo es posible determinar si las palabras que uno de ellos acepta el otro las rechaza? (es decir, $L(AF_1) \cap L(AF_2) = \emptyset$ y $L(AF_1) \cup L(AF_2) = \Sigma^*$). Justifique su respuesta. Hint: El complemento L^c de un lenguaje L aceptado por el AF $(K, \Sigma, \delta, s, F)$, es aceptado por $(K, \Sigma, \delta, s, K - F)$.

3. Demuestre que para algunos lenguajes en 2^{Σ^*} no existe un AFD que los acepte.

4. En comunicaciones digitales, la derivada de un tren de pulsos, p.ej. "0011100", es una señal que tiene "1" en las cifras que cambian, y "0" en las que permanecen constantes, como "0010010" para el ejemplo. Diseñe un autómata de Moore para obtener la derivada de la entrada.

5. Dos AF son iguales, $A == B$ (atención: no se dice equivalentes) cuando la única diferencia entre ellos es el nombre de sus estados, siendo su tamaño y estructura los mismos. Defina formalmente (usando notación de conjuntos) cuándo $(K_1, \Sigma, \delta_1, s_1, F_1) == (K_2, \Sigma, \delta_2, s_2, F_2)$. Hint: hacer un mapeo de estados.

6. Para simplificar un autómata $M = (K, \Sigma, \delta, s, F)$, que tiene dos estados equivalentes $q_i, q_k \in K$, se quiere eliminar uno de ellos, sea q_k. Defina formalmente cada uno de los componentes del autómata M', en que se ha eliminado de M el estado q_k. Ponga especial cuidado en definir las transiciones en M'.

7. Suponga una variante de los autómatas finitos, los autómatas con aceptación (AA), en que, en vez de los estados finales, hay estados de aceptación, tales que si el autómata pasa por uno de ellos, aunque sea una vez, la palabra es aceptada, independientemente de que al final se llegue o no a un estado de aceptación.

   (a) Dibuje un AA que acepte las palabras sobre $\{a, b\}$ que comienzan por "bb" o terminan con "aaa". (Marque los estados de aceptación por nodos $\oplus$).

   (b) Defina formalmente los AA, así como la noción de lenguaje aceptado por un AA, usando para ello la relación entre configuraciones $C1 \vdash C2$. (Evite en sus definiciones el uso de "...").

   (c) Pruebe que los AA son equivalentes a los AF, dando un procedimiento para construir un AF a partir de cualquier AA dado.

(d) Pruebe su procedimiento del inciso anterior transformando el AA del primer inciso a AF.

8. Suponga una variante de los autómatas finitos deterministas, los autómatas con rechazo (AR), en que, además de los estados finales, hay estados de rechazo, tales que si el autómata pasa por uno de ellos, aunque sea una vez, la palabra es rechazada, independientemente de que al final se llegue o no a un estado final. Se supone que si no se pasa por un estado de rechazo, la aceptación de una palabra depende de que al final se llegue a un estado final.

    (a) Dibuje un AR que acepte las palabras sobre $\{a, b\}$ que no contengan las cadenas ªbaab”ni ªba”. Marque los estados de aceptación por nodos $\otimes$.

    (b) Defina formalmente los AR, así como la noción de lenguaje aceptado por un AA, usando para ello la relación entre configuraciones $C_1 \vdash C_2$. (Evite en sus definiciones el uso de "...").

9. Un autómata finito casi determinista (AFCD) es un AFN pero en el cual nunca hay la posibilidad de elegir entre dos caminos a tomar. Los AFCD son de hecho una abreviatura de los AFD, donde se omiten los "infiernos", y se pueden incluir varios caracteres en un arco. Un ejemplo de AFCD está en la siguiente figura 3.14.

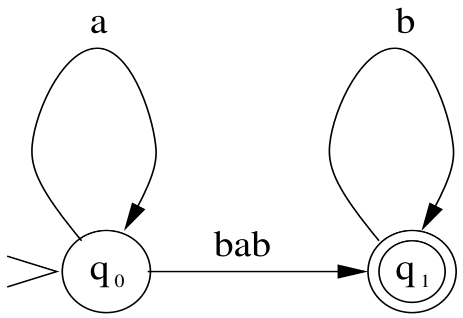


Figura 3.14: Ejemplo de AFCD

¿Es posible probar que un AFN dado es AFCD? Si es así, proponga un método sistemático para probarlo.

10. Suponga unos autómatas no deterministas con salida (AFNDS), en que las flechas son de la forma "w/y", donde "w" y "y" son palabras formadas respectivamente con el alfabeto de entrada y de salida (pueden ser la palabra vacía).

    (a) Defina formalmente los AFNDS.

    (b) Defina formalmente la noción de función calculada.

11. Se dice que un AFNDS es "confluente" cuando la salida obtenida es la misma independientemente de qué trayectoria se siga cuando haya varias opciones.

    (a) Pruebe que todo autómata de Mealy, visto como AFNDS, es confluente.

(b) ¿Es posible decidir si un AFNDS es confluente? Pruebe su respuesta proponiendo un procedimiento para decidir, o mostrando porqué es imposible.

12. Un estado "q" de un AFD es "inaccesible" si no hay ninguna trayectoria que, partiendo del estado inicial, llegue a "q". Esta, sin embargo, no es una definición formal.

    (a) Defina formalmente cuándo un estado "q" es inaccesible, utilizando para ello la relación de paso entre configuraciones.

    (b) Proponga un procedimiento para obtener el conjunto de los estados accesibles en un AFD. (Hint: considere cómo evoluciona el conjunto de "estados accesibles" ante posibles transiciones).

13. Decimos que los lenguajes de dos AFD son "casi iguales" si difieren únicamente en una palabra. Dados M_1 y M_2, un procedimiento para decidir si $L(M_1)$ es casi igual a $L(M_2)$ consiste en:

    (a) Hacer la comparación de M_1 y M_2, y detectar una palabra que acepta M_1 y no M_2, sea w.

    (b) Hacer un AFN que acepte únicamente w, sea M_w.

    (c) Combinar M_2 con M_w, dando un AFN M_2' que acepta $L(M_2)$¿¿w

    (d) Convertir M_2' a AFD, dando M_2''

    (e) Comparar M_2'' con M_1.

    Pruebe la receta anterior con los AFD siguientes:

    $M1 = (\{1,2\},\{a,b\},\{((1,a),2)((1,b),2),((2,a),2),((2,b),2)\},1,\{1\}),$

    $M2 = (\{3,4,5\},\{a,b\},\{((3,a),4),((3,b),5),((4,a),5),((4,b),5),((5,a),5),((5,b),5)\},3,\{4\})$

14. Decimos que un AFN cae en "deadlock" cuando no hay una transición que indique adonde ir ante el símbolo de entrada. Pruebe que es posible/no es posible saber, dado un AFN M en particular, si M podría o no caer en deadlock para alguna palabra w (proponga un método de decisión).

15. Suponga las Expresiones Regulares con Salida (ERS), que son como las ER, pero tienen asociada una salida a la entrada que representan. Se tiene la siguiente sintaxis: ER/S significa que cuando se recibe una palabra representada por ER, se produce una salida S. Las subexpresiones de una ERS se consideran similarmente. Por ejemplo, en la ERS "$(a/1+b/0)^*/00$", por cada "a" que se recibe se saca un "1"; por cada "b", un "0", y al terminarse la cadena de entrada se produce un "00". El operador "/" tiene la precedencia mas alta, o sea que la ERS "ab/0" significa que el "0" está asociado a la "b"; puede ser necesario usar parentesis para establecer la precedencia deseada. En general hay que distinguir entre el alfabeto de entrada ($\{a,b\}$ en el ejemplo) y el de salida ($\{0,1\}$ en el ejemplo).

(a) Defina una ERS que al recibir cada par de "aa" consecutivas emita un "0", mientras que al recibir un par de "bb" consecutivas emita un "1". ("aaa" contiene sólo un par).

(b) Defina formalmente el conjunto de las ERS (Las ERS que se definirían son las ERS "formales", con todos los parentesis y operadores necesarios, sin tomar en cuenta cuestiones de precedencia de operadores, simplificaciones, etc.).

(c) Proponga un procedimiento general para pasar de ERS a autómatas de Mealy.

(d) Muestre su funcionamiento con la ERS del inciso (a).

16. Diseñar un autómata de Mealy o de Moore que recibe un flujo de "1" y "0", y cada vez que recibe una secuencia "11" la reemplaza por "00".

Capítulo 4

Propiedades de los lenguajes regulares

En esta sección estudiaremos algunas características de los lenguajes regulares, y pondremos en relación sus representaciones vistas precedentemente, es decir, las expresiones regulares, los autómatas finitos y las gramáticas regulares.

4.1 Equivalencia de expresiones regulares y autómatas finitos

Aún cuando por varios ejemplo hemos visto que lenguajes representados por expresiones regulares son aceptados por autómatas finitos, no hemos probado que para cualquier expresión regular exista un autómata finito equivalente, y viceversa. Esto se establece en el siguiente

Teorema.- Un lenguaje es regular ssi es aceptado por algún autómata finito.

Prueba de la parte "sólo si".- La prueba de la parte "sólo si" de este teorema consiste en dar un procedimiento para transformar en forma sistemática una expresión regular en un autómata finito que acepte su lenguaje. Dicho procedimiento se describe a continuación:

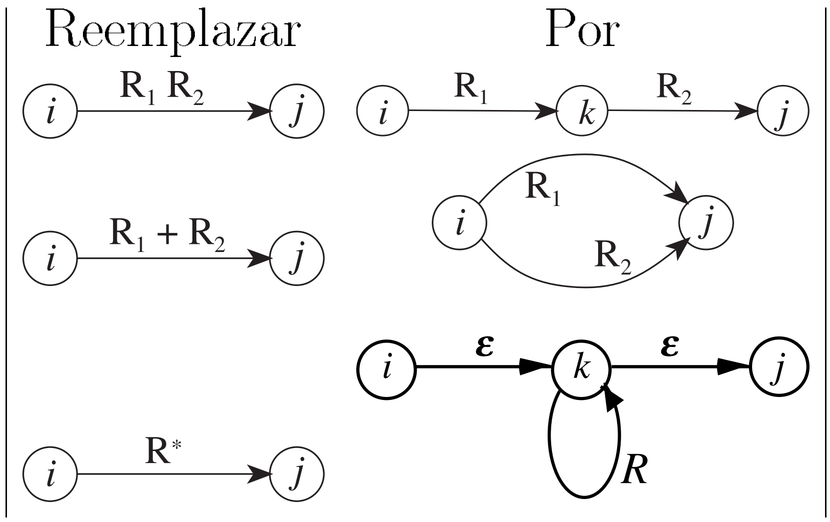
La idea es hacer una *transformación gradual* que vaya conviertiendo la ER en AF.

Para hacer la transformación gradual de ER a AFN se requiere alguna representación de los lenguajes regulares intermedia entre las ER y los AFN.

Dicho "eslabón perdido" son las *gráficas de transición*. Estas últimas son esencialmente AFN en que las etiquetas de las flechas tienen expresiones regulares, en lugar de palabras. Las gráficas de transición (GT) son por lo tanto quíntuplos $(K, \Sigma, \Delta, s, F)$ en donde $\Delta \in K \times ER \rightarrow K$. [1]

[1]Ejercicio: proponer una definición formal de configuración, cálculo y palabra aceptada para las GT.

Tabla 4.1: Eliminación de operadores para pasar de ER a AF



A partir de una ER es trivial obtener una GT que acepte el mismo lenguaje. En efecto, sea R una ER; entonces, si

$$G_1 = (\{q_0, q_1\}, \Sigma, \{(q_0, R, q_1)\}, q_0, \{q_1\})$$

entonces $L(G) = L(R)$.

Los AFN son un subconjunto propio de las GT, puesto que las palabras en las etiquetas de un AFN pueden ser vistas como expresiones regulares que se representan a sí mismas. Por lo tanto, lo que falta por hacer es *transformar gradualmente* G_1 en G_2, luego en G_3, etc., hasta llegar a un G_n tal que en las flechas no haya más que caracteres sólos (o bien la palabra vacía). En efecto, $G_n \in$AFN. Este es un proceso de *eliminación gradual* de los operadores de las ER.

Para eliminar los operadores de las ER en G_i, aplicamos reemplazamientos de ciertas transiciones por otras, hasta que no sea posible aplicar ninguno de estos reemplazamientos. Este proceso se ilustra en la tabla 4.1.

Ejemplo.- Dada la ER $(a + ba)^*bb$, obtener el AFN que acepta el lenguaje de dicha ER.

Solución: Aplicamos una sucesión de transformaciones, ilustradas en las figuras 4.1(a)-(d). [2]

La equivalencia de G_1, G_2, ..., G_n se asegura por el hecho de que cada una de las transformaciones preserva la equivalencia.

[2]Ejercicio: hacer este mismo ejemplo, pero en la representación formal de las GT, en vez de las figuras.

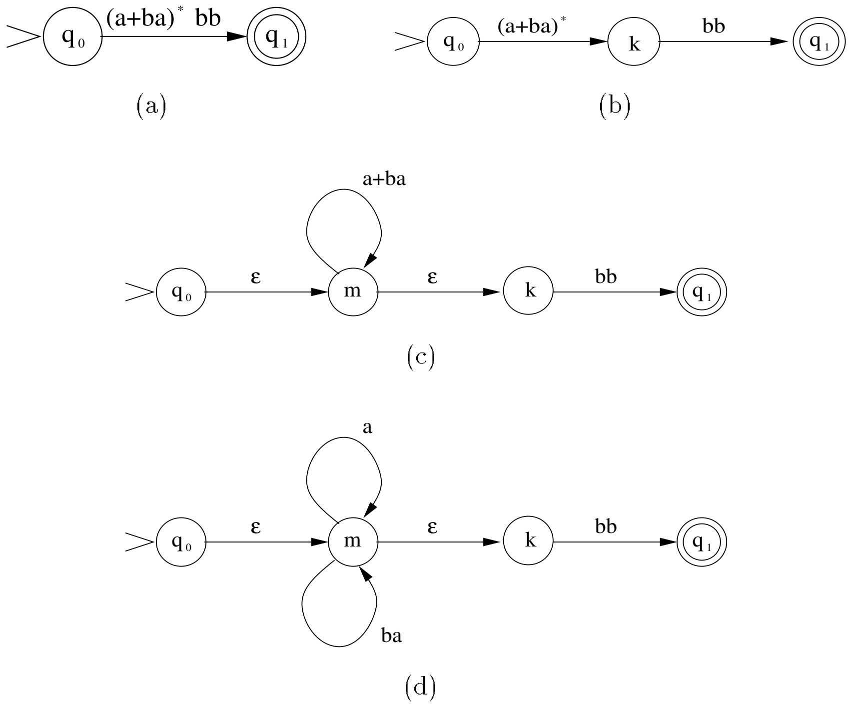


Figura 4.1: Transformación ER$\rightarrow$AF

Prueba de la parte "si"

La prueba de la parte "si" del teorema consiste en dar un procedimiento para transformar en forma sistemática un autómata finito en una expresión regular equivalente. Un procedimiento para hacerlo consiste en ir eliminando gradualmente nodos de una GT, que inicialmente es el AFN que se quiere transformar, hasta que sólo queden un nodo inicial y un nodo final.

Dicho procedimiento comprende los siguientes pasos:

1. El primer paso en este procedimiento consiste en añadir al autómata finito un nuevo estado inicial i, mientras que el antiguo estado inicial q_0 deja de ser inicial, y un nuevo estado final f, mientras que los antiguos estados finales $q_i \in F$ dejan de ser finales; además se añade una transición vacía del nuevo estado inicial al antiguo, (i, ε, q_0), y varias transiciones de los antiguos estados finales al nuevo: $\{(q_i, \varepsilon, f) | q_i \in F\}$. Esta transformación tiene por objeto que haya un sólo estado inicial, al que no llegue ninguna transición, y uno sólo final, del que no salga ninguna transición. Esta condición se requiere para llevar a cabo el siguiente paso.

2. El segundo paso consiste en eliminar nodos intermedios en la GT. El procedimiento de eliminación de nodos intermedios es directo. En la figura 4.2 se representa un nodo

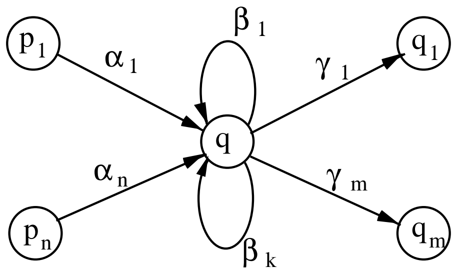


Figura 4.2: Nodo a eliminar

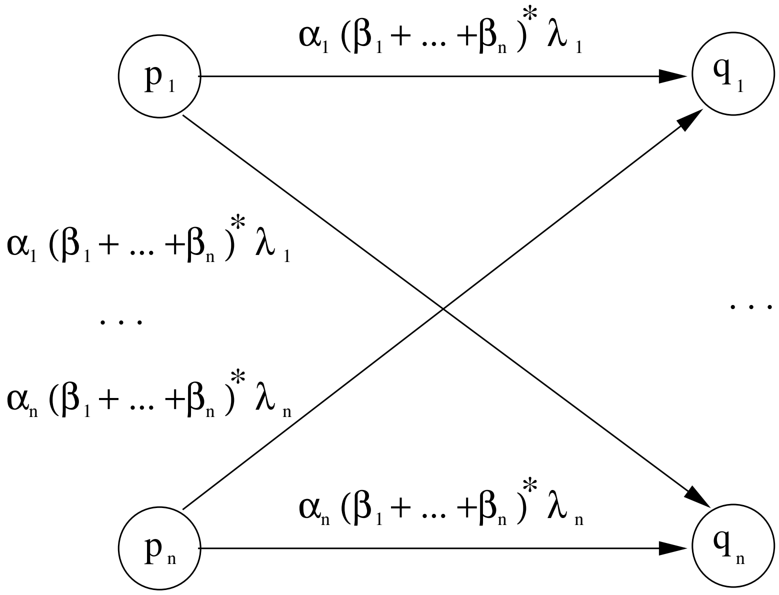


Figura 4.3: GT sin el nodo eliminado

intermedio que se quiere eliminar, q, y los nodos entre los que se encuentra. Este esquema se adapta a todos los casos que pueden presentarse. En dicha figura, α_i,β_i, γ_i son expresiones regulares. Para eliminiar el nodo q, reemplazamos la parte de la GT descrita en la figura 4.2 por el subgrafo representado en la figura 4.3.

3. Cuando el paso 2 esté terminado, tendremos una GT de la forma de la figura 4.4. Esta GT se puede transformar en otra con una sola transición, fusionando todas las transiciones en una sola, con etiqueta $R_1 + R_2 + \ldots + R_n$. Esta etiqueta será la ER buscada.

La corrección de estos pasos se desprende del hecho de que tanto la eliminación de nodos como la fusión de transiciones que se hace al final, preservan ambos la equivalencia.

Ejemplo.- Obtener una ER para el AFD de la figura siguiente:

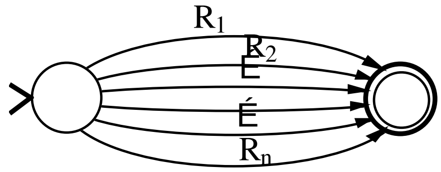
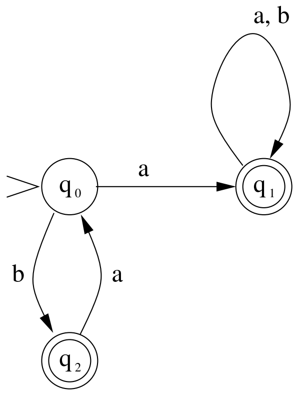
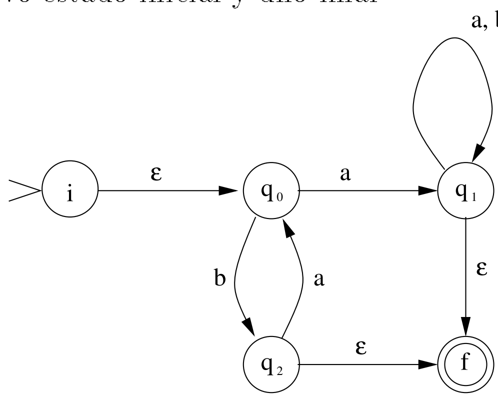


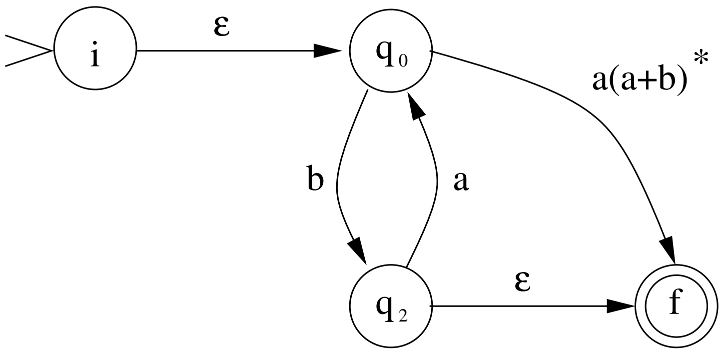
Figura 4.4: GT tras la eliminación de nodos intermedios



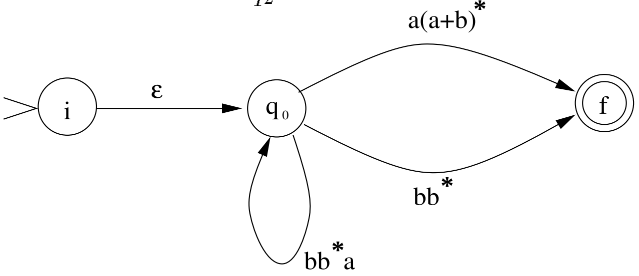
Paso 1.- Añadir un nuevo estado inicial y uno final



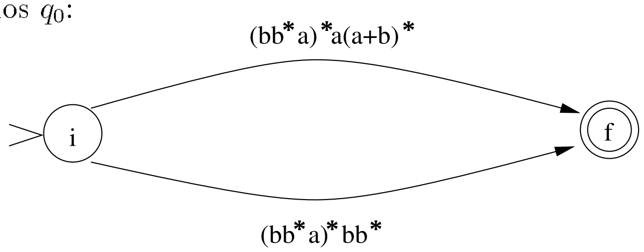
Paso 2.- Eliminación de nodos intermedios. Eliminamos primero el nodo q_1. Una manera sencilla de conceptualizar la eliminación de nodos, tal vez más intuitiva que la aplicación directa del diagrama de la figura 4.2 consiste en considerar qué trayectorias o "rutas" pasan por el nodo a eliminar. Por ejemplo, en la figura de arriba vemos sólamente una trayectoria que pasa por q_1, la cual va de q_0 a f. Ahora nos proponemos eliminar el nodo q_1, pero sin modificar "lo que se gasta" para pasar de q_0 a f. Es fácil ver que para pasar de q_0 a f se gasta primero una a y luego algún número de repeticiones de a o b (para llegar de q_1 a f no se gasta nada). Esto corresponde a la ER $a(a+b)^*$, que será la etiqueta de la nueva "ruta directa" de q_0 a f, sin pasar, por q_1, como se aprecia en la siguiente figura:



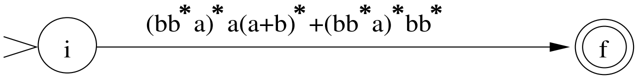
Paso 3.- Después elminamos el nodo q_2:



Paso 4.- Eliminamos q_0:



Paso 5.- Finalmente fusionamos las expresiones que están en paralelo:



Por lo que finalmente la ER buscada es $(bb^*a)^*a(a+b)^* + (bb^*a)^*bb^*$.

Con este resultado establecemos la completa equivalencia entre las ER y los autómatas finitos (no deterministas). Al establecer la equivalencia de los AFN con las ER, automáticamente queda establecida la equivalencia entre las ER y los AFD. Este es un resultado de gran trascendencia tanto teórica como práctica, pues por una parte muestra la importancia de la clase de los lenguajes regulares, y por otra ofrece un grupo de herramientas prácticas, tales como la minimización de AFD, que pueden ser puestas al servicio de las ER.

4.2 Gramáticas regulares

En esta sección veremos otra manera de representar los lenguajes regulares, además de las Expresiones Regulares que ya vimos.

4.2.1 Gramáticas formales

La representación de los lenguajes regulares que aquí estudiaremos se fundamenta en la noción de *gramática formal*. Intuitivamente, una gramática es un conjunto de reglas para formar correctamente las frases de un lenguaje; así tenemos la gramática del español, del francés, etc. La formalización que presentaremos de la noción de gramática es debida a N. Chomski, y está basada en las llamadas *reglas*.

Una regla es una expresión de la forma $\alpha \to \beta$, en donde tanto α como β son cadenas de símbolos en donde pueden aparecer tanto elementos del alfabeto Σ como unos nuevos símbolos, llamados *variables*. [3] La aplicación de una regla $\alpha \to \beta$ a una palabra $u\alpha v$ produce la palabra $u\beta v$. En consecuencia, las reglas de una gramática pueden ser vistas como reglas de reemplazo.

4.2.2 Gramáticas regulares

Nosotros nos vamos a interesar por el momento en las gramáticas cuyas reglas son de la forma $A \to aB$ o bien $A \to a$, donde A y B son variables, y a es un caracter terminal. A estas gramáticas se les llama *regulares*. Formalizamos esta noción con la siguiente

Definición.- Una gramática regular es un cuádruplo (V, Σ, R, S) en donde:

V es un alfabeto

Σ es un subconjunto de V, que contiene a los símbolos *terminales*.

R, el conjunto de reglas, es un subconjunto finito de $(V - \Sigma) \times [\Sigma(V - \Sigma) \cup \Sigma]$.

S, el símbolo inicial, es un elemento de $V - \Sigma$.

Usualmente las reglas no se escriben como pares ordenados (A, aB), sino como $A \to aB$; esto es simplemente cuestión de notación.

Las reglas permiten establecer una relación entre cadenas en V^*, que es la *relación de*

[3] Tratándose de los compiladores, se les llama "terminales" a los elementos de Σ, y "no terminales" a las variables.

derivación, $\Rightarrow_G$, para una gramática G. Esta relación se define de la siguiente manera:

Definición.- $\alpha \Rightarrow_G \beta$ (leído "α deriva en un paso β") ssi existen cadenas $x, y \in V^*$, tales que $\alpha = xuy$, $\beta = xvy$, y existe una regla $u \rightarrow v \in R$. [4]

Cuando se sobreentiende que nos estamos refiriendo a una gramática G, el subíndice de $\Rightarrow_G$ se puede omitir, quedando simplemente como $\Rightarrow$.

La cerradura reflexiva y transitiva de $\Rightarrow$ se denota por $\stackrel{*}{\Rightarrow}$. Una palabra $w \in V^*$ es *derivable* a partir de G si existe una secuencia de derivación $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow w$, es decir, $S \stackrel{*}{\Rightarrow} w$.

Definición.- El *lenguaje generado* por una gramática G, $L(G)$, es igual a $\{w \in \Sigma^* | S \stackrel{*}{\Rightarrow} w\}$.

Ejemplo.- Sea la gramática G con las siguientes reglas (Σ, V y S quedan sobreentendidas):

1. $S \rightarrow aA$
2. $S \rightarrow bA$
3. $A \rightarrow aB$
4. $A \rightarrow bB$
5. $A \rightarrow a$
6. $B \rightarrow aA$
7. $B \rightarrow bA$

Se puede probar [5] que $S \stackrel{*}{\Rightarrow} bababa$, por lo que $bababa \in L(G)$. De hecho, $L(G)$ es el conjunto de las palabras sobre $\{a, b\}$ de longitud par y terminadas en a.[6]

4.2.3 Autómatas finitos y gramáticas regulares

Similarmente a como hicimos en la sección anterior, aquí vamos a establecer la equivalencia entre las gramáticas regulares y los lenguajes regulares -y por ende los autómatas finitos. Este resultado es establecido por el siguiente

[4] Esta definición no está restringida a las gramáticas regulares.
[5] Se deja como ejercicio al lector.
[6] Más adelante veremos cómo hacer este tipo de pruebas.

Teorema.- La clase de los lenguajes generados por alguna gramática regular es exactamente la de los lenguajes regulares.

La prueba de este teorema consiste en proponer un procedimiento para, a partir de una gramática dada, construír un autómata finito, y viceversa.

Dicho procedimiento es directo, y consiste en asociar a los símbolos no terminales de la gramática (las variables) los estados de un autómata. Así, para cada regla $A \to bC$ en la gramática tenemos una transición (A, b, C) en el autómata.

Sin embargo, queda pendiente el caso de las reglas $A \to b$. Para estos casos, se tienen transiciones (A, b, Z), donde Z es un nuevo estado para el que no hay un no terminal asociado; Z es el único estado final del autómata.

Ejercicio.- Describir formalmente la construcción del autómata $(K, \Sigma, \Delta, s, F)$ a partir de la gramática regular (V, Σ, R, S).

Ejemplo.- Obtener un autómata finito para la gramática regular G siguiente:

1. $S \to aA$
2. $S \to bA$
3. $A \to aB$
4. $A \to bB$
5. $A \to a$
6. $B \to aA$
7. $B \to bA$

Dicho autómata aparece en la figura 4.5.

Similarmente, es simple obtener, a partir de un AFD $(K, \Sigma, \delta, s, F)$, la gramática regular correspondiente. Para cada transición de la forma $((p, \sigma), q) \in \delta$, habrá en la gramática una producción $X_p \to \sigma X_q$, donde X_i es la variable en la gramática que corresponde al estado i del AFD. Queda, sin embargo, pendiente cómo obtener las reglas de la forma $X_p \to \sigma$, que son las que permiten terminar una derivación. Nos damos cuenta de que la aplicación de este tipo de producciones debe corresponder al consumo del último caracter de una palabra aceptada en el AFD. Ahora bien, al terminar una palabra aceptada en un AFD, necesariamente nos encontraremos en un *estado final*. De ahí concluímos que hay que incorporar a la gramática, por cada transición $((p, \sigma), q)$, donde $q \in F$, una regla adicional $X_p \to \sigma$, además de la regla $X_p \to \sigma X_q$ que se mencionó anteriormente.

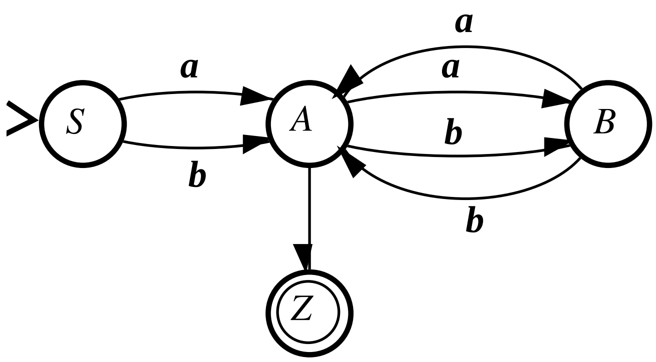


Figura 4.5: Autómata obtenido de la gramática

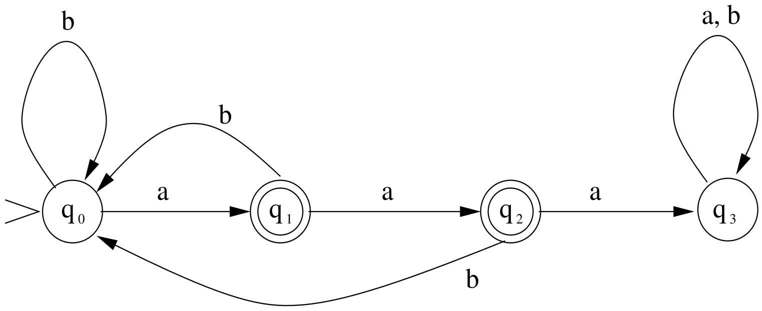


Figura 4.6: AFD que acepta palabras que no contienen 3 a's seguidas

Ejemplo.- Para el AFD de la figura 4.6, la gramática regular correspondiente contiene las reglas:

1.- $Q_0 \to aQ_1$	8.- $Q_3 \to bQ_3$
2.- $Q_0 \to bQ_0$	9.- $Q_0 \to a$
3.- $Q_1 \to aQ_2$	10.- $Q_0 \to b$
4.- $Q_1 \to bQ_0$	11.- $Q_1 \to a$
5.- $Q_2 \to aQ_3$	12.- $Q_1 \to b$
6.- $Q_2 \to bQ_0$	13.- $Q_2 \to b$
7.- $Q_3 \to aQ_3$	

Ejercicio.- Hacer la prueba de corrección de esta gramática. Esto proveerá una prueba de corrección del AFD de la figura 4.6.

4.3 Características de los lenguajes aceptados por los AF

La clase $\mathcal{R}$ de los lenguajes aceptados por los autómatas finitos es el conjunto de lenguajes para los cuales existe un autómata finito que los acepta. [7] Un aspecto muy interesante de $\mathcal{R}$ es que, aplicando varias operaciones comunes a sus elementos, como union, intersección, etc., obtenemos nuevos lenguajes que están también en $\mathcal{R}$. Esta propiedad de que nos mantengamos dentro de $\mathcal{R}$ a pesar de hacer operaciones con sus elementos se llama cerradura. Por ejemplo, decir que $\mathcal{R}$ es cerrado con respecto a la unión quiere decir que si $L_1, L_2 \in \mathcal{R}$ entonces $(L_1 \cup L_2) \in \mathcal{R}$. En general, un conjunto C es cerrado con respecto a la operación $\heartsuit$ si: [8]

$$x, y \in C \Rightarrow (x \heartsuit y) \in C$$
$$x \in C \Rightarrow \heartsuit x \in C$$

Teorema.- La clase $\mathcal{R}$ de los lenguajes aceptados por los autómatas finitos está cerrada respecto a:

- La unión
- La concatenación
- La cerradura de Kleene (*)
- El complemento
- La intersección

La prueba de cada una de las partes de este teorema se basa en la idea de suponer que tenemos autómatas que aceptan L_1 y L_2, sean M_1 y M_2 respectivamente; entonces hay que mostrar una manera de combinar M_1 y M_2 para que dicha combinación acepte el lenguaje $L_1 \heartsuit L_2$, donde $\heartsuit$ es la operación sobre la que se hace la cerradura.

4.3.1 Cerradura respecto a la unión

Sean $M_1 = (K_1, \Sigma_1, \Delta_1, s_1, F_1)$ y $M_2 = (K_2, \Sigma_2, \Delta_2, s_2, F_2)$ dos autómatas que aceptan los lenguajes $L_1, L_2 \in \mathcal{R}$ [9] Podemos entonces construir un AFN M_3 que acepte $L_1 \cup L_2$ de la

[7] Pregunta: ¿$\mathcal{R}$ es contable o incontable?

[8] En la primera fórmula $\heartsuit$ es un operador binario, mientras que en la segunda es un operador unario

[9] Sin pérdida de generalidad podemos suponer que K_1 y K_2 son disjuntos.

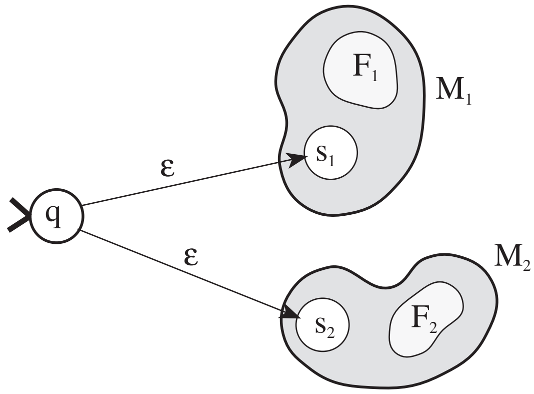


Figura 4.7: Cerradura respecto a la unión

siguiente manera: Sea q un nuevo estado que no está en K_1 ni en K_2. Entonces hacemos un autómata M_3 cuyo estado inicial es q, y que tiene transiciones vacías de q a s_1 y a s_2. Esta simple idea le permite escoger en forma no determinista entre irse al autómata M_1 o a M_2, según el que convenga: si la palabra de entrada w está en L_1, entonces escogemos irnos a M_1 y viceversa para L_2.

Formalmente $M_3 = (K_1 \cup K_2 \cup \{q\},\ \Sigma_1 \cup \Sigma_2,\ \Delta_1 \cup \Delta_2 \cup \{(q, \varepsilon, s_1), (q, \varepsilon, s_2)\},\ q,\ F_1 \cup F_2)$. En la figura 4.7 se representa gráficamente M_3.

Otra prueba de la cerradura de los lenguajes regulares respecto a la unión utiliza el operador "+" de las expresiones regulares; en efecto, si L_1 y L_2 son regulares, representados por las ER R_1 y R_2, se sabe que $R_1 + R_2$ representa $L_1 \cup L_2$.

Ejemplo.- Mostrar que es regular el lenguaje de las palabras en $\{a, b\}$ que empiezan o terminan en a. La solución $(a + \wedge)(a + b)^*(a + \wedge)$ no funciona porque representa palabras que ni comienzan ni terminan en a. Una solución muy simple consiste en definir una ER para el lenguaje de las palabras que comienzan en a y otra para las palabras que terminan en a, y luego simplemente combinarlas con el "+", es decir, $a(a + b)^* + (a + b)^*a$.

4.3.2 Cerradura respecto a la concatenación

Similarmente al caso anterior, sean $M_1 = (K_1, \Sigma_1, \Delta_1, s_1, F_1)$ y $M_2 = (K_2, \Sigma_2, \Delta_2, s_2, F_2)$ dos autómatas que aceptan los lenguajes $L_1, L_2 \in \mathcal{R}$ respectivamente. Podemos entonces construir un AFN M_3 que acepte L_1L_2 de la siguiente manera: Añadimos unas transiciones vacías que van de cada uno de los estados finales de M_1 al estado inicial de M_2; también se requiere que los estados finales de M_1 dejen de serlo.

Formalmente $M_3 = (K_1 \cup K_2,\ \Sigma_1 \cup \Sigma_2,\ \Delta_1 \cup \Delta_2 \cup \{(p, \varepsilon, s_2) | p \in F_1\},\ s_1,\ F_2)$

El funcionamiento de M_3 es como sigue: cuando se recibe una palabra $w = w_1w_2$, $w_1 \in L_1$,

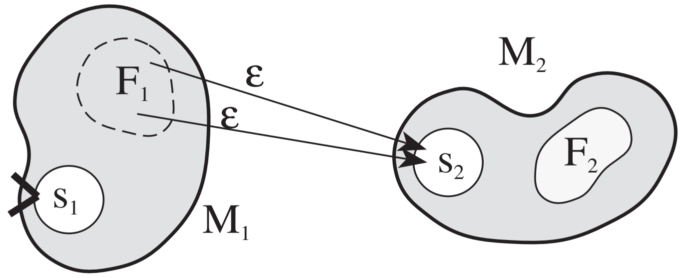


Figura 4.8: Cerradura respecto a la concatenación

$w_2 \in L_2$, entonces se empieza procesando w_1 exactamente como lo haría M_1, hasta llegar hasta alguno de los antiguos estados finales de M_1; entonces se empieza procesando w_2 como lo haría M_2; forzosamente debe ser posible llegar a un estado final de M_2, ya que por hipótesis M_2 acepta w_2. En la figura 4.8 se representa M_3.

Tomando como base las ER, es fácil ver que la concatenación de dos lenguajes regulares L_1 y L_2, representados por las ER R_1 y R_2 es simplemente R_1R_2, que evidentemente es regular.

Ejemplo.- Proponer una ER que represente las palabras que son representaciones "correctas" de números naturales pares. *Solución.-* Vamos a considerar que los numeros pares son naturales terminados en 0, 2, 4, 6 o 8. Esto sugiere que se concatene la ER para los naturales -sea $< NAT >$- con la ER $(0 + 2 + 4 + 6 + 8)$. Por otra parte, los naturales son cualquier cadena en $\{0, 1, \ldots 9\}$ que no empieza en 0. Su ER se puede obtener simplemente concatenando una cifra que no sea 0 con cualquier cadena hecha de cualquier cifra. Entonces la solución final es:

$$(1+2+3+4+5+6+7+8+9)(0+1+2+3+4+5+6+7+8+9)^*(0+2+4+6+8)$$

4.3.3 Cerradura respecto a la estrella de Kleene

El autómata que acepta $\mathcal{L}(M_1)^*$ se muestra en la figura 4.9; su formalización se deja como ejercicio. [10]

Considerando las ER, trivial que la cerradura de Kleene L^* de un lenguaje regular L es regular, pues si L está representado por la ER R, L^* lo está por R^*.[11]

[10] ¡Atención! ¿Porqué se requiere un nuevo estado inicial, en vez de mandar flechas directamente de los estados finales al estado inicial original?

[11] ¡Cuidado! el símbolo "$*$" en L^* representa la operación de cerradura de Kleene, mientras que el "$*$" de R^* es sólo un símbolo en una ER.

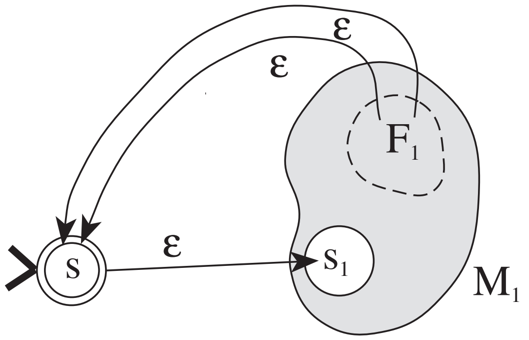


Figura 4.9: Cerradura respecto a la estrella de Kleene

4.3.4 Cerradura respecto al complemento

Si $M = (K, \Sigma, \Delta, s, F)$ es un autómata *determinista* que acepta $L \in \mathcal{R}$, entonces es posible contruír un autómata M^c que acepte el lenguaje complemento de L, esto es, $\Sigma^* - L$. En efecto, basta con intercambiar los estados finales de M en no finales y viceversa. Formalmente, $M^c = (K, \Sigma, \Delta, s, K - F)$. Así, cuando una palabra es rechazada en M, ella es aceptada en M^c y viceversa.[12]

Ejemplo.- Obtener un AF para el lenguaje en $\{a, b\}^*$ de las palabras que no contienen la cadena "*abaab*".[13] *Solución.*- Primero obtenemos un AFN para el lenguaje cuyas palabras sí contienen la cadena "*abaab*", que es simplemente el AFN (sea M_1) correspondiente a la ER $(a + b)^*abaab(a + b)^*$, utilizando para esto el procedimiento de conversión de ER a AFN. Luego convertimos M_1 a AFD por el procedimiento de los conjuntos de estados, obteniendo un AFD $M_2 = (K, \Sigma, \delta, s, F)$. Finalmente obtenemos el AF que acepta el complemento de M_2, esto es, $M_3 = (K, \Sigma, \delta, s, K - F)$.[14]

4.3.5 Cerradura respecto a la intersección

La cerradura de $\mathcal{R}$ respecto a la intersección se desprende directamente de su cerradura respecto al complemento y a la unión, ya que, por propiedades de las operaciones de conjuntos, tenemos:

$$L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$$

Esta fórmula sugiere un procedimiento práctico para obtener un AF que acepte la intersección de dos lenguajes dados. Esto se ilustra en el siguiente ejemplo.

[12] *Ejercicio.*- ¿Porqué es importante que el AF original sea determinista?
[13] Este ejercicio ya había sido propuesto antes, como un ejercicio *muy difícil.*
[14] Ejercicio: calcular en detalle los autómatas M_1, M_2 y M_3.

Ejemplo.- Obtener un AF para el lenguaje en el alfabeto $\{a, b\}$ en que las palabras son de longitud par y además contienen un número par de a. Este problema parece bastante difícil, pero se vuelve fácil utilizando la fórmula de intersección de lenguajes. En efecto, empezamos calculando los AFD para los lenguajes que cumplen independientemente las dos condiciones, es decir:

$$M_1 = AFD(AFN(((a + b)(a + b))^*)) \text{ para las palabras pares,}$$

$$M_2 = AFD(AFN((b^*ab^*ab^*)^*)) \text{ para numero par de } a$$

donde AFN obtiene el AFN a partir de una ER, y AFD convierte un AFN a AFD. Ahora obtenemos los AFD que aceptan el complemento de los lenguajes de M_1 y M_2; sean $COMP(M_1)$ y $COMP(M_2)$. Combinamos estos autómatas utilizando el procedimiento para la unión de lenguajes, dando un AFN M_3, el cual es convertido a AFD y finalmente "complementado", dando $COMP(AFD(M_3))$, que es el autómata buscado.

4.4 Limitaciones de los lenguajes regulares

Por el hecho de que los autómatas finitos "recuerdan" una cantidad limitada de información, no son capaces de reconocer ciertos lenguajes. Por ejemplo, para el lenguaje $\{a^n b^n\}$ no es posible construir un autómata finito, ni representarlo por una expresión regular o gramática regular. En efecto, las palabras son de la forma $aaa \ldots aabbb \ldots bb$, y el autómata debe recordar, al terminar el grupo de as, cuántas encontró, para poder comparar con el número de bs. Ahora bien, como la cantidad de as que puede haber en la primera mitad de la palabra es arbitraria, dicha cantidad no puede recordarse con una cantidad de memoria fija, como es la de los autómatas finitos.

4.4.1 El teorema de bombeo

Formalmente, vamos a establecer un teorema que precisa cuál es la limitación de los autómatas finitos.

Teorema.- Si L es un lenguaje regular infinito, entonces existen cadenas x, y, z tales que $y \neq \varepsilon$, y $xy^n z \in L$, para algún $n \geq 0$. (Teorema de bombeo).

Lo que este resultado establece es que, suponiendo que cierto lenguaje es regular, entonces forzosamente dicho lenguaje debe contener palabras en que una subcadena se repite cualquier número de veces, y estas palabras pueden ser indeseables en el lenguaje considerado. Es decir, hay palabras del lenguaje en que podemos insertar repetidamente ("bombear") una subcadena (y en el teorema) sin que el autómata se dé cuenta. Esta situación permite hacer pruebas por contradicción de que un lenguaje dado no es regular.

Pero veamos en primer lugar la prueba del teorema de bombeo. Supongamos que L es un lenguaje regular. Entonces existe un autómata M que lo acepta. Sea m el número de estados de M. Ahora supongamos una palabra en L, $w = \sigma_1\sigma_2 \ldots \sigma_n$, $\sigma_i \in \Sigma$, donde $n \geq m$. Como w debe ser aceptada, debe hacer un cálculo de la forma:

$$[[q_1, \sigma_1\sigma_2 \ldots \sigma_n]] \vdash_M [[q_2, \sigma_2 \ldots \sigma_n]] \vdash_M^* [[q_{n+1}, \varepsilon]]$$

Como M tiene sólamente m estados, y el cálculo tiene longitud $n + 1$, por el principio de correpondencia debe haber algunos estados que se repitan en el cálculo, es decir, $q_i = q_j$, para $0 \leq i < j \leq n + 1$. Entonces podemos detallar más el calculo anterior, el cual tiene la forma:

$$[[q_1, \sigma_1\sigma_2 \ldots \sigma_i \ldots \sigma_n]] \vdash_M^* [[q_i, \sigma_i \ldots \sigma_n]] \vdash_M^* [[q_j, \sigma_j \ldots \sigma_n]] \vdash_M^* [[q_{n+1}, \varepsilon]]$$

Como M regresa al mismo estado, la parte de la entrada que se consumió entre q_i y q_j, que es $\sigma_i \ldots \sigma_{j-1}$ puede ser eliminada, y por lo tanto la palabra $\sigma_1 \ldots \sigma_{i-1}\sigma_j \ldots \sigma_n$ será aceptada de todas maneras, mediante el cálculo siguiente:

$$[[q_1, \sigma_1 \ldots \sigma_{i-1}\sigma_j \ldots \sigma_n]] \vdash_M^* [[q_j, \sigma_j \ldots \sigma_n]] \vdash_M^* [[q_{n+1}, \varepsilon]]$$

De igual manera, la subcadena $\sigma_i \ldots \sigma_{j-1}$ puede ser insertada cualquier número de veces; entonces el autómata aceptará las palabras de la forma:

$$\sigma_1\sigma_2 \ldots \sigma_{i-1}(\sigma_i \ldots \sigma_{j-1})^k\sigma_j \ldots \sigma_n$$

Entonces, haciendo $x = \sigma_1\sigma_2 \ldots \sigma_{i-1}$, $y = \sigma_i \ldots \sigma_{j-1}$ y $z = \sigma_j \ldots \sigma_n$ tenemos el teorema de bombeo. Esto termina la prueba. QED.

Ejemplo.- Como un ejemplo de la aplicación de este teorema, probaremos que el lenguaje $\{a^nb^n\}$ no es regular. En efecto, supongamos que fuera regular. Entonces, por el teorema de bombeo, debe haber palabras de la forma xyz, a partir de una cierta longitud, en que la parte y puede repetirse cuantas veces sea. Existen 3 posibilidades:

1. Que y no contenga mas que a, es decir, $y = aa \ldots a$. En este caso, al repetir varias veces y, habrá más a s que b s y la palabra no tendrá la forma deseada. Es decir, suponiendo que $\{a^nb^n\}$ es regular hemos llegado a la conclusión de que contiene palabras con más a s que b s, lo cual es una contradicción.

2. Que y no contenga mas que b s. Este caso es similar al caso (1).

3. Que y contenga a s y b s, es decir, $y = aa \ldots abb \ldots b$. Pero en este caso, al repetirse y, las a s y b s quedarán en desorden en la palabra, la cual no tendrá la forma $a^n b^n$. También en este caso hay contradicción.

Por lo tanto el suponer que $\{a^n b^n\}$ es regular nos lleva a contradicción. Se concluye que $\{a^n b^n\}$ no es regular.

Es muy importante notar que para las pruebas por contradicción usando el teorema de bombeo hay que explorar *exhaustivamente todas las posibles maneras* de dividir la palabra w en xyz, y encontrar contradicción en cada posible división. Con una sola división posible en que no se encuentre una contradicción, la prueba fracasa. Al fracasar la prueba, no se puede concluir ni que el lenguaje es regular ni que no lo es; simplemente no se llega a ninguna conclusión.

Otros lenguajes que tampoco son regulares son : $\{ww\}$, que es el lenguaje cuyas palabras tienen dos mitades iguales, $\{ww^R\}$, que es el lenguaje cuyas palabras tienen dos mitades simétricas (w^R es el reverso de w, es decir, $(abaa)^R = aaba$; el lenguaje de los palíndromes, que se leen igual al derecho y al revés, como ANITA LAVA LA TINA, [15] el lenguaje de los paréntesis bien balanceados, como ()(()), ()()(), ((())), etc.

4.5 Ejercicios

1. Para una palabra w, un *sufijo* de w es cualquier subcadena s con que termina w, es decir $zs = w$, $w, z, s \in \Sigma^*$. Si L es un lenguaje, Sufijo(L) es el conjunto de sufijos de las palabras de L. Demuestre que si R es un lenguaje regular, Sufijo(R) también es regular.

2. Pruebe que los siguientes lenguajes no son regulares:

   (a) $\{a^n b^m \mid |n - m| \leq 3\}$

   (b) $\{a, b\}^* - \{a^n b^n\}$

3. Encuentre una ER para el lenguaje sobre $\{a, b\}$ donde las palabras no contienen las cadenas "abb" ni "bab". Hacerlo por un método general, que no sea aplicable sólo a las cadenas dadas en este ejemplo. (Hint: obtener un autómata para el lenguaje complemento).

4. Pruebe si los lenguajes regulares son cerrados con respecto a las siguientes operaciones:

   (a) Diferencia: Si R1 y R2 son regulares, R2-R1 es también regular.

   (b) Reverso: si $\{w\}$ es regular, $\{w^R\}$ también es regular.

[15] ¡Atención! este lenguaje no es igual a $\{ww^R\}$

5. Probar que es regular el lenguaje de las palabras en $\{0,1\}$ que representan en binario un número mayor que 300.

6. Usando el teorema de bombeo pruebe que los lenguajes siguientes no son regulares

   (a) $\{a^n b^m \mid m > n\}$ (Hint: En algún momento se puede necesitar considerar las palabras de la forma $a^n b^n + 1$).

   (b) $\{a^n b^{n+m} c^m\}$.

7. Dado que los lenguajes regulares son cerrados respecto al complemento y a la unión, es muy sencillo probar que también son cerrados respecto a la intersección, pues $L1 \cap L2 = (L_1^c \cup L_2^c)^c$.

   (a) ¿Serán cerrados los lenguajes regulares respecto a la diferencia de conjuntos — esto es, si L1 y L2 son regulares, será necesariamente L1-L2 regular? Pruebe su respuesta.

   (b) ¿Es regular el lenguaje $\{a^n b^m \mid n \neq m\}$? Pruebe su respuesta.

8. Consideremos el problema de saber si el lenguaje aceptado por un AFD M es vacío o no lo es.

   (a) Una primera idea sería simplemente verificar si el conjunto de estados finales es vacío ¿Porqué no funciona esta idea ?

   (b) Proponga un procedimiento que sí permita decidir si $L(M) = \emptyset$ (Hint: Utilice la comparación de autómatas).

   (c) Aplique el procedimiento de (b) para verificar si el lenguaje del siguiente AFD M es vacío: $(\{1,2,3\}, \{a,b\}, \{((1,a),2), ((1,b),1), ((2,a),2), ((2,b),1), ((3,a),2), ((3,b),1)\}, 1, \{3\})$

9. Pruebe que los siguientes lenguajes son / no son regulares:

   (a) $A = \{w \in \{a,b\}^* \mid |w| \geq 7\}$

   (b) $\{a^n b^n\} \cap A$

10. Sean dos lenguajes, L_A y L_B tales que L_A es subconjunto de L_B.

    (a) Si L_A es regular, ¿también lo será necesariamente L_B? (Probar)

    (b) Si L_B es regular, ¿también lo será necesariamente L_A? (Probar)

11. Sean dos lenguajes no regulares, L_A y L_B.

    (a) Su unión podría eventualmente ser regular? (Hint: considere el problema 3).

    (b) Su intersección podría eventualmente ser regular? (Hint: considere intersecciones finitas).

12. Dado un lenguaje regular L, ¿es regular $\{w^R \mid w \in L\}$, donde w^R es el reverso de w? Pruebe su respuesta.

13. Pruebe que los siguientes lenguajes en $\{a, b\}^*$ son/no son regulares:
    (a) $\{w \neq a^n b^n\}$ (Hint: use las propiedades de los lenguajes regulares)
    (b) $\{a^n b^n \mid n \leq 7\}$
14. Supóngase un tipo de gramáticas que llamaremos "semiregulares", por asemejarse a las gramáticas regulares, en que las reglas son de alguna de las siguientes formas:
    (a) $A \rightarrow \sigma B$
    (b) $A \rightarrow B\sigma$
    (c) $A \rightarrow \sigma$

    donde σ es un terminal, y A y B son no terminales. ¿Serán equivalentes a las gramáticas regulares? Pruebe su respuesta.
15. Para una palabra w, un "prefijo" de w es cualquier subcadena a con que empieza w, es decir $ab = w$, para $a, b, w \in \Sigma^*$. Si L es un lenguaje, Prefijo(L) es el conjunto de prefijos de las palabras de L; obviamente $L \subseteq Prefijo(L)$. Demuestre que los lenguajes regulares son cerrados respecto a la operación de obtener prefijos. Para esto haga lo siguiente:
    (a) Defina constructivamente una operación $f - cerr(q_f)$, que obtiene el conjunto de estados desde donde se puede llegar a un estado dado q_f; a partir de esto defina otra operación $F - Cerr(F)$ que obtiene el conjunto de estados desde los que se puede llegar a alguno de los estados del conjunto F;
    (b) Modifique el AFD que acepta L, añadiendo un nuevo estado y algunas transiciones vacías, para que el autómata resultante acepte Prefijo(L).
16. Encuentre una Expresión Regular que represente las palabras en $\{a, b\}^*$ que no contienen ni la subcadena "aaa" ni "bab" y que son de longitud impar, por el método siguiente:
    (a) Encontrar un AF que acepte las palabras que contienen "aaa"
    (b) Encontrar un AF que acepte las palabras que contienen "bab"
    (c) Encontrar un AF que acepte las palabras de longitud par
    (d) Combinar los AF de (a),(b) y (c) en un AF que acepta las palabras que contienen "aaa" o "bab" o son de longitud par
    (e) Obtener un AF que acepte el complemento de lo que acepta (d)
    (f) Convertir el AF de (e) a ER

Capítulo 5

Redes de Petri

En este capítulo estudiamos una forma de autómatas, las *Redes de Petri*, que son de gran interés tanto teórico como práctico. En lo que respecta al aspecto práctico, las Redes de Petri forman parte de las técnicas estándar de modelación discreta en sistemas de control en manufactura.

Nuestro enfoque -ligeramente diferente al de la mayoría de textos en Redes de Petri- busca poner de manifiesto las semejanzas y diferencias con los autómatas finitos que se han estudiado anteriormente.

5.1 Modelación de sistemas discretos

La modelación de fenómenos y procesos es interesante porque permite:

- Verificar hipótesis sobre dichos procesos;
- Efectuar predicciones sobre el comportamiento futuro;
- Hacer experimentos del tipo "¿qué pasaría si...?", sin tener que actuar sobre el proceso o fenómeno físico.

Llamamos eventos discretos a aquéllos en los que se considera su estado sólo en ciertos momentos, separados por intervalos de tiempo, sin importar lo que ocurre en el sistema entre estos momentos. Es como si la evolución del sistema fuera descrita por una secuencia de fotografías, en vez de un flujo continuo, y se pasa bruscamente de una fotografía a otra.

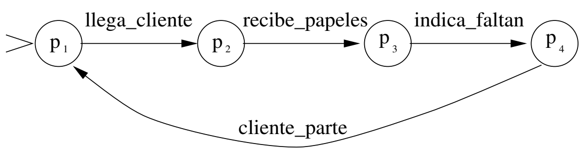
Usualmente se considera que la realidad es continua, y por lo tanto los sistemas discretos son sólamente una abstracción de ciertos sistemas, de los que nos interesa enfatizar su aspecto "discreto". Por ejemplo, en un motor de gasolina se dice que tiene cuatro tiempos: Admisión, Compresión, Ignición y Escape. Sin embargo, el pistón en realidad no se limita a pasar por cuatro posiciones, sino que pasa por todo un rango de posiciones continuas. Así, los "cuatro tiempos" son una abstracción de la realidad.

Una posibilidad para modelar secuencias de acciones / sucesos es el uso de los Autómatas Finitos (AF), como hemos visto en las secciones precedentes. Sin embargo, hay ciertos tipos de procesos para los cuales el modelado por medio de AF, si bien es posible en principio, no es práctico, pues los modelos así generados son demasiado complejos, y por lo tanto difíciles de manejar intuitivamente. Es por esto que se proponen las Redes de Petri como una alternativa a los AF para el modelado de una gran cantidad de sistemas discretos.

Vamos a ilustrar esta situación por medio de un ejemplo. El proceso que se quiere modelar es una oficina del gobierno, donde hay una ventanilla con un burócrata que atiende al público. Los pasos que se siguen en la atención a un contribuyente son siempre los mismos y se repiten indefinidamente, siguiendo la secuencia :

- Llega cliente a ventanilla
- Burócrata recibe papeles e indica que falta un certificado
- Cliente insulta a burócrata y parte
- Llega cliente, etc.

Dicha secuencia puede modelarse por el siguiente AF, en que las acciones que etiquetan las flechas son símbolos de un alfabeto $\Sigma = \{Llega\_cliente, Recibe\_papeles, \ldots\}$:



Como se vé, la modelación de un burócrata con AF no causa mayores problemas. Sin embargo, considérese ahora que la oficina tiene dos burócratas "trabajando" simultáneamente. En este caso, el modelado de la oficina requeriría el uso de estados que consideren todas las posibles combinaciones de los estados de cada burócrata, es decir, que mientras uno está siendo insultado por el cliente (p_4) el otro puede haber recibido los papeles (p_3), y así habría un estado (p_4, p_3), etc. Formalizando, se tendría un conjunto de estados $K' = K \times K$, donde K es $\{p_1, p_2, p_3, p_4\}$, es decir, se tienen 16 estados, y un AF excesivamente complejo aún para este ejemplo tan sencillo.[1]

[1] Realmente los estados de este AF no necesitan ser pares (q, q'), sino que tendría que haber un estado en K' asociado a cada par ordenado en $K \times K$.

La moraleja de este ejemplo es que:

Los AF no son adecuados para modelizar situaciones en que ocurren varios procesos concurrentemente.

Esta situación motiva que se busquen alternativas más adecuadas para el modelado de procesos concurrentes. Aquí entran en escena las Redes de Petri.

5.2 Introducción a Redes de Petri

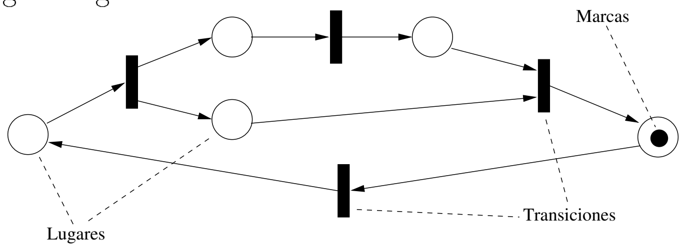
El desarrollo de las Redes de Petri (RP) comienza con la publicación en 1962 de la tesis doctoral "Kommunikation mit Automaten" de Carl A. Petri. Ahí se expone una teoría no enteramente formalizada, que tiene propósitos a la vez teóricos y prácticos. A partir de 1970, J.B. Dennis y su grupo, "Computation Structure Group" (MIT), se interesan en el trabajo de Petri, como medio para modelar algoritmos paralelos y estructuras concurrentes en hardware y software. El grupo de Dennis introduce refinamientos en la teoría básica de Petri. A partir de entonces, la teoría de las RP ha conocido cierto auge, y diversos autores han propuesto muy diversas variantes (RP coloreadas, RP temporizadas, etc.). Asimismo, las aplicaciones han invadido varios dominios técnicos, en particular los sistemas de manufactura. De estos datos históricos concluímos lo siguiente:

1. La definición de las RP se ha extendido con el tiempo
2. Actualmente hay varias definiciones alternativas
3. La teoría se ha desarrollado junto con la práctica

En vista de que la teoría de las RP es relativamente cambiante, nosotros vamos a adoptar un punto de vista afín a las técnicas de representación y análisis de autómatas que se han seguido en este curso. Aunque los conceptos esenciales coinciden con los expuestos en [Peterson 81], las definiciones son en muchos casos diferentes, para ajustarse a nuestras metas. Para dejar clara esta diferencia, vamos a llamar ÒRP booleanasÓ la variante de las RP que vamos a estudiar en este curso.

5.3 Redes de Petri Booleanas

Para representar las RP se utiliza mucho una representación gráfica, en la que se aprecian los elementos de la figura siguiente:



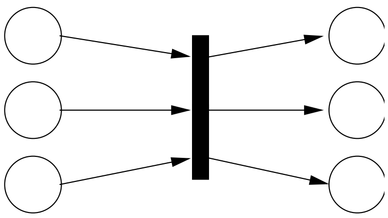
Se trata esencialmente de grafos dirigidos, en los que se aprecian dos tipos de nodos:

- Los *lugares*, representados gráficamente por círculos
- Las *transiciones*, representadas gráficamente por barras.

En los lugares puede haber o no una marca, representada por un punto "•". Intuitivamente, las marcas "circulan" de un lugar al otro, pasando "a través" de las transiciones, de acuerdo con la dirección de las flechas.

Un rasgo distintivo de las RP booleanas es que en ellas sólo se puede tener *una* marca o ninguna en cada uno de los lugares. En otras concepciones de las RP, los lugares pueden tener un número cualquiera de marcas. De este modo, en las RPB la presencia de una marca puede identificarse como una condición lógica; es decir, si hay una marca, el atributo *marca* del lugar en cuestión es cierto, y la ausencia de marca indica que dicho atributo es falso.

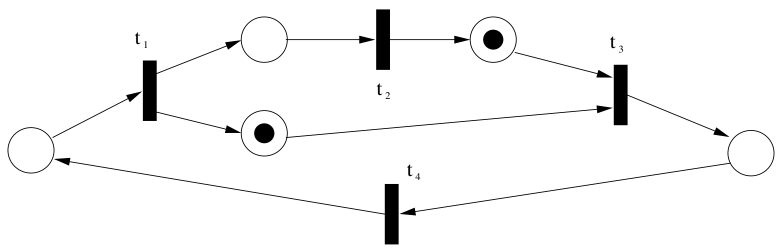
La relación entre los lugares y las transiciones está indicada en la figura por las flechas, que van de una transición a un lugar o viceversa (nunca hay flechas de una transición a otra o de un lugar a otro). Se dice que los lugares de entrada de una transición son aquellos lugares tales que hay una flecha de ellos a la transición en cuestión. Similarmente, un lugar es lugar de salida de una transición si existe una flecha de la transición a ese lugar. Estas relaciones están esquematizadas en la siguiente figura:



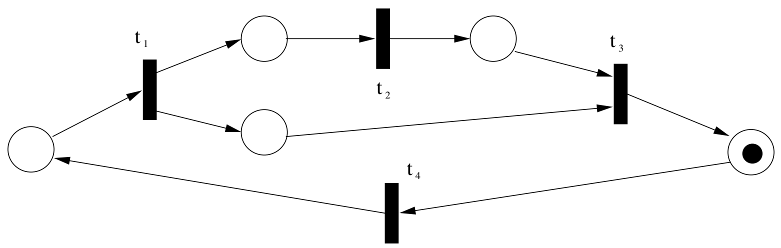
Lugares de entrada *Lugares de salida*

La noción central respecto al funcionamiento o "ejecución" de las RP es que una transición puede "disparar". Se dice que una transición está *lista para disparar* cuando todos sus lugares de entrada tienen marca. El disparo de una transición consiste en que se quitan las marcas de todos los lugares de entrada, y se pone una marca en cada lugar de salida, si no la había anteriormente.

Por ejemplo, supóngase la siguiente RP:



Como se puede observar, sólamente la transición t_3 está lista para disparar. El disparo de t_3 va a hacer que las marcas de sus lugares de entrada desaparezcan, y que se creen marcas en el único lugar de salida que tiene:



Por lo expuesto, debe quedar claro que si un lugar de salida tiene marca antes del disparo de la transición, dicho lugar simplemente va a quedar con la marca que tenía.

5.3.1 Formalización de las RPB

Una vez expuestos en forma intuitiva los elementos de las RPB, procedemos a su formalización. Una RPB está definida como un cuádruplo de la forma:

$RP = (P, T, I, O)$, donde:

P: Lugares

T: Transiciones

I,O: $T \rightarrow 2^P$

Como se vé, las funciones I (lugares de entrada) y O (lugares de salida), tienen como argumento una transición, siendo el resultado un conjunto de lugares.

Por ejemplo, sea la RPB de la tabla 5.1

Tabla 5.1: Ejemplo de RP

$RP_1 = (P, T, I, O)$	
$P = \{p_1, p_2, p_3, p_4, p_5\}$	
$T = \{t_1, t_2, t_3, t_4\}$	
$I(t_1) = \{p_1\}$	$O(t_1) = \{p_2, p_3, p_5\}$
$I(t_2) = \{p_2, p_3, p_5\}$	$O(t_2) = \{p_5\}$
$I(t_3) = \{p_3\}$	$O(t_3) = \{p_4\}$
$I(t_4) = \{p_4\}$	$O(t_4) = \{p_2, p_3\}$

Ejercicio.- Dibujar esta RP.

5.3.2 Inverso, Dual de una RP

Se puede apreciar que lo grafos representando las RP son simplemente grafos con dos tipos de nodos, P y T, en que los arcos van de P a T o de T a P. Esta simetría entre los lugares y las transiciones sugirió la noción de *RP inversa*, que se obtiene intercambiando las funciones I y O -esto equivale a cambiar de sentido la dirección de las flechas.

Similarmente, el *dual* de una RP se obtiene intercambiando los lugares por las transiciones.

Ejercicio.- Formalizar las nociones de dual e inverso de una RP.

Ejercicio.- Obtener los cuádruplos correspondientes al dual y al inverso de la RP descrita arriba (RP_1).

5.3.3 Marcaje

La presencia / ausencia de marcas (marcaje, μ) se formaliza como una función con resultado booleano:

$$\mu : P \rightarrow \{V, F\}$$

donde V es "verdadero" y F es "falso". μ puede también ser visto como un conjunto, que agrupa los lugares que sí tienen marca. Por ejemplo, considérese la RP de la tabla (5.1). Si los lugares que tienen marca son p_2, p_4 y p_5, el marcaje μ se puede representar como función de la manera siguiente:

$$\left| \begin{array}{ll} \mu(p_1) = F & \mu(p_2) = V \\ \mu(p_3) = F & \mu(p_4) = V \\ \mu(p_5) = V & \end{array} \right|$$

En cambio, representado como conjunto sería: $\mu = \{p_2, p_4, p_5\}$.

Para una RPB (P, T, I, O) existen exactamente $2^{|P|}$ marcajes diferentes. El hecho de que el conjunto de los marcajes posibles sea finito, permite resolver en las RPB ciertos problemas que no se pueden resolver en otras formalizaciones de las RP, como se verá más adelante.

Un par (R, μ), donde R es una RPB y μ es un marcaje, se llama *Red de Petri marcada*, y corresponde a los diagramas de RP con marcas que hemos visto anteriormente. En muchos casos nos interesa en particular el marcaje existente al empezar el funcionamiento de una RPB dada, el cual se llama *marcaje inicial* (μ_0).

5.3.4 Ejecución

La formalización que haremos a continuación de la ejecución de una RPB es bastante diferente a la forma en que se aborda el problema usualmente en los textos de Redes de Petri. Nuestra formalización se apega más bien al formato que hemos seguido para formalizar el funcionamiento de otros tipos de autómatas -por ejemplo, los autómatas finitos. Así, las nociones básicas son las de configuración y relación entre configuraciones.

La idea básica es identificar el marcaje de la RP a la configuración de un autómata. Es decir, el marcaje resume la situación global del autómata, y no se consideran entradas externas, por lo que no se necesita incorporar nociones como "lo que falta por leer de la palabra". Así, una configuración es un conjunto de lugares μ.

En seguida hay que definir cuándo y cómo es posible pasar de una configuración (marcaje) a otra, definiendo así una relación binaria $\vdash$ entre configuraciones. Dicha relación se puede definir de la siguiente forma:

Definición.- $\mu_1 \vdash_t \mu_2$ ssi $t \in T$ tal que:

1. si $p \in I(t)$, entonces $\mu_1(p)$
2. si $p \in I(t)$ y $p \notin O(t)$, entonces $\neg\mu_2(p)$
3. si $q \in O(t)$ entonces $\mu_2(q)$
4. si $p \notin (I(t) \cup O(t))$ entonces $\mu_1(p) = \mu_2(p)$.

La condición (1) quiere decir que para que se pueda pasar de μ_1 a μ_2 debe haber una transición t tal que todos sus lugares de entrada tienen marcas en μ_1. La condición (2) quiere decir que las marcas de los lugares de entrada se pierden en μ_2, siempre y cuando dicho lugar de entrada no sea al mismo tiempo lugar de salida. La condición (3) indica que los lugares de salida de las transiciones que disparan tienen marca en μ_2. La condición (4) dice simplemente que los nodos que no tienen conexión con la transición t, conservan el mismo marcaje que tenían antes de disparar t.

Otra formalización posible para el disparo de las transiciones es la siguiente:

Definición.- Si μ_2 es el resultado de disparar t a partir de μ_1, tenemos que: [2]

$$\mu_2 = (\mu_1 - I(t)) \cup O(t)$$

Ciertamente esta definición es más simple que la anterior, además de que brinda una fórmula para calcular efectivamente el marcaje resultante del disparo de una transición. [3]

Como de costumbre, un cálculo es una secuencia de configuraciones ligadas por la relación "$\vdash$", es decir: $\mu_1 \vdash_{t_1} \mu_2 \vdash_{t_2} \ldots \vdash_{t_{n-1}} \mu_n$.

Dado un cálculo $\mu_1 \vdash_{t_1} \mu_2 \vdash_{t_2} \ldots \vdash_{t_{n-1}} \mu_n$, la lista de transiciones que dispararon t_1, t_2, $\ldots$, t_{n-1} es llamada *traza de una ejecución*, o *secuencia de disparo*. El interés de una secuencia de disparo es que caracteriza completamente el funcionamiento de una RP. En efecto, dada una RP, un marcaje inicial, y una secuencia t_1, t_2, $\ldots$, t_n, es posible reproducir exactamente todos los marcajes intermedios por los que pasó la RP. [4]

5.4 Modelado con Redes de Petri

La motivación inicial por la que estudiamos las RP es la de superar las limitaciones de los AF para el modelado de los eventos discretos concurrentes. En esta sección veremos una metodología para aplicar las RP a dicho modelado.

Al construir un modelo de un objeto o fenómeno real, es necesario establecer los vínculos entre el objeto a modelar y los elementos del modelo. En el caso de las RP, que constan de eventos y transiciones ligados, se establecen las siguientes asociaciones:

LUGARES	$\rightarrow$	CONDICIONES (Se verifican o no)
TRANSICIONES	$\rightarrow$	EVENTOS(Ocurren)

[2] *Ejercicio*.- ¿Es equivalente la definición: $\mu_2 = (\mu_1 \cup O(t)) - I(t)$?

[3] *Ejercicio*.- Elaborar un programa computacional que calcule el marcaje resultante de disparar alguna de las transiciones listas para disparar, escogida aleatoriamente.

[4] Esto es una forma de determinismo en las RP. Hay variantes de las RP en que se ha buscado eliminar el determinismo.

Es decir, las transiciones se asocian a las acciones o eventos que ocurren en el sistema a modelar, mientras que los lugares se asocian a las condiciones que resultan de las acciones.

Por ejemplo, para el caso de los burócratas, podemos tener las siguientes acciones, asociadas a transiciones:

EVENTOS:

t_1 - *Llega cliente a ventanilla*

t_2 - *Burócrata recibe papeles*

t_3 - *Burócrata indica que falta un certificado*

t_4 - *Cliente insulta a burócrata y parte*

Similarmente, tenemos las siguientes condiciones:

CONDICIONES:

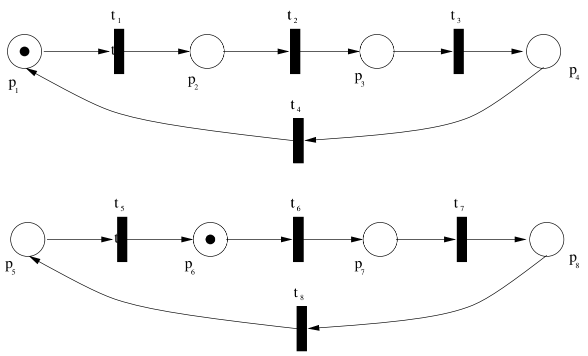
p_1 - *La ventanilla está libre*

p_2 - *Hay alguien en la ventanilla*

p_3 - *Burócrata examina papeles*

p_4 - *Cliente enojado*

Dichos eventos y condiciones aparecen en la siguiente RP con dos burócratas ÒtrabajandoÓ simultáneamente:



En esta figura, mientras la ventanilla del primer burócrata está libre, en la ventanilla del segundo ya se encuentra un cliente en espera (infructuosa) de ser atendido.

Desde luego, la manera de repartir los aspectos de un proceso entre condiciones y eventos depende enteramente de la forma en que la persona mire el proceso, y no hay realmente reglas generales (ni mucho menos formalizaciones) para ello. En ocasiones es utilizado un elemento de metodología para organizar las acciones y las condiciones; se trata de las *precondiciones* y las *postcondiciones*.

Una precondición de una acción es una condición necesaria para que ésta pueda ser ejecutada. Similarmente, una postcondición de una acción es una condición que describe el resultado de ejecutar la acción. Organizando las precondiciones y postcondiciones de las acciones, es posible visualizar más fácilmente las relaciones entre condiciones y acciones, y establecer las ligas que se traducirán en forma directa a una RP. Este proceso es ilustrado en el siguiente ejemplo.

Ejemplo 2.- En un taller de maquinado, llegan órdenes de fabricación de piezas por parte de los clientes, y dichas órdenes se trabajan en cuanto esto es posible, es decir, cuando existen las máquinas y los operarios necesarios. Una vez maquinadas las piezas, éstas se turnan al departamento de entregas.

Se tienen las siguientes condiciones:

a) El taller está ocioso

b) Llegó una orden y está en espera

c) El taller trabaja en la orden

d) La orden está terminada

Los eventos o acciones son las siguientes:

1.- Llega una orden

2.- El taller comienza a trabajar la orden

3.- La orden es terminada

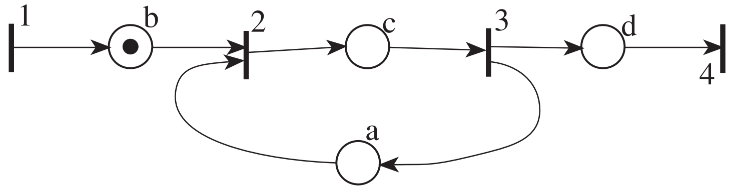
4.- La orden es mandada a entregas

Se forma un arreglo describiendo las precondiciones y postcondiciones de las acciones:

Evento	Precondición	Postcondición
1	-	b
2	a,b	c
3	c	d,a
4	d	-

De acuerdo con esta tabla, por ejemplo, la acción de comenzar a trabajar una orden (acción 2) sólo se puede efectuar si el taller está ocioso (condición a), y llegó una orden (condición b); como resultado de la acción de comenzar a trabajar una orden, se cumple la condición c -el taller trabaja en la orden.

A partir de la matriz de precondiciones y postcondiciones es posible trazar el diagrama de la RP, de forma tal que las precondiciones son los lugares de entrada de cada transición, y las postcondiciones son sus lugares de salida:



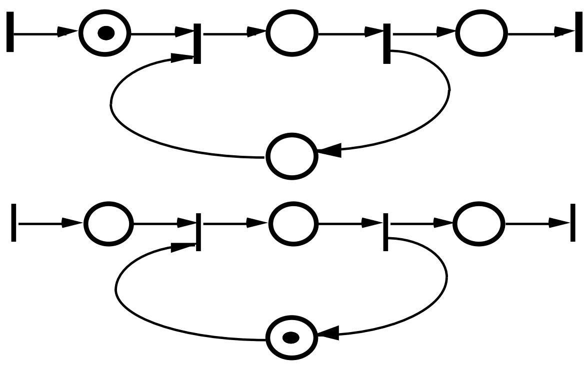
5.4.1 Entradas y salidas

En el ejemplo precedente se ilustra una manera de tomar en cuenta las entradas y salidas de la RP con el exterior: El disparo de la transición 1 se asocia a la llegada de una orden, lo cual ocasiona que se cree una marca en el lugar de salida de 1 (b). Similarmente, la salida de órdenes de trabajo terminadas se asocia con el disparo de la transición 4, que hace desaparecer la marca de su lugar de entrada (d), haciendo que la condición Òla orden está terminadaÓ deje de ser cierta.

Esta no es la única manera de considerar los entradas y salidas, y algunos autores prefieren considerar que hay lugares en los que “aparece” una marca cada vez que se produce una entrada, y otros lugares en que ÒdesapareceÓ una marca cada vez que se produce una salida.

5.4.2 Modelado de procesos concurrentes

Si en el ejemplo precedente se tienen dos talleres iguales en vez de uno, pero que pueden encontrarse en estados diferentes, este taller ÒdobleÓ puede ser modelado fácilmente por una RP que simplemente duplica la RP anterior:



Puede observarse que ambos talleres evolucionan uno independientemente del otro. Sin embargo, con RP pueden modelarse también procesos en que existen dependencias e interacciones. Esto es ilustrado con el siguiente ejemplo.

Ejemplo 3.- Se tiene un taller con 3 máquinas, representadas por M1, M2 y M3. Hay dos operadores, F1 y F2, donde F1 puede manejar las máquinas M1 y M2, mientras que F2 puede manejar M1 y M3. El maquinado de las piezas se hace en dos etapas: la primera, el maquinado "grueso", es efectuada utilizando la máquina M1, y posteriormente el maquinado "fino" es efectuado ya sea en M2 o en M3.

Se proponen las siguientes condiciones:

a) Orden espera ser maquinada por M1

b) Salida M1 espera maquinado por M2 y M3

c) La orden está completa

d) La máquina M1 está ociosa

e) La máquina M2 está ociosa

f) La máquina M3 está ociosa

g) El operador F1 está ocioso

h) El operador F2 está ocioso

i) La máquina M1 es operada por F1

j) La máquina M1 es operada por F2

k) La máquina M2 es operada por F1

l) La máquina M2 es operada por F2

Se tienen las siguientes acciones:

1.- Llega una orden

2.- Operador F1 empieza maquinado en M1

3.- Operador F1 termina maquinado en M1

4.- Operador F2 empieza maquinado en M1

5.- Operador F2 termina maquinado en M1

6.- Operador F1 empieza maquinado en M2

7.- Operador F1 termina maquinado en M2

8.- Operador F2 empieza maquinado en M3

9.- Operador F2 termina maquinado en M3

10.- La orden es mandada a entregas

La matriz de pre-post condiciones sería como sigue:

ACCION	PRECOND	POSTCOND
1	-	a
2	a,g,d	i
3	i	g,d,b
4	a,h,d	j
5	j	b,h,d
6	b,g,e	k
7	k	c,g,e
8	b,f,h	l
9	l	c,f,h
10	c	-

Finalmente, el diagrama de la RP quedaría como se aprecia en la figura 5.1.

En la figura 5.1 pueden apreciarse una parte superior y una inferior de la RP, que son

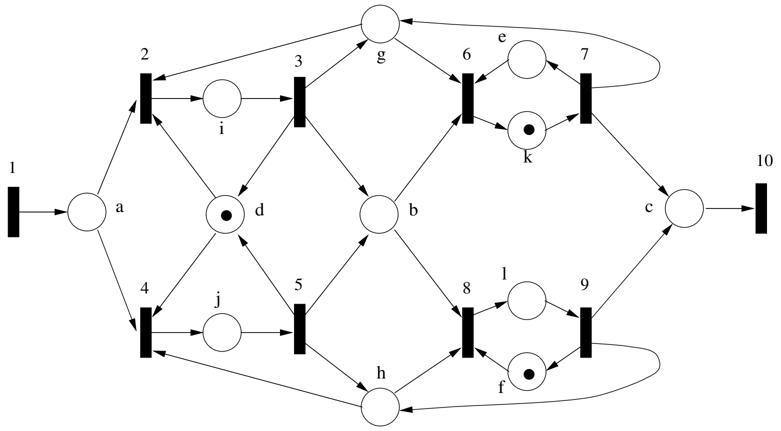


Figura 5.1: Red del ejemplo de maquinado

simétricas y que interactúan a través de los lugares que se encuentran enmedio (a, d, b y c). Por ejemplo, el lugar a representa una orden que llega, y que puede ser tomada por el operador F1 o por F2 -quien lo haga primero-, pero una vez que disparó la transición 2 o la 4, la marca en a desaparece, y el otro operador ya no puede recibir una orden, hasta que llegue una nueva. Similarmente, el lugar d sirve para arbitrar el uso de la máquina M1 entre F1 y F2.

5.4.3 Propiedades de modelado de las RP

En los ejemplos precedentes pueden observarse las propiedades esenciales de modelado de las RP, que son las siguientes:

- No-determinismo
- No-simultaneidad
- Paralelismo o concurrencia
- Asincronía
- Sincronización

El *no-determinismo* significa que en un momento dado puede disparar cualquiera de las transiciones "listas", es decir, aquellas cuyos lugares de entrada tienen todos marca. Así, para hacer que una RP funcione de manera determinista, es necesario garantizar que en todo momento sólamente una de sus transiciones estará lista para disparar.

La *no-simultaneidad* se refiere al hecho de que sólamente puede disparar una transición a la vez. Esta característica puede ser problemática para modelar situaciones en que hay eventos que pueden o no ser simultáneos; en RP se supondría que uno de ellos ocurre "ligeramente antes" que el otro.

El *paralelismo* o *concurrencia* es una de las características más importantes de las RP, y se refiere al hecho de que la ejecución de dos o más partes de una RP puede darse en forma "traslapada" o "concurrente". [5]

La *asincronía* consiste en que el momento exacto en que ocurre el disparo de una transición no depende de ningún evento; de hecho la duración de los eventos no forma parte del modelado en RP. Simplemente se supone que existe un orden parcial entre los eventos, que viene dado por la relación antes-después; esta información se encuentra en la tabla de precondiciones y postcondiciones.

La *sincronización* en RP no consiste en que dos eventos ocurran a la vez (cosa que es imposible por la no-simultaneidad), sino que es posible hacer depender el inicio de un evento de la terminación de otro. Así, en el ejemplo del taller con 3 máquinas, el nodo b fuerza a que, para iniciar la actividad 6 o la 8, sea necesario antes haber terminado la actividad 3 o la 5.

5.4.4 Problemas de sincronización en RP

Uno de los temas de mayor interés en el estudio de aplicaciones de RP es el de cómo garantizar una correcta interacción entre varios procesos. Del estudio de este tema se han identificado algunas estructuras de sincronización de aplicación general. De éstas, vamos a estudiar en particular estructuras para dos tipos de problemas: 1) Estructuras de programación paralela, y 2) Asignación de recursos.

Estructuras de programación paralela

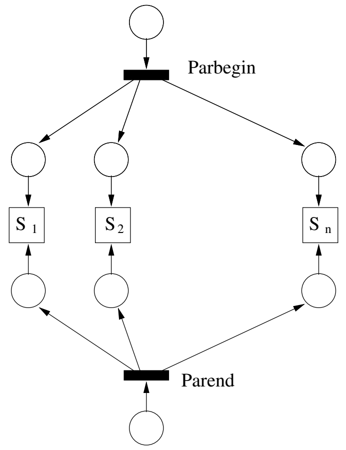
El "Computation Structure Group" (MIT) de Dennis estudió la aplicación de las RP a la estructura de algoritmos paralelos, tanto en el aspecto hardware como software. En lo que respecta al software, se propusieron mecanismos para introducir la posibilidad de ejecución en paralelo, cosa que no existe en los lenguajes imperativos tradicionales. Entre las construcciones propuestas encontramos el *parbegin*, complementado con el *parend*.

Las palabras claves `parbegin` y `parend` -que se añaden a un lenguaje imperativo cualquiera- definen un bloque de instrucciones que se pueden ejecutar en paralelo, en vez de hacerse en secuencia. El usuario, al encapsular un conjunto de instrucciones entre un `parbegin` y un

[5] Aunque debe quedar claro por el punto anterior (no-simultaneidad) que las transiciones de las distintas partes de la RP no pueden ser disparadas exactamente a la vez.

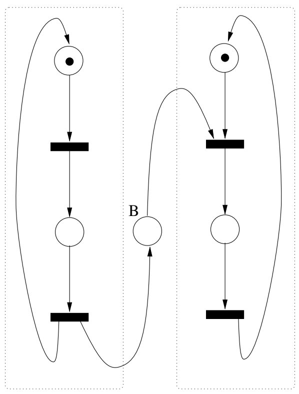
`parend`, está implícitamente declarando que el orden en que se ejecuten dichas instrucciones no es relevante.[6]

La estructura `parbegin` / `parend` se puede realizar en RP mediante una estructura como la de la figura siguiente:



Al dispararse la transición asociada al `parbegin`, se introduce una marca en cada lugar donde comienza un bloque S_i. Similarmente, para que pueda disparar la transición `parend`, es necesario que ya se hayan ejecutado todos los bloques S_i, pues de otra forma dicha transición no estará lista para disparar.

Otro ejemplo interesante de sincronización de procesos es el del esquema *productor / consumidor*. En dicho esquema se tienen dos procesos, el productor y el consumidor, que se comunican por medio de un "buffer". El proceso productor envía datos al buffer cuando éste se encuentra vacío, y se detiene cuando se llena. Similarmente, el consumidor toma datos del buffer cuando los hay, y permanece en espera mientras el buffer está vacío. La estructura productor / consumidor puede ser realizada en RP con una estructura como la siguiente:

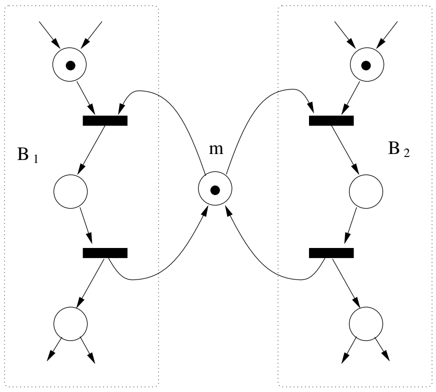


El análisis de esta RP se deja como ejercicio al lector.

[6] Desde luego, siendo ésta una estructura de software, las instrucciones de un bloque no se ejecutan relamente "a la vez" a menos que haya un procesador para cada una; el paralelismo se dá en el sentido de que hay una tarea independiente (task) por cada instrucción.

Asignación de recursos

En el ámbito de los sistemas operativos, un problema de asignación de recursos consiste en la elección de un proceso que hará uso del recurso en cuestión en forma transitoria. Es en este sentido que el problema de la asignación de recursos fué estudiado por el grupo de Dennis. El siguiente ejemplo muestra cómo garantizar que un recurso -el uso del CPU- será asignado en forma exclusiva a un proceso mientras dure la ejecución de una sección crítica:

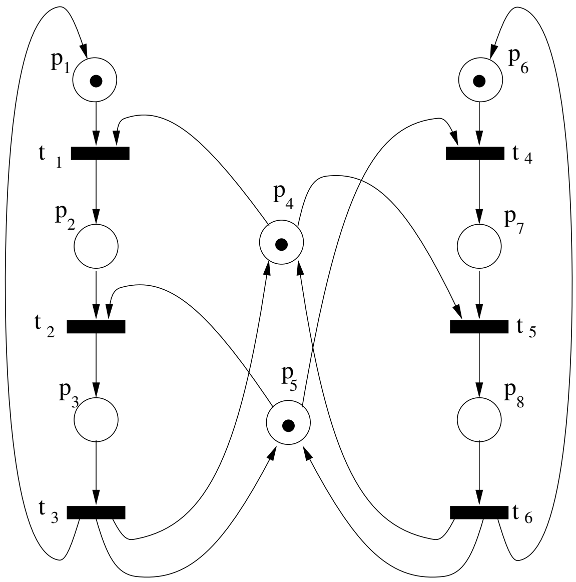


El proceso p_1, para poder disparar su primera transición, requiere tomar la marca del lugar m -quitándole en consecuencia a p_2 la posibilidad de ejecutarse. Al disparar la última transición de p_1, la marca es devuelta a m, con lo que se libera el recurso, el cual ya puede ser tomado por p_2 (o de nuevo por p_1!).

Deadlocks

Una situación clásica que aparece en el problema de asignación de recursos es la del "abrazo mortal" (deadlock). Esta situación consiste en que un proceso a tiene en su poder un recurso r, que requiere un proceso b para continuar su ejecución, mientras que b dispone de un recurso s que requiere a para ejecutar; el resultado es que ninguno de los dos procesos puede ejecutar. Es evidente que este tipo de situaciones son altamente indeseables en un sistema operativo, por lo cual se han llevado a cabo múltiples estudios para la prevención de deadlocks. En lo que respecta a las RP, lo primero que hay que asegurar es que el fenómeno de los deadlocks sea posible de modelizar; sólo así el estudio y solución a dicho problema podrá ser llevado a cabo dentro del cuadro de las RP.

En la figura siguiente se presenta un proceso, en la parte izquierda de la figura, constituido por las transiciones t_1, t_2 y t_3, y otro proceso, en la parte derecha, con transiciones t_4, t_5 y t_6. Los lugares p_4 y p_5 representan dos recursos que requieren tanto el proceso izquierdo como el derecho. De acuerdo con el marcaje de la figura, si dispara t_1, se quita la marca de p_4; si enseguida dispara t_4, desaparece la marca de p_5. En este momento el sistema entra en abrazo mortal, pues t_5 no podrá disparar sin la marca de p_4 (tomada por el proceso izquierdo), y t_2 tampoco podrá disparar, a falta de la marca de p_5 (tomada por el proceso derecho).



En vista del ejemplo anterior, se comprende la necesidad de poder garantizar que una RP - representando un esquema de asignación de recursos- no caerá en deadlock. Este problema exige un análisis detallado de la RP en cuestión, análisis para el cual se requieren ciertas herramientas conceptuales que serán estudiadas en la sección siguiente.

5.5 Análisis de Redes de Petri

El objetivo de analizar una RP es el de determinar las características del sistema que modelan. Así, nos interesa un análisis de tipo funcional, no morfológico.

5.5.1 Problemas de análisis de RP

Vamos a considerar cuatro problemas de análisis de RP, de los cuales 3 son clásicos de las RP en general, y el último es interesante para los propósitos de este curso. Estudiaremos así los problemas siguientes:

1. Viveza de una RP
2. Alcanzabilidad de un marcaje
3. Posibilidad de una secuencia de disparo
4. Determinismo de una RP

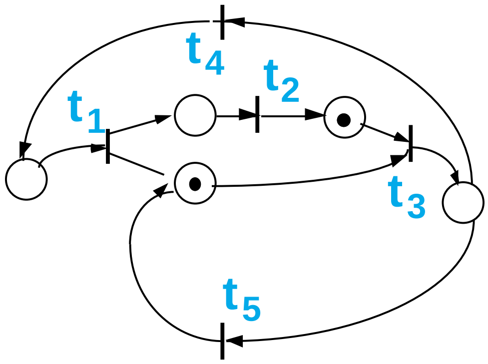


Figura 5.2: ¿Está viva?

Viveza de una RP

Una RP está *viva* mientras existe una transición que puede ser disparada. Si ninguna transición puede ser disparada, la RP está muerta o en deadlock. Dada una RP marcada (P, μ) es trivial verificar si está viva o no.

Una transición t está viva si existe una secuencia de disparo de la que forma parte. Evidentemente una RP está viva si tiene al menos una transición viva, e inversamente, una RP muerta no tiene ninguna transición viva.

Vamos a decir que una transición tiene *vida ilimitada* cuando hay cálculos que la hagan disparar en un futuro arbitrariamente lejano. Esta noción se formaliza con la manera siguiente:

Definición.- Dada una RP marcada $((P, T, I, O), \mu_0)$, una transición $t \in T$ tiene vida ilimitada si dado cualquier $i \in Nat$, hay alguna secuencia $\mu_0 \vdash \mu_1 \vdash \ldots \vdash \mu_n$, $n \geq i$ donde t puede disparar en μ_n.

En otras palabras, es posible encontrar un cálculo de cualquier longitud arbitraria en que la transición esté viva. Una RP tiene vida ilimitada si alguna de sus transiciones la tiene.

Por ejemplo, considérese la RP de la figura 5.2. En el momento descrito, la RP está viva, pues t_3 puede disparar, marcando su lugar de salida; como éste es el lugar de entrada de t_5, ésta podrá disparar a continuación. Sin embargo, después del disparo de t_5 ninguna transición podrá disparar, y la RP entrará en deadlock. Otra posibilidad es disparar t_3, t_4, t_1 y t_2, con lo que se regresa a la situación inicial. En conclusión, la RP mostrada tiene vida ilimitada. Es interesante el hecho de que una RP con vida ilimitada puede sin embargo morir; la vida ilimitada significa simplemente que, disparando las transiciones en forma adecuada, es posible mantener viva a la RP.

Decimos que una RP es inmortal si nunca puede caer en deadlock (la formalización de esta noción se deja como ejercicio). Obsérvese que toda RP inmortal tiene vida ilimitada, pero no viceversa. Si una RP es inmortal, no importa en qué orden se disparen las transiciones, nunca será posible hacerla caer en deadlock. Esta es una característica altamente deseable para algunos sistemas, como por ejemplo los sistemas operativos de las computadoras.

El (los) problema(s) de la viveza consiste(n) en determinar, dada una RP marcada, si está viva, si tiene vida ilimitada o bien si es inmortal, y similarmente para sus transiciones.

Alcanzabilidad

El problema de la alcanzabilidad consiste en saber si es o no posible llegar a un cierto marcaje desde un marcaje inicial. Formalmente:

Definición.- μ_n es *alcanzable* desde (P, μ) ssi $\mu \vdash^* \mu_n$.

El conjunto de marcajes alcanzables desde (P, μ) es $R(P, \mu)$. Claramente, el tamaño de $R((Q, T, I, O), \mu)$ tiene como límite superior $2^{|Q|}$.

El problema de la alcanzabilidad es muy importante porque varios de los problemas de decisión en RP pueden expresarse en términos de aquél.

Por ejemplo, el problema de la viveza de una transición puede transformarse en un problema de alcanzabilidad de un marcaje, de la manera siguiente: Sea una RP marcada $((P, T, I, O), \mu_0)$. La idea es que, para que una transición $t \in T$ dispare, se requiere que podamos alcanzar un marcaje μ que contenga marcas en todos los lugares de entrada de t: [7]

$$\mu_0 \vdash^* \mu, \; I(t) \subseteq \mu$$

La única dificultad que queda por resolver es que esto deja sin determinar el marcaje que deben tener los lugares $P - I(t)$, es decir, aquéllos que no son lugares de entrada de t. Una solución a este problema consiste en utilizar la relación de *subconjuntos* entre los marcajes, considerados éstos como conjuntos (de lugares marcados). Así tenemos la siguiente:

Definición.- Sea una red (P, T, I, O); $t \in T$ está viva en μ ssi existe $\mu' \in R((P, T, I, O), \mu)$ tal que $I(t) \subseteq \mu'$.

Similarmente, el problema de la vida ilimitada de una RP puede reducirse a la búsqueda de ciclos en la evolución de los marcajes, es decir, marcajes $\mu \in R(P, \mu_0)$ tales que $\mu \vdash \mu' \vdash^* \mu$, $\mu \neq \mu'$ [8]

[7] La implicación en la fórmula siguiente quiere decir que todos los lugares de entrada de t tienen marca.
[8] Pregunta: ¿Porqué no escribimos simplemente $\mu \vdash^* \mu$?.

Secuencia de disparo

El problema de la secuencia de disparo se puede expresar en la forma siguiente: Dada una RP marcada $((P,T,I,O),\mu)$, ¿es posible en algún momento (posiblemente futuro) disparar las transiciones t_i, t_j, ..., $t_n \in T$ en ese orden? En otras palabras, ¿existe un marcaje $\mu_1 \in R(P,T,I,O,\mu)$, tal que $\mu_1 \vdash_{t_i} \mu_i \vdash_{t_j} \mu_j \vdash_{t_k} \ldots \vdash_{t_n} \mu_n$?

Por ejemplo, la secuencia de disparo $< t_2, t_4, t_5 >$ en la RP de la figura 5.2 es imposible.

Se define un conjunto $S(R,\mu)$ formado por las secuencias de disparo posibles en (R,μ). Dicho conjunto puede ser infinito.

El problema de la secuencia de disparo es reducible al de la alcanzabilidad de un marcaje, de la manera siguiente: $< t_i,\ t_j,\ \ldots,\ t_n > \in S((P,T,I,O),\mu)$ ssi hay un marcaje $\mu_1 \in R(P,T,I,O,\mu)$ tal que a partir de μ_1 puede producirse la secuencia de disparo: $\mu_1 \vdash_{t_i} \mu_i \vdash_{t_j} \mu_j \vdash_{t_k} \ldots \vdash_{t_n} \mu_n$.

Determinismo

El *determinismo* en una RP consiste simplemente en que en todo momento sólo haya una transición que pueda disparar. El hecho de que sólo una transición puede disparar en el marcaje μ, se puede expresar formalmente de la manera siguiente, para una RP $((P,T,I,O),\mu)$, con $t_1, t_2 \in T$:

$$[I(t_1) \subseteq \mu, I(t_2) \subseteq \mu] \Rightarrow t_1 = t_2$$

Similarmente, se puede expresar que en ningún marcaje futuro habrán dos transiciones listas para disparar:

$$\mu \vdash^* \mu', [I(t_1) \subseteq \mu', I(t_2) \subseteq \mu'] \Rightarrow t_1 = t_2$$

A diferencia de los autómatas finitos, en el caso de las RP el determinismo es una propiedad que puede o no tenerse, dependiendo de la morfología particular de una RP dada. En otras palabras, hay que examinar cada RP para saber si es o no determinista. El *problema del determinismo* consiste precisamente en decidir si una RP dada es o no determinista.

El problema del determinismo de una RP se puede reducir al de la alcanzabilidad de los marcajes, pues se trata de encontrar un marcaje μ_2 en el cual dos transiciones puedan disparar. Una vez propuesto tal marcaje, se pregunta si es alcanzable algún marcaje μ_1 del que μ_2 sea subconjunto. Los detalles de esta formulación se dejan al lector.

5.5.2 Decidibilidad de los problemas de análisis

Los problemas de análisis expuestos en la sección precedente son en general difíciles, y han atraído gran cantidad de estudios. En el caso de las RP booleanas, la totalidad de dichos problemas de análisis es decidible, por el hecho de que se trata esencialmente de autómatas finitos presentados de otra manera. Esto deberá quedar más claro después de la discusión que sigue.

Vamos a discutir únicamente la decidibilidad del problema de la alcanzabilidad de marcajes, pues todos los otros problemas se pueden expresar en términos de éste.

Teorema.- El problema de la alcanzabilidad en RP booleanas es decidible.

La prueba de este teorema se basa en la construcción del llamado *árbol de alcanzabilidad.* Dicho árbol se define de la manera siguiente:

- Cada nodo es un marcaje
- Cada rama representa el disparo de una transición
- La raíz es el marcaje inicial

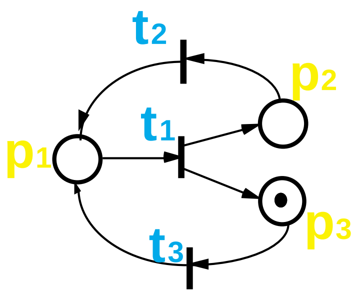
En el árbol de alcanzabilidad (AA) cada nodo tiene como hijos los marcajes a los que se puede llegar directamente: μ_2 es hijo de μ_1 ssi $\mu_1 \vdash \mu_2$.

De estas definiciones se podría desprender que si una RP tiene vida ilimitada, entonces su árbol de alcanzabilidad es de profundidad infinita, pues siempre hay transiciones que disparar. Para evitar esto se introduce la siguiente regla, que limita el crecimiento del AA:

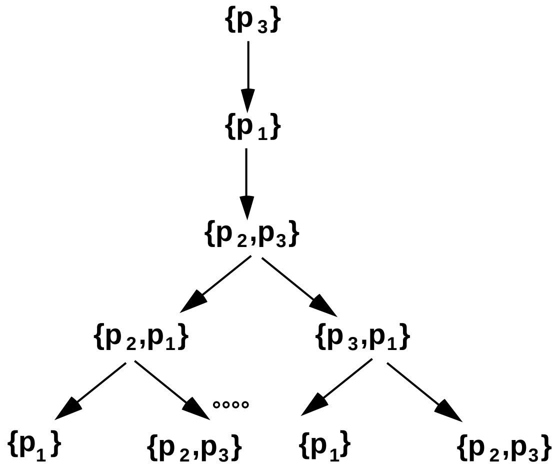
Cuando un marcaje se repite en el árbol, su nodo no se desarrolla.

De esta forma se garantiza que el AA es de tamaño finito. En efecto, la profundidad máxima del AA de $((P, T, I, O), \mu)$ no puede ser mayor que $2^{|P|}$, pues para crear un nuevo nivel se requiere que ninguno de los nodos que se expanden haya aparecido antes, y en el peor caso sólo se expande un nodo en cada nivel, habiendo $2^{|P|}$ marcajes posibles. Al mismo tiempo, la anchura del AA está limitada por $2^{|P|}$.

Por ejemplo, sea la siguiente RP marcada:



Su AA quedaría como sigue:



En este punto todos los marcajes en las hojas del AA están en deadlock o repetidos en éste.

La finitud del AA de toda RP booleana es fundamental para hacer que el problema de la alcanzabilidad de los marcajes sea decidible. En efecto, $\mu_1 \in R((P, T, I, O), \mu)$ ssi μ_1 está en el árbol. En consecuencia, el algoritmo de decisión para determinar si un marcaje es o no alcanzable es simplemente un recorrido del árbol en cualquier orden convencional (orden, preorden, etc.). Esto termina la prueba del teorema. QED

Corolario.- Los problemas de la viveza de una RP, de la secuencia de disparo y del determinismo de una RP booleana son todos decidibles. Los detalles de la prueba de este corolario se dejan al lector.

5.6 Lenguajes de Petri

En esta sección se estudian las RP como aceptadores de lenguajes, lo que permite establecer un puente natural con algunas nociones estudiadas en los autómatas finitos.

Se parte de la idea de identificar el disparo de una transición con la recepción de un caracter de entrada en un autómata. Es decir, si en el autómata se recibe una "a", en la RP dispara una transición con etiqueta a -suponiendo que se encuentre habilitada.

Esto nos permite hablar de *Lenguajes de Redes de Petri* (LRP) -que serán los lenguajes aceptados por una RP-, y comparar así las RP con los AF. Una RP va a estar caracterizada por el lenguaje que acepta.

Las transiciones de la RP tendrán etiquetas que serán letras del alfabeto Σ, y tendrán también lugares finales, como en la figura 5.3.

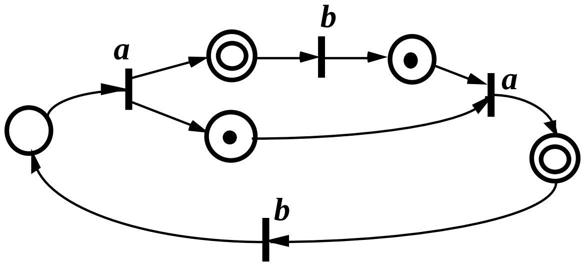


Figura 5.3: RP con lenguaje

Una palabra será aceptada por la RP cuando haya agotado todos sus caracteres en la entrada, y cuando el marcaje al que se llegue tenga una marca en al menos uno de los lugares finales.

Ejemplo.- La RP de la figura 5.3 acepta el lenguaje $a(ab)^*$.

Formalmente, una RP marcada con etiquetas en las transiciones sería de la forma $((P, T, I, O, F), \Sigma, \lambda, \mu)$, donde P, T, I, O y μ tienen el significado habitual; Σ es un alfabeto, y λ es una función de las transiciones T al alfabeto Σ; el conjunto F contiene los lugares finales.

Es necesario modificar la definición de la relación entre marcajes $\vdash$, para incorporar la parte de la palabra de entrada que se lleva leída. Así tendremos configuraciones de la forma $[[\mu, \alpha]]$, $\alpha \in \Sigma^*$, y en consecuencia la relación $\vdash$ se define como:

$[[\mu_1, \sigma\alpha]] \vdash_t [[\mu_2, \alpha]]$ ssi para $t \in T$, con $\lambda(t) = \sigma$, $t \in T$, $\alpha \in \Sigma^*$, se tiene: $\mu_2 = \mu_1 - I(t) \cup O(t)$.

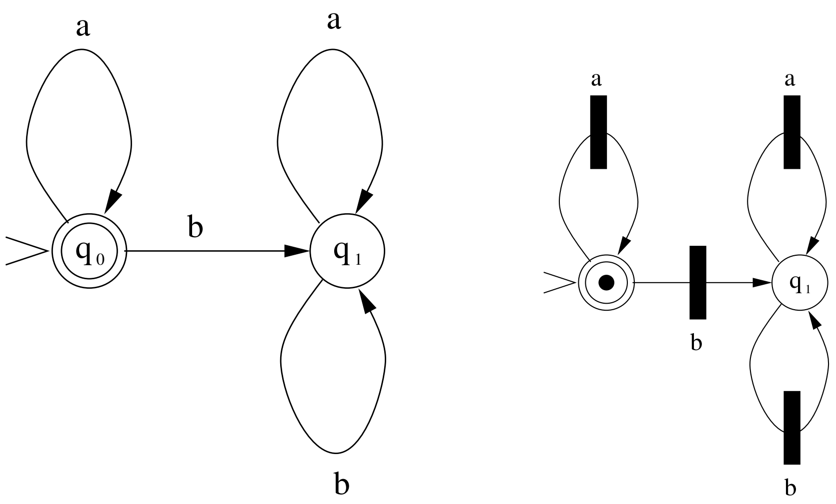
Así, una palabra w es aceptada por una RP marcada $((P, T, I, O), \mu_0)$ ssi $[[\mu_0, w]] \vdash^* [[\mu, \varepsilon]]$, con $\{p\} \in \mu$, donde $p \in F$.

Una vez definida la noción de palabra aceptada, es posible relacionar directamente las RP con los autómatas finitos que hemos visto anteriormente.

5.6.1 Equivalencia RP - AF

Tomando el enfoque de los LRP, la equivalencia entre las RP booleanas y los AF quedaría demostrada si es posible, a partir de una RP, construir un AF que acepte el mismo lenguaje, y viceversa.

Es trivial probar que para todo AF se puede construir una RP que acepte el mismo lenguaje. En efecto, si representamos gráficamente un AF, basta con poner en cada flecha una transición con la misma etiqueta. También hay que reemplazar un estado inicial por un marcaje inicial. Por ejemplo, en la figura 5.4 (a) y (b) se ilustran, respectivamente, un AF



(a) AF (b) RP

Figura 5.4: AF y RP equivalentes

y una RP equivalentes.

Formalmente, hay que expresar $((P,T,I,O),\mu_0)$ en términos de (K,Σ,δ,s,F). Esto es enteramente directo y se deja como ejercicio al lector.

La equivalencia en el sentido RP→AF requiere del uso del *árbol de alcanzabilidad*. Dado un AA, se construye un AF (K,Σ,δ,s,F) en el que cada estado de K es uno de los marcajes que aparecen en el AA, y siempre que en el AA existe un marcaje μ con un hijo μ_t por el disparo de la transición t, con $\lambda(t)=\sigma$, se agrega una transición $((\mu,\sigma),\mu_t)$ al AF. Los detalles de esta construcción se dejan como ejercicio al lector. Esto completa la prueba de equivalencia RP-AF.

Ejercicio.- El procedimiento de construcción RP-AF ¿genera necesariamente un AF determinista?.

Comentarios bibliográficos

Se puede encontrar una excelente introducción a las redes de Petri en el libro [6]. Para las presentes notas, se hizo una adaptación al enfoque del libro [5]. Otro texto en redes de Petri, más moderno que el de Peterson, y en español (el original; no es traducción), es el texto de Silva [9].

La fuente original es la tesis doctoral de Petri [7], de la que se desconoce su disponibilidad.

5.7 Ejercicios

1. Para la RP mostrada en la figura 5.5, construya el árbol de alcanzabilidad de marcajes, y con base en él determine la solución a las siguientes preguntas:

   (a) ¿La transición t_0 está viva?

   (b) ¿La transición t_2 tiene vida ilimitada?

   (c) ¿La RP tiene vida ilimitada?

   (d) ¿La RP es deteminista?

   (e) ¿La RP es inmortal?

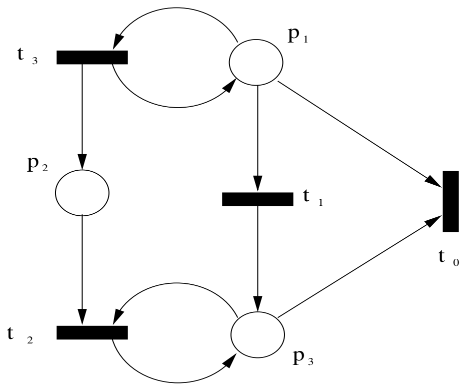


Figura 5.5: Red de Petri

2. Suponga un rancho de ordeña en el que hay 3 vacas lecheras V_1, V_2 y V_3, que se pueden ordeñar a la vez, salvo por el hecho de que sólo hay 2 rancheros R_1 y R_2, y cada vaca sólo puede ser ordeñada por un ranchero, armado de su respectiva cubeta C_1 y C_2. Cuando se llena una cubeta de leche, debe ser vaciada en el depósito por los dos rancheros juntos, pues lleva 80 lts. de leche.

   Modele el funcionamiento del ordeñadero con RP booleanas.

3. Suponga una variante de las redes de Petri booleanas en que, cuando "cae" una marca en un lugar que ya estaba marcado, la marca desaparece en vez de conservarse. Defina para este caso la relación entre marcajes $\vdash$.

4. Una RP se llama "segura"cuando no puede ocurrir que una marca caiga donde ya habia anteriormente otra marca.

   (a) Demuestre que para una RP booleana es posible verificar si la RP es segura, y dar un procedimiento que permita hacerlo.

   (b) Aplicar el procedimiento del inciso anterior a la red a la figura 5.5.

5. Una RP es "peligrosa"cuando a partir de un marcaje inicial, en todo momento es posible seguir alguna secuencia que eventualmente la lleve a "deadlock".

(a) Proponga un método para decidir si una RP es peligrosa.

(b) Aplique el metodo de (a) a la RP mostrada en la figura 5.5.

6. Se dice que una RP es "conservativa" cuando el número de marcas permanece constante aún cuando ocurran disparos.

   (a) Defina formalmente el hecho de que una RP es conservativa, dando una condición necesaria y suficiente que se debe satisfacer, de la forma Una RP es conservativa ssi <condición>.

   (b) ¿Se puede decidir si una RP dada es conservativa o no lo es? Proponga, en caso afirmativo, un procedimiento para decidirlo.

7. Se dice que una RP es "segura" cuando no puede caer una marca en un lugar que ya estaba marcado. En caso contrario es insegura.

   (a) Defina formalmente el hecho de que una RP es insegura, dando una condición lógica que se debe satisfacer, de la forma "Una RP es insegura ssi <condición>. (Hint: la condición debe indicar formalmente el hecho de que hay algún marcaje alcanzable en el que, además de haber marcas en todos los lugares de entrada de una cierta transición, también hay marcas en alguno de los lugares de salida).

   (b) ¿Se puede decidir si una RP dada es segura o no lo es? Proponga, en caso afirmativo, un procedimiento para decidirlo (Hint: utilice el árbol de alcanzabilidad).

8. Suponga que en una RP $((P, T, I, O), \mu)$ se ponen etiquetas σ a las transiciones, donde $\sigma \in \Sigma$ es un símbolo del alfabeto. Se define así una función de etiquetado $\Omega : T \to \Sigma$. Una transición sin lugares de entrada (que llamaremos transición de entrada) puede disparar si el símbolo que llega es igual a la etiqueta. Similarmente, una transición sin lugares de salida (transición de salida) puede disparar al estar lista, produciendo como salida el símbolo de su etiqueta. Una RP puede tener varias transiciones de entrada y varias de salida, pero cada una de ellas tendrá una sóla etiqueta.

   Al igual que los AF de Mealy y de Moore, estas RP podrán transformar un flujo de datos de entrada en datos de salida, o bien calcular funciones (el cálculo de funciones requiere que la RP sea determinista). Vamos a suponer que las transiciones con lugares de entrada disparan en cuanto están listas.

   (a) Diseñe una RP etiquetada que recibe cadenas de unos y ceros, y cada vez que reciba un grupo de dos unos seguidos, convierte el segundo uno a cero. Ejemplo: $f(010110) = 010100$.

   (b) Establezca la definición de configuración y de relación $\vdash$ entre configuraciones, así como la definición de función calculada.

Parte II

Lenguajes libres de contexto y sus máquinas

Capítulo 6

Gramáticas y lenguajes libres de contexto

La representación de los lenguajes regulares que aquí estudiaremos se fundamenta en la noción de gramática formal. Intuitivamente, una gramática es un conjunto de reglas para formar correctamente las frases de un lenguaje; así tenemos la gramática del español, del francés, etc. La formalización que presentaremos de la noción de gramática es debida a N. Chomski, y está basada en las llamadas reglas.[1]

Una regla es una expresión de la forma $\alpha \to \beta$, en donde tanto α como β son cadenas de símbolos en donde pueden aparecer tanto elementos del alfabeto Σ como unos nuevos símbolos, llamados variables.[2] La aplicación de una regla $\alpha \to \beta$ a una palabra $u\alpha v$ produce la palabra $u\beta v$. En consecuencia, las reglas de una gramática pueden ser vistas como reglas de reemplazo.

Definición.- Una gramática es un cuádruplo (V, Σ, R, S) en donde:

- V es un *alfabeto*
- Σ es un subconjunto de V, que contiene a los símbolos *terminales*.
- R, el conjunto de *reglas*, es un subconjunto finito de $V^* \times V^*$
- S, el *símbolo inicial*, es un elemento de $V - \Sigma$.

Usualmente las reglas no se escriben como pares ordenados (α, β), sino como $\alpha \to \beta$; esto es simplemente cuestión de notación.

[1] Repetiremos, para facilidad de lectura, algunas definiciones dadas en capítulos anteriores.

[2] Tratándose de los compiladores, se les llama "terminales" a los elementos de Σ, y "no terminales" a las variables.

Las reglas permiten establecer una relación entre cadenas en V^*, que es la *relación de derivación*, $\Rightarrow_G$ para una gramática G. Esta relación se define de la siguiente manera:

Definición.- $\alpha \Rightarrow_G \beta$ ssi existen cadenas $x, y \in V^*$, tales que $\alpha = xuy$, $\beta = xvy$, y existe una regla $u \to v$ en R.

La cerradura reflexiva y transitiva de $\Rightarrow_G$ se denota por $\Rightarrow_G^*$. Una palabra $w \in V^*$ es *derivable* a partir de G si existe una secuencia de derivación $S \Rightarrow_G \alpha_1 \Rightarrow_G \alpha_2 \Rightarrow_G \ldots \Rightarrow_G w$, es decir, $S \Rightarrow_G^* w$.

Definición.- El lenguaje generado por una gramática G, esto es, $L(G)$, es igual a $\{w \in \Sigma^* : S \Rightarrow_G^* w\}$.

6.1 La jerarquía de Chomski

Es posible restringir la forma de las reglas de manera que se acomoden a patrones predeterminados. Por ejemplo, se puede imponer que el lado izquierdo de las reglas sea elemento de V (en vez de V^*). Al restringir las reglas de la gramática se restringen también las palabras que se pueden generar; no es extraño que las reglas de formas más restringidas generan los lenguajes más reducidos. N. Chomski propuso varias formas estándares de reglas que se asocian a varias clases de languajes, que ordenó de manera tal que forman una jerarquía, es decir, los lenguajes más primitivos están incluídos en los más complejos. Así tenemos las siguientes familias de lenguajes:

1. Gramáticas regulares: las reglas son elementos de $(V - \Sigma) \times \Sigma((V - \Sigma) \cup \{\varepsilon\})$, es decir, son de la forma $A \to aB$ o bien $A \to a$. [3]

2. Gramáticas Libres de Contexto (GLC): las reglas son elementos de $(V - \Sigma) \times V^*$.

3. Gramáticas sensitivas al contexto: las reglas son de la forma $\alpha A \beta \to \alpha \Gamma \beta$, con $\alpha, \beta, \Gamma \in V^*$, $A \in V - \Sigma$.

6.2 Gramáticas libres y sensitivas al contexto

Las GLC deben su nombre a una comparación con otro tipo de gramáticas, las llamadas *sensitivas al contexto*, definidas arriba, donde para una regla $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, el símbolo A sólo puede generar β cuando se encuentra rodeado por el "contexto" $\alpha_1 \ldots \alpha_2$. En cambio, en las GLC no es necesario especificar un contexto, por esto se llaman "libres de contexto".

[3] Estas gramáticas ya fueron discutidas en el capítulo 3.

Las gramáticas sensitivas al contexto son estrictamente más poderosas que las GLC; [4] un ejemplo es el lenguaje de las cadenas de la forma $a^n b^n c^n$, para el que no hay ninguna GLC. En cambio, una gramática sensitiva al contexto sería la siguiente (sólo damos las reglas):

1. $S \to aBTc$	2. $T \to ABTc$
3. $T \to ABc$	4. $BA \to BX$
5. $BX \to YX$	6. $YX \to AX$
7. $AX \to AB$	8. $aA \to aa$
9. $aB \to ab$	10. $bB \to bb$

En esta gramática, las reglas 1 a 3 generan A, a, B y c no necesariamente en orden (las A y B van a estar alternadas). Luego las reglas 4 a 7 permiten reordenar las A y B, para que todas las A queden antes que todas las B, [5] y finalmente las reglas 8 a 10 permiten generar los terminales sólamente cuando las letras están en el orden correcto. Como un ejemplo, la palabra $aaabbbccc$ se puede generar de la forma siguiente:

$S \Rightarrow_1 aBTc \Rightarrow_2 aBABTcc \Rightarrow_3 aBABABccc \Rightarrow_4 aBXBXBccc \Rightarrow_5 aYXYXBccc \Rightarrow_6 aAXAXBccc \Rightarrow_7 aABABBccc \Rightarrow_4 aABXBBccc \Rightarrow_5 aAYXBBccc \Rightarrow_6 aAAXBBccc \Rightarrow_7 aAABBBccc \Rightarrow_8 aaABBBccc \Rightarrow_8 aaaBBBccc \Rightarrow_9 aaabBBccc \Rightarrow_{10} aaabbBccc \Rightarrow_{10} aaabbbccc$.

6.3 Lenguajes Libres de Contexto (LLC)

Los LLC son aquellos lenguajes que pueden ser generados por una gramática libre de contexto (GLC). En una GLC las reglas son de la forma $A \to \alpha$, donde A es un no-terminal, y α es una cadena compuesta de terminales y de no terminales, en cualquier orden. Se vé por lo tanto que el formato de las reglas es menos rígido que para las gramáticas regulares, y así toda gramática regular es GLC pero no viceversa.

Los LLC son interesantes porque permiten modelar muchos lenguajes útiles, en particular los lenguajes de programación, tales como Pascal, lisp, C, etc. Por otra parte, el análisis automático de los LLC es computacionalmente mucho más eficiente que el de otras clases de lenguajes más generales. Teóricamente son importantes porque están asociados a los autómatas de pila, que veremos más adelante.

Formalmente, una GLC es un cuádruplo (V, Σ, R, S), en donde V es el *alfabeto* (un conjunto de símbolos), $\Sigma \subseteq V$ es el alfabeto de los símbolos *terminales*, $S \in V - \Sigma$ es el *símbolo inicial*, y $R \subseteq (V - \Sigma) \times V^*$ es el conjunto de *reglas*.

La definición de la relación de derivación $\Rightarrow_G$, y de su cerradura transitiva y reflexiva

[4]Trate de definir formalmente qué significa "estrictamente más poderosas".

[5]De hecho bastaría con una regla $BA \to AB$, salvo que ésta no comple con el formato de las gramáticas sensitivas al contexto.

$\Rightarrow_G^*$, así como de lenguaje generado por la gramática son las mismas que las vistas para las gramáticas regulares.

Ejemplo.- La siguiente GLC (V, Σ, R, S) genera el lenguaje P de los paréntesis bien balanceados, $\{(), (()), ()(), (())(), (()())(), \ldots\}$: $V = \{S, (,)\}$, $\Sigma = \{(,)\}$, y R contiene las siguientes reglas:

1. $S \rightarrow (S)$
2. $S \rightarrow SS$
3. $S \rightarrow ()$

Nótese que el lenguaje de los paréntesis bien balanceados no es regular (ejercicio: probar esta aseveración). Esto prueba que los LLC forman una clase estrictamente mas grande que la de los lenguajes regulares.

Prueba de corrección.- Para hacer la prueba por inducción en la longitud de la derivación, necesitamos primero generalizar el enunciado de forma que sea aplicable a las palabras con variables que aparecen a la mitad de la derivación. Esto es, necesitamos un lenguaje extendido donde se admita que las palabras contengan variables. Hacemos la siguiente definición:

$$P_X = \{\alpha \in V^* | elim(S, \alpha) \in P\}$$

Es decir, eliminando las "S" de las palabras de P_X, obtenemos palabras de paréntesis bien balanceados.

Base de la inducción.- En 0 pasos, se tiene (trivialmente) una palabra en P_X.

Hipótesis de inducción.- En k pasos, se generan palabras en P_X, de la forma $\alpha S \beta$, con $\alpha, \beta \in V^*$.

Paso de inducción.- A la palabra $\alpha S \beta$, generada en k pasos, se le pueden aplicar las reglas 1-3. Evidentemente la aplicación de las reglas 2 y 3 generan palabras $\alpha SS\beta$ y $\alpha\beta$ en LX. Aunque es menos evidente, la aplicación de la regla 1 produce palabras $\alpha(S)\beta$, que también están en L_X.

Finalmente, la última regla que debe aplicarse es la 3, resultando en una palabra con los paréntesis bien balanceados. QED

Prueba de completez.- Mientras que en la prueba que acabamos de hacer mostramos que todas las palabras que se generan con la gramática dada tienen sus paréntesis bien balanceados, faltaría por probar que toda palabra hecha de paréntesis bien balanceados puede ser efectivamente generada por la gramática. Esto último es lo que se conoce como *completez*. Por ejemplo, si eliminamos la regla 2 de la gramática, de todas maneras la prueba de corrección que acabamos de hacer seguiría funcionando, pero en cambio no habrá completez, porque algunas palabras, como $(())(())$ no pueden ser generadas por la gramática.

En el caso que nos ocupa, vamos a hacer una prueba por inducción sobre la longitud de la palabra.

Base de la inducción: La gramática puede generar todas las palabras de longitud 0 (Por la regla 3).

Hipótesis de inducción: La gramática puede generar todas las palabras de longitud menor o igual a k. (Claramente k es par).

Paso de inducción: Notamos que para una palabra dada w en P (esto es, que tiene los paréntesis bien balanceados), $|w| = k + 2$ sólo hay dos posibilidades: [6].

1. w se puede partir en w_1 y w_2, $w = w_1 w_2$, de forma tal que $w_1, w_2 \in P$.
2. w no se puede partir en dos partes.

En el caso 1, aplicando inicialmente la regla $S \to SS$, se debe poder generar w_1 a partir de la S de la izquierda, por hipótesis de inducción, ya que $|w_1| \leq k$. Similarmente para w_2, con la S de la derecha.

En el caso 2, $w = (w')$, donde $w' \in P$, es decir, al quitar los dos paréntesis más externos se tiene una palabra con paréntesis bien balanceados (¿Porqué?). Como $|w'| = k$, por hipótesis de inducción w' se puede generar con la gramática. La palabra w se puede entonces generar aplicando primero la regla $S \to (S)$, y luego continuando con la derivación de w' que existe por hipótesis de inducción.

Esto completa la prueba. QED

6.3.1 Arboles de derivación

Las GLC tienen las propiedad de que las derivaciones pueden ser representadas en forma arborescente. Así, por ejemplo, la palabra $(()())()$ puede ser derivada por la gramática de los paréntesis bien balanceados de la manera que se ilustra en la figura 6.1.

En dicha figura se puede apreciar la *estructura* que se encuentra implícita en la palabra $(()())()$. A estas estructuras se les llama *árboles de derivación*, y son de vital importancia para la teoría de los compiladores de los lenguajes de programación.

[6] El paso de inducción se hace en $k+2$ y no en $k+1$ porque todas las palabras en P tienen longitud par

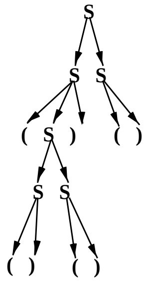


Figura 6.1: Paréntesis bien balanceados

Se puede considerar que un árbol de derivación es más abstracto que una derivación "lineal" -es decir, una sucesión $S \Rightarrow \ldots \Rightarrow w$- en el sentido de que para un sólo árbol de derivación puede haber varias derivaciones lineales, según el orden en que se decida "expander" los no terminales. Por ejemplo, para el árbol de la figura arriba, hay al menos las derivaciones siguientes:

1. $S \Rightarrow_2 SS \Rightarrow_3 (S)S \Rightarrow_1 (S)() \Rightarrow_2 (SS)() \Rightarrow_1 (S())() \Rightarrow_1 (()())()$.
2. $S \Rightarrow_2 SS \Rightarrow_1 S() \Rightarrow_3 (S)() \Rightarrow_2 (SS)() \Rightarrow_1 (()S)() \Rightarrow_1 (()())()$.

Formalmente, un árbol de derivación es un grafo dirigido arborescente [7] definido de la manera siguiente:

Definición.- Sea $G = (V, \Sigma, R, S)$ una GLC. Entonces un árbol de derivación cumple las siguientes propiedades:

1. Cada nodo tiene una etiqueta [8]
2. La raíz tiene etiqueta S.
3. La etiqueta de los nodos que no son hojas debe estar en $V - \Sigma$.
4. Si un nodo n tiene etiqueta A, y los nodos $n_1, \ldots, n_m$ son sus hijos (de izquierda a derecha), con etiquetas respectivamente $A_1, \ldots, A_m$, entonces $A \rightarrow A_1, \ldots, A_m \in R$.
5. Si un nodo n tiene etiqueta ε, entonces n es un nodo hoja, y es el único hijo de su nodo padre.

[7] Un grafo arborescente se caracteriza por no tener ciclos, y por el hecho de que existe una trayectoria única para llegar de la raíz a un nodo cualquiera.

[8] Formalmente, una etiqueta es una función que va del conjunto de nodos al conjunto de símbolos de donde se toman las etiquetas, en este caso $V \cup \{\varepsilon\}$.

Definición.- La cadena de caracteres que resulta de concatenar los caracteres terminales encontrados en las etiquetas de los nodos hoja, en un recorrido en orden del árbol de derivación, se llama el *producto* del árbol.

Es decir, al efectuar un recorrido en orden del árbol de derivación recuperamos la cadena a partir de la cual se construyó dicho árbol. Asi, el problema de compilar una cadena de caracteres consiste en construír el árbol de derivación a partir del producto de éste.

6.3.2 Ambigüedad en GLC

La correspondencia entre los árboles de compilación y sus productos no es necesariamente biunívoca. En efecto, hay GLC en las cuales para ciertas palabras hay más de un árbol de compilación. Sea por ejemplo la siguiente GLC, para expresiones aritméticas sobre las variables x y y.

1. $E \to E + E$
2. $E \to E * E$
3. $E \to x$
4. $E \to y$

Con esta gramática, para la expresión $x + y * x$ existen los dos árboles de derivación de las figuras 6.2(a) y (b).

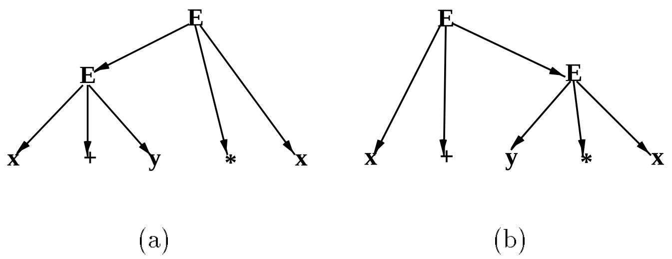


Figura 6.2: Dos árboles para $x + y * x$

En este ejemplo, el hecho de que existan dos árboles de derivación para una misma expresión es indeseable, pues cada árbol indica *una manera distinta* de estructurar la expresión. En efecto, en el árbol de la izquierda, al resultado de la suma $(x + y)$ se multiplica con x, mientras que en el de la derecha sumamos x al resultado de multiplicar x con y; por lo tanto el significado que se asocia a ambas expresiones puede ser distinto.

Existen técnicas para eliminar la ambigüedad de una GLC; en general estas técnicas consisten en introducir nuevos no-terminales de modo que se eliminen los árboles de derivación no deseados. Para nuestro ejemplo de los operadores aritméticos, es clásica la solución que consiste en introducir, además de la categoría de las Expresiones (no-terminal E), la de los *términos* (T) y *factores* (F), dando la gramática con las reglas siguientes:

1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to x$

Con esta nueva GLC, el árbol de derivación de la figura 6.2.a se elimina, quedando finalmente una adaptación del árbol de 6.2.b a la GLC con términos y factores, lo cual se deja como ejercicio al lector.

Sin embargo, estas técnicas de eliminación de ambigüedad no son siempre aplicables, y de hecho hay algunos LLC para los que es imposible encontrar una gramática no ambigua; estos lenguajes se llaman inherentemente ambiguos. Un ejemplo, dado en [Hopcroft, Ullman 79] junto con la prueba correspondiente, es el siguiente:

$$L = \{a^n b^n c^m d^m\} \cup \{a^n b^m c^m d^m\},\ n{\geq}1, m{\geq}1$$

6.3.3 Derivaciones izquierda y derecha

En una gramática no ambigua G, a una palabra $w \in L(G)$ corresponde un sólo árbol de derivación A_G; sin embargo, puede haber varias derivaciones para obtener w a partir del símbolo inicial, $S \Rightarrow \ldots \Rightarrow w$. Una manera de hacer única la manera de derivar una palabra consiste en restringir la elección del símbolo que se va a "expander" en curso de la derivación. Por ejemplo, si tenemos en cierto momento de la derivacion la palabra $(S())(S)$, en el paso siguiente podemos aplicar alguna regla de la gramática ya sea a la primera o a la segunda de las S. En cambio, si nos restringimos a aplicar las reglas sólo al no terminal que se encuentre más a la izquierda en la palabra, entonces habrá una sóla opción posible.

Desde luego, el hecho de elegir el no terminal más a la izquierda es arbitrario; igual podemos elegir el no terminal más a la derecha.

Definición.- Se llama *derivación izquierda* de una palabra w a una secuencia $S \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n \Rightarrow w$ en donde, para pasar de w_i a w_{i+1}, se aplica una regla al no terminal de w_i que se encuentre más a la izquierda.

Teorema.- Para una gramática no ambigua G, y una palabra $w \in L(G)$, existe sólamente una derivación izquierda $S \Rightarrow^* w$.

Prueba: La derivación izquierda corresponde a un recorrido en preorden del árbol de derivación, expandiendo los no terminales que vamos encontrando en el camino. Ahora bien, se sabe que existe un sólo recorrido en preorden para un árbol dado.

6.4 Propiedades de los LLC

Los LLC tienen propiedades similares a las que vimos para los lenguajes regulares. Las principales son:

6.4.1 Propiedades de cerradura

Teorema.- Los LLC son cerrados con respecto a la unión, concatenación, y cerradura de Kleene.

Prueba de la cerradura respecto a la unión.- Sean $G_1 = (V_1, \Sigma_1, R_1, S_1)$ y $G_2 = (V_2, \Sigma_2, R_2, S_2)$ dos GLC; se puede suponer, sin pérdida de generalidad, que los no terminales de G_1 y G_2 son disjuntos. [9] La GLC que genera $L(G_1) \cup L(G_2)$ es

$$G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\}, S)$$

.

En efecto, para una palabra $w \in L(G_1)$ la derivación comienza aplicando $S \to S_1$, y después se continúa con la derivación a partir de S_1 (recuérdese que por hipótesis $w \in L(G_1)$). [10] Similarmente se hace para una palabra $w \in L(G_2)$.

Ejemplo.- Obtener una GLC para el lenguaje $\{a^n b^m | n \neq m\}$. Expresamos este lenguaje como la unión de otros dos:

$$\{a^n b^m | n \neq m\} = \{a^n b^m | n < m\} \cup \{a^n b^m | n > m\}$$

[9] Ejercicio: justificar porqué no hay pérdidad de generalidad.
[10] Ejercicio: Explicar porqué es necesario suponer que los no terminales son disjuntos.

Cada uno de estos dos lenguajes tiene una GLC fácil de obtener; por ejemplo, para las palabras de la forma $a^n b^m$ con más as que bs, tenemos la gramática:[11]

1. $S \to aSb$
2. $S \to Sb$
3. $S \to b$

Prueba de la cerradura respecto a la concatenación.- Similarmente, $L(G_1)L(G_2)$ es generado por la siguiente GLC:

$$G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \to S_1 S_2\}, S)$$

.

Ejemplo.- Definimos el lenguaje de los "prefijos palíndromes" como aquel formado por palabras que tienen un prefijo (no vacío) que se lee igual de izquierda a derecha que de derecha a izquierda. Por ejemplo, las palabras $aabab$, aba y $aabaa$ son prefijos palíndromes.[12] Proponer una GLC que genere exactamente el lenguaje de los prefijos palíndromes en el alfabeto $\{a, b\}$. El problema parece difícil, pero podemos considerar cada palabra de este lenguaje como formada por dos partes: la parte palíndrome y el resto de la palabra. Dicho de otra forma, el lenguaje L_{PP} de los prefijos palíndromes es igual a la concatenación de L_P y L_R, donde L_P es el lenguaje de los palíndromes y L_R genera la parte restante de las palabras. El lenguaje de los palíndromes en $\{a, b\}$ tiene una gramática muy simple, con las reglas $S \to aSa$, $S \to bSb$, $S \to \varepsilon$.[13] Por otra parte, como la "parte restante" que viene después de la parte palíndrome puede ser cualquier cosa, está claro que L_R es simplemente $\{a, b\}^*$, que por ser regular es LLC, y que tiene una GLC con las reglas: $T \to aT$, $T \to bT$, $T \to \varepsilon$. La GLC de L_{PP} es la combinación de ambas gramáticas, de acuerdo con la fórmula dada más arriba.[14]

Prueba de la cerradura respecto a la cerradura de Kleene.- Similarmente, $L(G_1)^*$ es generado por la GLC:

$$G = (V_1, \Sigma_1, R_1 \cup S_1 \to \varepsilon, S_1 \to S_1 S_1, S_1)$$

.

Los LLC no son cerrados respecto a la intersección ni al complemento. Sin embargo, para probar esto es necesario obtener antes otros resultados, que se presentan a continuación.

[11] Ejercicio: probar la corrección de esta gramática

[12] La última de ellas puede ser vista de dos maneras distintas como prefijo palíndrome.

[13] Ejercicio: probar la corrección de esta gramática.

[14] Ejercicio: obtener en detalle dicha gramática.

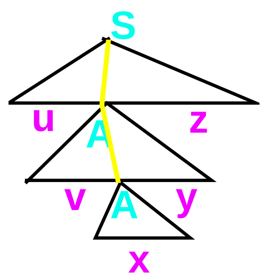


Figura 6.3:

6.4.2 Teorema de bombeo para los LLC

Teorema.- Existe para cada $G \in$ GLC un número k tal que toda $w \in L(G)$, donde $|w| > k$, puede ser escrita como $w = uvxyz$, de tal manera que v y y no son ambas vacías, y que $uv^n xy^n z \in L(G)$ para cualquier $n \geq 0$.

Este teorema es similar en esencia al teorema de bombeo para los lenguajes regulares; nos dice que siempre hay manera de introducir ("bombear") subrepticiamente subcadenas a las palabras de los LLC. Nos sirve para probar que ciertos lenguajes no son LLC.

Prueba.- Basta con probar que hay una derivación

$$S \Rightarrow^* uAz \Rightarrow^* uvAyz \Rightarrow^* uvxyz = w$$

pues al aparecer el mismo no-terminal en dos puntos de la derivación, es posible insertar ese "trozo" de la derivación cuantas veces se quiera (incluyendo cero). Esa parte de la derivación, que tiene la forma $uAz \Rightarrow^* uvAyz$, es una especie de "ciclo" sobre el no-terminal A, que recuerda lo que ocurría con el teorema de bombeo para los lenguajes regulares.

Para probar que existen en la derivación ciclos de la forma $uAz \Rightarrow^* uvAyz$, la idea será verificar que el tamaño vertical del árbol (su profundidad) es mayor que la cantidad de no-terminales disponibles. En consecuencia, algún no-terminal debe repetirse.

Primero, la cantidad de no-terminales para una gramática (V, Σ, R, S) es $|V - \Sigma|$.

A continuación examinemos el problema de verificar si los árboles de derivación pueden tener una profundidad mayor que $|V - \Sigma|$.

Sea $m = max(\{|\alpha| \,|A \rightarrow \alpha \in R\})$. Ahora bien, un árbol de profundidad p tiene a lo más m^p hojas (¿porqué?), y por lo tanto un árbol A_w para w, con $|w| > m^p$ tiene profundidad mayor que p. Así, toda palabra de longitud mayor que $m^{|V-\Sigma|}$ tendrá necesariamente una profundidad mayor que $|V - \Sigma|$, y por lo tanto, algún no-terminal estará repetido en la derivación; sea A ese no-terminal. Vamos a representar el árbol de derivación en la figura 6.3.

Como se vé, hay un subárbol del árbol de derivación (el triángulo intermedio en la figura 6.3) en el que el símbolo A es la raíz y también una de las hojas. Está claro que ese subárbol puede ser ya sea removido o insertado cuantas veces se quiera, y quedará siempre un árbol derivación válido; cada vez que dicho subárbol sea insertado, las subcadenas v e y se repetirán una vez más. Esto completa la prueba. En la figura se aprecia porqué es importante que v e y no sean ambas vacías. QED

Ejemplo.- El lenguaje $\{a^n b^n c^n\}$ no es LLC. Esto se prueba por contradicción. Supóngase que $\{a^n b^n c^n\}$ es LLC. Entonces, de acuerdo con el teorema de bombeo, para una cierta k, $a^{k/3} b^{k/3} c^{k/3}$ puede ser escrita como $uvxyz$, donde v e y no pueden ser ambas vacías. Existen dos posibilidades:

1. v o y contienen varias letras (combinaciones de a, b o c). Pero, según el teorema, uv^2xy^2z es de la forma $a^n b^n c^n$, lo cual es imposible, ya que al repetir v o y, forzosamente las letras quedarán en desorden;

2. Tanto v como y (el que no sea vacío) contienen un sólo tipo de letra (repeticiones de a, b o c). En este caso, si $uvxyz$ es de la forma $a^n b^n c^n$, uv^2xy^2z no puede ser de la misma forma, pues no hemos incrementado en forma balanceada las tres letras, sino a lo más dos de ellas.

En ambos casos se contradice la hipótesis de que $\{a^n b^n c^n\}$ es LLC.

Este ejemplo nos permite probar que los LLC no son cerrados respecto a la intersección ni respecto a la complementación:

Teorema.- Los LLC no son cerrados con respecto a la intersección.

Prueba.- Los lenguajes L_1 y L_2 formados por las palabras de la forma $a^n b^n c^m$ y $a^m b^n c^n$ respectivamente son LLC. [15] Sin embargo, su intersección es el lenguaje $\{a^n b^n c^n\}$, que acabamos de probar que no es LLC.

Teorema.- Los LLC no son cerrados con respecto a la complementación.

Prueba.- Si los LLC fueran cerrados con respecto a la complementación, también lo serían con respecto a la intersección, ya que, de acuerdo con las identidades de la teoría de conjuntos, $L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$. [16]

Debe tenerse cuidado al interpretar la "no cerradura" de los LLC con respecto a la complementación e intersección. En efecto, esto *no* quiere decir, por ejemplo, que el complemento de un LLC necesariamente no será LLC. En el siguiente ejemplo se dá un caso específico.

[15] Ejercicio.- Probar este hecho.
[16] L^c es una abreviatura para $\Sigma^* - L$.

Ejemplo.- Probar que el complemento del lenguaje $\{a^n b^n\}$ es LLC. Para esto, vamos a clasificar las palabras de $L = \{a^n b^n\}^c$ en dos categorías:

1. Las que contienen la cadena "*ba*", esto es, $w = \alpha ba\beta$
2. Las que no contienen "*ba*", esto es, $w \neq \alpha ba\beta$

Claramente esta clasificación es exhaustiva. El objetivo de esta clasificación es distinguir las causas por las que una palabra en $\{a, b\}^*$ no es de la forma $a^n b^n$: la primera es que tiene letras en desorden –esto es, contiene la cadena "*ba*"– como en "*abba*"; la segunda es que, no habiendo letras en desorden, la cantidad de a's y b's no es la misma, como en "*aaaa*", "*abbb*", etc.

El caso (1) es muy simple, pues el lenguaje L_1 cuyas palabras contienen la cadena "*ba*" es regular y por lo tanto LLC.

Es fácil ver que el caso (2) corresponde al lenguaje $L_2 = \{a^n b^m | n \neq m\}$, pues como no tiene b inmediatamente antes que a, todas las a están antes de todas las b. L_2 puede ser expresado como la unión de dos lenguajes LLC, como se vió en un ejemplo presentado anteriormente, y por la cerradura de los LLC a la unión, se concluye que L_1 es LLC.

Finalmente, $\{a^n b^n\}^c = L_1 \cup L_2$, y por la cerradura de los LLC a la unión, se concluye que L es LLC.

6.4.3 Propiedades de decidabilidad

Hay ciertas preguntas sobre los lenguajes libres de contexto y sus gramáticas que es posible contestar, mientras que hay otras preguntas que no se pueden contestar en el caso general. Vamos a examinar primero dos preguntas que sí se pueden contestar con seguridad y en un tiempo finito. Para estas preguntas es posible dar un algoritmo o "receta" tal que, siguiéndolo paso por paso, se llega a concluir un *si* o un *no*. Tales algoritmos se llaman *algoritmos de decisión*, pues nos permiten decidir la respuesta a una pregunta. Las preguntas que vamos a contestar son las siguientes:

Teorema.- Dadas una gramática G y una palabra w, es posible decidir si $w \in L(G)$ cuando las reglas de G cumplen la propiedad: *"Para toda regla $A \rightarrow \alpha$, $|\alpha| > 1$, o bien $\alpha \in \Sigma$, es decir, el lado derecho tiene varios símbolos, o si tiene exactamente un símbolo, éste es terminal."*

Prueba: La idea para probar el teorema es que cada derivación incrementa la longitud de la palabra, porque el lado derecho de las reglas tiene en general más de un símbolo. En vista de que la longitud de la palabra crece con cada paso de derivación, sólo hay que examinar las derivaciones hasta una cierta longitud finita. Por ejemplo, la gramática de los paréntesis bien balanceados cumple con la propiedad requerida:

1. $S \rightarrow ()$
2. $S \rightarrow SS$
3. $S \rightarrow (S)$

Como en esta gramática el lado derecho mide 2 o más símbolos, la aplicación de cada regla reemplaza un símbolo por dos o más. Por lo tanto, para saber si hay una derivación de la palabra $()(())$, que mide 6 símbolos, sólo necesitamos examinar las derivaciones (izquierdas) de 5 pasos a lo más -y que terminan en una palabra hecha únicamente de terminales. Estas derivaciones son las siguientes:

1 paso:
$S \Rightarrow ()$

2 pasos:
$S \Rightarrow (S) \Rightarrow (())$

3 pasos:
$S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow ((()))$
$S \Rightarrow SS \Rightarrow ()S \Rightarrow ()()$

4 pasos:
$S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow (((S))) \Rightarrow (((())))$
$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow (()S) \Rightarrow (()())$
$S \Rightarrow SS \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()(())$
$S \Rightarrow SS \Rightarrow (S)S \Rightarrow (())S \Rightarrow (())()$

5 pasos:
$S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow (((S))) \Rightarrow ((((S)))) \Rightarrow ((((()))))$
$S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow ((SS)) \Rightarrow ((()S)) \Rightarrow ((()()))$
$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow (()S) \Rightarrow (()(S)) \Rightarrow (()(()))$
$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow ((())S) \Rightarrow ((())())$
$S \Rightarrow SS \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()((()))$
$S \Rightarrow SS \Rightarrow ()S \Rightarrow ()SS \Rightarrow ()()S \Rightarrow ()()()$
$S \Rightarrow SS \Rightarrow (S)S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())(())$
$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((()))S((()))()$
$S \Rightarrow SS \Rightarrow SSS \Rightarrow ()SS \Rightarrow ()()S \Rightarrow ()()()$

Es fácil ver que éstas son las únicas posibles derivaciones. [17]

[17] Ejercicio: hallar el método que se siguió para obtener las derivaciones mostradas, y probar que no se puede "escapar" ninguna derivación.

Con base en este grupo de derivaciones es simple probar que la palabra "(()()(" -de 6 caracteres de longitud- no pertenece al lenguaje generado por la gramática, pues si así fuera, estaría entre alguna de las palabras derivadas en 5 pasos o menos.

En el caso general se incluyen reglas de la forma $A \to a$, con $a \in \Sigma$. Para empezar observamos que las reglas de la forma $A \to a$ producen exclusivamente un terminal, por lo que, en el peor caso, se aplicaron tantas veces reglas de este tipo como letras tenga la palabra generada. Por ejemplo, sea la gramática de las expresiones aritméticas:

1. $E \to E + E$
2. $E \to E * E$
3. $E \to x$
4. $E \to y$

Esta gramática tiene reglas, como $E \to x$ y $E \to y$ que tienen en su lado derecho un caracter. Entonces, dada una expresión aritmética como $x * y + x$, que tiene 5 símbolos, a lo más se usan ese tipo de reglas en 5 ocasiones (de hecho se vé que en una derivación de $x * y + x$ ese tipo de reglas se usa exactamente en 3 ocasiones). Ahora bien, para generar 5 terminales con reglas de la forma $A \to a$ se requieren 5 no-terminales. Esos 5 no-terminales se generan con las reglas de la forma $A \to a$, donde $|a| > 1$. En el peor de los casos, $|a| = 2$, por lo que se requerirán 4 pasos de derivación para llegar a los 5 no-terminales. Eso dá un total de 5+4 = 9 pasos de derivación. Así, si queremos determinar en forma segura si la palabra $x * y + x$ pertenece o no al lenguaje generado por la gramática, sólo tenemos que examinar las derivaciones de longitud menor o igual a 9.

En general, para una palabra w de longitud l hay que examinar las derivaciones de longitud hasta $2 * l - 1$. Si la palabra se encuentra al final de alguna de esas derivaciones, la palabra pertenece al lenguaje, y en caso contrario no pertenece al lenguaje. Esto termina la prueba del teorema. QED

Desde luego, el procedimiento seguido en la prueba del teorema no es el más eficiente, ni mucho menos; es ampliamente preferible construir el árbol de compilación, aunque la prueba del teorema por este camino hubiera sido más difícil.

Nótese que en el enunciado del teorema nos estamos restringiendo a las GLC que satisfacen la condición: para toda regla $A \to \alpha$, $|\alpha| > 1$, o bien $\alpha \in \Sigma$, es decir, el lado derecho tiene varios símbolos, o si tiene exactamente un símbolo, éste es terminal. Cabe preguntarse si esto constituye una limitación, en el sentido de que hay muchas GLC que no cumplen dicha condición. De hecho la respuesta es no, pues existe un procedimiento para pasar de una GLC arbitraria a una GLC que satisface la condición del teorema.

Corolario .- Dada cualquier GLC G, es posible decidir si $w \in L(G)$.

La prueba de este corolario consiste en dar un procedimiento para transformar una GLC cualquiera G en una GLC G' que satisface las condiciones del teorema arriba enunciado.

6.4.4 Transformación de las GLC

La condición del teorema se puede reexpresar de manera "negativa" de la manera siguiente: la GLC no debe contener reglas de la forma $A \to \varepsilon$, que producen la cadena vacía, ni tampoco reglas de la forma $A \to B$, en que un no- terminal produce otro no-terminal. Así, si es posible eliminar esos tipos de reglas sin que la gramática altere su significado, entonces cualquier GLC puede modificarse para que cumpla con la condición del teorema. Pero primero veamos cómo es posible eliminar cada tipo de reglas que violan la condición.

6.4.5 Eliminación de reglas $A \to \varepsilon$

Supóngase la siguiente gramática para los paréntesis bien balanceados: [18]

$$S \to \varepsilon,\ S \to (S),\ S \to SS$$

Si queremos una GLC equivalente, pero sin emplear producciones vacías (como $S \to \varepsilon$), una idea sería, dando marcha atrás, suponer de dónde viene la S que queremos cambiar por ε. Sólo hay otras dos reglas en la gramática, de modo que esa S tuvo que ser generada ya sea por $S \to (S)$ o por $S \to SS$. En el caso de $S \to (S)$, una solución sería, en vez de hacer la derivación

$$S \Rightarrow^* \alpha S\beta \Rightarrow \alpha(S)\beta \Rightarrow \alpha()\beta,\ \alpha \in \Sigma*, \beta \in V^*$$

mejor hacer directamente la derivación

$$S \Rightarrow^* \alpha S\beta \Rightarrow \alpha()\beta$$

agregando una regla $S \Rightarrow ()$ a la gramática. Y en caso de que la S provenga de la regla $S \to SS$, se puede cambiar la derivación

$$S \Rightarrow^* \alpha S\beta \Rightarrow \alpha SS\beta \Rightarrow \alpha S\beta$$

por la derivación

[18] Sus reglas.

$$S \Rightarrow^* \alpha S\beta \Rightarrow \alpha SS\beta \Rightarrow \alpha S\beta$$

usando una nueva regla $S \to S$, o mejor aún, simplemente reemplazarla por

$$S \Rightarrow^* \alpha S\beta$$

sin ninguna regla adicional (la parte de la derivación $\alpha S\beta \Rightarrow \alpha SS\beta \Rightarrow \alpha S\beta$ desaparece por completo, pues no sirve de nada!).

Resumiendo, la idea que permite eliminar las reglas $A \to \varepsilon$ es la de irse un paso atrás, para examinar de dónde provino el no-terminal A que queremos eliminar, y por cada regla $B \to \alpha A\beta$ de la gramática agregar una regla $B \to \alpha\beta$, en que directamente ya se reemplazó A por ε. Una vez hecho esto, se pueden suprimir todas las reglas de la forma $A \to \varepsilon$, pues resultan redundantes.

Por ejemplo, sea la GLC de los paréntesis bien balanceados:

$$S \to (S),\ S \to SS,\ S \to \varepsilon.$$

Aplicando mecánicamente la transformación a dicha gramática, se tiene:

$$S \to (S),\ S \to SS,\ S \to (),\ S \to S$$

La regla $S \to S$ es evidentemente inútil y se puede eliminar, pero dejemos esto para el siguiente párrafo, en que nos ocuparemos de la eliminación de reglas de esa forma.

Otra cuestión más importante aún debe haber saltado a la vista escrutadora del lector perpicaz: la nueva GLC no es exactamente equivalente a la anterior! En efecto, la GLC original generaba la palabra vacía ε, mientras que la GLC transformada no la genera. Desde luego, el hecho de que una GLC contenga reglas de la forma $A \to \varepsilon$ no significa que el lenguaje contenga forzosamente a la palabra vacía; considérese por ejemplo la siguiente gramática:

$$S \to (A),\ S \to AA,\ A \to (A),\ A \to AA,\ A \to \varepsilon$$

cuyo lenguaje no contiene a la palabra vacía.

En caso de que el lenguaje en cuestión *realmente* contenga a la palabra vacía, no es posible estrictamente eliminar todas las producciones vacías sin alterar el significado de la gramática. En estos casos vamos a expresar el lenguaje como la unión $\{\varepsilon\} \cup L(G')$, donde G' es la gramática transformada. Este pequeño ajuste no modifica los resultados que obtuvimos arriba.

6.4.6 Eliminación de reglas $A \to B$

Supongamos ahora que se tiene la gramática con las reglas siguientes:

$$S \to (S),\ S \to BB,\ S \to (),\ B \to S$$

Claramente esta GLC es equivalente a la gramática dada anteriormente para generar los paréntesis bien balanceados. La única diferencia es que, en vez de utilizar la regla $S \to SS$, se tiene una regla $S \to BB$, y luego las B se transforman en S por la regla $B \to S$. Pero, ¿para que usar esos intermediarios, como B en este caso, cuando es posible generar directamente SS a partir de S? La idea de eliminar las reglas de la forma $A \to B$ viene de observar que dichas reglas no producen nada útil, simplemente introducen símbolos intermediarios, que es posible eliminar.

Es posible localizar todos los pares de no-terminales A y B tales que $A \Rightarrow^* B$ (ejercicio: ¿porqué?). Ahora bien, si hay reglas $B \to \Gamma_i$ en la gramática, entonces es posible añadir reglas $A \to \Gamma_i$ sin modificar el lenguaje. Ahora bien, si hacemos esto siempre que sea posible, las reglas de la forma $A \to B$ se vuelven inútiles, pues toda derivación:

$$\ldots \Rightarrow \alpha A \beta \Rightarrow^* \alpha B \beta \Rightarrow \alpha \Gamma_i \beta \Rightarrow \ldots$$

puede transformarse en:

$$\ldots \Rightarrow \alpha A \beta \Rightarrow \alpha \Gamma_i \beta \Rightarrow \ldots$$

sin modificar el lenguaje. Esto completa la prueba del corolario. QED

6.5 Formas Normales

En ocasiones es necesario expresar una GLC siguiendo un formato más preciso de las reglas que la simple forma $A \to \alpha$. Por ejemplo, en la sección anterior establecimos que las reglas debían tener en el lado derecho más símbolos que en el lado izquierdo, para así poder decidir si $w \in L(G)$, dadas una GLC G y una palabra w. Similarmente, existen otros formatos útiles para diferentes propósitos, que reciben el nombre de *formas normales*. Vamos a estudiar una de las formas normales más conocidas, la *forma normal de Chomsky* (FNCH).

La FNCH consiste en que las reglas pueden tener dos formas:

1. $A \to a$, $a \in \Sigma$
2. $A \to BC$, con $B, C \in (V - \Sigma)$

Esta forma normal, aparentemente tan arbitraria, tiene por objeto facilitar el análisis sintáctico de una palabra de entrada, siguiendo la estrategia siguiente: Se trata de construir el árbol de derivación de w de arriba hacia abajo (top- down), y por consiguiente se supone inicialmente que el símbolo inicial S puede producir la palabra w. En seguida se procede a dividir la palabra de entrada w en dos pedazos, $w = \alpha\beta$, para luego tomar alguna regla $S \to AB$, y tratar de verificar si se puede derivar a a partir de A y b a partir de B, es decir: $S \Rightarrow^* w$ ssi:

1. $w \in \Sigma$, hay una regla $S \to w$
2. $w = \alpha\beta$, hay una regla $S \to AB$, con $A \Rightarrow^* \alpha, B \Rightarrow^* \beta$

Esta manera de generar dos nuevos problemas similares al problema inicial, pero con datos más pequeños, es típicamente un caso de recursión. Este hecho permite pensar en un sencillo procedimiento recursivo para "compilar" palabras de un LLC. Sea $CC(A,u)$ la función que verifica si $A \Rightarrow^* u$. Entonces un algoritmo de análisis sintáctico sería el siguiente:

$CC(A,w)$:

1. Si $|w| > 1$, dividirla en u y v, $w = uv$;
   Para cada regla de la forma $A \to UV$, intentar $CC(U,u)$ y $CC(V,v)$
2. Si $|w| = 1$, buscar una regla $A \to w$.

Si en el punto 1 la división de la palabra no nos llevó a una compilación exitosa (es decir, los llamados recursivos $CC(U,u)$ y $CC(V,v)$ no tuvieron éxito), puede ser necesario dividir la palabra de otra manera. Dicho de otra forma, puede ser necesario ensayar todas las formas posibles de dividir una palabra en dos partes, antes de convencerse de que ésta pertenece o no a nuestro lenguaje. Aún cuando esto puede ser muy ineficiente computacionalmente, es innegable que el algoritmo es conceptualmente muy sencillo.

El siguiente problema a examinar es si efectivamente es posible transformar una GLC cualquiera G en otra GLC G' que está en la FNCH. Vamos a efectuar esta transformación en dos etapas: en una primera etapa, llevaremos G a una forma intermedia G_{temp}, para pasar después de G_{temp} a G'.

En G_{temp} las reglas son de las formas:

1. $A \to a, a \in \Sigma$
2. $A \to \beta, \beta \in (V - \Sigma)^+$

En G_{temp}, los lados derechos de las reglas son, ya sea un terminal, o una cadena (no vacía) de no-terminales. La manera de llevar una GLC cualquiera a la forma intermedia consiste en introducir reglas $A \to a$, $B \to b$, etc., de modo que podamos poner, en vez de un terminal a, el no-terminal A que le corresponde, con la seguridad de que después será posible obtener a a partir de A. Por ejemplo, considérese la siguiente GLC:

1.- $S \to aX$

2.- $S \to bY$

3.- $X \to Ya$

4.- $X \to ba$

5.- $Y \to bXX$

6.- $Y \to aba$

Como se vé, el obstáculo para que esta GLC esté en la forma intermedia es que en los lados derechos de varias reglas (1, 2, 3, 5) se mezclan los terminales y los no-terminales. Por otra parte, hay reglas (4, 6) que en el lado derecho tienen varios terminales. Entonces añadimos las reglas:

7.- $A \to a$

8.- $B \to b$

y modificamos las reglas (1,2,3,5), reemplazando a por A y b por B:

$1'$.- $S \to AX$

$2'$.- $S \to BY$

$3'$.- $X \to YA$

$4'$.- $X \to BA$

$5'$.- $Y \to BXX$

$6'$.- $Y \to ABA$

con lo que la gramática ya está en la forma intermedia. La equivalencia de la nueva gramática con respecto a la original es muy fácil de probar.

Luego, para pasar de G_{temp} a la FNCH, puede ser necesario dividir los lados derechos de algunas reglas en varias partes. Esto se formaliza de la manera siguiente:

$A \rightarrow B\beta, |\beta| > 1$, se transforma en $A \rightarrow BW$ y $W \rightarrow \beta$.

(El no-terminal W debe ser nuevo, es decir, no formar previamente parte de la gramática). Cada vez que se aplica esta transformación, el lado derecho de la regla afectada se reduce en longitud en una unidad, por lo que, aplicándola repetidas veces, se debe poder llegar siempre a reglas cuyo lado derecho tiene exactamente dos no-terminales. Para el ejemplo visto arriba, la regla $5'$ se convierte en:

$5''$.- $Y \rightarrow BW$

$5'''$.- $W \rightarrow XX$

Similarmente se puede transformar la regla $6'$, dejando la gramática (reglas $1'$, $2'$, $3'$, $4'$, $5''$, $5'''$, $6''$, $6'''$, 7, 8) en la FNCH.

6.6 Ejercicios

1. Sea $L = \{a^n b^m c^p d^q \mid n = m = p + q\}$. ¿Es L libre de contexto? Proponga (y explique) una GLC o pruebe que no es posible.

2. Muestre que la siguiente gramática es / no es ambigua: $G = (V, \Sigma, R, S)$, con:

   $V =$ { PROG, IF, STAT, if, then, else, condición, stat}

   $\Sigma =$ { if, then, else, condición, stat}

   $R =$ { PROG $\rightarrow$ STAT, STAT $\rightarrow$ if condición then STAT, STAT $\rightarrow$ if condición then STAT else STAT, STAT $\rightarrow$ stat}

   $S =$ PROG

3. Para el lenguaje $\{a^i b^j c^k \mid \neg(i = j = k)\}$:

   (a) Proponga una gramática libre de contexto.

   (b) Pruebe por inducción que dicha gramática es correcta.

4. Para el lenguaje en $\{a, b\}^*$ en que las palabras tienen el mismo número de a's que de b's,

   (a) Proponga una gramática libre de contexto.

   (b) Pruebe por inducción sobre la longitud de la derivación que dicha gramática es correcta.

(c) Pruebe que la gramática es completa.

5. Pruebe mediante el teorema de bombeo que el lenguaje $\{a^n b^{n+m} c^{n+m+k}, n, m, k = 1, 2, 3, \ldots\}$ no es libre de contexto. (Hint: las cadenas v e y se pueden repetir 0 veces).

6. Proponga una gramática libre de contexto para el lenguaje $\{a^n b^{n+m} c^m\}$ (Hint: use las propiedades de cerradura de los LLC).

7. La siguiente gramática genera el lenguaje en $\{a, b\}^*$ en que las palabras tienen la misma cantidad de a que de b.

   (a) $S \rightarrow aSb$

   (b) $S \rightarrow bSa$

   (c) $S \rightarrow SS$

   (d) $S \rightarrow e$

   (a) Hacer una derivación izquierda de la palabra "babbabaa" usando esta gramática.

   (b) Probar la corrección de esta gramática. Ponga cuidado en formar el enunciado generalizado (que se aplica a las palabras con variables), en enunciar la hipótesis de inducción, y en aplicar el enunciado generalizado para el último paso de la derivación.

   (c) Transforme esta gramática a la forma normal de Chomsky. Para esto, elimine las producciones vacías, las producciones "inútiles", etc.

   (d) Suponiendo que el analizador de Chomsky se llama mediante la expresión $CC(V, w)$, donde V es una variable y w una cadena, haga el árbol de llamadas que permiten compilar la palabra "aababb".

8. Conteste las siguientes preguntas, justificando la respuesta:

   (a) ¿La concatenación de un lenguaje regular con un libre de contexto será necesariamente libre de contexto?

   (b) ¿Todo lenguaje libre de contexto tendrá algún subconjunto que sea regular?

   (c) ¿Todo lenguaje libre de contexto será subconjunto de algún lenguaje regular?

   (d) Si $A \cup B$ es libre de contexto, ¿será A libre de contexto?

   (e) ¿La intersección de un lenguaje regular con un libre de contexto será regular?

9. ¿La unión de un LLC con un Lenguaje regular es LLC? (Probar)

10. ¿La intersección de un LLC con un Lenguaje Regular es Regular? (Probar)

11. Llamamos "útil" a un símbolo no terminal A de una gramática libre de contexto que cumple con dos propiedades:

    (a) $S \Rightarrow^* aAb$, $a, b \in V^*$, donde V es el alfabeto (terminales y no terminales),

(b) $A \Rightarrow^* w$, $w \in \Sigma^*$, donde Σ es el alfabeto de los terminales.

Dada una cierta GLC y un símbolo no terminal A, ¿Es decidible si A es útil o no lo es? Pruebe su respuesta, y en caso afirmativo proponga el método de decisión.

12. Suponga que un lenguaje L es libre de contexto. Considere el reverso de ese lenguaje, LR, formado por las palabras de L escritas en orden inverso. ¿Es necesariamente libre de contexto LR ? Pruebe formalmente su respuesta (no basta con dar "razones de peso").

13. ¿El lenguaje $\{w = a^i b^m c^n \mid i > m > n\}$ es libre de contexto? Pruebe su respuesta.

14. Pruebe que los siguientes lenguajes son / no son regulares:

    (a) La unión de un LLC con un Lenguaje regular

    (b) La intersección de un LLC con un Lenguaje Regular

15. Conteste las siguientes preguntas, justificando la respuesta:

    (a) ¿El reverso de un lenguaje regular es también regular? (Por ejemplo, $Reverso(\{ab, baa\}) = \{ba, aab\}$).

    (b) ¿El reverso de un lenguaje libre de contexto será también libre de contexto?

Capítulo 7

Autómatas de Pila

Puesto que los autómatas finitos no son suficientemente poderosos para aceptar los LLC, [1] cabe preguntarnos *qué tipo de autómata se necesitaría para aceptar los LLC.*

Una idea es *agregar algo* a los AF de manera que de alguna manera se incremente su poder de cálculo.

Para ser más concretos, tomemos por ejemplo el lenguaje de los paréntesis bien balanceados, que sabemos que es propiamente LLC. [2] ¿Qué máquina se requiere para distinguir las palabras de paréntesis bien balanceados de las que tienen los paréntesis desbalanceados? Una primera idea podría ser la de una máquina que tuviera un registro aritmético que le permitiera *contar* los paréntesis; dicho registro sería controlado por el control finito, quien le mandaría símbolos I para incrementar en uno el contador y D para decrementarlo en uno. A su vez, el registro mandaría un símbolo Z para indicar que está en cero, o bien N para indicar que no está en cero. Entonces para analizar una palabra con paréntesis lo que haríamos sería llevar la cuenta de cuántos paréntesis han sido abiertos pero no cerrados; en todo momento dicha cuenta debe ser positiva o cero, y al final del cálculo debe ser exactamente cero. Por ejemplo, para la palabra $(())()$ el registro tomaría sucesivamente los valores $1, 2, 1, 0, 1, 0$. [3]

Como un segundo ejemplo, considérese el lenguaje de los palíndromes (palabras que se leen igual al derecho y al revés, como ANITALAVALATINA). Aquí la máquina contadora no va a funcionar, porque se necesita recordar toda la primera mitad de la palabra para poder compararla con la segunda mitad. Más bien pensaríamos en una máquina que tuviera la capacidad de recordar cadenas de caracteres arbitrarias, no números. Siguiendo esta idea, podríamos pensar en añadir al AF un almacenamiento auxiliar, que llamaremos *pila*, donde se podrán ir depositando caracter por caracter cadenas arbitrariamente grandes, como se

[1] ¡Cuidado! Esto no impide que un LLC en particular pueda ser aceptado por un AF, cosa trivialmente cierta si tomamos en cuenta que todo lenguaje regular es a la vez LLC.

[2] "Propiamente LLC" quiere decir que el lenguaje en cuestión es LLC pero no regular.

[3] Ejercicio: diseñar en detalle la tabla describiendo las transiciones del autómata.

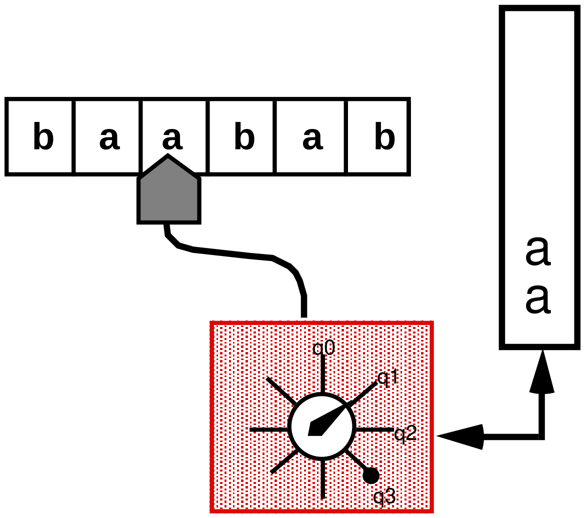


Figura 7.1: Autómata con una pila auxiliar

aprecia en la figura 7.1.

La *pila* funciona de manera que el último caracter que se almacena en ella es el primero en salir (orden "LIFO"), como si empiláramos platos uno encima de otro, y naturalmente el primero que quitaremos es el último que hemos colocado. Un aspecto crucial de la pila es que sólo podemos modificar el "tope" de la pila, que es el extremo por donde entran o salen los caracteres. Los caracteres a la mitad de la pila no son accesibles sin quitar antes los que están encima de ellos.

La pila tendrá un alfabeto propio, que puede o no coincidir con el alfabeto de la palabra de entrada. Esto se justifica porque puede ser necesario introducir en la pila caracteres especiales usados como separadores, según las necesidades de diseño del autómata.

7.1 Formalización de los AP

Un autómata de pila es un séxtuplo $(K, \Sigma, \Gamma, \Delta, s, F)$, donde:

- K es un conjunto de estados
- Σ es el alfabeto de entrada
- Γ es el alfabeto de la pila
- $s \in K$ es el estado inicial
- $F \subseteq K$ es un conjunto de estados finales,

- $\Delta \subseteq (K \times \Sigma^* \times \Gamma^*) \times (K \times \Gamma^*)$ es la relación de transición.

Ahora describiremos el funcionamiento de los AP. Si tenemos una transición $((p, u, \beta)(q, \gamma)) \in \Delta$, el AP hace lo siguiente:

- Estando en el estado p, consume u de la entrada;
- Saca β de la pila;
- Llega a un estado q;
- Mete γ en la pila

Las operaciones típicas en las pilas –el "push" y el "pop"– pueden ser vistas como casos particulares de las transiciones de nuestro AP; en efecto, si sólo queremos meter la cadena γ a la pila, se haría con la transición $((p, u, \varepsilon)(q, \gamma))$ ("push"), mientras que si sólo queremos sacar caracteres de la pila se hará con la transición $((p, u, \beta)(q, \varepsilon))$ ("pop").

Ahora formalizaremos el funcionamiento de los AP, para llegar a la definición del lenguaje aceptado por un AP. Para ello seguiremos el mismo método que usamos en el caso de los AF, método que reposa completamente en la noción de *configuración*.

Definición.- Una configuración es un elemento de $K \times \Sigma^* \times \Gamma^*$.

Por ejemplo, una configuración podría ser $[[q, abbab, \otimes aa\#a]]$ –obsérvese que seguimos la misma notación que para representar las configuraciones de los AF. Puede verse que las transiciones se definen como una *relación*, no como una función, por lo que de entrada se les formaliza como autómatas no deterministas.

Ahora definimos la relación $\vdash$ entre configuraciones de la manera siguiente:

Definición.- Sea $M = (K, \Sigma, \Gamma, \Delta, s, F)$ un AP, entonces $[[p, ux, \beta\alpha)| \vdash_M [[q, x, \gamma\alpha]]$ ssi existe $((p, u, \beta), (q, \gamma)) \in \Delta$. En general vamos a omitir el subíndice de $\vdash_M$, quedando simplemente como $\vdash$. La cerradura reflexiva y transitiva de $\vdash$ es $\vdash^*$. [4]

Definición.- Un AP $M = (K, \Sigma, \Gamma, \Delta, s, F)$ acepta una palabra $w \in \Sigma^*$ ssi $[[s, w, \varepsilon]] \vdash_M^* [[p, \varepsilon, \varepsilon]]$, donde $p \in F$. $L(M)$ es el conjunto de palabras aceptadas por M.

Ejemplo.- Considérese el lenguaje $\{wcw^R\}$, $w \in \{a, b\}$, donde w^R representa la palabra w al revés (por ejemplo, $(perro)^R = orrep$). El siguiente AP acepta este lenguaje:

$$K = \{s, f\}, F = \{f\}, \Sigma = \{a, b, c\}, \Gamma = \{a, b\}$$

[4] En el capítulo de los AF se define la cerradura reflexiva y transitiva.

Δ está representada en la siguiente tabla:

(s, a, ε)	(s, a)
(s, b, ε)	(s, b)
(s, c, ε)	(f, ε)
(f, a, a)	(f, ε)
(f, b, b)	(f, ε)

La estrategia seguida para el diseño de esta máquina consiste en hacer un grupo de transiciones (las dos primeras) que toda a y b que se reciba, se almacena en la pila como va llegando. En el momento en que se recibe la c, el AP cambia a un estado f que "recuerda" que ya pasó la primera mitad de la palabra. A partir de ese momento, hay que comparar cada letra que se vaya recibiendo en la entrada con una letra que se saca de la pila, y si al final de la palabra de entrada la pila también está vacía, se aceptará aquélla.

Por ejemplo, la siguiente tabla muestra un cálculo que permite aceptar la palabra $w = abcba$:

Estado	Falta leer	Pila	Transición
s	$abcba$	ε	
s	$bcba$	a	1
s	cba	ba	2
f	ba	ba	3
f	a	a	5
f	ε	ε	4

Ahora consideremos otro ejemplo, el lenguaje $\{ww^R\}$, $w \in \{a, b\}$. La única diferencia con respecto al lenguaje del ejemplo anterior es que no aparece la c que nos indicaba la división de las dos mitades de las palabras. El siguiente AP acepta este lenguaje:

$K = \{s, f\}, F = \{f\}, \Sigma = \{a, b, c\}, \Gamma = \{a, b\}$

Δ está representada en la siguiente tabla:

(s, a, ε)	(s, a)
(s, b, ε)	(s, b)
$(s, \varepsilon, \varepsilon)$	(f, ε)
(f, a, a)	(f, ε)
(f, b, b)	(f, ε)

Como se vé, la única diferencia con respecto al autómata del ejemplo anterior es que en la tercera transición, en vez de consumir la c y hacer en consecuencia el cambio de estado de s

a f, ahora *simplemente hace el cambio de estado* sin consumir ninguna entrada ni modificar la pila ! Esta transición parece muy peligrosa, porque se puede "disparar" en cualquier momento, y si no lo hace exactamente cuando hemos recorrido ya la mitad de la palabra, el AP podrá llegar al final a un estado que no sea final, rechazando en consecuencia la palabra de entrada. Entonces, ¿cómo saber que estamos exactamente a la mitad de la palabra?

Conviene en este punto recordar que en un autómata no determinista una palabra es aceptada cuando *existe un cálculo que permite aceptarla*, independientemente de que un cálculo en particular se vaya por un camino erróneo. Lo importante es, pues, que *exista* un cálculo que acepte la palabra en cuestión. Por ejemplo, la siguiente tabla muestra un cálculo que permite aceptar la palabra $w = abba$:

Estado	Falta leer	Pila	Transición
s	$abba$	ε	
s	bba	a	1
s	ba	ba	2
f	ba	ba	3
f	a	a	5
f	ε	ε	4

Ejercicio.- Diseñar un AP para el lenguaje $\{w \in \{a,b\}^* | \#aenw = \#benw\}$

7.2 Relación entre AF y AP

Teorema.- Todo lenguaje aceptado por un AF es también aceptado por un AP

Este resultado debe quedar intuitivamente claro, puesto que los AP son una extensión de los AF.

Prueba: Sea $(K, \Sigma, \Delta, s, F)$ un AF; el AP $(K, \Sigma, \emptyset, \Delta', s, F)$, con $\Delta' = \{((p, u, \varepsilon)(q, \varepsilon)) \mid (p, u, q) \in \Delta\}$ acepta el mismo lenguaje.

7.3 Relación entre AP y LLC

Ahora vamos a establecer el resultado por el que iniciamos el estudio de los AP, es decir, verificar si son efectivamente capaces de aceptar los LLC.

Teorema.- Los autómatas de pila aceptan exactamente los LLC.

La prueba de este teorema se puede dividir en dos partes:

1. Si M es un AP, entonces $L(M)$ es un LLC
2. Si L es un LLC, entonces hay un AP M tal que $L(M) = L$

Vamos a presentar únicamente la prueba con la parte 2, que es la más interesante en tanto que propone un procedimiento para pasar de una gramática a un AP "equivalente".

Sea una gramática $G = (V, \Sigma, R, S)$. Entonces un AP M que acepta exactamente el lenguaje generado por G se define como sigue:

$$M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$$

donde Δ contiene las siguientes transiciones:

1. Una transición $((p, \varepsilon, \varepsilon)(q, S))$
2. Una transición $((q, \varepsilon, A)(q, x))$ para cada $A \to x \in R$
3. Una transición $((q, \sigma, \sigma)(q, \varepsilon))$ para cada $\sigma \in \Sigma$

Ejemplo.- Obtener un AP que acepte el LLC generado por la gramática con reglas:

1. $S \to aSa$
2. $S \to bSb$
3. $S \to c$

Las transiciones del AP correspondiente están dadas en la tabla siguiente:

1	$(p, \varepsilon, \varepsilon)$	(q, S)
2	(q, ε, S)	(q, aSa)
3	(q, ε, S)	(q, bSb)
4	(q, ε, S)	(q, c)
5	(q, a, a)	(q, ε)
6	(q, b, b)	(q, ε)
7	(q, c, c)	(q, ε)

El funcionamiento de este AP ante la palabra *abcba* aparece en la siguiente tabla:

Estado	Falta leer	Pila
p	$abcba$	ε
q	$abcba$	S
q	$abcba$	aSa
q	$bcba$	Sa
q	$bcba$	$bSba$
q	cba	Sba
q	cba	cba
q	ba	ba
q	a	a
q	ε	ε

Vamos a justificar intuitivamente el método que acabamos de introducir para obtener un AP equivalente a una gramática dada. Si observamos las transiciones del AP, veremos que sólamente tiene dos estados, p y q, y que el primero de ellos desaparece del cálculo en el primer paso; de esto concluímos que el AP no utiliza los estados para "recordar" características de la entrada, y por lo tanto reposa exclusivamente en el almacenamiento de caracteres en la pila. En efecto, podemos ver que las transiciones del tipo 2 (transiciones 2-4 del ejemplo), lo que hacen es reemplazar en la pila una variable por la cadena que aparece en el lado derecho de la regla correspondiente. Dado que la (única) transición de tipo 1 (transición 1 del ejemplo) coloca el símbolo inicial en la pila, a continuación lo que hacen las reglas de tipo 2 es realmente *efectuar toda la derivación dentro de la pila de la palabra de entrada*, reemplazando un lado izquierdo de una regla por su lado derecho. Una vez hecha la derivación de la palabra de entrada, –la cual estaría dentro de la pila, sin haber aún gastado un sólo caracter de la entrada– podemos compararla caracter por caracter con la entrada, por medio de las transiciones de tipo 3.

Existe sin embargo un problema técnico: si observamos la "corrida" para la palabra $abcba$, nos daremos cuenta de que no estamos aplicando las reglas en el orden descrito en el párrafo anterior, esto es, primero la transición del grupo 1, luego las del grupo 2 y finalmente las del grupo 3, sino que más bien en la cuarta línea de la tabla se consume un caracter a (aplicación de una transición del grupo 3) seguida de la aplicación de una transición del grupo 2. Esto no es casualidad; lo que ocurre es que *las variables no pueden ser reemplazadas por el lado derecho de una regla si dichas variables no se encuentran en el tope de la pila.* En efecto, recuérdese que los AP sólo pueden accesar el caracter que se encuentra en el tope de la pila. Por esto, se hace necesario, antes de reemplazar una variable por la cadena del lado derecho de una regla, "desenterrar" dicha variable hasta que aparezca en el tope de la pila, lo cual puede hacerse consumiendo caracteres de la pila (y de la entrada, desde luego) mediante la aplicación de transiciones del tipo 3.

De la construcción del AP que hemos descrito, concluímos con la siguiente proposición:

$$S \to^* w \text{ ssi } [[p, w, \varepsilon]] \vdash^*_{M(G)} [[q, \varepsilon, \varepsilon]]$$

donde $M(G)$ denota al AP construído a partir de la gramática G por el procedimiento recién descrito.

Todavía nos queda por probar que para todo AP hay una gramática equivalente. A este respecto remitimos al lector a la referencia [Lewis,Papadimitriou].

La equivalencia de los AP y de las GLC permite aplicar todas las propiedades de los LLC a los AP. En particular, sabemos que si $L = L_1 \cup L_2$, y tanto L_1 como L_2 son LLC, existe un AP para L. Sin embargo, no hemos visto ningún procedimiento específico para obtener un AP para L dados los AP para L_1 y para L_2. Esto no es difícil de hacer, y así tenemos el siguiente resultado:

Si $M_1 = (K_1, \Sigma_1, \Gamma_1, \Delta_1, s_1, F_1)$ y $M_2 = (K_2, \Sigma_2, \Gamma_2, \Delta_2, s_2, F_2)$ son dos AP (disjuntos) que aceptan los lenguajes L_1 y L_2 respectivamente, el siguiente AP acepta $L_1 \cup L_2$:

$$M_{1\cup 2} = (K_1 \cup K_2 \cup \{s\}, \Sigma_1 \cup \Sigma_2, \Gamma_1 \cup \Gamma_2, \{((s, \varepsilon, \varepsilon), (s_1, \varepsilon)), ((s, \varepsilon, \varepsilon), (s_2, \varepsilon))\} \cup \Delta_1 \cup \Delta_2, s, F_1 \cup F_2))$$

Ejemplo.- Diseñar un AP que acepte el lenguaje $L = \{a^i b^j c^k | \neg(i = j = k)\}$. Nos damos cuenta de que L es la unión de dos lenguajes, que son:

$$L = \{a^i b^j c^k | i \neq j\} \cup \{a^i b^j c^k | j \neq k\}$$

Para cada uno de estos dos lenguajes es fácil obtener su AP. Para el primero de ellos, el AP almacenaría primero las a's en la pila, para luego ir descontando una b por cada a de la pila; las a's deben acabarse antes de terminar con las b's o bien deben sobrar a's al terminar con las b's; las c's no modifican la pila y simplemente se verifica que no haya a o b después de la primera c. Dejamos los detalles como ejercicio para el lector.

7.4 AP deterministas

En muchas aplicaciones prácticas el no determinismo de los autómatas es indeseable, en el sentido de que en la práctica no es suficiente con saber que "existe una manera" de aceptar la palabra de entrada, sino que el autómata *debe* encontrar el cálculo que conduce a la aceptación de la entrada. Dicho de otra manera, si nuestro AP fracasó en el intento de aceptar la palabra de entrada, nosotros quisiéramos poder atribuir este hecho a que la palabra no pertenece al lenguaje, y no simplemente a un posible cálculo que se fué por un camino equivocado. Por ejemplo imagínese un compilador de C++ que al recibir un programa correcto a veces lo aceptara y a veces no –desde luego sus usuarios no estarían muy contentos.

Por estas razones, se ha hecho necesario definir autómatas de pila que se comporten en forma determinista. El determinismo se interpretaría como el hecho de que en cualquier punto de un cálculo, no debe poderse ir a dos configuraciones distintas. Esta condición se puede formalizar de la manera siguiente:

$$C \vdash C_1 \;\&\; C \vdash C_2 \;\Rightarrow C_1 = C_2$$

Esta fórmula puede leerse como "si de una configuración dada puedo pasar a otras dos configuraciones, éstas son iguales (la misma)". Esta es una forma (muy frecuentemente utilizada en matemáticas) de decir que algo es único.

Vamos a utilizar otra definición de determinismo, equivalente a la anterior, pero que en la práctica va a resultar más útil, porque la definición anterior, al hacer referencia a "toda configuración" posible, nos deja en la imposibilidad de verificar si es cierta o falsa para un caso dado.

Definición.- Una transición $((p, u, \beta)(q, \xi))$ es aplicable a una configuración $[[p, w, \gamma]]$ si $w = u\delta$, y $\gamma = \beta\eta$.

Vamos a proponer un método para determinar si en un AP en particular se tiene la propiedad del determinismo. Primero es necesario definir unos conceptos auxiliares:

Definición.- Dos cadenas α y β son *consistentes* si $\alpha = \beta\gamma$ o bien $\beta = \alpha\gamma$.

Definición.- Dos transiciones $((p_1, w_1, \gamma_1)(q_1, \delta_1))$, $((p_2, w_2, \gamma_2)(q_2, \delta_2))$ son *compatibles* si $p_1 = p_2$, w_1 y w_2 son consistentes y γ_1 y γ_2 son consistentes

Definición.- Un AP es determinista si no tiene dos transiciones compatibles.

Ejercicio.- Mostrar que el AP del ejemplo antes mostrado para el lenguaje $\{wcw^R\}$ es determinista, mientras que el de $\{ww^R\}$ no lo es.

Es importante observar que el determinismo es una propiedad que puede o no tener un AP dado, y no, como en el caso de los AF, una consecuencia de definir las transiciones como una relación o como una función.

Finalmente hacemos notar –sin probarlo– que los lenguajes que pueden ser aceptados por algún AP determinista son un subconjunto propio de los LLC. Esto quiere decir que al restringirnos a AP deterministas se pierde poder de cálculo. Una implicación práctica de esto es que no se pueden hacer compiladores deterministas para todos los LLC.

7.5 Compiladores LL

El método visto en la sección precedente para obtener un AP a partir de una GLC puede ser considerado como una manera de construir un *compilador* para el lenguaje correspondiente a la GLC dada. Desde luego, para tener un verdadero compilador se requiere que el AP resultante sea determinista, pues sería inaceptable que un mismo compilador diera resultados diferentes al compilar varias veces un mismo programa.

Una manera de forzar a que un AP no determinista se vuelva determinista consiste en proveer un método para decidir, cuando hay varias transiciones aplicables, cual de ellas va a ser efectivamente aplicada. En el caso de los compiladores esto se puede hacer mediante el llamado *principio de previsión*.

El principio de previsión consiste en que podamos "observar" un caracter de la palabra de entrada que aún no ha sido leído (esto es llamado en inglés "lookahead", mirar hacia adelante). El caracter leído por adelantado nos permite en algunas ocasiones decidir adecuadamente cual de las transiciones del AP conviene aplicar.

Ejemplo.- Supóngase la GLC con reglas $S \rightarrow aSb$, $S \rightarrow \varepsilon$, que representa el lenguaje $\{a^n b^n\}$. La construcción del AP correspondiente es directa y la dejamos como ejercicio. Ahora bien, teniendo una palabra de entrada $aabb$, la traza de ejecución comenzaría de la manera siguiente:

Estado	Falta leer	Pila
p	$aabb$	ε
q	$aabb$	S

En este punto, no se sabe si reemplazar en la pila S por ε o por aSb, al ser transiciones aplicables tanto $((q,\varepsilon,S),(q,\varepsilon))$ como $((q,\varepsilon,S),(q,aSb))$. En cambio, si tomamos en cuenta que el siguiente caracter en la entrada será a, es evidente que no conviene reemplazar S por ε, pues entonces la a de entrada no podría ser cancelada. Entonces hay que aplicar la transición $((q,\varepsilon,S),(q,aSb))$. Continuamos la ejecución:

Estado	Falta leer	Pila
...	...	...
q	$aabb$	aSb
q	abb	Sb [5]
q	abb	$aSbb$
q	bb	Sbb

Al ver que el siguiente caracter de entrada será una b, nos damos cuenta de que no conviene reemplazar en la pila S por aSb, pues la b de la entrada no podrá cancelarse contra la a de la pila. Entonces aplicamos la otra transición disponible, que es $((q,\varepsilon,S),(q,\varepsilon))$. La ejecución continúa:

Estado	Falta leer	Pila
...	...	...
q	bb	bb
q	b	b
q	ε	ε

con lo cual la palabra de entrada es aceptada. Resumiendo, en este ejemplo la regla para decidir sobre la transición a aplicar, basándose en la previsión del siguiente caracter a leer, fué esta: si el siguiente caracter es a, reemplazar en la pila S por aSb, y si es b, reemplazar S por ε. Esta regla puede ser representada mediante la siguiente tabla:

	a	b	ε
S	aSb	ε	

En esta tabla, las columnas (a partir de la segunda) se refieren al siguiente caracter a leer (la "previsión"), habiendo una columna marcada "ε" por si en vez de haber un caracter siguiente se encuentra el fin de la palabra. La primera columna contiene la variable que se va a reemplazar en la pila por lo que indique la celda correspondiente en la tabla. [6]

A un AP aumentado con su tabla de previsión se le llama "compilador LL" por las siglas en inglés "Left to right Leftmost derivation", porque efectivamente dentro de la pila se lleva a cabo una derivación izquierda; el lector puede comprobar esto en el ejemplo anterior. A un compilador LL que considera una previsión de un caracter, como lo que hemos visto, se le llama "LL(1)"; en general, un compilador de tipo LL que toma en cuenta una previsión de k caracteres es LL(k).

La razón por la que es necesario a veces hacer una previsión de más de un caracter es porque para ciertas gramáticas no es suficiente una predicción de un solo caracter. Considérese, por ejemplo, la gramática con reglas $S \to aSb$, $S \to ab$, que también genera el lenguaje $\{a^n b^n\}$. Hacemos el inicio de la ejecución del AP correspondiente:

Estado	Falta leer	Pila
p	$aabb$	ε
q	$aabb$	S

En este punto, reemplazando S por aSb o por ab de todos modos se produce la a de la previsión, por lo que dicha predicción no establece ninguna diferencia entre las transiciones $((q, \varepsilon, S), (q, aSb))$ y $((q, \varepsilon, S), (q, ab))$. Este ejemplo en particular puede sacarse adelante haciendo una transformación de la gramática, conocida como "factorización izquierda", que consiste en añadir a la gramática una variable nueva (sea por ejemplo A), que produce "lo que sigue después del caracter común", en este caso a. Así, la gramática queda como (sus reglas): $S \to aA$, $A \to Sb$, $A \to b$. Con esta gramática ya es posible decidir entre las distintas transiciones considerando una previsión de un solo caracter, como se aprecia en la siguiente ejecución del AP correspondiente:

[6] Ejercicio: hacer nuevamente la traza de ejecución para la palabra abb, utilizando la tabla de previsión.

Estado	Falta leer	Pila
p	$aabb$	ε
q	$aabb$	S
q	$aabb$	aA
q	abb	A [7]
q	abb	Sb
q	abb	aAb
q	bb	Ab [8]
q	bb	bb
q	b	b
q	ε	ε

La tabla de previsión entonces debe haber sido:

	a	b	ε
S	aA	aA	
A	Sb	b	

Ahora veremos de una manera más sistemática cómo construir la tabla de previsión. Supongamos una GLC sin producciones vacías –lo cual prácticamente no representa una pérdida de generalidad. Necesitamos hacer las siguientes definiciones:

Definición.- El operador $first : \Sigma^* \rightarrow 2^\Sigma$ obtiene todos los caracteres con los que empieza una cadena derivable a partir de su argumento. Por ejemplo, para la GLC con reglas $S \rightarrow aA$, $A \rightarrow Sb$, $A \rightarrow b$, tenemos que $first(S) = \{a\}$, o bien $first(bab) = \{b\}$.

$first(\alpha)$ se calcula sistemáticamente a partir de las siguientes propiedades:

- Si $\alpha = au$, $a \in \Sigma$, $first(\alpha) = \{a\}$
- Si $\alpha = uv$, $u, v \in \Sigma^*$, $first(\alpha) = first(u)$
- Si $A \rightarrow \alpha \ \in \ R$, $first(\alpha) \subseteq first(A)$

Ejemplos: $first(aA) = \{a\}$, por la primera regla; $first(Sb) = first(S)$ por la segunda; a su vez, $first(S) = first(aA)$ por la tercera regla [9], de donde $first(S) = \{a\}$. Similarmente, $first(A) = first(Sb) \cup first(b) = first(S) \cup \{b\} = \{a\} \cup \{b\} = \{a, b\}$.

Ahora estamos en condiciones de dar un procedimiento para construir la tabla de previsión: supongamos que estamos tratando de llenar una celda de la tabla donde el renglón corresponde a la variabla X y la columna a la constante σ. Si hay en la gramática una regla $X \rightarrow \alpha$ donde $\sigma \in first(\alpha)$, el lado derecho α se pone en dicha celda:

[9]Pusimos "=" en vez de "$\subseteq$" porque sólo hay una regla con S del lado izquierdo.

...	σ	...	ε
...	...	...	...
X	α	...	...

Por ejemplo, con este procedimiento se obtiene la siguiente tabla de previsión para la gramática con reglas $S \to aA$, $A \to Sb$, $A \to b$:

	a	b	ε
S	aA		
A	Sb	b	

Esta tabla es casi idéntica a la que habíamos supuesto anteriormente para la misma gramática.

Puede ocurrir que en una celda de la tabla de previsión queden los lados derechos de varias reglas; esto es, si la celda corresponde la columna de la constante σ y al renglón de la variable X, y hay dos reglas distintas $X \to \alpha$ y $X \to \beta$, donde $\sigma \in \alpha$ y $\sigma \in \beta$, entonces tanto α como $beta$ tendrían derecho a estar en esa celda de la tabla. Cuando esto ocurre, simplemente se concluye que la tabla no se puede construir y que la gramática no es del tipo LL(1).

7.6 Compilación LR(1)

Como se puede apreciar en toda la sección precedente, los compiladores de tipo LL son esencialmente "predictores" que tratan de llevar a cabo la derivación en la pila, siempre reemplazando las variables por lo que éstas deban producir. Pero aún en gramáticas bastante sencillas, se vuelve demasiado difícil adivinar, aún con la ayuda de la previsión, qué regla de reemplazo hay que aplicar a una variable en el tope de la pila. Por esto, se han propuesto otros compiladores, llamados LR ("Left to right Rightmost derivation"), que no tratan de adivinar una derivación, sino que tratan de ir "reconociendo" grupos de caracteres que son reemplazados en la pila por una variable, hasta llegar eventualmente al símbolo inicial. Entonces, los compiladores LR funcionan al revés que los LL: es como si recorrieran el árbol de derivación *de abajo hacia arriba*, por lo que se le llama también compilación *ascendente*.

La construcción de un compilador LR a partir de una GLC se basa en otro procedimiento, distinto al visto anteriormente, para obtener un AP partiendo de una GLC. El AP que se construye con este método funciona de la manera siguiente:

- Se introducen terminales a la pila, mediante transiciones de la forma $((p, \sigma, \varepsilon), (p, \sigma))$. A esta acción se le llama "desplazar".

- Cuando el contenido del tope de la pila coincide con el lado derecho w de una regla $A \to w$, se reemplaza en la pila w por A, mediante una transición de la forma $((p, \varepsilon, w), (p, A))$. A esta acción se le llama "reducir".

La dificultad que se presenta en la práctica es que muchas veces es posible tanto desplazar como reducir. Aún peor, cuando se reduce puede ser posible hacerlo de varias formas distintas. Vamos a ilustrar esta situación con un ejemplo, que es el mismo que hemos visto anteriormente, esto es, la gramática para el lenguaje $\{a^n b^n\}$ con las reglas $S \to aA$, $A \to Sb$, $A \to b$. Dada la palabra $aabb$, el AP correspondiente tendría una traza de ejecución como sigue:

Falta leer	Pila	Acción
$aabb$	ε	Desplazar
aab	a	Desplazar
bb	aa	Desplazar
b	baa	Reducir por $A \to b$
b	Aaa	Reducir por $S \to aA$
b	Sa	Desplazar
ε	bSa	Reducir por $A \to Sb$
ε	Aa	Reducir por $S \to aA$
ε	S	Exito

En este ejemplo en particular es muy fácil discernir cuándo hacer cada una de las acciones, si damos preferencia a "reducir" sobre "desplazar". En efecto, nunca hubo conflicto respecto a cómo reducir, pues en cada momento sólo hubo una reducción aplicable. Sin embargo, en otros ejemplos es mucho más difícil determinar qué acción llevar a cabo, y existe un procedimiento para construir una tabla de previsión para compiladores LR(1), que puede ser consultado en la referencia [1].

Ahora formalizaremos el procedimiento para construir el AP de tipo LR a partir de una GLC (V, Σ, R, S):

- Hay 4 estados: i (inicial), f (final), p y q.
- $((i, \varepsilon, \varepsilon), (p, \#)) \in \Delta$. Esta transición coloca un "marcador" # en el fondo de la pila, para luego reconocer cuando la pila se ha vaciado.
- $((p, \sigma, \varepsilon), (p, \sigma)) \in \Delta$ para cada $\sigma \in \Sigma$. Estas transiciones permiten hacer la acción de desplazar.
- $((p, \varepsilon, \alpha), (p, A)) \in \Delta$ para cada regla $A \to \alpha \in R$. Estas transiciones efectúan las reducciones.
- $((p, \varepsilon, S), (q, \varepsilon)) \in \Delta$; esta transición reconoce cuando se llegó al símbolo inicial.

- $((q, \varepsilon, \#), (f, \varepsilon)) \in \Delta$; esta transición se asegura de que se haya vaciado la pila antes de aceptar la palabra.

Este procedimiento es directo y dejamos los ejemplos como ejercicios (ver sección de ejercicios).

7.7 Ejercicios

1. Sea un autómata pushdown $M = (K, \Sigma, \Gamma, \Delta, s, F)$ que acepta el lenguaje de paréntesis bien formados, incluyendo los paréntesis redondos "(", ")", así como los paréntesis cuadrados "[", "]", es decir: $L(M) = \{e, (), [], ()[], [](), (()), ([]), [()], [[]], \ldots\}$.
   (a) Proponer K, Σ, Γ, Δ, s y F
   (b) Dar el cálculo producido por la palabra "([]]".
2. Proponga una gramática libre de contexto o autómatas pushdown para el lenguaje: $\{a^i b^j c^k \mid i = j - k\}$
3. Considere una variante de los autómatas pushdown, que podríamos llamar "autómatas de fila", en los que en vez del stack, que se accesa en orden LIFO, se tiene una fila que se accesa en orden FIFO.
   (a) Dé una definición formal de los autómatas de fila.
   (b) Pruebe que el lenguaje $\{a^n b^n\}$ es aceptado / no es aceptado por algún autómata de fila.
   (c) Encuentre un lenguaje libre de contexto que no sea aceptado por ningún autómata de fila.
   (d) ¿Existe, para todo lenguaje aceptado por un Autómata de fila un AP que acepta el mismo lenguaje? (En otras palabras, ¿son equivalentes?) Pruebe su respuesta.
4. Considere una variante de los autómatas pushdown, los AP por estado final (APEF), en los que para aceptar una palabra basta con que al final de ésta el autómata se encuentre en un estado final, sin necesidad de que la pila esté vacía.
   (a) Dé una definición formal de los APEF, incluyendo la definición de lenguaje aceptado.
   (b) Proponga un APEF que acepte el lenguaje $\{a^n b^n\}$.
5. Considere el lenguaje $\{a^n b^m c^p d^q \mid n + m = p + q\}$
   (a) Proponga un AP que lo acepte.
   (b) Suponga la siguiente GLC (sus reglas) que genera dicho lenguaje:

i. $<AD> \rightarrow\ a <AD> d$
ii. $<AD> \rightarrow\ b <BD> d$
iii. $<BD> \rightarrow\ b <BD> d$
iv. $<BD> \rightarrow\ b <BC> c$
v. $<BC> \rightarrow\ b <BC> c$
vi. $<BC> \rightarrow\ e$
vii. $<AD> \rightarrow\ a <AC> c$
viii. $<AC> \rightarrow\ a <AC> c$
ix. $<AC> \rightarrow\ b <BC> c$

El símbolo inicial es $<AD>$. Pruebe la corrección de la GLC por inducción sobre la longitud de la derivación.

6. La siguiente gramática genera el lenguaje en $\{a, b\}^*$ en que las palabras tienen la misma cantidad de a's que de b's.

   (a) $S \rightarrow aSb$
   (b) $S \rightarrow bSa$
   (c) $S \rightarrow SS$
   (d) $S \rightarrow e$

   (a) Hacer una derivación izquierda de la palabra "*babbabaa*" usando esta gramática.
   (b) Probar la corrección de esta gramática. Ponga cuidado en formar el enunciado generalizado (que se aplica a las palabras con variables), en enunciar la hipótesis de inducción, y en aplicar el enunciado generalizado para el último paso de la derivación.
   (c) Transforme esta gramática a la forma normal de Chomsky. Para esto, elimine las producciones vacías, las producciones "inútiles", etc.
   (d) Proponga un autómata de pila que acepte el lenguaje del problema.

7. Proponga máquinas lo memos poderosas que sea posible para que acepten los siguientes lenguajes:

   (a) $\{(), [], <>, ([]), [< ()(<>) > ()], \ldots\}$
   (b) $\{(), (()), ((())), (((()))), \ldots\}$
   (c) $\{(), ()(), ()()(), \ldots\}$

8. Las máquinas reales tienen siempre límites a su capacidad de almacenamiento. Así, la pila infinita de los autómatas de pila puede ser limitada a un cierto tamaño fijo. Suponga una variante de los AP, los APn, en que la pila tiene un tamaño fijo n.

   (a) Proponga una definición de APn y de palabra aceptada por un APn.

(b) Pruebe (constructivamente) que los APn son equivalentes a los AF. (Hint: se puede asociar a cada par $(q, \sigma_1\sigma_2 \ldots \sigma_n)$, donde q es un estado del APn y $\sigma_1\sigma_2 \ldots \sigma_n$ es el contenido de la pila, un estado del AF).

(c) Pruebe su método con el APn de pila de tamaño 2 (caben dos caracteres), con relación de transición como sigue: $\Delta = \{((q_0, a, e), (q_0, a)), ((q_0, b, a), (q_1, e)), (q_1, b, a), (q_1, e))\}$, donde q_0 es inicial y q_1 es final.

9. Considere el lenguaje $L = \{a^n b^{n+m} c^m\}$

   (a) Proponga una GLC que genere L

   (b) Elimine de la gramática las producciones vacías y las inútiles, si las hay

   (c) Pruebe por inducción que la gramática es correcta

   (d) Obtenga el AP correspondiente, del tipo LL

   (e) Obtenga la tabla de previsión LL(1), calculando primero el "$first$" de cada variable de la gramática

   (f) Obtenga un AP de tipo LR para la gramática, si esto es posible, escribiendo primero una traza que indique cómo debería comportarse el LR para aceptar la palabra $abbbcc$.

Parte III

Máquinas de Turing y sus lenguajes

Capítulo 8

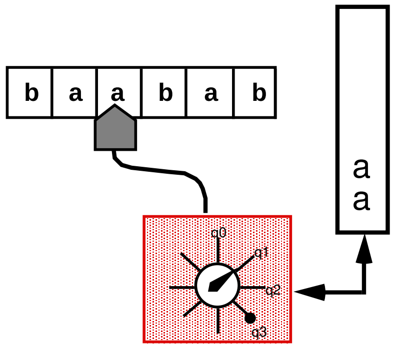
Máquinas de Turing

Así como en secciones anteriores vimos cómo al añadir al autómata finito básico una pila de almacenamiento auxiliar, aumentando con ello su poder de cálculo, cabría ahora preguntarnos qué es lo que habría que añadir a un autómata de pila para que pudiera analizar lenguajes como $\{a^n b^n c^n\}$. Partiendo del AP básico (figura 8.1(a)), algunas ideas podrían ser:

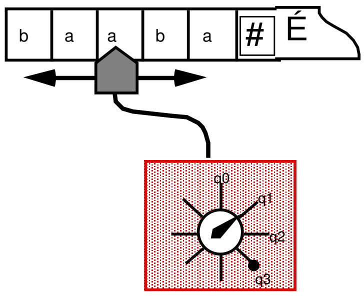
1. Añadir aún otra pila adicional;
2. Poner varias cabezas lectoras de la entrada;
3. Permitir la escritura en la cinta, además de la lectura de caracteres.

Aunque estas ideas –y otras aún más fantasiosas– pueden ser interesantes, vamos a enfocar nuestra atención a una propuesta en particular que ha tenido un gran impacto en el desarrollo teórico de la computación: la *máquina de Turing*.

En los años 30, un grupo de matemáticos, entre los que estaban A. Church, E. Post y A. Turing, buscaban una formalización adecuada de nociones tales como algoritmo, función calculable, problema soluble, y otras. Se generaron entonces varias propuestas, entre las que destacan el cálculo lambda de Church y la máquina de Turing. El cálculo lambda, aunque falló en su pretensión de servir de base a todas las matemáticas, propuso una notación y un modelo de funciones calculables que es muy usado hoy en día. Turing, por su parte, propuso un modelo de máquina abstracta, como una extensión de los autómatas finitos, que resultó ser de una gran simplicidad y poderío a la vez. Post propuso aún otro modelo de máquina abstracta, basada en la idea de un diagrama de flujo. Luego, Church hizo la conjetura de que los tres tipos de máquinas eran equivalentes en poder de cálculo, y más aún, afirmó que la máquina de Turing era el modelo de máquina abstracta más poderoso que puede haber. Esta conjetura, llamada "tesis de Church", no ha podido ser probada ni refutada hasta nuestros días.



(a) Autómata de pila



(b) Máquina de Turing

Figura 8.1:

La máquina de Turing (MT, ver figura 8.1(b)) tienen, como los autómatas que hemos visto antes, un control finito, una cabeza lectora y una cinta donde puede haber caracteres, y donde eventualmente viene la palabra de entrada. La cinta es infinita hacia la derecha, llenándose con el caracter blanco (que representaremos con "#").

En la MT la cabeza lectora es de lectura y escritura, por lo que la cinta puede ser modificada en curso de ejecución. Además, en la MT la cabeza se mueve bidireccionalmente (izquierda y derecha), por lo que puede pasar repetidas veces sobre un mismo segmento de la cinta.

La operación de la MT consta de los siguientes pasos:

1. Lee caracter en la cinta
2. Efectúa una transición de estado
3. Realiza una acción en la cinta

Las acciones que puede ejecutar en la cinta la MT pueden ser:

- Escribe un símbolo en la cinta, o
- Mueve la cabeza a la izquierda o a la derecha

Estas dos acciones son exclusivas, es decir, se hace una o la otra, pero no ambas a la vez.

La palabra de entrada en la MT está escrita inicialmente en la cinta, como es habitual en nuestros autómatas. Como la cinta es infinita hacia la derecha, inicialmente toda la parte de la cinta a la derecha de la palabra de entrada está llena del caracter blanco (#).

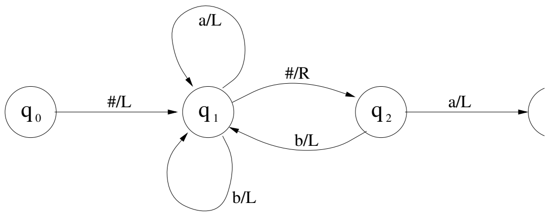


Figura 8.2: MT que acepta palabras que empiezan con a

Al iniciar la operación de la MT, la cabeza lectora está posicionada en el blanco siguiente a la derecha de la palabra de entrada, como en la figura 8.1(b).

Decimos que en la MT se llega al "final de un cálculo" cuando se alcanza un estado especial llamado *halt* en el control finito, como resultado de una transición. [1] Al llegar al *halt*, se detiene la operación de la MT, y *se acepta* la palabra de entrada. Así, en la MT no hay estados finales. En cierto sentido el *halt* sería entonces el único estado final, sólo que además detiene la ejecución. Así, como no hay estados de "rechazo", si no queremos que la palabra sea aceptada tenemos que hacer que la MT caiga en un ciclo infinito.[2]

Al diseñar una MT que acepte un cierto lenguaje, en realidad diseñamos el autómata finito que controla la cabeza y la cinta, el cual es un autómata con salida (de Mealy o de Moore, ver sección 3.7). Así, podemos usar la notación gráfica utilizada para aquellos autómatas para indicar su funcionamiento. En particular, cuando trazamos una flecha que va de un estado p a un estado q con etiqueta σ/L, quiere decir que cuando la entrada al control finito (esto es, el caracter leído por la cabeza de la MT) es σ, la cabeza lectora hace un movimiento a la izquierda, indicada por el caracter L (left, en inglés); similarmente cuando se tiene una flecha con σ/R el movimiento es a la derecha. Cuando la flecha tiene la etiqueta σ/ξ, donde ξ es un caracter, entonces la acción al recibir el caracter σ consiste en escribir el caracter ξ en la cinta. Con estos recursos es suficiente para diseñar algunas MT, como en el siguiente ejemplo.

Ejemplo.- Diseñar (el control finito de) una MT que acepte las palabras en $\{a, b\}$ que comiencen con a. La solución se muestra en la figura 8.2. En efecto, dado que la cabeza lectora se encuentra en el cuadro blanco a la derecha de la palabra de entrada, lo primero que hay que hacer es ir a la izquierda al leer un blanco es irse a la izquierda. Luego hay que realizar un ciclo en el que se desplaza la cabeza a la izquierda hasta encontrar el blanco, tras lo cual el caracter a la derecha debe ser el primero. Si es una "a", la palabra se acepta, y en caso contrario se hace que la MT caiga en un ciclo infinito.

Nótese que la acción inmediatamente antes de caer en el "halt" es irrelevante; igual se podía haber puesto "a/a" o "a/R" como etiqueta de la flecha.

[1] Estrictamente, *halt* no es un estado, como veremos luego.
[2] Luego veremos otra forma de no aceptar una palabra ("colgar" la configuración).

8.1 Formalización de la MT

Habiendo en la sección precedente hecho un recuento intuitivo de las características fundamentales de la MT, ahora procedemos a su formalización, esto es, a su modelización matemática en términos de la teoría de conjuntos.

Una MT es un cuádruplo (K, Σ, δ, s) donde:

- K es un conjunto de estados;
- Σ es un alfabeto, donde $\# \in \Sigma$;
- $s \in K$ es el estado inicial;
- $\delta : (K \times \Sigma) \rightarrow (K \cup \{h\}) \times (\Sigma \cup \{L, R\})$ es la función de transición.

La expresión de la función de transicíón parece algo complicada, pero puede entenderse de la siguiente manera: la función de transición del control finito debe considerar como entradas el estado anterior, que es un elemento de K, así como el caracter leído en la cinta, que es elemento de Σ. Por eso a la izquierda de la flecha aparece la expresión $\delta : (K \times \Sigma)$. Luego, el resultado de la función de transición debe incluir el siguiente estado, que es elemento de K. Sin embargo, hay también la posibilidad de que el control finito caiga en *halt*, representado por el símbolo h, y $h \notin K$ (h no es realmente un estado sino un "seudoestado"). Entonces en vez de tener un estado siguiente en K, tenemos un elemento de $K \cup \{h\}$. Otro resultado de la función de transición es la *acción* a ejecutar por la MT, que puede ser una escritura o un movimiento a la izquierda o a la derecha. La acción "mover cabeza a la izquierda" se representa por el símbolo L, y similarmente R para la derecha. En el caso de la escritura, en vez de usar un símbolo o comando especial, simplemente se indica el caracter que se escribe, el cual es un elemento de Σ. Desde luego, para que no haya confusión se requiere que ni L ni R estén en Σ. Resumiendo, el resultado de la función de transición debe ser un elemento de $(K \cup \{h\}) \times (\Sigma \cup \{L, R\})$.

Así, si $\delta(q, a) = (p, b)$, esto quiere decir que estando la MT en el estado q con la cabeza lectora sobre un caracter a, la función de transición enviará al autómata a un estado p, y adicionalmente hará una acción b; si $b \in \Sigma$, la acción será de escritura. Similarmente si $\delta(q, a) = (p, L)$, la cabeza de la MT hará un movimiento a la izquierda además de la transición de estado.

Por ejemplo, sea la MT siguiente: $K = \{s\}$, (sólo está el estado inicial), $\Sigma = \{a, \#\}$, $\delta(s, a) = (s, R)$, $\delta(s, \#) = (s, h)$. Puede verse por la función de transición que esta MT ejecuta un ciclo repetitivo en que mueve la cabeza hacia la derecha en tanto siga leyendo un caracter a, y se detiene (hace *halt*) en cuanto llega a un blanco.

Configuración

Como en otros autómatas que hemos visto en secciones anteriores, en las MT la *configuración* resume la situación en que se encuentra la MT en cualquier punto intermedio de un cálculo, de manera tal que con sólo las informaciones contenidas en la configuración podamos reconstruir dicha situación y continuar el cálculo.

Las informaciones necesarias para resumir la situación de una MT en medio de un cálculo son:

- Estado en que se encuentra la MT
- Contenido de la cinta
- Posición de la cabeza

Ahora el problema es cómo representar formalmente cada uno de los tres componentes de la configuración, tratando de hacerlo en la forma más similar posible a como representamos la configuración para otros tipos de autómatas.

No hay problema con el estado en que se encuentra la MT, que es directamente un elemento de K. Respecto al contenido de la cinta, existe la dificultad de que como es infinita hacia la derecha, no podemos representarla toda por una cadena de caracteres, que siempre será de tamaño finito. Vamos a tomar la solución de representar el contenido de la cinta por *la cadena de caracteres desde el inicio izquierdo de la cinta hasta la cabeza lectora o hasta el último caracter no blanco antes del inicio de una infinidad de blancos a la derecha.*

El contenido de la cinta sería un elemento de Σ^*, pero podemos precisar aún más sus características. En particular, hay que indicar de alguna manera que el último caracter no puede ser un blanco. Esto lo haremos concatenando un caracter no-blanco a la derecha de una cadena cualquiera; esto es, el contenido de la cinta sería un elemento de $\Sigma^*(\Sigma - \{\#\})$.

Ahora tratemos de representar la posición de la cabeza lectora. No funciona la solución que habíamos adoptado para los AF y AP, en que representábamos de una vez el contenido de la cinta y la posición de la cabeza limitándose a representar con una cadena lo que falta por leer de la palabra –esto es, tirando a la basura la parte a la izquierda de la cabeza lectora–, pues en el caso de las MT hay movimiento de la cabeza a la izquierda, por lo que los caracteres a la izquierda de la cabeza podrían eventualmente ser leidos. Otra solución sería representar la posición simplemente por un número natural. Sin embargo, adoptaremos la solución consistente en dividir la cinta en tres pedazos: uno a la izquierda de la cabeza, otro de un sólo caracter en la posición de la cabeza, y otro a la derecha de la cabeza, hasta donde comience la sucesión infinita de blancos. Así quedan representados simultáneamente el contenido de la cinta y la posición de la cabeza. Entonces estos serían representados por un elemento de: $\Sigma^* \times \Sigma \times \Sigma^*(\Sigma - \{\#\})$. Una última dificultad técnica es que si a la derecha de

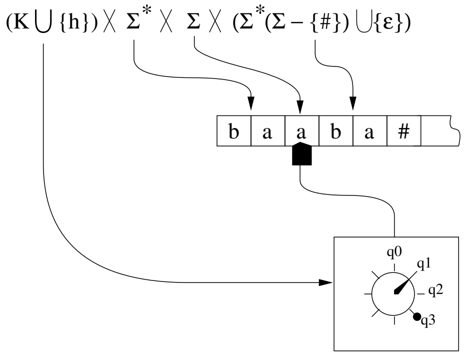


Figura 8.3: Configuración en MT

la cabeza hay únicamente blancos, entonces la parte de la cinta a la derecha sería la palabra vacía ε; desafortunadamente $\varepsilon \notin \Sigma^*(\Sigma - \{\#\})$. [3] Una solución sencilla a esto consiste en simplemente añadir ε a la expresión precedente, quedando $\{\varepsilon\} \cup \Sigma^*(\Sigma - \{\#\})$. Finalmente, la configuración es un elemento de:

$$(K \cup \{h\} \times \Sigma^* \times \Sigma \times (\Sigma^*(\Sigma - \{\#\}) \cup \{\varepsilon\})$$

(Ver figura 8.3)

Como en los AF y los AP, en las MT vamos a indicar las configuraciones encerradas entre dobles paréntesis, como en $[[q, aa, a, bb]]$, que indica que la MT en cuestión se encuentra en el estado q, habiendo a la izquierda de la cabeza una cadena "aa", bajo la cabeza una "a", y a su derecha –antes de la secuencia infinita de blancos– una cadena "bb". Para simplificar aún más la notación, podemos indicar por un caracter subrayado la posición de la cabeza lectora; así en vez de tener cuatro componentes la configuración tendrá únicamente dos, como por ejemplo en $[[q, aa\underline{a}bb]]$, que es equivalente al ejemplo que acabamos de dar.

8.1.1 Relación entre configuraciones

Vamos a definir una relación binaria "$C_1 \vdash C_2$" que nos indica que la MT puede pasar de la configuración C_1 a la configuración C_2.

Definición.- La relación $\vdash$ en $C \times C$ –donde C es el conjunto de configuraciones– se define por casos, de la siguiente manera:

Caso escritura:

$$[[p, w, a, u]] \vdash [[q, w, b, u]]$$

ssi $\delta(p, a) = (q, b), b \in \Sigma$

[3] ¿Porqué?

Caso de movimiento a la izquierda, parte derecha no vacía:

$$[[p, wd, a, u]] \vdash [[q, w, d, au]]$$

ssi $\delta(p, a) = (q, L)$, $a \neq \#$ o $u \neq \varepsilon$

Caso de movimiento a la izquierda, parte derecha vacía:

$$[[p, wd, \#, \varepsilon]]| - [[q, w, d, \varepsilon]]$$

ssi $\delta(p, \#) = (q, L)$

Caso de movimiento a la derecha, parte derecha no vacía:

$$[[p, w, a, du]] \vdash [[q, wa, d, u]]$$

ssi $\delta(p, a) = (q, R)$

Caso de movimiento a la derecha, parte derecha vacía:

$$[[p, w, a, \varepsilon]] \vdash [[q, wa, \#, \varepsilon]]$$

ssi $\delta(p, a) = (q, R)$

Ejemplos:

$$\begin{array}{ll}
\text{Si } \delta(q_1, a) = (q_2, b), & [[q_1, bb\underline{a}]] \vdash [[q_2, bb\underline{b}]] \\
\text{Si } \delta(q_1, a) = (q_2, R), & [[q_1, b\underline{a}b]] \vdash [[q_2, ba\underline{b}]] \\
 & [[q_1, ba\underline{b}]] \vdash [[q_2, bab\underline{\#}]] \\
\text{Si } \delta(q_1, a) = (q_2, L), & [[q_1, aaba\underline{b}]] \vdash [[q_2, aab\underline{a}b]] \\
 & [[q_1, a\underline{b}b]] \vdash [[q_2, \underline{a}bb]] \\
 & [[q_1, ab\#\underline{\#}]] \vdash [[q_2, ab\underline{\#}]]
\end{array}$$

8.1.2 Configuración "colgada"

Supóngase una MT en la situación que se ilustra en la figura 8.4, es decir, con la cabeza lectora en el último cuadro a la izquierda de la cinta. Si en dicha situación aún se tiene una acción de mover la cabeza a la izquierda, esta transición no nos llevará a una configuración válida, pues "la cabeza se sale de la cinta", y el cálculo no puede continuar. Decimos que se llega a una *configuración colgada*, que no es una configuración en sentido estricto. Formalmente, se llega a una configuración colgada si tenemos una configuración $[[q, \varepsilon, a, u]]$, y ocurre una transición $\delta(q, a) = (p, L)$.

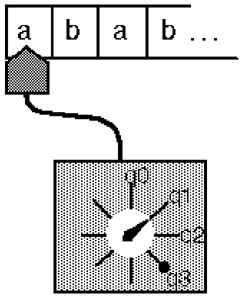


Figura 8.4: Configuración colgada

8.1.3 Cálculos en MT

Igual que en otros tipos de autómatas que hemos visto anteriormente, en las MT un cálculo es una secuencia $C_1, C_2, \ldots, C_n$ de configuraciones tal que $C_i \vdash C_{i+1}$. Un cálculo puede ser visto en términos computacionales como una "traza de ejecución", que nos describe de una manera muy exacta la forma en que una MT responde ante una entrada en particular. Por ejemplo, sea la MT siguiente (dada ya como ejemplo anteriormente): $K = \{s\}$, $\Sigma = \{a, \#\}$, $\delta(s, a) = (s, R)$, $\delta(s, \#) = (s, h)$. Ante la configuración $[[s, a, a, aa]]$ se presenta el cálculo siguiente:

$$[[s, a\underline{a}aa]] \vdash [[s, aa\underline{a}a]] \vdash [[s, aaa\underline{a}]] \vdash [[s, aaaa\underline{\#}]] \vdash [[h, aaaa\underline{\#}]]$$

Para $i > j$, se puede llegar de una configuración C_i a C_j en uno o varios pasos; esto se indica en forma compacta utilizando la cerradura reflexiva y transitiva de la relación $\vdash$, denotada por $\vdash^*$, quedando $C_i \vdash^* C_j$.

8.1.4 Palabra aceptada

Con las definiciones dadas ahora estamos en condiciones de definir las nociones de palabra aceptada y lenguaje aceptado:

Definición.- Una palabra w es *aceptada* por una MT M si

$$[[s, \#w, \#, \varepsilon]] \vdash^* [[h, \alpha, a, \beta]]$$

Como se vé, el único criterio para que la palabra de entrada w se acepte es que se llegue a *halt* en algún momento, independientemente del contenido final de la cinta, el cual es visto como "basura". Por ejemplo, la MT del último ejemplo acepta cualquier palabra de entrada.

Decimos de que un lenguaje L es *Turing-aceptable* si hay alguna MT que dá *halt* para toda entrada $w \in L$.

Ejemplo.- Probar que hay lenguajes Turing-aceptables que no son libres de contexto. Proponemos el lenguaje $a^n b^n c^n$, que se sabe que no es LLC. Ahora construiremos una MT que lo acepte. La estrategia para el funcionamiento de dicha MT consistirá en ir haciendo "pasadas" por la palabra, descontando en cada una de ellas una a, una b y una c; para descontar esos caracteres sin tener que recorrer el resto de la palabra simplemente los reeplazaremos por una caracter "$*$". Cuando ya no encontremos ninguna a, b o c en alguna pasada, si queda alguna de las otras dos letras la palabra no es aceptada; en caso contrario se llega a *halt*. Es útil, antes de emprender el diseño de una MT, tener una idea muy clara de cómo se quiere que funcione. Para eso se puede detallar el funcionamiento con algún ejemplo representativo, como en la tabla siguiente, para la palabra $aabbcc$. La posición de la cabeza se indica por el símbolo "$\triangle$".

#	a	a	b	b	c	c	#
							$\triangle$
#	a	a	b	b	c	c	#
						$\triangle$	
	...		...		...		
#	a	a	b	b	c	c	#
$\triangle$							
#	a	a	b	b	c	c	#
	$\triangle$						
#	$*$	a	b	b	c	c	#
	$\triangle$						
#	$*$	a	$*$	b	$*$	c	#
							$\triangle$
	...		...		...		

Obsérvese que, a excepción de los estados, que no se indican, acabamos de dar la serie de configuraciones por las que debe pasar la MT. Esto es de gran ayuda para establecer el diagrama de transiciones entre estados, que quedaría como se ilustra en la figura 8.5. Como se puede apreciar ahí, cuando la MT espera recibir a (estado q_2) y recibe b o c, se le "cicla" simplemente escribiendo la misma letra que se leyó; se hace algo similar para los otros casos de error.

8.1.5 MT para cálculos de funciones

Hasta el momento hemos visto las MT como analizadoras de palabras cuyo fin es determinar si la palabra de entrada pertenece o no al lenguaje aceptado. Sin embargo, las MT también pueden ser utilizadas para calcular resultados u operaciones a partir de la entrada. En vez de considerar como "basura" el contenido de la cinta al llegar al *halt*, podríamos verlo como

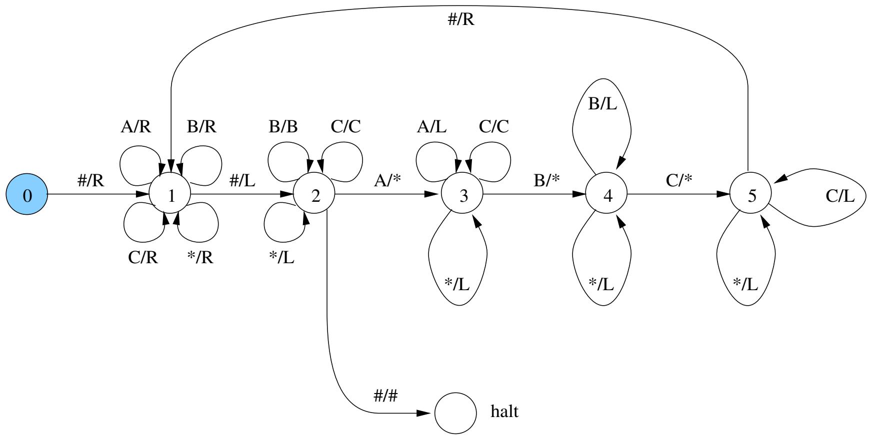


Figura 8.5: MT para el lenguaje $a^n b^n c^n$

un resultado calculado. Para poder interpretar sin ambigüedad el contenido final de la cinta como resultado, vamos a requerir que cumpla con un formato estricto: debe haber en el cuadro más izquierdo de la cinta un blanco, seguido de la palabra que se considera como el resultado en sí, seguido de una infinidad de blancos, sobre el primero de los cuales debe quedar posicionada la cabeza lectora.

Para precisar estas nociones, utilizamos la notación formal: Una MT calcula un resultado u a partir de una entrada w si:

$$[[s, \#w, \#, \varepsilon]] \vdash^* [[h, \#u, \#, \varepsilon]].$$

Como se sabe, las funciones en matemáticas sirven precisamente para describir la relación entre un resultado y una entrada. Podemos relacionar esta noción con la definición anterior de la manera siguiente: Una MT M calcula una función $f : \Sigma^* \rightarrow \Sigma^*$ si para toda entrada w, M calcula un resultado u tal que $f(w) = u$.

Si hay una MT que calcula una función f, decimos que f es *Turing-calculable*. Aunque parezca increíble, no todas la funciones son Turing-calculables. En el siguiente capítulo estudiaremos varias clases de funciones calculables.

Ejemplo.- Construir una máquina de Turing que reste dos numeros naturales en unario, esto es, $f(x, y) = x - y$. Desde luego, como las MT reciben un solo argumento, para realizar una función de dos argumentos como la resta en realidad se recibe un sólo argumento que contiene un símbolo para separar dos partes de la entrada. Por ejemplo, la resta de $5 - 3$ quedaría indicada por la cadena "$11111 - 111$", lo que sería el argumento de entrada; desde luego, el resultado en este caso sería la cadena "11". La cabeza lectora al final debe estar posicionada en el blanco a la derecha del residuo. En caso de que el sustraendo sea mayor que el minuendo, el resultado es cero. A esta forma de resta sin resultados negativos se le llama a veces "monus" en vez de "menos".

La estrategia para construir esta MT sería ir "descontando" cada 1 del minuendo contra otro 1 del sustraendo, reemplazando ambos por un caracter arbitrario –sea "$*$". Cuando se termine el sustraendo, se borran los caracteres inútiles de manera que queden sólo los restos del minuendo. Para evitar tener que recorrer el residuo, descontamos caracteres del minuendo de derecha a izquierda. Resumiendo, tendríamos una secuencia de configuraciones de la cinta como las siguientes (la última línea indica la configuración en la que debe dar *halt*).

$$
\begin{array}{cccccccc}
\# & 1 & 1 & 1 & - & 1 & 1 & \# \\
 & & & & & & & \triangle \\
\# & 1 & 1 & 1 & - & 1 & 1 & \# \\
 & & & & & & \triangle & \\
\# & 1 & 1 & 1 & - & 1 & * & \# \\
 & & & & & & \triangle & \\
 & \ldots & & \ldots & & \ldots & & \\
\# & 1 & 1 & * & - & 1 & * & \# \\
 & & & \triangle & & & & \\
 & \ldots & & \ldots & & \ldots & & \\
\# & 1 & * & * & - & * & * & \# \\
 & & \triangle & & & & & \\
 & \ldots & & \ldots & & \ldots & & \\
\# & 1 & \# & \# & \# & \# & \# & \# \\
 & & \triangle & & & & &
\end{array}
$$

Dejamos como ejercicio hacer el diagrama de estados del control finito de esta MT.

Recomendamos utilizar el paquete computacional "Turing's World", de John Barwise y Jon Etchemendy (Stanford University) para diseñar gráficamente (¡y ejecutar!) máquinas de Turing en un ambiente interactivo muy amigable.

8.1.6 Problemas de decisión

Un caso particular de funciones es aquél en que el resultado sólo puede ser *sí* o *no*. Si representamos el *sí* con Y y el *no* con N, estamos considerando funciones $g : \Sigma^* \to \{Y, N\}$. En este caso, la MT sirve para decidir si la entrada tiene una propiedad P o no la tiene.

Por ejemplo, si la propiedad P consiste en que la entrada es de longitud par, la MT correspondiente debe generar los cálculos siguientes:

$$[[s, \#w\#]] \vdash^* [[h, \#Y\#]]$$

si $|w|$ es par, y

$$[[s, \#w\#]] \vdash^* [[h, \#N\#]]$$

si $|w|$ es non.

Ejercicio.- Diseñar una MT que decida si la entrada es de longitud par.

Definición.- Decimos que un lenguaje L es *Turing-decidible* si hay alguna MT que entrega un resultado Y ssi la entrada w está en L.

Debe quedar claro que para que una MT entregue como resultado Y o N, es condición indispensable que la palabra de entrada haya sido aceptada. Esto tiene la consecuencia siguiente:

Proposición.- Un lenguaje es Turing-decidible sólamente si es Turing-aceptable.

Si un lenguaje no es Turing-decidible se dice que es *indecidible*. En la siguiente sección veremos lenguajes indecidibles.

8.2 Tesis de Church

A raíz de la propuesta por A. Turing de su modelo de máquina abstracta como formalización de la noción de algoritmo, hubo diversos intentos de encontrar otros modelos de máquinas u otros formalismos que fueran más poderosos que las MT, en el mismo sentido que las MT son más poderosas que los AF y los AP. [4] Sin embargo, todos los intentos fueron infructuosos, hasta el punto de que A. Church, a la sazón inventor del cálculo lambda –uno de los sistemas competidores de la MT–, propuso la conjetura de que *no puede existir una máquina abstracta más poderosa que la MT*, sin dar una prueba formal de ello. Hasta nuestros dias la llamada "tesis de Church" no ha podido ser probada ni refutada. Puede ser expresada en los términos –informales– siguientes: *Todo aquello que es algorítmicamente calculable, puede ser calculado por una MT básica*, donde "MT básica" se refiere a la MT que hemos estudiado hasta ahora, por comparación con otras MT "extendidas" que se propusieron en un intento de ganar más poder de cálculo.

Así como en los autómatas finitos agragar el no-determinismo no redundó en un aumento en el poder de cálculo, en las MT las extensiones de la MT básica resultaron ser equivalentes a la MT básica.

Podemos considerar comparaciones de la MT con:

1. Extensiones a la MT
   (a) MT con cinta infinita a la izquierda
   (b) MT con varias cintas, varias cabezas

[4] Ejercicio: formalice la afirmación de que las MT son más poderosas que los AP.

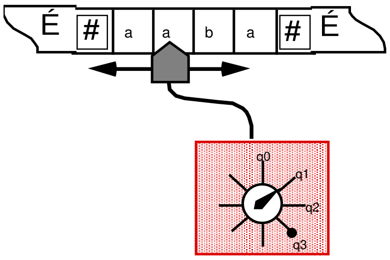


Figura 8.6: MTI

(c) MT con no determinismo

2. Otras máquinas de cinta

3. Otros paradigmas (Post, Gramáticas)

De todas estas posibilidades, sólo consideraremos en detalle la cinta infinita a la izquierda y las máquinas de Post. Las comparaciones restantes pueden ser encontradas en la referencia [Lewis, Papadimitriou].

8.2.1 MT con cinta infinita a la izquierda

Mientras que en la MT básica hay un límite a la cinta por el lado izquierdo, en la MT con cinta infinita a la izquierda (abreviado MTI) la cinta se extiende indefinidamente por el lado izquierdo (ver figura 8.6).

Una de las diferencias con la MT básica es que la MTI no se puede "colgar" al salirse la cabeza por el extremo izquierdo de la cinta. Entonces la única manera de no llegar al *halt* en la MTI es ciclándose.

La formalización de la MTI es simplemente una adaptación de la formalización que vimos para la MT. [5]

8.2.2 Equivalencia MT–MTI

Según la tesis de Church la MTI debe ser equivalente a la MT básica. Vamos a probar este hecho en dos partes:

1. MTI es tan poderosa como MT

[5]Ejercicio: adaptar las definiciones de configuración, relación $\vdash$ y palabra aceptada para las MTI.

2. MT es tan poderosa como MTI

Vamos a empezar con la primera parte. Formalmente, se quiere decir que para toda MT existe una MTI tal que:

1. Si MT acepta w, MTI también;
2. Si MT se cicla con w, MTI también;
3. Si MT calcula $f(w)$, MTI también

Las pruebas que vamos a considerar esta sección se basan en el principio de la *simulación*. Esta consiste informalmente en que la máquina simuladora actúa como lo haría la máquina simulada. Formalmente consiste en un mapeo μ que asocia a cada configuración de la máquina simuladora M_{ora} una configuración de la máquina simulada M_{ada}, y a cada acción de M_{ora} una acción de M_{ada}, de modo tal que se cumpla la correspondencia de los tres puntos señalados arriba.

La simulación de la MT por parte de la MTI es trivial, puesto que la parte izquierda de la cinta puede quedarse simplemente sin utilizar, ejecutando la MTI *exactamente las mismas acciones que la MT*, y la configuración de MTI siendo también la misma que la de MT, salvo que hay que expresar la parte de la cinta en forma ligeramente distinta. [6]

Ahora vamos a considerar la parte de la prueba en que la MTI puede ser simulada por una MT básica, que es desde luego la parte interesante.

Concretamente, lo que necesitamos probar es que, en caso de llegar la MTI al *halt*, tendremos la correspondencia siguiente:

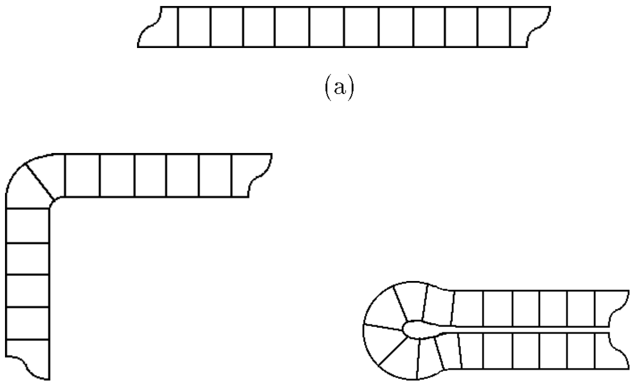
Si $[[s_{MTI}, w\underline{\#}]] \vdash^*_{MTI} [[h, u\underline{a}v]]$, entonces $[[s_{MT}, \#w\underline{\#}]] \vdash^*_{MT} [[h, \#u\underline{a}v]]$.

Con esto se garantiza el cumplimiento de las tres condiciones arriba enunciadas. Hay que probar además que si la MTI se cicla con w, también MT se cicla.

Lo que vamos a hacer para efectuar la simulación puede verse conceptualmente como "doblar" la cinta de la MTI por la mitad, de manera que va a quedar una cinta que sólo es infinita a la derecha; esto se ilustra en la secuencia de la figura 8.7.

Suponiendo que numeramos los cuadros de la cinta de la MTI, con un cuadro central 0, teniendo a la derecha los cuadros 1, 2, etc., y a la iaquierda los cuadros -1, -2, etc., la cinta ya doblada se vería como en la figura 8.8(a). Nótese que en el extremo izquierdo la cinta tiene un caracter "\$", que no tiene un correspondiente en la cinta original de la MTI, pero que nos va a servir en el curso de la simulación. El resto de la cinta parece estar hecho de

[6] Ejercicio: Exprese formalmente la correspondencia μ entre las configuraciones de MT y MTI.

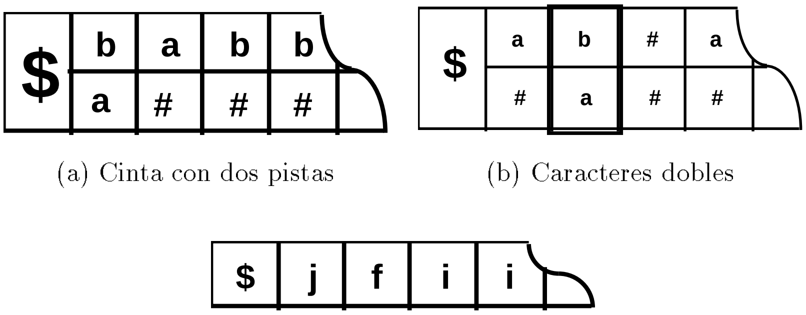


(a)

(b) (c)

Figura 8.7: Doblamiento de la cinta

dos niveles, uno superior y otro inferior, correspondiendo el primero a la parte derecha de la cinta de la MTI y el segundo a la parte izquierda de la cinta de la MTI.



(a) Cinta con dos pistas (b) Caracteres dobles

(c) Cinta codificada

Figura 8.8:

Aún cuando esta imagen conceptual de la cinta de la MT nos servirá para ilustrar intuitivamente las ideas que se ponene en juego en la simulación, está claro que las MT básicas no tienen nada como cintas divididas en parte inferior y parte superior. Por ello, vamos a explicar cómo se puede representar en una MT básica una cinta que correspondería a la imagen de la cinta "doblada".

Considérese una extensión del alfabeto Σ a un Σ^+, que contiene a Σ y además un conjunto de caracteres $\sigma_{\rho,\tau}$, cada uno de los cuales representa un cuadro "doble" de la cinta doblada, donde habría una ρ en la parte superior y una τ en la parte inferior.

Los caracteres $\sigma_{\rho,\tau}$ pueden ser considerados como una forma de codificación de los grupos de dos caracteres en la cinta doblada. Por ejemplo, en la figura 8.8(b) en el cuadro doble indicado por un marco, hay una b en la parte superior y una a en el inferior; esto puede representarse con un nuevo caracter $\sigma_{b,a}$ que puede ser por ejemplo c o cualquier otro caracter que no esté en el Σ original. Para completar el ejemplo, a continuación presentamos una codificación completa de las combinaciones de dos caracteres del alfabeto $\{a, b, \#\}$:

$$\begin{array}{lll} \sigma_{\#,\#} = c & \sigma_{\#,a} = d & \sigma_{\#,b} = e \\ \sigma_{a,\#} = f & \sigma_{a,a} = g & \sigma_{a,b} = h \\ \sigma_{b,\#} = i & \sigma_{b,a} = j & \sigma_{b,b} = k \end{array}$$

Esta codificación de caracteres, para la cinta "doble" de la figura 8.8(a) se muestra en la figura 8.8(c).

Ahora vamos a entrar en la simulación propiamente dicha de la MTI por la MT. Como habíamos mencionado antes, hay que establecer un mapeo μ de la cinta de la MTI a la de MT, y de las acciones de la MTI por acciones de la MT. Habiendo hecho ya el primer punto, con el "doblamiento" de la cinta de MTI y la consiguiente codificacion de caracteres, ahora queda pendiente el segundo punto, esto es, establecer la correspondencia de acciones de la MTI en la MT.

Del hecho de haber "doblado" la cinta de la MTI resulta que ahora la posición de la cabeza lectora resulta en cierta forma ambigua, pues al quedar "pegados" dos cuadros de la antigua cinta de la MTI, con la sóla posición de la cabeza en la MT simuladora no se sabe en cuál de ellos se encontraba la MTI simulada. Por ejemplo, en la figura 8.8(a) se aprecia que la posición de la cabeza en la MT simuladora en el tercer cuadro corresponde tanto a la posición 2 como a la -1. En vista de esta situación es necesario especificar además si la MT simuladora se encuentra en la "pista superior" o en la "pista inferior". Desde luego, una MT no tiene varias pistas; lo que ocurre es que en realidad va a tener un conjunto de estados K' cuyos estados $k_{q,i}$ representan que conceptualmente se estaría en el estado q y la pista i: si $i = 1$ entonces es la pista superior, y si $i = 2$ es la pista inferior. Claramente la MT simuladora tiene al menos el doble de estados que la MTI original.[7]

Ahora pasamos a establecer la equivalencia de las acciones de la MT simuladora con la MTI simulada. El problema en realidad se reduce a obtener las transiciones δ_{MT} en base a las transiciones originales δ_{MTI}. Hay que distinguir los casos en los que la MT se encuentra en la "pista superior" de cuando está en la "pista inferior".

Si la MT simula a la MTI en la "pista superior", se encuentra en un estado $k_{q,1}$. Entonces si en MTI tenemos una transición $\delta_{MTI}(q, a) = (p, L)$, en la MT tendremos una transición $\delta_{MT}(k_{q,1}, \sigma_{a,\rho}) = (k_{p,1}, L)$, y similarmente para la acción R. En el caso de la escritura, si $\delta_{MTI}(q, a) = (p, b)$, con $b \in \Sigma$, en la MT tendremos $\delta_{MT}(k_{q,1}, \sigma_{a,\rho}) = (k_{p,1}, \sigma_{b,\rho})$. [8]

[7] Ejercicio: justifique este hecho.

[8] Ejercicio: Explique con sus propias palabras este resultado.

Similarmente, si la MTI se encuentra en la "pista inferior", se encontrará en un estado $k_{q,2}$, y por lo tanto a una transición $\delta_{MTI}(q,a) = (p, L)$, en la MT corresponde una transición $\delta_{MT}(k_{q,2}, \sigma_{a,\rho}) = (k_{p,2}, R)$. Obsérvese que el movimiento a la izquierda en la pista inferior se traduce como un movimiento a la derecha. [9] Un cambio de sentido similar se observa en el caso del movimiento a la derecha. Para el caso de la escritura en la "pista inferior", si $\delta_{MTI}(q,a) = (p,b)$, con $b \in \Sigma$, en la MT tendremos $\delta_{MT}(k_{q,2}, \sigma_{\rho,a}) = (k_{p,2}, \sigma_{\rho,b})$.

Falta aún considerar el caso en que la MT simuladora llegara, estando en la pista superior, al extremos izquierdo de su cinta, y para simular que la MTI continúa moviéndose a la izquierda, debe hacer un "cambio de pistas", pasando a la "pista inferior" –en realidad de un estado $k_{q,1}$ a un estado $k_{q,2}$. En la MT simuladora la necesidad de "cambiar las pistas" se detecta por el hecho de que el caracter bajo la cabeza lectora es "\$". El "cambio de pista" se logra añadiendo las transiciones siguientes en δ_{MT}:

$$\delta_{MT}(k_{q,1}, \$) = (k_{q,2}, R)$$
$$\delta_{MT}(k_{q,2}, \$) = (k_{q,1}, R)$$

Aún un detalle técnico que no escapará al lector atento es que en la MT la parte de la cinta que se extiende hacia el infinito por la derecha está llena de caracteres "#", mientras que en las transiciones de la MT simuladora se requieren caracteres $\sigma_{a,b}$ que se entienden como a en la pista superior y b en la inferior. Nótese que no es lo mismo un caracter # que un caracter $\sigma_{\#,\#}$. En otras palabras, en la MT simuladora las pistas superior e inferior van a estar separadas *hasta un cierto punto de la cinta*, después del cual tendremos la cinta llena de caracteres "#". Por lo anterior, cuando la MT simuladora se encuentra con un caracter "#", se debe separar en el equivalente de dos caracteres, uno para cada pista, reemplazándolo por $\sigma_{\#,\#}$. En resumen, hay que añadir a la MT las transiciones siguientes:

$$\delta_{MT}(k_{q,1}, \#) = (k_{q,1}, \sigma_{\#,\#})$$
$$\delta_{MT}(k_{q,2}, \#) = (k_{q,2}, \sigma_{\#,\#})$$

Finalmente, el *halt* en la MT simuladora se producirá cuando se llegue a un estado $k_{h,1}$ o bien $k_{h,2}$ –que *no* son el *halt*, sino que son estados de la MT. Para llegar al verdadero *halt* se requieren transiciones adicionales, que son de las formas siguientes:

$$\delta_{MT}(k_{h,1}, \sigma_{a,b}) = (h, \sigma_{a,b})$$
$$\delta_{MT}(k_{h,2}, \sigma_{a,b}) = (h, \sigma_{a,b})$$

Con lo que hemos visto hasta el momento es suficiente para realizar la simulación de MTI por MT desde el punto de vista de la aceptación o rechazo de palabras, pero no desde el punto de vista de el cálculo de resultados. Respecto a esto, sería necesario restablecer al final de la simulación el formato de la cinta en MT, eliminando los caracteres "dobles" de la cinta, recorriendo el resultado, etc. Estos detalles pueden consultarse en la referencia [Lewis,Papadimitriou]. [10]

[9] ¿Porqué?

[10] Ejercicio: hacer una simulación completa de una MTI.

8.3 Máquinas de Post

Otra propuesta de máquina abstracta que compitió con la MT en ser la más poderosa expresión de lo "algorítmicamente calculable" fué la de las máquinas del americano Post. Conceptualmente las máquinas de Post tienen poca relación con el modelo básico de máquinas que hemos visto hasta el momento –básicamente derivaciones de los AF.

Las máquinas de Post (MP) están basadas en el concepto de *diagramas de flujo*, tan habituales en nuestros días por la enseñanza de la programación en lenguajes imperativos (C, Pascal, ensamblador, etc.). En un diagrama de flujo se va siguiendo las flechas que nos llevan de la ejecución de una *acción* a la siguiente; a este recorrido se le llama "flujo de control". Algunas acciones especiales son *condicionales*, en el sentido de que tienen varias flechas de salida, dependiendo la que uno tome del cumplimiento de cierta condición. [11]

Más específicamente, los diagramas de flujo de Post consideran unas acciones muy elementales cuyo efecto eventualmente es alterar el valor de una única variable x. La variable x es capaz de almacenar una cadena de caracteres arbitrariamente grande.

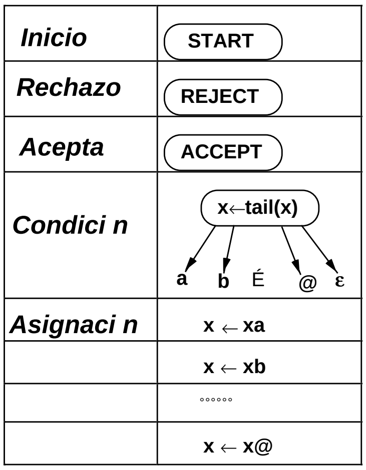


Figura 8.9: Acciones en MP

En la figura 8.9 presentamos un resumen de las acciones de la MP, las cuales son:

Inicio. La acción **START** indica el punto en que empieza a recorrerse el diagrama de flujo.

Rechazo. La acción **REJECT** indica que la palabra de entrada no es aceptada (es rechazada). Además termina la ejecución del diagrama.

Acepta. La acción **ACCEPT** indica que la palabra de entrada es aceptada. También termina la ejecución del diagrama.

[11] Pensamos que el lector está habituado a los diagramas de flujo, por lo que no abundaremos en ejemplos y explicaciones.

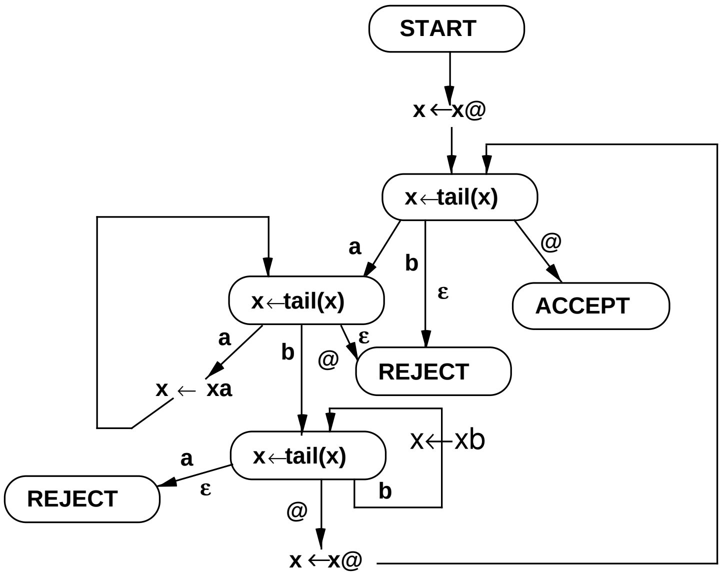


Figura 8.10: MP para $\{a^n b^n\}$

Condicional. La acción $x \leftarrow tail(x)$ tiene el efecto de quitar el primer caracter de la palabra almacenada en la variable x; la continuación del diagrama dependerá de cuál fué el caracter que se quitó a x, habiendo varias salidas de la condicional, indicadas con sendos símbolos, que corresponden al caracter que se quitó a la variable. En otras palabras, si la palabra de entrada es $\sigma_1, \sigma_2, \ldots, \sigma_n$, el camino que tomemos para seguir el diagrama será el indicado con un símbolo que concida con σ_1. Hay además una salida marcada con ε, para el caso de que la variable x contenga la palabra vacía (antes de tratar de quitarle el caracter).

Asignación. Las acciones de la forma $x \leftarrow xa$, donde $a \in \Sigma$, tienen el efecto de añadir a la variable x el caracter a por la derecha. Así, si $x = \alpha$ antes de la asignación, después de ella tendremos $x = \alpha a$. Hay una instrucción $x \leftarrow xa$ para cada caracter $a \in \Sigma$.

Ejemplo.- La MP de la figura 8.10 acepta el lenguaje $\{a^n b^n\}$. Dejamos como ejercicio al lector seguir el diagrama de flujo para comprobar que acepta la palabra $aabb$ y no la palabra abb, así como describir en palabras la estrategia que utiliza esta MP para determinar que el número de a es igual al número de b.

8.3.1 Formalización de las MP

Recordemos antes que nada que la formalización de una máquina abstracta reviste dos aspectos: uno es formalizar los componentes de una máquina en particular, esto es, las informaciones que hacen diferente a una máquina de las demás de su clase, [12] mientras que

[12]Este era el caso de las quíntuplas $(K, \Sigma, \delta, s, F)$ para los AF.

el otro aspecto es el de caracterizar el *funcionamiento* de las máquinas que tratamos de formalizar. En el primer aspecto, las MP podrían ser caracterizadas como *grafos*, donde los nodos serían las acciones, y los vértices serían las flechas del diagrama de Post. Esto es, una MP sería básicamente un conjunto de nodos N, clasificados de acuerdo con las acciones que tienen asociadas, así como una función de transición que determine cuál es el nodo siguiente en el diagrama. Así tendremos:

Definición.- Una MP es una tripleta (N, Σ, δ), donde:

- $N = N_A \cup N_T \cup \{START, ACCEPT, REJECT\}$, siendo N_A el conjunto de nodos de asignación y N_C el conjunto de nodos condicionales. En otras palabras, los nodos están clasificados según la acción que tienen asociada. Adicionalmente N_A está clasificado según la letra que se añade por la derecha, es decir, $N_A = N_{A\sigma_1} \cup N_{A\sigma_2} \cup \ldots N_{A\sigma_n}$
- Como de costumbre, Σ es el alfabeto, que no incluye el caracter @.
- δ es la función de transición que nos indica cuál es el siguiente nodo al que hay que ir:

$$\delta : N - \{ACCEPT, REJECT\} \times \Sigma \cup \{@, \varepsilon\} \to N - \{START\}$$

Como se vé, el nodo destino de δ depende del nodo anterior y de un caracter (el caracter suprimido, en el caso de la acción condicional –en todas las demás acciones el caracter es irrelevante y el destino debe ser el mismo para todo caracter).

Ejercicio.- Representar formalmente la MP de la figura 8.10.

Ahora trataremos de formalizar el funcionamiento de las MP. Como habitualmente, nos apoyaremos en la noción de configuración. En la configuración debemos resumir todas las informaciones que caracterizan completamente la situación en que se encuentra una MP a mitad de un cálculo. En la configuración de una MP vamos a considerar, evidentemente, el punto en que nos encontramos al recorrer el diagrama de flujo –lo que formalmente se representaría como un nodo $n \in N$. [13] Además necesitamos considerar el contenido de la variable, que es una palabra formada por letras del alfabeto, pudiendo aparecer además el caracter especial @. Entonces la configuración es un elemento de $N \times (\Sigma \cup \{@\})^*$. Por ejemplo, una configuración sería $[[n, ab@aa]]$.

La relación entre dos configuraciones $C_1 \vdash_M C_2$, que significa que se puede pasar en la MP M de la configuración C_1 a C_2, se define de la manera siguiente:

Definición.- $[[m, au]] \vdash [[n, bw]], a, b \in \Sigma \cup \{\varepsilon, @\},\ u, w \in (\Sigma \cup \{@\})^*$ ssi $\delta(m, a) = n$, y

1. Si $m \in N_T,\ u = bw$

[13] Al decir que estamos en un nodo n, significa que aún no se ejecuta la acción del nodo n.

2. Si $m \in N_{A\sigma}$, $a = b, w = u\sigma$

3. Si $m = s, a = b, u = w$

Definición.-Una palabra $w \in \Sigma^*$ es aceptada por una MP M ssi $[[START, w]] \vdash_M^* [[ACCEPT, v]]$.

Una palabra puede no ser aceptada ya sea porque se cae en un $REJECT$ o bien porque la MP cae en un ciclo infinito.

Ejercicio.- Definir similarmente a como se hizo con las MT la noción de *función calculada*.

8.3.2 Equivalencia entre MP y MT

El mismo Post comprobó la equivalencia entre sus diagramas de flujo y las máquinas de Turing, lo que contribuyó a reforzar la conjetura establecida por A. Church –esto es, que la MT es la más poderosa expresión de lo algorítmicamente calculable.

Teorema de Post.- Para toda MT hay una MP que acepta el mismo lenguaje, o que calcula la misma función, y viceversa.

La prueba del teorema de Post se hace mostrando que una MT puede ser *simulada* por una MP, y viceversa. Al simular MT en MP mostramos que estas últimas son al menos tan poderosas como las primeras (en el sentido de que pueden hacer todo lo que haga MT); similarmente en el sentido contrario. Al establecer ambas direcciones de la prueba se muestra la equivalencia MP-MT. Por "simular" entendemos que, por cada acción de la MT, la MP haga una acción correspondiente, de manera tal que al final del cálculo, una palabra sea aceptada en Post ssi sería aceptada también en Turing; similarmente para el sentido contrario de la prueba.

La simulación de la MT involucra los siguientes aspectos:

- Codificar las configuraciones de la MT en configuraciones de la MP
- Para cada acción de MT, encontrar un diagrama en MP que haga lo mismo. [14]

La codificación de la configuración de la MT en una configuración "equivalente" de la MP involucra considerar cómo codificar cada una de las informaciones de la configuración de MT. En particular, hay que pensar cómo expresar en MP el contenido de la cinta de MT, así como la posición de la cabeza lectora.

[14]Ejercicio: definir formalmente, en términos de configuraciones de MP y MT, qué quiere decir que una acción de MP "hace lo mismo" que la acción correspondiente de MT.

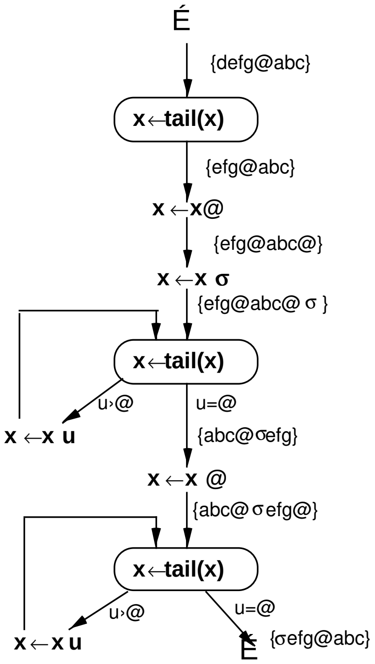


Figura 8.11: Escritura en MP

Sea una configuración $[[q, w, u, v]]$ en MT. Entonces en la variable de la MP tendríamos: $uv@w$. Como se vé, el primer caracter de la variable es el mismo caracter sobre el que está la cabeza lectora en la MT; luego sigue a la derecha la misma cadena que en la MT. En cambio, la parte izquierda de la cinta en MT es colocada en la variable de MP separada por el caracter especial "@". Por ejemplo, si en MT tenemos una cinta de la forma $aba\underline{a}bbb$, la variable de MP contendrá la cadena $abbb@aba$.

Ahora hay que considerar cómo "traducir" las acciones de una MT a acciones correspondientes en una MP. Consideramos los siguientes casos:

- Escritura de caracter: Sea una transición $\delta(p, d) = (q, \sigma)$, donde $\sigma \in \Sigma$. Al paso entre configuraciones de MT:

  $$[[p, abc\underline{d}efg]] \vdash [[q, abc\sigma efg]]$$

  corresponde el paso de x a x' como sigue:

  $$x = defg@abc \;\; x = \sigma efg@abc$$

  Para hacer la transformación indicada (de x a x') en MP, hay que encontrar un diagrama que la efectúe. Un diagrama que cumple con esta función aparece en la figura 8.11.

- Movimiento a la derecha: Al paso entre configuraciones de MT:

  $$[[p, abc\underline{d}efg]] \vdash [[q, abcd\underline{e}fg]]$$

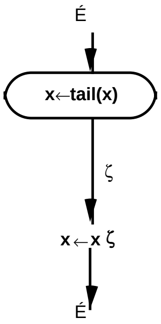


Figura 8.12: Movimiento a la derecha en MP

corresponde el paso de x a x':

$$x = defg@abc \;\; x' = efg@abcd$$

Este paso de x a x' se puede hacer con el (muy simple) diagrama de MP de la figura 8.12.

- Movimiento a la izquierda: A un paso entre configuraciones en MT:

$$[[p, abc\underline{d}efg]] \vdash [[q, ab\underline{c}defg]]$$

corresponde el paso de x a x':

$$x = defg@abc \;\; x' = cdefg@ab$$

El diagrama de la MP que hace dicha operación es dejado como ejercicio (medianamente difícil) al lector.

La prueba de equivalencia MT-MP en el otro sentido –esto es, la simulación por parte de una MT de una MP– es mucho más simple. Primero se toma el contenido inicial de la variable de entrada como palabra de entrada de la MT. Luego cada una de las operaciones de MP ($x \leftarrow x\sigma$, $x \leftarrow tail(x)$, $ACCEPT$, $REJECT$) pueden ser simuladas por la MT correspondiente. Dejamos los detalles de esta prueba al lector.

8.4 Límites de las MT

Aunque parezca increíble, hay problemas que no se pueden resolver como una secuencia determinista de operaciones elementales. que es lo esencial de las MT. Estos problemas son llamados *algorítmicamente irresolubles*. Vamos a concentrar nuestra atención en problemas del tipo: dados una palabra w y (la descripción de) un lenguaje L, decidir si $w \in L$, que son llamados "problemas de pertenencia de palabras"(word problems). Decimos que un lenguaje

L es decidible si hay una MT para decidir el problema de la pertenencia de palabras. Muchos otros problemas que no son del tipo mencionado pueden sin embargo expresarse en términos de éstos mediante una transformación adecuada; por ejemplo, el problema de determinar si dos gramáticas G_1 y G_2 son equivalentes, puede expresarse de la manera siguiente: Para toda $w \in L(G_1)$, decidir si $w \in L(G_2)$.

Para caracterizar en forma precisa los problemas que rebasan las capacidades de las MT, tendremos las siguientes definiciones:

1. Todo lenguaje Turing-decidible es Turing-aceptable
2. Si L es Turing-decidible, L^c es Turing-decidible
3. L es decidible ssi L y L^c son Turing-aceptables

La prueba de 1 es muy sencilla, pues para decidir un lenguaje L, la MT debe primero que nada llegar al *halt* para toda palabra de $w \in L$, con lo que necesariamente acepta w. También el punto 2 es sencillo, pues dada una MT M que decide el lenguaje L, producimos una máquina M' que decide L^c cambiando en M el resultado Y por N y viceversa.

La prueba de 3 es más complicada. Supongamos que tenemos dos MT, M y M^c, que aceptan respectivamente los lenguajes L y L^c. Ponemos a funcionar ambas máquinas "en paralelo", analizando ambas la misma palabra w. Ahora bien, si $w \in L$, eventualmente M llegará al *halt*. Si $w \notin L$, entonces $w \in L^c$, y en algún momento M^c se detendrá. Ahora consideremos una MT adicional M^*, que "observa" a M y a M^c, y que si M se para, entrega una salida Y, mientras que si M^c se para, entrega una salida N. Es evidente que para toda palabra w, M^* decidirá Y o N, por lo que el lenguaje es decidible.

Hay varios detalles técnicos de la prueba anterior que requerirían formalizarse más, como el hecho de que una MT ordinaria puede simular a dos MT en paralelo, que además son "observadas" por una tercera MT. Dejamos como ejercicio pensar en una manera de probar estos aspectos.

8.4.1 El problema del paro de MT

Ahora vamos a considerar un problema irresoluble que históricamente tuvo mucha importancia porque fué el primer problema que se probó irresoluble. Una vez que se cuenta con un primer problema irresoluble, la prueba de que otros problemas son irresolubles consiste en probar que éstos pueden ser reducidos al problema de referencia. Este primer problema irresoluble es el del *paro de la MT*.

El problema del paro de la MT consiste en determinar algorítmicamente –esto es, mediante una MT– si una MT dada M va a parar o no cuando analiza la palabra de entrada w.

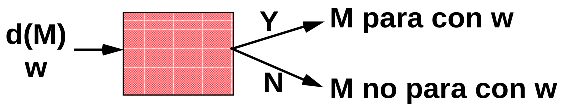


Figura 8.13: El problema del paro de una MT

Desde luego, como una MT analiza el comportamiento de otra, se requiere que esta última sea dada como entrada a la primera; esto puede ser hecho mediante una *codificación* de la MT que debe analizarse. Una manera simple de codificar una MT es considerando la cadena de símbolos de su representación como cuádruplo (K, Σ, δ, s). Denotaremos con $d(M)$ la codificación de una MT M. [15]

Teorema.- No existe ninguna MT tal que, para cualquier palabra w y cualquier MT M, decida si $w \in L(M)$.

En la figura 8.13 se muestra cómo debería funcionar la MT que resolvería el problema del paro.

Prueba.- (Debida a M.Minsky). Por contradicción.- Sea A la MT de la figura 8.14(a). Entonces construimos otra MT B, como se representa en la figura 8.14(b), esto es, se tiene una única entrada con la codificación $d(M)$ de la MT M, y se pasa esta palabra a una MT *copiadora*, que duplica la entrada $d(M)$. La salida de la copiadora será dos veces $d(M)$. Esto es pasado como entrada a una máquina A' que es A modificada [16] de la siguiente manera: a la salida Y de A la cambiamos de forma que en vez de dar el *halt* se cicle; debe quedar claro que esto siempre puede hacerse. Ahora bien, comparando A con A' se vé que la salida Y correponde al hecho de que M para con $d(M)$.

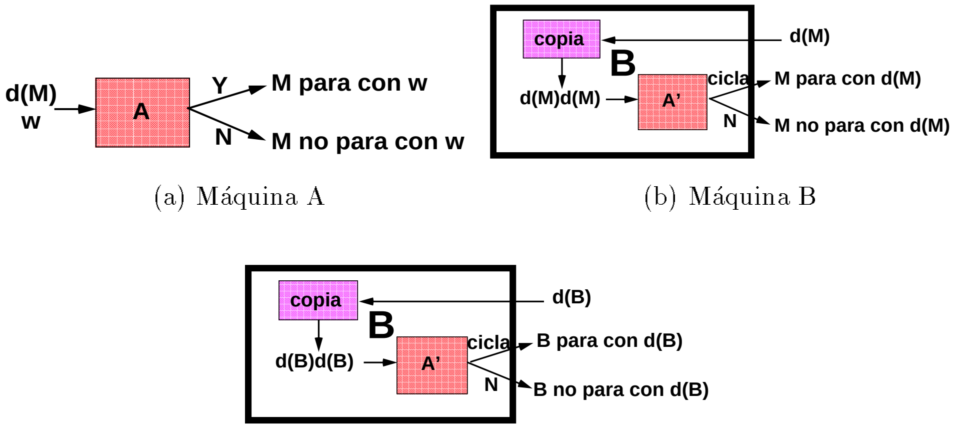
Finalmente supongamos que aplicamos la máquina B a una entrada formada por la misma máquina codificada, esto es, $d(B)$. Entonces cuando B se cicla, esto corresponde a la salida que indica que "B se para con $d(B)$", lo cual es contradictorio. Similarmente, B entrega un resultado N –esto es, se para– en el caso que corresponde a "B no se para con $d(B)$", que también es contradictorio. Esto se ilustra en la figura 8.14(c).

Utilizando el problema del paro de la MT como referencia, se ha probado que otros problemas son también insolubles. Entre los más conocidos, tenemos los siguientes:

- El problema de la equivalencia de las gramáticas libres de contexto.
- La ambigüedad de las GLC.
- El problema de la pertenencia de palabras para gramáticas sin restricciones.

[15] Esta solución para codificar una MT no es perfecta, pues el alfabeto usado para codificar una MT arbitraria no puede determinarse de antemano; no haremos por el momento caso de este detalle técnico.

[16] Obsérvese que la segunda repetición de $d(M)$ es de hecho la palabra w que se supone que es sometida a M.



(a) Máquina A

(b) Máquina B

(c) Contradicción

Figura 8.14: Prueba del paro de MT

No haremos la prueba de estos resultados; remitimos al lector a las referencias [5], [3].

8.5 Ejercicios

1. Diseñe un esquema de máquina de Turing para calcular la función $|log_2 n|$, usando las máquinas básicas vistas. Describa las acciones efectuadas sobre la cinta.

2. Una variante de la MT consiste en hacer que la máquina haga un movimiento y también escriba en cada acción. Dichas máquinas son de la forma (K, Σ, δ, s), pero δ es una función de $(K \times S)$ a $(K \cup \{h\}) \times \Sigma \times \{L, R, S\}$, donde el "movimiento"S significa que la cabeza permanece en el lugar en que estaba. Dé la definición formal de la relación $\vdash$ ("produce en un paso").

3. Proponga una MT (o diagrama) que:

   (a) Acepte las palabras de la forma $a^n b^m$, $n, m > 0$.

   (b) Decida si en una palabra $a^n b^m$ se cumple $m < n$.

   Es la misma máquina para los incisos (a) y (b).

4. Un autómata de dos pilas (A2P) es una extensión directa de un autómata de pila, pero que tiene dos pilas en vez de una, como en la figura 8.15.

   Formalice los A2P en la forma más similar posible a los AP vistos en clase. Defina formalmente las nociones de:

   (a) Configuración.

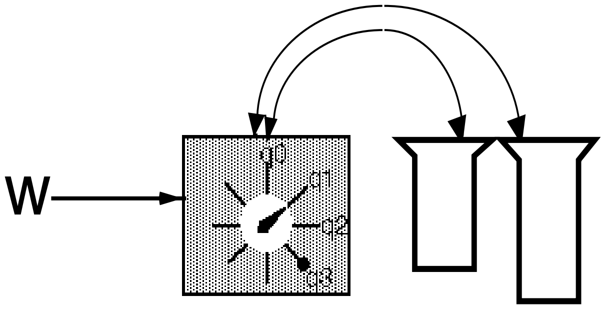


Figura 8.15: Automáta de dos pilas (A2P)

(b) Palabra aceptada y lenguaje aceptado.

(c) Proponga un A2P que acepte el lenguaje $\{a^n b^n c^n\}$.

(d) ¿Tienen los A2P el poder de cálculo de las MT? (Es decir, ¿todo lenguaje Turing-aceptable es aceptado por algún A2P?). Pruebe su respuesta. Hint: mostrar cómo simular una MT con A2P.

(e) Adapte las definiciones de A2P, configuración y palabra aceptada para A2Pn.

(f) Dos A2Pn son equivalentes ssi aceptan el mismo lenguaje. Demuestre que el problema de la equivalencia de los A2Pn es / no es decidible.

5. Suponga un subconjunto MP1 de las máquinas de Post, con la restricción de que no tienen las instrucciones $x \leftarrow x\sigma$. Ahora bien, MP1 es equivalente a AF.

(a) Demuestre esta afirmación constructivamente, proponiendo un método sistemático para pasar de una MP1 a un AF que acepte el mismo lenguaje.

(b) Pruebe el método propuesto en (a) con la MP1 dada en la figura 8.16.

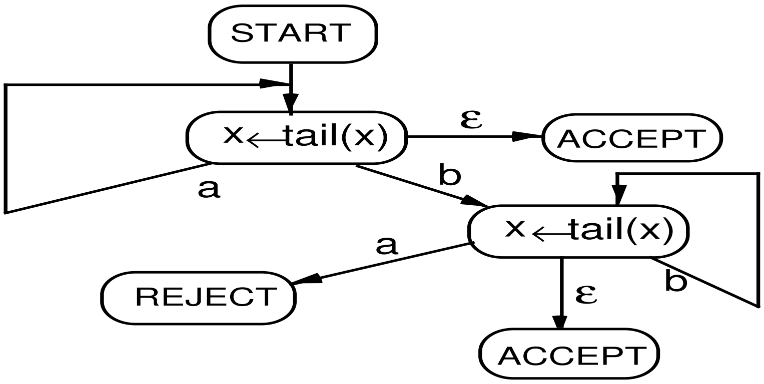


Figura 8.16: Máquina de Post

(c) Pruebe si la MP1 del inciso anterior acepta o no el lenguaje (abb*a)*, basándose en algún procedimiento sistemático (explicar cuál es dicho procedimiento sistemático).

(d) Si en una variante MP2 se permiten asignaciones $x \leftarrow \sigma$, donde $\sigma \in \Sigma^*$, ¿a qué tipo de autómata correspone MP2? ¿Porqué?

6. Si a las máquinas de Post les cambiamos ligeramente las asignaciones para que sean de la forma $x \leftarrow \sigma x$, ¿siguen siendo equivalentes a las MT? Pruebe su afirmación; ya sea:

   (a) Si son equivalentes, probando que pueden hacer lo mismo (por ejemplo, por simulación).

   (b) Si no son equivalentes a Turing, pero son equivalentes a alguna máquina inferior, probando dicha equivalencia.

   (c) Si no son equivalentes a Turing, encontrando algún lenguaje que una sí puede aceptar y la otra no (habría que probar esto).

7. Suponga una variante de las máquinas de Turing, las MT2 en que se tienen dos cintas (cinta 1 y cinta 2) en vez de una; ambas son infinitas hacia la derecha y tienen sendas cabezas lectoras. Por cada transición, se leen a la vez los caracteres de las dos cintas, y el control finito determina la acción a realizar (simultáneamente) en las cintas, que pueden ser movimientos o escrituras, como en una MT normal. Las acciones en las dos cintas son independientes, esto es, en la cinta 1 puede tener una acción L y en la cinta 2 escribir, etc., en un sólo movimiento. La palabra de entrada se escribe en la cinta 1.

   (a) Proponga una definición formal de las MT2

   (b) Defina las nociones de configuración y palabra aceptada.

   (c) Defina función calculada, suponiendo que el resultado queda en la cinta 2.

8. Proponer una MT (su diagrama) que:

   (a) Acepte el lenguaje vacío ($\emptyset$)

   (b) Decida el lenguaje vacío

   (c) Acepte el lenguaje $\{\varepsilon\}$

   (d) Decida el lenguaje $\{\varepsilon\}$

   (e) Una Máquina de Post que acepte el lenguaje vacío

   (f) Acepte el lenguaje $\{\varepsilon\}$

9. Suponga una variante de las MT, las MTS en que se puede al mismo tiempo escribir en la cinta y hacer los movimientos a la izquierda y a la derecha (L y R); cuando se quiere sólamente escribir (sin mover la cabeza) se hace un movimiento nulo (N).

   (a) Defina las MTS, así como su funcionamiento (hasta definir palabra aceptada).

   (b) Pruebe que las MTS son tan poderosas como la MT clásica (muestre cómo obtener a partir de una MT la MTS equivalente).

   (c) Pruebe ahora lo recíproco, mostrando cómo obtener una MT clásica a partir de una MTS dada.

10. Conteste las siguientes preguntas, justificando la respuesta:

    (a) ¿El complemento de un lenguaje Turing-decidible es también Turing-decidible?

    (b) ¿El complemento de un lenguaje Turing-decidible es Turing-aceptable?

    (c) ¿Todo lenguaje Turing-decidible será subconjunto de algún lenguaje libre de contexto?

    (d) ¿La intersección de un Turing-aceptable con un libre de contexto será libre de contexto?

11. Es sabido que el problema de la equivalencia de MT es indecidible. Sin embargo, para algunos subconjuntos de las MT sí es posible decidir la equivalencia. Para las siguientes MT (no deterministas), probar rigurosamente su equivalencia / no equivalencia respecto a la aceptación / rechazo de palabras (es decir, que los lenguajes aceptados son iguales), describiendo el método utilizado para esta prueba:

$$MT_1 = (\{f, g, j, k, m\}, \{a, b, \#\}, \delta_1, f)$$

f	#	g	L
g	#	h	#
g	a	L	L
g	b	f	L
f	a	f	L
f	b	f	L
j	a	m	L
j	b	k	L
k	a	m	L
k	b	f	L
k	#	h	#
m	a	m	L
m	b	k	L

$$MT_2 = (\{o, q, n, p\}, \{a, b\}, \delta_2, q)$$

q	#	o	L
o	#	h	#
o	a	n	L
o	b	p	L
n	a	n	L
n	b	o	L
p	a	p	L
p	b	p	L

12. Si limitamos el tamaño de la cinta de una máquina de Turing a una cantidad fija k de cuadros, dando una variante que llamaremos MTk,

(a) ¿disminuye por ello el poder de cálculo? ¿A qué tipo de autómatas serían equivalentes las MTk ? Pruebe su respuesta.

(b) ¿Es posible decidir si dos MTk son equivalentes? Pruebe su respuesta, y en el caso afirmativo, proponga el método de decisión correspondiente.

Bibliografía

[1] A. Aho, J. Ullman.- *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1978.

[2] G. Brookshear.- *Teoría de la Computación*, Addison Wesley Iberoamericana, 1993.

[3] J. Hopcroft, J. Ullman.- *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, 1979.

[4] D. Kelley.- *Teoría de Autómatas y Lenguajes Formales*, Prentice Hall Hispanoamericana, 1995.

[5] H.R. Lewis, Ch.H. Papadimitriou.- *Elements of the Theory of Computation*, Prentice Hall, 1981.

[6] J.L. Peterson.- *Petri net theory and the modeling of systems*, Prentice Hall, 1981.

[7] C. Petri.- *Kommunication mit Automaten*, Universidad de Bonn, 1962.

[8] S. Sahni.- *Concepts in Discrete Mathematics*, Camelot Publishing Co. 1985.

[9] M. Silva.- *Las Redes de Petri en la Automática y la Informática*, Editorial AC, 1985

[10] T. Sudkamp.- *LANGUAGES AND MACHINES.- An Introduction to the Theory of Computer Science*, Addison Wesley, 1994.