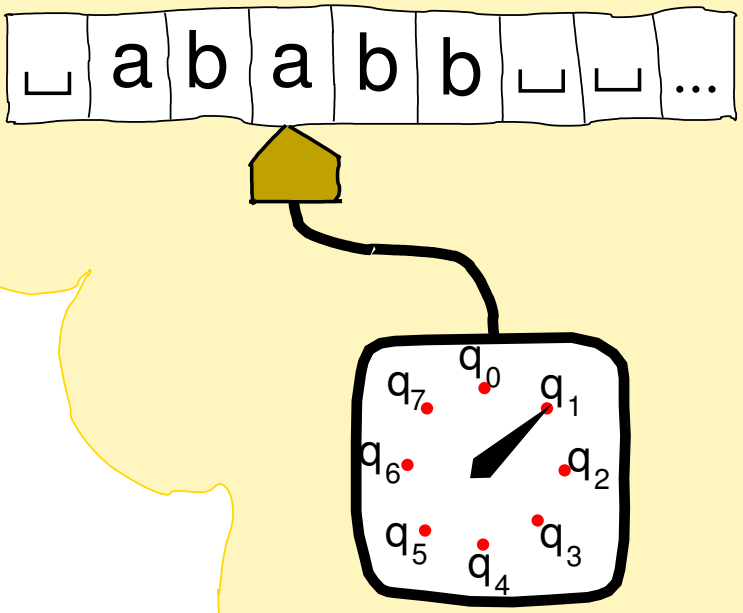
AUTOMATAS Y LENGUAJES

Un enfoque de diseño



Ramón Brena
Tec de Monterrey
Verano 2003

Prefacio

En años recientes se ha visto la aparición de un buen número de textos en el tema de Lenguajes Formales y Autómatas (Ver al final referencias [10], [7], [23], [8], [3], [21], 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 que indique su 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 autómatas 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 [23]. Estos nuevos textos han reemplazado en muchas universidades a los "clásicos" [6] y aún [10] -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, [1] en Monterrey, México. Durante este lapso, aunque ciertamente se fue 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 comprenderlo de manera eficiente. Aquí el énfasis no está tanto en hacer el curso "más fácil" para los estudiantes, sino en asegurarse de que éstos cuenten con los elementos para que *ellos mismos* reconstruyan estos contenidos dentro de su cabeza; no se trata, pues, simplemente de "vaciar" información en la cabeza del estudiante. La teoría educativa que sustenta esta forma de trabajo esta basada en el "aprendizaje por reestructuración" [18].

El texto está presentado de manera tal que es posible para el alumno estudiar el material *antes* de cubrir el tema en clase; de hecho esta es la forma en que se utiliza en el ITESM, contrariamente a muchas clases tradicionales, en las que el alumno se presenta a la exposición del profesor y ya *luego* estudia el texto. En el ITESM la clase no se utiliza principalmente para exposición del profesor, sino que se hacen ejercicios, problemas en equipo, miniexámenes semanales, etc. Esta situación exige del texto que sea comprensible sin tener *ninguna noción* del tema adquirida previamente, por lo que tuvimos que incluir explicaciones claras que permitan al alumno reconstruir en su mente la idea intuitiva, y -sobre todo- ejemplos. A lo largo del texto, cada una de las nociones presentadas es seguida inmediatamente por un ejemplo ilustrativo.

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 aplica en ambos niveles. La diferencia fundamental entre el enfoque del curso de nivel profesional y el

[1] Abreviatura de "Instituto Tecnológico y de Estudios Superiores de Monterrey".

de maestría estriba en que el curso de nivel ingeniero enfatiza los aspectos de "saber hacer", (por ejemplo, saber comparar dos autómatas deterministas), mientras que el curso de nivel maestría enfatiza el "saber justificar" (por ejemplo, probar por inducción que una gramática es correcta).

El material cuyo nivel es propiamente de maestría es identificado por medio de una barra vertical al margen, como en el presente párrafo. Esto incluye también las secciones de ejercicios.

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. Por ejemplo, primero se presentan las nociones de manera intuitiva, y solamente después se procede a su formalización.

- Es aplicable tanto al nivel de maestría como en carreras de ingeniería en computación, mostrando en forma explícita y gráfica qué secciones están destinadas a cada nivel.

- Siendo un libro más orientado a estudiantes de ingeniería que de matemáticas, se enfatizan los temas que tienen comúnmente aplicación en su campo profesional, como los autómatas finitos. Esta es la razón por la que se cubren con mucho más detalle estos temas que otros de interés más teórico, como la calculabilidad en máquinas de Turing. Sabemos de alumnos que han conseguido un buen empleo no universitario gracias a su conocimiento de autómatas finitos.

- Por la misma razón del punto anterior, ciertos temas que tradicionalmente se exponen con una motivación matemática, como las propiedades de los "Lenguajes Regulares", en este texto se presentan en el contexto de *métodos de diseño*, lo que es consistente con nuestro enfoque ingenieril. Es este aspecto lo que justifica el subtítulo "un enfoque de diseño" de este texto.

- Ofrecemos *metodologías* para resolver ciertas clases de problemas, tales como el diseño de expresiones regulares y gramáticas, que no son presentadas en otros libros de teoría de autómatas, o lo hacen de forma mucho más escueta. Inclusive algunos temas, tales como las propiedades de cerradura de los lenguajes regulares a la unión de conjuntos, se presentan aquí como una herramienta de solución de problemas de diseño, y no simplemente por el interés matemático del tema.

- Presentamos errores frecuentes de los estudiantes de esta materia, permitiendo de este modo que el lector se beneficie de una extensa experiencia directa en la enseñanza de la materia.

- Los algoritmos no se presentan en forma de "pseudocódigo", es decir, usando estructuras de control de lenguajes imperativos (p.ej. *while*, *for*, etc.), sino que damos una

interpretación intuitiva de los resultados intermedios obtenidos por los algoritmos. Pensamos que este enfoque brinda una mejor comprensión de los algoritmos, pues es más fácil recordar ideas que lineas de código.

- ¡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 libro, en tanto que libro electrónico, es un archivo estándar de tipo PDF, con "hiperligas" que permiten localizar rápidamente figuras, citas, páginas del texto, etc.
- ¡El libro es gratuito! En efecto, no se distribuye con fines de lucro. Esto no pretende atentar contra la industria editorial, sino apoyar el aprendizaje del área de autómatas y lenguajes en América Latina, región que no está sobrada de recursos como para querer engrosar los flujos de capital hacia los grandes centros editoriales del mundo.

La estructura de este texto es la siguiente: después de una breve revisión de las nociones preliminares de matemáticas, en los primeros capítulos (2-3) 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.

En los capítulos 4 y 5 veremos los *Lenguajes Libres de Contexto* y los *Autómatas de Pila*.

Finalmente, a partir del capítulo 6 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.

Típicamente, en un curso de nivel profesional se inicia con el capítulo de preliminares, y se continúa con los capítulos 2-3. Se enfatizan los capítulos 4 y 5, así como la teoría de los compiladores. A continuación se cubren los aspectos básicos de las Máquinas de Turing (inicio de capítulo 6).

En el curso de maestría, la revisión de preliminares casi se omite, y se cubren los capítulos 2-3, sólo que en un nivel de profundidad mayor que en el caso del nivel profesional. [2] Luego se procede a estudiar los Autómatas de Pila, las Máquinas de Turing y la Tesis de Church (capítulos 4, 5 y 6), con énfasis en las pruebas.

Agradezco la colaboración del Dr. José Luis Aguirre en la corrección de los errores en versiones previas, así como en sugerencias para mejorar la exposición de ciertos temas. 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.

[2] Recordar que el material de nivel maestría está indicado con una barra vertical en el margen.

Índice general

Parte I

Lenguajes regulares y sus máquinas

Capítulo 2

Autómatas finitos

El término máquina evoca algo hecho en metal, usualmente ruidoso y grasoso, que ejecuta tareas repetitivas que requieren de mucha fuerza o velocidad o 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 fluidos. 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 modelar 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.

El modelado de una máquina en lo relacionado con secuencias o ciclos de acciones se aproxima más al enfoque que adoptaremos en este curso. Las máquinas que estudiaremos son abstracciones matemáticas que capturan solamente 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.

2.1. Modelado de sistemas discretos

Antes de definir los autómatas finitos, empezaremos examinando las situaciones de la realidad que pueden ser modeladas usando dichos autómatas. De esta manera, iremos de lo más concreto a lo más abstracto, facilitando la comprensión intuitiva del tema.

El modelado de fenómenos y procesos es una actividad que permite:

- Verificar hipótesis sobre dichos procesos;
- Efectuar predicciones sobre el comportamiento futuro;
- Hacer simulaciones (eventualmente computarizadas);
- 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 solamente 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.

La noción más básica de los modelos de eventos discretos es la de *estado*. Un estado es una situación en la que se permanece un cierto lapso de tiempo. Un ejemplo de la vida real es el de los "estados civiles" en que puede estar una persona: soltera, casada, viuda, divorciada, etc. De uno de estos estados se puede pasar a otro al ocurrir un *evento* o acción, que es el segundo concepto básico de la modelación discreta. Así, por ejemplo, del estado "soltero" se puede pasar al estado "casado" al ocurrir el evento "boda". Similarmente, se puede pasar de "casado" a "divorciado" mediante el evento "divorcio". En estos modelos se supone que se permanece en los estados un cierto tiempo, pero por el contrario, los eventos son instantáneos. Esto puede ser más o menos realista, dependiendo de la situación que se está modelando. Por ejemplo, en el medio rural hay bodas que duran una semana, pero desde el punto de vista de la duración de una vida humana, este tiempo puede considerarse despreciable. En el caso del evento "divorcio", pudiera ser inadecuado considerarlo como instantáneo, pues hay divorcios que duran años. En este caso, el modelo puede refinarse definiendo un nuevo estado "divorciándose", al que se llega desde "casado" mediante el evento "inicio divorcio".

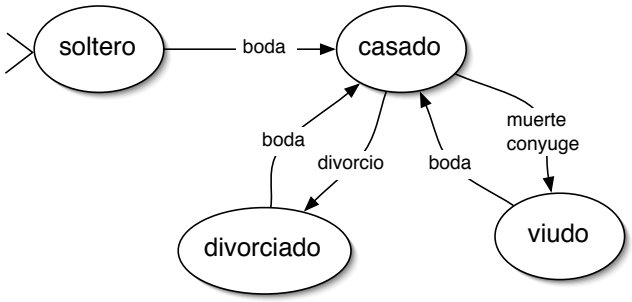


Figura 2.1: Modelo de estados civiles de una persona

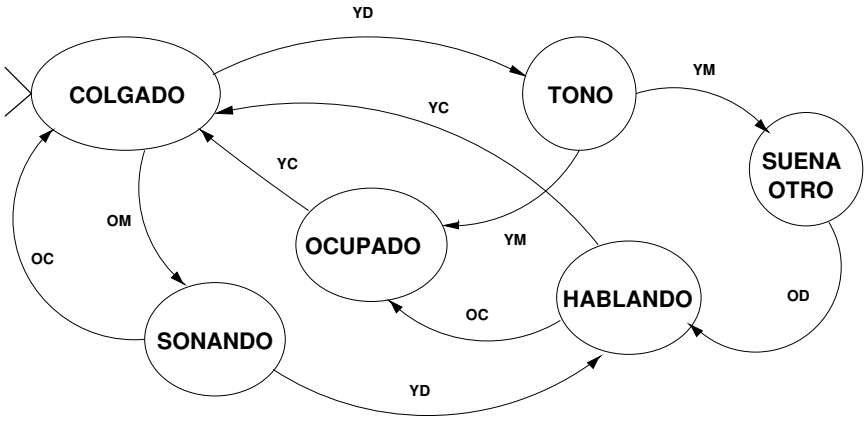


Figura 2.2: Modelo en eventos discretos de un teléfono

Es sumamente práctico expresar los modelos de estados y eventos de manera gráfica. Los estados se representan por óvalos, y los eventos por flechas entre los óvalos, llamadas *transiciones*. Dentro de cada estado se escribe su nombre, mientras que al lado de las transiciones se escribe el nombre del evento asociado, como en la figura 2.1. El estado donde se inicia tiene una marca ">", en este caso "soltero".

En la figura 2.2 se presenta un modelo simplificado del funcionamiento de un aparato telefónico. En esta figura los nombres de los estados se refieren al aparato desde donde llamo, contesto, etc., y en caso contrario se especifica que es el otro ("suena otro", que se refiere al aparato telefónico del interlocutor). En las transiciones, la "Y" inicial se refiere a acciones que hace uno mismo (por ejemplo, "YD", que es "yo descuelgo"), mientras que la "O" se refiere al otro teléfono. La "C" de "YC" se refiere a "colgar", mientras que la "M" es "marcar". Así, el significado de las transiciones YC, OC, YM, OM, YD y OD deben quedar claras.

En este ejemplo suponemos que el estado en que inicia el proceso (que llamaremos *estado inicial*) es con el auricular colgado, sin sonar aún. A partir de esa situación, pueden ocurrir varios eventos que nos lleven a un nuevo estado, como por ejemplo que empiece a sonar o bien que alguien descuelgue para marcar un número.

Desde luego, elaborar modelos "adecuados" de un proceso real es un arte que requiere

práctica, pero en general los siguientes lineamientos pueden ser útiles:

1. Diferenciar entre los eventos que se consideran instantáneos y aquellos que tienen una duración considerable: estos últimos se asocian a los estados. Los estados son la base de un diseño de los modelos que estamos estudiando, pues "recuerdan" las situaciones básicas por las que pasa el proceso.

2. Las condiciones asociadas a los estados deben ser *excluyentes*, esto es, no deben verificarse varias simultáneamente. Por ejemplo, una persona no es soltera y casada a la vez.

3. Las condiciones asociadas a los estados de un modelo bien hecho deben ser *comprensivas*, lo que quiere decir que entre todas ellas cubren todos los casos posibles. Por ejemplo, en el modelo de estados civiles suponemos que una persona es ya sea soltera, o bien casada, o bien divorciada, sin haber otras opciones. Si necesitamos considerar el concubinato como otra condición, habría que modificar el modelo.

4. Los eventos instantáneos son asociados a los eventos. En el ejemplo, el levantar el auricular (que se supone una acción instantánea) es una transición, mientras que se supone que puede transcurrir un tiempo antes de que el usuario marque un número, por lo que hay un estado entre estos dos eventos.

En el ejemplo del teléfono, estamos considerando que al descolgar el auricular, el tono de marcar está inmediatamente disponible, aunque en ciertas ciudades esta suposición puede ser una simplificación inaceptable. En cambio, en el mismo ejemplo consideramos que la persona que contesta el teléfono no lo hace inmediatamente, sino que hay un inicio y un fin del timbre -aunque mi suegra acostumbra contestar el teléfono antes de que se complete el primer timbrazo. Para los eventos con duración, es necesario identificar un evento de inicio y otro de terminación, como en el ejemplo del divorcio que mencionamos antes. Desde luego, la decisión de qué eventos son instantáneos y cuales tienen duración depende enteramente de qué es importante en el problema particular que se desea modelar.

Los errores que más frecuentemente se cometen al hacer modelos de estados y eventos son:

- Confundir estados con eventos; por ejemplo, tener un estado "salir de casa", que razonablemente corresponde a un evento instantáneo. [1]

- Proponer conjuntos de estados no excluyentes, esto es, que se traslapan, como sería tener estados "Se encuentra en Acapulco" y "Se encuentra fuera de Guadalajara", pues pueden verificarse ambos simultáneamente, lo que no es posible en los estados.

[1] Si no se quiere que "salir de casa" sea un evento instantáneo, se debe reexpresar de forma que su duración sea evidente, como en "preparándose para salir de casa".

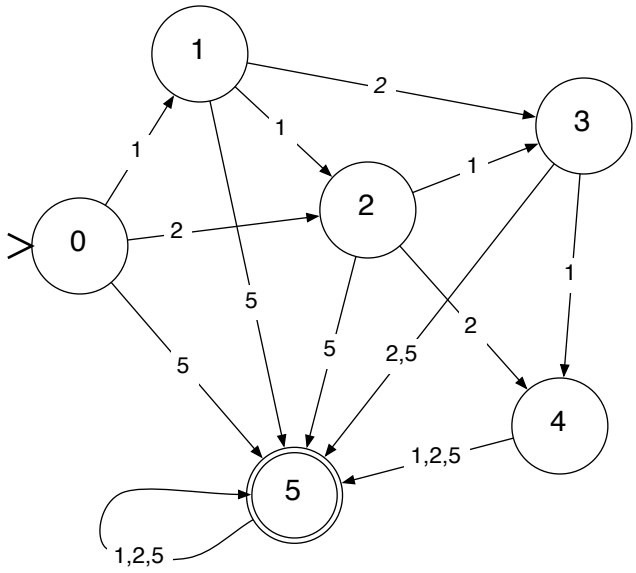


Figura 2.3: Modelo con estados finales

- Proponer conjuntos de estados no comprensivos, donde falta algún caso o situación por considerar.

En situaciones muy complejas, donde varios procesos evolucionan concurrentemente, el modelado de eventos discretos por medio de estados y eventos no es adecuado, pues los diagramas son demasiado grandes. En estos casos se requieren herramientas más sofisticadas, como las llamadas "redes de Petri" [16].

2.1.1. Estados finales

El propósito de algunos modelos de estados y eventos es el de reconocer secuencias de eventos "buenas", de manera que se les pueda diferencias de las secuencias "malas". Supóngase, por ejemplo, que se quiere modelar el funcionamiento de una máquina automática vendedora de bebidas enlatadas. Dicha máquina acepta monedas de valor 1, 2 y 5, y el precio de cada lata es de 5. Vamos a considerar que el evento llamado "1" es la introducción de una moneda de valor 1 en la máquina, el evento "2" para la moneda de valor 2, etc.

La primera cuestión que hay que resolver para diseñar nuestro modelo es decidir cómo son los estados. Una buena idea sería que cada estado recordara lo que se lleva acumulado hasta el momento. El estado inicial, desde luego, recordaría que se lleva acumulado 0. Con estas ideas podemos hacer un diagrama de estados y eventos como el de la figura 2.3. Muchas transiciones en dicho diagrama son evidentes, como el paso del estado "1" al "3" tras la introducción de una moneda de valor 2. En otros casos hay que tomar una decisión de diseño conflictiva, como en el caso en que en el estado "4" se introduzca una moneda de valor 2. En el diagrama presentado, se decidió que en ese caso se va al estado "5", lo que en la práctica

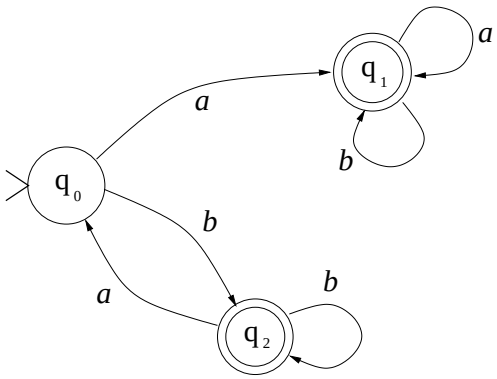


Figura 2.4: Notación gráfica

puede querer decir que la máquina entrega un cambio al usuario, o bien simplemente se queda con el sobrante.

Un aspecto muy importante del modelo de la figura 2.3 es que el estado "5" es un estado especial, llamado estado *final*, e identificado por un óvalo de doble trazo. Los estados finales indican que cuando se llega a ellos, la secuencia de eventos que llevó hasta ahí puede considerarse como "aceptable". Por ejemplo, en la máquina vendedora de latas, la secuencia de eventos "meter 2", "meter 1", "meter 2" puede considerarse aceptable porque totaliza 5. En la figura puede observarse que dicha secuencia hace pasar por los estados 0, 2, 3 y 5, donde este último es final. De este modo el diagrama nos permite diferencias las secuencias aceptables respecto a otras que no lo son, como la secuencia "meter 1", "meter 2", "meter 1", que lleva al estado 4, que no es final. Obsérverse que la secuencia "meter 5", "meter 5", "meter 5" también es aceptable –desde luego, desde el punto de vista de la máquina, aunque seguramente no lo sea desde el punto de vista del cliente.

2.2. Máquinas de estados finitos

A partir de ahora vamos a considerar modelos de estados y eventos un poco más abstractos que los que hemos visto antes. Retomemos el ejemplo de la máquina vendedora de latas, que vimos en la sección 2.1.1. En ese modelo pudimos reconocer secuencias de eventos "aceptables", como la secuencia de monedas 2, 2, 1 con respecto a secuencias no aceptables, como 1, 1, 1. A partir de ahora los nombres de los eventos van a estar formados por un caracter, y les llamaremos *transiciones* en vez de "eventos". De este modo, en vez de un evento "meter 1" vamos a tener una transición con el caracter "1", por ejemplo. Desde luego, la elección de qué caracter tomar como nombre de la transición es una decisión arbitraria.

Además, las secuencias de eventos van a representarse por concatenaciones de caracteres, esto es, por *palabras*. Así, en el ejemplo de la máquina vendedora la palabra "1121" representa la secuencia de eventos "meter 1", "meter 1", "meter 2", "meter 1".

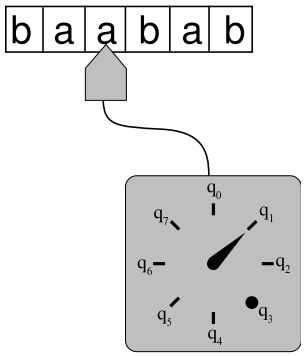


Figura 2.5: Componentes de una máquina abstracta

Desde el punto de vista abstracto que vamos a adoptar a partir de ahora, nuestras máquinas pueden ser visualizadas como dispositivos con los siguientes componentes: (ver figura 2.5)

- Una cinta de entrada;
- Una cabeza de lectura (y eventualmente escritura);
- 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; también puede recorrerse un lugar a la derecha (o a la izquierda también, según el tipo de máquina). 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.

2.2.1. Funcionamiento de los autómatas finitos

Como se había comentado antes, el funcionamiento de los autómatas finitos consiste en ir pasando de un estado a otro, a medida que va recibiendo los caracteres de la palabra de entrada. Este proceso puede ser seguido fácilmente en los diagramas de estados. Simplemente hay que pasar de estado a estado siguiendo las flechas de las transiciones, para cada caracter de la palabra de entrada, empezando por el estado inicial. Por ejemplo, supóngase que tenemos el autómata de la figura 2.4 y la palabra de entrada "*bb*". El autómata inicia su operación en el estado q_0 –que es el estado inicial–, y al recibir la primera b pasa al estado q_2, pues en el diagrama hay una flecha de q_0 a q_2 con la letra b. Luego, al recibir la segunda

b de la palabra de entrada, pasará del estado q_2 a él mismo, pues en la figura se puede ver una flecha que de q_2 regresa al mismo estado, con la letra b.

Podemos visualizar el camino recorrido en el diagrama de estados como una "trayectoria" recorrida de estado en estado. Por ejemplo, para el autómata finito de la figura 2.4 la trayectoria seguida para la palabra ab consiste en la secuencia de estados: q_0, q_1, q_1.

Los estados son el único medio de que disponen los AF 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 de su memoria podría ser enorme.

2.3. Definición formal de autómatas finitos

Al describir una máquina de estados finitos en particular, debemos incluir las informaciones que varían de un autómata a otro; es decir, no tiene sentido incluir descripciones generales aplicables a todo autómata. Estas informaciones son exactamente las que aparecen en un diagrama de estados y transiciones, como los que hemos presentado antes.

En esta sección vamos a presentar un formato matemático para representar las mismas informaciones que contiene un diagrama de estados. Como se utiliza terminología matemática en vez de dibujos, decimos que se trata de una *notación formal*. En particular, utilizamos nociones de la teoría de conjuntos que fueron ya presentadas en el capítulo 1.

Definición.- 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. [2]

La función de transición indica a qué estado se va a pasar sabiendo cuál es el estado actual y el símbolo 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

[2] que puede ser el mismo en el que se encontraba.

estado, se llama *determinismo*. La definición dada arriba corresponde a los *autómatas finitos determinístas*, abreviado "AFD" [3]

Ejemplo.- El autómata finito determinista de la figura 2.4 puede ser expresado formalmente como: $M = (K, \Sigma, \delta, q_0, F)$, donde:

- $K = \{q_0, q_1, q_2\}$
- $\Sigma = \{a, b\}$
- $\delta = \{((q_0, a), q_1), ((q_0, b), q_2), ((q_1, a), q_1), ((q_1, b), q_1), ((q_2, a), q_0), ((q_2, b), q_2)\}$
- $F = \{q_1, q_2\}$

La función de transición δ puede ser expresada mediante una tabla como la siguiente, para este ejemplo:

$\mathbf{q}$	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

Es fácil ver que la diferencia entre los diagramas de estado y los AFD en notación formal es solamente de notación, siendo la información exactamente la misma, por lo que es sencillo pasar de una representación a la otra.

Tanto en los diagramas de estado como en la representación formal hay que tener cuidado en respetar las condiciones para que tengamos un autómata válido; en particular, el número de transiciones que salen de cada estado debe ser igual a la cantidad de caracteres del alfabeto, puesto que δ es una función que está definida para todas las entradas posibles. [4]

Para el ejemplo de la figura 2.4, donde el alfabeto es $\{a, b\}$, de cada estado deben salir exactamente dos transiciones, una con a y otra con b.

Otra condición es que debe haber exactamente un estado inicial. En cambio, la cantidad de estados finales puede ser cualquiera, inclusive cero, hasta un máximo de $|K|$ (la cantidad de estados).

[3] Después veremos otros autómatas finitos, llamados *no determinístas*.

[4] Recuérdese que una función no puede tener más de un resultado (en este caso, un estado de llegada) para cada entrada (en este caso, un estado de salida y un caracter consumido).

En la notación formal también hay que seguir las transiciones, que ahora no son representadas como flechas, sino como elementos del conjunto δ de transiciones. Tomando nuevamente el autómata de la figura 2.4 y la palabra de entrada *bb*, la operación se inicia en el estado inicial q_0; luego, al recibir la primera b, usando la transición $((q_0, b), q_2)$ pasa a q_2, y luego, al recibir la segunda b de la palabra de entrada, por medio de la transición $((q_2, b), q_2)$ pasa al estado q_2 –de hecho permanece en él.

De una manera más general, si un AFD se encuentra en un estado q y recibe un caracter σ pasa al estado q' ssi $\delta(q, \sigma) = q'$, esto es, si $((q, \sigma), q') \in \delta$.

Palabras aceptadas

Los autómatas finitos que hemos visto pueden ser utilizados para reconocer ciertas palabras y diferenciarlas de otras palabras.

Decimos que un AFD reconoce o *acepta* una palabra si se cumplen las siguientes condiciones:

1. Se consumen todos los caracteres de dicha palabra de entrada, siguiendo las transiciones y pasando en consecuencia de un estado a otro;
2. al terminarse la palabra, el estado al que llega es uno de los estados *finales* del autómata (los que tienen doble círculo en los diagramas, o que son parte del conjunto F en la representación formal).

Así, en el ejemplo de la figura 2.4, el autómata acepta la palabra *bb*, pues al terminar de consumirla se encuentra en el estado q_2, el cual es final.

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

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

Por ejemplo, el autómata de la figura 2.4 acepta las palabras que empiezan con a, así como las palabras que contienen *aa*, y también las que terminan en b, como por ejemplo *abab*, *aaaaa*, *baaa*, etc. En cambio, no acepta *baba* ni *bba*, *babba*, etc. Nótese que tampoco acepta la palabra vacía ε. Para que un AFD acepte ε se necesita que el estado inicial sea también final.

Formalización del funcionamiento de los AFD

El funcionamiento 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

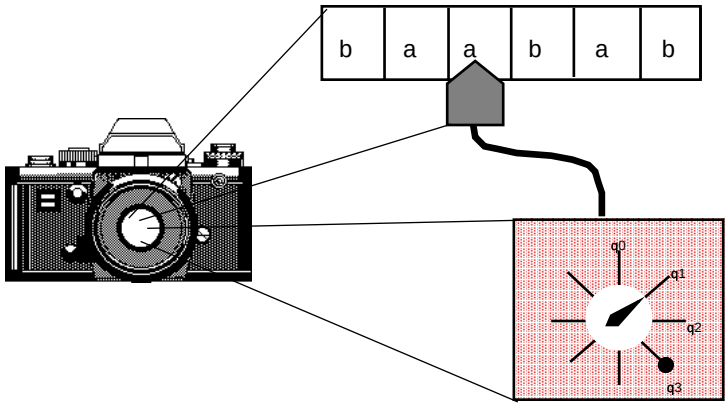


Figura 2.6: La configuración es como una fotografía de la situación de un autómata en medio de un cálculo

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 2.6. 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 2.5 sería: $(q_1, abab)$.

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

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.- $[[q_1, \sigma w]] \vdash_M [[q_2, w]]$ para un $\sigma \in \Sigma$ si y sólo si 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 $\vdash_M^*$. 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).

Ejemplo.- Probar que el AFD de la figura 2.4 acepta la palabra *babb*.

Solución.- Hay que encontrar una serie de configuraciones tales que se pueda pasar de una a otra por medio de la relación $\vdash_M$. La única forma posible es la siguiente: [5]

$$[[q_0, babb]] \vdash_M [[q_2, abb]] \vdash_M [[q_0, bb]]$$
$$\vdash_M [[q_2, b]] \vdash_M [[q_2, \varepsilon]].$$

Como $q_2 \in F$, la palabra es aceptada.

Definición.- Un *cálculo* en una máquina M es una secuencia de configuraciones C_1, C_2, $\ldots$, 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:

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

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

[5] En los AFD's, para cada palabra de entrada sólo hay una secuencia posible de configuraciones, precisamente porque son deterministas.

y sean $[[r, w']]$ y $[[s, w']]$ las primeras configuraciones distintas en los dos cálculos. [6] 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.

2.4. Métodos de diseño de AFDs

Considérese el problema de construir un AFD que acepte exactamente un lenguaje dado. Este problema es comúnmente llamado "problema de diseño". No es conveniente proceder por "ensayo y error", puesto que en general hay que considerar demasiadas posibilidades, y es muy fácil equivocarse. Más aún, hay dos maneras de equivocarse al diseñar un AFD: [7]

1. Que "sobren palabras", esto es, que el autómata acepte algunas palabras que no debería aceptar. En este caso decimos que la solución es *incorrecta*.

2. Que "falten palabras", esto es, que haya palabras en el lenguaje considerado que no son aceptadas por el AFD, cuando deberían serlo. En este caso decimos que la solución es *incompleta*.

Por ejemplo, supongamos que alguien propone el autómata de la figura 2.4 para el lenguaje de las palabras en el alfabeto $\{a, b\}$ que no tienen varias a's seguidas. Esta solución es defectuosa, porque:

1. Hay palabras, como "baa", que tiene a's seguidas y sin embargo son aceptadas por el AFD;

2. Hay palabras, como "ba", que no tienen a's seguidas y sin embargo no son aceptadas por el AFD.

Como se ve, es posible equivocarse de las dos maneras a la vez en un sólo autómata.

La moraleja de estos ejemplos es que es necesario diseñar los AFD de una manera más sistemática.

El elemento más importante en el diseño sistemático de autómatas a partir de un lenguaje *consiste en determinar, de manera explícita, qué condición "recuerda" cada uno de los estados del AFD*. El lector debe concientizarse de que este es un principio de diseño importantísimo, verdaderamente básico para el diseño metódico de autómatas.

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

[7] Estos errores no son excluyentes, y es posible que se presenten ambos a la vez.

(a) Diseño de estados (b) AFD completo

Figura 2.7: Diseño de AFD para palabras con número impar de a's

Recuérdese que la única forma de memoria que tienen los AFD es el estado en que se encuentran. Así, el diseño del AFD inicia con la propuesta de un conjunto de estados que "recuerdan" condiciones importantes en el problema considerado. Posteriormente se proponen las transiciones que permiten pasar de un estado a otro; esta última parte es relativamente sencilla una vez que se cuenta con los estados y sus condiciones asociadas.

Ejemplo.- Diseñar un AFD que acepte las palabras en el alfabeto $\{a, b\}$ en que la cantidad de a's es impar.

Solución.- Las condiciones relevantes para este problema -que deben ser "recordadas" por los estados correspondientes- son:

- El número de a's recibidas hasta el momento es par (estado P);
- El número de a's recibidas hasta el momento es impar (estado I);

Al iniciar la operación del autómata no se ha recibido aún ninguna a, por lo que debemos encontrarnos en el estado P (el cero es un número par), y por lo tanto el estado P es inicial.

Para determinar qué estados son finales, debemos fijarnos en cuáles corresponden con el enunciado original de las palabras aceptadas. En este caso vemos que el estado I es el que corresponde, por lo que es final, mientras que P no corresponde y no es final.

Los estados P e I aparecen en la figura 2.7(a). Esta es la primera etapa del diseño de un AFD. En nuestro método de diseño es importante *trazar las transiciones únicamente después de haber determinado cuáles son los estados y sus características.* Ahora ya podemos trazar las transiciones, lo cual es una tarea relativamente sencilla, si ya tenemos el diseño de los estados. Por ejemplo, si estamos en P y recibimos una a, claramente debemos irnos a I, porque la cantidad de a's pasa de ser par a impar. Similarmente se hacen las otras transiciones. El resultado se muestra en la figura 2.7(b).

Ejemplo.- Diseñar un AFD que acepte exactamente el lenguaje en el alfabeto $\{0, 1\}$ en que las palabras no comienzan con 00.

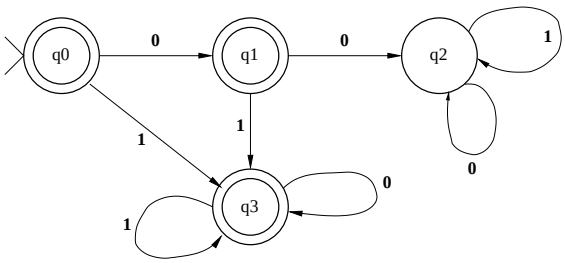


Figura 2.8: AF para palabras que no empiezan en "00"

Solución.- Para emprender el diseño en forma metódica, comenzamos por determinar las condiciones que es importante recordar, y asociamos un estado a cada una de estas condiciones, según la tabla siguiente:

Estado	**Condición**
q_0	No se han recibido caracteres
q_1	Se ha recibido un cero al inicio
q_2	Se han recibido dos ceros iniciales
q_3	Se recibió algo que no son dos ceros iniciales

Claramente tanto q_0 como q_1 deben ser estados finales, mientras que q_2 no debe ser final. 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 como se ilustra en la figura 2.8.

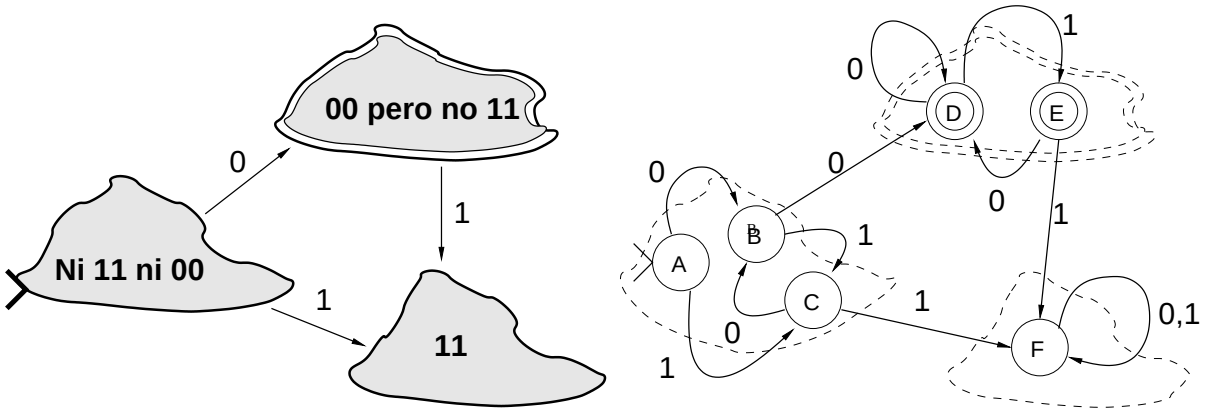
En este ejemplo se puede apreciar que en ocasiones es necesario completar el conjunto de estados al momento de hacer las transiciones.

2.4.1. Diseño por conjuntos de estados

Es posible llevar un paso más allá el método de asociar una condición a cada estado: vamos a asociar condiciones a *grupos de estados* más que a estados individuales. De esta manera aumentaremos el grado de abstracción en la etapa inicial de diseño, haciendo posible en consecuencia atacar problemas más complejos con menos posibilidades de equivocarse.

Este método consiste en identificar inicialmente condiciones asociadas al enunciado del problema, aunque éstas no sean suficientemente específicas para asociarse a estados individuales.

Describiremos este método mediante su aplicación a un ejemplo particular: Diseñar un AFD que acepte las palabras del lenguaje en $\{0, 1\}$ donde las palabras no contienen la



(a) Grupos de estados (b) Detalle de estados

Figura 2.9: Diseño de AFD por grupos de estados

subcadena 11 pero sí 00.

Inmediatamente a partir del enunciado identificamos las siguientes situaciones:

- Las letras consumidas hasta el momento no contienen ni 00 ni 11.
- Contienen 00 pero no 11
- Contienen 11.

Estas condiciones cumplen dos requisitos que siempre se deben cumplir en este tipo de diseños:

- Las condiciones deben ser *excluyentes*, lo que quiere decir que no deben poder ser ciertas dos o más al mismo tiempo.
- Las condiciones deben ser *comprensivas*, lo que quiere decir que no faltan casos por considerar.

Los grupos de estados, así como las transiciones que provocan que se pase de uno a otro, se representan como "nubes" en la figura 2.9(a). En dicha figura también se ilustran unas nubes "dobles" para indicar que son condiciones finales –en este ejemplo, la condición "Contienen 00 pero no 11"–, así como la condición inicial con un símbolo ">".

Estos diagramas no son aún AFD, pero casi. Lo que falta por hacer es refinar cada grupo de estados, considerando lo que ocurre al recibir cada uno de los posibles caracteres de entrada. La forma en que se subdivide cada grupo de estados ("nube") en estados individuales se detalla a continuación:

- Las letras consumidas hasta el momento no contienen ni 00 ni 11.
  1. Inicial, no se han recibido caracteres.
  2. Se acaba de recibir un 0.
  3. Se acaba de recibir un 1.
- Contienen 00 pero no 11.
  1. Se acaba de recibir un 0.
  2. Se acaba de recibir un 1.
- Contienen 11 (no hay subcondiciones).

Esto nos da un total de 6 estados, cada uno de los cuales tiene una condición muy específica asociada (son los estados "A" a "F" en la figura 2.9(b)). El siguiente paso es hacer el diseño detallado de las transiciones, lo que por experiencia consideramos que es relativamente fácil para cualquier alumno. El resultado se muestra en la figura 2.9(b). En este diagrama se puede notar que los estados de una nube "final" son también finales; esto debe verificarse siempre.

Hacemos notar que en este ejemplo en particular, encontrar directamente las condiciones asociadas a los estados puede ser algo difícil; por ejemplo, encontrar directamente la condición "Las letras consumidas hasta el momento no contienen ni 00 ni 11 y se ha recibido un 0" (estado "B" en la figura 2.9(b)) requeriría ciertamente más inventiva de la que tenemos derecho a presuponer en el lector. En este sentido el diseñar primero los grupos de estados permite manejar la complejidad del problema de manera más modular y gradual.

En cualquier caso, ya sea que se encuentren directamente las condiciones para cada estado, o primero para grupos de estados, consideramos importante que primero se determinen los estados con sus condiciones asociadas, *y solamente después* se tracen las transiciones, en vez de ir proponiendo sin ningún orden los estados y las transiciones a la vez, lo que muy frecuentemente conduce a errores.

2.4.2. Diseño de AFD por complemento

En ocasiones, para un cierto lenguaje L, es más sencillo encontrar un AFD para el lenguaje exactamente contrario –técnicamente hablando, complementario $L^c = \Sigma^* - L$. En estos casos, una solución sencilla es hallar primero un AFD para L^c, y luego hacer una transformación sencilla para obtener el autómata que acepta L.

Si $M = (K, \Sigma, \delta, s, F)$ es un autómata *determinista* que acepta un lenguaje regular L, para construir un autómata M^c que acepte el lenguaje complemento de L, esto es, $\Sigma^* - L$, 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. [8]

Ejemplo.- Obtener un AF para el lenguaje en $\{a, b\}^*$ de las palabras que no contienen la cadena "*abaab*".

Solución.- Primero obtenemos un AFD M_1 para el lenguaje cuyas palabras sí contienen la cadena "*abaab*". Diseñamos M_1 sistemáticamente usando grupos de estados, uno que recuerda que la palabra no contiene aun *abaab* y otro que recuerda que ya se reconoció dicha cadena, como aparece en la figura 2.10(a). Luego detallamos cada uno de estos grupos de estados, introduciendo estados individuales que recuerdan lo que se lleva reconocido de la cadena *abaab*, como se muestra en la figura 2.10(b) –el grupo de estados que recuerda que ya se reconoció la cadena *abaab* tiene un sólo estado, pues no hay condiciones adicionales que recordar. Finalmente, la solución será un AFD donde cambiamos los estados finales por no finales y viceversa en M_1, como se muestra en 2.10(c).

Desde luego, el ejemplo descrito es muy sencillo, pero luego veremos otras herramientas que se pueden usar en combinación con la obtención del complemento de un AF, para resolver en forma sistemática y flexible problemas de diseño aparentemente muy difíciles.

2.5. Equivalencia de autómatas finitos.

Decimos que dos autómatas que aceptan el 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[9] que acepten un mismo lenguaje? La respuesta es afirmativa, y una prueba consiste en exhibir un ejemplo.

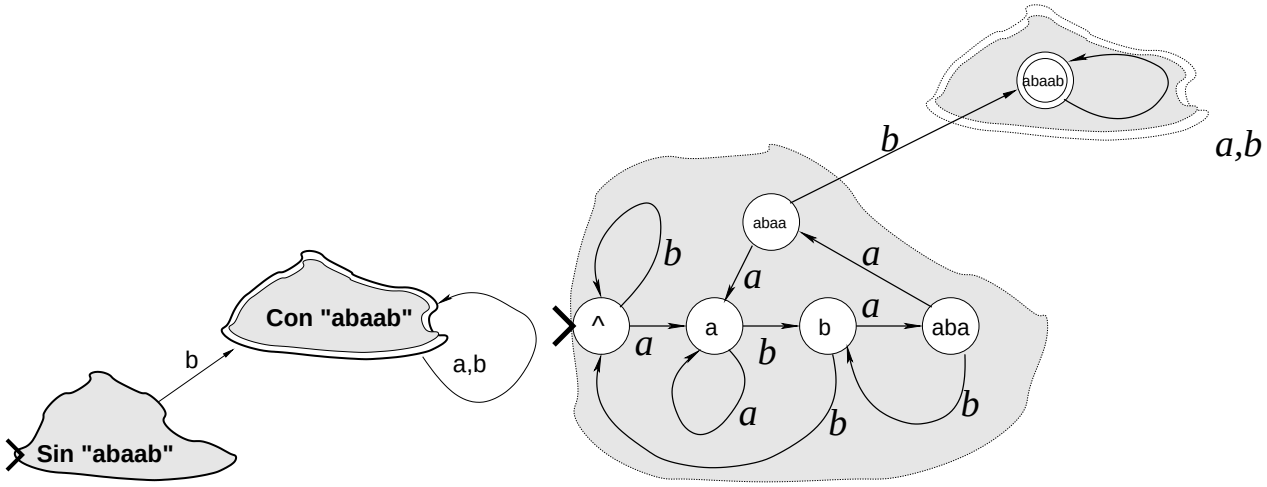
Por ejemplo, los autómatas (a) y (b) de la figura 2.11 aceptan ambos el lenguaje a^*.

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

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

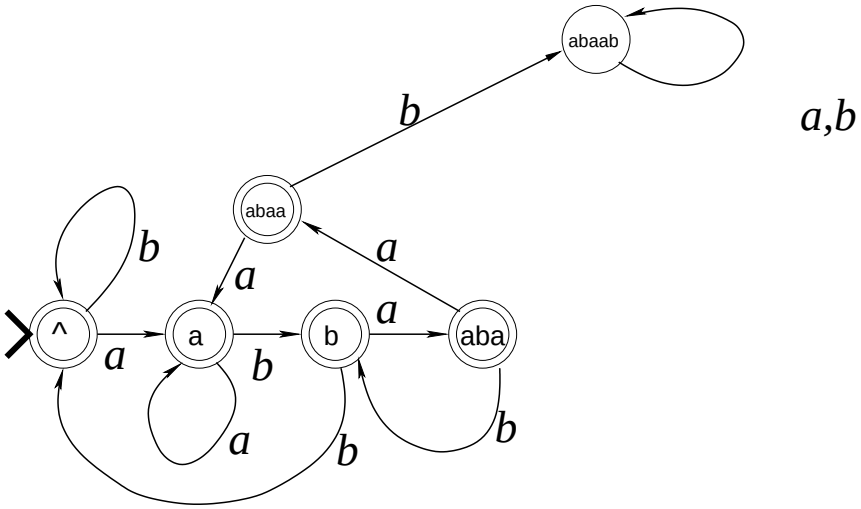
[8] Es *muy importante* notar que el método de diseño por complemento sólo se aplica a los autómatas deterministas, y no a los llamados "no deterministas", que veremos luego.

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



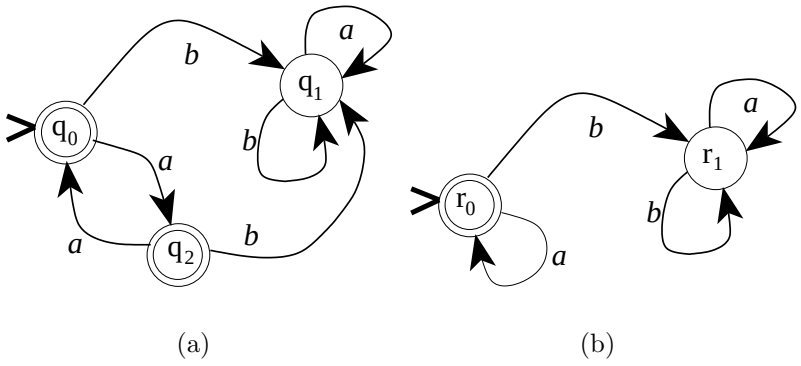
(a)

(b)



(c)

Figura 2.10: Diseño del AF para palabras sin *abaab*



(a) (b)

Figura 2.11: Autómatas equivalentes

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.

El algoritmo mencionado en el teorema de Moore consiste en la construcción de un *árbol de comparación de autómatas*. Este árbol 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 pensó en explorar todas las posibles combinaciones de estados mediante un árbol. Dicho árbol de comparación se construye de la manera siguiente, para dos autómatas $M = (K, \Sigma, \delta, s, F)$ y $M' = (K', \Sigma', \delta', s', F')$:

1. Inicialmente la raíz del árbol es el par ordenado (s, s') que contiene los estados iniciales de M y M' respectivamente;

2. Si en el árbol hay un par (r, r'), para cada caracter en Σ se añaden como hijos suyos los pares (r_σ, r'_σ) donde $r_\sigma = \delta(r, \sigma)$, $r'_\sigma = \delta(r', \sigma)$, si no estén ya.

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

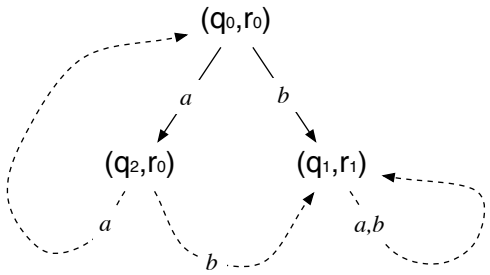
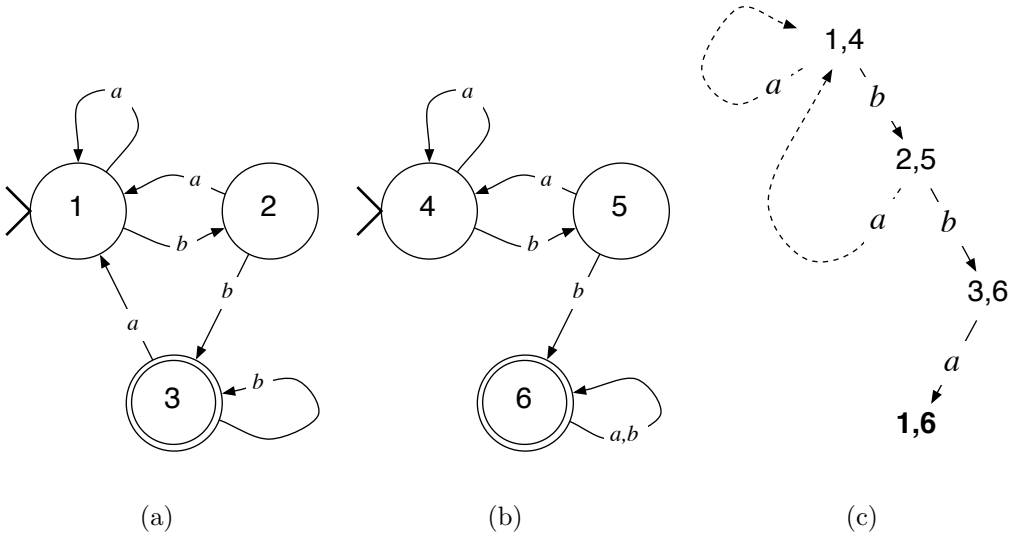


Figura 2.12: Árbol de comparación de AF



(a) (b) (c)

Figura 2.13: AFDs no equivalentes

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

Ejemplo.- Sean los autómatas M y M' de la figuras 2.11(a) y (b) respectivamente. El árbol de comparación se muestra en la figura 2.12. En dicho árbol se muestran adicionalmente, con línea punteada, las ramas que van a nodos ya existentes, como la que va de (q_2, r_0) a (q_0, r_0). Estas ramas con línas punteada no son, estrictamente hablando, parte del árbol, pero pensamos que mejoran la comprensión del diagrama.

Se concluye que M y M' son equivalentes.

En el caso de que los autómatas que se comparan no sean equivalentes, la construcción del árbol de comparación permite encontrar al menos una palabra en que los lenguajes aceptados por ellos difieren. Considérense, por ejemplo, los autómatas de las figuras 2.13 (a) y (b). Una parte del árbol de comparación se muestra en la figura 2.13(c), hasta donde se encuentra el

primer par de estados incompatibles.

Analizando el árbol de la figura 2.13(c), vemos que para llegar desde la raíz del árbol hasta el par incompatible (1,6), hay que gastar los caracteres b, b y a, esto es, la palabra bba. Así llegamos a la conclusión de que el autómata de la figura 2.13(a) no acepta la palabra bba, mientras que el de la figura 2.13(b) sí la acepta, y por lo tanto sus lenguajes aceptados difieren al menos en la palabra bba.

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 del árbol siempre termina (no se "cicla")

2. Si en el árbol 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 el árbol aparecerán estados incompatibles.

El punto 1 se prueba fácilmente porque, los nodos del árbol siendo todos distintos, son un subconjunto de $K \times K'$, que es finito, por lo que el árbol no puede extenderse indefinidamente.

Para probar el punto 2 basta con recorrer en el árbol la trayectoria que lleva al par de estados incompatibles, (r, r'), $r \in F$, $r' \notin F'$. Simplemente concatenamos los caracteres de entrada σ en dicha trayectoria, y obtendremos una palabra $w$tal que si la aplicamos 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 en el árbol, caracter por caracter, debemos llegar a un par incompatible. [10]

Por otra parte, el punto 3 implica que si no hay pares incompatibles en el árbol, 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.

[10] Reflexione porqué se está seguro de que es posible seguir w sobre el árbol, caracter por caracter. ¿No podría "atorarse" el proceso?.

2.6. Simplificación de Autómatas finitos

Una de las mejores cualidades de los AFD es que existen métodos mecánicos para simplificarlos, hasta poder llegar al AFD más sencillo posible para un lenguaje dado.

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. [11]

Como un primer ejemplo, considérense los AFD de las figuras 2.11 (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 a's, 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 2.11(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:

Lema: Dado un AFD $M = (K, \Sigma, \delta, q, F)$ y dos estados $q_1, q_2 \in K$, tendremos que $q_1 \approx q_2$ ssi $(K, \Sigma, \delta, q_1, F) \approx (K, \Sigma, \delta, q_2, F)$. [12]

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 2.11(a), para verificar si $q_0 \approx q_2$, habría que comparar

[11] El hecho de que para todo lenguaje regular existe un AFD mínimo, es un hecho para nada evidente, que rebasa los alcances de este libro. Esto se discute en la referencia [7].

[12] No damos la prueba, ver sección de ejercicios.

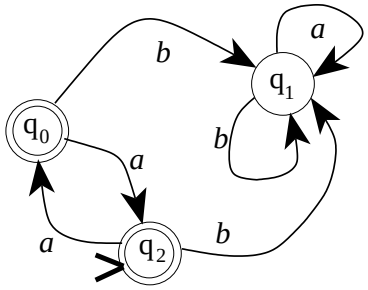


Figura 2.14: Cambio de estado inicial

dicho AFD con el de la figura 2.14, en el que se cambió el estado inicial por el otro estado que se quiere comparar. En este ejemplo, dicha comparación de AFDs da un resultado de equivalencia, por lo que se concluye que los estados son redundantes.

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 2.11(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 2.11(b).

La eliminación de estados redundantes de un AFD es una manera de simplificar AFDs, y puede usarse iteradamente para simplificar al mínimo. Sin embargo, el trabajo que implica es mucho, y para AFDs grandes, examinar cada par de estados es poco práctico.

Vamos, en consecuencia, a examinar métodos más organizados para localizar los estados redundantes y minimizar los AFDs.

2.6.1. Tabla de estados distinguibles

Vamos a definir la noción de estados *distinguibles*, que intuitivamente quiere decir que si dos estados son distinguibles, ya no pueden ser equivalentes. La definición es inductiva:

- Los estados p y q son distinguibles si son incompatibles (es decir, uno es final y el otro no final). Esta es la base de la inducción.

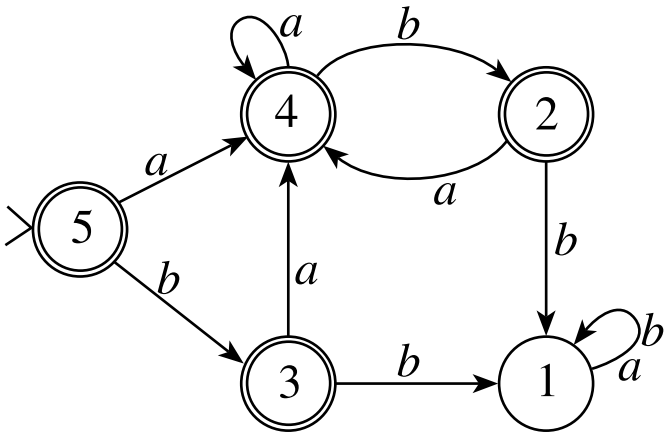


Figura 2.15: AFD a simplificar

- Si tenemos transiciones $\delta(p_0, \sigma) = p$ y $\delta(q_0, \sigma) = q$ donde p y q son distinguibles, entonces también p_0 y q_0 son distinguibles. Este es el paso inductivo.

Por ejemplo, considérese el AFD de la figura 2.15. Claramente los estados 1 y 3 son distinguibles, porque no son compatibles. Puede ser menos obvio ver que los estados 4 y 3 son distinguibles, pero podemos ver que, aunque ambos son finales, el caracter b nos lleva de 4 a 2, y similarmente de 3 a 1, y vemos que 2 y 1 son distinguibles al no ser compatibles.

En ocasiones se requieren varios pasos intermedios para determinar que un par de estados es distinguible (esto no ocurre en el ejemplo recién visto).

Teorema.- Dos estados son equivalentes (o "redundantes") ssi no son distinguibles. Este resultado se prueba en la referencia [7]. Su utilidad estriba en que es relativamente sencillo verificar si dos estados son distinguibles.

Una manera de organizar el trabajo de verificar qué pares de estados de un AFD son distinguibles, consiste en construir una tabla en que los renglones y las columnas son los nombres de los estados, y en cada cruce de renglón con columna se indica con una $\times$ cuando son distinguibles.

Por ejemplo, para el AFD de la figura 2.15, empezamos con la tabla vacía de la figura 2.16(a). Obsérvese que en la tabla se omite la diagonal principal, pues no tiene caso confrontar cada estado contra sí mismo. En la tabla 2.16(b) se aprecian signos "$\times$" en las celdas (2,1), (3,1), (4,1) y (5,1) que se obtienen directamente del hecho de que son pares de estados incompatibles –por lo tanto distinguibles. En la figura 2.16(c) se ha agregado una marca en la casilla (4,2), que viene del hecho de que con el caracter b las transiciones nos llevan de 2 a 1, y de 4 a 2, pero el par (2,1) ya estaba marcado como distinguible. Finalmente, en la tabla 2.16(d) se pusieron marcas en (4,3), (5,2) y (5,3), haciendo análisis similares. Es fácil convencerse de que no hay forma de hacer distinguibles los pares (3,2) y (5,4), los cuales, de

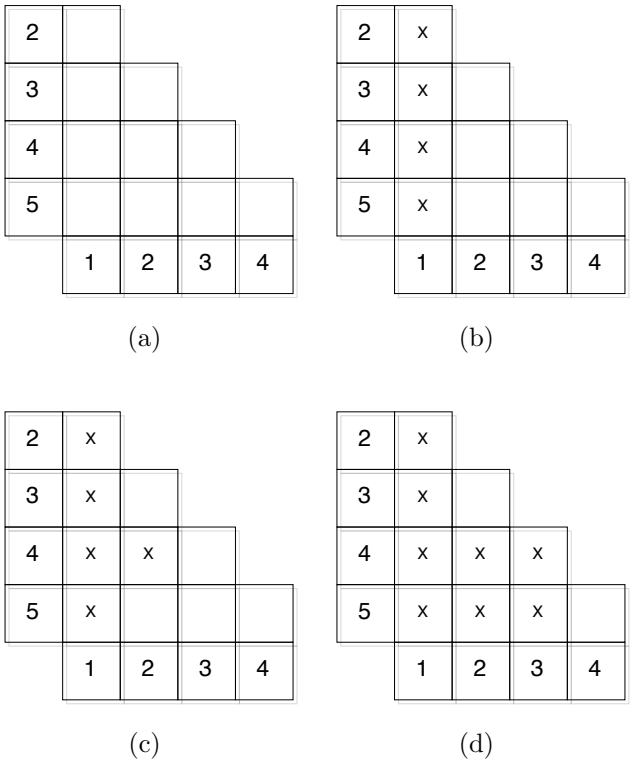


Figura 2.16: Tabla de estados distinguibles

acuerdo con el teorema presentado, son pares de estados equivalentes.

Una vez que detectamos los pares de estados equivalentes, podemos proceder a eliminar uno de ellos, de la forma que hemos visto. En el ejemplo de la figura 2.16(d), como hay dos pares de estados redundantes, el AFD mínimo tiene 3 estados.

En autómatas grandes, el procedimiento puede volverse algo complicado, pues es necesario examinar repetidamente cada celda de la tabla para verificar que los cuadros aún no marcados siguen sin ser distinguibles, hasta que en una de las iteraciones ya no se agregue ninguna marca a la tabla.

2.6.2. Simplificación por clases de equivalencia

Existe otro método de simplificación de estados, de hecho más comúnmente usado que el que hemos presentado, debido a que permite organizar más sistemáticamente el trabajo.

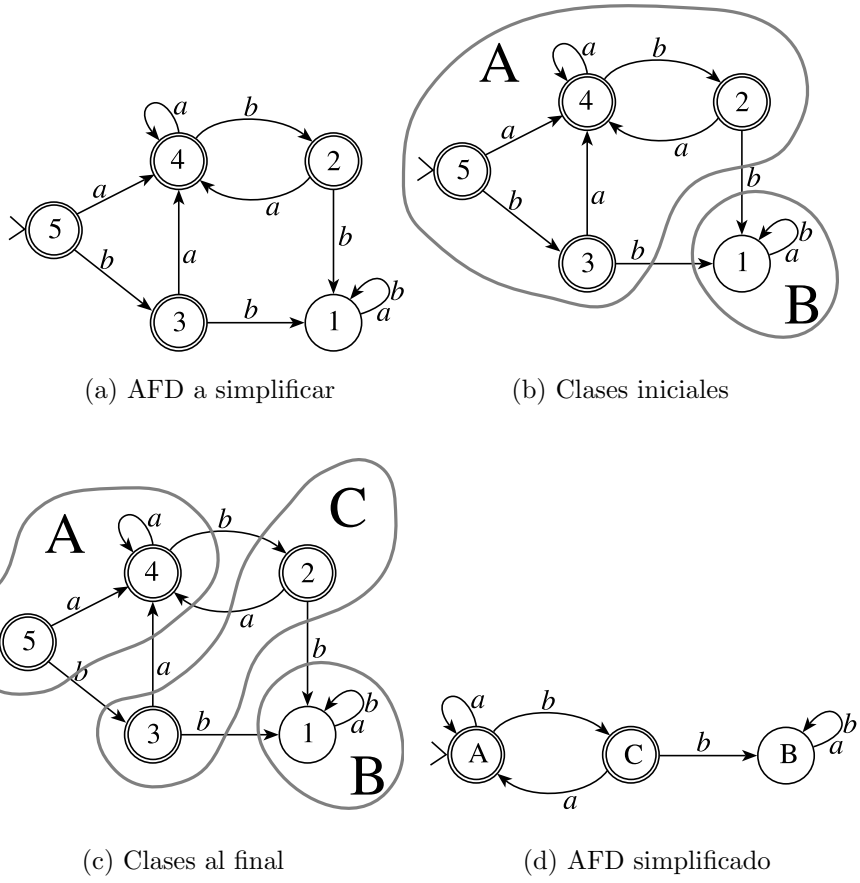
Este algoritmo, que llamaremos "simplificación por clases de equivalencia", sigue un orden de operaciones inverso a la eliminación gradual de estados redundantes que hemos visto antes: en vez de ir reduciendo el número de estados, comienza con grupos de estados,

o "clases", que se van dividiendo en clases más pequeñas, hasta que el proceso de división ya no pueda continuarse.

La idea es formar clases de estados de un autómata que, hasta donde se sabe en ese momento, podrían ser equivalentes. Sin embargo, al examinar las transiciones de varios estados de una misma clase, puede a veces inferirse que después de todo no deben permanecer en la misma clase. En ese momento la clase en consideración se "divide". Luego se examinan las transiciones de las clases que se formaron, a ver si es necesario dividirlas nuevamente, y así en adelante, hasta que no se halle evidencia que obligue a dividir ninguna clase.

Al terminar el proceso de división de clases, cada una de las clases representa un estado del autómata simplificado. Las transiciones del autómata simplificado se forman a partir de las transiciones de los estados contenidos en cada clase.

Antes de formalizar el proceso, vamos a explicarlo con ayuda de un ejemplo.



(a) AFD a simplificar (b) Clases iniciales

(c) Clases al final (d) AFD simplificado

Figura 2.17: Simplificación por clases de equivalencia

Ejemplo.- Considérese el AFD de la figura 2.17(a). Las primeras dos clases de equivalencia que se forman contienen, respectivamente, a los estados finales y a los estados no finales,

los cuales evidentemente no podrían ser equivalentes (esto es, estar en una sola clase de equivalencia [13]). Estas dos clases se encuentran indicadas en la figura 2.17(b).

Ahora vamos a examinar si todos los estados de cada clase tienen transiciones "similares", lo que en nuestro caso quiere decir que van a una misma clase de equivalencia. Por ejemplo, tomemos los estados 3 y 4 de 2.17(b). Al recibir el símbolo a, desde 3 nos vamos a la clase $\{2, 3, 4, 5\}$, lo que también ocurre desde el estado 4. Hasta aquí 3 y 4 se comportan similarmente. Ahora examinamos las transiciones con b: desde 3 nos iríamos a la clase $\{1\}$, mientras que desde 4 iríamos a la clase $\{2, 3, 4, 5\}$. Concluímos que 3 y 4 no pueden coexistir en una misma clase de equivalencia, por lo que la clase $\{2, 3, 4, 5\}$ debe dividirse. Haciendo el mismo análisis con los demás estados, dividimos $\{2, 3, 4, 5\}$ en $\{2, 3\}$ y $\{4, 5\}$, como aparece en la figura 2.17(c). En este punto ya no es posible dividir alguna de las 3 clases existentes, pues las transiciones de sus estados son "similares". Concluimos que estas son las clases de equivalencia más finas que pueden formarse.

Tomando las clases de equivalencia de 2.17(c) como estados, formamos el AFD que aparece en 2.17(d). Obsérvese que las transiciones de 2.17(d) son las de cualquiera de los estados contenidos en cada clase; simplemente registramos a qué clase de equivalencia se llega con cada símbolo de entrada. El estado inicial corresponde a la clase de equivalencia que contenga el antiguo estado inicial, y los estados finales del nuevo AFD vienen de las clases de equivalencia que contienen estados finales del antiguo AFD.

Formalmente, el procedimiento es como sigue, para un AFD $(K, \Sigma, \delta, s, F)$:

1. Inicialmente se tienen las clases F y $K - F$
2. Repetir para cada clase:
   - Sea q un estado de la clase. Para cada uno de los otros estados, q', verificar si $\delta(q, \sigma)$ va a dar a la misma clase de equivalencia que $\delta(q', \sigma)$, para cada caracter σ.
   - Si la respuesta es sí, la clase no necesita dividirse.
   - Si la respuesta es no, dividir la clase en dos subclases: la que agrupa a los estados que tuvieron transiciones "similares" a q, y la de los estados con transiciones "diferentes" a q (que no van a dar a la misma clase de equivalencia con un mismo símbolo σ).

Por ejemplo, consideremos la clase $\{2, 3, 4, 5\}$ de la figura 2.17(b). Tomando como referencia al estado 2, nos damos cuenta de que el estado 3 tiene transiciones similares (con a a la clase $\{2, 3, 4, 5\}$, con b a la clase $\{1\}$), mientras que los estados 4 y 5 tienen transiciones diferentes a las de 2 (con a y con b van a la clase $\{2, 3, 4, 5\}$); esto ocasiona que la clase $\{2, 3, 4, 5\}$ se parta en dos. Luego habría que examinar las nuevas clases, $\{1\}$, $\{2, 3\}$ y $\{4, 5\}$; en este caso sucede que ya no se necesita dividir ninguna de ellas.

[13] ¿Porqué?

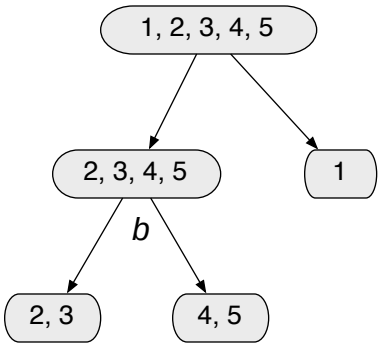


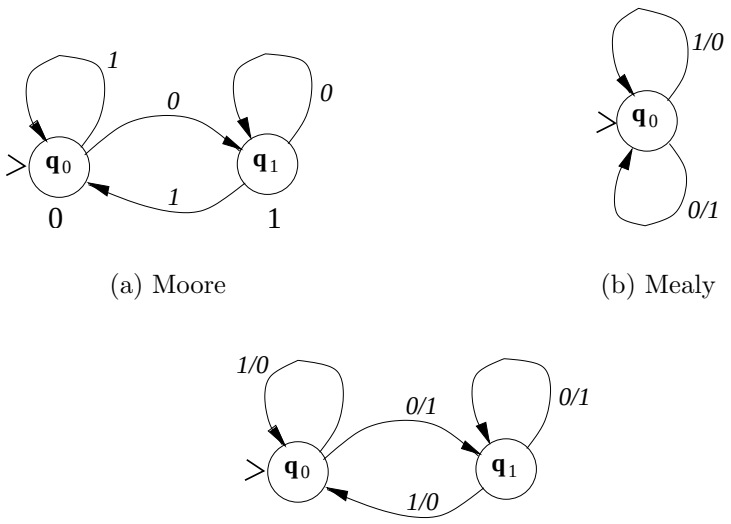
Figura 2.18: Clases de equivalencia organizadas en árbol

En la práctica, en vez de trazar líneas sobre el diagrama de estados, es conveniente organizar la información de las clases de equivalencia en árboles, en donde cada nodo contiene los estados de una clase de equivalencia. Inicialmente están todos los estados del AFD en una clase, como en la raíz del árbol en la figura 2.18, para el AFD de la figura 2.17(a), e inmediatamente se dividen en finales y en no finales, como en el seguiente nivel en esa misma figura. Luego, para el nodo $\{2, 3, 4, 5\}$ examinamos si las transiciones con los caracteres de entrada, en este caso a y b, llevan a las mismas clases, y verificamos que en el caso de b los estados 2 y 3 van a un no final, mientras que 4 y 5 van a un final, por lo que ese nodo se divide en dos, como se aprecia en el tercer nivel de la figura. Ahí también se puede apreciar un símbolo b bajo el nodo $\{2, 3, 4, 5\}$, indicando a causa de qué caracter la clase de equivalencia se dividió. Examinando las transiciones en las clases de equivalencia que quedan en las hojas del árbol, vemos que ya no hay razón para dividirlas más. Finalmente, las clases de equivalencia resultantes son $\{1\}$, $\{2, 3\}$ y $\{4, 5\}$, que corresponden a los 3 estados que tendrá el AFD minimizado.

2.7. 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, [14] 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, según si la salida depende 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*, propuestos respectivamente por G. Mealy [13] y E. Moore [15].

[14] Haremos una breve descripción de los compiladores en la sección 5.6.



(a) Moore

(b) Mealy

(c) Moore transformado en Mealy

Figura 2.19: Autómatas de Moore y Mealy

2.7.1. Máquinas de Moore

En las máquinas de Moore la salida depende del estado en que se encuentra el autómata. Dicha salida es producida una vez, y cuando se llega a otro estado (o al mismo) por efecto de una transición, se produce el símbolo de salida asociado al estado al que se llega. Algunos estudiantes encuentran útil la analogía de los autómatas de Moore con nociones de electricidad: es como si cada estado tuviera un "nivel de voltaje" que se produce en la salida mientras el control se encuentre en dicho estado.

Las máquinas de Moore se representan gráficamente como cualquier AFD, al que se añade, al lado de cada estado, la salida asociada, que es una cadena de caracteres. Por ejemplo, consideremos un autómata que invierte la entrada binaria recibida (esto es, cambia un 1 por 0 y un 0 por 1). Dicho autómata se representa gráficamente en la figura 2.19(a).

Para formalizar los autómatas de Moore una idea sencilla es añadir a un AFD estándar una función que asocie a cada estado una palabra de salida; llamaremos λ a esta función. También vamos a agregar un alfabeto de salida Γ, que puede ser distinto al de entrada. Todos los demás aspectos permanecen igual que en un AFD.

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 que 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 es una cadena de caracteres tomados de Γ.

Ejemplo.- La siguiente máquina de Moore formaliza el diagrama de la figura 2.19(a):

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

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

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 $q_i = \delta(q_{i-1}, a_i), a_i \in \Sigma$, para $1 \leq i \leq n$.

2.7.2. Máquinas de Mealy

En las máquinas de Mealy la salida producida depende *de la transición* que se ejecuta, y no solamente del estado. Por esto, en la notación gráfica las etiquetas de las flechas son de la forma σ/w, donde σ es el caracter que se consume de entrada, y w es la palabra que se produce en la salida. Por ejemplo, el diagrama para el inversor binario, implementado como máquina de Mealy, se presenta en la figura 2.19(b).

Para formalizar las máquinas de Mealy, una idea podría ser aumentarle a las transiciones la palabra producida en la salida. Sin embargo, por modularidad se prefiere definir una función de salida λ, pero que, a diferencia de las máquinas de Moore, ahora toma como entrada *un estado y un caracter de entrada*. En efecto, podemos darnos cuenta de que es lo mismo que la salida dependa del estado y un caracter, a que dependa de una transición. [15]

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 $\lambda : K \times \Sigma \to \Gamma^*$, esto es, toma un elemento de $K \times \Sigma$ –que incluye un estado y un caracter de entrada– y produce una palabra formada por caracteres de Γ.

Ejemplo.- El inversor de Mealy de la figura 2.19(b) se puede representar formalmente de la siguiente forma:

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

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

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.

Los criterios para diseñar tanto máquinas de Moore como de Mealy son básicamente los mismos que para cualquier otro AFD, por lo que no presentaremos aquí métodos especiales

[15] Esto suponiendo que no hay varias transiciones distintas entre dos mismos estados.

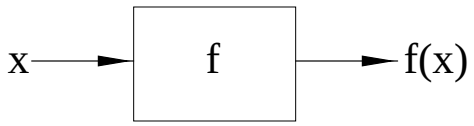


Figura 2.20: Función como "caja negra"

de diseño.

2.7.3. 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.

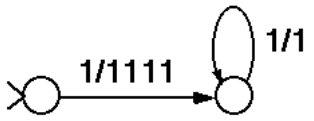
La transformación de una máquina de Moore en máquina de Mealy es trivial, pues hacemos $\lambda_{Mealy}(q, a) = \lambda_{Moore}(\delta_{Moore}(q, a))$, es decir, simplemente obtenemos qué salida producirá una transición de Mealy viendo la salida del estado al que lleva dicha transición en Moore. Por ejemplo, la máquina de Mealy de la figura 2.19(b) se puede transformar de esta manera a la máquina de Moore que aparece en la figura 2.19(c).

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 [7].

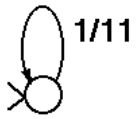
2.7.4. Cálculo de funciones en AF

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 2.20, 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 2.21(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 entrega un caracter en la salida, hasta que se acabe la entrada. Debe quedar claro que los tres caracteres que le saca de ventaja la salida al primer caracter de entrada se conservan hasta el final de 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 $x + 3$.

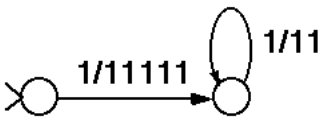


(a) $f(x) = x + 3$

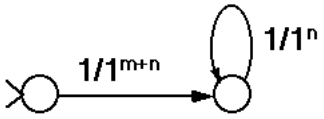


(b) Función $f(x) = 2x$

Figura 2.21: Funciones aritméticas en Mealy



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



(b) $f(x) = nx + m$

Figura 2.22: Funciones lineales en Mealy

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 2.21(b) entrega una salida que es la entrada multiplicada por dos. Aun más, el AF de la figura 2.22(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 2.22(b) muestra cómo calcular una función $f(x) = nx + m$.

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 tripleta

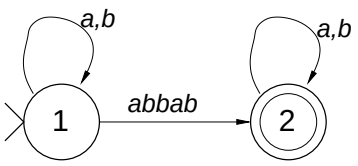


Figura 2.23: AFN para palabras que contienen *abbab*

$[[q, \alpha, \beta]] \in K \times \Sigma^* \times \Gamma^*$, 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 $q = \delta(p, \sigma)$ y $\xi = \lambda(q, \sigma)$.

Por ejemplo, dado el AF de Mealy de la figura 2.19(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 2.19(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 $f(1101) = 0010$.

2.8. 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. Inclusive se permite que las transiciones tengan como etiqueta palabras de varias letras o hasta la palabra vacía. A estos autómatas finitos se les llama *no determinísticos* o *no deterministas* (abreviado AFN), 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, un AFN que acepte

las palabras en $\{a, b\}$ que contienen la subcadena *abbab* se ilustra en la figura 2.23.

Hacemos notar en este punto que, dado que los AFN tienen menos restricciones que los AFD, resulta que los AFD son un *caso particular* de los AFN, por lo que todo AFD es de hecho un AFN. [16]

Hasta aquí sólo vemos ventajas de los AFN sobre los AFD. Sin embargo, en los autómatas no determinísticos se presenta una dificultad para poder saber qué camino tomar a partir de un estado dado cuando se presenta un símbolo, pues puede haber varias opciones. Por ejemplo, tomando el autómata de la figura 2.23, si se nos presenta una palabra como *abbaba*, no sabremos si tomar la transición del estado 1 al 2, gastando *abbab*, y ya en 2 gastar a, o bien gastar en 1 todas las letras de la palabra de entrada, siguiendo las transiciones de 1 a sí mismo. El problema en este ejemplo es particularmente grave porque en uno de los casos se llega a un estado final y en el otro no. Veremos más adelante cómo enfrentar este tipo de situaciones.

Además, puede ocurrir que, estando en un nodo n, y habiendo un símbolo de entrada a, no exista ninguna flecha que salga de n con etiqueta a (esto no ocurre en el ejemplo de la figura 2.23).

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

2.8.1. Representación formal de los AFN

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 $K \times \Sigma^* \times K$.

Por ejemplo, el AFN de la figura 2.23 quedaría representado matemáticamente por el siguiente quíntuplo:

$$(\{1, 2\}, \{a, b\}, \{(1, a, 1), (1, b, 1), (1, abbab, 2), (2, a, 2), (2, b, 2)\}, 1, \{2\})$$

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 $(q_1, w, q_2) \in \Delta$, que es una transición 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. [17]

Vamos a definir la noción de palabra aceptada en términos de la representación gráfica

[16] Sin embargo, la representación formal de los AFN no es idéntica a la de los AFD.

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

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. [18]

Ejemplo.- Verificar si la palabra *baabbaba* es aceptada por el AFN de la figura 2.23. Solución: La palabra *baabbaba* puede ser dividida en cuatro pedazos, $p_1 = b$, $p_2 = a$, $p_3 = abbab$, y $p_4 = a$, cuya concatenación produce la palabra original. Ahora bien, podemos seguir la siguiente secuencia de estados (trayectoria) en el AFN dado:

Estado	**Cadena que consume**	**Produce estado**
1	b	1
1	a	1
1	$abbab$	2
2	a	2

Así probamos que la cadena *baabbaba* 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 *ababab* no es aceptada por el autómata, pues la transición que liga el estado inicial 1 con el final 2 incluye dos b's seguidas, que no hay en la palabra dada, por lo que no es posible llegar al estado final y la palabra no podrá ser aceptada.

2.8.2. Diseño de AFN

Como sugerimos al inicio de esta sección, en los AFN es posible aplicar métodos modulares de diseño, que permiten manejar mejor la complejidad de los problemas. Son estos métodos modulares los que describiremos en esta sección. [19]

AFN para la unión de lenguajes

Si ya contamos con dos AFN, sean M_1 y M_2, es posible combinarlos para hacer un nuevo AFN que acepte la unión de los lenguajes que ambos autómatas aceptaban.

[18] Se puede expresar la definición de palabra aceptada en términos de la noción de configuración (ver ejercicios).

[19] En muchos libros estos temas se agrupan, desde un punto de vista más matemático que ingenieril, en una sección de "propiedades de los lenguajes regulares", pero nosotros hemos preferido aplicarlos directamente a mejorar las habilidades de diseño de AFN de los alumnos de computación.

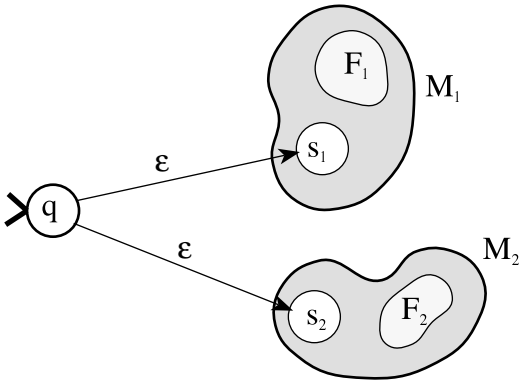


Figura 2.24: AFN para la unión de dos lenguajes

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. [20] Podemos entonces construir un AFN M_3 que acepte $L_1 \cup L_2$ de la 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 similarmente a M_2 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 2.24 se representa gráficamente M_3.

Ejemplo.- Diseñar un autómata no determinista que acepte las palabras sobre $\{a, b\}$ que tengan un número par de a o que terminen en bb.

Solución.- En la figura 2.25(a) se presenta un AFN que acepta las palabras que contienen un número par de a's, y en 2.25(b) otro que acepta las palabras que terminan en bb. Finalmente, en 2.25(c) está el AFN que acepta el lenguaje dado.

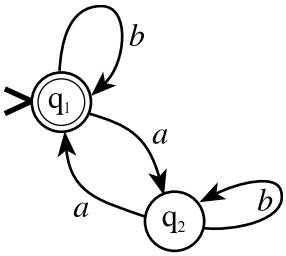
AFN para la concatenación de lenguajes

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 respectivamente. Podemos entonces construir un AFN M_3 que acepte $L_1 L_2$ 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.

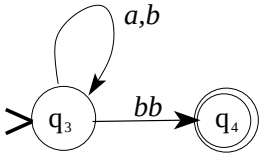
Formalmente $M_3 = (K_1 \cup K_2,\ \Sigma_1 \cup \Sigma_2,\ \Delta_1 \cup \Delta_2 \cup \{(p, \varepsilon, s_2) | p \in F_1\},\ s_1,\ F_2)$

El funcionamiento de M_3 es como sigue: cuando se recibe una palabra $w = w_1 w_2$, $w_1 \in L_1$, $w_2 \in L_2$, entonces se empieza procesando w_1 exactamente como lo haría M_1, hasta llegar

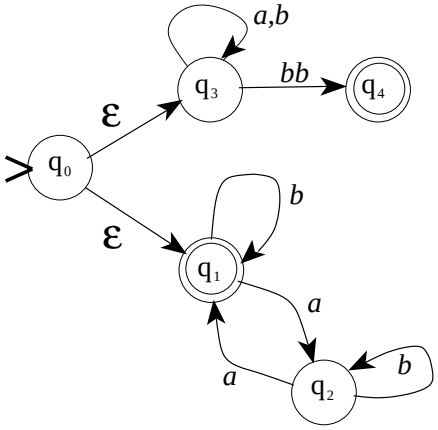
[20] Sin pérdida de generalidad podemos suponer que K_1 y K_2 son disjuntos.



(a) Número par de a



(b) Terminan en bb



(c) Combinación de los dos

Figura 2.25: Combinación de AFNs

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 2.26 se representa M_3.

Ejemplo.- Construir un AFN que acepte el lenguaje en $\{a, b\}$ donde las a's vienen en grupos de al menos dos seguidas, y los grupos de a's que son repeticiones de aaa están a la derecha de los que son repeticiones de aa, como en $baabaaa$, aaa, $baab$ o $baaaaa$. Esta condición no se cumple, por ejemplo, en $bbaaabaa$ ni en $aaabaaaa$.

Solución.- Un AFN, ilustrado en la figura 2.27(a), acepta palabras que contienen b's y grupos de aa en cualquier orden. Otro AFN –figura 2.27(b)– acepta un lenguaje similar, pero con grupos de aaa. La solución es su concatenación, que se presenta en la figura 2.27(c).

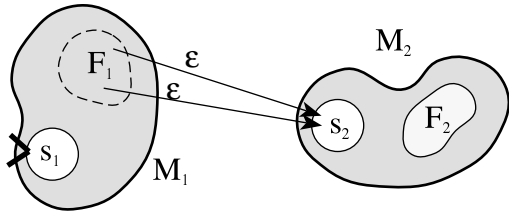


Figura 2.26: AFN para la concatenación de dos lenguajes

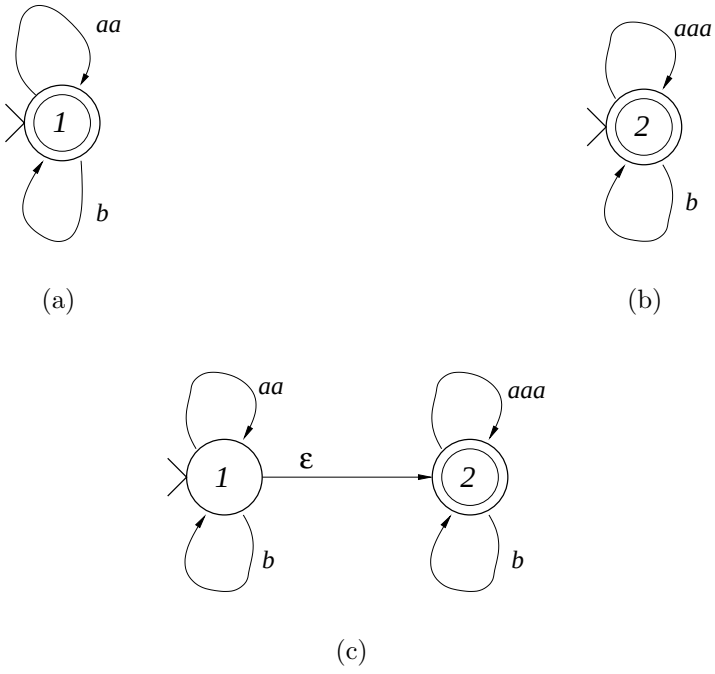


Figura 2.27: Concatenación de dos AFN

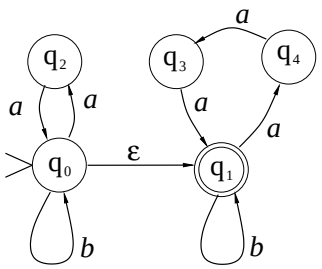


Figura 2.28: AFN a transformar en AFD

2.8.3. Equivalencia de AFD Y AFN

Los autómatas finitos determinísticos (AFD) son un subconjunto propio de los no determinísticos (AFN), lo que quiere decir que todo AFD es un AFN. [21] 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 idea de considerar el conjunto de estados en los que podría encontrarse el AFN al haber consumido una cierta entrada.

El método de los conjuntos de estados

Dado un AFN M, consideremos la idea de mantener un conjunto de estados Q_i en los que sería posible estar en cada momento al ir consumiendo las letras de una palabra de entrada.

Por ejemplo, considérese el AFN de la figura 2.28. Queremos analizar qué sucede cuando este AFN recibe 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" ningún caracter podría estar también en el estado q_1, o sea que el proceso arranca con el conjunto de estados $Q_0 = \{q_0, q_1\}$. 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 $Q_1 = \{q_0, q_1\}$. Y así en adelante. La tabla siguiente resume los conjuntos de estados por los que se va pasando para este ejemplo:

[21] Salvo por el hecho de que δ es una función y Δ una relación.

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 de entrada puede ser aceptada. Otra conclusión –mucho más útil que la anterior– es darse cuenta de que si consideramos a los conjuntos de estados Q_i como una especie de "mega-estados" de cierto autómata, entonces hemos estado en realidad siguiendo los pasos de ejecución de un AFD con "mega-estados".

Una vez que comprendemos lo anterior, nos damos cuenta de que, si en vez de considerar una palabra en particular, como fue *baaaaab*, consideramos cada posible caracter que puede llegar al estar en un "mega-estado", entonces podremos completar un AFD, que deberá ser equivalente al AFN dado. [22] 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 2.28. 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 2.28 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 2.29(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 2.29(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 cual se representa con un "lazo" a sí mismo en la figura 2.29(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 2.29(c).

Un detalle importante a observar en este procedimiento es que en ocasiones no hay estados adonde ir; por ejemplo, a partir del conjunto de estados $\{q_2, q_4\}$, con b no llegamos a ningún estado. En casos como éste, consideramos que habrá una transición con b a un nuevo conjunto de estados *vacío*, esto es $\{\}$, como se aprecia en la figura 2.29(c). Por supuesto, este estado vacío tendrá transiciones con a y con b a sí mismo.

[22] La equivalencia formal se discute más adelante.

Ahora tomemos una pausa y respiremos hondo. 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 construido un AFD!. Unicamente falta determinar cuáles de los nuevos estados son finales y cuáles 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 2.29(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.

Una transformación inofensiva

Cuando queremos aplicar el método descrito en los párrafos precedentes, una dificultad que puede presentarse es que algunas flechas del autómata tienen como etiquetas palabras de varias letras, y desde luego no podemos tomar "un pedazo" de una transición. Esta situación se aprecia en el AFN de la figura 2.30. En efecto, si a partir del estado inicial intentamos consumir la entrada "a", vemos que no hay una transición que permita hacerlo, aún cuando hay una transición (q_0, aa, q_1) cuya etiquete empieza con a.

Una solución a esta dificultad es normalizar a 1 como máximo 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 2.30, 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. [23]

Por ejemplo, para el AFN de la figura 2.30 se tiene el AFN transformado de la figura 2.28.

Formalización del algoritmo de conversión

Vamos ahora a precisar el método de conversión de AFN a AFD con suficiente detalle como para que su programación en computadora sea relativamente sencilla. Sin embargo, no vamos a describir el algoritmo en términos de ciclos, instrucciones de asignación, condicionales, etc., que son típicos de los programas imperativos. Más bien vamos a presentar un conjunto de definiciones que capturan los resultados intermedios en el proceso de conversión de AFN a AFD. Estas definiciones permiten programar en forma casi directa el algoritmo

[23] Probar que esta transformación preserva la equivalencia (ver ejercicios).

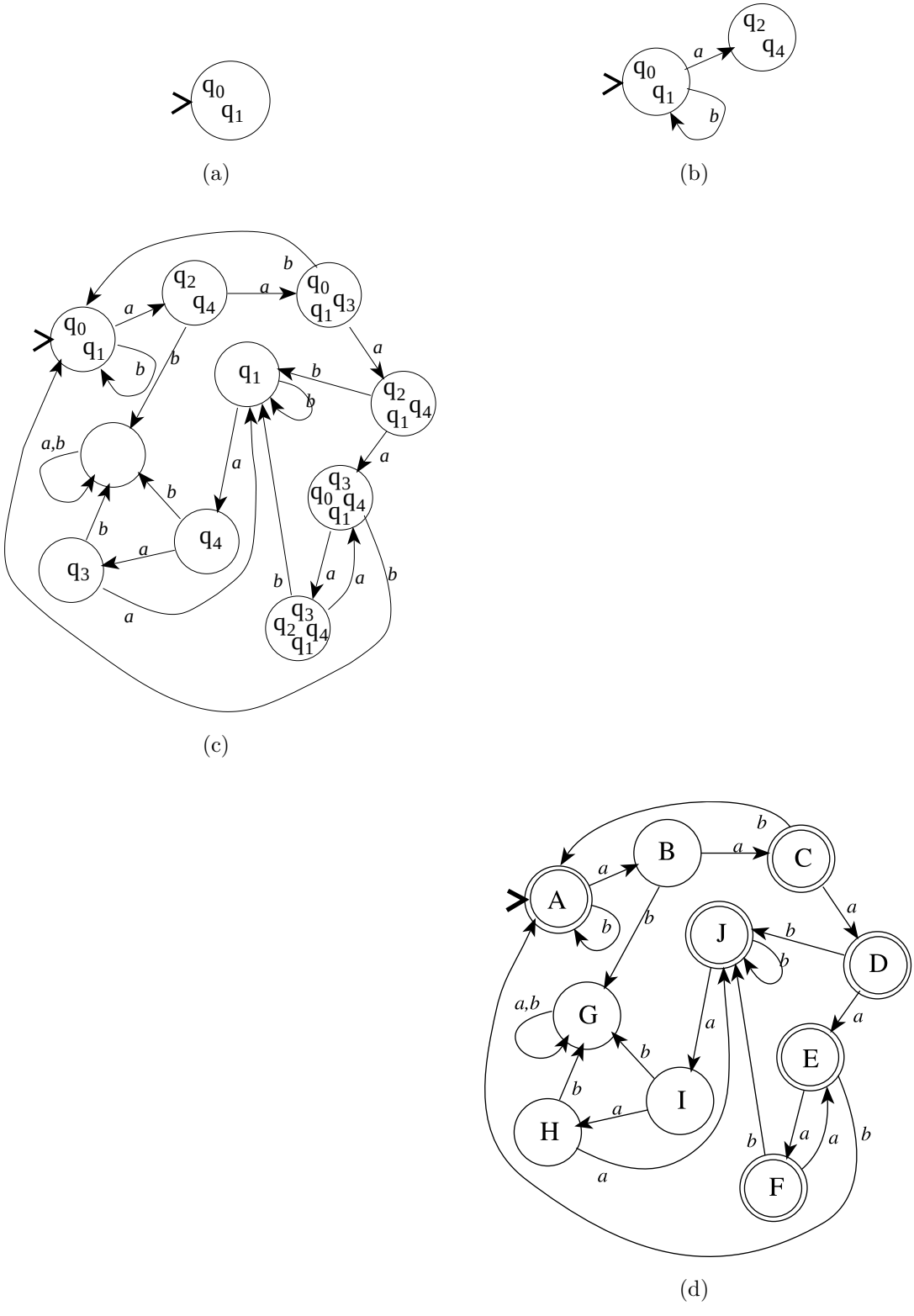


Figura 2.29: Transformación de AFN a AFD

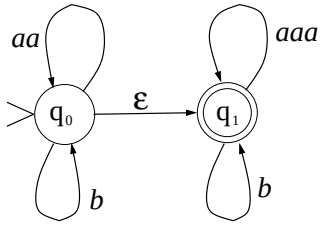


Figura 2.30: AFN con transiciones de varias letras

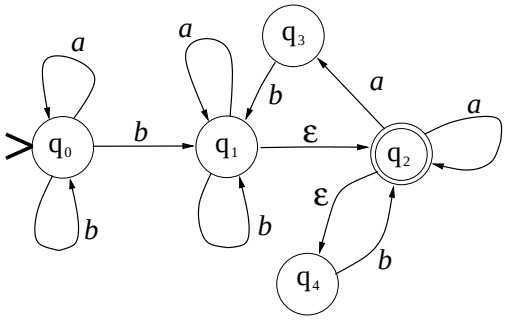


Figura 2.31: AFN con transiciones vacías

de conversión, si se utiliza un lenguaje de programación adecuado, preferentemente de tipo funcional, como por ejemplo Scheme [22].

Vamos a ir presentando las definiciones partiendo de la más sencilla, hasta llegar a la más compleja.

Primero introducimos una función $transicion(q, \sigma)$, que a partir de un estado q y un caracter dado σ obtiene el conjunto de estados a los que se puede llegar desde q directamente gastando el caracter σ. Por ejemplo, tomando el AFN de la figura 2.31, tenemos que $transicion(q_0, b) = \{q_0, q_1\}$. Similarmente, $transicion(q_1, b) = \{q_1\}$, y $transicion(q_3, a) = \{\}$. Se puede definir matemáticamente de la forma siguiente:

$$transicion(q, \sigma) = \{p \mid (q, \sigma, p) \in \Delta\}$$

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. Así, en la figura 2.28, se puede pasar de q_2 a q_0 y luego continuar "gratis" de q_0 a q_1, por lo que en realidad se tiene que considerar a q_1 como uno de los estados a los que se puede llegar desde $\{q_1, q_2\}$ gastando una a. Por lo tanto, hay que modificar la definición anterior.

Vamos a definir una función auxiliar $cerr\text{-}\varepsilon(q)$ que es el conjunto de estados a los que se puede llegar desde el estado q pasando por transiciones vacías. Además, si con una transición vacía se llega a otro estado que también tiene transiciones vacías, hay que continuar añadiendo a $cerr\text{-}\varepsilon(q)$ los estados a los que se llegue, hasta que no sea posible añadir nuevos

estados. Por ejemplo, en la figura 2.31, *cerr-ε*$(q_1) = \{q_1, q_2, q_4\}$, *cerr-ε*$(q_2) = \{q_2, q_4\}$, y *cerr-ε*$(q_0) = \{q_0\}$.

cerr-ε(q) se acostumbra llamar *cerradura al vacío* porque matemáticamente es la cerradura de q con la relación $\{(x, y) \mid (x, \varepsilon, y) \in \Delta\}$. [24]

La función *cerr-ε*(q) se puede definir como sigue:

Definición.- La cerradura al vacío *cerr-ε*(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-ε*(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-ε*$(\{q_1, \ldots, q_n\})$ es igual a *cerr-ε*$(q_1) \cup, \ldots, \cup$ *cerr-ε*(q_n).

Ejemplo.- Sea el AFN de la figura 2.31. Entonces *CERR-ε*$(\{q_1, q_3\}) = \{q_1, q_2, q_3, q_4\}$.

Con la función de cerradura al vacío ya estamos en condiciones de proponer una versión de la función *transicion* que tome en cuenta las transiciones vacías. Llamaremos a esta función "*transicion-ε*", y la definimos de forma que *transicion-ε*(q, σ) sea el conjunto de estados a los que se puede llegar desde q gastando σ, inclusive pasando por transiciones vacías. El algoritmo es como sigue, para un estado q y un caracter σ:

1. Calcular $Q_0 =$ *cerr-ε*(q)
2. Para cada estado de Q_0, obtener *transicion*(q, σ), y unir todos los conjuntos obtenidos, dando por resultado un conjunto Q_1.
3. *transicion-ε*$(q, \sigma) =$ *CERR-ε*(Q_1).

Por ejemplo, tomando la figura 2.31, para calcular *transicion-ε*(q_1, a), los pasos serían como sigue:

1. $Q_0 = \{q_1, q_2, q_4\}$
2. *transicion*$(q_1, a) = \{q_1\}$, *transicion*$(q_2, a) = \{q_2, q_3\}$, y *transicion*$(q_4, a) = \{\}$, por lo que uniendo estos conjuntos, $Q_1 = \{q_1, q_2, q_3\}$.

[24] Consultar la definición de *cerradura de una relación* en el capítulo de preliminares.

3. *transicion-*$\varepsilon(q_1, a) = CERR\text{-}\varepsilon(\{q_1, q_2, q_3\}) = \{q_1, q_2, q_3, q_4\}$.

Como última definición, es directo extender la función *transicion-*$\varepsilon(q, \sigma)$, que se aplica a un estado y un caracter, a una función que se aplique a un *conjunto de estados* y un caracter; llamamos a esta función *TRANSICION-*$\varepsilon(Q, \sigma)$, para un conjunto de estados Q y un caracter σ. Simplemente aplicamos *transicion-*$\varepsilon(q, \sigma)$ para cada uno de los estados $q \in Q$, y juntamos los resultados en un solo conjunto. Por ejemplo, en la figura 2.31 *TRANSICION-*$\varepsilon(\{q_0, q_2\}, a) = \{q_0, q_2, q_3, q_4\}$.

Finalmente resumimos el proceso global de transformación de un AFN a un AFD en el siguiente algoritmo.

Algoritmo de transformación AFN – AFD:

Dado un AFN $(K, \Sigma, \Delta, s, F)$, un AFD equivalente se obtiene por los siguientes pasos:

1. El conjunto de estados inicial es *cerr-*$\varepsilon(s)$.
2. El alfabeto del AFD es el mismo del AFN.
3. Para cada conjunto de estados Q ya presente, hacer:
   a) Añadir el conjunto de estados *TRANSICION-*$\varepsilon(Q, \sigma)$ para cada caracter σ del alfabeto, si no ha sido creado aún.
   b) Añadir transiciones $((Q, \sigma), Q_\sigma)$ para cada conjunto de estados Q_σ creado en el paso anterior.
4. Los conjuntos de estados que contengan un estado en F serán finales.

Recuérdese que lo que llamamos "conjunto de estados" en el algoritmo se refiere a conjunto de estados del AFN original, pero que serán simplemente estados en el AFD que se está creando.

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 construido 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;
- 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*-$\varepsilon(s)$, 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 $N = (K, \Sigma, \Delta, s, F)$. [25] Esta prueba se puede dividir en dos partes:

$\mathcal{L}(N) \subseteq \mathcal{L}(D)$. Si una palabra $w = \sigma_0\sigma_1 \ldots \sigma_n, \sigma_i \in \Sigma \cup \{\varepsilon\}$, es aceptada por N, entonces existe una secuencia estados $q_0, q_1, \ldots, q_{n+1}$, por los que pasa N en el cálculo:

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

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*-$\varepsilon(q_0)$, 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 $Q_\sigma = \mathit{TRANSICION}\text{-}\varepsilon(Q_D, \sigma)$, que va a ser un "estado" de D que va a contener a q_σ (¿Porqué?).

[25] Se supone que N ya sufrió la "transformación inofensiva" definida el la página 66.

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 reconstruir, 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

2.8.4. Más diseño de AFN: Intersección de lenguajes

Los problemas de diseño de AFN en que se combinan dos condiciones *que se deben cumplir simultáneamente* son particularmente difíciles de resolver. Un ejemplo de estos problemas sería: "obtener un AFN que acepte las palabras que contengan la cadena *abb* un número impar de veces y *ba* un número par de veces".

En los métodos de diseño de AFD propusimos trabajar con *grupos de estados*, y desde luego esto es aplicable también a los AFN. Sin embargo, sería aún mejor contar con un método *modular* que nos permitiera combinar de una manera sistemática las soluciones parciales para cada una de las condiciones. Ahora bien, esto es posible, si consideramos la siguiente propiedad de la intersección de conjuntos:

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

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

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's. 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. El AFD M_1 de la figura 2.32(a) acepta las palabras de longitud par, mientras que M_2 de 2.32(b) acepta las palabras con un número par de a's.

Ahora obtenemos los AFD que aceptan el complemento de los lenguajes de M_1 y M_2, cambiando los estados finales por no finales y viceversa; sean M_1^C y M_2^C.

Es *muy importante* notar que sólo es posible complementar AFD's y no cualquier AFN. En efecto, si en un AFN simplemente cambiamos estados finales por no finales y viceversa, en general llegaremos a un resultado erróneo (esto es, el autómata resultante no es equivalente al original). [26]

[26] Verifíquese esto tratando de hacer directamente la complementación del AFN $(\{1\}, \{a, b\}, \{(1, a, 1)\}, 1, \{1\})$, el cual originalmente acepta las palabras con a's, pero al cambiar finales por no finales ya no acepta ninguna palabra, en vez de aceptar las palabras con b's, como podríamos

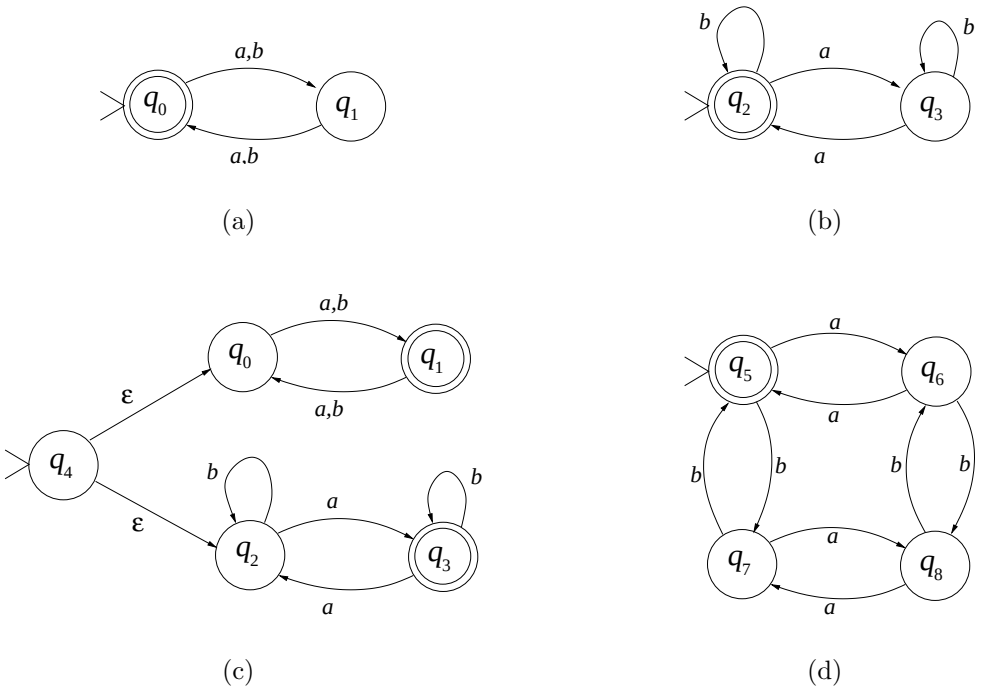


Figura 2.32: Intersección de dos AFN

Combinamos estos autómatas utilizando el procedimiento para la unión de lenguajes, dando un AFN M_3 (figura 2.32(c)), el cual es convertido a un AFD M_4. Finalmente, este AFD es simplificado y "complementado", dando M_4^C (figura 2.32(d)), que es el autómata buscado.

2.9. Ejercicios

1. Trazar un diagrama de estados y eventos que modele:

   a) El paso de una persona de un estado civil a otro: considere al menos los estados civiles "soltero", "casado", "divorciado", "viudo". Considere al divorcio como un proceso con duración (no instantáneo).

   b) El proceso de conexión de una terminal a un servidor Unix, desde el punto de vista del usuario (esto es, recibir mensaje pidiendo nombre de usuario, suministrar nombre de usuario, recibir petición de "password", etc.).

   c) El proceso de retiro de dinero en un cajero automático.

---

haber supuesto.

2. Diseñar directamente [27] AFD's que acepten los siguientes lenguajes; para cada ejemplo, establecer claramente lo que "recuerda" cada estado antes de trazar las transiciones. Escribir además cada AFD resultante usando la notación formal.

   a) Las palabras en $\{a, b\}$ que contienen un número par de a.

   b) Las palabras del lenguaje en $\{0, 1\}$ con a lo más un par de unos consecutivos.

   c) las palabras del lenguaje en $\{a, b\}$ que tienen un número impar de ocurrencias de la subcadena ab.

3. Diseñar usando el método del complemento un AFD que acepte las palabras en $\{a, b\}$ que no inicien con $abab$.

4. Utilizar el método de los grupos de estados ("nubes") para diseñar directamente (sin pasar por AFN) AFD's para los siguientes lenguajes:

   a) lenguaje en $\{0, 1\}$ donde las palabras no contienen la subcadena 11 pero sí 00.

   b) lenguaje en $\{a, b\}$ donde las palabras son de longitud par y tienen un número par de a.

   c) lenguaje en $\{a, b\}$ donde las palabras que contienen aba terminan en bb.

5. Minimizar cada uno de los AFD's del problema anterior:

   a) Por eliminación de estados redundantes, usando la tabla.

   b) Por clases de equivalencia.

6. Se puede usar el algoritmo de minimización para comparar dos AFD M_1 y M_2: dos AFD son equivalentes ssi sus AFD mínimos son iguales. Aquí la igualdad de M_1 y M_2 se entiende en cuanto a la estructura de los AFD, pero los nombres de los estados pueden ser diferentes.

7. 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. Definir formalmente cada uno de los componentes del autómata M', en que se ha eliminado de M el estado q_k. Poner especial cuidado en definir las transiciones en M'.

8. Calcular en detalle los autómatas M_1, M_2 y M_3 del ejemplo de la sección 2.4.2.

9. 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.

10. 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".

[27] Aqui "directamente" quiere decir que no se haga por transformación de otro tipo de soluciones, tales como las AFN.

11. 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 aceptada por M_1 pero no por 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 $L(M_2) \cup \{w\}$

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

    e) Comparar M_2'' con M_1.

    Pruebe la receta anterior con los AFD siguientes:

    a) $M_1 = (\{1,2\}, \{a,b\}, \{((1,a),2)((1,b),2),((2,a),2),((2,b),2)\}, 1, \{1\})$

    b) $M_2 = (\{3,4,5\}, \{a,b\}, \{((3,a),4),((3,b),5),((4,a),5),((4,b),5),((5,a),5),((5,b),5)\}, 3, \{4\})$

12. Decimos que dos AFD M_1 y M_2 son *iguales* –atención: no "equivalentes"– si sólo difieren eventualmente en el nombre de los estados. Definir formalemente la igualdad de AFDs como una relación de isomorfismo entre los estados de ambos.

13. Definir "lenguaje aceptado" para los AFN en términos de las configuraciones y del paso de una configuración a otra.

14. Dada la representación formal de un AFD $(K, \Sigma, \delta, s, F)$, obtener la representación formal de un AFN tal que los diagramas de ambos sean idénticos (esto es, hacer los ajustes necesarios para poder considerar al AFD como AFN).

15. Sean dos autómatas finitos no deterministas AFN_1 y AFN_2

    a) ¿Cómo es posible determinar si el lenguaje que acepta AFN_1 es subconjunto del que acepta AFN_2? Justifique su respuesta. Ayuda: Utilizar la propiedad de los conjuntos $A \subseteq B$ ssi $A \cup B = B$.

    b) Aplicar el procedimiento anterior para determinar si los AFN siguientes aceptan o no lenguajes complementarios:

        1) $(\{1,2\}, \{a,b\}, \{(1,a,1),(1,b,1),(1,aa,2),(2,a,2),(2,b,2)\}, 1, \{2\})$
        2) $(\{1,2,3\}, \{a,b\}, \{(1,a,2),(1,b,1),(2,a,3),(2,b,1),(3,a,3),(3,b,3)\}, 1, \{1,2\})$

16. Probar que al reemplazar toda transición (p, uv, q) por (p, u, i) y (i, v, q), creando un nuevo estado no final i, el AFN seguirá aceptando todas las palabras que aceptaba antes de hacer la transformación. (Sugerencia: hay que examinar los cálculos que permitían aceptar una palabra antes de la transformación, y mostrar que en el AFN transformado debe haber un cálculo modificado que permite aceptar la misma palabra).

17. Suponga una variante de los autómatas finitos, los autómatas con aceptación (AA), en que, además de los estados finales, hay estados de aceptación, tales que si el autómata pasa por uno de ellos, aunque sea una vez, la palabra será aceptada, independientemente del tipo de estado al que se llegue al agotar la palabra.

    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.

18. Suponga otra 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 o de rechazo. 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 "*abba*". Marque los estados de rechazo por nodos $\otimes$.

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

19. Un autómata finito casi determinista (AFCD) es un AFN 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 2.33. ¿Es posible probar que

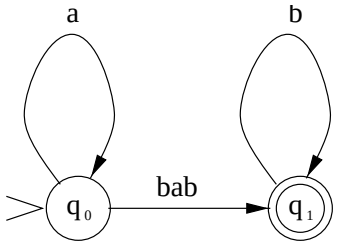


Figura 2.33: Ejemplo de AFCD

    un AFN dado es AFCD? Si es así, proponga un método sistemático para probarlo.

20. 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.

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

1) Pruebe que todo autómata de Mealy, visto como AFNDS, es confluente.
2) ¿Es posible decidir si un AFNDS es confluente? Pruebe su respuesta proponiendo un procedimiento para decidir, o mostrando porqué es imposible.

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

b) Proponer un procedimiento para obtener el conjunto de los estados accesibles en un AFD. (Ayuda: considerar cómo evoluciona el conjunto de "estados accesibles" ante posibles transiciones).

22. Decimos que un AFN "se traba" 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 "trabarse" para alguna palabra w (proponga un método de decisión).

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

24. Hemos visto que es posible diseñar modularmente un AFN que acepte la intersección de los lenguajes aceptados por M_1 y M_2. ¿Será también posible combinar M_1 y M_2 de manera que obtengamos un AFN que acepte la diferencia $L_1 - L_2$ de sus lenguajes? Proponga un método para hacerlo.

25. ¿Es regular el reverso de un lenguaje regular? (El reverso de una palabra $\sigma_1\sigma_2 \ldots \sigma_n$ es $\sigma_n \ldots \sigma_1$).

26. 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) Proponer un procedimiento que permita decidir si $L(M) = \emptyset$ (Ayuda: Utilizar la comparación de autómatas).

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

27. Probar el lema de la sección 2.6.

Capítulo 3

Expresiones Regulares y Gramáticas Regulares

En este capítulo estudiaremos la clase de lenguajes aceptados por los AF, la de los lenguajes *regulares*, que es al mismo tiempo una de las de mayor utilidad práctica. Como se aprecia en la figura 1.3, los Lenguajes Regulares son los más simples y restringidos dentro de la jerarquía de Chomsky que presentamos anteriormente. Estos lenguajes pueden además ser descritos mediante dos representaciones que veremos: las *Expresiones Regulares* y las *Gramáticas Regulares*.

3.1. Lenguajes Regulares

Los *lenguajes regulares* se llaman así porque sus palabras contienen "regularidades" o repeticiones de los mismos componentes, como por ejemplo en el lenguaje L_1 siguiente:

$$L_1 = \{ab, abab, ababab, abababab, \ldots\}$$

En este ejemplo se aprecia que las palabras de L_1 son simplemente repeticiones de "ab" cualquier número de veces. Aquí la "regularidad" consiste en que las palabras contienen "ab" algún número de veces.

Otro ejemplo más complicado sería el lenguaje L_2:

$$L_2 = \{abc, cc, abab, abccc, ababc, \ldots\}$$

La regularidad en L_2 consiste en que sus palabras comienzan con repeticiones de "ab",

seguidas de repeticiones de "c". Similarmente es posible definir muchos otros lenguajes basados en la idea de repetir esquemas simples. Esta es la idea básica para formar los *lenguajes Regulares*.

Adicionalmente a las repeticiones de esquemas simples, vamos a considerar que los lenguajes finitos son también regulares por definición. Por ejemplo, el lenguaje $L_3 = \{anita, lava, la, tina\}$ es regular.

Finalmente, al combinar lenguajes regulares uniéndolos o concatenándolos, también se obtiene un lenguaje regular. Por ejemplo, $L_1 \cup L_3 = \{anita$, $lava$, la, $tina$, ab, $abab$, $ababab$, $abababab$, $\ldots\}$ es regular. También es regular una concatenación como $L_3L_3 = \{anitaanita$, $anitalava$, $anitala$, $anitatina$, $lavaanita$, $lavalava$, $lavala$, $lavatina$, $\ldots\}$ [1]

3.1.1. Definición formal de Lenguajes Regulares

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

- L es finito;
- L es la unión o la concatenación de otros lenguajes regulares R_1 y R_2, $L = R_1 \cup R_2$ o $L = R_1R_2$ respectivamente.
- 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.

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. [2] Entonces su concatenación es $\{a\}\{a, b\}^*$, que también es regular.

[1] Recuérdese que la concatenación de dos lenguajes L_1 y L_2 se define como el conjunto de las palabras formadas concatenando una de L_1 con una de L_2, ver sección 1.4.2.

[2] 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.

3.2. 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. El conjunto ER de las expresiones regulares sobre Σ contiene las cadenas en el alfabeto $\Sigma \cup \{$"$\wedge$", "+", "$\bullet$", "*", "(", ")", "Φ"$\}$ que cumplen con lo siguiente:

1. "$\wedge$" y "Φ" $\in ER$
2. Si $\sigma \in \Sigma$, entonces $\sigma \in ER$.
3. Si $E_1, E_2 \in ER$, entonces "("E_1"+"E_2")" $\in ER$, "("E_1"$\bullet$"E_2")" $\in ER$, "("E_1")*" $\in ER$.

Las comillas " " enfatizan el hecho de que estamos definiendo cadenas de texto, no expresiones matemáticas [3]. 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", "$((a \bullet b(c)^*)$".

3.2.1. Significado de las ER

Las ER son simplemente fórmulas cuyo propósito es representar cada una de ellas un lenguaje. Así, el significado de una ER es simplemente el lenguaje que ella representa.

Por ejemplo, la ER "Φ" representa el conjunto vacío $\{\}$.

Para comprender intuitivamente la manera en que las ER representan lenguajes, consideremos el proceso de verificar si una palabra dada w pertenece o no al lenguaje representado por una ER dada. Vamos a decir que una palabra "*empata*" con una expresión regular si es parte del lenguaje que esta representa.

La palabra vacía ε "empata" con la ER $\wedge$.

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

Una palabra de una letra como "a" empata con una ER consistente en la misma letra "a", "b" empata "b", etc.

Luego, una palabra $w = uv$, esto es w está formada de dos pedazos u y v, empata con una expresión $(U \bullet V)$ a condición de que u empate con U y v empate con V. Por ejemplo, abc empata con $(a \bullet (b \bullet c))$ porque abc puede ser dividida en a y bc, y a empata con a en la ER, mientras que bc empata con $(b \bullet c)$ separando b y c de la misma manera.

Similarmente, cuando la ER es de la forma $(U + V)$, puede empatar con una palabra w cuando esta empata con U o bien cuando empata con V. Por ejemplo, bc empata $(a+(b \bullet c))$.

Una palabra w empata con una expresión U^* cuando w puede ser partida en pedazos $w = w_1 w_2, \ldots$ de tal manera que cada pedazo w_i empata con U. Por ejemplo, $caba$ empata con $(((c + b) \bullet a))^*$ porque puede partirse en los pedazos ca y ba, y ambos empatan con $((c + b) \bullet a)$, lo cual es fácil de verificar.

A continuación definiremos formalmente la correspondencia entre la representación (una ER) y el lenguaje representado.

Definición.- El significado de una ER es una función $\mathcal{L} : ER \to 2^{\Sigma^*}$ (esto es, una función que toma como entrada una expresión regular y entrega como salida un lenguaje), definida de la manera siguiente:

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

Para calcular el significado de una ER en particular, se aplica a ella la función $\mathcal{L}$. Las ecuaciones dadas arriba se aplican repetidamente, hasta que el símbolo $\mathcal{L}$ desaparezca.

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

$\mathcal{L}(\text{“}(((a + b))^* \bullet a)\text{”}) = \mathcal{L}(\text{“}((a + b))^*\text{”})\mathcal{L}(\text{“}a\text{”})$ -usando 4,

$= \mathcal{L}(\text{“}(a + b)\text{”})^*\{a\}$ -por 6 y 3,

$= (\mathcal{L}(\text{“}a\text{”}) \cup \mathcal{L}(\text{“}b\text{”}))^*\{a\}$ -aplicando 5,

$= (\{a\} \cup \{b\})^*\{a\} = \{a, b\}^*\{a\}$ -usando 3 y simplificando.

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

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 "•" y finalmente "+". Además se supone que los operadores "•" y "+" son asociativos.
- Eventualmente omitiremos el operador "•", 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.

3.2.2. Metodología de diseño de las ER

Al tratar de encontrar una ER para un lenguaje dado, mientras más complejo sea el lenguaje es obvio que resulta más 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 que no tienen *bb* y además no terminan en *b*. [4] Esto es equivalente a decir que toda *b* está seguida de una *a* o una *c*. Esto quiere decir que la ER de este contexto va ser de la forma:

$$(\ldots b(a+c)\ldots)^*$$

donde los detalles que faltan están representados por las "...". Lo que falta por considerar es que puede haber cualquier cantidad de *a*'s o *c*'s en el $< contexto_1 >$, por lo que dicho contexto queda como:

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

Similarmente se puede obtener la expresión para $< contexto_2 >$, que es $(a+c+(a+c)b)^*$, por lo que finalmente la ER del problema es:

$$(b(a+c)+a+c)^*bb(a+c+(a+c)b)^*$$

Un importante elemento de metodología -que se aplicó en este ejemplo- consiste en transformar los enunciados de lenguajes de manera que sean más fácilmente representables por ER. En particular, los enunciados "negativos", del tipo "...las palabras que no contengan *bb*" son particularmente difíciles, porque en las ER no hay ningún operador para representar "lo que no forma parte del lenguajes", sino que los operadores (como la unión o la estrella de Kleene) tienden a añadir más palabras. En consecuencia, es necesario convertir un enunciado sobre lo que no se permite en otro enunciado sobre lo que sí se permite. Por ejemplo, si en un lenguaje las palabras no deben contener la cadena "bb", ¿qué es lo que sí pueden contener? Aquí podemos hacer un análisis por casos, considerando que podemos tener una *b* sola, o también una *b* seguida de una *a*. Como hay dos casos, podemos pensar en utilizar el operador "+" para combinar esos casos, y así en adelante.

También puede ser útil modificar la forma lógica en que se enuncian los lenguajes. Por ejemplo, el enunciado "palabras que si empiezan en 00, terminan en 11", puede modificarse de la manera siguiente: "palabras que ya sea no empiezan en 00 o bien terminan en 11", utilizando la conocida equivalencia de lógica $P \Rightarrow Q \equiv \neg P \vee Q$. Lo que aquí se gana es que

[4] Pues si terminaran en *b*, esta última *b* se juntaría a la *bb* de la mitad, violando la condición del problema.

hacemos evidente la estructura de casos, que se puede resolver usando el operador "+" de las ER. Por supuesto, además apareció en este enunciado una parte expresada negativamente, "...no empiezan en 00", pero ésta es fácil de transformar en un enunciado positivo, como por ejemplo "...son la palabra vacía, o bien empiezan en 1, o bien empiezan en 01", el cual también se resuelve fácilmente por casos.

Ejemplo.- Obtener una ER que represente el lenguaje en $\{a, b\}$ tal que si una palabra contiene la subcadena aa, entonces no debe contener bb.

Solución: Transformando lógicamente el enunciado, representamos la condición "contiene la subcadena aa" por el símbolo C_{aa}, y la condición "no contiene bb" por $\neg C_{bb}$. Entonces la condición del problema es:

$$C_{aa} \Rightarrow \neg C_{bb}$$

Por las equivalencias lógicas vistas en la sección 1.2, esta condición es equivalente a:

$$\neg C_{aa} \vee \neg C_{bb}$$

Es decir que las palabras no contienen aa o bien no contienen bb. Esto corresponde a la estructura:

$$< sin\ aa > \ + \ < sin\ bb >$$

Vamos a resolver la primera parte, siendo la otra enteramente similar. Para que las palabras no contengan aa, pueden contener cualquier secuencia con b, o bien secuencias en que toda a esté separada de otra a por al menos una b. Como la cantidad de a's es cualquiera, necesariamente debemos hacer intervenir una estrella de Kleene, como en la estructura siguiente:

$$\ldots (b + \ \ldots a \ldots)^* \ldots$$

Una idea para precisar más esta estructura sería pensar que antes y después de la a debe haber una b, como en la expresión $(b + bab)^*$. Aunque esta ER es correcta, no es completa, pues hay palabras como ab que no son representadas por ella. Entonces pensaremos que *después* de la a esté la b que la separa de otras eventuales a's. La estructura así se precisa: [5]

$$\ldots (b + ab)^* \ldots$$

Ahora bien, pensando en qué puede aparecer al inicio y al final de una palabra, la subexpresión $(b + ab)^*$ por sí misma es ya capaz de representar palabras que comiencen ya sea con a o con b, por lo que podemos omitir el contexto del lado izquierdo. En cambio, $(b + ab)^*$ no

[5] Los "contextos" aquí fueron representados simplemente con "...". El uso de puntos suspensivos o de nombres para representar un contexto es simplemente cuestión de conveniencia en cada caso.

es capaz de representar palabras terminadas en a, como ba. Habría que añadir esta posibilidad. Pero si hacemos que el contexto derecho sea a, vamos a excluir palabras tales como ab. Entonces el contexto derecho puede ser a o b, que se resuelve con la expresión $a + b$, dando como resultado:

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

Pero aún esta expresión presenta un problema, que es el no poder representar a la palabra vacía. Esto se puede resolver de dos maneras: la menos elegante, que es simplemente añadir "$+\wedge$" a la ER, quedando como $(b+ab)^*(a+b)+\wedge$, o una solución más elegante que consiste en observar que la expresión $(b + ab)^*$ ya representaba palabras terminadas en b, por lo que en realidad el contexto derecho consistiría en agregar una a o nada en absoluto, quedando la ER como $(b + ab)^*(a + \wedge)$. Este es un resultado correcto y completo. Dejamos pendiente la solución del contexto $< sin\ bb >$.

En la sección de ejercicios de este capítulo se proponen muchos problemas de diseño de ER. Es importante emprender estos ejercicios siguiendo los elementos de metodología que hemos presentado (adaptar expresiones conocidas, diseñar estructuras con "contextos", transformar los enunciados), y no dejándose llevar por la primera "corazonada genial", que generalmente nos lleva a expresiones erróneas, principalmente por incompletez.

3.2.3. 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.

Peor aún, a diferencia de los AFD que vimos en el capítulo 2, no existen procedimientos algorítmicos para comparar directamente dos ER; la comparación tiene que hacerse pasando por una conversión a AFD que veremos más adelante.

Sin embargo, en algunos casos resulta útil aplicar ecuaciones de *equivalencia* entre las ER, que son expresiones de la forma $ER_1 = ER_2$, cuyo significado es que el lenguaje de ER_1 es el mismo que el de ER_2 (contienen las mismas palabras).

Por ejemplo, la equivalencia $R + S = S + R$ quiere decir que la suma de expresiones regulares es conmutativa, por lo que si tenemos dos ER específicas, como a^* y b^*ab, entonces la ER a^*+b^*ab será equivalente a la ER b^*ab+a^*, y ambas representarán las mismas palabras.

La equivalencia $R + S = S + R$ puede ser muy obvia, pues se basa directamente en la conmutatividad de la unión de conjuntos, pero hay otras como $(R^*S)^* = \wedge + (R + S)^*S$ que son mucho menos intuitivas.

A continuación damos una lista de las principales equivalencias de ER, clasificadas 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. $R^*R = RR^*, \quad R(SR)^* = (RS)^*R$
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 $x = r_0s_1r_1s_2r_2 \ldots s_nr_n$. Pero esta misma palabra puede agruparse de otra manera: $x = (r_0s_1)(r_1s_2) \ldots (r_{n-1}s_n)r_n$. 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. La técnica a usar para verificar que $P = Q$, donde $P, Q \in ER$, es formar una serie de equivalencias $P = R_1 = R_2 = \ldots = R_n = Q$, usando las equivalencias dadas arriba para hacer reemplazamientos.

Ejemplo: Verificar que las ER $(ab + a)^*a$ y $a(ba + a)^*$ son equivalentes, usando las equivalencias presentadas arriba.

Solución:

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

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

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

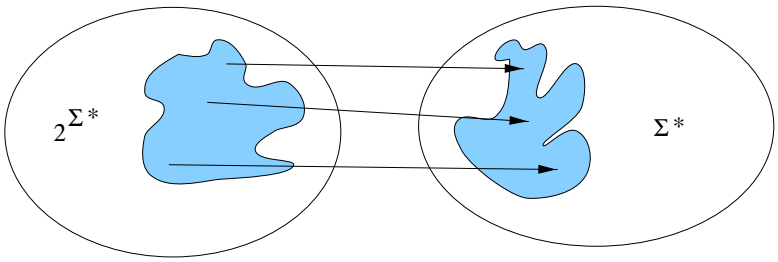


Figura 3.1: Representaciones de lenguajes

$= [a^*a](b[a^*a])^*$ -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).

El uso de estas equivalencias para verificar que dos ER denotan el mismo lenguaje en general no es muy práctico, pues en muchos casos es difícil ver intuitivamente qué equivalencia conviene aplicar, además de que hay el riesgo de que al aplicarla nos alejemos de la solución en vez de acercarnos a ella. Para empeorar la situación, no hay metodologías generales que nos ayuden a diseñar estas pruebas. Es por esto que normalmente probaremos la equivalencia de dos ER usando el procedimiento de conversión a AFD que veremos en la sección 3.4.

3.3. Límites de las representaciones textuales

Nos podemos preguntar qué tantos lenguajes se pueden representar con las ER. En otras secciones mostraremos que dichos lenguajes coinciden con los que pueden ser aceptados por algún autómata finito. Por lo pronto, en esta sección vamos a establecer un límite que existe no sólamente para las ER, sino para cualquier forma de representar lenguajes mediante texto.

En la figura 3.1 se ilustra el mapeo que pretendemos entre los lenguajes, que son elementos 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.

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 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 más 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 Cantor, que establece que el conjunto potencia de los números naturales, $2^{\mathbb{N}}$, es incontable. En efecto, observamos que el conjunto de todos los lenguajes, que es 2^{Σ^*}, tiene el mismo tamaño que $2^{\mathbb{N}}$, pues $\mathbb{N}$ y Σ^* son del mismo tamaño, que es lo mismo que decir que Σ^* es contable, lo cual es sencillo de probar [6] Así podemos concluir que, como $2^{\mathbb{N}}$ es incontable, 2^{Σ^*} también lo es. QED.

Se sabe que los conjuntos incontables son propiamente más "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.

3.4. Equivalencia de expresiones regulares y autómatas finitos

Aún cuando por varios ejemplos 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 de Kleene.- Un lenguaje es regular si y sólo si es aceptado por algún autómata finito.

Vamos a presentar una prueba de esta afirmación, no tanto por el interés matemático que tiene, sino porque nos brinda procedimientos estándar extremadamente útiles para transformar una expresión regular en autómata finito y viceversa.

[6] La prueba es uno de los ejercicios al final de esta sección.

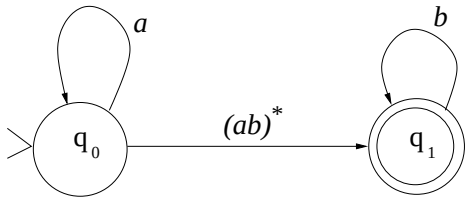


Figura 3.2: Una gráfica de transición

3.4.1. Conversión de ER a AF

La prueba de que si un lenguaje es regular entonces es aceptado por un AF 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 utilizar alguna representación de los lenguajes regulares que sea intermedia entre las ER y los AFN.

Una posible solución es el uso de 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 $\Delta \in K \times ER \times K$.

En la figura 3.2 se ilustra un ejemplo de GT. En este ejemplo en particular es fácil ver que debe aceptar palabras que tienen primero una sucesión de a's, luego repeticiones de ab, y finalmente repeticiones de b's. Esta GT se representaría formalmente como el quíntuplo:

$$(\{q_0, q_1\}, \{a, b\}, \{(q_0, a, q_0), (q_0, (ab)^*, q_1), (q_1, b, q_1)\}, q_0, \{q_1\})$$

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.

Ahora procederemos a describir el procedimiento de transformación de ER a AFN.

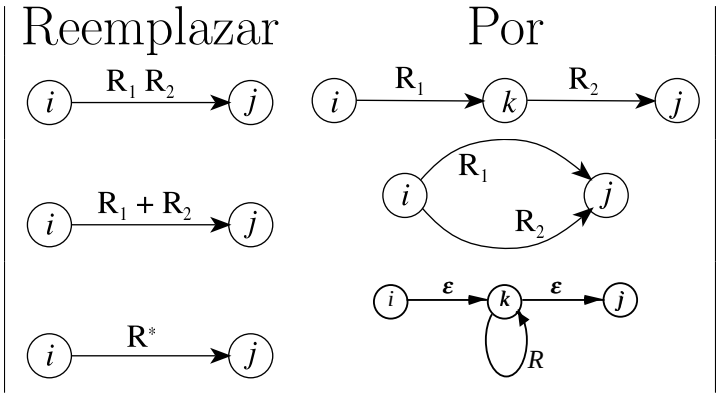
A partir de una ER es trivial obtener una GT que acepte el mismo lenguaje. En efecto, sea R una ER; entonces, si

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

entonces $L(G) = L(R)$. Por ejemplo, la GT asociada a la ER $(a + ba)^*bb$ se ilustra en la figura 3.3(a).

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 solos (o bien la palabra

Cuadro 3.1: Eliminación de operadores para pasar de ER a AF



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. Las transformaciones elementales se ilustran en la Tabla 3.1.

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 3.3(a)-(d).

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

3.4.2. Conversión de AF a ER

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 únicamente 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

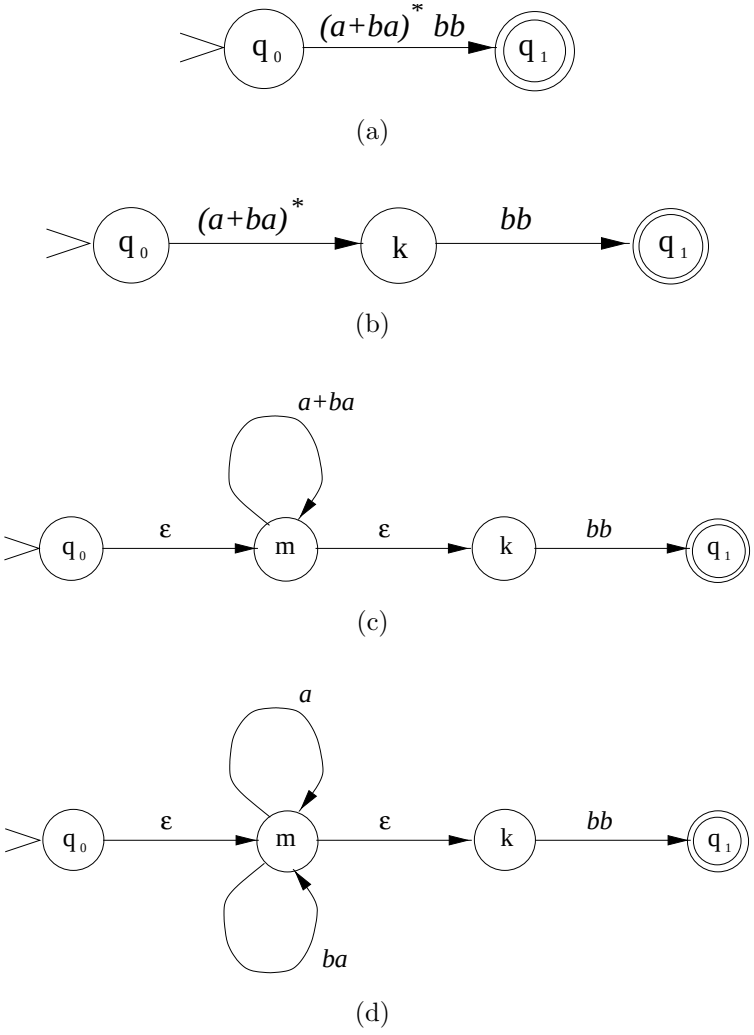
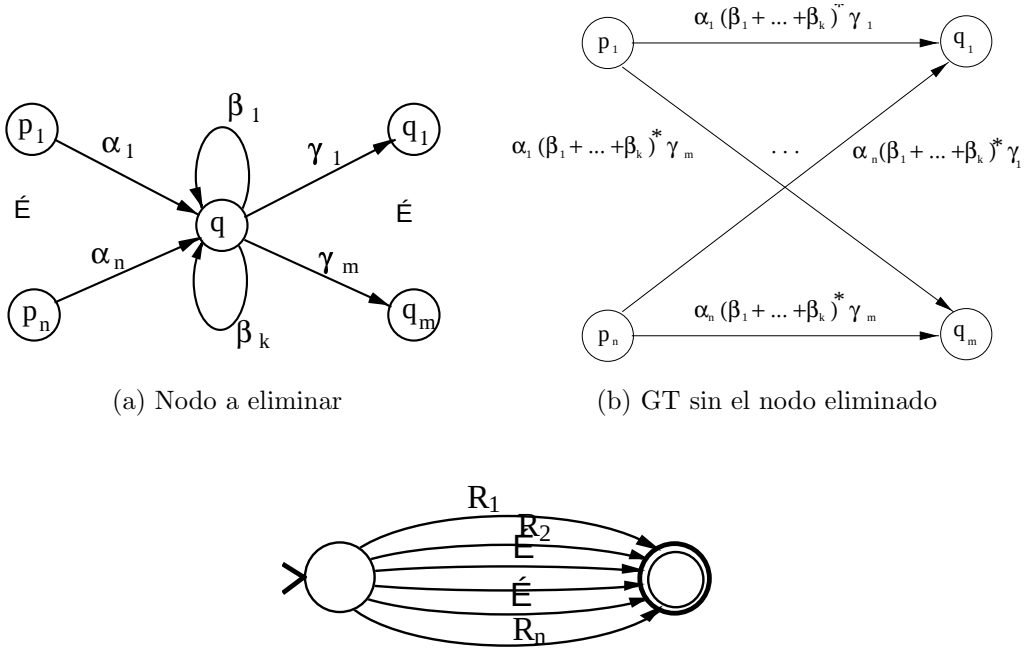


Figura 3.3: Transformación ER→AF



(a) Nodo a eliminar

(b) GT sin el nodo eliminado

(c) GT tras la eliminación de nodos intermedios

Figura 3.4: Eliminación de nodos

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 estado inicial al que no llegue ninguna transición, y un solo estado final, del que no salga ninguna transición. Esta condición se requiere para llevar a cabo el siguiente paso. [7] Desde luego, hay muchos autómatas que ya cumplen estas condiciones sin necesidad de añadir un nuevo estado inicial o un nuevo estado final.

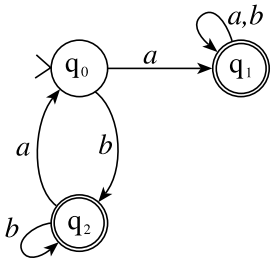
2. El segundo paso consiste en eliminar *nodos intermedios* en la GT. Se llama nodo intermedio a aquel que se encuentra en una trayectoria entre el estado inicial y el final. El procedimiento de eliminación de nodos intermedios es directo. La idea es que *al suprimir el nodo en cuestión, no se alteren las cadenas que hay que consumir para pasar de uno a otro de los nodos vecinos.* En otras palabras, al suprimir dicho nodo, se deben reemplazar las transiciones que antes tomaban ese nodo como punto intermedio para ir de un nodo vecino a otro, por otras transiciones que vayan del nodo vecino origen al nodo vecino destino, pero ahora sin pasar por el nodo eliminado. Para comprender cabalmente el procedimiento, hay que seguir el ejemplo dado más adelante. En la figura 3.4(a) se representa un nodo q intermedio que se quiere eliminar, 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 eliminar el nodo q, reemplazamos la parte de la GT descrita en la figura 3.4(a) por el subgrafo representado en la figura

[7] Mas adelante se presenta un ejemplo de cada una de las operaciones involucradas en este procedimiento.

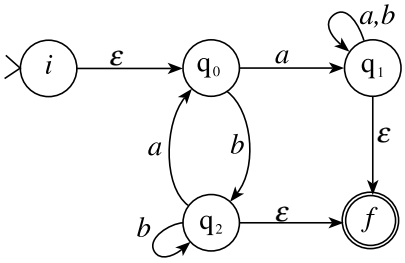
3.4(b). Los nodos intermedios se pueden eliminar en cualquier orden.

3. Cuando después de aplicar repetidamente el paso 2 ya no sea posible hacerlo, tendremos una GT de la forma de la figura 3.4(c). 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.

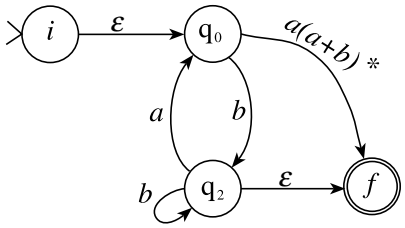
Ejemplo.- Obtener una ER para el AFD de la figura siguiente:



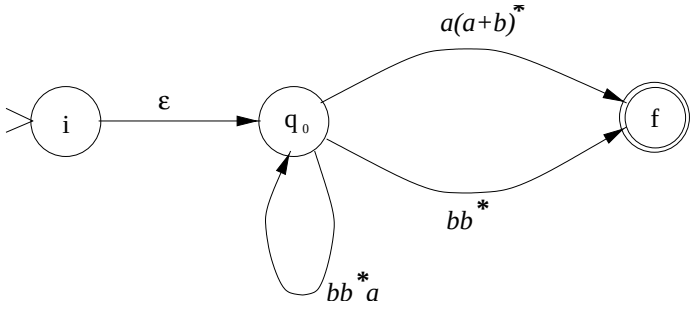
Paso 1.- Añadir un nuevo estado inicial y uno final



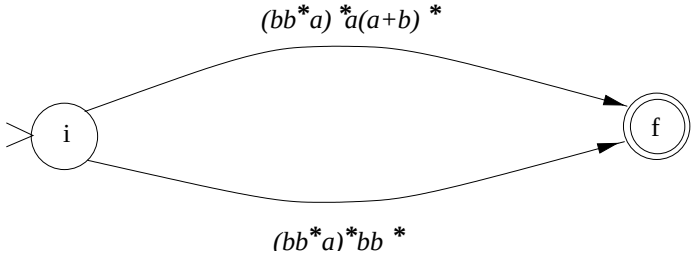
Paso 2.- Eliminación de nodos intermedios. Eliminamos primero el nodo q_1. Para ello, consideramos qué trayectorias o "rutas" pasan por el nodo a eliminar. Por ejemplo, en la figura de arriba vemos solamente una trayectoria que pasa por q_1, la cual va de q_0 a f. Ahora nos proponemos eliminar el nodo 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:



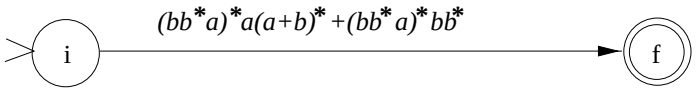
Paso 3.- Después eliminamos el nodo q_2:



Paso 4.- Eliminamos q_0:



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



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

La corrección de cada paso de transformación se desprende del hecho de que tanto la eliminación de nodos como la fusión de transiciones que se hace al final, preservan ambos la igualdad del lenguaje aceptado.

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.

3.5. Gramáticas regulares

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

3.5.1. 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. Chomsky [4], y está basada en las llamadas *reglas gramaticales*.

Una regla es una expresión de la forma $\alpha \rightarrow \beta$, 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*. Los símbolos que no son variables son *constantes*. [8] Por ejemplo, una posible regla gramatical es $X \rightarrow aX$. La aplicación de una regla $\alpha \rightarrow \beta$ 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. Por ejemplo, si tenemos una cadena de símbolos $bbXa$, le podemos aplicar la regla $X \rightarrow aX$, dando como resultado la nueva cadena $bbaXa$.

3.5.2. Gramáticas regulares

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

Ejemplo.- Sea una gramática con las siguientes reglas:

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. $B \rightarrow bA$

La idea para aplicar una gramática es que se parte de una variable, llamada *símbolo inicial*, y se aplican repetidamente las reglas gramaticales, hasta que ya no haya variables en la palabra. En ese momento se dice que la palabra resultante es *generada* por la gramática, o en forma equivalente, que la palabra resultante es parte del lenguaje de esa gramática.

[8] En la terminología de los compiladores, se les llama "terminales" a los elementos de Σ, y "no terminales" a las variables.

Por ejemplo, en la gramática que acabamos de presentar, si consideramos que las variables son S (que será el símbolo inicial), A y B, y las constantes a y b, partiendo de S podemos producir bA (por la segunda regla), luego de bA podemos pasar a ba (por la quinta regla). Como ba tiene sólo constantes, podemos concluir que la palabra ba es parte del lenguaje generado por la gramática dada. De hecho el lenguaje generado por esta gramática es el de las palabras en $\{a, b\}$ de longitud par terminadas en a. [9]

Formalizamos estas nociones con las siguientes definiciones:

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

V es un alfabeto de *variables*,

Σ es un alfabeto de *constantes*,

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

S, el símbolo inicial, es un elemento de V.

Por ejemplo, la gramática que presentamos arriba se representaría formalmente como:

$$(\{S, A, B\}, \{a, b\}, \{(S, aA), (S, bA), (A, aB), (A, bB), (A, a), (B, aA), (B, bA)\}, S)$$

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

La aplicación de una gramática se formaliza con las siguientes nociones:

Una cadena uXv *deriva en un paso* una cadena $u\alpha v$, escrito como $uXv \Rightarrow u\alpha v$, si hay una regla $X \to \alpha \in R$ en la gramática.

Una cadena $w \in \Sigma^*$ (esto es, formada exclusivamente por constantes) es *derivable* a partir de una gramática G si existe una secuencia de pasos de derivación $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow w$.

A una secuencia de pasos de derivación le llamamos simplemente *derivación.*

Dicho de otra manera, una palabra $w \in \Sigma^*$ es derivable a partir de G ssi $S \overset{*}{\Rightarrow} w$, donde $\overset{*}{\Rightarrow}$ denota la cerradura reflexiva y transitiva de $\Rightarrow$.

Definición.- El *lenguaje generado* por una gramática G, $L(G)$, es igual al conjunto de las palabras derivables a partir de su símbolo inicial.

Esto es, $L(G) = \{w \in \Sigma^* | S \overset{*}{\Rightarrow} w\}$.

[9] Más adelante veremos cómo probar rigurosamente que una gramática efectivamente corresponde a un lenguaje dado.

Frecuentemente es fácil mostrar que una palabra dada w es derivable a partir del símbolo inicial S; por ejemplo, en la gramática presentada arriba, se puede mostrar que $S \Rightarrow \ldots \Rightarrow bababa$ (esto es, que la palabra $bababa$ puede ser derivada a partir del símbolo inicial S, por lo que $bababa \in L(G)$. Dejamos este ejemplo como ejercicio (ver sección de ejercicios).

Ejemplo.- Proponer una gramática que genere el lenguaje de las palabras en $\{a, b\}$ que contienen la subcadena bb, como abb, $ababba$, etc.

Vamos a utilizar las variables de una manera similar a como se utilizaban en los AF los estados, esto es, como memorias para "recordar" situaciones. Así tendremos las siguientes variables:

- A, que recuerda que aún no se produce ninguna b.
- B, que recuerda que se produjo una b.
- C, que recuerda que ya se produjeron las dos b's.

Ahora podemos proponer reglas, preguntándonos a qué situación se llega al producir una a o b. Por ejemplo, a partir de A, si se produce una a se debe llegar a la misma A, pero si llega una b se llegará a la variable B. Con estas ideas se proponen las siguientes reglas:

1. $A \rightarrow aA$
2. $A \rightarrow bB$
3. $B \rightarrow aA$
4. $B \rightarrow bC$
5. $C \rightarrow aC$
6. $C \rightarrow bC$

Finalmente, para terminar la producción de una palabra hecha solamente de constantes es necesaria al menos una regla que no produzca variables en su lado derecho. Tal regla no se encuentra aún en la gramática dada. Como las palabras correctas tienen bb, pensamos que una regla adicional podría ser $C \rightarrow a$ y también $C \rightarrow b$. En efecto, con tales reglas podemos producir, por ejemplo, la palabra $abba$, mediante la derivación siguiente:

$$A \Rightarrow aA \Rightarrow abB \Rightarrow abbC \Rightarrow abba$$

Sin embargo, también podemos verificar que la palabra abb, que pertenece al lenguaje, no puede producirse con las reglas dadas. Hace falta aún otra regla, $B \rightarrow b$, con la que se completa nuestra gramática.

Al diseñar gramáticas regulares, podemos incurrir en los mismos errores que en los AF, es decir, que sean incorrectas (producen palabras que no deberían) o bien incompletas (no pueden generar palabras que pertenecen al lenguaje), o bien ambas cosas a la vez.

No vamos a examinar métodos particulares de diseño de gramáticas regulares; en vez de ello mejor vamos a examinar métodos por los que es muy simple convertir las gramáticas regulares a AF y viceversa.

3.5.3. Autómatas finitos y gramáticas regulares

De manera similar 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

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, construir 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.

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. $B \to bA$

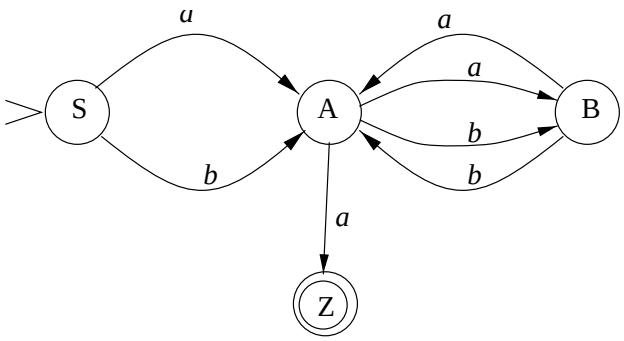


Figura 3.5: Autómata obtenido de la gramática

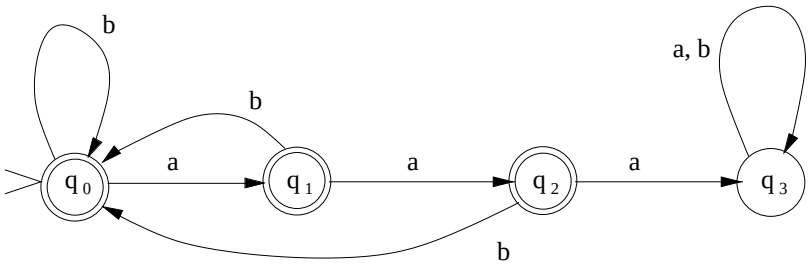


Figura 3.6: AFD que acepta palabras que no contienen 3 a's seguidas

Dicho autómata aparece en la figura 3.5.

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 regla $X_p \to \sigma X_q$, 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 reglas 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.

Ejemplo.- Para el AFD de la figura 3.6, la gramática regular correspondiente contiene las reglas:

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

Nótese que en este ejemplo el AFD acepta la palabra vacía, mientras que la GR no es capaz de generarla. De hecho *ninguna* GR puede generar ε. En estos casos nos tenemos que contentar con generar un lenguaje igual al aceptado por el AFD *excepto por la palabra vacía*.

3.6. Limitaciones de los lenguajes regulares

Los AF están limitados a los estados de que disponen como único medio para "recordar" la serie de símbolos recibidos hasta un momento dado. Puesto de otra manera, debemos considerar que, en un AF, la única traza de los símbolos recibidos es el estado en que se encuentra. Por lo mismo, varias secuencias distintas de caracteres que llevan a un mismo estado son consideradas como indistinguibles. Por ejemplo, para el AFD de la figura 2.4, las secuencias de caracteres *bab* y *bbbbb* son indistinguibles, pues ambas llevan al estado q_2. Esta limitación de los AF los hace finalmente incapaces de distinguir las palabras aceptables de las no aceptables en ciertos lenguajes, más complicados que los lenguajes regulares.

Por ejemplo, para el lenguaje $\{a^n b^n\}$ no es posible construir un autómata finito que lo acepte, ni representarlo por una expresión regular o gramática regular. En efecto, supongamos que un AFD está recorriendo una palabra $a^n b^n$, entonces al terminar el grupo de a's el autómata debe recordar cuántas encontró, para poder comparar con el número de b's. Ahora bien, como la cantidad de a's 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.

3.6.1. 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. 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 solamente m estados, y el cálculo tiene longitud $n+1$, por el principio de correspondencia 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 $\{a^n b^n\}$ 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 caracteres distintos a "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 $\{a^n b^n\}$ 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 caracteres distintos de b. 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 [10] ; el lenguaje de las palabras palíndromas, que se leen igual al derecho y al revés, como por ejemplo ANITALAVALATINA, [11] el lenguaje de los paréntesis bien balanceados, como ()(()), ()()(), ((())), etc.

3.7. Ejercicios

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

2. Encontrar Expresiones Regulares que representen los siguientes lenguajes (se presentan en orden de dificultad creciente):

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

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

   c) El lenguaje $\{101, 1110\}$.

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

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

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

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

   h) 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. [12]

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

[10] w^R es el reverso de w, es decir, $(abaa)^R = aaba$

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

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

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

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

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

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

n) Conjunto de palabras en $\{0, 1\}$ tales que no hay ningún par de ceros consecutivos después [13] de un par de unos consecutivos, como "0000110", "0001000", etc.

ñ) $\{w \in \{a, b, c\}^* \mid |w| \neq 3\}$

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

p) $\{w \in \{a, b, c\}^* \mid w \neq \alpha abc\beta\}$, donde α y β representan cualquier cadena de caracteres (esto es, las palabras en este lenguaje no deben contener la subcadena abc).

q) Lenguaje en $\{a, b\}$ tal que $|w|$ es par, y además la cantidad de a's que aparecen en w es par.

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

s) 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 "$baaaab$".

t) 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.

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

3. Demostrar la siguiente equivalencia por identidades de Expresiones Regulares:

   $(ab^*)^*a = a + a(a + b)^*a$

4. Verificar si los siguientes pares de ER son equivalentes, usando equivalencias o bien encontrando un contraejemplo:

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

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

5. Probar que $a^* + b^*$ no es equivalente a $(a + b)^*$

6. Convertir la ER $(a + ab)^*aba(a + ba)^*$ a AFN.

7. Convertir a ER el AFD siguiente: $(\{1, 2, 3\}, \{a, b\}, \{((1, a), 1), ((1, b), 2), ((2, a), 3), ((2, b), 1), ((3, a), 2), ((3, b), 3)\}, 1, \{1, 3\})$.

[13] En cualquier posición a la derecha

8. Demostrar la siguiente equivalencia $(ab^*)^*a = a + a(a + b)^*a$ por equivalencia de autómatas (Primero hay que convertir las ER a AFN, luego estos a AFD, los cuales pueden ser comparados).

9. 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

10. Para la gramática de la página 96, mostrar una derivación de la palabra *bababa*.

11. Comprobar si la ER $(a + \wedge)(ba)^*(b + \wedge)$ es equivalente a la GR cuyas reglas son: $A \to bB, B \to aA, A \to a, B \to b$, convirtiendo ambas a AFN, luego a AFD, y comparando.

12. Verificar si es vacía la intersección del lenguaje representado por la ER $(\wedge + b)(a + ab)^*$, con el lenguaje representado por la GR cuyas reglas son: $S \to aS, S \to bT, T \to aS, T \to bU, U \to aU, U \to bU, U \to a, U \to b$, utilizando el método siguiente: Primero se convierten a AFD tanto la ER como la GR. Luego se calcula su intersección, utilizando el método expuesto en el capítulo precedente. Finalmente, el resultado del paso anterior, que es un AFD, se compara con el AFD $(\{q_0\}, \{a, b\}, \{((q_0, a), q_0), ((q_0, b), q_0)\}, q_0, \{\})$, que acepta el lenguaje vacío.

13. Probar las equivalencias:

   a) $R \bullet \Phi = \Phi \bullet R = \Phi$, para una ER R

   b) $\Phi^* = \wedge$

14. Demostrar que para todo alfabeto Σ, el lenguaje Σ^* es contable. (Ayuda: Tratar de ordenar las cadenas de Σ^* de menor a mayor).

15. 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.

   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*".

c) Probar que el operador "−" no aumenta el poder de las ER, en el sentido de que para toda ER con operador "−" hay una ER equivalente sin "−".

16. Lo mismo del problema anterior, con un operador de intersección, "&" tal que $R_1 \& R_2$ representa las palabras representadas simultáneamente por R_1 y por R_2. En este caso proponer una ER para el lenguaje en $\{a, b\}$ donde las palabras contienen la subcadena "*abaab*" y un número impar de *b*'s.

17. Proponer una definición formal de configuración, cálculo y palabra aceptada para las GT.

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

19. Hacer la prueba de corrección de la gramática de la página 100. Esto proveerá una prueba de corrección del AFD de la figura 3.6.

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

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

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

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

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

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

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

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

c) $\{w \neq a^n b^n\}$ (Ayuda: use los métodos para combinar AFN's)

d) $\{a^n b^n \mid n \leq 7\}$

22. 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)

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

a) ¿Su unión podría eventualmente ser regular? (Ayuda: considere dos lenguajes complementarios).

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

c) ¿Su concatenación podría ser regular? (Ayuda: tomar $\{a^n b^m | n \geq m\}$ como L_A, y buscar un L_B "adecuado").

24. Supóngase un tipo de gramáticas que llamaremos "semirregulares", por asemejarse a las gramáticas regulares, en que las reglas son de alguna de las siguientes formas:

    a) $A \to \sigma B$

    b) $A \to B\sigma$

    c) $A \to \sigma$

    donde σ es un terminal, y A y B son no terminales. ¿Serán equivalentes a las gramáticas regulares? Pruebe su respuesta.

25. 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 más 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).

Parte II

Lenguajes libres de contexto y sus máquinas

Capítulo 4

Gramáticas y lenguajes libres de contexto

Los Lenguajes Libres de Contexto (abreviado LLC) forman una clase de lenguajes más amplia que los Lenguajes Regulares, de acuerdo con la Jerarquía de Chomsky (ver sección 1.5). Estos lenguajes son importantes tanto desde el punto de vista teórico, por relacionar las llamadas *Gramáticas Libres de Contexto* con los *Autómatas de Pila*, como desde el punto de vista práctico, ya que casi todos los lenguajes de programación están basados en los LLC. En efecto, a partir de los años 70's, con lenguajes como Pascal, se hizo común la práctica de formalizar la sintaxis de los lenguajes de programación usando herramientas basadas en las *Gramáticas Libres de Contexto*, que representan a los LLC. Por otra parte, el análisis automático de los LLC es computacionalmente mucho más eficiente que el de otras clases de lenguajes más generales.

Retomaremos aquí las nociones relacionadas con las gramáticas, que fueron introducidas en la sección 3.5, pero haciendo las adaptaciones necesarias para los LLC.

Una *regla* es una expresión de la forma $\alpha \rightarrow \beta$, en donde tanto α como β son cadenas de símbolos en donde pueden aparecer tanto elementos del alfabeto Σ (llamados *constantes*) como unos nuevos símbolos, llamados *variables*. [1]

Una gramática es básicamente un conjunto de reglas. [2]

Consideremos, por ejemplo, la siguiente gramática para producir un pequeño subconjunto del idioma español:

1. $< frase > \rightarrow < sujeto > < predicado >$

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

[2] Adelante precisaremos las definiciones.

2. $<sujeto> \rightarrow <articulo><sustantivo>$
3. $<articulo> \rightarrow el \mid la$
4. $<sustantivo> \rightarrow perro \mid luna$
5. $<predicado> \rightarrow <verbo>$
6. $<verbo> \rightarrow brilla \mid corre$

donde el símbolo "|" separa varias alternativas. [3] En esta gramática se supone que las variables son $<frase>$, $<sujeto>$, $<articulo>$, $<sustantivo>$, $<predicado>$ y $<verbo>$, mientras que las constantes son *el*, *la*, *perro* y *luna*. La variable $<frase>$ será considerada el *símbolo inicial*.

Como vimos en la sección 3.5, la idea para aplicar una gramática es que se parte de una variable, llamada *símbolo inicial*, y se aplican repetidamente las reglas gramaticales, hasta que ya no haya variables en la palabra. En ese momento se dice que la palabra resultante es *generada* por la gramática, o en forma equivalente, que la palabra resultante es parte del lenguaje de esa gramática.

Por ejemplo, podemos usar la gramática que acabamos de presentar, para generar la frase "el perro corre". En efecto, partiendo del símbolo inicial $<frase>$, aplicando la primera regla podemos obtener $<sujeto><predicado>$. Luego, reemplazando $<sujeto>$ por medio de la segunda regla, obtenemos $<articulo><sustantivo><predicado>$; aplicando la tercera regla, llegamos a *el* $<sustantivo><predicado>$. Por la cuarta regla se llega a *el perro* $<predicado>$; por la quinta a *el perro* $<verbo>$, y finalmente, por la sexta, llegamos a *el perro corre*.

Desde luego, usando esta misma gramática podemos producir frases que tienen menos sentido, como "la perro brilla". Para asegurar la coherencia en el uso de artículos, sustantivos y verbos se requeriría una gramática más sofisticada, y aún así sería posible producir frases sin sentido. [4]

4.1. Gramáticas y la jerarquía de Chomsky

Es posible restringir la forma de las reglas gramaticales de manera que se acomoden a patrones predeterminados. Por ejemplo, se puede imponer que el lado izquierdo de las reglas sea una variable, en vez de una cadena arbitraria de símbolos. Al restringir las reglas de la gramática se restringen también las palabras que se pueden generar; no es extraño

[3] La notación "|" es en realidad una abreviatura; una regla $X \rightarrow \alpha|\beta$ es equivalente a las dos reglas $X \rightarrow \alpha$ y $X \rightarrow \beta$.

[4] Muchos políticos son versados en estas artes...

que las reglas de formas más restringidas generan los lenguajes más reducidos. N. Chomsky propuso [4] varias formas estándares de reglas que se asocian a varias clases de lenguajes, que ordenó de manera tal que forman una jerarquía, es decir, los lenguajes más primitivos están incluidos en los más complejos. [5] Así tenemos las siguientes clases de gramáticas, asociadas a familias de lenguajes:

1. Gramáticas *regulares*, o de *tipo 3*: las reglas son de la forma $A \to aB$ o bien $A \to a$, donde A y B son variables y a es constante. [6] Estas gramáticas son capaces de describir los lenguajes regulares.

2. Gramáticas *Libres de Contexto* (GLC), o de *tipo 2*: las reglas son de la forma $X \to \alpha$, donde X es una variable y α es una cadena que puede contener variables y constantes. Estas gramáticas producen los lenguajes *Libres de Contexto* (abreviado "LLC").

3. Gramáticas *sensitivas al contexto* o de *tipo 1*: las reglas son de la forma $\alpha A\beta \to \alpha\Gamma\beta$, donde A es una variable y α,β y Γ son cadenas cualesquiera que pueden contener variables y constantes.

4. Gramáticas no restringidas, o de *tipo 0*, con reglas de la forma $\alpha \to \beta$, donde α no puede ser vacío, que generan los lenguajes llamados "recursivamente enumerables". [7]

Los lenguajes de tipo 0 incluyen a los de tipo 1, estos a los de tipo 2, etc. En la figura 1.3 ya se había presentado la relación entre los lenguajes de tipo 0, 2 y 3.

4.2. Lenguajes y gramáticas libres de contexto (LLC y GLC)

Podemos ver que la gramática del español dada arriba es una GLC, pero no podría ser una gramática regular, pues hay varias reglas que no corresponden al formato de las reglas de las gramáticas regulares. Se ve por lo tanto que el formato de las reglas es menos rígido en las GLC que en las gramáticas regulares, y así toda gramática regular es GLC pero no viceversa.

Por ejemplo, el lenguaje $\{a^n b^n\}$ –que no es regular, como vimos en la sección 3.6– tiene la gramática libre de contexto con las siguientes reglas:

1. $S \to aSb$
2. $S \to ab$

[5] La jerarquía de Chomsky fue presentada inicialmente en la sección 1.5.
[6] Estas gramáticas ya fueron discutidas en el capítulo 3.5.
[7] Las dos últimas clases de lenguajes serán discutidas en el capítulo 6.

Como vimos en el caso de las gramáticas regulares, aplicar una regla $X \to \alpha$ de una gramática consiste en reemplazar X por α en una palabra. Por ejemplo, la regla $S \to aSb$ se puede aplicar a una palabra $aaSbb$ para obtener la palabra $aaaSbbb$, en donde es fácil ver que reemplazamos S por aSb.

Al proceso de aplicar una regla se le conoce como "paso de derivación", y se denota usando una flecha gruesa "⇒", como en $aaSbb \Rightarrow aaaSbbb$ (aplicando una regla $S \to aSb$). Una secuencia de pasos de derivación a partir de una variable especial de la gramática llamada "símbolo inicial" se llama simplemente *derivación*. Por ejemplo, una derivación de la palabra "$aaabbb$" utilizando la gramática de $\{a^n b^n\}$ sería (suponiendo que S es el símbolo inicial):

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$$

Como un ejemplo adicional, la gramática con las reglas siguientes permite generar expresiones aritméticas con sumas y multiplicaciones de enteros:

1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to CF$
6. $F \to C$
7. $C \to 0|1|2|3|4|5|6|7|8|9$

El símbolo inicial aquí es E, las constantes son $+$, $*$ y las cifras $0 \ldots 9$; E, T, F, C son variables.

Con esta gramática podemos generar, por ejemplo, la expresión $25 + 3 * 12$ de la manera siguiente:

EXPRESION	JUSTIFICACION
E	Símbolo inicial, inicia derivación
$\Rightarrow E + T$	Aplicación 1a. regla
$\Rightarrow T + T$	2a. regla, sobre la E
$\Rightarrow F + T$	4a. regla, sobre la T izquierda
$\Rightarrow CF + T$	5a. regla, sobre F
$\Rightarrow 2F + T$	7a. regla
$\Rightarrow 2C + T$	6a. regla
$\Rightarrow 25 + T$	7a. regla
$\Rightarrow 25 + T * F$	3a. regla
$\Rightarrow 25 + F * F$	4a. regla
$\Rightarrow 25 + C * F$	6a. regla, sobre la F izquierda
$\Rightarrow 25 + 3 * F$	7a. regla
$\Rightarrow 25 + 3 * CF$	5a. regla
$\Rightarrow 25 + 3 * 1F$	7a. regla
$\Rightarrow 25 + 3 * 1C$	6a. regla
$\Rightarrow 25 + 3 * 12$	7a. regla

Más adelante veremos una herramienta, los "árboles de derivación", que permiten encontrar más fácilmente y visualizar mejor la derivación de las palabras a partir del símbolo inicial, aunque su formalización es menos directa que la simple derivación paso a paso que hemos mostrado.

4.3. Formalización de las GLC

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

- V es un *alfabeto de variables*, también llamadas *no-terminales*.
- Σ es un *alfabeto de constantes*, también llamadas *terminales*. Suponemos que V y Σ son disjuntos, esto es, $V \cap \Sigma = \emptyset$.
- R, el conjunto de *reglas*, es un subconjunto finito de $V \times (V \cup \Sigma)^*$.
- S, el *símbolo inicial*, es un elemento de V.

Ejemplo.- La gramática de $\{a^n b^n\}$ que presentamos antes se representa formalmente como:

$$(\{S\}, \{a, b\}, \{(S, aSb), (S, ab)\}, S)$$

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

Definición.- Una cadena $\alpha \in (V \cup \Sigma)^*$ es *derivable* a partir de una gramática (V, Σ, R, S) si hay al menos una secuencia de pasos de derivación que la produce a partir del símbolo inicial S, esto es:

$$S \Rightarrow \ldots \Rightarrow \alpha$$

Definición.- El lenguaje L(G) generado por una gramática (V, Σ, R, S) es el conjunto de palabras hechas exclusivamente de constantes, que son derivables a partir del símbolo inicial:

$$L = \{w \in \Sigma^* | S \Rightarrow \ldots \Rightarrow w\}$$

Es posible formalizar la definición de lenguaje aceptado sin tener que recurrir a los puntos suspensivos "...", que son un recurso poco elegante en la formalización matemática desde el punto de vista de que recurren a la imaginación del lector para reemplazarlos por la sucesión que se representa. A continuación damos esta formalización alternativa.

Las reglas permiten establecer una relación entre cadenas en $(V \cup |\Sigma)^*$, 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 $x, y \in (V \cup \Sigma)^*$, tales que $\alpha = xuy$, $\beta = xvy$, y existe una regla $u \rightarrow v$ en R.

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

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

4.4. Diseño de GLC

El problema del diseño de GLC consiste en proponer, dado un lenguaje L, una GLC G tal que su lenguaje generado es exactamente L. Decimos que una GLC G es *correcta* con respecto al lenguaje dado L cuando el lenguaje generado por G no contiene palabras que estén fuera de L, es decir, $\mathcal{L}(G) \subseteq L$, donde $\mathcal{L}(G)$ denota el lenguaje generado por G. Similarmente, decimos que G es *completa* cuando G es capaz de generar al menos las palabras de L, es decir, $L \subseteq \mathcal{L}(G)$ Al diseñar gramáticas, es posible cometer las mismas dos clases de errores que hemos mencionado para el diseño de expresiones regulares y autómatas finitos:

1. Que "sobren palabras", esto es, que la gramática genere algunas palabras que no debería generar. En este caso, la gramática sería *incorrecta*.

2. Que "falten palabras", esto es, que haya palabras en el lenguaje considerado para las que no hay ninguna derivación. En este caso, la gramática sería *incompleta*.

Aún cuando no hay métodos tan sistemáticos para diseñar las GLC como los que vimos para diseñar Expresiones Regulares o Autómatas Finitos, es posible al menos reutilizar gramáticas conocidas, y ya sea modificarlas para ajustar el lenguaje generado, o combinar varias en una sola. Este último es un método particularmente eficaz, en el que profundizaremos en esta sección.

4.4.1. Adaptación de GLC

Muchas veces es posible hacer modificaciones sencillas a una gramática conocida para obtener la del lenguaje requerido. Por ejemplo, supóngase que queremos obtener una gramática que genere el lenguaje $\{a^n b^m | n > m\}$. Una buena idea sería partir de la gramática que hemos visto anteriormente, para el lenguaje similar $\{a^n b^n\}$, cuya gramática tiene las siguientes reglas:

1. $S \to aSb$
2. $S \to ab$

Observamos que es necesario prever alguna regla para producir cualquier cantidad de a's antes de las b's, pues hay palabras como $aaaab$ que necesitan ser generadas. Para esto proponemos una regla $S \to aS$. Aplicando iteradamente esta regla podemos producir palabras como la mencionada:

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaaab$$

Sin embargo, aún añadiendo esta regla subsiste el problema de que podríamos generar palabras incorrectas, pues cualquier palabra con igual cantidad de a's y de b's se genera utilizando únicamente las reglas de la gramática para $\{a^n b^n\}$.

Hay al menos dos maneras de solucionar este problema:

1. Podemos pensar en que la a que asegura que haya más a's que b's se produzca al inicio de la derivación, mediante la inclusión de un nuevo símbolo inicial, sea S_0, que produce aS, mediante una regla $S_0 \to aS$. Por ejemplo, generaríamos $aaaab$ del modo siguiente:
$$S_0 \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaaab$$
2. Otra manera es producir la a que garantiza más a's que b's al final de la derivación, reemplazando la regla $S \to ab$ por $S \to a$. La misma palabra se derivaría como:
$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaSb \Rightarrow aaaab$$

4.4.2. GLC para unión de lenguajes

Muchos lenguajes pueden ser expresados en forma útil como la unión de otros dos lenguajes, para los cuales conocemos las gramáticas que los generan. Por ejemplo, el lenguaje $\{a^n b^m | n \neq m\}$ se puede expresar como la unión de los lenguajes:

$$\{a^n b^m | n \neq m\} = \{a^n b^m | n < m\} \cup \{a^n b^m | n > m\}$$

Para cada uno de los lenguajes que se unen es fácil obtener una gramática –de hecho ya hemos diseñado aquí gramáticas para lenguajes como éstos.

La manera de combinar dos gramáticas con símbolos iniciales S_1 y S_2 respectivamente, para producir la unión de los lenguajes originales, consiste en crear un nuevo símbolo inicial S (S_1 y S_2 dejan de ser iniciales), tomar las reglas tanto de una gramática como de otra, y añadir dos nuevas reglas, $S \to S_1$ y $S \to S_2$, para que el nuevo símbolo inicial sea capaz de generar cualquiera de los dos antiguos símbolos iniciales; a partir del primer paso, se continúa la derivación utilizando alguna de las dos gramáticas originales, sin utilizar las reglas de la otra. Para garantizar esto último se supone que las dos gramáticas originales no tienen ninguna variable en común.

Definimos formalmente la gramática que genera la unión de lenguajes de la manera siguiente: 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 las variables de G_1 y G_2 son disjuntas. 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 [8] Similarmente se hace para una palabra $w \in L(G_2)$.

Por ejemplo, para el lenguaje $\{a^n b^m | n \neq m\} = \{a^n b^m | n < m\} \cup \{a^n b^m | n > m\}$, las gramáticas originales tendrían reglas:

$\{a^n b^m \mid n > m\}$	$\{a^n b^m \mid n < m\}$
1) $S_1 \to aS_1b$	4) $S_2 \to aS_2b$
2) $S_1 \to aS_1$	5) $S_2 \to S_2b$
3) $S_1 \to a$	6) $S_2 \to b$

La gramática combinada tendría las reglas 1-6, mas las reglas $S \to S_1$ y $S \to S_2$. El símbolo inicial es S. Así, por ejemplo, para derivar la palabra *aaaab* seguimos los pasos:

[8] recuérdese que por hipótesis $w \in L(G_1)$.

$$S \Rightarrow S_1 \Rightarrow aS_1 \Rightarrow aaS_1 \Rightarrow aaaS_1 \Rightarrow aaaaS_1 \Rightarrow aaaab$$

4.4.3. Mezcla de gramáticas

En ocasiones es necesario combinar dos gramáticas, de una manera similar a la unión que acabamos de presentar, pero permitiendo que las gramáticas a combinar tengan un mismo símbolo inicial. Llamamos a esto *mezcla* de gramáticas.

Ejemplo.- Diseñar una GLC para el lenguaje $\{a^n b^m, n \leq m \leq 2n\}$, esto es, donde la cantidad de b's está entre la cantidad de a's y el doble de ésta, como en las palabras *aabbb*, *aabb* y *aabbbb*. Una solución es "mezclar" una GLC para el lenguaje $\{a^n b^n$ con otra para el lenguaje $\{a^n b^{2n}$, cuyas GLC son respectivamente:

$\{a^n b^n\}$	$\{a^n b^{2n}\}$
1) $S \rightarrow aSb$	3) $S \rightarrow aSbb$
2) $S \rightarrow \varepsilon$	4) $S \rightarrow \varepsilon$

La GLC "mezclada" contendría simplemente la unión de todas las reglas de las dos gramáticas. [9] Así, por ejemplo, para generar la palabra *aabbb*, se tendría la siguiente derivación:

$$S \Rightarrow_1 aSb \Rightarrow_3 aaSbbb \Rightarrow_2 aabbb$$

En esta derivación puede apreciarse que es posible obtener una palabra "intermedia" entre *aabb* y *aabbbb*, como es *aabbb* simplemente aplicando algunas veces la regla 1, y otras la regla 3, según se requiera para la palabra que hay que derivar.

4.4.4. GLC para la concatenación de lenguajes

En ocasiones un lenguaje L puede ser expresado como la concatenación de otros dos L_1 y L_2, esto es, $L = L_1 L_2$. Por ejemplo, el lenguaje $\{a^n b^m | n > m\}$ puede ser expresado como la concatenación de a^k con $\{a^n b^n\}$, y desde luego es fácil encontrar una gramática para a^k, mientras que la de $\{a^n b^n\}$ ya la conocíamos. [10] Ahora bien, hay una manera de combinar modularmente las gramáticas de L_1 y L_2 para obtener la de L.

[9] Desde luego, siendo las reglas 2 y 4 idénticas, resultan en una sóla regla al unir las gramáticas, pues en los conjuntos no hay repetición.

[10] Ya habíamos obtenido la gramática de $\{a^n b^m | n > m\}$ por modificación de otra gramática, pero el método aquí mostrado tiene la ventaja de que es modular.

En efecto, para obtener las reglas de la nueva gramática, simplemente juntamos las reglas de las originales –las cuales tienen símbolos iniciales S_1 y S_2– y agregamos una nueva regla $S \to S_1 S_2$, haciendo además a S el nuevo símbolo inicial.

Ejemplo.- Definimos el lenguaje de los "prefijos palíndromos" como aquel formado por palabras que tienen una parte izquierda de más de un caracter que es palíndromo (se lee igual de izquierda a derecha que de derecha a izquierda). Por ejemplo, las palabras *aabab*, *aba* y *aabaa* [11] son prefijos palíndromos, mientras que las palabras *baa*, *a* y *abbb* no lo son. Proponer una GLC que genere exactamente el lenguaje de los prefijos palíndromos en el alfabeto $\{a, b\}$.

El problema parece difícil, pero podemos considerar cada palabra de este lenguaje como formada por dos partes: la parte palíndroma y el resto de la palabra. Dicho de otra forma, el lenguaje L_{PP} de los prefijos palíndromos es igual a la concatenación de L_P y L_R, donde L_P es el lenguaje de los palíndromos y L_R genera la parte restante de las palabras. El lenguaje de los palíndromos en $\{a, b\}$ tiene una gramática muy simple, con las siguientes reglas:

1. $S \to aSa$
2. $S \to bSb$
3. $S \to a$ (palíndromos impares)
4. $S \to b$ (palíndromos impares)
5. $S \to \varepsilon$ (palíndromos pares)

Por ejemplo, la palabra *aabaa* se puede derivar de la siguiente manera:

$$S \Rightarrow_1 aSa \Rightarrow_1 aaSaa \Rightarrow_4 aabaa$$

Por otra parte, como la "parte restante" que viene después de la parte palíndroma 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 de concatenación dada más arriba.

Formalmente, si tenemos las gramáticas $G_1 = (V_1, \Sigma_1, R_1, S_1)$ y $G_2 = (V_2, \Sigma_2, R_2, S_2)$, el lenguaje $L(G_1)L(G_2)$ es generado por la siguiente GLC:

$$G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \to S_1 S_2\}, S)$$

.

[11] Esta última puede ser vista de dos maneras distintas como prefijo palíndromo.

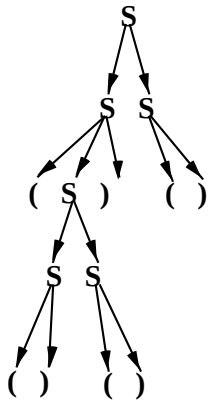


Figura 4.1: Paréntesis bien balanceados

4.5. Arboles de derivación

Las GLC tienen la propiedad de que las derivaciones pueden ser representadas en forma arborescente. Por ejemplo, considérese la gramática siguiente para producir el lenguaje de los paréntesis bien balanceados, que tiene palabras como (()), ()(), (()())(), pero no a (() ni)(: [12]

1. $S \rightarrow SS$
2. $S \rightarrow (S)$
3. $S \rightarrow ()$

Usando esta gramática, la palabra (()())() puede ser derivada de la manera que se ilustra en la figura 4.1. En dicha figura se puede apreciar la *estructura* que se encuentra implícita en la palabra (()())(). A estas estructuras se les llama *árboles de derivación*, o también *árboles de compilación* –por usarse extensivamente en los compiladores– y son de vital importancia para la teoría de los compiladores de los lenguajes de programación.

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 solo árbol de derivación puede haber varias derivaciones lineales, según el orden en que se decida "expandir" los no terminales. Por ejemplo, para el árbol de la figura arriba, hay al menos las derivaciones siguientes (anotamos como subíndice de $\Rightarrow$ el número de regla aplicado):

1. $S \Rightarrow_1 SS \Rightarrow_2 (S)S \Rightarrow_3 (S)() \Rightarrow_1 (SS)() \Rightarrow_3 (S())() \Rightarrow_3 (()())()$.
2. $S \Rightarrow_1 SS \Rightarrow_3 S() \Rightarrow_2 (S)() \Rightarrow_1 (SS)() \Rightarrow_3 (()S)() \Rightarrow_3 (()())()$.

Formalmente, un árbol de derivación es un grafo dirigido arborescente [13] 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 [14]
2. La raíz tiene etiqueta S.
3. La etiqueta de los nodos que no son hojas debe estar en V, y las de las hojas en $\Sigma \cup \{\varepsilon\}$.
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 $A \to A_1, \ldots, A_m \in R$.

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. Así, el problema de "compilar" una cadena de caracteres consiste en construir el árbol de derivación a partir del producto de éste.

4.5.1. Ambigüedad en GLC

La correspondencia entre los árboles de derivación y sus productos no es necesariamente biunívoca. En efecto, hay GLC en las cuales para ciertas palabras hay más de un árbol de derivación. Sea por ejemplo la siguiente GLC, para expresiones aritméticas sobre las variables x y y.

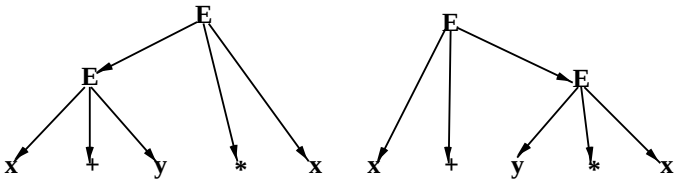
1. $E \to E + E$
2. $E \to E * E$
3. $E \to x$
4. $E \to y$

Con esta gramática, para la expresión $x + y * x$ existen los dos árboles de derivación de las figuras 4.2(a) y (b).

[12] Esta gramática puede ser diseñada adaptando la de $\{a^n b^n\}$, reemplazando a por (y b por), y agregando la primera regla, que toma en cuenta la posibilidad de tener varios grupos de paréntesis anidados.

[13] 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.

[14] 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 $V \cup \Sigma \cup \{\varepsilon\}$.



(a) (b)

Figura 4.2: 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.

Se dice que una gramática es *ambigua* ssi alguna palabra del lenguaje que genera tiene más de un árbol de derivación. Nótese que la ambigüedad, como la estamos definiendo, es una propiedad de la gramática, no de su lenguaje generado. Para un mismo lenguaje puede haber una gramática ambigua y una no ambigua.

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. $F \to x$
7. $F \to y$

Con esta nueva GLC, el árbol de derivación de la figura 4.2(a) se elimina, quedando finalmente una adaptación del árbol de 4.2(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 libre de contexto no ambigua; estos lenguajes se llaman inherentemente ambiguos. Un ejemplo, dado en [7] junto con la prueba correspondiente, es el siguiente:

$$L = \{a^n b^n c^m d^m\} \cup \{a^n b^m c^m d^m\},\ n{\geq}1, m{\geq}1$$

4.5.2. 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 "expandir" en curso de la derivación. Por ejemplo, si tenemos en cierto momento de la derivación 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 solo al no terminal que se encuentre más a la izquierda en la palabra, entonces habrá una sola 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. Similarmente se puede definir una derivación derecha.

Ejemplo.- Para la gramática no ambigua con reglas $S \to AB$, $A \to a$, $B \to b$, la palabra ab se produce con la derivación izquierda:

$$S \Rightarrow AB \Rightarrow aB \Rightarrow ab$$

mientras que también se puede producir con la derivación derecha:

$$S \Rightarrow AB \Rightarrow Ab \Rightarrow ab$$

Teorema.- Para una gramática no ambigua G, y una palabra $w \in L(G)$, existe solamente 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 solo recorrido en preorden para un árbol dado.

4.6. Pruebas de corrección y completez

Es posible en general hacer pruebas matemáticas de que una gramática corresponde a un lenguaje dado. Esto tiene la gran ventaja de que dicha correspondencia ya no es una simple convicción intuitiva, sino que adquiere el rango de certeza matemática. En ciertas aplicaciones, donde es extremadamente importante asegurarse de que no hay errores, las pruebas que garantizan esta certeza son de un gran valor.

Las pruebas que permiten establecer la correspondencia entre un lenguaje y una gramática dados requieren dos partes:

1. Prueba de *corrección*, que garantiza que todas las palabras que se producen al utilizar la gramática efectivamente corresponden a la descripción del lenguaje dado;
2. Prueba de *completez*, que se asegura de que al producir palabras con la gramática, no falten palabras del lenguaje dado.

En general las pruebas de corrección son más sencillas y siguen un patrón más predecible que las de completez, como podremos constatar en los ejemplos que siguen. Las pruebas de corrección se hacen por *inducción*, más precisamente por inducción *en la longitud de la derivación*.

La idea de una prueba por inducción basada en la longitud de la derivación es esencialmente mostrar que todas las palabras por las que se pasa en medio del proceso de derivación cumplen una propiedad, que es básicamente el enunciado del lenguaje. Dichas pruebas siguen el siguiente esquema:

1. Lo primero que hay que hacer es establecer un enunciado, relacionado con la definición del lenguaje considerado, pero algo modificado de manera que se pueda aplicar a las palabras intermedias en el proceso de derivación, las cuales pueden contener variables tanto como constantes.
2. Luego se prueba, como *base de la inducción*, que para las palabras intermedias de la derivación producidas en al menos k_0 pasos, la propiedad se cumple.
3. A continuación se hace el paso de inducción propiamente dicho. Para esto primero se supone que la propiedad se cumple tras haber hecho i pasos de derivación (esto es la *hipótesis de inducción*), y luego se prueba que también se cumple al hacer un paso más de derivación (esto es, para las palabras derivadas en $i+1$ pasos). Al concluir este paso, se ha probado que *todas* las palabras intermedias en el proceso de derivación cumplen con la propiedad.
4. Finalmente, hay que particularizar la propiedad para la *última* palabra de la derivación, que es la que sólo contiene constantes. Con esto se termina la prueba.

Ejemplo.- Probar la corrección de la gramática siguiente que genera el lenguaje P de los paréntesis bien balanceados (presentamos las reglas):

1. $S \to (S)$
2. $S \to SS$
3. $S \to ()$

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:

$$P_X = \{\alpha \in (V \cup \Sigma)^* | elim(S, \alpha) \in P\}$$

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 genera 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, lo que nos da una palabra con los paréntesis bien balanceados. QED

Las pruebas de completez muestran que todas las palabras del lenguaje en cuestión pueden efectivamente ser generadas utilizando la gramática dada. Esto puede ser en ocasiones difícil, y no hay "recetas" tan uniformes como para las pruebas de corrección.

Nótese que la completez y la corrección de una gramática son propiedades independientes, y una gramática dada puede tener una, las dos o ninguna. 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.

Vamos a presentar un ejemplo de prueba de completez para la gramática de los paréntesis bien balanceados dada más arriba, para mostrar el tipo de consideraciones que hay que hacer para llevar a término la prueba.

Prueba de completez.- 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 2 (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: [15].

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

4.7. 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 solo 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".

Las gramáticas sensitivas al contexto son estrictamente más poderosas que las GLC; 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): [16]

[15] El paso de inducción se hace en $k + 2$ y no en $k + 1$ porque todas las palabras en P tienen longitud par

[16] Esta gramática produce palabras de al menos 6 caracteres, o sea de el lenguaje $\{a^n b^n c^n | n > 1\}$.

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, [17] y finalmente las reglas 8 a 10 permiten generar los terminales solamente cuando las letras están en el orden correcto. Como un ejemplo, la palabra *aaabbbccc* se puede generar de la forma siguiente:

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

4.8. Transformación de las GLC y Formas Normales

En muchas situaciones se considera conveniente modificar las reglas de la gramática, de manera que cumplan las reglas con propiedades tales como no producir la cadena vacía del lado derecho, o bien simplemente por cuestiones de estandarización o facilidad de implementación computacional. Desde luego, cuando hablamos de "modificar las reglas de la gramática", se entiende que esto debe hacerse *sin modificar el lenguaje generado.*

Por ejemplo, la presencia de reglas que producen vacío en la gramática puede ser fuente de dificultades tales como la ambigüedad, o la posibilidad de tener derivaciones arbitrariamente largas. Tomemos por ejemplo la siguiente gramática para los paréntesis bien balanceados (damos sólo las reglas):

1. $S \to SS$
2. $S \to (S)$
3. $S \to \varepsilon$

Con esta gramática es posible hacer derivaciones arbitrariamente largas de una palabra tan sencilla como "()"(el subíndice de las flechas indica la regla utilizada):

[17] De hecho bastaría con una regla $BA \to AB$, salvo que ésta no cumple con el formato de las gramáticas sensitivas al contexto.

$$S \Rightarrow_1 SS \Rightarrow_1 SSS \Rightarrow_1 \ldots \Rightarrow_3 SSS \Rightarrow_3 SS \Rightarrow_3 S \Rightarrow_2 (S) \Rightarrow_3 ()$$

Si pudiéramos tener una gramática equivalente, pero sin reglas que produzcan la cadena vacía, ya no sería posible hacer derivaciones arbitrariamente largas. Esto puede ser una ventaja a la hora de determinar si una palabra se deriva o no de una gramática (ver sección 4.10).

4.8.1. Eliminación de reglas $A \to \varepsilon$

Consideremos nuevamente la gramática para los paréntesis bien balanceados. Si queremos una GLC equivalente, pero sin emplear producciones vacías (como $S \to \varepsilon$), una idea sería analizar "en reversa" la derivación de donde 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

$$S \Rightarrow \ldots \Rightarrow \alpha S \beta \Rightarrow \alpha(S)\beta \Rightarrow \alpha()\beta,\ \alpha \in \Sigma^*, \beta \in (\Sigma \cup V)^*$$

mejor hacer directamente la derivación

$$S \Rightarrow \ldots \Rightarrow \alpha S \beta \Rightarrow \alpha()\beta$$

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

$$S \Rightarrow \ldots \Rightarrow \alpha S \beta \Rightarrow \alpha S S \beta \Rightarrow \alpha S \beta$$

por la derivación

$$S \Rightarrow \ldots \Rightarrow \alpha S \beta \Rightarrow \alpha S \beta$$

usando una nueva regla $S \to S$, o mejor aún, simplemente reemplazarla por

$$S \Rightarrow \ldots \Rightarrow \alpha S \beta$$

sin ninguna regla adicional (la parte de la derivación $\alpha S \beta \Rightarrow \alpha S S \beta \Rightarrow \alpha S \beta$ 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 perspicaz: ¡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:

$$S \to (A),\ A \to (A),\ A \to AA,\ A \to \varepsilon$$

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.

4.8.2. 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. A continuación veremos cómo.

Supongamos que hay reglas $A \to B$ y $B \to \Gamma_i$ 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 prueba que la gramática modificada es equivalente a la original.

Por ejemplo, aplicando esta transformación a la gramática del ejemplo, la regla "inútil", que tratamos de eliminar, es $B \to S$. Se producen las nuevas reglas siguientes:

- $B \to (S)$, al combinar $B \to S$ con $S \to (S)$
- $B \to BB$, al combinar $B \to S$ con $S \to BB$
- $B \to ()$, al combinar $B \to S$ con $S \to ()$

La gramática queda entonces con las reglas:

$$S \to (S),\ S \to BB,\ S \to (),\ B \to (S),\ B \to BB,\ B \to ()$$

4.8.3. Eliminación de reglas inaccesibles

Considérese una gramática con reglas:

$$S \to aXbb,\ X \to bSa,\ Y \to SX$$

Es fácil comprender que la tercera regla es inútil, porque no hay nadie que produzca la Y necesaria para que dicha regla se aplique. A reglas como éstas se les llama *inaccesibles*.

Definición.- Una regla $X \to \alpha$ de una gramática (V, Σ, R, S) es inaccesible si no hay una derivación $S \Rightarrow \alpha_1 X \alpha_2$, donde $\alpha_1, \alpha_2 \in (V \cup \Sigma)^*$.

En términos prácticos, si vemos que una variable X no aparece en el lado derecho de ninguna regla de la gramática, podemos asegurar sin arriesgarnos que la regla $X \to \alpha$ es inaccesible.

Para eliminar una regla inaccesible no se necesita hacer ninguna otra modificación a la gramática mas que simplemente borrarla. La equivalencia de la gramática sin la regla inaccesible y la original está garantizada por el hecho de que dicha regla no participa en ninguna derivación.

4.8.4. Formas Normales

En ocasiones es necesario expresar una GLC siguiendo un formato más preciso de las reglas que la simple forma $A \to \alpha$. Estos "estándares" 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 $B, C \in V$

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 (llamada "top-down" en inglés), 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: $S \Rightarrow \ldots \Rightarrow w$ ssi:

1. $w \in \Sigma$, hay una regla $S \to w$
2. $w = \alpha\beta$, hay una regla $S \to AB$, con $A \Rightarrow \ldots \Rightarrow \alpha$, y $B \Rightarrow \ldots \Rightarrow \beta$

Por ejemplo, considérese la siguiente gramática para el lenguaje de los paréntesis bien balanceados, en forma normal de Chomsky (damos sus reglas): [18]

1. $S \to XY$
2. $X \to ($
3. $Y \to SZ$
4. $Z \to)$
5. $S \to SS$
6. $S \to XZ$

Supongamos que tenemos una palabra como (())(), y queremos verificar si se puede derivar a partir de esta gramática. Hay que "partir" dicha palabra en dos pedazos, y escoger alguna

[18] Luego veremos cómo calcular esta forma normal.

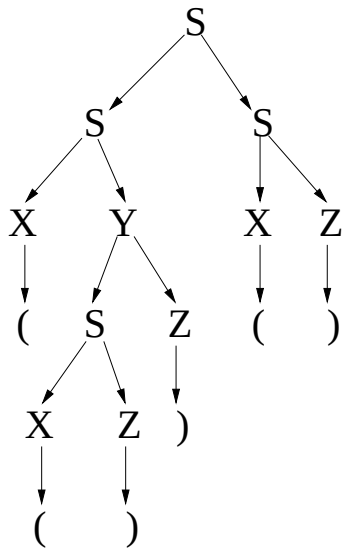


Figura 4.3: Arbol de la palabra (())()

regla que produzca dos variables. Escogemos la quinta regla, $S \to SS$, y partimos la palabra en los pedazos (()) y (). Para que SS pueda generar (())() ahora se necesitará que la primera S pueda generar (()), y la segunda pueda generar (). Estos son subproblemas muy similares al problema inicial. Tomemos el primero, es decir, a partir de S generar (()). Escogemos la regla $S \to XY$, y partimos la palabra en (y ()). Ahora X tiene la responsabilidad de generar (y Y la de generar ()). Por la segunda regla, X genera directamente (. Ahora tomamos el problema de generar ()) a partir de Y. Escogemos la regla $S \to SZ$, y la separación en los pedazos () y). Entonces Z produce directamente), y queda por resolver cómo S produce (). Para ello, escogemos la regla $S \to XZ$, y finalmente X produce (y Z se encarga de), con lo que terminamos el análisis. El árbol de compilación se presenta en la figura 4.3.

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 $A \Rightarrow \ldots \Rightarrow u$. 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. $A \to a$, con $a \in \Sigma$
2. $A \to \beta$, con $\beta \in VV^*$

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 ve, 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. Si tenemos una regla $X \to X_1X_2 \ldots X_n$, la dividimos en dos reglas, una $X \to X_1W$ y otra $W \to X_2 \ldots X_n$, donde W es una nueva variable, es decir, no debe 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:

$5''$.- $Y \to BW$

$5'''$.- $W \to XX$

Similarmente se puede transformar la regla $6'$, dejando la gramática (reglas $1'$, $2'$, $3'$, $4'$, $5''$, $5'''$, $6''$, $6'''$, 7, 8) en la FNCH.

4.9. Limitaciones de los LLC

En esta sección veremos cómo verificar que un lenguaje dado *no es LLC*. Esto puede ser muy útil, para evitarnos el trabajo de tratar inútilmente de diseñar GLCs de lenguajes que no tienen ninguna. Una herramienta para esto es aplicar el llamado "teorema de bombeo", que se presenta enseguida.

4.9.1. 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 $uv^nxy^nz \in L(G)$ 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 $|V|$.

A continuación examinemos el problema de verificar si los árboles de derivación pueden tener una profundidad mayor que $|V|$.

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 $m^{|V|}$ tendrá necesariamente una profundidad mayor que $|V|$, 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 4.4.

Como se ve, hay un subárbol del árbol de derivación (el triángulo intermedio en la figura 4.4) en el que el símbolo A es la raíz y también una de las hojas. Está claro que ese

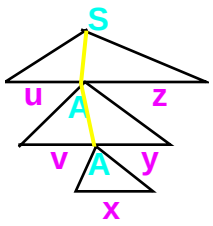


Figura 4.4:

subárbol puede ser insertado o quitado cuantas veces se quiera, y quedará siempre un árbol de 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 y 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.

Al haberse probado que el lenguaje $\{a^n b^n c^n\}$ no es LLC, podemos probar que la intersección de dos LLC no es necesariamente un LLC:

Teorema.- La intersección de dos LLC no es necesariamente LLC.

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. Sin embargo, su intersección es el lenguaje $\{a^n b^n c^n\}$, que acabamos de probar que no es LLC.

Algo similar ocurre con respecto a la operación de complemento del lenguaje, que si se recuerda, en el caso de los lenguajes regulares, su complemento daba otro lenguaje regular:

Teorema.- El complemento de un LLC no necesariamente produce otro LLC.

Prueba.- Si el complemento de un LLC fuera también LLC, lo mismo ocurriría con la

intersección, ya que, de acuerdo con las identidades de la teoría de conjuntos, $L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$. [19]

Debe tenerse cuidado al interpretar estos resultados. En efecto, esto *no* quiere decir, por ejemplo, que el complemento de un LLC necesariamente no será LLC. En el siguiente ejemplo se da un caso específico.

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 vio 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.

4.10. Propiedades de decidibilidad de los LLC

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 *sí* 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:

[19] L^c es una abreviatura para $\Sigma^* - L$.

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 $A \to \alpha$, $|\alpha| > 1$, o bien $\alpha \in \Sigma$, es decir, el lado derecho tiene varios símbolos, o si tiene exactamente un símbolo, éste es terminal."*

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. $S \to ()$
2. $S \to SS$
3. $S \to (S)$

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:
$S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow (((S))) \Rightarrow ((((S)))) \Rightarrow (((((S))))) \Rightarrow ((((()))))$
$S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow ((SS)) \Rightarrow (((S)S)) \Rightarrow ((()S)) \Rightarrow ((()()))$
$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow (()S) \Rightarrow (()(S)) \Rightarrow (()(()))$
$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow ((())S) \Rightarrow ((()))()$
$S \Rightarrow SS \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()((()))$

$S \Rightarrow SS \Rightarrow ()S \Rightarrow ()SS \Rightarrow ()()S \Rightarrow ()()()$
$S \Rightarrow SS \Rightarrow (S)S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())(())$
$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((()))S((()))()$
$S \Rightarrow SS \Rightarrow SSS \Rightarrow ()SS \Rightarrow ()()S \Rightarrow ()()()$

Es fácil ver que éstas son las únicas posibles derivaciones. [20]

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 $A \rightarrow a$, con $a \in \Sigma$. Para empezar observamos que las reglas de la forma $A \rightarrow a$ 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. $E \rightarrow E + E$
2. $E \rightarrow E * E$
3. $E \rightarrow x$
4. $E \rightarrow y$

Esta gramática tiene reglas, como $E \rightarrow x$ y $E \rightarrow y$ 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 ve 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 $A \rightarrow a$ se requieren 5 no-terminales. Esos 5 no-terminales se generan con las reglas de la forma $A \rightarrow a$, 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 da 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

Nótese que en el enunciado del teorema nos estamos restringiendo a las GLC que satisfacen la condición: 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. Cabe preguntarse

[20] Ejercicio: hallar el método que se siguió para obtener las derivaciones mostradas, y probar que no se puede "escapar" ninguna derivación.

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 satisfaga 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.

4.11. Ejercicios

1. Proponer una gramática libre de contexto que genere las palabras binarias que comienzan con 1.

2. Considerar el lenguaje en $\{a, b\}$ en que las palabras tienen la misma cantidad de a's que de b's. Proponer:

   a) Una GLC incorrecta para este lenguaje, esto es, que genere palabras que no debería;

   b) Una GLC incompleta, esto es, que no pueda generar algunas de las palabras de este lenguaje;

   c) Una GLC que sea a la vez incorrecta e incompleta para este lenguaje.

   d) Una GLC correcta y completa;

   e) Una derivación izquierda de la palabra *abaababb* usando esta última gramática.

3. Proponer gramáticas libres de contexto para los siguientes lenguajes:

   a) El lenguaje $\{a^i b^j c^k \mid \neg(i = j = k)\}$

   b) El lenguaje en $\{a, b\}^*$ en que las palabras tienen la misma cantidad de a's y b's.

   c) Las palabras en $\{a, b, c\}$ en que hay más a's que c's (la cantidad de b's puede ser cualquiera).

   d) Un lenguaje de paréntesis, llaves y corchetes bien balanceados. Por ejemplo, las palabras "()[]", "([])" y "()[[]]" son correctas, mientras que "[[]" y "([)]" no lo son. Nótese que en esta última palabra los paréntesis solos están balanceados, así como los corchetes solos, pero su combinación no lo está.

   e) $\{a^i b^j c^k \mid i = j - k\}$

   f) El lenguaje $\{a^n b^{n+m} c^m\}$ (Ayuda: usar la concatenación de lenguajes).

   g) El lenguaje $\{a^n b^k c^m, n \leq k \leq n + m\}$ (Ayuda: usar la mezcla de gramáticas, y la solución al problema anterior).

4. Transformar las gramáticas del problema 3 a la forma normal de Chomsky. Para esto,

a) Eliminar las producciones vacías,

b) Las reglas "inútiles",

c) Las reglas inaccesibles,

d) Poner en la "forma intermedia" (sólo variables o sólo constantes en el lado derecho de las reglas).

e) Limitar a 2 la longitud máxima del lado derecho de las reglas.

5. Mostrar que la siguiente gramática es / no es ambigua: $G = (V, \Sigma, R, S)$, con:

   $V = \{\text{PROG}, \text{IF}, \text{STAT}\}$

   $\Sigma = \{\text{if}, \text{then}, \text{else}, \text{condición}, \text{stat}\}$

   $R = \{\text{PROG} \rightarrow \text{STAT},\ \text{STAT} \rightarrow \text{if condición then STAT},$
   $\text{STAT} \rightarrow \text{if condición then STAT else STAT},\ \text{STAT} \rightarrow \text{stat}\}$

   $S = \text{PROG}$

6. Contestar las siguientes preguntas, justificando la respuesta:

   a) ¿La concatenación de un lenguaje regular con uno 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?

   f) ¿La unión de un lenguaje libre de contexto con un lenguaje regular es libre de contexto?

   g) ¿La intersección de un lenguaje libre de contexto con un lenguaje regular es regular?

   h) ¿El reverso de un lenguaje libre de contexto es también libre de contexto? (Ayuda: considerar una transformación para obtener el reverso del lado derecho de las reglas).

7. Probar la corrección de las gramáticas propuestas en el ejercicio 3. Poner especial cuidado al generar el enunciado generalizado, así como al aplicarlo a los casos especiales.

8. 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.

9. Probar 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. (Ayuda: las cadenas v e y se pueden repetir 0 veces).

10. Llamamos "útil" a un símbolo no terminal A de una gramática libre de contexto que cumple con dos propiedades:

a) $S \Rightarrow^* aAb$, $a, b \in (V \cup \Sigma)^*$, donde V es el alfabeto de las variables y Σ (terminales y no terminales),

b) $A \Rightarrow^* w$, $w \in \Sigma^*$.

Dada una cierta GLC y un símbolo no terminal A, ¿Es decidible si A es útil o no lo es? Pruebe la respuesta, y en caso afirmativo proponga el método de decisión.

11. ¿El lenguaje $\{w = a^i b^m c^n \mid i > m > n\}$ es libre de contexto? Probar la respuesta.

Capítulo 5

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 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 $1, 2, 1, 0, 1, 0$. Recomendamos al lector tratar de diseñar en detalle la tabla describiendo las transiciones del autómata.

Como un segundo ejemplo, considérese el lenguaje de los palíndromos (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 aprecia en la figura 5.1. A estos nuevos autómatas con una pila auxiliar los llamaremos

[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.

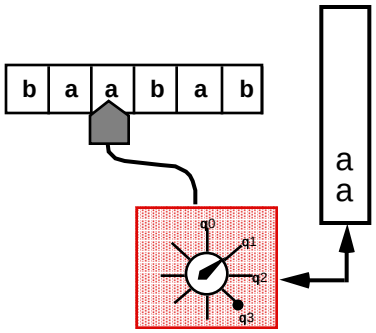


Figura 5.1: Autómata con una pila auxiliar

Autómatas de Pila, abreviado AP.

5.1. Funcionamiento de los Autómatas de Pila (informal)

La *pila* funciona de manera que el último caracter que se almacena en ella es el primero en salir ("LIFO" por las siglas en inglés), 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 su "tope", 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.

Al iniciar la operación de un AP, la pila se encuentra vacía. Durante la operación del AP, la pila puede ir recibiendo (y almacenando) caracteres, según lo indiquen las transiciones ejecutadas. Al final de su operación, para aceptar una palabra, la pila debe estar nuevamente vacía.

En los AP las transiciones de un estado a otro indican, además de los caracteres que se consumen de la entrada, también lo que se saca del tope de la pila, asi como también lo que se mete a la pila.

Antes de formalizar los AP, vamos a utilizar una notación gráfica, parecida a la de los diagramas de los autómatas finitos, como en los AP de las figuras 5.2 (a) y (b). Para las transiciones usaremos la notación "$w/\alpha/\beta$", donde w es la entrada (secuencia de caracteres) que se consume, α es lo que se saca de la pila, y β lo que se mete a la pila.

Por ejemplo, la transición "$a/\varepsilon/b$" indica que se consume de la entrada un caracter a, no

se saca nada de la pila, y se mete b a la pila. Se supone que primero se ejecuta la operación de sacar de la pila y luego la de meter.

Al igual que los AF, los AP tienen estados *finales*, que permiten distinguir cuando una palabra de entrada es aceptada.

De hecho, para que una palabra de entrada sea aceptada en un AP se deben cumplir todas las condiciones siguientes:

1. La palabra de entrada se debe haber agotado (consumido totalmente).
2. El AP se debe encontrar en un estado final.
3. La pila debe estar vacía.

5.2. Diseño de AP

El problema de diseño de los AP consiste en obtener un AP M que acepte exactamente un lenguaje L dado. Por *exactamente* queremos decir, como en el caso de los autómatas finitos, que, por una parte, todas las palabras que acepta efectivamente pertenecen a L, y por otra parte, que M es capaz de aceptar todas las palabras de L.

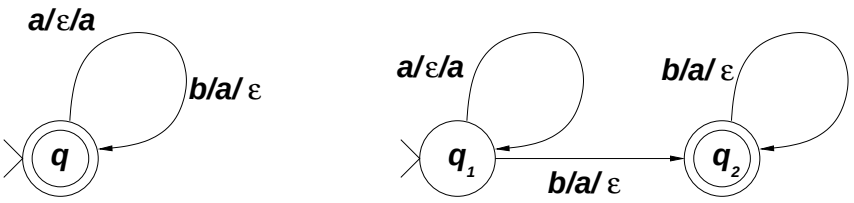
Aunque en el caso de los AP no hay metodologías tan generalmente aplicables como era el caso de los autómatas finitos, siguen siendo válidas las ideas básicas del diseño sistemático, en particular establecer claramente qué es lo que "recuerda" cada estado del AP antes de ponerse a trazar transiciones a diestra y siniestra. Para los AP, adicionalmente tenemos que establecer una estrategia clara para el manejo de la pila.

En resumen, a la hora de diseñar un AP tenemos que repartir lo que requiere ser "recordado" entre los estados y la pila. Distintos diseños para un mismo problema pueden tomar decisiones diferentes en cuanto a qué recuerda cada cual.

Ejemplo.- Diseñar un AP que acepte exactamente el lenguaje con palabras de la forma $a^n b^n$, para cualquier número natural n.

Una idea que surge inmediatamente es la de utilizar la pila como "contador" para recordar la cantidad de a's que se consumen, y luego confrontar con la cantidad de b's. Una primera versión de este diseño utiliza un sólo estado q, con transiciones $a/\varepsilon/a$ y $b/a/\varepsilon$ de q a sí mismo, como en la figura 5.2(a).

Para verificar el funcionamiento del autómata, podemos simular su ejecución, listando las situaciones sucesivas en que se encuentra, mediante una tabla que llamaremos "traza de ejecución". Las columnas de una traza de ejecución para un AP son: el estado en que se



(a) Incorrecto (b) Correcto

Figura 5.2: AP para el lenguaje $a^n b^n$

encuentra el autómata, lo que falta por leer de la palabra de entrada, y el contenido de la pila.

Por ejemplo, la traza de ejecución del AP del último ejemplo, para la palabra *aabb*, se muestra a continuación: [3]

Estado	**Por leer**	**Pila**
q	$aabb$	ε
q	abb	a
q	bb	aa
q	b	a
q	ε	ε

Concluímos que el AP efectivamente puede aceptar palabras como $a^n b^n$. Sin embargo, hay un problema: ¡el AP también acepta palabras como *abab*, que no tienen la forma deseada! (es fácil construir la traza de ejecución correspondiente para convencerse de ello). El problema viene de que no hemos recordado cuando se terminan las a y principian las b, por eso ha sido posible mezclarlas en *abab*. Una solución es utilizar los estados para memorizar las situaciones de estar consumiendo a o estar consumiendo b. El diagrama de estados correspondiente se muestra en la figura 5.2(b).

Ejemplo.- Proponer un AP que acepte el lenguaje de los palíndromos con un número par de símbolos, esto es, palabras que se leen igual de izquierda a derecha y de derecha a izquierda, y que tienen por tanto la forma ww^R, donde w^R es el *reverso* de w (esto es, invertir el orden), en el alfabeto $\{a, b\}$. Por ejemplo, las palabras *abba*, *aa* y *bbbbbb* pertenecen a este lenguaje, mientras que *aab* y *aabaa* no.

Una estrategia de solución para diseñar este AP sería almacenar en la pila la primera mitad de la palabra, y luego irla comparando letra por letra contra la segunda mitad. Tendríamos dos estados s y f, para recordar que estamos en la primera o segunda mitad de la palabra. En la figura 5.2 se detalla este AP.

[3] Suponemos que el tope de la pila está del lado izquierdo, aunque en este ejemplo da lo mismo.

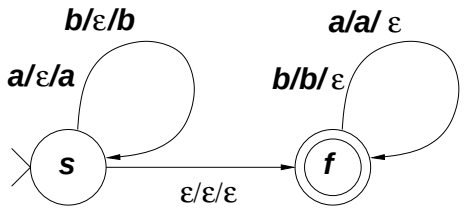


Figura 5.3: AP para el lenguaje $\{ww^R\}$

Se puede apreciar en el AP de dicha figura la presencia de una transición de s a f, en que ni se consumen caracteres de la entrada, ni se manipula 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

5.2.1. Combinación modular de AP

En los AP también es posible aplicar métodos de combinación modular de autómatas, como hicimos con los autómatas finitos. En particular, es posible obtener AP que acepten la unión y concatenación de los lenguajes aceptados por dos AP dados.

En el caso de la unión, dados dos AP M_1 y M_2 que aceptan respectivamente los lenguajes L_1 y L_2, podemos obtener un AP que acepte la unión $L_1 \cup L_2$, introduciendo un nuevo estado inicial s_0 con transiciones $\varepsilon/\varepsilon/\varepsilon$ a los dos antiguos estados iniciales s_1 y s_2, como se ilustra en la figura 5.4. [4]

[4]El procedimiento de combinación de AP para obtener la unión de autómatas puede ser descrito en forma más precisa utilizando la representación formal de los AP, que se estudia en la siguiente sección; sin embargo, hacer esto es directo, y se deja como ejercicio (ver sección de ejercicios).

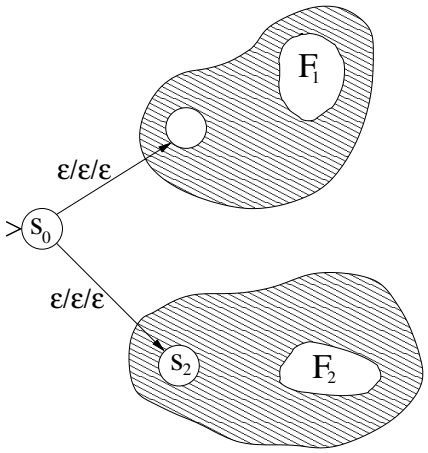


Figura 5.4: Unión de AP

Ejemplo.- Obtener un AP que acepte el lenguaje $\{a^n b^m | n \neq m\}$. Claramente este lenguaje es la unión de $\{a^n b^m | n > m\}$ con $\{a^n b^m | n < m\}$, por lo que basta obtener los AP de cada uno de ellos, y combinarlos con el método descrito.

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.

También es posible obtener modularmente un AP que acepte la concatenación de los lenguajes aceptados por dos AP dados. De hecho ya vimos en el capítulo 4 que la unión de dos lenguajes libres de contexto es también libre de contexto, pues tiene una gramática libre de contexto.

Sin embargo, la construcción de un AP que acepte la concatenación de dos lenguajes a partir de sus respectivos AP M_1 y M_2, es ligeramente más complicada que para el caso de la unión. La idea básica sería poner transiciones vacías que vayan de los estados finales de M_1 al estado inicial de M_2. Sin embargo, existe el problema: hay que garantizar que la pila se encuentre vacía al pasar de M_1 a M_2, pues de otro modo podría resultar un AP incorrecto. Para esto, es posible utilizar un caracter especial, por ejemplo "@", que se mete a la pila antes de iniciar la operación de M_1, el cual se saca de la pila antes de iniciar la operación de M_2. Los detalles se dejan como ejercicio (ver sección de ejercicios).

5.3. 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 de la forma $((p, u, \beta), (q, \gamma)) \in \Delta$, 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 –típicamente llamadas en inglés 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 $\vdash^*$.

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.- Formalizar el AP de la figura 5.2, que acepta el lenguaje $\{ww^R\}$, $w \in \{a, b\}$.

Solución.- El AP es el séxtuplo $(K, \Sigma, \Gamma, \Delta, s, F)$, donde

$K = \{s, f\}, F = \{f\}, \Sigma = \{a, b, c\}, \Gamma = \{a, b\}$

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)

5.4. 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.

5.5. 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.

Vamos a examinar la prueba de esta afirmación, no solamente por el interés por la rigurosidad matemática, sino sobre todo porque provee un método de utilidad práctica para transformar una GLC en un AP. 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 consideramos de mayor relevancia práctica. La otra parte de la prueba (que también es un procedimiento de conversión) puede consultarse en la referencia [21].

Sea una gramática $G = (V, \Sigma, R, S)$. Entonces un AP M que acepta exactamente el lenguaje generado por G se define como sigue:

$$M = (\{p, q\}, \Sigma, V \cup \Sigma, \Delta, p, \{q\})$$

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 solamente 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 solo 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 solo 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:

$$S \Rightarrow^* w \text{ ssi } [[p, w, \varepsilon]] \vdash^*_{M(G)} [[q, \varepsilon, \varepsilon]]$$

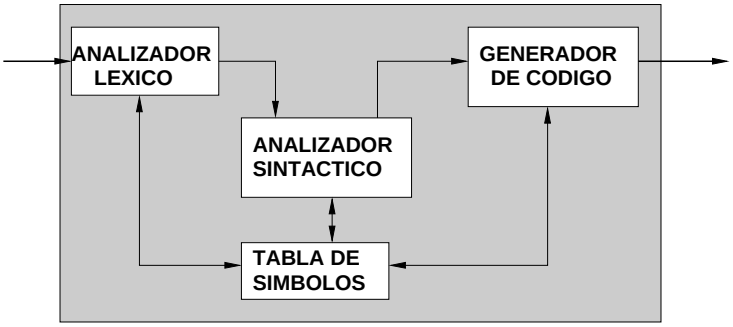


Figura 5.5: Diagrama de un compilador

donde $M(G)$ denota al AP construido 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 [10].

La equivalencia de los AP y de las GLC permite aplicar todas las propiedades de los LLC para resolver problemas de diseño de AP.

5.6. Compiladores LL

El método que hemos visto 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.

De una manera muy general, un compilador –como los que se usan para traducir un lenguaje de programación al lenguaje de máquina– está compuesto por las partes que se ilustran en la figura 5.5. Sus principales partes son:

- Un *analizador léxico*, que recibe los caracteres del archivo de entrada, y entrega los llamados "tokens", que representan los elementos del lenguaje –tales como las palabras claves (como "begin", "integer", etc.), los operadores (tales como "+"), los identificadores propuestos por el usuario, y otros elementos. Generalmente varios caracteres corresponden a un sólo "token". Así, los demás componentes del compilador ya no consideran la entrada como una secuencia de caracteres, sino como una secuencia de "tokens". Un beneficio adicional del analizador léxico es que "filtra" caracteres inútiles desde el punto de vista de la traducción que se quiere hacer, como por ejemplo los comentarios del programador. El analizador léxico puede ser considerado como un autómata con salida (como los autómatas de Moore y de Mealy de la sección 2.7), y son muchas veces construidos a partir de la definición de "tokens" mediante Expresiones Regulares.

- Un *analizador sintáctico*, que toma como entrada los "tokens" y verifica que su secuencia corresponde a la definición del lenguaje dada por medio de una gramática libre de contexto. Mediante el uso de herramientas adecuadas, como el generador de compiladores "yacc" [9], es posible producir un analizador sintáctico a partir de la definición del lenguaje mediante una gramática.
- Un *generador de código*, que guiado por el analizador sintáctico, produce realmente el resultado de la compilación, que es la traducción del lenguaje fuente al lenguaje deseado (generalmente lenguaje ensamblador).
- Una *tabla de símbolos*, que registra las definiciones de identificadores dadas por el usuario en su programa, y las utiliza posteriormente para resolver las referencias que se hacen a ellos en el programa a traducir.

Para una descripción detallada de los compiladores y de las técnicas usadas para construirlos, véase la referencia [1].

5.6.1. Principio de previsión

Desde luego, para tener un verdadero compilador se requiere que se trate de un AP 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 $S \to aSb$, $S \to \varepsilon$, 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
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, fue 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 que ha de ser leído (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. [5]

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

[5] Ejercicio: hacer nuevamente la traza de ejecución para la palabra abb, utilizando la tabla de previsión.

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):

1. $S \to aA$
2. $A \to Sb$
3. $A \to b$

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:

Estado	Falta leer	Pila	Comentario
p	$aabb$	ε	
q	$aabb$	S	
q	$aabb$	aA	
q	abb	A	Se decide reemplazar A por Sb.
q	abb	Sb	
q	abb	aAb	
q	bb	Ab	Se decide reemplazar A por b.
q	bb	bb	
q	b	b	
q	ε	ε	

La tabla de previsión entonces debe haber sido:

	a	b	ε
S	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.- Supongamos una gramática (V, Σ, R, S). El operador $first : (V \cup \Sigma)^+ \rightarrow 2^\Sigma$, cuyo argumento es una cadena de símbolos (al menos uno) que puede contener variables y constantes, y cuyo resultado es un conjunto de caracteres, obtiene todos los caracteres con los que puede empezar una cadena derivable a partir de su argumento. Por ejemplo, para la GLC con reglas $S \rightarrow aA$, $A \rightarrow Sb$, $A \rightarrow b$, nos damos cuenta de que las cadenas que se pueden derivar a partir de S tienen que empezar con a, porque lo único que puede producir S es aA, que empieza con a. Por ello, $first(S) = \{a\}$.

$first(\alpha)$ se calcula sistemáticamente a partir de las siguientes propiedades:

- Si $\alpha = \sigma$, $\sigma \in \Sigma$, entonces $first(\alpha) = \{\sigma\}$
- Si $\alpha = xv$, $x \in (V \cup \Sigma)$, $v \in (V \cup \Sigma)^*$, $first(\alpha) = first(x)$
- Si $A \in V$, $first(A) = first(\alpha_1) \cup \ldots \cup first(\alpha_n)$, para todas las reglas $A \rightarrow \alpha_i$, para $i = 1 \ldots n$.

Ejemplos.- Seguimos utilizando la gramática $S \rightarrow aA$, $A \rightarrow Sb$, $A \rightarrow b$.

- $first(aA) = first(a) = \{a\}$, aplicando la segunda y luego primera regla.
- $first(Ab) = first(A)$ por la segunda regla; $first(A) = first(b) \cup first(Sb)$ por la tercera regla, y $first(Sb) = first(S) = first(aA) = first(a) = \{a\}$, por lo que finalmente $first(Ab) = \{a, b\}$.

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 variable 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:

$\ldots$	σ	$\ldots$	ε
$\ldots$	$\ldots$	$\ldots$	$\ldots$
X	α	$\ldots$	$\ldots$

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 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 a 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 $\sigma \in first(\alpha)$ y $\sigma \in first(\beta)$, entonces tanto α como β 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).

5.7. Compiladores LR(0)

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" cadenas que correspondan al lado derecho de una regla gramatical, para reemplazar por el lado izquierdo. Así, estos compiladores encuentran la derivación "en reversa", reemplazando lados derechos de reglas por lados izquierdos, hasta llegar eventualmente al símbolo inicial. Entonces, los compiladores LR recorren el árbol de derivación *de abajo hacia arriba*, por lo que se llaman también compiladores *ascendentes*.

De hecho, el reconocimiento del lado derecho de una regla no se hace respecto a la entrada, sino respecto al tope de la pila, pero para esto se necesita primero haber pasado caracteres de la entrada a la pila. Las dos operaciones básicas de un compilador LR son:

Desplazamiento que consiste en pasar un caracter de la entrada al tope de la pila,

Reducción que consiste en reemplazar en el tope de la pila el lado derecho de una regla gramatical por el lado izquierdo de la misma. [6]

[6] De hecho se toma el *reverso* del lado derecho de una regla, ver más adelante.

Estas operaciones se aplican, en un orden "adecuado", hasta que en la pila quede sólo el símbolo inicial. Desde luego, la dificultad está en encontrar las secuencias de desplazamientos y reducciones que llevan a la situación final deseada. La idea de cómo combinar los desplazamientos con las reducciones se comprende en el siguiente ejemplo: Supongamos la gramática para el lenguaje $\{a^n b^n\}$ con las reglas $S \to aA$, $A \to Sb$, $A \to b$. Dada la palabra $aabb$, se tendría una traza de ejecución como sigue:

Falta leer	Pila	Acción
$aabb$	ε	Desplazar
abb	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

Obsérverse, en el quinto renglón de la tabla, que en el tope de la pila está la cadena Aa, mientras que el lado derecho de la regla que corresponde es aA. Vemos así que lo que se saca de la pila es el lado derecho de la regla, pero "volteado" de izquierda a derecha; técnicamente decimos que el *reverso* del lado derecho de una regla coincide con el tope de la pila. Esto se refleja en las definiciones que damos en seguida.

En este ejemplo en particular es relativamente fácil discernir cuando hacer cada una de las acciones. Sin embargo, en otros ejemplos es mucho más difícil determinar qué acción llevar a cabo; 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.
- Hay una transición $((i, \varepsilon, \varepsilon), (p, \#)) \in \Delta$. Esta transición coloca un "marcador" # en el fondo de la pila, para luego reconocer cuando la pila se ha vaciado.
- Se tienen transiciones $((p, \sigma, \varepsilon), (p, \sigma)) \in \Delta$ para cada $\sigma \in \Sigma$. Estas transiciones permiten hacer la acción de desplazar.
- Hay transiciones $((p, \varepsilon, \alpha^R), (p, A)) \in \Delta$ para cada regla $A \to \alpha \in R$, donde α^R es el *reverso* de α, esto es, α "volteado" de izquierda a derecha. Estas transiciones efectúan las reducciones.

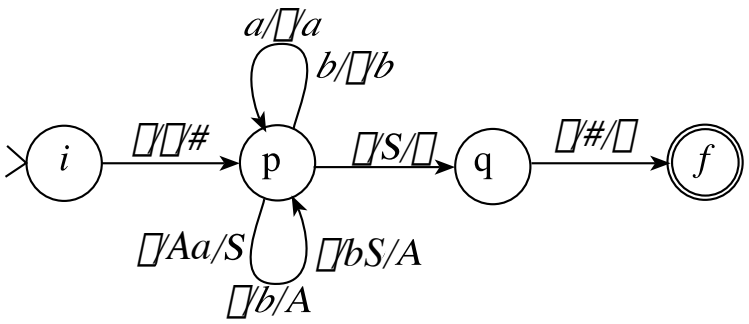


Figura 5.6: AP de tipo LR

- Tenemos una transición $((p, \varepsilon, S), (q, \varepsilon)) \in \Delta$; esta transición reconoce cuando se llegó al símbolo inicial.
- Finalmente hay una transición $((q, \varepsilon, \#), (f, \varepsilon)) \in \Delta$; esta transición se asegura de que se haya vaciado la pila antes de aceptar la palabra.

Este procedimiento es directo. Por ejemplo, en la figura 5.6 se representa el autómata correspondiente a la GLC que hemos estado considerando, esto es, $S \to aA$,$A \to Sb$, $A \to b$. Más que en la construcción del AP, las dificultades pueden estar en el uso del AP, pues al ser éste no determinista, en ciertas situaciones puede presentarse un conflicto, en que no se sepa si desplazar un caracter más a la pila o bien reducir por alguna regla. Y aún en este caso puede haber varias reducciones posibles. Por ejemplo, sugerimos hacer la traza de ejecución en el AP de la figura 5.6 de la palabra *aabb*.

Una posible solución a estos conflictos puede ser adoptar una política en que –por ejemplo– la reducción tenga preferencia sobre el desplazamiento. Esto funciona adecuadamente en el ejemplo recién presentado, pero no funciona en otros casos. En el caso general, es necesario usar técnicas más complicadas, que incluyen previsión de caracteres (esto es, LR(1), LR(2), etc.); dichas técnicas pueden consultarse en la referencia [1].

5.8. Ejercicios

1. Sea un autómata de pila $M = (K, \Sigma, \Gamma, \Delta, s, F)$ que acepta el lenguaje de paréntesis bien formados, incluyendo los paréntesis redondos "(", ")", así como los paréntesis cuadrados "[", "]", es decir: $L(M) = \{e, (), [], ()[], []()), (()), ([]), [()], [[]], \ldots\}$.

   a) Dibujar el diagrama del AP que acepta el lenguaje descrito.

   b) Representar formalmente, dando K, Σ, Γ, Δ, s y F.

   c) Dar un cálculo producido por la palabra errónea "([]", con las columnas "Estado", "Por leer" y "pila", como en los ejemplos dados.

2. Proponga un autómata de pila para el lenguaje:
   $\{a^i b^j c^k \mid i = j - k\}$ [7]

3. Considere el lenguaje en $\{a, b\}^*$ en que las palabras tienen la misma cantidad de a's que de b's, que es generado por la siguiente gramática:

   1.- $S \rightarrow aSb$

   2.- $S \rightarrow bSa$

   3.- $S \rightarrow SS$

   4.- $S \rightarrow \varepsilon$

   a) Diseñar directamente (sin convertir a partir de una gramática) un AP que acepte dicho lenguaje, usando una pila que almacene el exceso de a's o de b's (basta con dibujar el diagrama de estados).

   b) Construir otro AP, convirtiendo la GLC dada a AP de tipo LL.

   c) Lo mismo que el anterior, para un AP de tipo LR.

   d) Para cada uno de los incisos anteriores, hacer una traza de ejecución para la palabra "*abbaba*", en forma de una tabla, usando las columnas "estado", "por leer", "pila", y "acción aplicada".

4. 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:

   1) $<AD> \rightarrow\ a\ <AD>\ d$
   2) $<AD> \rightarrow\ b\ <BD>\ d$
   3) $<AD> \rightarrow\ e$
   4) $<AD> \rightarrow\ a\ <AC>\ c$
   5) $<BD> \rightarrow\ b\ <BD>\ d$
   6) $<BD> \rightarrow\ b\ <BC>\ c$
   7) $<BD> \rightarrow\ e$
   8) $<BC> \rightarrow\ b\ <BC>\ c$
   9) $<BC> \rightarrow\ e$
   10) $<AC> \rightarrow\ a\ <AC>\ c$
   11) $<AC> \rightarrow\ b\ <BC>\ c$
   12) $<AC> \rightarrow\ e$

   El símbolo inicial es $<AD>$. Pruebe la corrección de la GLC por inducción sobre la longitud de la derivación.

[7] Ayuda: al despejar en la ecuación la j el problema parece ser más fácil, pues permite aplicar un diseño modular.

c) Obtenga un AP de tipo LL(1) a partir de esta gramática.

d) Lo mismo, para un AP de tipo LR(0).

e) Para los dos incisos anteriores, obtener una traza de ejecución, en forma de tabla con columnas "estado", "por leer", "pila" y "acción", para la palabra de entrada "*aaacdd*".

5. Para el AP de la figura 5.6, y la palabra de entrada *aabb*:

   a) Construir una traza de ejecución, con columnas "estado", "falta leer", "pila", y "acción".

   b) Localizar los renglones de la tabla anterior donde se presenta un conflicto, e indicar si es de desplazamiento contra reducción o bien de una reducción contra otra reducción.

6. 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.

   g) Indique si hay o no conflictos "shift-reduce" o "reduce-reduce" en el AP del inciso anterior, utilizando la traza de ejecución para la palabra de entrada *abbbcc*. ¿Es posible escribiendo resolver los conflictos con los criterios de preferir "reduce" a "shift", para los conflictos shift-reduce, y en caso de conflicto reduce-reduce preferir la transición que reduzca más símbolos?

7. Completar y detallar formalmente el procedimiento de combinación modular de AP para la concatentación de lenguajes, delineado en la sección 5.2.1.

8. Formalice el procedimiento para obtener un AP que acepte la unión de los lenguajes aceptados respectivamente por $(K_1, \Sigma_1, \Gamma_1, \Delta_1, s_1, F_1)$ y $(K_2, \Sigma_2, \Gamma_2, \Delta_2, s_2, F_2)$.

9. Considere una variante de los autómatas pushdown, que podríamos llamar "autómatas de fila", en los que en vez de la pila, que se accesa en orden UEPS ("LIFO"), se tiene una fila que se accesa en orden PEPS ("FIFO").

   a) Dé una definición formal de los autómatas de fila.

   b) Pruebe que el lenguaje $\{a^n b^n\}$ es aceptado por algún autómata de fila.

   c) ¿Piensa que los autómatas de pila y de fila son equivalentes? Justifique de manera informal.

10. Considere una variante de los autómatas de pila, 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 $\{a^n b^n\}$.

11. Proponga máquinas lo menos poderosas que sea posible para que acepten los siguientes lenguajes:

    a) $\{(), [], <>, ([]), [< ()(<>) > ()], \ldots\}$

    b) $\{(), (()), ((())), (((()))), \ldots\}$

    c) $\{(), ()(), ()()(), \ldots\}$

12. 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. (Ayuda: 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.

Parte III

Máquinas de Turing y sus lenguajes

Capítulo 6

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 6.1(a)), algunas ideas podrían ser:

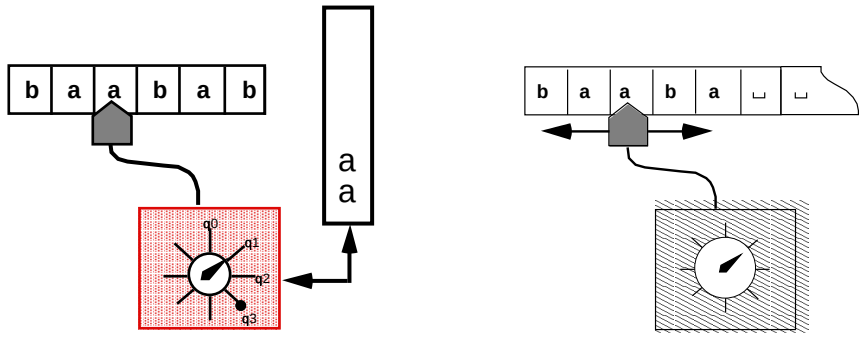
1. Añadir otra pila;
2. Poner varias cabezas lectoras de la entrada;
3. Permitir la escritura en la cinta, además de la lectura de caracteres.

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*.

A. Turing propuso [24] en los años 30 un modelo de máquina abstracta, como una extensión de los autómatas finitos, que resultó ser de una gran simplicidad y poderío a la vez. La máquina de Turing es particularmente importante porque es la más poderosa de todas las máquinas abstractas conocidas (esto último será discutido en la sección 6.5).

6.1. Funcionamiento de la máquina de Turing

La máquina de Turing (abreviado MT, ver figura 6.1(b)) tiene, 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 de longitud infinita hacia la derecha, hacia donde se extiende indefinidamente, llenándose los espacios con



(a) Autómata de pila (b) Máquina de Turing

Figura 6.1:

el caracter blanco (que representaremos con "⌴"). La cinta *no es infinita* hacia la izquierda, por lo que hay un cuadro de la cinta que es el extremo izquierdo, como en la figura 6.1(b).

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 un 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 excluyentes, 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, pero iniciando a partir de la segunda posición de la cinta, siendo el primer cuadro un caracter blanco. Como la cinta es infinita, inicialmente toda la parte de la cinta a la derecha de la palabra de entrada está llena del caracter blanco (⌴).

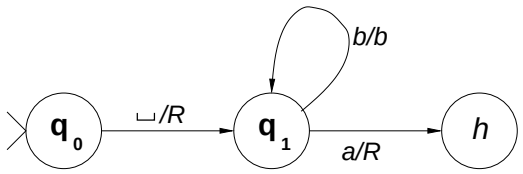


Figura 6.2: MT que acepta palabras que empiezan con a

Por definición, al iniciar la operación de la MT, la cabeza lectora está posicionada en el caracter blanco a la izquierda de la palabra de entrada, el cual es el cuadro más a la izquierda de la cinta.

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. Representaremos al halt por "h". [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.

Cuando queremos que una palabra no sea aceptada, desde luego debemos evitar que la MT llegue al *halt*. Podemos asegurarnos de ello haciendo que la MT caiga en un ciclo infinito (ver ejemplos adelante).

El lenguaje aceptado por una MT es simplemente el conjunto de palabras aceptadas por ella. [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, ver sección 2.7). 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 6.2. Si la primera letra es una "a", la palabra se acepta, y en caso contrario se hace que la MT caiga en un ciclo infinito, leyendo y escribiendo "b". 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] No traduciremos el término "halt", que en inglés significa detener, porque es tradicional usar exactamente este nombre en máquinas de Turing.

[2] Más adelante daremos definiciones formales.

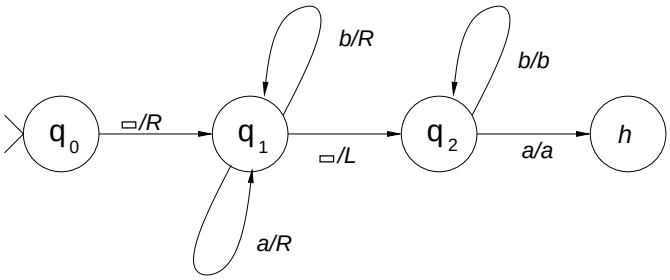


Figura 6.3: MT que acepta palabras que terminan con a

Ejemplo.- Diseñar una MT que acepte las palabras en $\{a, b\}$ que terminen con a. Aunque este ejemplo parece bastante similar al precedente, en realidad es más complicado, pues para ver cual es la última letra, hay que ir hasta el blanco a la derecha de la palabra, luego regresar a la última letra y verificar si es una "a". Una solución se muestra en la figura 6.3.

Ejemplo.- Probar que hay lenguajes que no son libres de contexto, pero que pueden ser aceptados por una máquina de Turing. 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 simplemente los reemplazaremos por un 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 "$\triangle$".

$$
\begin{array}{cccccccc}
\sqcup & a & a & b & b & c & c & \sqcup \\
\triangle & & & & & & & \\
\sqcup & a & a & b & b & c & c & \sqcup \\
 & \triangle & & & & & & \\
\sqcup & * & a & b & b & c & c & \sqcup \\
 & \triangle & & & & & & \\
\sqcup & * & a & b & b & c & c & \sqcup \\
 & & \triangle & & & & & \\
\sqcup & * & a & b & b & c & c & \sqcup \\
 & & & \triangle & & & & \\
\sqcup & * & a & * & b & c & c & \sqcup \\
 & & & \triangle & & & & \\
 & \ldots & & \ldots & & \ldots & & \\
\sqcup & * & a & * & b & * & c & \sqcup \\
 & & & & & & & \triangle \\
 & \ldots & & \ldots & & \ldots & & \\
\sqcup & * & a & * & b & * & c & \sqcup \\
\triangle & & & & & & & \\
\sqcup & * & a & * & b & * & c & \sqcup \\
 & \triangle & & & & & & \\
 & \ldots & & \ldots & & \ldots & &
\end{array}
$$

Lo que falta por hacer es diseñar los estados de la MT, lo cual es relativamente simple y que dejamos como ejercicio (ver sección de ejercicios).

6.2. 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 quíntuplo $(K, \Sigma, \Gamma, \delta, s)$ donde:

- K es un conjunto de estados tal que $h \in K$;
- Σ es el alfabeto de entrada, donde $\sqcup \notin \Sigma$;
- Γ es el alfabeto de la cinta, donde $\sqcup \in \Gamma$ y $\Sigma \subseteq \Gamma$
- $s \in K$ es el estado inicial;
- $\delta : (K - \{h\} \times \Gamma) \to K \times (\Gamma \cup \{L, R\})$ es la función de transición.

La expresión de la función de transición parece algo complicada, pero puede entenderse de la siguiente manera: la función de transición del control finito debe considerar como entradas el estado actual, que es un elemento de K –pero que no puede ser h– así como el caracter leído en la cinta, que es elemento de Γ. Por eso a la izquierda de la flecha aparece la expresión $\delta : (K - \{h\} \times \Gamma)$. Luego, el resultado de la función de transición debe incluir el siguiente estado, que es elemento de K. 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 Γ. Desde luego, para que no haya confusión se requiere que ni L ni R estén en Γ. Resumiendo, el resultado de la función de transición debe ser un elemento de $K \times (\Gamma \cup \{L, R\})$.

Así, si $\delta(q, a) = (p, b)$, donde $b \in \Gamma$, 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 escribirá el caracter b. 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: $K = \{s, h\}$, (sólo está el estado inicial, además del "halt"), $\Sigma = \{a\}$, $\Gamma = \{a, \sqcup\}$, $\delta(s, a) = (s, R)$, $\delta(s, \sqcup) = (h, R)$. 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.

Nótese que, puesto que δ es una función, en principio debe tenerse por cada estado y cada caracter una transición. Esto quiere decir que cada estado debe tener una cantidad de flechas de salida igual a $|\Gamma|$. Por ejemplo, si $\Gamma = \{\sqcup, a, b\}$, y $K = \{q, h\}$, entonces debe haber flechas de salida de q con a, de q con b y también de q con $\sqcup$. [3] En la práctica, sin embargo, una gran cantidad de flechas tiende a hacer menos comprensible un diagrama, por lo que solamente incluiremos las flechas "necesarias", suponiendo en consecuencia que las demás flechas pueden tener una salida cualquiera, sin afectar esto el funcionamiento de la MT. Por ejemplo, a partir del estado inicial podemos suponer, sin arriesgarnos, que no es posible encontrar más que el caracter blanco, por lo que no tiene caso dibujar flechas del estado inicial con a, con b, etc. [4]

6.2.1. 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.

[3] Nótese que h es una excepción, pues no debe tener ninguna flecha de salida.

[4] Desde luego, si se regresa al estado inicial después de haber ejecutado otras transiciones, sí será posible encontrar otros caracteres.

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, no podemos representarla toda por una cadena de caracteres, que siempre será de tamaño finito. Vamos a tomar la solución de tomar en cuenta únicamente la parte de la cinta hasta antes de donde empieza la sucesión infinita de blancos, pues esta última realmente no contiene ninguna información útil. Por ejemplo, en la figura 6.4(a) dicha parte de la cinta es "$\sqcup\ a\ \sqcup\ \sqcup\ b\ b\ a\ b$".

El siguiente problema es cómo caracterizar la posición de la cabeza lectora. Recordemos 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. El problema es que esta solución no funciona, 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 leídos de nuevo o hasta modificados. Otra solución sería representar la posición por un número entero que indicara la posición actual con respecto a alguna referencia. Sin embargo, adoptaremos la solución consistente en dividir la cinta dentro de la configuración en tres pedazos:

- La parte de la cinta a la izquierda de la cabeza, que es un elemento de Γ^*.
- El cuadro en la posición de la cabeza lectora, que es un elemento de Γ.
- La parte de la cinta a la derecha de la cabeza lectora, hasta antes de la sucesión de blancos que se extiende indefinidamente a la derecha.

La parte a la derecha de la cabeza lectora es, desde luego, un elemento de Γ^*, pero podemos hacer una mejor caracterización de ella considerando que el último caracter de ella no es blanco. Así, sería un elemento de $\Gamma^*(\Gamma - \{\sqcup\})$. Sin embargo, hay un problema técnico: esta expresión no incluye la cadena vacía, la cual puede producirse cuando todos los caracteres a la derecha de la cabeza son blancos. La solución es simplemente añadir este

(a) (b)

Figura 6.4: Contenido de la cinta en una configuración

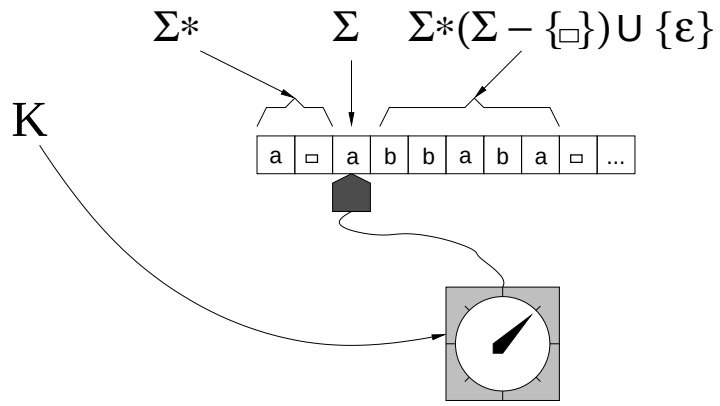


Figura 6.5: Configuración en MT

caso, por lo que finalmente la parte a la derecha de la cabeza lectora es un elemento de $\Gamma^*(\Gamma - \{\sqcup\}) \cup \{\varepsilon\}$.

Por ejemplo, la cinta de la figura 6.4(a) se representa con las cadenas de caracteres $\sqcup a \sqcup \sqcup$, b y bab (parte izquierda, cuadro bajo la cabeza, y parte derecha, respectivamente), mientras que la cinta de la figura 6.4(b) sería representada por ε, $\sqcup$ y $\sqcup\sqcup\sqcup b\, a \sqcup\, b$.

Finalmente, la configuración es un elemento de:

$$K \times \Gamma^* \times \Gamma \times (\Gamma^*(\Gamma - \{\sqcup\}) \cup \{\varepsilon\}$$

(Ver figura 6.5)

Como en los AF y los AP, en las MT vamos a indicar las configuraciones encerradas entre dobles corchetes, 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.

6.2.2. 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 $\delta(p, a) = (q, b)$, donde $b \in \Gamma$

Caso de movimiento a la izquierda, parte derecha no vacía:

$$[[p, wd, a, u]] \vdash [[q, w, d, au]]$$

ssi $\delta(p, a) = (q, L)$, donde $a \neq \sqcup$ o bien $u \neq \varepsilon$

Caso de movimiento a la izquierda, parte derecha vacía:

$$[[p, wd, \sqcup, \varepsilon]] \vdash [[q, w, d, \varepsilon]]$$

ssi $\delta(p, \sqcup) = (q, L)$

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:

$$[[p, w, a, \varepsilon]] \vdash [[q, wa, \sqcup, \varepsilon]]$$

ssi $\delta(p, a) = (q, R)$

Ejemplos:

$$\begin{array}{ll}
\text{Si } \delta(q_1, a) = (q_2, b), & [[q_1, bb\underline{a}]] \vdash [[q_2, bb\underline{b}]] \\
\text{Si } \delta(q_1, a) = (q_2, R), & [[q_1, b\underline{a}b]] \vdash [[q_2, ba\underline{b}]] \\
 & [[q_1, ba\underline{b}]] \vdash [[q_2, bab\underline{\sqcup}]] \\
\text{Si } \delta(q_1, a) = (q_2, L), & [[q_1, aaba\underline{b}]] \vdash [[q_2, aab\underline{a}b]] \\
 & [[q_1, a\underline{b}b]] \vdash [[q_2, \underline{a}bb]] \\
 & [[q_1, ab\sqcup\underline{\sqcup}]] \vdash [[q_2, ab\underline{\sqcup}]]
\end{array}$$

6.2.3. Configuración "colgada"

En el caso de que la cabeza lectora se encuentre en el cuadro de la cinta más a la izquierda, y se trate de hacer un movimiento a la izquierda, se produce un error llamado "configuración colgada", que tiene como consecuencia que la MT no pueda seguir funcionando, y desde luego no podrá ser aceptada la palabra de entrada.

Formalmente, si tenemos una configuración de la forma $[[p, \varepsilon, a, u]]$ y la transición es $\delta(p, a) = (q, L)$, no existe una configuración C tal que $[[p, \varepsilon, a, u]] \vdash C$.

En general vamos a evitar el uso intencional de las configuraciones colgadas, de modo que si no queremos que una palabra sea aceptada, se hará que la MT se cicle en vez de "colgarse".

6.2.4. 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): $K = \{s\}$, $\Sigma = \{a, \sqcup\}$, $\delta(s, a) = (s, R)$, $\delta(s, \sqcup) = (h, \sqcup)$. Ante la configuración $[[s, a, a, aa]]$ se presenta el cálculo siguiente:

$$[[s, a\underline{a}aa]] \vdash [[s, aa\underline{a}a]] \vdash [[s, aaa\underline{a}]] \vdash [[s, aaaa\underline{\sqcup}]] \vdash [[h, aaaa\underline{\sqcup}]]$$

Se puede llegar de una configuración C_i a C_j, para $i \leq j$ en cero o varios pasos; esto se indica en forma compacta utilizando la cerradura reflexiva y transitiva de la relación $\vdash$, denotada por $\vdash^*$, quedando $C_i \vdash^* C_j$.

6.2.5. Palabra aceptada

Con las definiciones dadas ahora estamos en condiciones de definir formalmente las nociones de palabra aceptada y lenguaje aceptado:

Definición.- Una palabra $w \in \Sigma^*$, es *aceptada* por una MT M si

$$[[s, \varepsilon, \sqcup, w]] \vdash^* [[h, \alpha, a, \beta]]$$

donde $a, \in \Gamma$, $\alpha, \beta \in \Gamma$. Como se ve, 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 da *halt* para toda entrada $w \in L$.

6.3. 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 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, caracterizado por los siguientes puntos:

- La palabra de salida no debe contener ningún caracter blanco ($\sqcup$).
- La palabra de salida comienza en el segundo caracter de la cinta, teniendo a su izquierda un blanco y a su derecha una infinidad de blancos.
- La cabeza estará posicionada en el primer blanco a la derecha de la palabra de salida.

Se puede apreciar que el formato para la palabra de salida es muy similar al de la palabra de entrada, salvo que en la primera, la cabeza está posicionada en el caracter a la derecha de la palabra.

Ejemplo.- Supongamos la función *reverse*, que invierte el orden en que aparecen las letras en la palabra de entrada; así, $reverse(aabb) = bbaa$. Si inicialmente el contenido de la cinta es de la forma $\underline{\sqcup}aabb\sqcup\ldots$, donde el caracter subrayado indica la posición de la cabeza, la cinta al final debe quedar como: $\sqcup bbaa\underline{\sqcup}\ldots$.

Es muy importante ceñirse estrictamente a este formato, y no caer en ninguno de los siguientes errores (frecuentes, desgraciadamente):

- Aparece algún espacio blanco dentro del resultado, como en la cinta $\sqcup bbaa\sqcup ab\underline{\sqcup}\ldots$.
- El resultado no está posicionado empezando en el segundo cuadro de la cinta, como en $\sqcup\sqcup\sqcup bbaa\underline{\sqcup}\ldots$.
- La cabeza no está ubicada exactamente en el cuadro a la derecha del resultado, como en la cinta $\sqcup bba\underline{a}\sqcup\ldots$.

- Aparece "basura" (caracteres no blancos) en la cinta, a la derecha o izquierda del resultado, como en la cinta $\sqcup bbaa\underline{\sqcup}\sqcup\sqcup b\sqcup\ldots$.

Para precisar estas nociones, utilizamos la noción formal de configuración : Una MT calcula un resultado $u \in \Sigma^*$ a partir de una entrada $w \in \Sigma^*$ si:

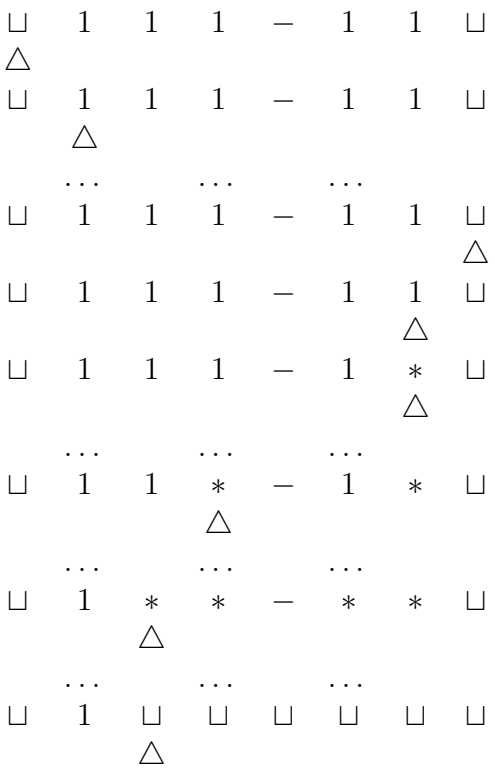
$$[[s, \varepsilon, \sqcup, w]] \vdash^* [[h, u, \sqcup, \varepsilon]]$$

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 $f : \Sigma^* \to \Sigma^*$ 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*.

Ejemplo.- Construir una máquina de Turing que reste dos números 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 solo 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*).



Dejamos como ejercicio hacer el diagrama de estados del control finito de esta MT (ver sección de ejercicios).

6.4. Problemas de decisión

Un caso particular de funciones es aquel en que el resultado sólo puede ser *sí* o *no*. Si representamos el *sí* con 1 y el *no* con 0, estamos considerando funciones $g : \Sigma^* \to \{1, 0\}$. 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, para una palabra de entrada como *aaba* la salida seria 1, y para *bab* sería 0.

La MT correspondiente debe generar los cálculos siguientes:

$$[[s, \varepsilon, \sqcup, w]] \vdash^* [[h, \sqcup 1, \sqcup, \varepsilon]]$$

si $|w|$ es par, y

$$[[s, \varepsilon, \sqcup, w]] \vdash^* [[h, \sqcup 0, \sqcup, \varepsilon]]$$

si $|w|$ es non.

Un diseño para la MT que decide si una entrada en el alfabeto $\Sigma = \{a, b\}$ es de longitud par aparece en la figura 6.6. La estrategia en este diseño es primero recorrer la cabeza al

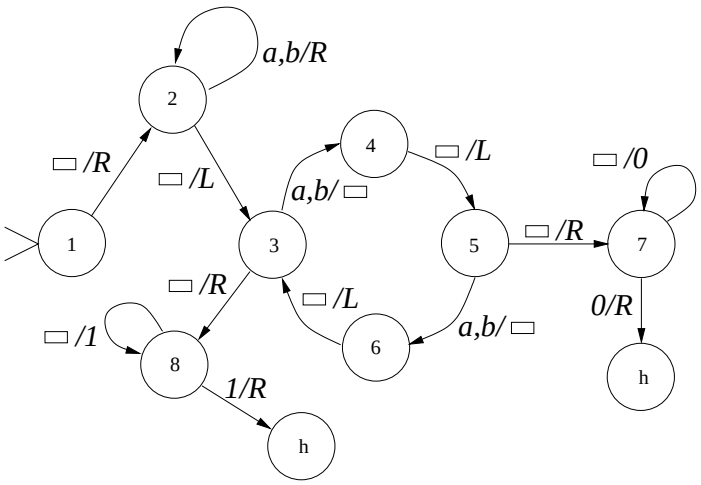


Figura 6.6: MT que decide si la entrada es de longitud par

extremo derecho, y luego ir borrando los caracteres de entrada, de derecha a izquierda, y "recordando" mediante los estados 3 y 5 si la cantidad de letras es, hasta el momento, par o impar, respectivamente. Al terminar de borrar la palabra de entrada, según que se haya terminado en el estado 3 o 5, se escribe 1 o 0 en la cinta, y se llega a halt.

Definición.- Decimos que un lenguaje L es *Turing-decidible* si hay alguna MT que entrega un resultado 1 si la entrada w está en L, y un resultado 0 en caso contrario.

Debe quedar claro que para que una MT entregue como resultado 1 o 0, es condición indispensable que la palabra de entrada haya sido aceptada. Esto tiene la consecuencia siguiente:

Proposición.- Un lenguaje es Turing-decidible solamente si es Turing-aceptable.

Si un lenguaje no es Turing-decidible se dice que es *indecidible*. Más adelante veremos lenguajes indecidibles.

6.4.1. Relación entre aceptar y decidir

Las siguientes propiedades que relacionan "Turing-decidible" con "Turing-aceptable" son útiles para comprender mejor ambas nociones:

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 1 por 0 y viceversa.

La prueba de 3 es más complicada. De hecho no probaremos que si L y L^c son Turing-aceptables entonces L es decidido por alguna MT, sino más bien que hay un *procedimiento mecánico* para decidir L. Se supone que, por la llamada *Tesis de Church*, que veremos luego, ambos enunciados son equivalentes. 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 1, mientras que si M^c se para, entrega una salida 0. Es evidente que para toda palabra w, M^* decidirá 1 o 0, por lo que el lenguaje es decidible.

6.5. Tesis de Church

Ha habido diversos intentos de encontrar otros modelos de máquinas u otros formalismos que sean más poderosos que las MT, en el mismo sentido que las MT son más poderosas que los AF y los AP. (Decimos que una tipo de máquina M_A es más poderoso que un tipo M_B cuando el conjunto de lenguajes aceptados por alguna máquina en M_B es un subconjunto propio de los aceptados por M_A). Por ejemplo, independientemente de Turing, Emil Post propuso aún otro modelo de máquina abstracta, basada en la idea de un diagrama de flujo [12].

También se han tratado de hacer "extensiones" a la MT, para hacerla más poderosa. Por ejemplo, se propusieron MT no deterministas. Sin embargo, todos los intentos han sido infructuosos al encontrarse que dichas extensiones son equivalentes en poder de cálculo a la MT original [10].

El mismo A. Turing propuso, en la llamada "Tesis de Turing", que todo aquello que puede ser calculado, podrá ser calculado en una MT, y que no podrá haber una máquina abstracta que calcule algo que la MT no pueda calcular [11]. Más aún, A. Church, a la sazón inventor del cálculo lambda –uno de los sistemas competidores de la MT–, propuso la conjetura de que en realidad no puede haber ningún modelo de cómputo más poderoso que los desarrollados hasta entonces, que incluían la MT, su cálculo lambda, así como otras máquinas abstractas, como la máquina de Post.

Hasta nuestros días la llamada "tesis de Church" no ha podido ser probada ni refutada. La tesis de Church, sin embargo, no se considera un teorema que pudiera ser eventualmente probado, sino simplemente una hipótesis de trabajo.

6.5.1. Comparación de las MT con otras máquinas

Podemos considerar comparaciones de la MT con:

1. Extensiones a la MT
   - *a*) MT con varias cintas, varias cabezas
   - *b*) MT con no determinismo
2. Otras máquinas de cinta
3. Otros paradigmas (máquinas de Post, Gramáticas)

De todas estas posibilidades, sólo consideraremos las máquinas de Post. Las comparaciones restantes pueden ser encontradas en la referencia [10].

Las pruebas que vamos a considerar 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.

6.6. Máquinas de Post

En esta sección presentaremos los elementos de la máquina propuesta por E. Post, de manera similar a como aparecen en [12].

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, etc.). La utilidad práctica de los diagramas de flujo es una de las razones para incluir el estudio de las máquinas de Post en este texto, aún cuando en muchos otros textos se prefiere comparar a las MT con los "sistemas de reescritura", con el "cálculo lambda" y otras alternativas.

En un diagrama de flujo se van 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. [5]

[5] Pensamos que el lector está habituado a los diagramas de flujo, por lo que no abundaremos en ejemplos y explicaciones.

Más específicamente, los diagramas de flujo de Post, llamados "Máquinas de Post" (MP), 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.

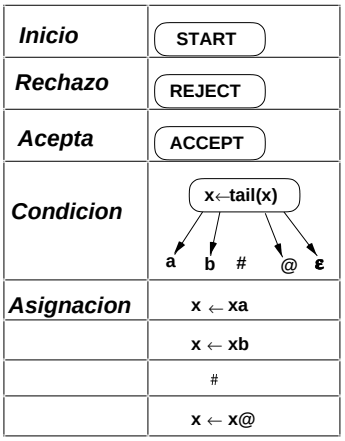


Figura 6.7: Acciones en MP

En la figura 6.7 presentamos un resumen de las acciones de la MP, las cuales son: [6]

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.

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 fue 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 coincida 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 $a \in \Sigma$.

[6] Hemos utilizado los nombres en inglés para las acciones de la MP, simplemente por compatibilidad con la gran mayoría de los textos disponibles.

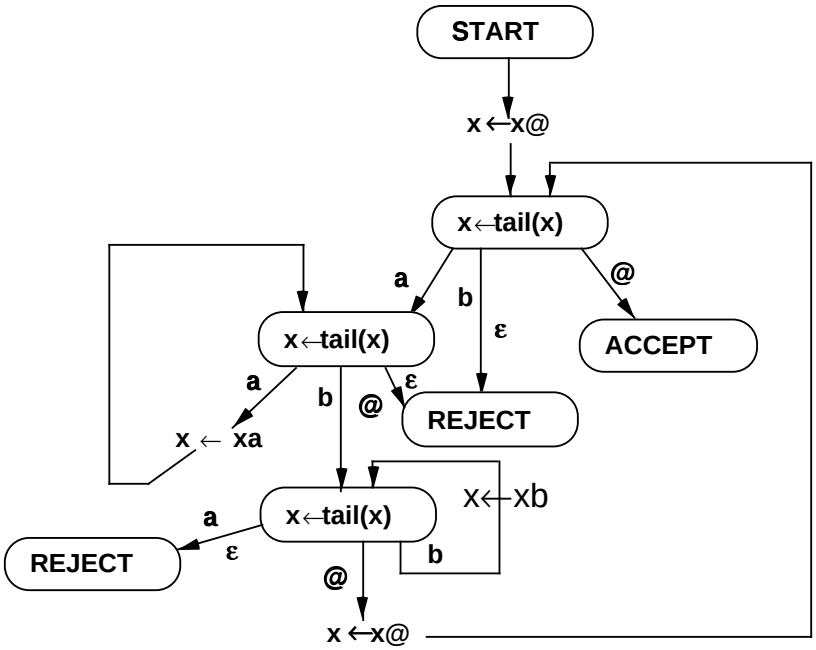


Figura 6.8: MP para $\{a^n b^n\}$

Ejemplo.- La MP de la figura 6.8 acepta el lenguaje $\{a^n b^n\}$. En efecto, siguiendo dicha figura, podemos ver que la variable x toma los siguientes valores al recorrer el diagrama de flujo:

Acción	**Valor de** x
START	$aabb$
$x \leftarrow x@$	$aabb@$
$x \leftarrow tail(x)$	$abb@$
$x \leftarrow tail(x)$	$bb@$
$x \leftarrow xa$	$bb@a$
$x \leftarrow tail(x)$	$b@a$
$x \leftarrow tail(x)$	$@a$
$x \leftarrow xb$	$@ab$
$x \leftarrow tail(x)$	ab
$x \leftarrow x@$	$ab@$
$x \leftarrow tail(x)$	$b@$
$x \leftarrow tail(x)$	$@$
$x \leftarrow tail(x)$	ε
$x \leftarrow x@$	$@$
$x \leftarrow tail(x)$	ε
ACCEPT	

Como se puede observar en este listado, las letras a, b y el caracter @ salen de la variable x por la izquierda, por la acción de $x \leftarrow tail(x)$, y luego entran por la derecha, como resultado

de una acción $x \leftarrow x\sigma$, donde σ es el caracter u se añade a la derecha. En la MP de este ejemplo, comparando las líneas 2 y 10 del listado podemos ver que en la 10 ya se ha eliminado una a y una b. Iterando en el diagrama es posible cancelar cada a con cada b hasta que se agoten las letras.

6.6.1. 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, [7] mientras que 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:

- $N = N_A \cup N_C \cup \{START, ACCEPT, REJECT\}$, 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:

$$\delta : N - \{ACCEPT, REJECT\} \times \Sigma \cup \{@, \varepsilon\} \rightarrow N - \{START\}$$

Como se ve, 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).

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 $n \in N$. [8] Además necesitamos considerar el contenido de la variable, que es una palabra formada por letras del alfabeto, pudiendo aparecer además el

[7] Este era el caso de las quíntuplas $(K, \Sigma, \delta, s, F)$ para los AF.

[8] Al decir que estamos en un nodo n, significa que aún no se ejecuta la acción del nodo n.

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 $m \in N_T,\ u = bw$
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*.

6.6.2. 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 es 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.

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.

Sea una configuración $[[q, w, u, v]]$ en MT. Entonces en la variable de la MP tendríamos: $uv@w$. Como se ve, 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 $\sigma \in \Gamma$. 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 6.9.

- Movimiento a la derecha: Al paso entre configuraciones de MT:

  $$[[p, abc\underline{d}efg]] \vdash [[q, abcd\underline{e}fg]]$$

  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 6.10.

- 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 (ver sección de ejercicios).

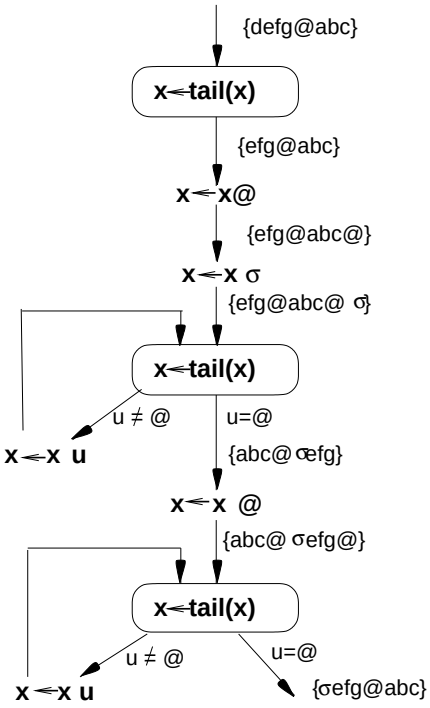


Figura 6.9: Escritura en MP

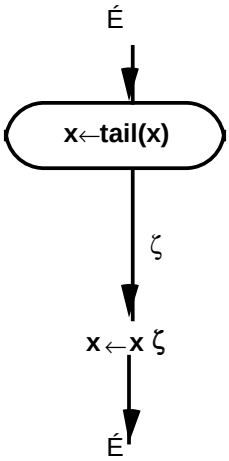


Figura 6.10: Movimiento a la derecha en MP

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 nuevamente los detalles de esta prueba al lector (ver sección de ejercicios).

6.7. 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 $w \in L(G_2)$.

6.7.1. El problema del paro de MT

Ahora vamos a considerar un problema irresoluble que históricamente tuvo mucha importancia porque fue 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. 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. [9]

Teorema.- No existe ninguna MT tal que, para cualquier palabra w y cualquier MT M, decida si $w \in L(M)$.

[9]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.

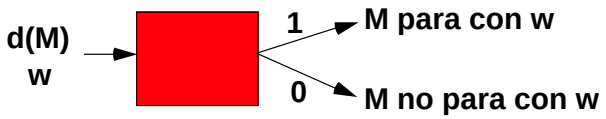


Figura 6.11: El problema del paro de una MT

En la figura 6.11 se muestra cómo debería funcionar la MT que resolvería el problema del paro.

Prueba [10] Por contradicción.- Sea A la MT de la figura 6.12(a). Entonces construimos otra MT B, como se representa en la figura 6.12(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 [11] de la siguiente manera: a la salida 1 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 ve que la salida 1 corresponde 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 0 –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 6.12(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.

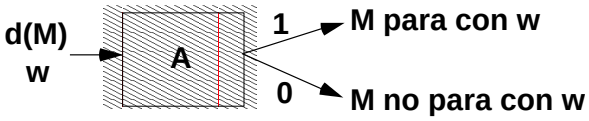
No haremos la prueba de estos resultados; remitimos al lector a las referencias [10], [7].

6.8. MT en la jerarquía de Chomsky

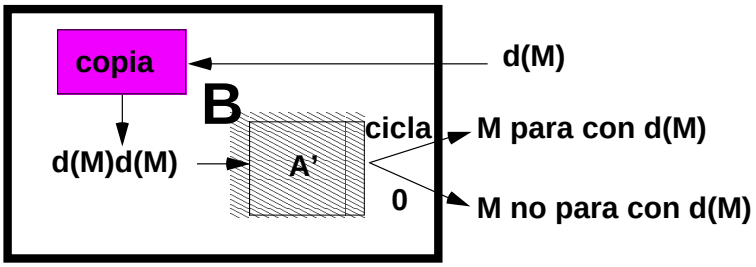
En conclusión, las MT no son capaces de aceptar todos los lenguajes posibles en 2^{Σ^*}. Sin embargo, este hecho puede ser establecido simplemente a partir de la enumerabilidad de las

[10] Esta prueba es debida a M. Minsky [14], aunque la primera prueba data de Turing [24].

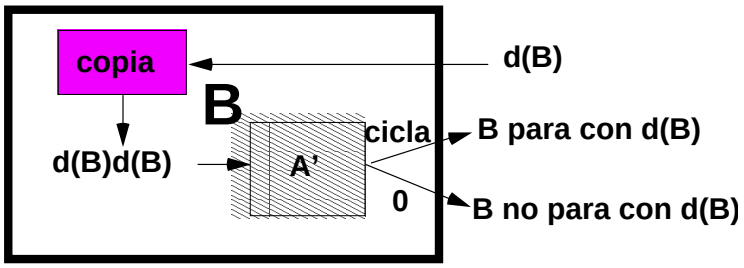
[11] Obsérvese que la segunda repetición de $d(M)$ es de hecho la palabra w que se supone que es sometida a M.



(a) Máquina A



(b) Máquina B



(c) Contradicción

Figura 6.12: Prueba del paro de MT

MT: puesto que las MT son cuádruplos (K, Σ, δ, s) -y por lo tanto elementos de un producto cartesiano-, al ser enumerable cada uno de los componentes necesariamente el cuádruplo es también enumerable. En efecto:

- Los conjuntos de los estados posibles son enumerables si estandarizamos los nombres de los estados a q_0, q_1, q_2, etc., lo cual evidentemente no altera ningún aspecto del funcionamiento de la MT (ver sección de ejercicios).
- Similarmente, un alfabeto estándar σ_0, σ_1, σ_2, etc., puede codificar cualquier alfabeto en particular. Así, también los alfabetos son enumerables.
- La función de transición es parte de otros productos cartesianos de estados y caracteres, por lo que es también enumerable.
- Los estados iniciales trivialmente son enumerables, siguiendo la estandarización del primer punto.

Ahora bien, al ser enumerables las MT, resulta que no puede mapearse un elemento de 2^{Σ^*} con una MT distinta, y por lo tanto hay lenguajes que no tienen una MT que los acepte.

Desde luego, el resultado anterior no ayuda a localizar exactamente qué lenguajes no son aceptados por ninguna MT; esto ya se había hecho para algunos lenguajes en la sección precedente.

Resulta útil entonces ubicar la parte de los lenguajes que sí pueden aceptar las MT con respecto a otras clases de lenguajes, siguiendo la estructura de clases de lenguajes llamada "jerarquía de Chomsky", que presentamos en las secciones 1.5 y 4.1

Recordando la jerarquía de Chomsky, que clasifica los lenguajes en categorías, y la forma en que se asocian distintos tipos de máquinas a dichas categorías de lenguajes, ahora estamos en condiciones de refinar la tabla que fue presentada en la sección 4.1, de la manera siguiente (indicamos entre paréntesis las secciones de este libro donde se presenta cada tema):

Tipo de autómata	**Lenguaje que procesa**	**Gramática que lo genera**
Autómatas finitos (2.2)	Lenguajes Regulares (3)	Gramáticas regulares (3.5)
Autómatas de Pila (5)	Lenguajes Libres de Contexto (4)	Gram. libres de contexto (4)
Autóm. linealmente acotados	Leng. Sensitivos al Contexto (4.7)	Gram. sensitivas al contexto (4.7)
Máq. de Turing decidiendo (6.4)	Lenguajes Recursivos	
Máq. de Turing aceptando (6.2.5)	Leng.Recursiv. Enumerables	Gram. no restringidas (4.1)

En esta tabla hemos diferenciado la clase de lenguajes que pueden ser *decididos* por una MT, que son llamados "recursivos", de los lenguajes que pueden ser *aceptados* por una MT, que son los "recursivamente enumerables", aunque no hemos definido ninguno de ellos más que por su relación con las MT. [12]

Asimismo hemos mencionado, por completez, la clase de los lenguajes Sensitivos al Contexto, que fueron citados en la sección 4.7, aunque no hemos estudiado los autómatas "linealmente acotados" en este libro; pueden estudiarse en las referencias [21] o [7].

De acuerdo con la presentación de la jerarquía de Chomsky que hicimos al estudiar las gramáticas en la sección 4.1, las MT son equivalentes en poder de cálculo a las gramáticas no restringidas. La prueba de esto puede ser consultada en diversas referencias [10], [7].

[12] "Recursivamente enumerables" es solamente otro nombre para "Turing aceptable", usado en textos como [7].

Así, de acuerdo con la Tesis de Church, los lenguajes recursivamente enumerables son el más extenso conjunto de lenguajes que pueden ser algorítmicamente analizados.

6.9. Ejercicios

1. Diseñe un diagrama de estados para la MT del ejemplo de la sección 6.2.5, esto es para aceptar el lenguaje $a^n b^n c^n$. Obtenga también la representación formal de dicha MT.

2. Diseñe un diagrama de máquina de Turing para calcular la función $|log_2 n|$, usando las máquinas básicas vistas. Describa las acciones efectuadas sobre la cinta.

3. Complete el diseño de la MT para el ejemplo de la sección 6.3, esto es para calcular restas de números en unario. Exprese esta MT usando la representación formal.

4. Diseñar una MT que decida si la entrada es de longitud par, para palabras en $\{a, b\}^*$.

5. 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 $m < n$.

6. 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 $\{\varepsilon\}$

7. Representar formalmente la MP de la figura 6.8.

8. Probar la enumerabilidad de los conjuntos de estados con nombres uniformizados q_0, q_1, q_2, etc. Ayuda: Considere una representación binaria de cada conjunto de estados, tomando 1 si el estado en cuestión está presente, y 0 si no está.

9. 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").

10. Supongamos unas "MTCE" que son como las MT, pero en vez de tener una cinta infinita a la derecha, tienen una "cinta estirable", que inicialmente contiene sólo cuadros en la cinta para la palabra de entrada y para un blanco a cada lado de dicha palabra, y que cuando se se mueve a la derecha fuera de la cinta, automáticamente es creado un nuevo cuadrito, según se va requiriendo. Formalizar las MTCE, en particular la definición de palabra aceptada.

11. 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. (Este problema completa el enunciado del Teorema de Post).

12. 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 6.13. Formalice los A2P en la forma

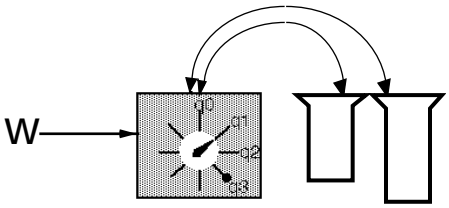


Figura 6.13: Automáta de dos pilas (A2P)

más similar posible a los AP vistos en clase. Defina formalmente las nociones de:

a) Configuración.

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. Ayuda: 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.

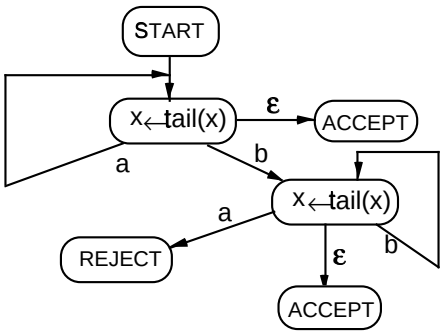


Figura 6.14: Máquina de Post

13. 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 6.14.

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 corresponde MP2? ¿Porqué?

14. 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).

15. 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 solo 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.

16. 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 solamente 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.

17. 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?

18. 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:

$$MT_1 = (\{f, g, j, k, m\}, \{a, b\}, \{a, b, \sqcup\}, \delta_1, f)$$

f	$\sqcup$	g	L
g	$\sqcup$	h	$\sqcup$
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	$\sqcup$	h	$\sqcup$
m	a	m	L
m	b	k	L

$$MT_2 = (\{o, q, n, p\}, \{a, b\}, \{a, b, \sqcup\}, \delta_2, q)$$

q	$\sqcup$	o	L
o	$\sqcup$	h	$\sqcup$
o	a	n	L
o	b	p	L
n	a	n	L
n	b	o	L
p	a	p	L
p	b	p	L

19. 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.

20. Realizar el diagrama de MP que simula el movimiento de la cabeza a la izquierda en una MT (esto es parte de la prueba de equivalencia de MP y MT).

21. Completar la prueba de equivalencia de las MP y las MT, detallando la simulación de la MP en una MT, siguiendo las ideas esbozadas en la página 191.

Bibliografía

[1] A. Aho, J. Ullman.- *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1978.

[2] J. Barwise, J. Etchemendy.- *Turing's World 3.0 -An Introduction to Computability Theory*, CSLI Publications, Stanford, CA, 1993.

[3] G. Brookshear.- *Teoría de la Computación*, Addison Wesley Iberoamericana, 1993.

[4] N. Chomsky.- *Aspects of the Theory of Syntax*, Cambridge, MIT Press, 1965.

[5] V. Drobot.- *Formal languages and automata theory*, Computer Science Press, 1989.

[6] J. Hopcroft, J. Ullman.- *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, 1979.

[7] J. Hopcroft, R. Motwani, J. Ullman.- *Introduction to Automata Theory, Languages and Computation, Second Edition*, Addison Wesley, 2001.

[8] D. Kelley.- *Teoría de Autómatas y Lenguajes Formales*, Prentice Hall Hispanoamericana, 1995.

[9] J. Levine, T. Mason, D. Brown.- *Lex & Yacc*, O'Reilly & Associates, 1992.

[10] H.R. Lewis, Ch.H. Papadimitriou.- *Elements of the Theory of Computation*, Prentice Hall, 1981.

[11] P. Linz.- *An Introduction to Formal Languages and Automata*, D. C. Heath and Company, 1990.

[12] Z. Manna.- *Mathematical Theory of Computation*, McGraw Hill, 1974.

[13] G. Mealy.- *A method for synthesizing sequential circuits*, BSTJ n.34, 1955, pp1045-1079.

[14] M. Minsky.- *Computation: Finite and Infinite Machines*, Prentice Hall, 1967.

[15] E. Moore.- *Gedanken-experiments on sequential machines*, en "Automata Studies" (C. Shannon, J. McCarthy eds.), Princeton Univ. Press, 1956.

[16] J.L. Peterson.- *Petri Net Theory and the Modeling of Systems*, Prentice Hall, 1981.

[17] C. Petri.- *Kommunication mit Automaten*, Universidad de Bonn, 1962.

[18] J.I.. Pozo.- *Teorías cognitivas del aprendizaje*, 2da Ed., Morata, Madrid, España, 1993.

[19] S. Sahni.- *Concepts in Discrete Mathematics*, Camelot Publishing Co. 1985.

[20] M. Silva.- *Las Redes de Petri en la Automática y la Informática*, Editorial AC, 1985.

[21] M. Sipser.- *Introduction to the Theory of Computation*, PWS Pub. Co., 1997.

[22] G. Springer, D.P. Friedman.- *Scheme and the Art of Programming*, The MIT Press, 1989.

[23] T. Sudkamp.- *LANGUAGES AND MACHINES.- An Introduction to the Theory of Computer Science*, Addison Wesley, 1994.

[24] A. Turing.- *On computable numbers with an application to the Entscheidungs-problem*, Proc. London Math. Soc., v.2, n.42, pp230-265.

Índice alfabético