Teoría de autómatas y lenguajes formales

Dean Kelley
Departamento de Matemáticas y Ciencias de la Computación
Gustavus Adolphus College

Traducción:
Mª Luisa Díez Platas
Facultad de Informática
Universidad Pontificia de Salamanca en Madrid

Revisión técnica:
Luis Joyanes Aguilar
Facultad de Informática
Universidad Pontificia de Salamanca en Madrid

PRENTICE HALL
Madrid • México • Santafé de Bogotá • Buenos Aires • Caracas • Lima • Mon
San Juan • San José • Santiago • São Paulo • White Plains

DEAN KELLEY
Teoría de autómatas y lenguajes formales

C/ Núñez de Balboa, 120
28006 Madrid

ISBN 0-13-518705-2
Depósito Legal: M-21607-2001
Última reimpresión, 2001

Traducido de:
AUTOMATA AND FORMAL LANGUAGES: AN INTRODUCTION

PRENTICE HALL
© 1995
ISBN: 0-13-497777-7

Editor de la edición en español: Juan Stumpf
Diseño de cubierta: DIGRAF
Composición: AULA DOCUMENTAL DE INVESTIGACIÓN
Impreso por ENCO Artes Gráficas, S.L.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro está impreso con papel y tintas ecológicos

A mis Padres
... y al tío Fudd

Contenido

Prólogo

Este libro es el resultado de las anotaciones tomadas en el curso de introducción a la teoría de la computación, impartido en la universidad *Gustavus Adolfus*. Dicho curso abarca los temas de lenguajes formales y autómatas, máquinas de Turing y computabilidad a través de la resolubilidad. El curso va dirigido a estudiantes de segundo año de Ciencias de la Computación y hasta ahora este curso tenía como principal prerrequisito el haber seguido con anterioridad un curso sobre lectura y tratamiento de las demostraciones matemáticas.

El libro está dirigido a estudiantes con los conocimientos matemáticos mínimos. El Capítulo 0 trata de los preliminares matemáticos necesarios para poder abordar la lectura del libro en su totalidad.

El nivel de conocimientos matemáticos en los capítulos siguientes es inicialmente bajo, pero se eleva a medida que los temas lo requieren y la capacidad del estudiante se incrementa. Trataré de evitar las demostraciones matemáticas rigurosas en la medida de lo posible, en especial en los primeros capítulos. Por tanto, se evitará dar muchos detalles de las demostraciones que sean tediosas. Por otro lado, trataré de presentar teoremas y definiciones de la forma más precisa posible. La mayoría de mis razonamientos tienen la intención de motivar mas que la de ser matemáticamente completos o elegantes.

Creo, además, que los ejercicios contribuyen, en gran medida, al buen aprendizaje del proceso. Al final de cada sección, los ejercicios planteados pretenden ilustrar, revisar y ampliar los conceptos vistos en las mismas. Hay ejerci-

cios desde bastante fáciles a muy difíciles. La mayoría de los ejercicios más fáciles pretenden reforzar las ideas vistas en la sección, mientras que los más difíciles ilustran y amplían dichas ideas.

Los Capítulos del 1 al 7 terminan con unas colecciones de problemas. Dichos problemas pretenden animar al lector a investigar sobre los temas tratados. Generalmente (aunque no siempre), el material tratado en las colecciones de problemas representan lo que yo considero necesario para estar interesado en realizar incursiones a través de la línea de desarrollo principal del texto.

El texto consta de ocho capítulos. A continuación se realiza una breve descripción de cada uno de ellos.

El Capítulo 0 cubre los preliminares matemáticos y lógicos. Consiste en una rápida revisión de la lógica y la teoría de conjuntos, siendo un capítulo bastante completo. Además, en este capítulo se repasan todos los conocimientos matemáticos necesarios para entender el resto del texto.

En el Capítulo 1 se presentan las definiciones básicas y la notación usada para alfabetos, cadenas y lenguajes. Se definen y estudian las operaciones elementales sobre cadenas y lenguajes.

El Capítulo 2 trata de los lenguajes y las expresiones regulares. Se definen los autómatas finitos y se establece la relación de los mismos con los lenguajes regulares. Se introduce el no determinismo. Además, se estudian las propiedades fundamentales de los lenguajes regulares (lema del bombeo, algoritmos de decisión, etc.).

El Capítulo 3 introduce los conceptos sobre gramáticas desarrollando, además, las propiedades de las gramáticas independientes del contexto y los autómatas de pila. Se presentan varias simplificaciones y formas normales de gramáticas.

El Capítulo 4 es el primero de los cuatro capítulos cuyo tema central son las máquinas de Turing. Este capítulo contiene definiciones básicas, las distintas versiones de máquinas de Turing e introduce la idea de funciones Turing-computables y lenguajes reconocidos por las máquinas de Turing.

El Capítulo 5 estudia las relaciones entre las máquinas de Turing y los lenguajes formales. Además, establece la jerarquía de Chomsky.

En el Capítulo 6 se habla de la resolubilidad. Comienza con el problema de parada de las máquinas de Turing, después trata el problema de la irresolubilidad del problema de la correspondencia de Post y presenta algunos problemas irresolubles para los lenguajes y gramáticas independientes del contexto. El capítulo termina con el estudio de las funciones computables totales.

El Capítulo 7 es una introducción a la complejidad computacional del reconocimiento del lenguaje. Se estudia en función de los recursos de espacio y tiempo (de la máquinas de Turing).

Aunque la mayoría del material presentado se corresponde con el contenido usual de un curso de introducción a la teoría, la novedad de este texto quizás radique en el nivel con el cual se enfoca. He tratado de transmitir todos estos conceptos a estudiantes que no sean avezados matemáticos, de forma que puedan comprenderlos al mismo tiempo que desarrollan su capacidad matemática.

Entiendo que éste es el material suficiente para cubrir un curso de un semestre de cuatro días a la semana. Generalmente trato los Capítulos del 1 al 5 en su totalidad, con la rapidez con que los estudiantes son capaces de asimilarlos. Este material es el corazón de cualquier curso de teoría de lenguajes formales y no me importa tomarme el tiempo necesario para que los estudiantes puedan digerirlo. Dependiendo de la audiencia, imparto algunas clases del Capítulo 0 o simplemente asigno trabajos a cerca del mismo. La mayoría del material presente en el Capítulo 0, proviene de un curso corto (2 créditos) de demostraciones matemáticas que una vez se impartió en la universidad. Siempre trato de desarrollar el Capítulo 6 en su totalidad, aunque depende del tiempo que quede del semestre. Al principio me sorprendió que, cuando el tiempo apremia, la resolubilidad puede ser presentada bastante bien por medio de conferencias cortas.

Me gustaría agradecer a mi amigo Ding-Zhu Du de la Universidad de Minnesota, Minneapolis, por sugerirme el Problema 1.7, el cual trata de la desigualdad de McMillan. El desarrollo anterior al Lema 2.8.3, usado en el lema de Arden, fue sugerido por una de las primeras personas que revisaron este libro. Me gustaría agradecerle, a él o a ella, por llamar mi atención sobre el mismo (debido a Brzozowski). Me gustaría también expresar mi aprecio a mis amigos T. J. Morrison y D. J. Malmanger por su estímulo y apoyo moral a lo largo de este proyecto aparentemente infinito. Finalmente, me gustaría dar las gracias a las siguientes personas por sus comentarios cuando revisaron el manuscrito: Moon Jung Chung (Michigan State University), Ronald K. Friesen (Texas A&M University), Micha Hofri (University of Houston), Robert Kline (West Chester University) y S. A. Kovatch (General Electric).

Dean Kelley

Teoría de autómatas y lenguajes formales

0

Preliminares matemáticos

0.1 LÓGICA ELEMENTAL

Para el estudio de la teoría de la computación se necesitan tres herramientas matemáticas básicas. Una de ellas es la notación teórica establecida, otra el dominio de los conceptos de funciones y relaciones, y la tercera son unos buenos conocimientos de inducción matemática. La capacidad para usar la notación teórica establecida depende, fundamentalmente, del conocimiento de las definiciones básicas de símbolos y sus significados. Conocer las otras dos herramientas depende de la capacidad para entender razonamientos lógicos. Por tanto, comenzaremos con una presentación de las ideas fundamentales de la lógica para pasar a establecer los mecanismos matemáticos requeridos.

En lógica, una *proposición* o *sentencia* es una frase de la cual se puede determinar si es verdadera o falsa. Las frases "$2 + 1$ es 5", "$3 > \sqrt{8}$" y "17 es un número primo" son proposiciones, mientras que "ven a nuestra fiesta", "¿qué hora es?" y "esta proposición es falsa" no lo son. Si P y Q son proposiciones, se dice que P es *equivalente* a Q si para todos los casos tienen el mismo valor de verdad. Por eso las frases "$3 < 5$" y "π es irracional" son equivalentes, como lo son las frases "$\frac{1}{2}$ es un entero" y "$4 < 3$", puesto que sus valores de verdad son los mismos.

Si P es una proposición, su *negación* se denota por $\neg P$. Si P es verdadera, $\neg P$ es falsa, y si P es falsa, $\neg P$ es verdadera. $\neg P$ se lee "no P". Por ejemplo, si P es la proposición "$3 < 5$", $\neg P$ es "$3 \geq 5$". Dado que el valor de verdad de $\neg P$

depende del valor de verdad de P, podemos usar una tabla, llamada *tabla de verdad*, para indicar la dependencia:

P	$\neg P$
V	F
F	V

La tabla de verdad presenta los valores de verdad de $\neg P$ correspondientes a los valores de verdad de P.

La *conjunción* de las proposiciones P y Q se denota por $P \wedge Q$ y se lee "P y Q". La proposición compuesta $P \wedge Q$ es verdadera sólo cuando P y Q sean verdaderas simultáneamente. Por eso podemos obtener la tabla de verdad siguiente:

P	Q	$P \wedge Q$
V	V	V
V	F	F
F	V	F
F	F	F

Es importante hacer notar que para una proposición compuesta la tabla de verdad debe considerar todas las posibles combinaciones de los valores de verdad correspondientes a los componentes de la misma.

Considérense las conjunciones siguientes

1. $3 < \sqrt{17}$ y $25 = 5^2$.
2. $3 < \sqrt{17}$ y $26 = 5^2$.
3. $3 \geq \sqrt{17}$ y $25 = 5^2$.
4. $3 \geq \sqrt{17}$ y $26 = 5^2$.

De estas cuatro proposiciones compuestas, sólo la primera es verdadera. En todas las demás, al menos uno de los componentes es falso, lo que hace que la conjunción sea falsa.

La *disyunción* de las proposiciones P y Q se denota por $P \vee Q$. Es verdadera cuando al menos una de las dos es verdadera. Otra forma de decir esto es que $P \vee Q$ es falsa solamente cuando P y Q son falsas a la vez. $P \vee Q$ se lee "P o Q".

La proposición $P \rightarrow Q$ se llama proposición *condicional* y tiene la siguiente tabla de verdad

P	Q	$P \rightarrow Q$
V	V	V
V	F	F
F	V	V
F	F	V

La condicional se lee "si P entonces Q". Para entender los valores de verdad de $P \rightarrow Q$, considérese la proposición "Si el sol brilla, entonces Carlos juega al béisbol". Para determinar cuando es verdadera esta proposición, nos preguntamos si la persona que la ha hecho está diciendo la verdad. Tenemos cuatro casos que corresponden a las cuatro líneas de la tabla de verdad precedente.

En el primer caso (el sol brilla y Carlos juega al béisbol), se ha dicho la verdad. En el segundo caso (el sol brilla y Carlos no juega al béisbol), no se ha dicho la verdad. En los dos últimos casos (el sol no brilla y Carlos juega al béisbol; el sol no brilla y Carlos no juega al béisbol), no deberíamos decir que la persona que hizo la proposición es un mentiroso puesto que el sol no brilla y él, o ella, sólo dijeron lo que ocurriría si el sol brillara.

En la condicional $P \rightarrow Q$, la proposición P se llama *hipótesis*, *condición* o *antecedente*, mientras que Q se llama *conclusión* o *consecuente*.

La *recíproca* de la condicional $P \rightarrow Q$ es la proposición $Q \rightarrow P$.

La *contrapuesta* de $P \rightarrow Q$ es $(\neg Q) \rightarrow (\neg P)$. Advierta que $P \rightarrow Q$ y su contraposición son equivalentes puesto que tienen los mismos valores de verdad para todos los casos, como muestra la siguiente tabla de verdad:

P	Q	$P \rightarrow Q$	$\neg Q$	$\neg P$	$\neg Q \rightarrow \neg P$
V	V	V	F	F	V
V	F	F	V	F	F
F	V	V	F	V	V
F	F	V	V	V	V

Consideremos la proposición $P \rightarrow Q \wedge Q \rightarrow P$. Es fácil ver que la proposición es verdadera sólo cuando P y Q tienen los mismos valores de verdad. Esta

proposición en forma abreviada es $P \leftrightarrow Q$, la cual se lee "P si y sólo si Q". Se llama proposición *bicondicional.*

Las proposiciones $\neg (P \wedge Q)$ y $(\neg P) \vee (\neg Q)$ son equivalentes (ésta es una de las leyes de De Morgan; véase Ejercicio 0.1.2). Considérese la proposición

$$\neg (P \wedge Q) \leftrightarrow (\neg P) \vee (\neg Q)$$

A causa de la equivalencia de proposiciones, obtenemos que ambos lados de la bicondicional tienen el mismo valor de verdad en todos los casos. Por tanto la bicondicional es verdad en todos los casos. Esto sugiere el siguiente teorema:

Teorema 0.1.1. Sea P y Q proposiciones para las cuales $P \leftrightarrow Q$ es siempre verdadera. Entonces P y Q son equivalentes. Por otro lado, si P y Q son equivalentes, entonces la bicondicional $P \leftrightarrow Q$ es siempre verdadera.

Un proposición es una *tautología* si es siempre verdadera. Fíjese que si P y Q son equivalentes entonces, según el teorema previo, $P \leftrightarrow Q$ es una tautología. Por tanto, la equivalencia puede ser definida como: P y Q son equivalentes si $P \leftrightarrow Q$ es una tautología.

Cuando la proposición condicional $P \rightarrow Q$ es una tautología, se escribe $P \Rightarrow Q$. De forma similar podemos escribir $P \Leftrightarrow Q$ si la bicondicional $P \leftrightarrow Q$ es una tautología. Fíjese, que substancialmente, esto no significa más que la condicional (o bicondicional) es una proposición verdadera. La verdad de $P \rightarrow Q$ depende de los valores de verdad de P y Q. Por otro lado, $\neg (A \wedge B) \rightarrow (\neg A) \vee (\neg B)$ es una proposición verdadera sea cuales sean los valores de verdad de sus componentes A y B. Esto puede ser representado por $\neg (A \wedge B) \Rightarrow (\neg A) \vee (\neg B)$.

Una *contradicción* es una proposición que siempre es falsa. Por tanto, la negación de una tautología es una contradicción.

Una *frase abierta* o *función proposicional* es una proposición que contiene una variable. Por ejemplo, la frase "$x^2 + 2x + 16 = 0$" contiene la variable x, al igual que la frase "x fue el primer presidente de los Estados Unidos". La colección de objetos que pueden ser sustituidos por una variable en una frase abierta se llama *conjunto de significados* de esa variable. Llamaremos *conjunto de verdad* de la frase abierta, al conjunto de objetos pertenecientes al conjunto de significados para los cuales la frase abierta se convierte en una proposición verdadera al sustituir la variable por ellos. Si se considera que el conjunto de significados para la frase abierta $x^2 + 2x + 16 = 0$ es el de los números reales, entonces el conjunto de verdad es vacío. Si el conjunto de significados incluye además $-1 \pm i\sqrt{15}$ entonces el conjunto de verdad tiene algún elemento.

Si P es una frase abierta que contiene la variable x, se escribe $P(x)$. Generalmente, cuando se muestra una frase abierta, el conjunto de significados para la(s) variable(s) que contiene es explícitamente declarado o se deduce fácilmente del contexto. Ciertos operadores indican la forma de seleccionar elementos del conjunto de significados. Esos operadores son los *cuantificadores* universal y existencial.

Una frase de la forma "para todo x del conjunto de significados $P(x)$ es verdadera" se dice que es una frase *universalmente cuantificada*. Esto indica que el conjunto de verdad de $P(x)$ esta compuesto por todos los objetos pertenecientes al conjunto de significados de x. Esto, en forma abreviada, se escribe $\forall x\, P(x)$, lo cual se lee "para todo x, $P(x)$". Fíjese que $\forall x\, P(x)$ ya no es una frase abierta, puesto que su verdad o falsedad puede ser determinada. Por ejemplo, si $P(x)$ es la frase abierta "$x + 1 > x$" y el conjunto de significados es la colección de todos los números reales, entonces $\forall x\, P(x)$ es una proposición verdadera.

Una frase de la forma "existe un x en el conjunto de significados para el cual $P(x)$ es verdadera" se dice que está *cuantificada existencialmente*. Esto indica que algún elemento del conjunto de significados es un valor que, al sustituir a x, hace que $P(x)$ sea verdadera. Lo cual quiere decir que algún elemento del conjunto de significados está también en el conjunto de verdad de $P(x)$. Esto en forma abreviada, se escribe $\exists x\, P(x)$ y se lee "existe un x tal que $P(x)$", o "para algún x, $P(x)$". Tenga en cuenta que $\exists x\, P(x)$ ya no es una frase abierta.

Teorema 0.1.2. $\neg(\forall x\, P(x))$ es equivalente a $\exists x \neg P(x)$.

Demostración. Supongamos que $\neg \forall x\, P(x)$ es verdadera. Entonces $\forall x\, P(x)$ es falsa, y por tanto, el conjunto de verdad de $P(x)$ no es todo el conjunto de significados de x. Entonces, el conjunto de verdad de $\neg P(x)$ contiene algún elemento. Por tanto la proposición $\exists x \neg P(x)$ es verdadera.

Ahora supongamos que $\neg \forall x\, P(x)$ es falsa. Entonces $\forall x\, P(x)$ es verdadera, así que el conjunto de verdad de $P(x)$ es todo el conjunto de significados de x. Por lo tanto, el conjunto de verdad de $\neg P(x)$ es vacío, lo que implica que $\exists x \neg P(x)$ es falsa.

Con esto hemos demostrado que las dos proposiciones tienen exactamente los mismos valores de verdad y, por tanto, son equivalentes. □

Si $P(x)$ es una frase abierta, entonces un *contraejemplo* para $\forall x\, P(x)$ es un elemento, t, del conjunto de significados de forma que $P(t)$ sea falsa. Se puede demostrar una proposición de la forma "Si para cada $x\, P(x)$, entonces $Q(x)$" probando que $\forall x\,(P(x) \rightarrow Q(x))$ es verdadera. Esto puede ser refutado si se proporciona un contraejemplo. Por ejemplo, la proposición "si n es primo enton-

ces $2^n - 1$ es primo" podría ser refutada si se encuentra un contraejemplo tal como $n = 11$.

Ejercicios de la Sección 0.1

0.1.1. Obtener la tabla de verdad para $P \vee Q$.

0.1.2. Probar que $\neg (P \wedge Q)$ es equivalente a $(\neg P) \vee (\neg Q)$. Probar que $\neg (P \vee Q)$ es equivalente a $(\neg P) \wedge (\neg Q)$. Estas dos equivalencias se conocen como las *leyes de De Morgan*.

0.1.3. Probar que $P \wedge (Q \vee R)$ es equivalente a $(P \wedge Q) \vee (P \wedge R)$ y que $P \vee (Q \wedge R)$ es equivalente a $(P \vee Q) \wedge (P \vee R)$.

0.1.4. Probar que P y $\neg (\neg P)$ son equivalentes.

0.1.5. Simplificar $\neg ((\neg P) \vee (\neg Q))$.

0.1.6. Simplificar $\neg ((\neg Q) \wedge (\neg P))$.

0.1.7. ¿Son equivalentes $P \rightarrow Q$ y $Q \rightarrow P$?

0.1.8. ¿Cuáles de las siguientes proposiciones son tautologías?

(a) $P \leftrightarrow \neg (\neg P)$.

(b) $\neg (P \vee Q) \leftrightarrow \neg P \wedge \neg Q$.

(c) $P \vee \neg P$.

(d) $(P \vee \neg P) \wedge (Q \vee \neg Q)$.

(e) $P \wedge \neg P$.

(f) $(P \wedge \neg P) \rightarrow Q$.

(g) $(P \leftrightarrow Q) \rightarrow (P \rightarrow Q)$.

(h) $(P \rightarrow Q) \leftrightarrow \neg P \vee Q$.

0.1.9. Probar que las siguientes proposiciones son contradicciones:

(a) $(P \rightarrow Q) \wedge (P \wedge \neg Q)$.

(b) $((P \vee Q) \wedge \neg P) \wedge (\neg Q)$.

(c) $(P \wedge Q) \wedge (\neg P)$.

0.1.10. Mostrar que $\neg \exists x\, P(x)$ es equivalente a $\forall x \neg P(x)$.

0.2 DEFINICIONES BÁSICAS

Ahora veremos las nociones básicas sobre teoría de conjuntos.

Un *conjunto* es una colección de objetos llamados *elementos* del conjunto. Si A es un conjunto y a es un elemento de A, se usa la notación $a \in A$ (se lee "a es un elemento de A"). Se usa la notación $b \notin A$ cuando es necesario indicar que b *no* es un elemento de A.

Si sabemos que A contiene exactamente los elementos $a_1, a_2, \ldots, a_n$, lo indicamos escribiendo $A = \{a_1, a_2, \ldots, a_n\}$. Por ejemplo, el conjunto de los números naturales menores que 6 puede ser escrito $A = \{0, 1, 2, 3, 4, 5\}$. Esta notación

puede ser extendida a los conjuntos para los cuales no es posible listar todos sus elementos, tales como $\mathbb{N} = \{0, 1, 2, \ldots\}$ o $\mathbb{N}^+ = \{1, 2, 3, \ldots\}$. Un conjunto sólo se caracteriza por sus elementos y no por el orden en el cual se listan. Por eso $\{1, 2, 3\}$ y $\{2, 1, 3\}$ denotan el mismo conjunto.

Los conjuntos A y B son *iguales* si contienen exactamente los mismos elementos. Por tanto, si $A = \{1, 2, 3\}$ y $B = \{2, 1, 3\}$, se puede escribir que $A = B$. Fíjese que $\{a\}$ y a no son lo mismo. Tenemos que $a \in A$, pero $a \neq \{a\}$. También el conjunto $\{\{a, b\}\}$ tiene un único elemento que es el conjunto $\{a, b\}$. Por otro lado, $\{a, b\}$ tiene dos elementos, a y b. Por consiguiente, $\{\{a, b\}\} \neq \{a, b\}$.

Si A y B son conjuntos y todos los elementos de A son también elementos de B, se escribe $A \subseteq B$ y se dice que A es un *subconjunto* de B. Por ejemplo, si $A = \{1, 2, 3\}$ y $B = \{0, 1, 2, 3, 4, 5\}$, se tiene $A \subseteq B$. Por otro lado, B no es un subconjunto de A, porque los elementos 0, 4 y 5 de B no lo son de A.

Obsérvese que si $A \subseteq B$ y $B \subseteq A$ simultáneamente, entonces todos los elementos de A están en B y todos los elementos de B están en A. Por lo tanto, si $A \subseteq B$ y $B \subseteq A$, tenemos que $A = B$.

Teorema 0.2.1. Si $A \subseteq B$ y $B \subseteq C$, entonces $A \subseteq C$.

Demostración. Sea $x \in A$. Entonces, si $A \subseteq B$ se obtiene que $x \in B$. Además, puesto que $B \subseteq C$ y $x \in B$, tenemos que $x \in C$. Por tanto, dado que x era un elemento arbitrario de A, resulta que $A \subseteq C$. □

Para completar las definiciones, es conveniente considerar un conjunto especial $\emptyset$, llamado conjunto *vacío* o *nulo*, el cual no tiene elementos. El conjunto vacío es un subconjunto de todos los conjuntos; por lo cual se puede escribir $\emptyset \subseteq A$, para todo conjunto A.

Algunas veces es conveniente describir el contenido de un conjunto en términos de una propiedad que sea característica de todos los elementos del conjunto. Sea $P(x)$ una proposición sobre x. La notación $\{x \mid P(x)\}$, que se interpreta como "el conjunto de todos los x tales que $P(x)$", denota el conjunto de todos los x para los cuales $P(x)$ es una proposición verdadera. Por ejemplo, $\mathbb{Z}^+ = \{x \mid x \in \mathbb{N} \text{ y } x > 0\}$ describe el conjunto de los enteros positivos. $A = \{x \mid x \in \mathbb{N} \text{ y } x < 5\}$ es el conjunto $\{0, 1, 2, 3, 4\}$.

Supongamos que A es un conjunto. Definimos *conjunto potencia* de A como $2^A = \{B \mid B \subseteq A\}$. Por ejemplo, sea $A = \{a, b, c\}$. Entonces 2^A es el conjunto

$$\{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$$

Obsérvese que $\emptyset \in 2^A$ y $A \in 2^A$

Supongamos que I es un conjunto. Si para todo $\alpha \in I$ tenemos que A_α es un conjunto, entonces $\{A_\alpha \mid \alpha \in I\}$ se llama *familia indexada de conjuntos*. Por

ejemplo, si para todo $n > 0$, $A_n = [-1/n, 1/n]$, entonces $\{A_n | n \in \mathbb{Z}^+\}$ es la familia de los intervalos cerrados desde $-1/n$ a $1/n$ para $n = 1, 2, 3 \ldots$

0.3 OPERACIONES CON CONJUNTOS

En aritmética se puede sumar, restar o multiplicar dos números. En la teoría de conjuntos existen tres operaciones que son análogas a las anteriores. La *unión* de conjuntos A y B se denota por $A \cup B$ y es un conjunto formado por los elementos que aparecen en A, en B o en ambos. Por tanto $A \cup B = \{x | x \in A \text{ o } x \in B\}$.

La *intersección* de A y B es el conjunto $A \cap B = \{x | x \in A \text{ y } x \in B\}$. Obsérvese que si $x \in A \cap B$ entonces se puede decir que x aparece simultáneamente en A y B.

Por ejemplo, si $A = \{0, 1, 2, 3, 4, 5\}$ y $B = \{2, 3, 5, 9\}$ entonces $A \cup B = \{0, 1, 2, 3, 4, 5, 9\}$ y $A \cap B = \{2, 3, 5\}$.

Obsérvese que $\mathbb{Z}^+ \cup \{0\} = \mathbb{N}$, mientras que $\mathbb{N} \cap \mathbb{Z}^+ = \mathbb{Z}^+$.

Se dice que los conjuntos A y B son *disjuntos* si $A \cap B = \emptyset$.

Teorema 0.3.1. Dados los conjuntos A y B, se tiene lo siguiente:

1. $\emptyset \cup A = A$.
2. $\emptyset \cap A = \emptyset$.
3. Si $A \subseteq B$, entonces $A \cap B = A$.
4. Si $A \subseteq B$, entonces $A \cup B = B$.
5. $A \cap A = A = A \cup A$.
6. (a) $A \cup B = B \cup A$.
   (b) $A \cap B = B \cap A$.
7. (a) $A \cup (B \cup C) = (A \cup B) \cup C$.
   (b) $A \cap (B \cap C) = (A \cap B) \cap C$.
8. (a) $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.
   (b) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.

Demostración. Dejaremos la demostración de la mayoría de las propiedades para que sea realizada por el lector.

7. (a) Un elemento x satisface

$$\begin{aligned} x \in A \cup (B \cup C) &\Leftrightarrow x \in A \text{ o } x \in (B \cup C) \\ &\Leftrightarrow x \in A \text{ o } (x \in B \text{ o } x \in C) \\ &\Leftrightarrow (x \in A \text{ o } x \in B) \text{ o } x \in C \\ &\Leftrightarrow x \in (A \cup B) \text{ o } x \in C \\ &\Leftrightarrow x \in (A \cup B) \cup C \end{aligned}$$

Por tanto tenemos que $A \cup (B \cup C) \subseteq (A \cup B) \cup C$ y también $(A \cup B) \cup C \subseteq A \cup (B \cup C)$ con lo que ambos conjuntos son iguales.

8. (b) Un elemento x satisface

$$\begin{aligned} x \in A \cup (B \cap C) &\Leftrightarrow x \in A \text{ o } x \in (B \cap C) \\ &\Leftrightarrow x \in A \text{ o } (x \in B \text{ y } x \in C) \\ &\Leftrightarrow (x \in A \text{ o } x \in B) \text{ y } (x \in A \text{ o } x \in C) \\ &\Leftrightarrow x \in (A \cup B) \text{ y } x \in (A \cup C) \\ &\Leftrightarrow x \in (A \cup B) \cap (A \cup C) \end{aligned}$$

con lo que tenemos que cada uno de estos conjuntos $A \cup (B \cap C)$ y $(A \cup B) \cap (A \cup C)$ es un subconjunto del otro. Por tanto los dos conjuntos son iguales. □

Si A y B son dos conjuntos cualesquiera, el *complemento* de B con respecto a A (también llamado *complemento relativo*) es el conjunto

$$A - B = \{x | x \in A \text{ y } x \notin B\}$$

Por lo tanto, $A - B$ esta compuesto por todos los elementos de A que no están también en B. Por ejemplo, si $A = \{0, 2, 4, 6, 8, 10\}$ y $B = \{0, 1, 2, 3, 4\}$, entonces $A - B = \{6, 8, 10\}$, mientras que $B - A = \{1, 3\}$.

Es conveniente pensar que todos los conjuntos aquí tratados se consideran subconjuntos de un conjunto *universal* U. Los complementos pueden ser formados con respecto a este conjunto universal. Si A es un conjunto, entonces $U - A$ es el conjunto de todos los elementos que no están en A. Conviene denotar tales complementos mediante $\overline{A}$; de forma que $U - A = \overline{A}$. Obsérvese que $\overline{\emptyset} = U$ y $\overline{U} = \emptyset$.

Teorema 0.3.2. Dados los conjuntos A y B:

1. $A - B = A \cap \overline{B}$.
2. $\overline{(A \cap B)} = \overline{A} \cup \overline{B}$.
3. $\overline{(A \cup B)} = \overline{A} \cap \overline{B}$.
4. $\overline{\overline{A}} = A$.

Demostración. Vamos a probar los Apartados 1 y 3 y dejaremos el resto para el lector.

Un elemento x satisface

$$\begin{aligned} x \in A - B &\Leftrightarrow x \in A \text{ y } x \notin B \\ &\Leftrightarrow x \in A \text{ y } x \in \overline{B} \\ &\Leftrightarrow x \in A \cap \overline{B} \end{aligned}$$

Por lo tanto, tenemos que $A - B \subseteq A \cap \overline{B}$ y $A \cap \overline{B} \subseteq A - B$, así que tenemos que $A - B = A \cap \overline{B}$ con lo que (1) queda probado.

Para el Apartado 3, un elemento satisface

$$\begin{aligned} x \in \overline{A \cup B} &\Leftrightarrow x \notin A \cup B \\ &\Leftrightarrow x \notin A \, y \, x \notin B \\ &\Leftrightarrow x \in \overline{A} \text{ y } x \in \overline{B} \\ &\Leftrightarrow x \in \overline{A} \cap \overline{B} \end{aligned}$$

Por lo tanto, tenemos que $\overline{A \cup B} \subseteq \overline{A} \cap \overline{B}$ y $\overline{A} \cap \overline{B} \subseteq \overline{A \cup B}$, con lo que los dos conjuntos son iguales. □

Téngase en cuenta que los apartados (2) y (3) del Teorema 0.3.2 se conocen como las leyes de De Morgan para conjuntos.

Dados dos conjuntos A y B, su *producto cartesiano*, $A \times B$, es el conjunto de todos los pares ordenados de los que el primer elemento proviene de A y el segundo de B. Así que,

$$A \times B = \{(a, b) \mid a \in A \text{ y } b \in B\}$$

Por ejemplo, si $A = \{1, 2, 3\}$ y $B = \{5, 6\}$ entonces

$$A \times B = \{(1, 5), (2, 5), (3, 5), (1, 6), (2, 6), (3, 6)\}$$

Obsérvese que dos pares ordenados son iguales si y sólo si los elementos correspondientes de los mismos son iguales. Por lo tanto $(a, b) = (c, d)$ sólo cuando $a = c$ y $b = d$. Luego un par ordenado es distinto que un conjunto de dos elementos.

Ejercicios de la Sección 0.3

0.3.1. Probar las siguientes afirmaciones:

(a) Si $A \subseteq B$, entonces $2^A \subseteq 2^B$.

(b) Si $A \cap B = A \cup B$, entonces $A = B$.

(c) Si $A = B$ entonces $A \cap B = A \cup B$.

0.3.2. Si $A \subseteq B$, entonces para todo conjunto C, se obtiene que $A \cup C \subseteq B \cap C$ y $A \cap C \subseteq B \cap C$. Probarlo.

0.3.3. Si $A \subseteq C$ y $B \subseteq D$ ¿se cumple que $C \cup D \subseteq A \cup B$?

0.3.4. Probar o refutar las siguientes afirmaciones:

(a) Si $A \cup B = A \cup C$, entonces $B = C$.

(b) Si $A \cap B = A \cap C$, entonces $B = C$.

0.3.5. Sea $\{B_\alpha | \alpha \in I\}$ una familia indexada de conjuntos. Se usará la notación $\cup_{\alpha \in I} B_\alpha$ para indicar la unión de todos los B_α y $\cap_{\alpha \in I} B_\alpha$ para indicar la intersección de todos los B_α. Probar que para todo conjunto A se cumplen las siguientes igualdades:

(a) $A \cap (\cup_{\alpha \in I} B_\alpha) = \cup_{\alpha \in I} (A \cap B_\alpha)$.

(b) $A \cup (\cap_{\alpha \in I} B_\alpha) = \cap_{\alpha \in I} (A \cup B_\alpha)$.

0.3.6. Demostrar las siguientes igualdades:

(a) $A - B = A - (B \cap A)$.

(b) $B \subseteq \overline{A}$ si y sólo si $A \cap B = \emptyset$.

(c) $(\cap_{\alpha \in I} B_\alpha) - A = \cap_{\alpha \in I} (B_\alpha - A)$.

(d) $(A - B) - C = (A - C) - (B - C) = A - (B \cup C)$.

(e) $A \cap \overline{B} = \emptyset$ y $\overline{A} \cap B = \emptyset$ si y sólo si $A = B$.

0.3.7. ¿Son ciertos los siguientes resultados?

(a) $2^A \cap 2^B = 2^{A \cap B}$.

(b) $2^A \cup 2^B = 2^{A \cup B}$.

0.3.8. Los pares ordenados (x, y) se definen formalmente por medio de la siguiente igualdad $(x, y) = \{\{x\}, \{x, y\}\}$. Usando la definición anterior, mostrar que $(a, b) = (c, d)$ si y sólo si $a = c$ y $b = d$.

0.3.9. Dados los conjuntos A, B y C, probar que:

(a) $A \times (B \cap C) = (A \times B) \cap (A \times C)$.

(b) $A \times (B \cup C) = (A \times B) \cup (A \times C)$.

(c) $A \times (B - C) = (A \times B) - (A \times C)$.

0.4 RELACIONES Y FUNCIONES

Una *relación* del conjunto A con el conjunto B es un subconjunto de $A \times B$. Por tanto, si $R \subseteq A \times B$ y $(a, b) \in R$, se dice que *a está relacionado con b* bajo la relación R. Por ejemplo, si $A = \{2, 3, 4, 5\}$ y $B = \{1, 3, 5, 7, 9\}$, entonces $R = \{(2, 1), (2, 3), (5, 3), (5, 5)\}$ es una relación, y 2 está relacionado con 1 bajo esta relación.

Si A y B son el mismo conjunto, se dice que la relación es una *relación sobre A*. Por ejemplo, sea $R \subseteq \mathbb{N} \times \mathbb{N}$ definida por $(x, y) \in R$ si y sólo si $x \leq y$. R es la relación "menor o igual que" sobre $\mathbb{N}$.

La relación $R \subseteq A \times B$ define dos subconjuntos, uno de A y otro de B. Estos son

$$\text{Dom } (R) = \{a \mid a \in A \text{ y } (a, x) \in R \text{ para algún } x \in B\}$$

$$\text{Im } (R) = \{b \mid b \in B \text{ y } (y, b) \in R \text{ para algún } y \in A\}$$

y se conocen como el *dominio* y la *imagen* de R, respectivamente.

Por ejemplo, si $A = \{a, b, c, d, e\}$ y $B = \{1, 2, 3, 4, 5\}$ con $R = \{(a, 1), (a, 2), (b, 5), (c, 4)\}$, entonces se tiene que

$$\text{Dom } (R) = \{a, b, c\} \text{ e Im } (R) = \{1, 2, 4, 5\}$$

Si $R \subseteq A \times B$ es una relación de A con B, entonces el conjunto $R^{-1} = \{(b, a) \mid (a, b) \in R\}$ es un subconjunto de $B \times A$. Por consiguiente, ella misma es una relación de B con A. Llamaremos a R^{-1} *inversa* de la relación R.

Sea A un conjunto no vacío. Una colección $\mathcal{A}$ de subconjuntos no vacíos de A es una *partición* de A si se cumple lo siguiente:

1. Si B y C son conjuntos en $\mathcal{A}$, entonces o bien $B = C$ o $B \cap C = \emptyset$.
2. $A = \cup_{B \in \mathcal{A}} B$.

Intuitivamente, una partición de A divide a A en partes no vacías disjuntas. Por ejemplo, sea $A = \{x \mid x \in \mathbb{N} \text{ y } x \leq 10\}$ y sea

$$\mathcal{A} = \{\{0, 2, 4\}, \{1, 3, 5\}, \{6, 8, 10\}, \{7, 9\}\}$$

una partición de A. Por otro lado,

$$\mathcal{B} = \{\{0, 2, 4, 6\}, \{1, 2, 3, 5, 7\}, \{9, 10\}, \emptyset\}$$

no es una partición.

Veamos otro ejemplo interesante. Sea $\mathbb{Q}$ el conjunto de los números racionales. Para cada $r \in \mathbb{Q}$, sea

$$Q_r = \left\{(x, y) \in \mathbb{N} \times \mathbb{Z}^+ \,\middle|\, \frac{x}{y} = r\right\}$$

Por tanto $Q_{3/8}$ contiene a (3, 8), (6, 16), (9, 24) y así sucesivamente. Obsérvese que la colección $F = \{Q_r \mid r \in \mathbb{Q}\}$ es una partición de $\mathbb{N} \times \mathbb{Z}^+$. Para verlo, primero obsérvese que si Q_r y Q_s son elementos de F y si $(x, y) \in Q_r \cap Q_s$, entonces $s = x/y = r$ con lo que $s = r$ y por tanto $Q_r = Q_s$. Puesto que $Q_r \subseteq \mathbb{N} \times \mathbb{Z}^+$ para todo r, tenemos que $\bigcup_{r \in \mathbb{Q}} Q_r \subseteq \mathbb{N} \times \mathbb{Z}^+$. Por otro lado, si $(x, y) \in \mathbb{N} \times \mathbb{Z}^+$, entonces $x/y \in \mathbb{Q}$, y, por tanto, $(x, y) \in Q_r$, siendo $r = x/y$. Así $(x, y) \in \bigcup_{r \in \mathbb{Q}} Q_r$. De todo ello se concluye que $\bigcup_{r \in \mathbb{Q}} Q_r = \mathbb{N} \times \mathbb{Z}^+$.

Vamos a ver un nuevo ejemplo de una partición de un conjunto que es la colección $\{\mathbb{Z}_0, \mathbb{Z}_1, \ldots, \mathbb{Z}_{m-1}\}$, donde m es un entero positivo fijado y $\mathbb{Z}_i$ se define como

$$\mathbb{Z}_i = \{x \mid x \in \mathbb{Z} \text{ y } x - i = km \text{ para algún entero } k\}$$

Por ejemplo, si $m = 3$, tenemos $\mathbb{Z}_0 = \{0, \pm 3, \pm 6, \pm 9, \ldots\}$, $\mathbb{Z}_1 = \{\ldots, -5, -2, 1, 4, 7, \ldots\}$ y $\mathbb{Z}_2 = \{\ldots, -4, -1, 2, 5, 8, \ldots\}$.

Supongamos que $\mathcal{A}$ es una partición del conjunto X. Definamos una relación sobre X mediante

$$R = \{(x, y) \mid x \text{ e } y \text{ están en el mismo conjunto de } \mathcal{A}\}$$

Por ejemplo, si $X = \{0, 1, 2\}$ y $\mathcal{A} = \{\{0\}, \{1, 2\}\}$, entonces R sería el conjunto $R = \{(0, 0), (1, 1), (2, 2), (1, 2), (2, 1)\}$.

Una relación definida de esta manera tiene algunas propiedades interesantes. Primero obsérvese que, si $a \in X$ y puesto que $\mathcal{A}$ es una partición de X, existe algún $A \in \mathcal{A}$ para el cual $a \in \mathcal{A}$. Por tanto $(a, a) \in R$.

Segundo, si $(a, b) \in R$ ello significa que a y b están en el mismo conjunto de $\mathcal{A}$, con lo que también b y a están en el mismo conjunto de $\mathcal{A}$. Entonces $(b, a) \in R$.

Finalmente si $(a, b) \in R$ y $(b, c) \in R$ entonces a, b y c están en el mismo conjunto de $\mathcal{A}$. En consecuencia tenemos que $(a, c) \in R$.

En resumen, para la relación

$$R = \{(x, y) \mid x \text{ e } y \text{ están en el mismo conjunto de } \mathcal{A}\}$$

tendremos lo siguiente:

1. $(a, a) \in R$ para todo $a \in X$ (propiedad reflexiva).
2. Si $(a, b) \in R$, entonces $(b, a) \in R$ (propiedad simétrica).
3. Si (a, b) y (b, c) están en R, entonces $(a, c) \in R$ (propiedad transitiva).

Toda relación que tenga estas tres propiedades se dice que es una *relación de equivalencia.*

Supongamos que R es una relación de equivalencia sobre el conjunto X. Para cada $x \in X$, se define el conjunto $[x] = \{y \in X \mid (x, y) \in R\}$. El conjunto $[x]$ se llama *clase de equivalencia de x.*

Teorema 0.4.1. Las clases de equivalencia de una relación de equivalencia R sobre un conjunto X forman una partición de X.

Demostración. Para probar este teorema se necesita demostrar que las clases de equivalencia son disjuntas entre sí y que su unión es X. Primero veremos que son disjuntas entre sí.

Supongamos que $z \in [x] \cap [y]$. Entonces $(x, z) \in R$ y $(z, y) \in R$. Dado que R es transitiva, entonces $(x, y) \in R$. Por lo tanto, $x \in [y]$ e $y \in [x]$, y en consecuencia $(x, y) \in R$ e $(y, x) \in R$. Ahora bien, si $t \in [x]$, entonces $(t, x) \in R$ y, debido a la transitividad de R, $(t, y) \in R$. Por lo tanto $t \in [y]$, con lo que se obtiene $[x] \subseteq [y]$. A la inversa, si $t \in [y]$ entonces $(t, y) \in R$, y $(t, x) \in R$, de lo que se deriva que $t \in [x]$ e $[y] \subseteq [x]$. De todo lo visto se sigue que si $[x] \cap [y] \neq \emptyset$, entonces $[x] = [y]$.

Ahora bien, puesto que R es una relación de equivalencia sobre X, todo $x \in X$ debe satisfacer $(x, x) \in R$, con lo que $x \in [x]$. Esto significa que todo elemento de X está contenido en una clase de equivalencia, con su mismo nombre. □

De lo visto anteriormente se deduce el siguiente teorema:

Teorema 0.4.2. Cualquier partición $\mathcal{A}$ de un conjunto no vacío X define una relación de equivalencia sobre X.

Por lo tanto existe una relación muy estrecha entre las relaciones de equivalencia y las particiones.

Una *función* de A a B es una relación de A con B con unas características adicionales. El conjunto $f \subseteq A \times B$ es una función si Dom $(f) = A$ y si para cualquier pareja (x, y) y (x, z) que pertenezcan a f entonces $y = z$. Esto significa que para todo elemento x de A existe un único y tal que $(x, y) \in f$. Generalmente se escribe $f: A \to B$ y se usa la notación $f(x) = y$, donde $(x, y) \in f$. De hecho, esta notación puede ser extendida a relaciones en general, como se ha hecho con la R-imagen del Ejercicio 0.4.3.

Teorema 0.4.3. Sean las funciones $f: A \to B$ y $g: A \to B$. Entonces $f = g$ si y sólo si $f(x) = g(x)$ para todo x de A.

Demostración. Supongamos que $f = g$. Sea x un elemento de A. Entonces si $y = f(x)$, se tiene que $(x, y) \in f$ y por tanto $(x, y) \in g$. En consecuencia, $y = g(x)$.

A la inversa, supongamos que $f(x) = g(x)$ para todo x de A y supongamos que (x, y) es un elemento arbitrario de f. Entonces $y = f(x) = g(x)$ con lo que $(x, y) \in g$, obteniéndose que $f \subseteq g$. Por otro lado, si (x, y) es un elemento arbitrario de g, tenemos que $y = g(x) = f(x)$ y por tanto $(x, y) \in f$, obteniéndose $g \subseteq f$. Se concluye finalmente que, $f = g$. □

Para que $f \subseteq A \times B$ sea una función, se requiere que Dom $(f) = A$. Esto es más restrictivo que lo deseable para los objetivos de este libro. Definiremos *función total* como la clase de función que hemos definido previamente. Definiremos *función parcial* como una relación f que satisface las condiciones de que Dom $(f) \subseteq A$, y que si (x, y) y (x, z) pertenecen a f, entonces $y = z$. La única diferencia que aparece ahora es que en una función parcial de A a B el dominio de la función no necesita ser el conjunto A en su totalidad. Usaremos el término función, sin modificar, para referirnos a cualquier función parcial o total que cualificaremos sólo cuando sea necesario.

Sea una función $f: A \to B$. Si $X \subseteq A$, diremos que la *imagen de X bajo f* es

$$f(X) = \{y \in B \mid y = f(x) \text{ para algún } x \in X\}$$

Si $Y \subseteq B$, la *imagen inversa de Y bajo f* es el conjunto

$$f^{-1}(Y) = \{x \in A \mid f(x) = y \text{ para algún } y \in Y\}$$

Teorema 0.4.4. Sea $f: A \to B$ una función.

1. $f(\emptyset) = \emptyset$.
2. $f(\{x\}) = \{f(x)\}$ para todo $x \in A$.
3. Si $X \subseteq Y \subseteq A$, entonces $f(X) \subseteq f(Y)$.

4. Si $X \subseteq Y \subseteq B$, entonces $f^{-1}(X) \subseteq f^{-1}(Y)$.
5. Si X e Y son subconjuntos de B, entonces $f^{-1}(X - Y) = f^{-1}(X) - f^{-1}(Y)$.

Demostración. La demostración se obtiene fácilmente a partir de las definiciones precedentes, por lo que se deja la misma para el lector. □

Una función $f: A \to B$ se dice que es *uno a uno* o *inyectiva* si, para cualesquiera $(x, y) \in f$ y $(z, y) \in f$, entonces $x = z$. Esto quiere decir que si $f(x) = f(z)$ entonces $x = z$.

Una función f se dice que es *sobreyectiva* si, para cualquier $y \in B$, existe algún $x \in A$ para el cual $f(x) = y$.

La función $f: \mathbb{N} \to \mathbb{N}$ definida por $f(n) = n$ es a la vez inyectiva y sobreyectiva. La función $g: \mathbb{N} \to \mathbb{N}$, donde $g(n) = n + 1$, es inyectiva pero *no* sobreyectiva, ya que no existe ningún $x \in \mathbb{N}$, tal que $x + 1 = 0$. La función $h: \mathbb{R} \to \mathbb{R}$, donde $h(x) = x^2$, no es ni inyectiva, ni sobreyectiva. Si f es inyectiva y sobreyectiva a la vez, se llama *biyección* o *correspondencia uno a uno*. Obsérvese que si $f: A \to B$ es sobreyectiva entonces $f^{-1}(\{b\}) \neq \emptyset$ para todo $b \in B$. Si f es una biyección, entonces, para todo $b \in B$, $f^{-1}(\{b\})$ es un conjunto con un único elemento. Por tanto cuando f es una biyección, $f^{-1}: B \to A$ es una función.

Las funciones y las relaciones se pueden asociar de una manera adecuada. Sean las relaciones $R \subseteq A \times B$ y $S \subseteq B \times C$. Definimos la *composición* de R y S como

$$S \circ R = \{(a, c) \in A \times C \mid \text{para algún } b \in B, (a, b) \in R \text{ y } (b, c) \in S\}$$

Por tanto si $R = \{(0, 1), (0, 2), (1, 1)\}$ y $S = \{(1, a), (2, b)\}$, tendremos que $S \circ R = \{(0, a), (0, b), (1, a)\}$. Por otro lado, $R \circ S = \emptyset$, puesto que no hay símbolos que aparezcan simultáneamente como primer componente de un elemento de R y como segundo componente de un elemento de S. Por tanto, en general, $R \circ S$ y $S \circ R$ no son la misma.

La composición de funciones se realiza de la misma forma:

$$g \circ f = \{(a, b) \mid \text{para algún } y,\ f(a) = y \text{ y } b = g(y)\}$$

Por ejemplo, sea $f: \mathbb{R} \to \mathbb{R}$ definida como $f(x) = x + 1$ y $g: \mathbb{R} \to \mathbb{R}$ definida como $g(x) = x^2$. Entonces tenemos que

$$g \circ f(x) = g(f(x)) = g(x + 1) = (x + 1)^2$$

y

$$f \circ g(x) = f(g(x)) = f(x^2) = x^2 + 1$$

Ejercicios de la Sección 0.4

0.4.1. Sean A y B los conjuntos $A = \{2, 3, 4, 5\}$ y $B = \{1, 3, 5, 7, 9\}$. Sea R la relación

$$R = \{(x, y) \in A \times B \mid x < y\}$$

Listar los pares ordenados de R.

0.4.2. Demostrar las siguientes igualdades:

(a) $\text{Dom}\,(R^{-1}) = \text{Im}\,(R)$.

(b) $\text{Im}\,(R^{-1}) = \text{Dom}\,(R)$.

0.4.3. Sea $R \subseteq A \times B$ una relación de A con B. Sea $X \subseteq A$. Definimos *R-imagen* de X como

$$R\,(X) = \{y \in B \mid (x, y) \in R \text{ para algún } x \in X\}$$

Obsérvese que la R-imagen de X es la imagen de la relación R restringida al subconjunto $X \times B$ de $A \times B$. Sean D y E subconjuntos de A. Probar las siguientes igualdades:

(a) $R\,(D \cup E) = R\,(D) \cup R\,(E)$.

(b) $R\,(D \cap E) = R\,(D) \cap R\,(E)$.

(c) $\text{Dom}\,(R) = R^{-1}\,(B)$.

(d) $\text{Im}\,(R) = R\,(A)$.

0.4.4. Sean las relaciones $R \subseteq A \times B$ y $S \subseteq A \times B$. Entonces $R \cup S \subseteq A \times B$ es también una relación de A con B, al igual que $R \cap S$. Demostrar las siguientes afirmaciones:

(a) $\text{Dom}\,(R \cup S) = \text{Dom}\,(R) \cup \text{Dom}\,(S)$.

(b) $\text{Im}\,(R \cup S) = \text{Im}\,(R) \cup \text{Im}\,(S)$.

(c) $(R \cup S)\,(X) = R\,(X) \cup S\,(X)$ para cualquier $X \subseteq A$.

(d) $(R \cup S)^{-1} = R^{-1} \cup S^{-1}$.

(e) $(R \cap S)^{-1} = R^{-1} \cap S^{-1}$.

0.4.5. Sea $m = 5$. Encontrar los conjuntos $\mathbb{Z}_0, \mathbb{Z}_1, \mathbb{Z}_2, \mathbb{Z}_3, \mathbb{Z}_4$.

0.4.6. Probar que $\{\mathbb{Z}_0, \mathbb{Z}_1, \ldots, \mathbb{Z}_{m-1}\}$ constituye una partición de $\mathbb{Z}$ para un $m > 0$ determinado.

0.4.7. Determinar si cada una de las relaciones siguientes es una relación de equivalencia sobre el conjunto

$$A = \{0, 1, 2, 3, 4, 5\}$$

(a) $R_1 = \{(0, 0), (1, 1), (1, 2), (2, 2), (3, 3), (4, 4), (5, 5)\}$.

(b) $R_2 = R_1 \cup \{(2, 1)\}$.

(c) $R_3 = R_1 - \{(1, 2)\}$.

(d) $R_4 = R_2 \cup \{(2, 3), (1, 3), (3, 1), (3, 2)\}$.

0.4.8. Sea $\{\mathbb{Z}_0, \mathbb{Z}_1, \mathbb{Z}_2, \mathbb{Z}_3, \mathbb{Z}_4\}$ la partición de $\mathbb{Z}$ definida en la página 13. ¿Cuál es la relación de equivalencia sobre $\mathbb{Z}$ que genera esta partición?

0.4.9. Sea $\{A_1, A_2, \ldots, A_m\}$ una partición de A y $\{B_1, B_2, \ldots, B_n\}$ una partición de B. Probar que el siguiente conjunto es una partición de $A \times B$:

$$\{A_i \times B_j \mid i = 1, 2, \ldots, m \text{ y } j = 1, 2, \ldots, n\}$$

0.4.10. Sean A y B los conjuntos definidos de la siguiente manera

$$A = \{0, 1, 2, 3\}$$

y

$$B = \{-1, 0, \tfrac{1}{2}, 1, \tfrac{3}{2}, 2, 3, 4\}$$

¿Cuáles de las siguientes relaciones son funciones totales, cuáles funciones parciales y cuáles no son funciones?

(a) $f = \{(0, 1), (1, 2), (2, 3), (3, 4)\}$

(b) $f = \{(0, 0), (1, \tfrac{1}{2}), (2, 1), (3, \tfrac{3}{2})\}$

(c) $f = \{(0, 0), (1, 1), (1, -1), (2, 3)\}$

(d) $f = \{(0, 0), (1, 3), (2, 2)\}$

(e) $f = \{(0, 0)\}$

0.4.11. Sean $f: A \to C$ y $g: B \to D$ dos funciones tales que, $f(x) = g(x)$ para todo $x \in A \cap B$. Probar que $f \cup g$ es una función de $A \cup B$ en $C \cup D$.

0.4.12. Sea $f: A \to B$ una biyección. Probar que f^{-1} también es una biyección.

0.4.13. Demostrar que si $f^{-1}(f(X)) = X$ para todo $X \subseteq A$ entonces f es inyectiva. Demostrar que si $f(f^{-1}(Y)) = Y$ para todo $Y \subseteq B$, entonces f es sobreyectiva.

0.4.14. Sea $f: A \to A$ una función para la cual $f(f(x)) = x$ para todo $x \in A$. Probar que f es una relación simétrica sobre A.

0.4.15. Sean las funciones $f: A \to B$ y $g: C \to D$ y supongamos que $A \cap C = \emptyset$ y $B \cap D = \emptyset$. Probar que $f \cup g$ es sobreyectiva si f y g lo son. Probar que $f \cup g$ es inyectiva si f y g lo son.

0.4.16. Sean f y g las funciones definidas como

$$f = \{(x, y) \mid x \in \mathbb{N} \text{ e } y \in \mathbb{Z}^+ \text{ e } y = |x| + 1\}$$

$$g = \{(x, y) | x \in \mathbb{Z}^+ \text{ e } y \in \mathbb{N} \text{ e } y = -2\,|x|\}$$

Obsérvese que $f \subseteq \mathbb{N} \times \mathbb{Z}^+$ y $g \subseteq \mathbb{Z}^+ \times \mathbb{N}$ de modo que $f \circ g$ y $g \circ f$ están definidas. Describir $f \circ g$ y $g \circ f$.

0.4.17. Supongamos que A es un conjunto y que se define la relación

$$I_A = \{(a, a) | a \in A\}$$

(a) Probar que I_A es una función.

(b) Probar que I_A es una biyección.

(c) Sea $f: X \to Y$ una función. Probar que $f \circ I_X = I_Y \circ f = f$.

(d) Si $f: X \to Y$ es una biyección, demostrar que $f \circ f^{-1} = I_Y$ y que $f^{-1} \circ f = I_X$.

0.4.18. Dadas las funciones $f: A \to B$ y $g: B \to C$. Probar que las siguientes afirmaciones son ciertas:

(a) Si f y g son inyectivas, entonces $g \circ f$ también lo es.

(b) Si f y g son sobreyectivas, entonces $g \circ f$ también lo es.

0.5 INDUCCIÓN

Se dice que un subconjunto A de $\mathbb{N}$ es un *conjunto inductivo* si, para cada $a \in A$, entonces $a + 1$ también pertenece a A. Por ejemplo, el conjunto $\{5, 6, 7, \ldots\}$ es inductivo, pero el conjunto $\{0, 2, 4, 6, 8, 10, \ldots\}$ no lo es. Para que un conjunto sea inductivo no puede tener un número finito de elementos. La mayoría de los conjuntos de números naturales que contienen el 0 no son inductivos. El hecho de que haya una única colección de números naturales que contenga el 0 y sea inductiva se conoce por el *principio de la inducción matemática* (PIM).

A continuación se muestra una breve exposición del *principio de inducción matemática*.

Dado $A \subseteq \mathbb{N}$ tal que satisface lo siguiente:

1. $0 \in A$,
2. si $n \in A$, entonces $n + 1 \in A$,

entonces $A = \mathbb{N}$.

El principio de inducción matemática es muy usado en matemáticas. Proporciona un método apropiado para definir conjuntos de objetos en los cuales hay un primer objeto, un segundo objeto y así sucesivamente. En el mismo, se define el primer objeto y el $n + 1$ se define en términos del *enésimo*.

Por ejemplo, el factorial de un número natural puede ser definido inductivamente como

$$0! = 1$$

y

$$(n+1)! = (n+1) \cdot n!, \quad \text{para} \quad n > 0$$

El PIM también se usa para probar planteamientos acerca de proposiciones que en uno u otro nivel pueden estar indexados mediante $\mathbb{N}$. En tal prueba se muestra que la colección de índices es inductiva y contiene el 0 y, que por consiguiente, es $\mathbb{N}$.

Por ejemplo, la proposición "$n + 3 < 5\ (n + 1)$ para todo número natural n" se puede demostrar como sigue:

> Sea $A = \{n \in \mathbb{N} \mid n + 3 < 5\ (n + 1)\}$. Debemos probar que A es $\mathbb{N}$. Obsérvese que si n es 0 entonces se tiene que $n + 3 = 3$ y $5\ (n + 1)$ es 5, de modo que la proposición se cumple. Por tanto se obtiene que 0 pertenece a A. Ahora supongamos que $n \in A$. Tenemos que probar que $n + 1$ también está en A. Obsérvese que $(n + 1) + 3 = n + 4 = (n + 3) + 1$. Entonces:
>
> $$\begin{aligned} 5\ ((n+1)+1) &= 5n + 10 \\ &= 5\ (n+1) + 5 \\ &> (n+3) + 5 \\ &> (n+3) + 1 \\ &= (n+1) + 3 \end{aligned}$$
>
> Por tanto la proposición se cumple para $n + 1$ cuando se cumple para n, con lo que $n + 1 \in A$ cuando $n \in A$. Por el PIM, se obtiene que $A = \mathbb{N}$, así que la proposición se cumple para todos los números naturales. □

Los pasos de una demostración en la que se usa el PIM son bastante fáciles de recordar.

1. Probar que la proposición se cumple para 0.
2. Suponer que la proposición se cumple para n y probar que esto implica que se cumpla para $n + 1$.
3. Deducir que la proposición se cumple para todos los elementos de $\mathbb{N}$.

La etapa 1 se conoce como *etapa base* o *inicial*. La suposición de que la proposición se cumple para n en la etapa 2 se conoce como *hipótesis de inducción*. La etapa 2 se llama *etapa de inducción*.

Es conveniente usar el PIM para probar proposiciones sobre colecciones de números naturales que no contienen el 0. Por ejemplo, la proposición de la fórmula $1 + 2 + \ldots + (2n - 1) = n^2$ no tiene sentido (no es verdadera) para $n = 0$. Sin embargo, esta fórmula se cumple para todo $n \geq 1$. En este caso el conjunto de inducción elegido es distinto. Obsérvese que la fórmula se cumple para $n = 1$. Sea

$$S = \{n \in \mathbb{N} | \text{ la fórmula se cumple para } 1 + n\}$$

Fíjese en que $0 \in S$. Por tanto si $n \in S$ se tiene que

$$1 + 2 + \ldots + 2\,(2\,(1 + n) - 1) = (1 + n)^2$$

Con lo que se prueba que la fórmula se cumple para $n + 1$. Es decir, $n + 1 \in S$ siempre que $n \in S$. Para $n + 1$ se suma un valor apropiado en cada lado de la igualdad y se obtiene

$$\begin{aligned} 1 + 2 + \ldots + (2\,(1+n) - 1) + (2\,(1 + (n+1)) - 1) &= (1 + n)^2 + (2\,(1 + (n + 1)) - 1) \\ &= 1 + 2\,(n + 1) + (1 + n)^2 \\ &= (1 + (n + 1))^2 \end{aligned}$$

Entonces, por medio de PIM, se tiene que $S = \mathbb{N}$ y por lo tanto la fórmula se cumple para todo $n \geq 1$. □

En la práctica, el conjunto S no se especifica. Si se expresa una propiedad como $P\,(n)$ para todo $n \geq k$, la demostración se realiza de la siguiente manera:

1. (*etapa base*) Probar que $P\,(k)$ se cumple.
2. (*etapa de inducción*) Probar que si $P\,(n)$ es verdadera entonces $P\,(n + 1)$ es verdad para todo $n \geq k$.
3. (*conclusión*) Por las etapas 1 y 2 y el PIM, $P\,(n)$ es verdadera para todo $n \geq k$.

Por tanto, la demostración del ejemplo precedente puede volver a realizarse de esta forma:

Sea $n = 1$. Entonces se tiene que $1 = 1^2$, con lo que la fórmula se cumple. Ahora se supone que la fórmula se cumple para $n \geq 1$. Es decir,

$$1 + 2 + \ldots + (2\,(1 + n) - 1) = (1 + n)^2$$

Entonces se obtiene que

$$1 + 2 + \ldots + (2(1+n) - 1) + (2(1 + (n+1)) - 1)$$
$$= (1+n)^2 + (2(1 + (n+1)) - 1)$$
$$= 1 + 2(n+1) + (1+n)^2$$
$$= (1 + (n+1))^2$$

con lo que la fórmula se cumple para $n + 1$. Entonces, debido al PIM, la fórmula se cumple para todo $n \geq 1$. □

Ejercicios de la Sección 0.5

0.5.1. Probar que, para todo $n \in \mathbb{N}$,

$$2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1$$

0.5.2. Probar que, para todo $n \geq 1$,

$$2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 2$$

0.6 CARDINALIDAD

Para comparar los distintos tamaños de los conjuntos se usan funciones entre los mismos que son biyectivas, sobreyectivas o inyectivas. Dos conjuntos A y B son *equivalentes* si existe una biyección entre ellos. Se emplea la notación $A \cong B$ para denotarlo. Por tanto tenemos que $\{x, y, z\} \cong \{1, 2, 3\}$, mientras que $\{x, y, z\} \ncong \{1, 2\}$.

Ejemplo 0.6.1.

Veamos un ejemplo trivial. Sea $\mathcal{F} = \{f \mid f: \mathbb{N} \to \{0, 1\}\}$ el conjunto de todas las funciones de $\mathbb{N}$ en $\{0, 1\}$; entonces $\mathcal{F} \cong 2^{\mathbb{N}}$. Para probar esto se necesita una biyección $H: \mathcal{F} \to 2^{\mathbb{N}}$. Por tanto, se necesita una regla que asocie cada función de $\mathcal{F}$ con algún subconjunto de $\mathbb{N}$. Para $g \in \mathcal{F}$, sea $H(g) = \{x \mid g(x) = 1\}$. Obsérvese que todas las funciones de $\mathcal{F}$ tienen una imagen bajo H. Ahora probaremos que H es inyectiva y sobreyectiva.

Para ver que H es inyectiva, sean g_1 y g_2 unas funciones que pertenecen a $\mathcal{F}$ y supongamos que $H(g_1) = H(g_2)$. Sea $x \in \mathbb{N}$. Debemos obtener que $x \in H(g_1)$ o $x \notin H(g_1)$. Si $x \in H(g_1)$ entonces, dado que $H(g_1) = H(g_2)$, se obtiene que $g_1(x) = g_2(x) = 1$. Por otro lado, si $x \notin H(g_1)$, entonces $g_1(x) = g_2(x) = 0$. Luego en ambos casos tenemos que $g_1(x) = g_2(x)$ para un $x \in \mathbb{N}$ arbitrario, con lo que $g_1 = g_2$.

Para ver que H es sobreyectiva, sea $A \in 2^{\mathbb{N}}$ un subconjunto arbitrario de $\mathbb{N}$. Se define la función $g: N \to \{0, 1\}$ como

$$g(x) = \begin{cases} 0, & \text{si } x \notin A \\ 1, & \text{si } x \in A \end{cases}$$

Obsérvese que $g \in \mathcal{F}$ y que $H(g) = A$. Por tanto para cualquier elemento A de $2^{\mathbb{N}}$, se puede encontrar una función en $\mathcal{F}$ de forma que represente a A. De esto se deduce que H es sobreyectiva.

Teorema 0.6.1. Supongamos que $A \cong C$ y $B \cong D$ con $A \cap B = \emptyset$ y $C \cap D = \emptyset$. Entonces $A \cup B \cong C \cup D$.

Demostración. Puesto que $A \cong C$ y $B \cong D$, existen unas biyecciones $g: A \to C$ y $h: B \to D$. Definamos $f: A \cup B \to C \cup D$ como

$$f(x) = \begin{cases} g(x), & \text{si } x \in A \\ h(x), & \text{si } x \in B \end{cases}$$

Por el Ejercicio 0.4.15, f es biyección puesto que g y h lo son. Por consiguiente, $A \cup B \cong C \cup D$. □

Para cada número natural $k \geq 1$, se define $\mathbb{N}_k = \{1, 2, ..., k\}$. Dichos conjuntos se usan como "estándar de tamaño" con el que se compararan otros conjuntos.

Un conjunto A es *finito* si:

1. $A = \emptyset$, en cuyo caso A tiene *cardinalidad* 0.
2. $A \cong \mathbb{N}_k$, en cuyo caso A tiene *cardinalidad* k.

Un conjunto es *infinito* si no es finito.

Por ejemplo, $A = \{a, b, c, d, e\}$ es finito con cardinalidad 5 mientras que $\mathbb{N}$ es infinito. Para simplificar, si A es finito escribiremos $|A| = k$ para representar su cardinalidad.

Supongamos que A es finito con cardinal k y que $x \notin A$. Obsérvese que $\{x\} \cong \{k+1\}$ por lo que $A \cup \{x\}$ también es finito y su cardinal es $k+1$. Este resultado se obtiene a partir del Teorema 0.6.1. y se extiende a:

Teorema 0.6.2. Si A y B son conjuntos disjuntos finitos, entonces $A \cup B$ es también finito y $|A \cup B| = |A| + |B|$.

Demostración. Si $A = \emptyset$, entonces $A \cup B = B$, con lo que

$$|A \cup B| = 0 + |B| = |B|$$

Si $A \neq \emptyset$ y $B \neq \emptyset$, entonces sean $f: A \to \mathbb{N}_m$ y $g: B \to \mathbb{N}_n$ las biyecciones a partir de las cuales se obtiene que $|A| = m$ y $|B| = n$. Se define $h: \mathbb{N}_n \to H = \{m+1, m+2, ..., m+n\}$ como $h(x) = m + x$. Es obvio que h es una biyección y por tanto $\mathbb{N}_n \cong H$. Obsérvese que $\mathbb{N}_m \cup H = \{1, 2, ..., m+n\} = \mathbb{N}_{m+n}$ y la función $\hat{f}: A \cup B \to \mathbb{N}_{n+m}$ definida como

$$\hat{f}(x) = \begin{cases} f(x), & \text{si } x \in A \\ h \circ g(x), & \text{si } x \in B \end{cases}$$

es sobreyectiva e inyectiva. Por consiguiente, $A \cup B$ es finito y $|A \cup B| = m + n = |A| + |B|$. □

Una propiedad muy utilizada para conjuntos finitos es el *principio del palomar*. En esencia dice que si hay más palomas que agujeros se debe poner más de una paloma en el mismo agujero.

Teorema 0.6.3. (*Principio del palomar*). Sean A y B conjuntos finitos con $|A| > |B| > 0$ y $f: A \to B$ una función. Entonces f no es inyectiva.

Demostración. La demostración se realiza por inducción sobre $|B|$.

Si $|B| = 1$ y $|A| > |B|$, entonces A contiene al menos dos elementos distintos a_1 y a_2. Pero entonces $f(a_1) = f(a_2)$ por lo que f no es inyectiva. Por tanto el resultado se cumple para $|B| = 1$.

Ahora supongamos que el resultado se cumple para algún conjunto B tal que $0 < |B| \leq n$. Entonces sea B un conjunto de formà que $|B| = n + 1$. Fijado un elemento $b \in B$, obsérvese que $|B - \{b\}| = n$. Supongamos que A es un conjunto tal que $|A| > |B|$ y $f: A \to B$. Consideremos los dos casos siguientes para $f^{-1}(b)$:

Caso 1: Supongamos que $|f^{-1}(b)| \geq 2$. En este caso habrá dos elementos a_1 y a_2 de A, de forma que a_1 y a_2 están en $f^{-1}(b)$ o, lo que es lo mismo, $f(a_1) = f(a_2) = b$. En este caso f no es inyectiva.

Caso 2: Supongamos que $|f^{-1}(b)| \leq 1$. Obsérvese que $|A - f^{-1}(b)| \geq |A| - 1 > n = |B - \{b\}|$. Se define la función $g: A - f^{-1}(b) \to B - \{b\}$ como $g(x) = f(x)$. Obsérvese que, como $|B - f^{-1}(b)| = n$ y $|A - f^{-1}(b)| > |B - \{b\}|$, se satisface la hipótesis de inducción. Por lo tanto g no es inyectiva con lo que

existirán a_1 y a_2 en $A - f^{-1}(b)$ para los cuales $a_1 \neq a_2$ y $g(a_1) = g(a_2)$. Por consiguiente, $f(a_1) = f(a_2)$, de lo que se deduce que f tampoco es inyectiva.

En ambos casos, si el resultado se cumple para cualquier conjunto B con n elementos, también se cumple para cualquier B con $n + 1$ elementos. Por tanto, y por el PIM, la proposición se cumple para todo conjunto finito B con $|B| > 0$. □

Hay muchos ejemplos en los que se aplica este principio. Si 11 zapatos son elegidos al azar de una caja que contiene 10 pares de zapatos, al menos se obtiene un par completo. Si $n \neq m$, entonces $\mathbb{N}_n \not\cong \mathbb{N}_m$.

Corolario 0.6.4. Si A es un conjunto finito y B es un subconjunto propio de A, entonces $A \not\cong B$.

Hay dos tamaños para los conjuntos infinitos, "grande" y "muy grande". Un conjunto A es *enumerable* si $A \cong \mathbb{N}$. En este caso, $|A| = \aleph_0$ (alef cero). Un conjunto es *numerable* si es finito o enumerable.

El conjunto $\mathbb{Z}$ es enumerable mediante la función $f: \mathbb{N} \to \mathbb{Z}$ definida por

$$f(n) = \begin{cases} \dfrac{n}{2}, & \text{si } n \text{ es par} \\ \dfrac{-(1+n)}{2}, & \text{si } n \text{ es impar} \end{cases}$$

es una biyección, que transforma 0 en 0, 1 en –1, 2 en 1, 3 en –2 y así sucesivamente.

Teorema 0.6.5. Sea A un conjunto enumerable. Si $B \subseteq A$ es un conjunto infinito, entonces B es enumerable.

Demostración. Puesto que A es enumerable, existe una biyección $f: \mathbb{N} \to A$. Supongamos que tenemos que $f(n) = a_n$ por lo que A puede ser enumerado como $A = \{a_0, a_1, \ldots\}$. Sea n_0 el menor subíndice para el cual $a_{n_0} \in B$. Sea n_1 el menor subíndice para el cual $a_{n_1} \notin B - \{a_{n_0}\}$. Generalizando, sea n_k el menor subíndice para el cual $a_{n_k} \notin B - \{a_{n_0}, a_{n_1}, \ldots, a_{n_{k-1}}\}$. Puesto que B es infinito, $B - \{a_{n_0}, a_{n_1}, \ldots, a_{n_{k-1}}\} \neq \emptyset$ para todo k, con lo que hemos construido una correspondencia uno a uno entre $\mathbb{N}$ y B. Por tanto, B es enumerable. □

Puesto que los conjuntos finitos son numerables, se tiene que todo subconjunto de un conjunto numerable, es numerable.

Obsérvese *cada* conjunto infinito contiene un subconjunto enumerable. Para probarlo, sea X infinito. Entonces $X \neq \emptyset$, por lo cual se puede seleccionar un ele-

mento de X, que llamaremos x_0. Nuevamente, y puesto que X es infinito, tenemos que $X - \{x_0\} \neq \emptyset$ y se puede elegir $x_1 \in X - \{x_0\}$. Una vez definidos cada uno de los elementos $x_0, x_1, ..., x_k$, se sabe que $X - \{x_0, x_1, ..., x_k\} \neq \emptyset$ con lo cual se puede seleccionar un

$$x_{k+1} \in X - \{x_0, x_1, ..., x_k\}$$

El conjunto $\{x_k | k = 0, 1, 2, ...\}$ es un subconjunto enumerable de X.

Terminaremos este capítulo mostrando un conjunto *no numerable*. Para ello usaremos una técnica de demostración muy eficaz llamada *diagonalización*.

Teorema 0.6.6. El conjunto $2^{\mathbb{N}}$ no es numerable.

Demostración. Supongamos que $2^{\mathbb{N}}$ es numerable. Dado que es un conjunto infinito, debe suponerse que $2^{\mathbb{N}}$ es enumerable y que por lo tanto, puede ser enumerado de la forma $2^{\mathbb{N}} = \{A_0, A_1, ...\}$. Sea $D = \{n \in \mathbb{N}; n \notin A_n\}$. Obsérvese que $D \subseteq \mathbb{N}$ y, por tanto, $D = A_k$ para algún k. Consideremos dicho k. Si $k \in A_k$, entonces puesto que $A_k = D$, k no puede estar en A_k. Por otro lado, si $k \notin A_k$, entonces $k \notin D$ y por tanto k debe estar en A_k. Ambas posibilidades nos llevan a una contradicción. Por consiguiente, la suposición de que $2^{\mathbb{N}}$ es enumerable es incorrecta. □

Sabemos, por el Ejemplo 0.6.1, que la colección $\mathcal{F}$ de funciones de $\mathbb{N}$ en $\{0, 1\}$ es equivalente a $2^{\mathbb{N}}$. Ahora por el Teorema 0.6.6, sabemos que $\mathcal{F}$ no es numerable.

La técnica de la diagonalización se usa para la refutación de muchas afirmaciones. En la demostración precedente no se ve claramente donde se usa dicha técnica. Un ejemplo clásico de diagonalización es la demostración de que el intervalo abierto (0, 1) no es numerable. Supongamos que (0, 1) es numerable, por lo que puede ser representado por el conjunto $\{a_0, a_1, ...\}$. Entonces cada a_i será representado por su desarrollo decimal y por convención se usará la forma incompleta tanto para los números de esa forma como para los de forma completa.

Por tanto, 0,25 se representará como 0,24999... Bajo esta representación, dos números en (0, 1) son iguales si y sólo si los dígitos correspondientes son los mismos. Haremos una lista con los a_i

$$a_0 = 0.d_{00}d_{01}d_{02}...$$
$$a_1 = 0.d_{10}d_{11}d_{12}...$$
$$...$$
$$a_k = 0.d_{k0}d_{k1}d_{k2}...d_{kk}...$$
$$...$$

Para demostrar que (0, 1) no es numerable debemos encontrar un número $z \in (0, 1)$ tal que $z \neq a_i$ para cualquier i. Sea $z = 0.z_0z_1\ldots$, donde

$$z_k = \begin{cases} 5, & \text{si } a_{kk} \neq 5 \\ 2, & \text{si } a_{kk} = 5 \end{cases}$$

Obsérvese que z difiere de cada a_k en al menos una cifra decimal y que $0 < z < 1$. Por tanto, hemos encontrado el z que buscábamos, con lo que a_i no da cuenta de todos los números pertenecientes a (0, 1).

Aquí la diagonalización resulta obvia.

Ejercicios de la Sección 0.6

0.6.1. Dados los conjuntos A y B, si A es finito ¿$A \cap B$ es finito?

0.6.2. Probar que si $A \subseteq B$ y A es infinito, entonces B es infinito.

0.6.3. Dar, si es posible, un ejemplo de cada apartado:

(a) Un subconjunto infinito de un conjunto finito.

(b) Una familia $\{A_i | i \in \mathbb{N}\}$ de conjuntos finitos cuya unión sea finita.

(c) Una familia $\{A_i | i \in \mathbb{N}\}$ de conjuntos finitos cuya unión no lo sea.

(d) Una familia finita de conjuntos finitos cuya unión sea infinita.

(e) Unos conjuntos finitos A y B tales que $|A \cup B| \neq |A| + |B|$.

0.6.4. Probar que $\mathbb{N} \times \mathbb{N}$ es numerable usando la función $f: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ definida como $f(n, m) = 2^n 3^m$ además del Teorema 0.6.5.

0.6.5. Mostrar que $\mathbb{N}^k$ es enumerable para cualquier $k = 1, 2, \ldots$

0.6.6. Probar que $\mathbb{R}$ no es numerable.

1

Alfabetos y lenguajes

1.1 ALFABETOS, PALABRAS Y LENGUAJES

Todo lo descrito a continuación tiene al menos dos cosas en común:

Programas escritos en algún lenguaje de programación como Pascal.
Palabras inglesas.
Secuencias de símbolos que se usan para representar un valor entero.
Frases escritas en algún lenguaje natural como el inglés.

Primero, cada uno está compuesto por secuencias de símbolos tomados de alguna colección finita. En el caso de las palabras inglesas, la colección finita es el conjunto de las letras del alfabeto junto con los símbolos que se usan para construir palabras en inglés (tales como el guión, el apóstrofe y otros por el estilo). De forma similar, la representación de enteros son secuencias de caracteres del conjunto de los dígitos {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Los programas de ordenador escritos en Pascal y las frases en inglés también están compuestos por símbolos tomados de una colección finita. Sin embargo en estos, los conjuntos de símbolos son distintos. En el caso de los programas en Pascal, el conjunto de símbolos es una colección de identificadores legales de Pascal con una longitud menor o igual que una constante, palabras clave y palabras reservadas, símbolos especiales de Pascal y espacios en blanco tales como el retorno de carro, el carácter de salto de línea y el espacio manual.

Segundo, en todos los casos vistos las secuencias de símbolos que constituyen los elementos en cuestión tienen longitud finita, aunque no existen limitaciones en cuanto a la longitud de las mismas.

La noción de secuencia finita de símbolos es el elemento principal a ser tratado por este texto. Introduciremos la notación y los nombres a usar para dichas secuencias.

Un conjunto no vacío y finito de símbolos se conoce como *alfabeto*. Por ejemplo, el alfabeto inglés está formado por 26 símbolos. En otro contexto se puede considerar como alfabeto a la colección de todas las palabras inglesas correctas o la colección de todos los símbolos legales de Pascal (los identificadores de Pascal, palabras claves y reservadas, caracteres especiales, y así sucesivamente). Si Σ es un alfabeto, $\sigma \in \Sigma$ denota que σ es un símbolo de Σ. Por tanto, si

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

podemos decir que $0 \in \Sigma$.

Obsérvese que, puesto que un alfabeto es simplemente un conjunto finito no vacío, dados Σ_1 y Σ_2 alfabetos, se tiene que $\Sigma_1 \cup \Sigma_2$ también lo es. Es más, si $\Sigma_1 \cap \Sigma_2$, $\Sigma_1 - \Sigma_2$ y $\Sigma_2 - \Sigma_1$ son conjuntos no vacíos, también son alfabetos.

Una secuencia finita de símbolos de un determinado alfabeto se conoce como *palabra* sobre dicho alfabeto. Si el alfabeto es el alfabeto inglés, algunas palabras pueden ser PROGRAM, DIGIT, MOON y BLEAK. Es más, nuestra definición permite que BXWTEEMRE y JIPOQPY sean también palabras. Nuestra experiencia nos lleva a identificar el término palabra con las palabras de algún lenguaje natural. Por esta razón, a menudo se usa el término *cadena* en lugar de palabra con el fin de evitar esta idea preconcebida. A lo largo del texto se usarán por igual los términos cadena y palabra.

Obsérvese que si nuestro alfabeto base es el conjunto de todos los identificadores legales de Pascal cuya longitud es menor o igual que una constante, las palabras clave y reservadas, los símbolos especiales de Pascal, y así sucesivamente, un programa en Pascal bien construido, es una cadena. De la misma forma que basándonos en la definición se pueden formar palabras incorrectas a partir del alfabeto inglés también se pueden formar programas incorrectos sobre el alfabeto anterior. Aparentemente, las cadenas que constituyen programas en Pascal bien construidos deben cumplir ciertas restricciones, al igual que las palabras inglesas "legales" se construyen de una determinada manera sobre el alfabeto inglés.

Cada símbolo de un alfabeto es una cadena sobre dicho alfabeto. La *cadena vacía*, la cual se denota por el símbolo ε, es una palabra sobre *cualquier* alfabeto. La palabra vacía es una secuencia vacía de símbolos tomados de cualquiera

que sea el alfabeto en cuestión. La cadena vacía tiene ciertas propiedades que veremos más adelante.

Un *lenguaje* es un conjunto de palabras. Por tanto el conjunto {1, 12, 123, 1234, 12345, 123456} es un lenguaje sobre el alfabeto compuesto por dígitos. De forma similar, la colección de palabras inglesas "correctas" es un lenguaje sobre el alfabeto inglés. Obsérvese que si Σ es un alfabeto, también es un lenguaje —el formado por todas las cadenas con un único símbolo.

Los lenguajes pueden ser bastantes grandes, como es el caso de todas las palabras inglesas "correctas" o el lenguaje {1, 11, 111, 1111, 11111, ...} formado por todas las cadenas finitas de unos. Obsérvese que este lenguaje es infinito (aunque cada cadena del mismo tenga longitud finita). Cuando un lenguaje tiene un tamaño muy grande es difícil especificar que palabras le pertenecen. La especificación de las palabras de un lenguaje es uno de los temas principales de este libro, y le dedicaremos gran parte del tiempo.

Dado que un lenguaje es un conjunto de cadenas, se puede tener el lenguaje compuesto por ninguna cadena —el *lenguaje vacío*. Éste no es el mismo lenguaje que el que consta de la cadena vacía {ε}. El lenguaje vacío se denota de la misma forma que el conjunto vacío, $\emptyset$.

Supongamos que Σ es un alfabeto y w es una cadena sobre Σ. Si L es el lenguaje formado por algunas de las cadenas sobre Σ y si w está en L, entonces se tiene que $w \in L$ y se dice que *w es un elemento de L*, o w es un miembro de L. Por tanto,

$$121 \in \{1, 12, 121, 1212, 12121\}$$

Es necesario tener en cuenta el lenguaje compuesto por *todas* las cadenas sobre el alfabeto Σ. Se conoce como *cerradura de* Σ o *lenguaje universal sobre* Σ y se denota por Σ^*. Por ejemplo, si se tiene el alfabeto $\Sigma = \{1\}$, entonces

$$\Sigma^* = \{\varepsilon, 1, 11, 111, 1111, ...\}$$

Para cualquier alfabeto, Σ^* es infinito (ya que los alfabetos son no vacíos).

Ejercicios de la Sección 1.1

1.1.1. ¿De qué conjunto de símbolos se derivan las frases inglesas?

1.1.2. ¿Por qué el lenguaje vacío $\emptyset$ no es el mismo que {ε}?

1.2 OPERACIONES CON CADENAS

Si w es una cadena sobre cualquier alfabeto, su *longitud* se denota mediante el símbolo $|w|$. La longitud de w es el número de símbolos que tiene la cadena. Así que, si $w = 121$ sobre el alfabeto $\Sigma = \{1, 2\}$, entonces $|w| = 3$. La cadena vacía ε, no tiene símbolos con lo que $|\varepsilon| = 0$.

Si w y z son cadenas, la *concatenación* de w con z es la cadena que se obtiene al añadir a la cadena w la palabra z. Por ejemplo, si $w =$ "banana" y $z =$ "rama", la concatenación de w con z es la cadena "bananarama". La concatenación de las palabras w y z se denota como wz o $w \cdot z$. Obsérvese que se tiene que

$$|wz| = |w| + |z|$$

La concatenación de la palabra vacía ε con cualquier otra palabra w, no modifica a la palabra w. Por esta razón, ε se comporta como la *identidad* con respecto a la operación de concatenación.

Vamos a introducir la noción de *potencia* de una palabra sobre un alfabeto. Sea w una palabra; para $n \in \mathbb{N}$ se define

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

Por tanto, si $w = 122$ sobre el alfabeto $\Sigma = \{1, 2\}$, se tiene

$$w^0 = \varepsilon$$
$$w^1 = 122$$
$$w^2 = 122122$$
$$w^3 = 122122122$$

y así sucesivamente. Se dice que w^i es la *potencia i-ésima de w*.

Hasta ahora hemos usado el símbolo = de forma intuitiva, sin definirlo. Para ser más precisos, definiremos la igualdad de cadenas como sigue: si w y z son palabras, se dice que w *es igual a* z, si tienen la misma longitud y los mismos símbolos en la misma posición. Se denota mediante $w = z$.

Las nociones de *sufijo* y *prefijo* de cadenas sobre un alfabeto son análogas a las que se usan habitualmente. Si w y x son palabras, se dice que x es prefijo de w, si para alguna cadena y se obtiene que $w = xy$. Por ejemplo, si w es la cadena 121, entonces la cadena $x = 12$ es un prefijo de w e $y = 1$. Si se considera $y = \varepsilon$, entonces para $w = xy$ se tiene que $w = x$, con lo que toda palabra puede considerarse prefijo de sí misma. Introduciremos el término *prefijo propio* para denotar aquellas cadenas que son prefijos de una palabra pero no iguales a la misma. Por

eso $x = 121$ es un prefijo de la cadena $w = 121$, pero no es un prefijo propio de w. Finalmente hay que tener en cuenta que la palabra vacía ε es prefijo de cualquier palabra.

Una cadena w es una *subcadena* o *subpalabra* de otra cadena z si existen las cadenas x e y para las cuales $z = xwy$.

La *inversa* o *transpuesta* de una palabra w es la imagen refleja de w. Por ejemplo, si $w =$ "able" entonces su inversa es "elba". Para denotar la inversa de w se usa w^I. Una definición más precisa de la misma puede ser la siguiente:

$$w^I = \begin{cases} w, & \text{si } w = \varepsilon \\ y^I a, & \text{si } w = ay \text{ por tanto } a \in \Sigma \text{ y } y \in \Sigma^* \end{cases}$$

Por ejemplo, supongamos que $x =$ "able". Si se sigue la definición anterior para calcular w^I se tiene:

$$\begin{aligned} x^I = (\text{able})^I &= (\text{ble})^I\, a \\ &= (\text{le})^I\, ba \\ &= (\text{e})^I\, lba \\ &= (\varepsilon)^I\, elba \\ &= \varepsilon elba \\ &= elba \end{aligned}$$

Consideremos la concatenación de las palabras "*ab*" y "*cd*" que forma "*abcd*" sobre el alfabeto inglés. Sabemos que $(abcd)^I = dcba$. Obsérvese que $dcba = (cd)^I\,(ab)^I$. Por lo tanto, si w e y son cadenas y si $x = wy$, entonces $x^I = (wy)^I = y^I\, w^I$.

La inversa se "deshace" a sí misma. Obsérvese que

$$\begin{aligned} ((abcd)^I)^I &= (dcba)^I \\ &= abcd \end{aligned}$$

En general, $(x^I)^I = x$.

Ejercicios de la Sección 1.2

1.2.1. Sea $\Sigma = \{1\}$. ¿Se puede decir que para todo número natural n hay alguna palabra $w \in \Sigma^*$ para la cual $|w| = n$? Si w es una cadena de Σ^* para la cual $|w| = n$, ¿es única? ¿Qué ocurriría si $\Sigma = \{1, 2\}$?

1.2.2. Para una palabra w, ¿se puede decir que

$$|w^{i+j}| = |w^i| + |w^j|?$$

Encontrar una expresión para $|w^{i+j}|$ en términos de i, j y $|w|$.

1.2.3. ¿La cadena vacía ε es un prefijo propio de sí misma?

1.2.4. Definir las nociones de *sufijo* y *sufijo propio* de una cadena sobre un alfabeto.

1.2.5. Obtener todos los prefijos, sufijos y subpalabras de la palabra $w =$ "bar" sobre el alfabeto inglés.

1.2.6. Probar formalmente que $(wy)^I = y^I w^I$.

1.3 OPERACIONES CON LENGUAJES

Las ideas de concatenación, potencia e inverso se pueden extender al lenguaje en su totalidad. Sean A y B lenguajes sobre un alfabeto. Se define el lenguaje *concatenación* de A y B como

$$A \cdot B = \{w \cdot x | w \in A \text{ y } x \in B\}$$

Por tanto, $A \cdot B$ está formado por todas las cadenas que se forman concatenando cada cadena de A con todas las cadenas de B.

Por ejemplo, si $A = \{casa\}$ y $B = \{pájaro, perro\}$, entonces $A \cdot B$ sería el lenguaje $\{casapájaro, casaperro\}$.

Obsérvese que para formar el lenguaje concatenación $A \cdot B$ no es necesario que A y B sean lenguajes sobre el mismo alfabeto. Si A es un lenguaje sobre Σ_1 y B es un lenguaje sobre Σ_2, entonces $A \cdot B$ es un lenguaje sobre $\Sigma_1 \cup \Sigma_2$. Se suele escribir AB en lugar de $A \cdot B$, cuando la expresión resulta ambigua.

Dado que para toda palabra x, $x \cdot \varepsilon = x = \varepsilon \cdot x$, se obtiene que para cualquier lenguaje A, $A \cdot \{\varepsilon\} = \{\varepsilon\} \cdot A = A$. Es decir, el lenguaje cuyo único elemento es la palabra vacía, se comporta como la identidad para la operación de concatenación de lenguajes.

Al igual que para las cadenas, una vez que se ha definido la concatenación de lenguajes, se puede definir la *potencia*. Sea A un lenguaje sobre el alfabeto Σ. Definimos

$$A^n = \begin{cases} \{\varepsilon\}, & \text{si } n = 0 \\ A \cdot A^{n-1}, & \text{si } n \geq 1 \end{cases}$$

Por tanto, si $A = \{ab\}$ sobre el alfabeto inglés, se obtiene que

$$A^0 = \{\varepsilon\}$$
$$A^1 = A = \{ab\}$$
$$A^2 = A \cdot A^1 = \{abab\}$$
$$A^3 = A \cdot A^2 = \{ababab\}$$

Interesa tener en cuenta que de esta definición se obtiene que $\emptyset^0 = \{\varepsilon\}$.

Puesto que un lenguaje es una colección o conjunto de cadenas, se puede definir para el mismo la unión, intersección y sublenguaje, al igual que se definen para los conjuntos en general. Si A y B son lenguajes sobre el alfabeto Σ, entonces la *unión* de A y B se denota mediante $A \cup B$ y está formada por todas las palabras que pertenecen al menos a uno de los dos lenguajes. Por tanto,

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

La *intersección* de los lenguajes A y B es el lenguaje

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

Luego, $A \cap B$ está formado sólo por las palabras que pertenecen a los lenguajes A y B a la vez.

Veamos un ejemplo. Consideremos el alfabeto $\Sigma = \{0, 1\}$ y los lenguajes $A = \{\varepsilon, 0, 1, 10, 11\}$ y $B = \{\varepsilon, 1, 0110, 11010\}$. Entonces

$$A \cup B = \{\varepsilon, 0, 1, 10, 11, 0110, 11010\}$$

y

$$A \cap B = \{\varepsilon, 1\}$$

Antes de ver la relación que existe entre la concatenación y la intersección e unión de lenguajes, es conveniente definir formalmente sublenguaje y la igualdad de lenguajes. Si A y B son lenguajes sobre un alfabeto Σ y si todas las cadenas de A son también cadenas de B, entonces se dice que A es un *sublenguaje* de B. Dado que esto se corresponde exactamente con el concepto de subconjunto visto en la teoría de lenguajes, $A \subseteq B$ denota que A es un sublenguaje de B.

Para los lenguajes $A = \{a, aa, aaa, aaaa, aaaaa\}$ y $B = \{a^n | n = 0, 1, 2, \ldots\}$, se tiene que $A \subseteq B$. Obsérvese que cualquier lenguaje L sobre el alfabeto Σ es un sublenguaje de Σ^*, es decir, $L \subseteq \Sigma^*$.

Se dice que dos lenguajes A y B son *iguales* si contienen exactamente las mismas cadenas, es decir, son conjuntos iguales. Se denota con $A = B$. Los teoremas siguientes muestran la relación que existe entre sublenguajes e igualdad.

Teorema 1.3.1. Sean A y B dos lenguajes sobre el alfabeto Σ. Entonces $A = B$ si y sólo si $A \subseteq B$ y $B \subseteq A$.

Demostración. Supongamos en primer lugar que $A = B$. Tenemos que probar que $A \subseteq B$ y $B \subseteq A$. Supongamos que $x \in A$. Puesto que A y B tienen exactamente las mismas cadenas, se obtiene que $x \in B$, de lo que se deduce que $A \subseteq B$. Análogamente, si x es una cadena que pertenece a B, entonces como A y B tienen exactamente las mismas cadenas, se obtiene que $x \in A$ y por tanto $B \subseteq A$.

Supongamos ahora que $A \subseteq B$ y $B \subseteq A$. Esto significa que toda cadena de A está también en B y viceversa. Por tanto, A y B tienen exactamente las mismas cadenas, con lo que son iguales. □

El Teorema 1.3.1 proporciona una forma de determinar oportunamente si dos lenguajes son iguales. Lo usaremos para demostrar que la concatenación es distributiva con respecto a la unión.

Teorema 1.3.2. Dados los lenguajes A, B y C sobre un alfabeto Σ, se cumple que:

i. $A \cdot (B \cup C) = A \cdot B \cup A \cdot C$

ii. $(B \cup C) \cdot A = B \cdot A \cup C \cdot A$

Demostración. (i) Probaremos primero que $A \cdot (B \cup C) \subseteq A \cdot B \cup A \cdot C$. Sea $x \in A \cdot (B \cup C)$. Entonces $x = w \cdot y$ para las cadenas $w \in A$ e $y \in B \cup C$. Puesto que $y \in B \cup C$, entonces $y \in B$ o $y \in C$. Si $y \in B$, entonces $w \cdot y \in A \cdot B$, y por tanto, $w \cdot y \in A \cdot B \cup A \cdot C$. Por otro lado, si $y \in C$, entonces $w \cdot y \in A \cdot C$, con lo que de nuevo tenemos que $w \cdot y \in A \cdot B \cup A \cdot C$. En ambos casos se obtiene que

$$A \cdot (B \cup C) \subseteq A \cdot B \cup A \cdot C$$

Para probar que $A \cdot B \cup A \cdot C \subseteq A \cdot (B \cup C)$, supongamos que $x \in A \cdot B \cup A \cdot C$. Entonces $x \in A \cdot B$ o $x \in A \cdot C$. Si $x \in A \cdot B$, entonces $x = u \cdot v$ para las cadenas $u \in A$ y $v \in B$. Puesto que $v \in B$ entonces $v \in B \cup C$ y, por tanto, $uv \in A \cdot (B \cup C)$. Por otro lado, si $x \in A \cdot C$, entonces $x = w \cdot y$ para las cadenas $w \in A$ e $y \in C$. En este caso y puesto que $y \in C$, se tiene que $y \in B \cup C$, y por tanto, $w \cdot y \in A \cdot (B \cup C)$. Luego $A \cdot B \cup A \cdot C \subseteq A \cdot (B \cup C)$. Luego por el Teorema 1.3.1 se obtiene que $A \cdot (B \cup C) = A \cdot B \cup A \cdot C$.

La prueba de la parte (ii) es similar y se deja como ejercicio. □

La relación que tiene la concatenación con la intersección no es tan buena como con la unión. Generalmente, la concatenación no es distributiva con respecto a la intersección. Para verlo, supongamos que $A = \{a, \varepsilon\}$, $B = \{\varepsilon\}$ y

$C = \{a\}$. Obsérvese que $A \cdot B = \{a, \varepsilon\}$ y $A \cdot C = \{a^2, a\}$, por lo tanto $A \cdot B \cap A \cdot C = \{a\}$. Por otro lado, $B \cap C = \emptyset$, con lo que $A \cdot (B \cap C) = \emptyset$.

Si A es un lenguaje sobre algún alfabeto Σ, se define la *cerradura de Kleene* o *cerradura de estrella* de un lenguaje A como $A^* = \bigcup_{n=0}^{\infty} A^n$. Definiremos también la *cerradura positiva* de A como $A^+ = \bigcup_{n=1}^{\infty} A^n$. Obsérvese que las cadenas de la cerradura de Kleene se forman al realizar cero o más concatenaciones de las cadenas del lenguaje, mientras que la cerradura positiva se forma al realizar una o más concatenaciones.

Por ejemplo, supongamos que $A = \{a\}$ sobre el alfabeto inglés. Entonces tenemos que $A^0 = \{\varepsilon\}$, $A^1 = \{a\}$, $A^2 = \{a^2\}$ y así sucesivamente. Por tanto $A^* = \{\varepsilon, a, a^2, a^3, \ldots\}$. Por otro lado, $A^+ = \{a, a^2, a^3, \ldots\}$.

Obsérvese que si Σ es un alfabeto, entonces Σ^* está formado por todas las concatenaciones de 0 o más símbolos de Σ. Precisamente, ésta será la colección de cadenas que constituyen el lenguaje universal, el cual también se denota por Σ^*. Por tanto, nuestra notación es consistente. Además todo lenguaje sobre Σ es necesariamente un sublenguaje de Σ^*. Es más, si A es un lenguaje sobre Σ, se obtiene que $A_n \subseteq \Sigma^*$ para todo $n = 0, 1, 2, \ldots$, y por tanto, $A^* \subseteq \Sigma^*$ y $A^+ \subseteq \Sigma^+$. Obsérvese también que puesto que $A^n \subseteq A^*$ para todo n, se tiene que $A^+ \subseteq A^*$. Finalmente, puesto que $\emptyset^0 = \{\varepsilon\}$ y $\emptyset^n = \emptyset$ para todo $n \geq 1$, entonces $\emptyset^* = \{\varepsilon\}$ y $\emptyset^+ = \emptyset$.

Ejemplo 1.3.1

Consideremos $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ y definamos A como el lenguaje formado por las cadenas que no contienen ninguno de los dígitos 2, 3, ..., 9. Entonces $\varepsilon \in A$, $0 \in A$ y $1 \in A$. También $000101001 \in A$. Obsérvese que si $k > 1$ y $x \in A^k$ entonces $x = w_1 \cdot w_2 \cdot \ldots \cdot w_k$ donde las cadenas $w_i \in A$. Puesto que las cadenas w_i no contienen los dígitos 2, 3, ..., 9, la cadena x tampoco contiene ninguno de esos dígitos. Por tanto $x \in A$. Es decir, $A^k \subseteq A$ para $k \geq 1$.

Por otro lado, si $x \in A$, como $\varepsilon \in A$ se puede poner que

$$x = \varepsilon^{k-1} \cdot x$$

la cual es una cadena de A^k. De aquí que $A \subseteq A^k$ para $k \geq 1$, y por tanto $A = A^k$ para $k \geq 1$. De esto se obtiene que

$$A^+ = \bigcup_{k=1}^{\infty} A^k = \bigcup_{k=1}^{\infty} A = A$$

Además dado que $A^0 = \{\varepsilon\} \subseteq A$, se deduce que $A^* = A^0 \cup A^+ = A^0 \cup A = A$. Por tanto en algunos casos A^* y A^+ son el mismo.

Si A y B son lenguajes sobre Σ, definiremos la *diferencia* como

$$A - B = \{x | x \in A \text{ y } x \notin B\}$$

la cual es exactamente la misma definición que se vio en la teoría de conjuntos.

Ejemplo 1.3.2

Si A es como en el Ejemplo 1.3.1 y B es el lenguaje de todas las cadenas de ceros, entonces $A - B$ es el lenguaje de todas las cadenas de ceros y unos que tienen al menos un 1.

Definimos el *complemento* de un lenguaje A sobre el alfabeto Σ como

$$\overline{A} = \Sigma^* - A$$

la cual es también una definición análoga a la dada para el complemento en la teoría de conjuntos. Nuevamente nos referiremos al Ejemplo 1.3.1 para obtener que $\overline{A}$ es el lenguaje de todas las cadenas que contienen al menos uno de los dígitos 2, 3, ..., 9.

La concatenación y la diferencia de lenguajes son incompatibles de forma similar a como lo eran la concatenación y la intersección. En general, $A(B - C) \neq AB - AC$.

En el siguiente teorema se muestra una igualdad que relaciona las cerraduras A^+ y A^*. Aunque el resultado es obvio intuitivamente hablando, conviene ofrecer una demostración rigurosa de una parte del teorema puesto que es una buena forma de indicar cómo se demuestran tales igualdades.

Teorema 1.3.3. $A^+ = A \cdot A^* = A^* \cdot A$.

Demostración. Sea $x \in A^+$. De la definición de la cerradura positiva se obtiene que $x \in \bigcup_{k=1}^{\infty} A^k$, así que para algún $k_0 \geq 1$, se sigue que $x \in A^{k_0}$.

Puesto que $A^{k_0} = A \cdot A^{k_0 - 1}$, se obtiene que $x \in A \cdot A^{k_0 - 1}$, y por tanto

$$x \in \bigcup_{n=0}^{\infty} (A \cdot A^n) = A \cdot \bigcup_{n=0}^{\infty} A^n = A \cdot A^*$$

Esto prueba que $A^+ \subseteq A \cdot A^*$.

A la inversa, sea

$$x \in A \cdot A^* = A \cdot \bigcup_{n=0}^{\infty} A^n = \bigcup_{n=0}^{\infty} (A \cdot A^n)$$

Entonces, para algún $j \geq 0$, se deduce que

$$x \in A \cdot A^j = A^{j+1} \subseteq \bigcup_{k=1}^{\infty} A^k = A^+$$

Por lo tanto $A \cdot A^* \subseteq A^+$.
La demostración de $A^+ = A^* \cdot A$ es similar y se deja para el lector. □

Consideremos el lenguaje $A = \{ab\}$ sobre el alfabeto inglés. Tenemos que

$$A^+ = \{(ab)^i \mid i \geq 1\} = \{ab, abab, ababab, \ldots\}$$

Entonces podemos considerar el lenguaje $(A^+)^i$ para distintos exponentes i. Por ejemplo, si $i = 2$, se tiene

$$(A^+)^2 = A^+ \cdot A^+ = \{ab \cdot ab, ab \cdot abab, ab \cdot ababab, \ldots,$$
$$abab \cdot ab, abab \cdot abab, abab \cdot ababab, \ldots\}$$

Puesto que podemos obtener cada uno de los lenguajes $(A^+)^i$, entonces también se puede obtener $(A^+)^+$. Es más, puesto que $(A^+)^0 = \{\varepsilon\}$, también se puede considerar que $(A^+)^*$. Luego parece que tiene sentido preguntarse cómo son los lenguajes que son cerraduras de cerraduras. Las repuestas son sorprendentemente sencillas.

Si x es una cadena de $(A^+)^+$, entonces, puesto que $(A^+)^+ = \bigcup_{k=1}^{\infty} (A^+)^k$, tenemos que para algún $n \geq 1$, $x \in (A^+)^n$, y por tanto $x = x_1 \cdot x_2 \cdot \ldots \cdot x_n$, donde cada $x_i \in A^+$. Dado que $x_i \in A^+ = \bigcup_{t=1}^{\infty} A^t$, existe algún $t_i \geq 1$ para el cual $x_i \in A^{t_i}$. Por tanto cada

$$x_i = y_{i,1} \cdot y_{i,2} \cdot \ldots \cdot y_{i,t_i}$$

donde cada $y_{i,j} \in A$. Por tanto se obtiene

$$x = (y_{1,1}\, y_{1,2} \ldots y_{1,t_1}) \cdot (y_{2,1}\, y_{2,2} \ldots y_{2,t_2}) \cdot \ldots \cdot (y_{n,1}\, y_{n,2} \ldots y_{n,t_n})$$

Pero ésta es justamente una cadena perteneciente al lenguaje $A^{t_1 + t_2 + \ldots + t_n}$. Es más, puesto que $t_i \geq 1$ para cada i, se obtiene que

$$t_1 + t_2 + \ldots + t_n \geq 1$$

por lo que es una cadena de A^+. Por tanto $(A^+)^+ \subseteq A^+$. Por otro lado, puesto que $A^+ = (A^+)^1 \subseteq \bigcup_{k=1}^{\infty} (A^+)^k = (A^+)^+$, se obtiene que $A^+ \subseteq (A^+)^+$, de lo que se desprende que $A^+ = (A^+)^+$.

De forma similar se puede demostrar que $(A^*)^* = A^*$. Estos resultados se pueden interpretar como que no se añaden nuevas cadenas a los lenguajes A^* o A^+, aunque se vuelva a realizar sobre ellos cualquier tipo de cerradura. Esto desvela intuitivamente lo que significa el término *cerradura*.

También se puede desarrollar la idea de inverso o transpuesta de un lenguaje. El *inverso de un lenguaje A* es

$$A^I = \{x^I | x \in A\}$$

Por ejemplo si $A = \{\text{dog, bog}\}$, entonces $A^I = \{\text{god, gob}\}$.

Obsérvese que si se vuelve a realizar el inverso del inverso de todas las palabras de un lenguaje, entonces se obtiene, de nuevo, el lenguaje original. Por tanto, $(A^I)^I = A$.

El comportamiento del inverso es bueno para la mayoría de las operaciones sobre lenguajes, como se muestra en el Ejercicio 1.3.18.

El inverso de la concatenación no sólo invierte las palabras concatenadas de los lenguajes sino que también cambia el orden de la concatenación de los lenguajes, como se muestra en el siguiente teorema.

Teorema 1.3.4. $(A \cdot B)^I = B^I \cdot A^I$.

Demostración. Sea $x \in (AB)^I$. Entonces $x^I \in AB$, con lo que $x^I = yz$ para las cadenas $y \in A$ y $z \in B$. Por tanto, $x = (x^I)^I = (yz)^I = z^I \cdot y^I$. Pero dado que $z \in B$, entonces $z^I \in B^I$. Además $y \in A$, con lo que $y^I \in A^I$, y por tanto se obtiene que $x \in B^I A^I$, lo cual prueba que $(AB)^I \subseteq B^I \cdot A^I$. A la inversa, si $x \in B^I A^I$, entonces $x = uw$ para alguna palabra $u \in B^I$ y $w \in A^I$. Pero entonces $x^I = w^I u^I \in AB$, con lo que $x^I \in AB$ y $x \in (AB)^I$. Por eso $B^I A^I \subseteq (AB)^I$, y por tanto $(AB)^I = B^I A^I$. □

Ejercicios de la Sección 1.3

1.3.1. Para todo lenguaje A, ¿Qué es $A \cdot \emptyset$?

1.3.2. Sean $A = \{\text{the, my}\}$ y $B = \{\text{horse, house, hose}\}$ lenguajes sobre el alfabeto inglés. Obtener $A \cdot B$, $A \cdot A$ y $A \cdot B \cdot B$.

1.3.3. Se supone que $A = \{\varepsilon, a\}$. Obtener A^n para $n = 0, 1, 2, 3$. ¿Cuántos elementos tiene A^n para un *n arbitrario*? ¿Cuáles son las cadenas de A^n para un n arbitrario?

1.3.4. Se supone que $A = \{\varepsilon\}$. Obtener A^n para un n arbitrario.

1.3.5. Sean $A = \{\varepsilon, ab\}$ y $B = \{cd\}$. ¿Cuántas cadenas hay en $A^n B$ para un n arbitrario?

1.3.6. Sean $A = \{a\}$ y $B = \{b\}$. Obtener $A^n B$, AB^n y $(AB)^n$.

1.3.7. Sean $A = \{\varepsilon\}$, $B = \{aa, ab, bb\}$, $C = \{\varepsilon, aa, ab\}$ y $D = \emptyset$ el lenguaje vacío. Obtener $A \cup B$, $A \cup C$, $A \cup D$, $B \cup D$ y $A \cap B$, $B \cap C$, $C \cap D$, $A \cap D$. Suponer que F es un lenguaje cualquiera. Obtener $F \cup D$ y $F \cap D$.

1.3.8. Probar la parte ii del Teorema 1.3.2.

1.3.9. Si A y B_i son lenguajes sobre Σ para $i = 1, 2, 3, \ldots$, probar que

$$A \cdot \left(\bigcup_{i=1}^{\infty} B_i \right) = \bigcup_{i=1}^{\infty} (A \cdot B_i)$$

1.3.10. ¿Bajo qué condiciones $A^* = A^+$?

1.3.11. Obsérvese que para todo lenguaje A se tiene que $\varepsilon \in A^*$. ¿Cuándo $\varepsilon \in A^+$?

1.3.12. Probar que $\{\varepsilon\}^* = \{\varepsilon\} = \{\varepsilon\}^+$.

1.3.13. En los Ejemplos 1.3.1 y 1.3.2 ¿por qué ε no está en los lenguajes $A - B$ y $\overline{A}$?

1.3.14. Antes se obtuvo que $A^* = A^0 \cup A^+ = \{\varepsilon\} \cup A^+$. Cabría esperar que $A^+ = A^* - \{\varepsilon\}$. Probar que, en general, la expresión no es cierta. ¿Cuándo se cumplirá que $A^+ = A^* - \{\varepsilon\}$?

1.3.15. Sean A y B dos lenguajes sobre Σ. Probar que $\overline{A \cap B} = \overline{A} \cup \overline{B}$ y que $\overline{A \cup B} = \overline{A} \cap \overline{B}$.

1.3.16. Obtener los lenguajes A, B y C tales que $A(B - C) \neq AB - AC$.

1.3.17. Probar que $(A^*)^* = A^*$, $(A^*)^+ = A^*$ y $(A^+)^* = A^*$.

1.3.18. Demostrar que se cumplen las siguiente igualdades para los lenguajes A y B sobre el alfabeto Σ:

(a) $(A \cup B)^I = A^I \cup B^I$

(b) $(A \cap B)^I = A^I \cap B^I$

(c) $(\overline{A})^I = \overline{(A^I)}$

(d) $(A^+)^I = (A^I)^+$

(e) $(A^*)^I = (A^I)^*$

PROBLEMAS

1.1. Sea $\Sigma = \{a, b, ..., z\}$ el alfabeto inglés. Definir la relación $<$ sobre Σ^*, de forma que $x < y$ siempre que x preceda a y en orden alfabético (ordenación lexicográfica).

1.2. Sea A un lenguaje sobre el alfabeto Σ. ¿Cuándo se cumple $\overline{\overline{A}} = A$?

1.3. Sea $\Sigma = \{a, b, c\}$ y sea $L = \{c^i x c^j \mid i, j \geq 0\}$, donde x se restringe a $x = \varepsilon$, $x = aw$ o $x = wb$ para algún $w \in \Sigma$. ¿Se cumple que $L = \Sigma^*$? ¿Es cierto que $L^2 = \Sigma^*$?

1.4. Sea $\Sigma = \{a, b\}$. Lo siguiente es una definición recursiva del lenguaje A:

i. $\varepsilon \in A$.
ii. Si $x \in A$, entonces axb y bxa pertenecen a A.
iii. Si x e y pertenecen a A, entonces xy pertenece a A.
iv. No hay nada más en A.

(a) Probar que

$$A = \{w \in \Sigma^* \mid w \text{ tiene el mismo número de } aes \text{ que de } bes\}$$

(b) Si b y ε están en A ¿qué más palabras hay en A?
(c) Dar una definición recursiva para que $A \subseteq \{a, b\}^*$ contenga todas las palabras que tienen el doble de *aes* que *bes*.

1.5. Un *palíndromo* es una cadena que se lee igual hacia adelante que hacia atrás. Por ejemplo, la palabra"a" es un palíndromo, al igual que la cadena "radar". Dar una definición recursiva de un palíndromo (obsérvese que ε es un palíndromo).

1.6. Probar que para los lenguajes A y B, $(A \cup B)^* = (A^* B^*)^*$.

1.7. **Desigualdad de McMillan.** Aunque la definición de que un alfabeto es un conjunto finito de símbolos es matemáticamente correcta, plantea problemas cuando se trata de aplicarla. Por ejemplo, si $\Sigma = \{1, 11\}$, entonces $111 \in \Sigma^*$, pero no está claro si $111 = 1 \cdot 11$ ó $111 = 1 \cdot 1 \cdot 1$ ó $111 = 11 \cdot 1$. El problema es que, aunque desde un punto de vista teórico 11 es indivisible (es un único símbolo y no un par de unos), no tenemos forma de representarlo. Por tanto, aunque $w_1 = 1 \cdot 11$ y $w_2 = 11 \cdot 1$ parecen iguales, no lo son. Según la igualdad de cadenas, se deben tener los mismos símbolos y en la misma posición. En este problema se puede imponer una condición a los alfabetos cuyos símbolos son cadenas sobre otros alfabetos. Esta condición garantiza que no se tenga el problema anterior con respecto a la igualdad de cadenas. Se conoce como la desigualdad de MacMillan y se enuncia como sigue:

Desigualdad de McMillan: Sea Σ un alfabeto formado por r símbolos. Sean $a_1, a_2, ..., a_q$ cadenas no vacías sobre Σ. Si el conjunto $\{a_1, a_2, ..., a_q\}$ es un alfabeto, entonces

$$\sum_{i=1}^{q} r^{-|a_i|} \leq 1$$

La demostración de la desigualdad de McMillan no es difícil pero requiere una profundización en ciertas ideas sobre cadenas.

1. Obsérvese que para las cadenas w_1 y w_2, $|w_1| + |w_2| = |w_1 w_2|$, ¿cómo será un término arbitrario de $\left(\sum_{i=1}^{q} r^{-|a_i|}\right)^n$ después de que se realice el producto (pero antes de efectuar cualquier simplificación)?
2. Sea

$$I_k = \{(i_1, i_2, ..., i_n) | k = |a_{i_1}| + ... + |a_{i_n}|\}$$

Es decir, I_k es la colección de n-tuplas de los índices de las a_i, que pueden ser usadas para formar cadenas de longitud k. Sea $M_k = |I_k|$, que es el número de cadenas de las a_i que forman cadenas de longitud k. Probar que

$$\left(\sum_{i=1}^{q} r^{-|a_i|}\right)^n = \sum_{k=n}^{nt} r^{-k} M_k$$

donde $t = \max\{|a_1|, |a_2|, ..., |a_n|\}$.

3. Por la parte 2, puesto que $M_k \leq r^k$ para todo k, se obtiene que

$$\left(\sum_{i=1}^{q} r^{-|a_i|}\right)^n \leq n(t-1) \leq nt$$

Probar que si

$$\sum_{i=1}^{q} r^{-|a_i|} > 1$$

se llega a una contradicción y por tanto se cumple la desigualdad de McMillan.

La desigualdad de McMillan se usa para deducir un resultado más fuerte.

4. Sean $l_1, l_2, ..., l_q$, q números naturales y Σ un alfabeto con $r = |\Sigma|$. Existirán q cadenas $a_1, a_2, ..., a_q$ en Σ^* de longitudes $l_1, l_2, ..., l_q$, respectivamente, que formarán un alfabeto si y sólo si

$$\sum_{i=1}^{q} r^{-|l_i|} \leq 1$$

1.8. Cadenas exentas de cuadrados y exentas de cubos. Sea Σ un alfabeto. Una cadena $w \in \Sigma^*$ se dice que *está exenta de cuadrados* si w no es de la forma uv^2x para las subpalabras u, x y v, donde $x \neq \varepsilon$. La definición de cadena *exenta de cubos* es similar. La cadena w está *fuertemente exenta de cubos* si no contiene ninguna subcadena de la forma x^2a, donde $x \neq \varepsilon$ y a es el primer símbolo de x.

1. Probar que ninguna cadena de longitud mayor o igual que 4 sobre un alfabeto de cardinalidad 2 puede estar exenta de cuadrados.

   Sea $w \in \Sigma^*$. Una cadena $w' \in \Sigma^*$ para la cual $|w| = |w'|$ es una *interpretación de w* si se satisface la siguiente condición:

   Sea $1 \leq i \leq |w|$ y $1 \leq j \leq |w|$. Si los símbolos i-ésimo y j-ésimo de w son distintos, entonces los símbolos i y j de w' son también distintos.

   Por ejemplo, si $w = aaabab$, entonces *cdefgh* y *cdcece* son interpretaciones de w, sin embargo *cdefgc* no lo es.
2. Probar que si w está exenta de cuadrados, exenta de cubos o estrictamente exenta de cubos, entonces toda interpretación de w también lo está.

   Una ω-cadena (omega cadena) es una secuencia infinita de símbolos sobre un alfabeto. Es decir, ω-cadenas son cadenas de longitud infinita.
3. Sea w, una ω-cadena exenta de cuadrados, exenta de cubos o estrictamente exenta de cubos sobre Σ en la cual aparecen todos los símbolos de Σ. Supongamos que Σ' es un alfabeto que contiene estrictamente a Σ (es decir, $\Sigma \subset \Sigma'$). Probar que se puede construir una ω-cadena a partir de w, que estará también exenta de cuadrados, exenta de cubos o estrictamente exenta de cubos (como lo es w) y que contendrá todos los símbolos de Σ'.

   Si una cadena u ω-cadena w está exenta de cuadrados, se puede esperar que w contenga dos ocurrencias de la misma subcadena x, es decir, una subcadena de la forma $xy = zx$ para la cual $1 \leq |y| = |z| < |x|$. Algunas pruebas sencillas sugieren que la construcción de dicha w es más difícil de lo que parece. De hecho, ¡no es posible!
4. Probar que si w es una cadena o ω-cadena que contiene una subcadena xy de forma que $xy = zx$ y $1 \leq |y| = |z| < |x|$, entonces w no está exenta de cuadrados.

2

Lenguajes regulares

2.1 LENGUAJES SOBRE ALFABETOS

No es una coincidencia el que la mayoría de los lenguajes considerados hasta ahora hayan sido bastante sencillos. Los procesos vistos hasta ahora para determinar qué cadenas pertenecen a un lenguaje sobre algún alfabeto Σ resultan pesados y laboriosos excepto para Σ^* y algún otro lenguaje sencillo. Nuestro objetivo a partir de ahora será la definición de lenguajes —esto es, especificar exactamente qué cadenas componen un lenguaje—. Ya que todos los lenguajes sobre Σ son sublenguajes del lenguaje universal Σ^*, tiene sentido determinar primero cuántos sublenguajes tiene Σ^* para un alfabeto Σ en particular. Comenzaremos estudiando el propio Σ^*.

Como ejemplo, consideremos el alfabeto $\Sigma = \{a, b\}$. Para todo número natural n, hay sólo un número finito de palabras sobre Σ cuya longitud es n. (¿Cuántas hay?). Aún más, dichas cadenas se pueden ordenar lexicográficamente. Por lexicográficamente, se entiende la forma en la que estarían ordenadas en el diccionario. Por conveniencia, numeraremos ε como 0, después numeraremos las palabras de longitud 1 y, en general, numeraremos las palabras de longitud $n + 1$ después de las de longitud n. Así tenemos

ε	0
a	1
b	2
aa	3
ab	4
ba	5
bb	6
aaa	7

y así sucesivamente

De manera más general, supongamos que tenemos un alfabeto arbitrario Σ. Puesto que todos los alfabetos son finitos, podemos asignar un orden arbitrario a los caracteres pertenecientes a Σ. Así, sin pérdida de generalidad, podemos escribir

$$\Sigma = \{a_1, a_2, ..., a_n\}$$

Numeraremos las palabras de Σ^* de la misma forma.

ε	0
a_1	1
a_2	2
a_n	n
a_1a_1	$n+1$
a_1a_2	$n+2$

y así sucesivamente

Esta técnica de asignar números naturales a las cadenas de un lenguaje, se puede realizar de forma más precisa. Volviendo al ejemplo original, donde $\Sigma = \{a, b\}$, haremos que cada cadena sobre Σ sea representada por un número en binario, usando los dígitos 1 y 2 en vez de 0 y 1. Sea a el 1 y b el 2, entonces se obtiene

ε	0
a	1
b	2
aa	$11 = 3$
ab	$12 = 4$
$abaa$	$1211 = 19$

De esta forma, cada palabra está representada por un entero único. Esto no ocurriría si hubiésemos usado el 0 para representar a a y el 1 a b, ya que a^i estaría representada por 0, para todo $i \geq 0$.

Obsérvese que para todo número natural m, existe una única representación del mismo en base n. Por eso podemos encontrar una cadena en Σ^* correspondiente a m. Si $m > 0$, entonces debemos obtener la representación de m en base 2 (usando los dígitos 1 y 2 en vez de 0 y 1). Para ello, se concatenan los caracteres que corresponden a los dígitos que aparecen en la representación de m en base 2. Así, si $m = 32$, primero se convierte m a 11112 y entonces se concatenan las *aes* y la *bes* hasta obtener *aaaab*.

Hemos visto una forma de relacionar las cadenas de Σ^* con los números naturales, de forma que cada cadena está representada por un único número natural y cada número natural representa a una única cadena. Esto, esencialmente, define una función de $\mathbb{N}$ a Σ^*, con lo que, de hecho, se ha esbozado la demostración del siguiente teorema:

Teorema 2.1.1. Para todo alfabeto Σ, Σ^* es infinito numerable. □

Ahora que ya sabemos el tamaño de Σ^*, se puede determinar cuántos sublenguajes de Σ^* existen y, en consecuencia, cuántos lenguajes hay sobre el alfabeto Σ.

Teorema 2.1.2. El conjunto de todos los lenguajes sobre Σ no es numerable.

Demostración. Supongamos que el conjunto de todos los lenguajes sobre Σ es numerable. Llamaremos a dicho conjunto $\mathbb{L}$. Puesto que $\mathbb{L}$ es numerable, puede ser enumerado de la forma $A_0, A_1, A_2, \ldots$ Usaremos el método de la diagonalización para llegar a una contradicción.

Sabemos que Σ^* es numerable y por tanto puede ser enumerado como w_0, $w_1, \ldots$. Sea $B = \{w_i | w_i \notin A_i\}$. Luego B está formado por las palabras que no pertenecen al lenguaje que tiene el mismo índice que las mismas. Obsérvese que B es un conjunto de cadenas sobre Σ y que, por tanto, es un lenguaje. De aquí que $L = A_k$, para algún k. Obsérvese que si $w_k \in B$, entonces w_k no está en $A_k = B$. Por tanto, w_k está y no está en A_k, lo cual es una contradicción. Por otro lado, si $w_k \notin B$, entonces, de la definición de B, se obtiene que w_k es una cadena de $A_k = B$ y está en B. Por tanto, w_k está y no está en B, con lo que se llega a otra contradicción. Dado que w_k debe o estar en B o no estar en B, la suposición de que el conjunto de todos los lenguajes sobre Σ es numerable es falsa. Luego el conjunto no es numerable. □

El Teorema 2.1.2 proporciona una idea acerca de la magnitud del problema de especificar lenguajes: hay una cantidad innumerable de lenguajes que especificar sobre un alfabeto en particular. No existe ningún método de especificación de lenguajes que sea capaz de definir *todos* los lenguajes sobre un alfabeto. Esto significa que dado un método de representación de lenguajes, hay lenguajes que

no son representables. Por otro lado, unos métodos tienen mayor fuerza expresiva que otros, es decir, unos definen más lenguajes que otros. Con el estudio de estos métodos de investigación podemos hacernos una idea de la naturaleza misma de la computación.

Ejercicios de la Sección 2.1

2.1.1. Para el alfabeto $\Sigma = \{a, b\}$ y usando la aplicación $\Sigma^* \to \mathbb{N}$ dada en este apartado, ¿cuántas palabras de longitud 3 hay? ¿Y de longitud 5? ¿Y de longitud k? ¿Cuál es el último número asignado a las palabras de longitud 2? ¿Y a las de longitud 5? ¿Y a las de longitud k?

2.1.2. En el caso general de que haya n caracteres en el alfabeto Σ ¿cuántas palabras de longitud k habrá? Si ordenamos las palabras de Σ en orden lexicográfico y les asignamos números comenzando por el 0 para ε ¿Cuál será el número asignado a la última palabra de longitud k?

2.1.3. Usando los dígitos 1 y 2 en vez de 0 y 1, obtener la representación binaria del número decimal 22. ¿Cuál es la palabra perteneciente a $\Sigma^* = \{a, b\}^*$ correspondiente al número (en base decimal) 22?

2.1.4. Ampliaremos la idea precedente para $\Sigma = \{a, b, c\}$ usando una representación ternaria con los dígitos 1, 2, 3. Supongamos que a está asociado con 1, b con 2 y c con 3. ¿Cuál será el entero decimal que corresponde a la palabra *abbacca* de Σ^*? Encontrar la palabra de Σ^* correspondiente a 20.

2.2 LENGUAJES REGULARES Y EXPRESIONES REGULARES

El primer método para especificar lenguajes que vamos a estudiar define el conjunto de lenguajes llamado *lenguajes regulares* sobre un alfabeto. Los lenguajes regulares son interesantes desde el punto de vista práctico porque pueden ser usados para especificar la construcción de analizadores léxicos —programas que analizan un texto y extraen los lexemas (o unidades léxicas) que hay en el mismo. Para un alfabeto Σ dado, los lenguajes regulares sobre Σ son interesantes desde el punto de vista teórico porque ellos constituyen el menor conjunto de lenguajes sobre Σ que es cerrado con respecto a las operaciones de concatenación, la cerradura de Kleene y la unión de lenguajes y además contiene el lenguaje vacío $\emptyset$ y los lenguajes unitarios $\{a\}$ para $a \in \Sigma$.

Definición 2.2.1. Sea Σ un alfabeto. El conjunto de los lenguajes regulares sobre Σ se define recursivamente como sigue:

(a) $\emptyset$ es un lenguaje regular.

(b) $\{\varepsilon\}$ es un lenguaje regular.

(c) Para todo $a \in \Sigma$, $\{a\}$ es un lenguaje regular.

(d) Si A y B son lenguajes regulares, entonces $A \cup B$, $A \cdot B$ y A^* son lenguajes regulares.

(e) Ningún otro lenguaje sobre Σ es regular.

Por tanto, el conjunto de los lenguajes regulares sobre Σ está formado por el lenguaje vacío, los lenguajes unitarios incluido $\{\varepsilon\}$ y todos los lenguajes obtenidos a partir de la concatenación, unión y cerradura de estrella de lenguajes.

Ejemplo 2.2.1

Dado $\Sigma = \{a, b\}$, las siguientes afirmaciones son ciertas:

$\emptyset$ y $\{\varepsilon\}$ son lenguajes regulares.
$\{a\}$ y $\{b\}$ son lenguajes regulares.
$\{a, b\}$ es regular porque es la unión de $\{a\}$ y $\{b\}$.
$\{ab\}$ es regular.
$\{a, ab, b\}$ es regular.
$\{a^i \mid i \geq 0\}$ es regular.
$\{a^i b^j \mid i \geq 0 \text{ y } j \geq 0\}$ es regular.
$\{(ab)^i \mid i \geq 0\}$ es regular.

¿El lenguaje de todas las cadenas sobre $\{a, b, c\}$ que no tienen ninguna subcadena ac es un lenguaje regular? Para responder a esta pregunta, consideremos que A es ese lenguaje. Si A es regular, entonces puede ser escrito en la forma que se indica en la definición. Obsérvese que las unidades de construcción básica son los lenguajes $\{a\}$, $\{b\}$, $\{c\}$, $\emptyset$ y $\{\varepsilon\}$. Supongamos que w es una palabra perteneciente a A. Entonces w comienza por 0 ó más *ces*. Si las suprimimos obtenemos una subcadena w que no empieza por ningún carácter c. Esta subcadena estará constituida por *aes*, *bes* y *ces*, donde cualquier bloque de *ces* sigue a las *bes*. Es más, no puede haber ningún bloque de *ces* al principio de w'. Por eso se tiene

$$w' \in (\{a\} \cup \{b\}\ \{c\}^*)^*$$

y por tanto

$$w \in \{c\}^*\ (\{a\} \cup \{b\}\ \{c\}^*)^*$$

de lo cual se obtiene

$$A \subseteq \{c\}^*\ (\{a\} \cup \{b\}\{c\}^*)^*$$

Para probar la otra inclusión, obsérvese que si u es una cadena que tiene una subcadena ac entonces

$$u \notin \{c\}^* (\{a\} \cup \{b\}\ \{c\}^*)^*$$

ya que no hay forma de que una c pueda seguir a una a. Por tanto

$$\{c\}^* (\{a\} \cup \{b\}\ \{c\}^*)^* \subseteq A$$

Podemos simplificar la especificación de un lenguaje regular introduciendo un especie de abreviatura llamada *expresión regular*. Convenimos en escribir a en lugar del lenguaje unitario $\{a\}$. Por tanto

$a \cup b$	denota	$\{a, b\} = \{a\} \cup \{b\}$
ab	denota	$\{ab\}$
a^*	denota	$\{a\}^*$;
a^+	denota	$\{a\}^+$

Además, se establece que el orden de precedencia de los operadores *, $\cup$ y $\cdot$ es * primero, $\cdot$ el siguiente y $\cup$ el último. Esto reduce la necesidad del uso de paréntesis y hace que las expresiones sean más fáciles de leer. Por ejemplo, una expresión $(\{a\}^*\{b\}) \cup \{c\}$, se reduce a la expresión regular $a^*b \cup c$.

A continuación definiremos de forma recursiva lo que es una expresión regular sobre el alfabeto Σ, usando la notación convenida:

1. $\emptyset$ y ε son expresiones regulares.
2. a es una expresión regular para todo $a \in \Sigma$.
3. Si r y s son expresiones regulares, entonces $r \cup s$, $r \cdot s$ y r^* también lo son.
4. Ninguna otra secuencia de símbolos es una expresión regular.

Comparando esta definición con la definición de lenguajes regulares se deduce que toda expresión regular sobre Σ denota un lenguaje regular sobre Σ.

Por ejemplo, el lenguaje de todas las cadenas sobre $\{a, b, c\}$ que no tiene ninguna subcadena ac se denota mediante la expresión regular $c^* (a \cup bc^*)^*$.

Cuando sea necesario distinguir entre una expresión regular r y el lenguaje denotado por la misma, usaremos $L(r)$ para denotar el lenguaje. En cualquier caso, si se afirma que $w \in r$, ello equivale a decir que $w \in L(r)$. Si r y s son expresiones regulares sobre el mismo alfabeto y si $L(r) = L(s)$, entonces se dice que r y s son *equivalentes*. En el caso de que r y s sean equivalentes se puede es-

cribir $r = s$. También se puede usar $r \subseteq s$ en el caso de que $L(r) \subseteq L(s)$. Obsérvese que para obtener $r = s$ se debe demostrar que $r \subseteq s \subseteq r$.

Fíjese que de la definición de cerradura de estrella para lenguajes se obtiene que $\emptyset^* = \{\varepsilon\}$, y en términos de expresiones regulares se tiene que $\emptyset^* = \varepsilon$. Por tanto, se podría omitir ε en la definición de expresiones regulares. No obstante, como ε es una forma de abreviar $\emptyset^*$, se incluirá ε, más por conveniencia que por necesidad. Igualmente, abreviaremos la expresión rr^* por medio de r^+.

Obsérvese que hay muchas expresiones regulares que denotan el mismo lenguaje. Por ejemplo $(a^*b)^*$ y $\varepsilon \cup (a \cup b)^* b$ denotan el mismo lenguaje: el lenguaje de todas las cadenas con 0 ó más *aes* y *bes*, que son tanto la cadena vacía como las que tienen una b al final. Por tanto, $(a^*b)^* = \varepsilon \cup (a \cup b)^* b$. Se pueden simplificar las expresiones regulares reemplazándolas por otras equivalentes pero menos complejas. Por ejemplo, la expresión $ab \cup \varepsilon \cup (a \cup b)^* b$ puede ser sustituida por $ab \cup (a^*b)^*$.

Existen muchas equivalencias con respecto a expresiones regulares basadas en las correspondientes igualdades de lenguajes. Las resumimos en el siguiente teorema.

Teorema 2.2.2. Sean r, s y t expresiones regulares sobre el mismo alfabeto Σ. Entonces:

1. $r \cup s = s \cup r$.
2. $r \cup \emptyset = r = \emptyset \cup r$.
3. $r \cup r = r$.
4. $(r \cup s) \cup t = r \cup (s \cup t)$.
5. $r\varepsilon = \varepsilon r = r$.
6. $r\emptyset = \emptyset r = \emptyset$.
7. $(rs)\, t = r\,(st)$.
8. $r\,(s \cup t) = rs \cup rt$ y $(r \cup s)\, t = rt \cup st$.
9. $r^* = r^{**} = r^* r^* = (\varepsilon \cup r)^* = r^*\,(r \cup \varepsilon) = (r \cup \varepsilon)\, r^* = \varepsilon \cup rr^*$.
10. $(r \cup s)^* = (r^* \cup s^*)^* = (r^* s^*)^* = (r^* s)^*\, r^* = r^*\,(sr^*)^*$.
11. $r\,(sr)^* = (rs)^*\, r$.
12. $(r^* s)^* = \varepsilon \cup (r \cup s)^*\, s$.
13. $(rs^*)^* = \varepsilon \cup r\,(r \cup s)^*$.
14. $s\,(r \cup \varepsilon)^*\,(r \cup \varepsilon) \cup s = sr^*$.
15. $rr^* = r^* r$.

Muchas de estas igualdades se pueden demostrar mediante *reasociación*. Como muestra, consideremos la igualdad 11, $r\,(sr)^* = (rs)^*\,r$. Si $w \in r\,(sr)^*$, entonces $w = r_0\,(s_1\,r_1)\,\ldots\,(s_n\,r_n)$ para algún $n \geq 0$. Puesto que la concatenación es asociativa, se puede reasociar la última expresión, con lo que se obtiene $w = (r_0\,s_1)\,(r_1\,s_2)\,\ldots\,(r_{n-1}\,s_n)\,r_n \in (rs)^*\,r$. De aquí se obtiene que $r\,(sr)^* \subseteq (rs)^*\,r$ [o $L\,(r\,(sr)^*) \subseteq L\,((rs)^*\,r)$]. Igualmente se puede probar que $(rs)^*\,r \subseteq r\,(sr)^*$ con lo que se demuestra la igualdad.

Para probar igualdades también se puede hacer uso de las igualdades ya conocidas. Por ejemplo, si $r = s^*t$, entonces

$$\begin{aligned} r = s^*t &= (\varepsilon \cup s^+)\,t && \text{ya que } s^* = \varepsilon \cup s^+ \\ &= (\varepsilon \cup ss^*)\,t \\ &= \varepsilon t \cup ss^*t && \text{por (8)} \\ &= t \cup sr && \text{por (5)} \\ &= sr \cup t && \text{por (1)} \end{aligned}$$

lo cual prueba que $r = s^*t$, implica que $r = sr \cup t$.

Ejercicios de la Sección 2.2

2.2.1. Verificar, aplicando la definición de lenguaje regular, que los siguientes son lenguajes regulares sobre $\Sigma = \{a, b\}$:

(a) $\{a^i \mid i > 0\}$.

(b) $\{a^i \mid i > n\}$ para un $n \geq 0$ fijado.

(c) $\{w \in \Sigma^* \mid w$ termina con $a\}$.

2.2.2. Verificar que el lenguaje de todas las cadenas de *unos* y *ceros* que tienen al menos dos *ceros* consecutivos, es un lenguaje regular.

2.2.3. Los identificadores de Pascal son cadenas de longitud arbitraria compuestas por caracteres alfabéticos y por dígitos. Los identificadores de Pascal deben empezar por un carácter alfabético. ¿Es este lenguaje un lenguaje regular?

2.2.4. Obtener una expresión regular que represente el lenguaje de los identificadores de Pascal.

2.2.5. (a) Probar que $(r \cup \varepsilon)^* = r^*$.

(b) Probar que $(b \cup aa^*b) \cup (b \cup aa^*b)\,(a \cup ba^*b)^*\,(a \cup ba^*b)$ y $a^*b\,(a \cup ba^*b)^*$ son equivalentes.

2.2.6. Sobre $\Sigma = \{a, b, c\}$ ¿son equivalentes las parejas de expresiones regulares de cada apartado?

(a) $(a \cup b)^* a^*$ y $((a \cup b)\, a)^*$.

(b) $\emptyset^{**}$ y ε.

(c) $((a \cup b)\, c)^*$ y $(ac \cup bc)^*$.

(d) $b\,(ab \cup ac)$ y $(ba \cup ba)\,(b \cup c)$.

2.2.7. Simplificar:

(a) $\emptyset^* \cup a^* \cup b^* \cup (a \cup b)^*$.

(b) $((a^*b^*)^* \cdot (b^*a^*)^*)^*$.

(c) $(a^*b)^* \cup (b^*a)^*$.

(d) $(a \cup b)^*\, a\, (a \cup b)^*$.

2.2.8. Probar que $(aa)^* a = a\,(aa)^*$.

2.2.9. Simplificar las siguientes expresiones regulares:

(a) $(\varepsilon \cup aa)^*$.

(b) $(\varepsilon \cup aa)\,(\varepsilon \cup aa)^*$.

(c) $a\,(\varepsilon \cup aa)^*\, a \cup \varepsilon$.

(d) $a\,(\varepsilon \cup aa)^*\,(\varepsilon \cup aa) \cup a$.

(e) $(a \cup \varepsilon)\, a^*b$.

(f) $(\varepsilon \cup aa)^*\,(\varepsilon \cup aa)\, a \cup a$.

(g) $(\varepsilon \cup aa)\,(\varepsilon \cup aa)^*\,(\varepsilon \cup aa) \cup (\varepsilon \cup aa)$.

(h) $(\varepsilon \cup aa)\,(\varepsilon \cup aa)^*\,(ab \cup b) \cup (ab \cup b)$.

(i) $(a \cup b)\,(\varepsilon \cup aa)^*\,(\varepsilon \cup aa) \cup (a \cup b)$.

(j) $(aa)^*\, a \cup (aa)^*$.

(k) $a^*b\,((a \cup b)\, a^*b)^* \cup a^*b$.

(l) $a^*b\,((a \cup b)\, a^*b)^*\,(a \cup b)\,(aa)^* \cup a\,(aa)^* \cup a^*b\,((a \cup b)\, a^*b)^*$.

2.3 AUTÓMATA FINITO DETERMINISTA

Consideremos el lenguaje regular A representado por $c^*\,(a \cup bc^*)^*$. Si dada una cadena w se nos pregunta si w pertenece a A, debemos analizar no sólo los caracteres que aparecen en w, sino también sus posiciones relativas. Por ejemplo, la cadena abc^5c^3ab está en A, sin embargo $cabac^3bc$ no lo está. Podemos construir un diagrama que nos ayude a determinar los distintos miembros del lenguaje. Tal diagrama tiene la forma de un grafo dirigido con información adicional añadida, y se llama *diagrama de transición*. Los nodos del grafo se llaman *estados* y se usan para señalar, en ese momento, hasta qué lugar se ha analizado la cade-

na. Las aristas del grafo se etiquetan con caracteres del alfabeto y se llaman *transiciones*. Si el siguiente carácter a reconocer concuerda con la etiqueta de alguna transición que parta del estado actual, nos desplazamos al estado al que nos lleve la arista correspondiente. Naturalmente, nosotros debemos comenzar por un *estado inicial*, y cuando se hayan tratado todos los caracteres de la cadena correspondiente, necesitamos saber si la cadena es "legal". Para ello se marcan ciertos estados como *estados de aceptación* o *estados finales*. Si cuando ha sido tratada la cadena en su totalidad terminamos en un estado de aceptación, entonces la cadena es "legal". Marcaremos el estado inicial con una flecha (→) y alrededor de los estados de aceptación trazaremos un círculo.

Por ejemplo, el diagrama de la Figura 2.1 acepta todas las cadenas que están formadas por 0 ó más *aes* seguidas por una única *b*. Obsérvese que para toda cadena de la forma $a^k b$, para $k \geq 0$, el recorrido del diagrama termina en un estado de aceptación. El recorrido del mismo con cualquier otra cadena de *aes* y *bes* (incluida la cadena vacía) termina en cualquier otro estado, pero éste no es de aceptación.

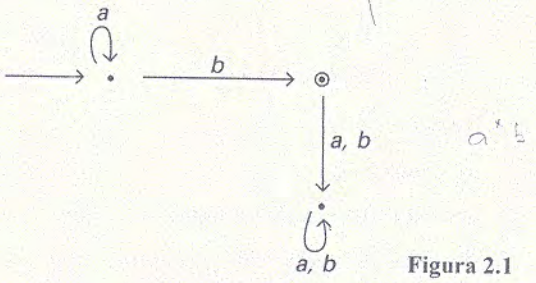


Figura 2.1

Considérese el lenguaje $A = \{(ab)^i \mid i \geq 1\}$, el cual está representado por la expresión regular $(ab)^+$. Obsérvese que una cadena de este lenguaje ha de tener al menos *una* copia de *ab*. Por tanto, si se construye un diagrama de transición para este lenguaje, el estado inicial no puede ser también estado de aceptación. Es más, si estando en el estado inicial se encuentra el carácter *b*, esto indica que la cadena no puede ser aceptada. Por tanto, hay dos transiciones a partir del estado inicial, una para *a* y otra para *b*. La transición correspondiente a , nos lleva a un estado en el que se espera encontrar como siguiente carácter de la cadena, una *b*. Si se obtiene *b*, entonces nos desplazaremos a un estado de aceptación. Luego, si no hay más caracteres a considerar, se habrá identificado una cadena legal. Si no se han agotado los caracteres de la cadena, tomaremos la transición apropiada que parta de dicho estado. El diagrama de transición para *A* es el que se muestra en la Figura 2.2.

Obsérvese que se tiene un único estado de aceptación. Si el análisis termina en cualquier otro estado, la cadena no está correctamente construida. Obsérvese,

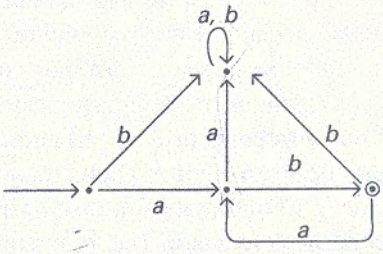


Figura 2.2

también, que una vez que se identifica un prefijo incorrecto, se realiza un desplazamiento a un estado que no es de aceptación y se permanece en el mismo.

Consideremos el lenguaje $(ab)^*$. En este caso si se acepta la cadena vacía. Por tanto, el estado inicial también es de aceptación. El diagrama de transición correspondiente se muestra en la Figura 2.3.

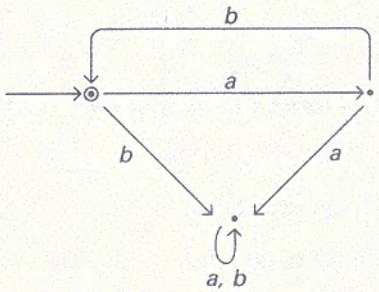


Figura 2.3

Vamos a etiquetar los estados del último diagrama de transición con las letras q_i, para $i = 0, 1, 2$. Obtendremos la Figura 2.4.

Podemos representar el diagrama de la Figura 2.4 por medio de una tabla que indica el siguiente estado al que desplazarse, desde un estado combinado con un símbolo de la entrada (Figura 2.5).

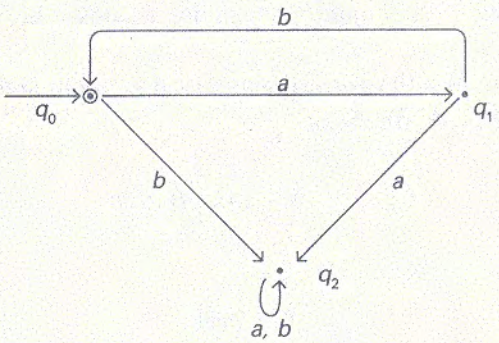


Figura 2.4

Obsérvese que la tabla para nuestro diagrama de transición tiene, para cada par estado actual-entrada, un único estado siguiente. Por tanto, para cada estado actual y símbolo de entrada, se puede determinar cuál será el estado siguiente. Se puede pensar que el diagrama representa la acción de alguna máquina. Esta máquina puede pasar por diferentes estados. El cambio de estado depende de la entrada y del estado en que se encuentre. Dicha máquina se llama *autómata finito*, una computadora ideal. El autómata finito se define en términos de sus estados, la entrada que acepta y su reacción ante la misma. Hay autómatas finitos de dos tipos, deterministas y no deterministas, dependiendo de cómo se defina la capacidad para cambiar de estado. El autómata que corresponde a la Figura 2.5 para $(ab)^*$ es determinista.

Estado\Entrada	a	b
q_0	q_1	q_2
q_1	q_2	q_0
q_2	q_2	q_2

Figura 2.5

Formalmente, un *autómata finito determinista* M es una colección de cinco elementos.

1. Un alfabeto de entrada Σ.
2. Una colección finita de estados Q.
3. Un estado inicial s.
4. Una colección F de estados finales o de aceptación.
5. Una función $\delta: Q \times \Sigma \rightarrow Q$ que determina el único estado siguiente para el par (q_i, σ) correspondiente al estado actual y la entrada.

Generalmente el término *autómata finito determinista* se abrevia como *AFD*. Usaremos $M = (Q, \Sigma, s, F, \delta)$ para indicar el conjunto de estados, el alfabeto, el estado inicial, el conjunto de estados finales y la función asociada con el AFD M.

Por ejemplo, el AFD correspondiente al ejemplo anterior se representa mediante $M = (Q, \Sigma, s, F, \delta)$, donde

$$Q = \{q_0, q_1, q_2\}$$
$$\Sigma = \{a, b\}$$
$$s = q_0$$
$$F = \{q_0\}$$

δ se define mediante la tabla de la Figura 2.6.

δ	a	b
q_0	q_1	q_2
q_1	q_2	q_0
q_2	q_2	q_2

Figura 2.6

La característica principal de un AFD es que δ es una *función*. Por tanto, δ se debe definir para todos los pares (q_i, σ) de $Q \times \Sigma$. Esto significa que sea cual sea el estado actual y el carácter de la entrada, *siempre* hay un estado siguiente y éste es único. Por tanto, para un par (q_i, σ) hay *uno y sólo un* valor de la función (estado siguiente), $\delta(q_i, \sigma)$. En otras palabras, el estado siguiente está totalmente determinado por la información que proporciona el par (q_i, σ).

Se puede crear un diagrama de transición a partir de la definición de un AFD. Primero, creamos y etiquetamos un nodo para cada estado. Entonces, para cada celda q_j de la fila correspondiente al estado q_i, trazamos una arista desde q_i a q_j, etiquetada con el carácter de entrada asociado a q_j. Finalmente, se marca el nodo s con una flecha, y se trazan unos círculos en todos los nodos de F para indicar cuáles son los estados de aceptación. Por tanto, el diagrama de transición para el AFD $M = \{Q, \Sigma, s, F, \delta\}$, donde

$$Q = \{q_0, q_1\}$$
$$\Sigma = \{a, b\}$$
$$F = \{q_0\}$$
$$s = q_0$$

y δ representada mediante la Figura 2.7

δ	a	b
q_0	q_0	q_1
q_1	q_1	q_0

Figura 2.7

se muestra en la Figura 2.8.

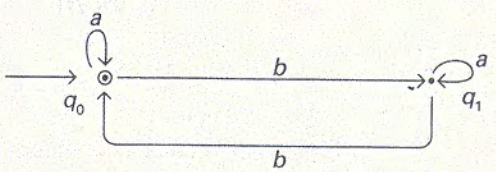


Figura 2.8

Consideremos otro ejemplo. El AFD $M = \{Q, \Sigma, s, F, \delta\}$ representado por

$$Q = \{q_0, q_1, q_2, q_3\}$$
$$\Sigma = \{a, b\}$$
$$(*)\quad s = q_0$$
$$F = \{q_0, q_1, q_2\}$$

y δ dada por la tabla de la Figura 2.9.

δ	a	b
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_0	q_3
q_3	q_3	q_3

Figura 2.9

El diagrama de transición correspondiente se muestra en la Figura 2.10.

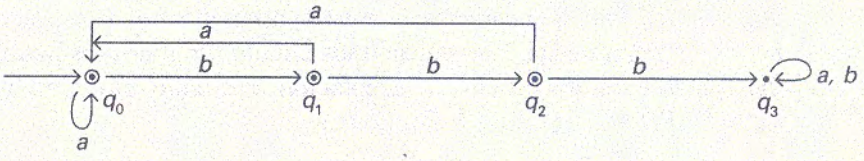


Figura 2.10

Ejercicios de la Sección 2.3

2.3.1. Obtener la expresión regular que representa al lenguaje formado por todas las cadenas sobre $\{a, b\}$ que tienen un número par de *bes*. Construir el diagrama de transición para este lenguaje.

2.3.2. Construir el diagrama de transición para el lenguaje dado por $c^*(a \cup bc^*)^*$. Convertir el diagrama en una tabla como la dada en la Figura 2.5, etiquetando los estados $q_0, q_1, \ldots$

2.3.3. Sea $M = \{Q, \Sigma, s, F, \delta\}$ dado por

$$Q = \{q_0, q_1, q_2, q_3\}$$
$$\Sigma = \{0, 1\}$$
$$F = \{q_0\}$$
$$s = q_0$$

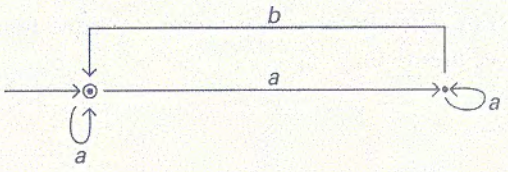
y δ dada por la tabla de la Figura 2.11.

δ	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Figura 2.11

Construir el diagrama de transición. Obtener la secuencia de estados por los que se pasa para aceptar la cadena 110101 (el carácter del extremo izquierdo es el primero en ser analizado).

2.3.4. ¿La Figura 2.12 es un diagrama de transición correspondiente a un AFD? ¿Por qué o por qué no?



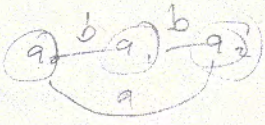


Figura 2.12

2.4 AFD Y LENGUAJES

Para trabajar con los AFD es necesario usar ciertas definiciones y notaciones. Si M es un AFD, entonces el *lenguaje aceptado por M* es

$$L(M) = \{w \in \Sigma^* \mid w \text{ es aceptada por } M\}$$

Por tanto, $L(M)$ es el conjunto de cadenas que hacen que M pase de su estado inicial a un estado de aceptación.

Por ejemplo, el lenguaje aceptado por el AFD (*), presentado en la última sección, es

$$L(M) = \{w \in \{a, b\}^* \mid w \text{ no contiene tres } bes \text{ consecutivas}\}$$

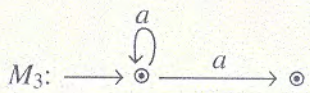
Merece la pena hacer hincapié en que $L(M)$ está formado por todas las cadenas aceptadas por M, y no que es un conjunto de cadenas que son todas aceptadas por M.

Para cada (q_i, σ) de $Q \times \Sigma$, $\delta(q_i, \sigma)$ es un estado perteneciente a Q, y él mismo puede ser emparejado con la entrada. Este par se transforma mediante δ en un nuevo estado de Q. En particular, si q_0 es el estado inicial de M y se tiene

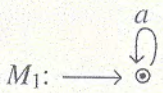
como entrada la cadena $\sigma_1\sigma_2\sigma_3$, el estado resultante se obtiene mediante la aplicación de $\delta(\delta(\delta(q_0, \sigma_1), \sigma_2), \sigma_3)$. Por ejemplo, para el AFD (*) de la última sección, se tiene $\delta(\delta(\delta(\delta(q_0, b), b), a), b) = q_1$ para la cadena *bbab*. Obsérvese la aplicación recursiva de M sobre la cadena. Adviértase, también, que escribir esta expresión es un proceso bastante laborioso. Nos pondremos de acuerdo en usar $\delta(q_0, bbab)$ para abreviar $\delta(\delta(\delta(\delta(q_0, b), b), a), b)$. Para ser más precisos, si $q_i \in Q$ y w es una cadena de la forma $a_i w'$ para algún $a_i \in \Sigma$ y una subcadena w', definiremos $\delta(q_i, w)$ como $\delta(\delta(q_i, a_i), w')$.

Diremos que dos AFD M_1 y M_2 son *equivalentes* si $L(M_1) = L(M_2)$. Por ejemplo, sean M_1 y M_2 sobre el alfabeto $\Sigma = \{a\}$, representados por los siguientes diagramas de transiciones

Ambos aceptan el lenguaje a^+ y, por tanto, son equivalentes. Por otro lado, sea M_3 dado por el siguiente diagrama



no es equivalente a M_1 o M_2 (¿por qué?). Obsérvese que M_4 dado por el diagrama de transición siguiente



es equivalente a M_3 y es más "sencillo" puesto que tiene menos estados. Los problemas del final del capítulo estudian las dificultades que existen para determinar si dos AFD son equivalentes y para transformar un AFD en otro equivalente que sea más sencillo.

Ejercicios de la Sección 2.4

2.4.1. Sea M un AFD. ¿Cuándo pertenecerá ε a $L(M)$?

2.4.2. Construir los AFD que aceptan cada uno de estos lenguajes sobre $\{a, b\}$:

(a) $\{w \mid$ toda a de w está entre dos $bes\}$

(b) $\{w \mid w$ contiene la subcadena $abab\}$

(c) $\{w \mid w$ no contiene ninguna de las subcadenas aa o $bb\}$

(d) $\{w \mid w$ tiene un número impar de aes y un número par de $bes\}$

(e) $\{w \mid w$ tiene ab y ba como subcadenas$\}$

2.4.3. Sea S el conjunto de todos los AFD sobre el alfabeto Σ. Sea $R \subseteq S \times S$ la relación definida de manera que: (M_1, M_2) esté en R si y sólo si M_1 es equivalente a M_2 (como autómatas finitos). Probar que R es una relación de equivalencia en S (y, por tanto, que la definición de equivalencia de AFD es consistente con el uso matemático habitual de los términos).

2.5 AUTÓMATA FINITO NO DETERMINISTA

Si se permite que desde un estado se realicen cero, una o más transiciones mediante el mismo símbolo de entrada, se dice que el autómata finito es *no determinista*. A veces es más conveniente diseñar autómatas finitos no deterministas (AFN) en lugar de deterministas. Consideremos el lenguaje $a^*b \cup ab^*$. Las cadenas pertenecientes a este lenguaje están formadas por algunas aes seguidas de una b o por una a seguida de varias bes. El AFD que acepta A se representa por medio del diagrama de transición de la Figura 2.13.

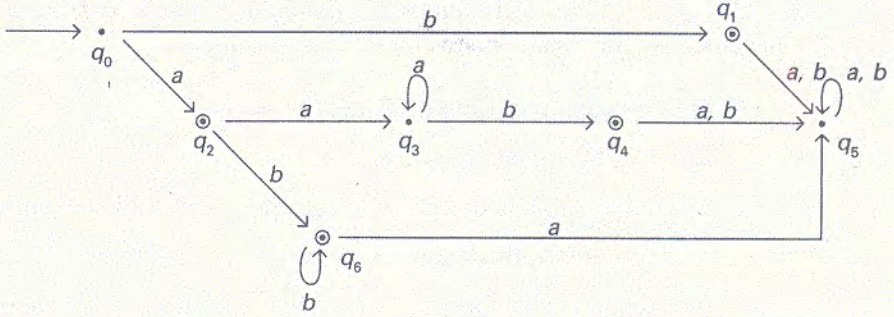


Figura 2.13

Aunque el lenguaje es relativamente sencillo, debemos detenernos en poder determinar si este diagrama de transición corresponde al AFD de A. Primero se debe comprobar que reconoce sólo las cadenas pertenecientes a A, y después, si representa a un AFD. Para ello, debemos comprobar que las reglas de transición constituyen una función, es decir, que de cada estado parte una y sólo una transición para cada símbolo del alfabeto.

Consideremos ahora el diagrama de transición de la Figura 2.14. Obsérvese que este diagrama acepta sólo las cadenas pertenecientes a A. Fíjese también que las reglas de transición no son una función de $Q \times \Sigma$ en Q porque no asigna un estado siguiente a los pares estado-entrada (q_4, a), (q_3, a), (q_3, b), (q_2, a) y (q_2, b). Es más, existe más de un estado siguiente correspondiente al par (q_0, a).

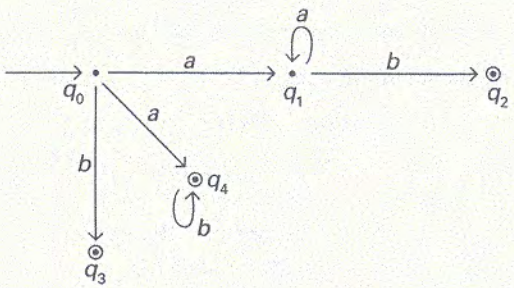


Figura 2.14

Este diagrama de transición representa a un AFN. Finalmente, obsérvese que en este diagrama es más fácil determinar qué lenguaje se acepta.

Si tratamos de definir el término autómata finito no determinista, veremos que la mayor parte de la definición se puede obtener a partir de la de AFD. Es decir, tendremos un conjunto finito de estados Q, un alfabeto de entrada Σ, un estado inicial o de partida s, un conjunto de estados de aceptación F y una regla de transición. La única diferencia que existe se encuentra en las reglas de transición. En un AFN, las reglas asocian pares (q, σ) con *cero o más estados*. Se puede decir que las reglas relacionan pares (q, σ) con colecciones o conjuntos de estados. Esto significa que la regla es una relación entre $Q \times \Sigma$ y Q, o sobre $(Q \times \Sigma) \times Q$. Por tanto, definiremos una *autómata finito no determinista* mediante una colección de cinco objetos $(Q, \Sigma, s, F, \Delta)$, donde

1. Q es un conjunto finito de estados.
2. Σ es el alfabeto de entrada.
3. s es uno de los estados de Q designado como estado de partida.
4. F es una colección de estados de aceptación o finales.
5. Δ es una relación sobre $(Q \times \Sigma) \times Q$ y se llama *relación de transición*.

Obsérvese que, puesto que Δ es una relación para todo par (q, σ) compuesto por el estado actual y el símbolo de la entrada, $\Delta(q, \sigma)$ es una colección de cero o más estados [es decir, $\Delta(q, \sigma) \subseteq Q$]. Esto indica que, para todo estado q, se pueden tener cero o más alternativas a elegir como estado siguiente, todas para el mismo símbolo de entrada.

Por ejemplo, el AFN descrito anteriormente para $A = a^*b \cup ab^*$ se representa por medio de

$$Q = \{q_0, q_1, q_2, q_3, q_4\}$$
$$F = \{q_2, q_3, q_4\}$$
$$s = q_0$$
$$\Sigma = \{a, b\}$$

y Δ dada por la tabla de la Figura 2.15.

Δ	a	b
q_0	$\{q_1, q_4\}$	$\{q_3\}$
q_1	$\{q_1\}$	$\{q_2\}$
q_2	Ø	Ø
q_3	Ø	Ø
q_4	Ø	$\{q_4\}$

Figura 2.15

Obsérvese que en la tabla de la relación de transición las celdas son conjuntos. El hecho de que existan celdas con Ø, indica que no existe ninguna transición desde el estado actual mediante la entrada correspondiente. Que para un par estado actual-entrada exista más de un posible estado siguiente indica que se puede elegir entre las distintas posibilidades. En el modelo no existe *nada* que *determine* la elección. Por esta razón, se dice que el comportamiento del autómata es *no determinista.*

Veamos otro ejemplo. Consideremos el AFN $M = (Q, \Sigma, s, F, \Delta)$ dado por

$$Q = \{q_0, q_1, q_2\}$$
$$\Sigma = \{a, b\}$$
$$s = q_0$$
$$F = \{q_0\}$$

y Δ dada por la tabla de la Figura 2.16. Este AFN tiene el correspondiente diagrama de transición que se muestra en la Figura 2.17.

Δ	a	b
q_0	$\{q_1\}$	Ø
q_1	Ø	$\{q_0, q_2\}$
q_2	$\{q_0\}$	Ø

Figura 2.16

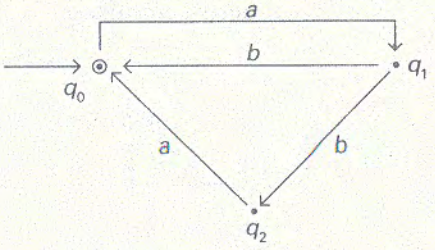


Figura 2.17

Este AFN acepta el lenguaje $(ab \cup aba)^*$. Obsérvese que cuando se está en el estado q_1, mediante el símbolo de entrada b, se puede pasar a dos posibles estados siguientes. Se puede elegir entre uno de estos estados. De nuevo, la elección de un estado no está determinada por el modelo. Si para el reconocimiento

de la cadena *aba*, se elige q_2 como estado siguiente desde el par (q_1, b) llegamos a un estado de aceptación; sin embargo, si elegimos q_0 no llegamos a un estado final. El análisis de cadenas mediante los AFN parece que implica ciertas conjeturas. Esta es una característica del no determinismo: cuando se debe realizar una elección y dicha elección no puede ser determinada por el modelo, debemos acertar la correcta. En un modelo de computación no determinista (del cual los AFN son una clase), asumimos que siempre se hace la elección correcta.

Como con los AFD, si M es un AFN, definimos el lenguaje aceptado por M por medio de

$$L(M) = \{w \mid w \text{ es una cadena aceptada por } M\}$$

donde una cadena w es aceptada por M, si M pasa de su estado inicial a un estado de aceptación o final al recorrer w (w es consumida en su totalidad).

Para determinar si una cadena pertenece a $L(M)$, se debe recorrer el diagrama de transición correspondiente a M. Debemos encontrar un camino que termine en un estado de aceptación cuando haya sido consumida toda la cadena. Durante el recorrido, debemos elegir de forma no determinista la transición de un estado a otro cuando existe más de una para el mismo símbolo. Para afirmar que la cadena no está en $L(M)$, debemos agotar todos las formas posibles de recorrer el diagrama de transición para dicha cadena. Para diagramas de transición sencillos, esto puede ocasionar un problema de gasto de tiempo. El siguiente ejemplo proporciona una forma de abordar este problema de una manera distinta, evitando la búsqueda exhaustiva por el diagrama.

Ejemplo 2.5.1

El diagrama de transición de la Figura 2.18 corresponde a un AFN que acepta el lenguaje

$$(a^*b^*)^* (aa \cup bb) (a^*b^*)^*$$

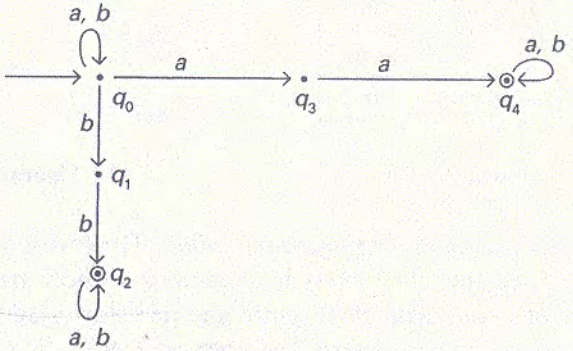


Figura 2.18

La tabla para Δ viene dada por la Figura 2.19

Δ	a	b
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$

Figura 2.19

La naturaleza recursiva del análisis de cadenas que vimos para los AFD se mantiene en los AFN si la notación se define con cuidado. Si $X \subseteq Q$, vamos a interpretar $\Delta(X, \sigma)$ como el conjunto de estados $\{p \mid q \in X \text{ y } p \in \Delta(q, \sigma)\}$. Por tanto, $\Delta(X, \sigma)$ es el conjunto de todos los estados siguientes a los que se puede llegar desde X con la entrada σ. Obsérvese que $\Delta(X, \sigma) = \bigcup_{q \in X} \Delta(q, \sigma)$ [recuérdese que $\Delta(q, \sigma)$ es un conjunto para cualquier estado q].

Por eso, en el ejemplo precedente,

$$\Delta(\{q_0, q_2, q_3\}, b) = \{q_0, q_1\} \cup \{q_2\} \cup \emptyset = \{q_0, q_1, q_2\}$$

Ahora obsérvese que para la cadena *abaab*, se obtiene que $\Delta(q_0, a) = \{q_0, q_3\}$, con lo que

$$\begin{aligned} \Delta(q_0, ab) &= \Delta(\Delta(q_0, a), b) \\ &= \Delta(\{q_0, q_3\}, b) \\ &= \{q_0, q_1\} \cup \emptyset \\ &= \{q_0, q_1\} \end{aligned}$$

Por tanto, podemos extender la notación usada para los AFD a los AFN y escribir $\Delta(q_0, abaab)$ para abreviar $\Delta(\Delta(\Delta(\Delta(\Delta(q_0, a), b), a), a), b)$.

En el Ejemplo 2.5.1, se tiene que $\Delta(q_0, abaab) = \{q_0, q_1, q_4\}$. La colección $\Delta(q_0, abaab)$ es el conjunto de todos los estados de M a los que se llega cuando se analiza la cadena *abaab*. Esta colección tienen en cuenta todos los posibles caminos o recorridos de M con esta cadena de entrada. $\Delta(q_0, abaab)$ contiene al menos un estado de aceptación, q_4, lo que indica que algún recorrido de este diagrama para la cadena *abaab* termina en un estado de aceptación. Por eso, *abaab* pertenece al lenguaje aceptado por este AFN.

Ejercicios de la Sección 2.5

2.5.1. Construir un AFD y el diagrama de transición asociado que acepte el lenguaje $(ab \cup aba)^*$. Compararlo con el AFN del Ejemplo 2.5.1.

2.5.2. Obtener un AFN (que no sea AFD) que acepte el lenguaje $ab^* \cup ab^*a$.

2.5.3. Usar la técnica precedente para determinar si las cadenas *babba* y *aabaaba* son aceptadas por el AFN del Ejemplo 2.5.1.

2.5.4. Sea M el AFN dado por $Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $s = q_0$, $F = \{q_1\}$ y Δ dada en la Figura 2.20. Determinar si a^2b, ba y b^2a están en $L(M)$. Dibujar el diagrama de transición para M.

Δ	a	b
q_0	$\{q_0, q_1\}$	$\{q_1\}$
q_1	$\emptyset$	$\{q_0, q_1\}$

Figura 2.20

2.6 EQUIVALENCIA DE AFN Y AFD

Hemos definido la equivalencia para los AFD. Extenderemos esta definición para la clase de todos los autómatas finitos (AFD y AFN) de forma que un autómata M es equivalente a un autómata M' si $L(M) = L(M')$.

Ejemplo 2.6.1

Los autómatas representados en la Figura 2.21 son equivalentes. Obsérvese que uno es determinista y el otro no determinista. Sin embargo, ambos aceptan el mismo lenguaje, $a(a \cup b)^*$.

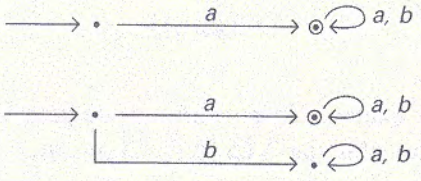


Figura 2.21

Ya que una función es un caso especial de relación (es decir, las funciones son relaciones que poseen requerimientos adicionales), las funciones de los AFD se consideran como relaciones en los AFN. En consecuencia, todo AFD es un AFN. La colección de lenguajes aceptados por los AFN incluye a todos los lenguajes aceptados por los AFD. De esto resulta que los AFN sólo aceptan los len-

guajes aceptados por los AFD. Por lo tanto, los AFN no son más potentes que los AFD con respecto a los lenguajes que aceptan. Para probar esto, necesitamos demostrar que todo lenguaje aceptado por un AFN también es aceptado por algún AFD.

Sea $M = (Q, \Sigma, s, F, \Delta)$ un AFN. En la sección anterior presentamos una forma de recorrer M, de la cual se obtenía la colección de todos los estados accesibles desde el estado inicial en cada una de las etapas de análisis de una cadena. Estas técnicas proporcionan la base para construir un AFD $M' = (Q', \Sigma', s', F', \delta)$ que acepte el mismo lenguaje que M. Esencialmente, lo que se pretende es hacer que cada estado de Q' se corresponda con un conjunto de estados de Q. Cuando se analiza una cadena con M, ésta se acepta cuando la colección final de estados contiene al menos un estado de aceptación perteneciente a F. Por tanto, haremos que F' sea el conjunto de estados de Q' que se correspondan con los conjuntos de estados (de Q) que contienen un estado de F. Haremos corresponder a s' con el conjunto $\{s\}$, $\Sigma' = \Sigma$, y definiremos δ de forma que nos desplacemos de un conjunto de estados de M a otro, como hace Δ.

Ejemplo 2.6.2

Consideremos el AFN M que acepta $a \cup (ab)^+$, representado por el diagrama de transición de la Figura 2.22. Para este AFN, tendremos

$$\Delta(q_0, a) = \{q_1, q_2\}$$
$$\Delta(q_0, b) = \emptyset$$
$$\Delta(\{q_1, q_2\}, a) = \emptyset$$
$$\Delta(\{q_1, q_2\}, b) = \{q_3\}$$
$$\Delta(\emptyset, a) = \Delta(\emptyset, b) = \emptyset$$
$$\Delta(q_3, a) = \{q_2\}$$
$$\Delta(q_3, b) = \emptyset$$
$$\Delta(q_2, a) = \emptyset$$
$$\Delta(q_2, b) = \{q_3\}$$

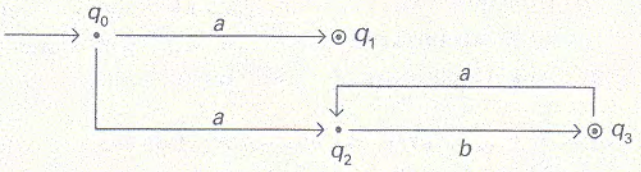


Figura 2.22

Por tanto, el diagrama de transición correspondiente al AFD M' que es equivalente a M viene dado en la Figura 2.23. Obsérvese que cada estado de M'

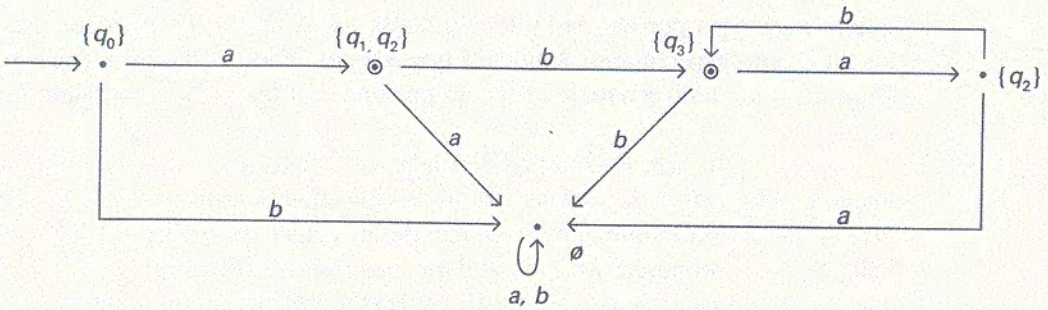


Figura 2.23

corresponde a un conjunto de estados de M. Los estados de aceptación de M' se corresponden con los conjuntos de estados de M que contienen estados de aceptación.

Se verifica fácilmente que la regla de transición es una función. Por tanto $M' = (Q', \Sigma', s', F', \delta)$ donde

$$Q' = \{\emptyset, \{q_0\}, \{q_2\}, \{q_3\}, \{q_1, q_2\}\}$$
$$\Sigma' = \Sigma$$
$$s' = \{q_0\}$$
$$F' = \{\{q_3\}, \{q_1, q_2\}\}$$

y δ viene dada por la tabla de la Figura 2.24.

δ	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_0\}$	$\{q_1, q_2\}$	$\emptyset$
$\{q_2\}$	$\emptyset$	$\{q_3\}$
$\{q_3\}$	$\{q_2\}$	$\emptyset$
$\{q_1, q_2\}$	$\emptyset$	$\{q_3\}$

Figura 2.24

Ahora demostraremos formalmente que todo lenguaje aceptado por un AFN es también aceptado por un AFD, con lo que probaremos que los lenguajes AFN y los lenguajes AFD están formados por la misma colección de lenguajes.

Teorema 2.6.1. Sea $M = (Q, \Sigma, s, F, \Delta)$ un AFN. Entonces existe un AFD $M' = (Q', \Sigma', s', F', \delta)$ que es equivalente a M.

Demostración. Definamos $M' = (Q', \Sigma', s', F', \delta)$ como sigue: sea $s' = \{s\}$, $\Sigma' = \Sigma$, $Q' = 2^Q$ (que es la colección de todos los subconjuntos de Q) y F' la colección de

todos los subconjuntos de Q' que contienen estados de F. Téngase en cuenta que hemos incluido en Q' y F' algún objeto más que en el ejemplo anterior. Sin embargo, esto no alterará la construcción de M'. Para cada conjunto $\{q_{i_1}, q_{i_2}, q_{i_n}\}$ de Q' y cada símbolo de entrada σ de Σ, definiremos δ como

$$\delta(\{q_{i_1}, q_{i_2}, ..., q_{i_n}\}, \sigma) = \{p_1, p_2, ..., p_k\}$$

si y sólo si

$$\Delta(\{q_{i_1}, q_{i_2}, ..., q_{i_n}\}, \sigma) = \{p_1, p_2, ..., p_k\}$$

Obsérvese que δ, definida de esta forma, es una función de $Q' \times \Sigma'$ en Q', puesto que está bien definida para todos los elementos de $Q' \times \Sigma'$.

Para probar que $L(M) = L(M')$, debemos demostrar que para toda cadena w, $\delta(s', w) = \{p_1, p_2, ..., p_j\}$ si y sólo si $\Delta(s, w) = \{p_1, p_2, ..., p_j\}$, con lo cual M' acepta w si y sólo si M acepta w. Probaremos esto por inducción sobre la longitud de w. Si la longitud de w es 0 (es decir, $w = \varepsilon$), entonces

$$\Delta(s, w) = \Delta(s, \varepsilon) = \{s\} = \delta(s', w)$$

Ahora supongamos que para toda cadena w de longitud menor o igual que m se tiene que $\Delta(s, w) = \delta(s', w)$. Supongamos que u es una cadena de longitud $m + 1$. Entonces, existirá algún $\sigma \in \Sigma$, de forma que se obtiene que $u = w\sigma$, donde w es una cadena de longitud m. En este caso, $\delta(s', w\sigma) = \delta(\delta(s', w), \sigma)$. Ahora, por la hipótesis de inducción, dado que w tiene longitud m, $\delta(s', w) = \{p_1, p_2, ..., p_j\}$ si y sólo si $\Delta(s, w) = \{p_1, p_2, ..., p_j\}$. Pero por la forma en la que hemos definido δ, tendremos que

$$\delta(\{p_1, p_2, ..., p_j\}, \sigma) = \{r_1, r_2, ..., r_k\}$$

si y sólo si

$$\Delta(\{p_1, p_2, ..., p_j\}, \sigma) = \{r_1, r_2, ..., r_k\}$$

Por lo que $\delta(s', w\sigma) = \{r_1, r_2, ..., r_k\}$ si y sólo si $\Delta(s, w\sigma) = \{r_1, r_2, ..., r_k\}$. Es decir, la igualdad se cumple para cadenas de longitud $m + 1$ si se cumple para cadenas de longitud m. Entonces por lo anterior tenemos que $\delta(s', w)$ es un estado de F' si y sólo si $\Delta(s, w)$ contiene algún estado de F. Por tanto, M' acepta w si y sólo si M acepta w. □

Obsérvese que, en la demostración, el AFD M' correspondiente al AFN M contiene muchos estados que no son accesibles desde el estado inicial. En la práctica, es una buena idea empezar con s' y añadir estados sólo cuando son el resultado de una transición desde un estado previamente añadido.

Ejercicios de la Sección 2.6

2.6.1. Construir el AFD correspondiente al AFN dado en la Figura 2.25. ¿Qué lenguaje es aceptado por dicho autómata?

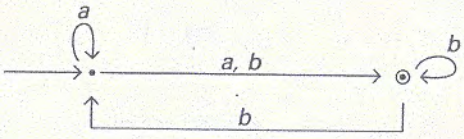


Figura 2.25

2.6.2. Encontrar un AFN que acepte $(ab \cup aab \cup aba)^*$. Convertir este AFN en un AFD.

2.6.3. Encontrar un AFN para $(a \cup b)^* aabab$. Convertirlo en un AFD.

2.6.4. Supongamos que M es un AFN que ya es determinista. ¿Qué se obtendrá si aplicamos a M la construcción dada en el Teorema 2.6.1?

2.7 ε-TRANSICIONES

Podemos ampliar la definición de autómata finito no determinista para incluir transiciones de un estado a otro que no dependan de ninguna entrada. Tales transiciones se llaman *ε-transiciones* porque al realizarse no consumen ningún símbolo de la entrada. Por ejemplo, los AFN de las Figuras 2.26 y 2.27 contienen ε-transiciones.

En el AFN de la Figura 2.26, el autómata puede cambiar su estado de q_0 a q_1 sin consumir nada en la entrada. Obsérvese que q_1 es el único estado de aceptación de este AFN. Si w es cualquier cadena de 0 o más *aes*, este autómata cicla sobre q_0 hasta que consume las *aes*. Una vez que la cadena se vacía, se desplaza a q_1 y lo acepta.

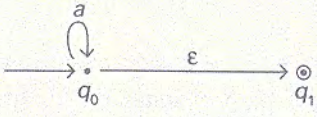


Figura 2.26

El AFN de la Figura 2.27 puede moverse del estado q_2 al estado q_0 sin consumir nada en la entrada. En ambos AFN, la decisión de elegir una ε-transición se realiza de la misma forma que la de cualquier otra transición con elección múltiple que exista en un AFN —basándose en algo que no determina el modelo. Por tanto, las ε-transiciones son consistentes con el matiz no determinista de nuestra versión previa de AFN.

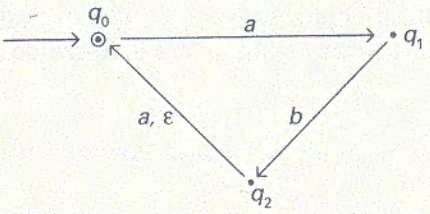


Figura 2.27

Si un AFN tiene ε-transiciones, la relación de transición Δ asocia pares de $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ con subconjuntos de Q. Es decir, Δ es una relación sobre $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$. Se puede añadir una columna en la tabla de Δ para colocar los pares de la forma (q_i, ε). Cuando hay ε-transiciones en un AFN es conveniente suponer que cada estado tiene una ε-transición que cicla en ese estado. Usaremos esto para sistematizar el cálculo de los AFN. Es decir, el AFN de la Figura 2.27 tendría la tabla de transición de la Figura 2.28.

Δ	a	b	ε
q_0	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\{q_2\}$	$\emptyset$
q_2	$\{q_0\}$	$\emptyset$	$\{q_0\}$

Figura 2.28

Para tratar de calcular el conjunto de los estados siguientes de un AFN que contiene ε-transiciones, debemos tener en cuenta las ε-transiciones "anteriores" y "posteriores" a la transición etiquetada con σ. Por ejemplo, consideremos el AFN M dado en la Figura 2.29.

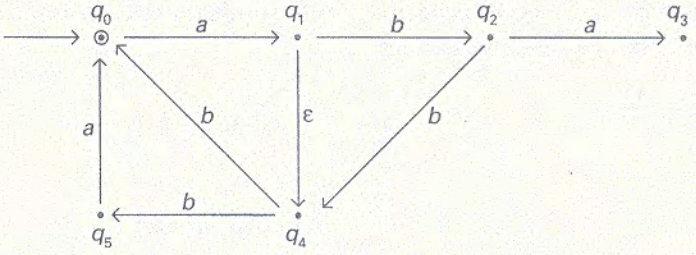


Figura 2.29

Como se ve en la Figura 2.29, el conjunto de estados siguientes al estado q_0 mediante la entrada a es el conjunto $\{q_1, q_4\}$ debido a la ε-transición que hay después de la transición con a. Igualmente, el conjunto de estados siguientes siendo q_1 el estado actual y b la entrada, es el conjunto $\{q_0, q_2, q_5\}$ debido a la ε-transición que existe antes de tomar la transición con b. Obsérvese que $\Delta(q_0, ababbb) = \{q_0, q_5\}$ con lo que $ababbb$ es aceptada por M.

Se puede sistematizar el proceso para calcular el conjunto de los estados siguientes en un AFN con ε-transiciones. Para todo estado $q \in Q$, definimos la ε-*cerradura de q* como

$$\varepsilon\text{–}c\,(q) = \{p \mid p \text{ es accesible desde } q \text{ sin consumir nada en la entrada}\}$$

Ampliaremos está definición para todo el conjunto de estados de la siguiente manera

$$\varepsilon\text{–}c\,(\{q_{i_1}, q_{i_2}, \ldots, q_{i_n}\}) = \bigcup_{k=1}^{n} \varepsilon\text{–}c\,(q_{i_k})$$

Por ejemplo, para el AFN de la Figura 2.30, se tiene que

$$\varepsilon\text{–}c\,(q_3) = \{q_3\}$$

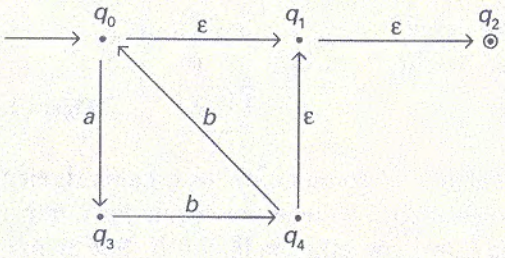


Figura 2.30

ya que cada estado es accesible desde sí mismo sin consumir ningún carácter de la entrada. También se obtiene

$$\varepsilon\text{–}c\,(q_0) = \{q_0, q_1, q_2\}$$

y

$$\varepsilon\text{–}c\,(q_4) = \{q_1, q_2, q_4\}$$

Para $q \in Q$ y $\sigma \in \Sigma$ se define

$$d\,(q, \sigma) = \{p \mid \text{ hay una transición de } q \text{ a } p \text{ etiquetada con } \sigma\}$$

La colección $d\,(q, \sigma)$ es la colección de estados que "siguen" directamente a q pasando por la transición etiquetada con σ. Ampliaremos la definición de d a los conjuntos como sigue

$$d(\{q_{i_1}, q_{i_2}, \ldots, q_{i_n}\}, \sigma) = \bigcup_{k=1}^{n} d(q_{i_k}, \sigma)$$

Por tanto, en el ejemplo precedente se tiene que

$$d(q_0, a) = \{q_3\}$$
$$d(q_0, b) = \emptyset$$
$$d(\{q_3, q_4\}, b) = \{q_0, q_4\}$$

Obsérvese que $\varepsilon\text{–}c\,(d\,(q, \sigma))$ es el conjunto de todos los estados accesibles desde q, primero mediante una transición con σ y después mediante una o más ε-transiciones. Por otro lado, $d\,(\varepsilon\text{–}c\,(q), \sigma)$ es el conjunto de todos los estados accesibles desde q tomando primero una o más ε-transiciones y después una transición con s.

Finalmente, obsérvese que $\varepsilon\text{–}c\,(d\,(\varepsilon\text{–}c\,(q), \sigma))$ es el conjunto de todos los estados accesibles desde q tomando primero una o más ε-transiciones, después una transición con σ y, por último, una o más ε-transiciones. Téngase en mente que permanecer en un estado es como tomar una ε-transición. Por tanto, $\varepsilon\text{–}c\,(d\,(\varepsilon\text{–}c\,(q), \sigma))$ es el conjunto de todos los estados siguientes al actual q mediante la entrada σ. Esto sistematiza el cálculo de conjuntos de estados siguientes. Primero se obtiene $\varepsilon\text{–}c\,(q)$, luego se calcula $d\,(\varepsilon\text{–}c\,(q), \sigma)$ y después se obtiene la ε-cerradura del conjunto de estados resultantes.

Ejemplo 2.7.1

Considérese el AFN con ε-transiciones dado en la Figura 2.31. Usando la fórmula anterior, obtendremos el conjunto de los estados siguientes al estado q por medio del símbolo de la entrada a:

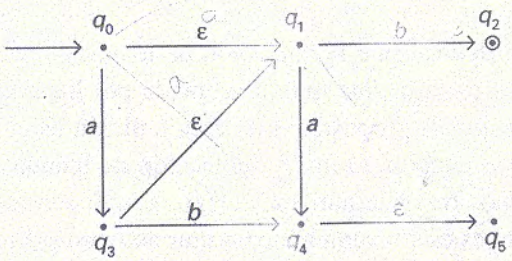


Figura 2.31

$$\varepsilon\text{–}c\,(q_0) = \{q_0, q_1\}$$
$$d\,(\varepsilon\text{–}c\,(q_0), a) = \{q_3, q_4\}$$
$$\varepsilon\text{–}c\,(\{q_3, q_4\}) = \{q_1, q_3, q_4, q_5\}$$

Así, mediante la entrada a, el conjunto de los estados siguientes es $\{q_1, q_3, q_4, q_5\}$. Es decir, $\Delta\,(q_0, a) = \{q_1, q_3, q_4, q_5\}$.

A partir de un AFN $M = (Q, \Sigma, s, F, \Delta)$ que tiene ε-transiciones, se puede construir un AFN sin ε-transiciones que acepte el mismo lenguaje. Se define $M' = (Q, \Sigma, s, F', \Delta')$ como

$$F' = F \cup \{q \mid \varepsilon\text{–}c\,(q) \cap F \neq \emptyset\}$$

y $\Delta'\,(q, \sigma) = \varepsilon\text{–}c\,(d\,(\varepsilon\text{–}c\,(q), \sigma))$, como antes.

Obsérvese que el autómata transformado M' no contiene ε-transiciones.

Ejemplo 2.7.2

El AFN de la Figura 2.31 se transforma en el AFN de la Figura 2.32, cuando todas las ε-transiciones son eliminadas mediante el proceso anterior.

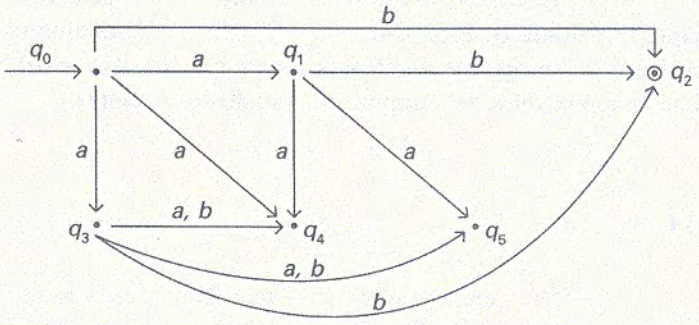


Figura 2.32

Entonces, se deduce que la colección de lenguajes aceptados por los AFN con ε-transiciones es la misma que la aceptada por los AFN sin ε-transiciones. Se ve fácilmente que la proposición inversa también es cierta. Por tanto, todos nuestros autómatas aceptan la misma colección de lenguajes. Para construir un autómata tendremos tres alternativas a elegir. Como veremos posteriormente, el uso de ε-transiciones es conveniente para unir autómatas finitos.

Ejercicios de la Sección 2.7

2.7.1. Calcular $\Delta(q_0, abb)$ y $\Delta(q_0, ab\alpha^2 b)$ para el AFN de la Figura 2.29.

2.7.2. Obtener ε–$c(\{q_1, q_4\})$ para el AFN de la Figura 2.30.

2.7.3. Obtener ε–$c(d(q_3, b))$ en el AFN de la Figura 2.30.

2.7.4. Usar la técnica estudiada para calcular $\Delta(q_3, b)$ en el Ejemplo 2.3.1.

2.7.5. Para el AFN dado en la Figura 2.33; (a) obtener la tabla de transición para Δ, (b) obtener la ε-cerradura (q_i) para $i = 0, 1, 2$, y (c) calcular $\Delta(q_0, a)$, $\Delta(q_0, b)$ y $\Delta(q_0, c)$ para la Figura 2.33.

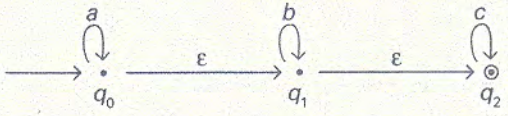


Figura 2.33

2.7.6. Para el AFN del Ejercicio 2.7.5, obtener el AFN que se obtiene al eliminar las ε-transiciones. Dar la tabla para Δ'.

2.8 AUTÓMATAS FINITOS Y EXPRESIONES REGULARES

Hasta ahora, hemos tratado de la relación entre autómata finito y expresiones regulares de una forma intuitiva. En esta sección, formalizaremos dicha relación por medio del teorema de Kleene (Teorema 2.8.4). De momento, vamos a ver algunas propiedades de los lenguajes aceptados por autómatas finitos.

Para un alfabeto Σ se pueden construir los AFN (y los AFD) que acepten palabras unitarias. Por ejemplo, el AFN de la Figura 2.34 acepta el lenguaje unitario $\{a\}$. Para ello se puede construir, incluso, una AFN que acepte el lenguaje vacío $\emptyset$. Dicho AFN se muestra en la Figura 2.35. Obsérvese que este autómata no acepta ninguna cadena.

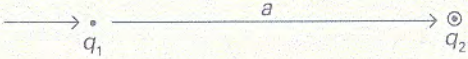


Figura 2.34

Figura 2.35

Supongamos que $M_1 = (Q_1, \Sigma_1, s_1, F_1, \Delta_1)$ y $M_2 = (Q_2, \Sigma_2, s_2, F_2, \Delta_2)$ son AFN. Podemos unir M_1 y M_2 en un nuevo AFN que acepte $L(M_1) \cup L(M_2)$, añadiendo un nuevo estado inicial s y dos ε-transiciones, una de s a s_1 y otra de s a s_2. La construcción formal de este nuevo AFN $M = (Q, \Sigma, s, F, \Delta)$ viene dado por $\Sigma = \Sigma_1 \cup \Sigma_2$, $F = F_1 \cup F_2$ y $Q = Q_1 \cup Q_2 \cup \{s\}$, donde s es el nuevo estado

inicial y Δ se define de forma que se incluyan todas las transiciones de Δ_1, Δ_2 y las dos nuevas ε-transiciones de s a s_1 y s_2. Conviene considerar las relaciones de transición Δ_1 y Δ_2 como colecciones de ternas ordenadas de $Q_1 \times \Sigma \times Q_1$ y $Q_2 \times \Sigma \times Q_2$, donde (q, σ, p) significa que existe una transición de q a p mediante el carácter σ (es decir, $p \in \Delta_i (q, \sigma)$). Usando esta notación se puede definir

$$\Delta = \Delta_1 \cup \Delta_2 \cup \{(s, \varepsilon, s_1), (s, \varepsilon, s_2)\}$$

Por ejemplo, los AFN de la Figura 2.36, los cuales aceptan ab^* y $(ab)^*$, respectivamente, se pueden unir formando el autómata con ε-transiciones de la Figura 2.37, el cual acepta $ab^* \cup (ab)^*$.

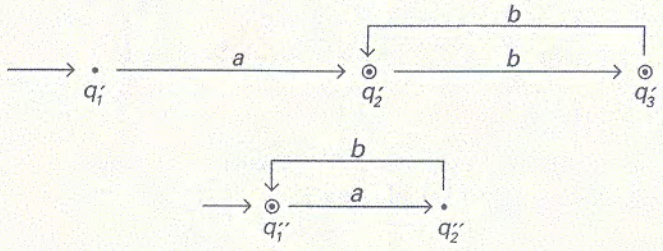


Figura 2.36

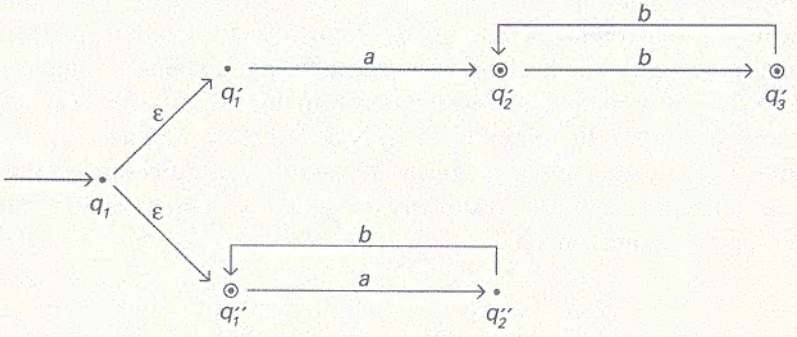


Figura 2.37

Sean $M_1 = (Q_1, \Sigma_1, s_1, F_1, \Delta_1)$ y $M_2 = (Q_2, \Sigma_2, s_2, F_2, \Delta_2)$ dos AFN. Podemos unirlos para formar un AFN que acepte $L(M_1)\, L(M_2)$. Se necesita un AFN que reconozca una cadena de $L(M_1)$ y después reconozca una de $L(M_2)$. Es decir, un recorrido hasta un estado de aceptación para admitir la cadena en su totalidad, primero debe pasar por un estado de aceptación de M_1 y después pasar (y terminar) en un estado de aceptación de M_2. Esto se realiza de forma no determinista

pasando del estado final de M_1 al estado inicial de M_2 por medio de una ε-transición.

Por ejemplo, los AFN de la Figura 2.38 aceptan los lenguajes $\{a\}$ y $\{b\}$, respectivamente. Uniéndolos como hemos dicho, se obtiene un AFN que acepta el lenguaje $\{ab\}$ (véase la Figura 2.39).

Figura 2.38

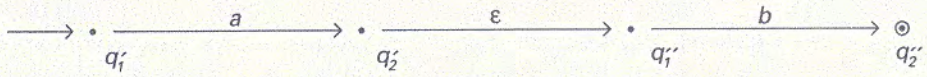


Figura 2.39

Obsérvese que el autómata que se obtiene tiene como estado inicial el estado inicial de M_1 y como estado(s) final(es) el(los) estado(s) final(es) de M_2. Por tanto, el AFN $M = (Q, \Sigma, s, F, \Delta)$ que acepta $L(M_1)\,L(M_2)$ viene dado por

$$
\begin{aligned}
Q &= Q_1 \cup Q_2 \\
s &= s_1 \\
F &= F_2 \\
\Delta &= \Delta_1 \cup \Delta_2 \cup (F_1 \times \{\varepsilon\} \times \{s_2\})
\end{aligned}
$$

La relación de transición Δ incluye todas las transiciones presentes en los dos AFN junto con todas las ternas de la forma (q, ε, s_2), donde q es un estado de aceptación de M_1. Es decir, $s_2 \in \Delta(q, \varepsilon)$ para todo $q \in F_1$.

Se puede deducir un procedimiento para construir una AFN que acepte $L(M)^*$ para el AFN $M = (Q, \Sigma, s, F, \Delta)$, como sigue. Primero, se añade un nuevo estado inicial s'; se hará que este estado sea además un estado de aceptación con el fin de que ε sea aceptada. Entonces, se permite una ε-transición desde s' a el antiguo estado inicial s. Por tanto, M comenzará una vez que M' se encuentre en s'. Se tendrá además, una ε-transición desde todos los estados de aceptación hasta el estado inicial s'. Una vez que la cadena de $L(M)$ ha sido agotada, el análisis puede continuar a partir del estado inicial de M o terminar en s'. El autómata resultante será $M' = (Q', \Sigma, s', F', \Delta')$, donde

$$Q' = Q \cup \{s'\}$$
$$F' = \{s'\}$$
$$\Delta' = \Delta \cup \{(s', \varepsilon, s)\} \cup (F \times \{\varepsilon\} \times \{s'\})$$

Obsérvese que en la definición de Δ' se incluyen las ε-transiciones necesarias además de las del AFN M original.

De la discusión anterior se obtiene el siguiente teorema.

Teorema 2.8.1. El conjunto de lenguajes aceptados por un autómata finito sobre el alfabeto Σ contiene $\emptyset$ y los lenguajes unitarios $\{a\}$ para todo $a \in \Sigma$. Este conjunto es cerrado con respecto a la unión, concatenación y la cerradura de estrella.

Dada una expresión regular r para construir un AFN (con ε-transiciones en todos los casos excepto para expresiones regulares triviales), podemos aplicar las técnicas precedentes a los términos de las expresiones regulares. Por tanto, todo lenguaje regular es aceptado por un autómata finito. Lo recíproco también es cierto, como veremos en el Lema 2.8.3. Es decir, todo lenguaje aceptado por una autómata finito es también un lenguaje regular. Por lo tanto, el conjunto de los lenguajes regulares es el mismo que el conjunto de los lenguajes aceptados por un autómata finito (Teorema 2.8.4).

Consideremos el autómata finito $M = (Q, \Sigma, s, F, \Delta)$ y supongamos que $s = q_0$ es el estado inicial. Para todo estado q_i, sea

$$A_i = \{w \in \Sigma^* \mid \Delta(q_1, w) \cap F \neq \emptyset\}$$

Es decir, A_i es el conjunto de las cadenas sobre Σ que hacen que M pase desde q_i hasta un estado de aceptación. Se dice que A_i es el conjunto de las cadenas *aceptadas por el estado* q_i. Obsérvese en que $A_0 = L(M)$. Adviértase, también, que es posible que $A_i = \emptyset$. Si $q_i \in F$, entonces se obtiene que $\varepsilon \in A_i$.

Como ejemplo, consideremos el autómata finito de la Figura 2.40. En el mismo, se tiene que

$$A_5 = \emptyset, \quad A_2 = \varepsilon$$
$$A_4 = \varepsilon, \quad A_1 = b$$
$$A_3 = a, \quad A_0 = ab \cup ba$$

Supongamos que $q_j \in \Delta(q_i, \sigma)$. Entonces A_i contiene a σA_j. De hecho, se tiene que

$$A_i = \cup \{\sigma A_j \mid q_j \in \Delta(q_i, \sigma)\}$$

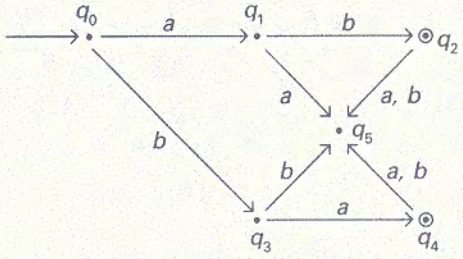


Figura 2.40

Esto proporciona las técnicas recursivas básicas para obtener una expresión regular a partir de un autómata finito. Como muestra, consideremos el ejemplo anterior. Obsérvese que

$$A_0 = aA_1 \cup bA_3, \qquad A_3 = aA_4 \cup bA_5$$
$$A_1 = bA_2 \cup aA_5, \qquad A_4 = \varepsilon \cup aA_5 \cup bA_5$$
$$A_2 = \varepsilon \cup aA_5 \cup bA_5, \qquad A_5 = \emptyset$$

Por tanto, se tiene un sistema de ecuaciones que se cumplen para $L(M)$. Se puede resolver por sustitución obteniendo que $L(M) = ab \cup ba$.

Considérese el autómata finito dado por la Figura 2.41. Del mismo modo se obtiene que

$$A_0 = aA_0 \cup bA_1$$
$$A_1 = \varepsilon$$

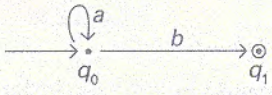


Figura 2.41

Resolviéndolo por sustitución, resulta que $A_0 = aA_0 \cup b$ y es imposible simplificarlo más. El siguiente lema muestra cómo resolver esto para obtener $A_0 = a^*b$ (que es lo que se espera tras inspeccionar el diagrama de transición).

Lema 2.8.2. (Lema de Arden) Una ecuación de la forma $X = AX \cup B$, donde $\varepsilon \notin A$, tiene una solución única $X = A^*B$.

Demostración. Obsérvese que $A^*B = (A^+ \cup \varepsilon)B = A^+B \cup B = A(A^*B) \cup B$. Por tanto, A^*B está contenida en toda solución. Supongamos que $X = A^*B \cup C$ es una solución, donde $C \cap A^*B = \emptyset$. Si se sustituye la expresión anterior en la ecuación $X = AX \cup B$, se obtiene

$$\begin{aligned} A^*B \cup C &= A\,(A^*B \cup C) \cup B \\ &= A^+B \cup AC \cup B \\ &= A^+B \cup B \cup AC \\ &= (A^+ \cup \varepsilon)\,B \cup AC \\ &= A^*B \cup AC \end{aligned}$$

Realizando en ambos lados la intersección con C se obtiene $C = AC \cap C$ (los otros términos son $\emptyset$). Por tanto, $C \subseteq AC$. Pero, como $\varepsilon \notin A$, la cadena más corta de AC debe ser más larga que la cadena más corta de C. Esto contradice que $C \subseteq AC$ a menos que $C = \emptyset$. Luego se debe tener que $C = \emptyset$ y, por tanto, A^*B es la única solución. □

Consideremos el autómata finito de la Figura 2.42. Aquí se tiene

$$\begin{aligned} A_0 &= aA_1 \\ A_1 &= aA_2 \cup bA_4 \\ A_2 &= aA_3 \cup bA_4 \\ A_3 &= \varepsilon \cup aA_3 \cup bA_4 \end{aligned}$$

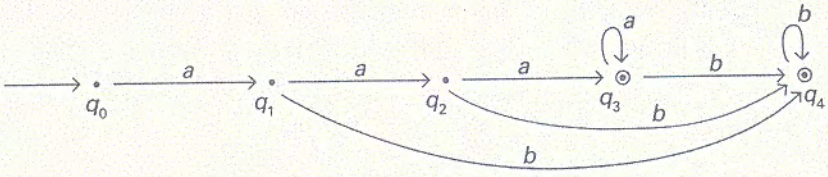


Figura 2.42

Sustituyendo y aplicando el lema de Arden cuando sea necesario, se obtiene

$$\begin{aligned} A_4 &= b^* \\ A_3 &= aA_3 \cup b^+ \cup \varepsilon \\ &= a^*b^* \\ A_2 &= a^+b^* \cup b^+ \\ A_1 &= a\,(a^+b^* \cup b^+) \cup b^+ \\ &= aa^+b^* \cup ab^+ \cup b^+ \\ A_0 &= aA_1 \\ &= a^2\,a^+b^* \cup a^2\,b^+ \cup ab^+ \end{aligned}$$

Entonces se deduce el siguiente lema.

Lema 2.8.3. Sea M un autómata finito. Entonces existe una expresión regular r para la cual $L(r) = L(M)$.

Del Lema 2.8.3 junto con las observaciones anteriores al Teorema 2.8.1, se obtiene el teorema de Kleene.

Teorema 2.8.4. (Kleene) Un lenguaje es regular si y sólo si es aceptado por un autómata finito.

Ejercicios de la Sección 2.8

2.8.1. Obtener un AFN que acepte ε.

2.8.2. Obtener un AFN que acepte $\{a\}$. Obtener otro AFN que acepte $\{b\}$. Usar las técnicas vistas en esta sección para unir estos AFN en uno que acepte el lenguaje $\{a, b\}$.

2.8.3. Obtener un AFN que acepte $(a \cup b)^* \cup (aba)^+$.

2.8.4. Obtener un AFN que acepte todas las cadenas de la forma *bowwow*, *bowwowwow*, *bowwowwowwowwow*, ... Conseguir un AFN que acepte todas las cadenas de la forma *ohmy*, *ohmyohmy*, *ohmyohmyohmy*, ... Unir los dos AFN para que se acepte la unión de los dos lenguajes. Téngase en cuenta que los símbolos de un alfabeto no tienen por qué ser caracteres de longitud uno.

2.8.5. Sea M_1 dado por la Figura 2.43 y M_2 dado en la Figura 2.44. Obtener un AFN que acepte $L(M_1)\,L(M_2)$. Obtener un AFN que acepte $L(M_2)\,L(M_1)$.

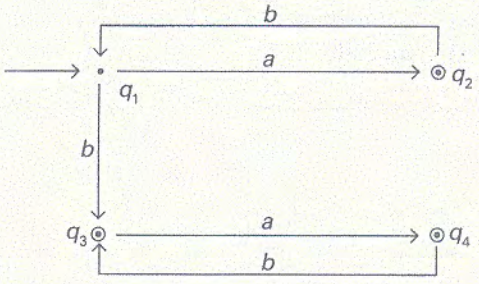


Figura 2.43

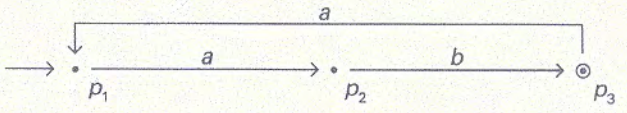


Figura 2.44

2.8.6. Sean $M_1 = (\{q_1, q_2, q_3\}, \{a, b\}, \{q_1\}, \{q_1\}, \Delta_1)$ y $M_2 = (\{p_1, p_2, p_3, p_4\}, \{0, 1\}, \{p_1\}, \{p_1, p_2\}, \Delta_2)$, donde Δ_1 y Δ_2 vienen dados en las tablas de la Figura 2.45. Obtener un AFN que acepte $L(M_1)\,L(M_2)$. Obtener un AFN que

acepte $L(M_2)\,L(M_1)\,L(M_1)$. Obtener finalmente, un AFN que acepte $(L(M_1))^2 \cup L(M_1)$.

Δ	a	b
q_1	$\{q_2, q_3\}$	$\emptyset$
q_2	$\emptyset$	$\{q_1\}$
q_3	$\{q_3\}$	$\{q_3\}$

Δ	0	1
p_1	$\{p_2\}$	$\emptyset$
p_2	$\emptyset$	$\{p_3, p_4\}$
p_3	$\{p_2\}$	$\emptyset$
p_4	$\{p_3\}$	$\emptyset$

Figura 2.45

2.8.7. Obtener un AFN para $(ab)^*$ a partir de los AFN que aceptan $\{a\}$ y $\{b\}$.

2.8.8. Obtener un AFN para $(aa \cup b)^*\,(bb \cup a)^*$ a partir de los AFN que aceptan $\{a\}$ y $\{b\}$.

2.8.9. Obtener un AFN para

$$((a \cup b)\,(a \cup b))^* \cup ((a \cup b)\,(a \cup b)\,(a \cup b))^*$$

a partir de los AFN para $\{a\}$ y $\{b\}$.

2.8.10. Si $M = (Q, \Sigma, s, F, \delta)$ es un autómata finito determinista, entonces el complemento de $L(M)$ [es decir, $\Sigma^* - L(M)$] es aceptado por el autómata $M = (Q, \Sigma, s, Q - F, \delta)$. ¿$M'$ es un AFD o un AFN? Obtener un AFD que acepte ab^*ab. Obtener un autómata finito que acepte $\{a, b\}^* - ab^*ab$.

2.8.11. Demostrar que $A_i = \bigcup_\sigma \{\sigma A_j \mid q_j \in \Delta(q_i, \sigma)\}$.

2.8.12. Obtener una expresión regular para el lenguaje aceptado por el autómata finito de la Figura 2.46.

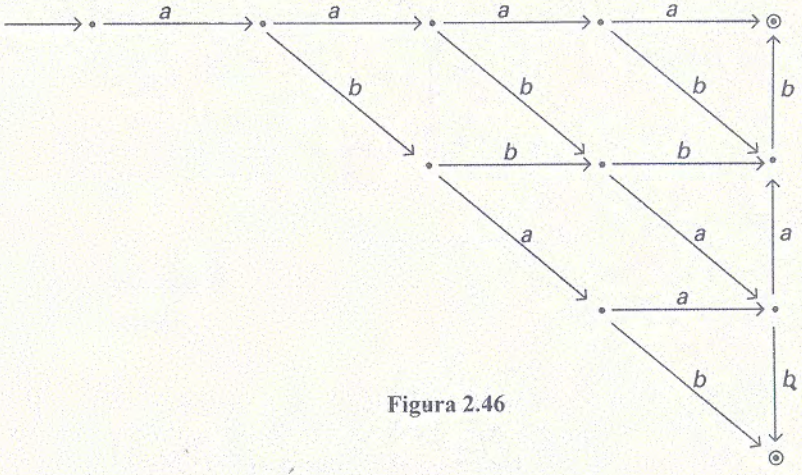


Figura 2.46

2.8.13. Obtener una expresión regular para el AFD de la Figura 2.47.

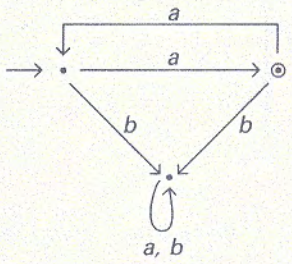
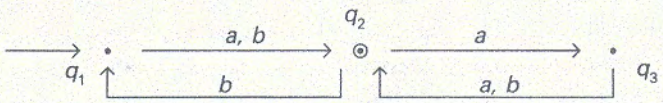


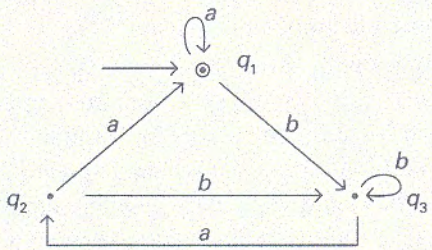
Figura 2.47

2.8.14. Obtener una expresión regular para los lenguajes aceptados por cada uno de los autómatas de la Figura 2.48.

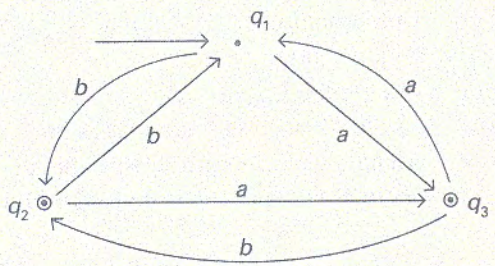
a.)



b.)



c.)



d.)

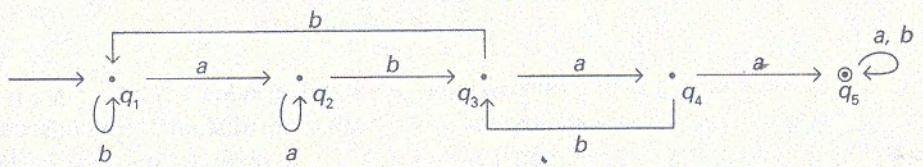


Figura 2.48

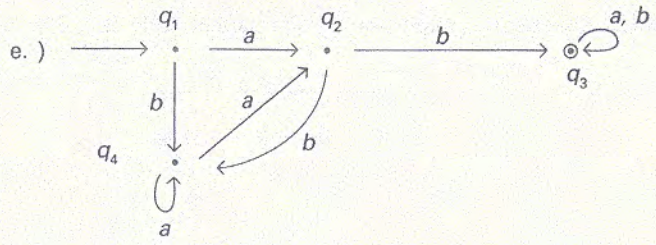


Figura 2.48 *(continuación)*

2.9 PROPIEDADES DE LOS LENGUAJES REGULARES

Los resultados de la última sección establecen la conexión entre autómata finito y expresión regular. Todo lo que es verdad para lenguajes regulares también es verdad para lenguajes aceptados por un autómata finito y viceversa. Así, por ejemplo, la colección de lenguajes regulares es cerrada con respecto a la concatenación, unión y cerradura de estrella porque los lenguajes aceptados por un autómata finito también lo son (Teorema 2.8.1).

Es importante preguntarse si, dado un lenguaje L, ¿L es regular? Desde luego, si L es finito, es regular y se podrá construir un autómata finito o una expresión regular para ellos de forma sencilla. También, si L es especificado ya sea por medio de un autómata finito o por una expresión regular, la respuesta es obvia. Por desgracia, hay relativamente pocos lenguajes que sean regulares y, en el caso de un lenguaje infinito, la búsqueda exhaustiva de una expresión regular o un autómata finito puede resultar inútil. En este caso, se necesita obtener algunas propiedades que compartan todos los lenguajes regulares infinitos y que no estén presentes en los lenguajes no regulares.

Supongamos que un lenguaje es regular y que, por tanto, es aceptado por un AFD $M = (Q, \Sigma, s, F, \delta)$, donde Q contiene n estados. Si $L(M)$ es infinito, podremos encontrar cadenas cuya longitud sea mayor que n. Supongamos que $w = a_1\, a_2 \ldots a_{n+1}$ es una de las cadenas de longitud $n + 1$ que pertenece a $L(M)$. Si tuviéramos

$$q_1 = \delta(s, a_1)$$
$$q_2 = \delta(q_1, a_2)$$

y así sucesivamente, obtendríamos los $n + 1$ estados, $q_1, q_2, \ldots, q_{n+1}$. Puesto que Q contiene sólo n estados, los q_i no serán todos distintos. En consecuencia, para algunos índices j y k, con $1 \leq j < k \leq n + 1$, se obtendrá que $q_j = q_k$. Por lo tanto, tendremos un ciclo en el camino que parte de s hasta un estado de aceptación, según se muestra en la Figura 2.49.

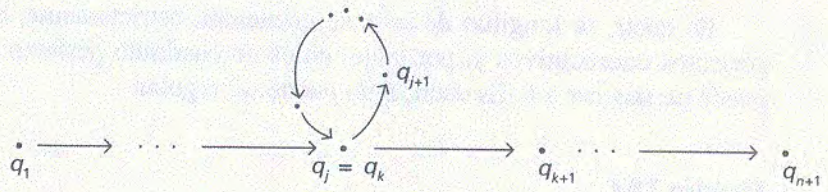


Figura 2.49

Puesto que $j < k$, se tiene que la "parte central", es decir, $a_{j+1} \ldots a_k$, tiene al menos longitud 1. Obsérvese además que la cadena $w' = a_1 \ldots a_j\, a_{k+1} \ldots a_{n+1}$ debe pertenecer también a $L(M)$. Por esto, se puede dar vueltas en el ciclo tantas veces como se quiera, de forma que $a_1 \ldots a_j\, (a_{j+1} \ldots a_k)^m\, a_{k+1} \ldots a_{n+1}$ estará en $L(M)$ para todo $m \geq 0$. Es decir, se puede "bombear" cero o más veces la parte central y seguir teniendo una cadena que sea aceptada por el autómata. Formalizaremos esto en el siguiente lema, conocido como *lema de bombeo*.

Lema 2.9.1. Sea L un lenguaje regular infinito. Entonces hay una constante n de forma que, si w es un cadena de L cuya longitud es mayor o igual que n, se tiene que $w = uvx$, siendo $uv^i x \in L$ para todo $i \geq 0$, con $|v| \geq 1$ y $|uv| \leq n$.

El lema de bombeo presenta una propiedad que debe tener todo lenguaje regular y nos facilita una forma de determinar si un lenguaje no es regular. Para demostrar que un lenguaje no es regular, se mostrará que, para cualquier valor n lo bastante grande, se tendrá al menos una cadena de longitud n o mayor que falle al ser "bombeada".

Por ejemplo, considérese el lenguaje

$$L = \{a^{i^2} \mid i \geq 1\}$$

Toda cadena de L debe tener una longitud que sea un cuadrado perfecto. Supongamos que L es regular y sea n la constante dada en el lema de bombeo. Obsérvese que $a^{n^2} \in L$ y que, por el lema de bombeo, tenemos $a^{n^2} = uvx$ de forma que $1 \leq |v| \leq n$ y $uv^i x \in L$ para todo $i \geq 1$. Entonces se obtiene que

$$\begin{aligned} n^2 &= |uvx| \\ &< |uv^2x| \\ &\leq n^2 + n \\ &< (n+1)^2 \end{aligned}$$

Es decir, la longitud de uv^2x se encuentra, estrictamente, entre cuadrados perfectos consecutivos y, por tanto, no es un cuadrado perfecto. Luego uv^2x no puede pertenecer a L. Es decir, L no puede ser regular.

Ejemplo 2.9.1

Consideraremos otro ejemplo. Sea el lenguaje $L = \{a^m b^m \mid m \geq 0\}$. Se ve claramente que L es infinito y, si L es regular, podremos aplicar el lema de bombeo. Sea n la constante del lema y consideremos $a^n b^n$, cuya longitud es mayor que n. Tendremos por tanto, que $a^n b^n = uvx$ para las cadenas u, v y x con $|v| \geq 1$ y $|uv| \leq n$. Nos centraremos en v, teniendo en cuenta que $|uv| \leq n$ fuerza a que v esté formado sólo por *aes*. Supongamos que $v = a^s$ para $s \geq 1$. Entonces, si $u = a^r$, se tiene que $x = a^{n-(r+s)}b^n$. De lo que se deduce que $uv^2x = a^r a^{2s} a^{n-(r+s)} b^n = a^{n+s} b^n$. Dado que $s \geq 1$, la cadena no puede pertenecer a L. Por tanto, L no puede ser regular ya que no satisface el lema de bombeo.

Que $\{a^n b^n \mid n \geq 0\}$ no sea regular y, por tanto, no sea aceptado por un autómata finito saca a la luz las propiedades comunes a todos los lenguajes regulares. Durante el análisis de una cadena por medio de un autómata finito, sólo tenemos a nuestra disposición, en cada paso, el estado y el símbolo actual. Cuando analizamos las *bes* no tenemos información sobre cuántas *aes* han sido analizadas.

Otra forma de decirlo es que la cantidad de memoria necesaria para aceptar o rechazar una cadena debe ser limitada. Si consideramos los estados como memoria, el hecho de que el conjunto de estados sea finito provoca dichas limitaciones. Podríamos construir un autómata de estados *no finito*, que aceptara dicho lenguaje. Podría estar formado por un estado inicial que también fuera estado de aceptación, con un camino para cada $a^n b^n$, para todo $n > 0$. Véase la Figura 2.50. Obsérvese que hay un número infinito de estados, por lo que la memoria en esta clase de autómatas no está limitada.

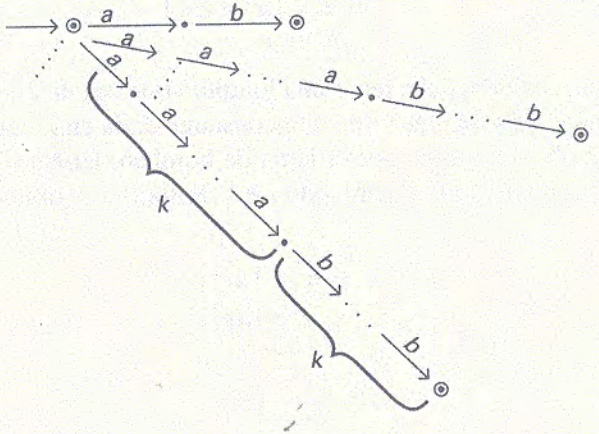


Figura 2.50

Además de ser una herramienta que determina si un lenguaje es regular, el lema de bombeo proporciona los medios necesarios para determinar si un autómata finito acepta cualquier lenguaje *no vacío* y si el lenguaje aceptado es finito o infinito.

Teorema 2.9.2. Sea M un autómata finito con k estados.

1. $L(M) \neq \emptyset$ si y sólo si M acepta una cadena de longitud menor que k.
2. $L(M)$ es infinito si y sólo si M acepta una cadena de longitud n, donde $k \leq n < 2k$.

Demostración. **1.** Si M acepta una cadena de longitud menor que k, entonces $L(M) \neq \emptyset$. A la inversa, supongamos que $L(M) \neq \emptyset$. Entonces existirá algún $w \in L(M)$. Necesitamos probar que $L(M)$ contiene una cadena de longitud menor que k. Si $|w| < k$, quedará probado. Sin embargo, supongamos que $|w| \geq k$. Gracias a lo expuesto con anterioridad al Lema 2.9.1, sabemos que debería haber un ciclo en el camino que se recorre para aceptar w y por tanto, se puede poner que $w = uvx$ para algunas cadenas u, v y x, donde u es la parte anterior al ciclo, v es la parte del ciclo y x es posterior al ciclo. Por tanto, $|v| \geq 1$ y se tiene que $uv^i x \in L(M)$ para todo $i \geq 0$. En particular, $ux \in L(M)$. Si $|ux| < k$, quedaría probado. Apliquemos ahora el proceso anterior a ux. Obsérvese que $|ux| < |uvx|$, de modo que de aplicaciones repetidas del proceso precedente, eliminando cada vez la subpalabra "central", siempre que la cadena siga perteneciendo a $L(M)$, podrá obtenerse una cadena de $L(M)$ cuya longitud será menor que k.

2. Supongamos que $w \in L(M)$ con $k \leq |w| < 2k$. En lo visto antes del Lema 2.9.1, se estableció que M debe tener un ciclo en el camino que acepta w y, por tanto, $w = uvx$ para algunas cadenas u, v, x con $|v| \geq 1$. En consecuencia, $uv^i x \in L(M)$ para todo i, con lo que $L(M)$ es infinito. Al contrario, supongamos que $L(M)$ es infinito. Entonces no todas las cadenas pueden tener longitud menor que k, y por tanto existirá alguna cadena $w \in L(M)$ cuya longitud sea al menos k. Si $|w| < 2k$, quedaría probado. Si no, aplicando a w la construcción anterior al Lema 2.9.1, se obtiene $w = uvx$, donde $|v| \geq 1$ y $|uv| \leq k$, con lo que $|v| \leq k$. Ahora tenemos que $|w| \geq 2k$ y que $|v| \leq k$, lo que produce que $|ux| \geq k$. Como en la parte 1, si $|ux| < 2k$, quedaría probado. De lo contrario, aplicaremos repetidamente este proceso a ux hasta encontrar una cadena cuya longitud se encuentre entre k y $2k - 1$. □

El Teorema 2.9.2 nos proporciona un procedimiento para decidir si $L(M)$ es vacío y si $L(M)$ es finito o infinito. Dado que los alfabetos son colecciones finitas, podemos realizar dichos procedimientos en tiempo finito, con lo que pode-

mos afirmar que los mismos nos facilitan algoritmos para resolver dichos problemas. Sin embargo, dichos algoritmos no son particularmente eficientes. Para los AFD hay una forma más rápida de poderlo realizar, eliminando los estados que, para cualquier entrada, no sean alcanzables desde el estado inicial. $L(M)$ no será vacío si queda algún estado final. Entonces, si se eliminan todos los estados no finales desde los cuales no pueda ser alcanzado ningún estado no final y comprobamos los ciclos, se puede determinar si $L(M)$ es finito o infinito.

Los problemas del final del capítulo tratan otro problema de decisión, la equivalencia de lenguajes regulares.

Una vez que hemos visto algunos ejemplos de lenguajes no regulares, hay otras técnicas para comprobar la regularidad además de usar el lema de bombeo.

Supongamos que L y K son lenguajes sobre Σ. De las leyes de De Morgan para conjuntos se obtiene que

$$(\Sigma^* - L) \cup (\Sigma^* - K) = \Sigma^* - (L \cap K)$$

Por tanto, se tiene que

$$\begin{aligned} L \cap K &= \Sigma^* - (\Sigma^* - (L \cap K)) \\ &= \Sigma^* - ((\Sigma^* - L) \cup (\Sigma^* - K)) \end{aligned}$$

Entonces, por el Ejercicio 2.8.10, sabemos que, si L y K son aceptados por un AFD, entonces $\Sigma^* - L$ y $\Sigma^* - K$ también lo son. Por tanto, si L y K son regulares; entonces $\Sigma^* - L$ y $\Sigma^* - K$ también lo son. También sabemos que la unión de lenguajes regulares es regular, así que $(\Sigma^* - L) \cup (\Sigma^* - K)$ es regular y su complemento $\Sigma^* - ((\Sigma^* - L) \cup (\Sigma^* - K))$ también es regular. Por tanto, $L \cap K$ es regular cuando L y K lo son.

La intersección es una técnica muy usada para determinar la regularidad de los lenguajes. Por ejemplo, sea $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Entonces, el lenguaje de los enteros no negativos viene dado por $L_1 = 0 \cup \{1, 2, ..., 9\}\,\Sigma^*$. El lenguaje de todas las cadenas de dígitos terminadas en 0, 2, 4, 6, u 8 viene dado por $L_2 = \Sigma^*\{0, 2, 4, 6, 8\}$. Obsérvese que ambos son regulares. El lenguaje de todos los enteros no negativos divisibles por 2, vendrá dado por $L = L_1 \cap L_2$ y también será regular.

Ejemplo 2.9.2

Sea $\Sigma = \{a, b\}$. Usaremos la intersección para probar que el lenguaje $L = \{ww^I \mid w \in \Sigma^*\}$ no es regular. Primero, fíjese que por el lema de bombeo $L_1 = \{a^n b^{2k} a^n \mid n, k \geq 0\}$ no es regular. Segundo, $L_2 = \{a^k b^n a^m \mid k, n, m \geq 0\}$ es regular (se denota mediante la expresión regular $a^* b^* a^*$). Finalmente, obsérvese

que $L_2 \cap L = L_1$. Si L fuera regular, entonces L_1 sería regular. Por tanto, L no puede ser regular.

Ejercicios de la Sección 2.9

2.9.1. Probar que $\{a^p \mid p \text{ es primo}\}$ no es un lenguaje regular.

2.9.2. Probar que $\{a^n b a^m b a^{m+n} \mid n, m \geq 1\}$ no es un lenguaje regular.

2.9.3. Determinar si los siguientes lenguajes son regulares y decir o probar por qué si o por qué no.

(a) $\{a^i b^{2i} \mid i \geq 1\}$

(b) $\{(ab)^i \mid i \geq 1\}$

(c) $\{a^{2n} \mid n \geq 1\}$

(d) $\{a^n b^m a^{n+m} \mid n, m \geq 1\}$

(e) $\{a^{2^n} \mid n \geq 0\}$

(f) $\{w \mid w = w^i \text{ para } w \in \{a, b\}^*\}$

(g) $\{wxw^i \mid w, r \in \{a, b\}^+\}$

2.9.4. Usar el procedimiento del Teorema 2.9.2 para decidir si $L(M)$ es finito o infinito según el autómata de la Figura 2.51.

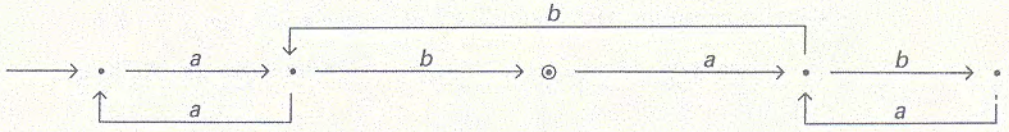


Figura 2.51

2.9.5. Usar las afirmaciones posteriores al Teorema 2.9.2 para determinar si el AFD de la Figura 2.52 acepta un lenguaje no vacío. Si el lenguaje no es vacío, determinar si el lenguaje aceptado es finito o infinito.

2.9.6. Sea $\Sigma = \{a, b\}$.

(a) Construir los AFD que acepten a^*b y ab^*.

(b) A partir de los AFD de la parte (a), construir los AFD que acepten $\Sigma^* - a^*b$ y $\Sigma^* - ab^*$.

(c) A partir de los AFD de la parte (b), construir un AFD que acepte la unión $(\Sigma^* - a^*b) \cup (\Sigma^* - ab^*)$.

(d) Usar el resultado de la parte (c) para construir un AFD que acepte el lenguaje $a^*b \cap ab^*$.

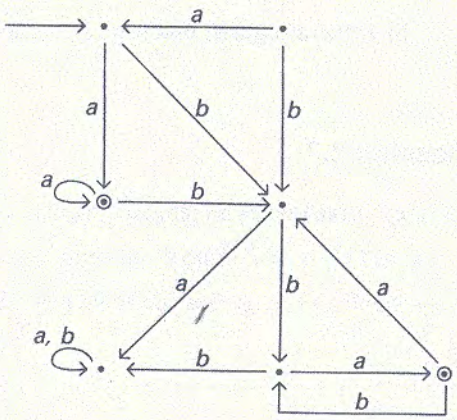


Figura 2.52

2.9.7. La construcción del Ejercicio 2.9.6, aunque efectiva, no es particularmente eficiente. En la parte (c) se obtiene un AFN a partir de la unión y después se transforma en un AFD para la intersección de lenguajes regulares sobre el mismo alfabeto haciendo uso del producto cartesiano. Sean $M_1 = (Q_1, \Sigma, s_1, F_1, \delta_1)$ y $M_2 = (Q_2, \Sigma, s_2, F_2, \delta_2)$ dos AFD. Definir $M = (Q_1 \times Q_2, \Sigma, (s_1, s_2), F_1 \times F_2, \delta)$, donde (s_1, s_2) denota los pares ordenados de estados. La función de transición δ se define para todos los pares ordenados $(q_i, p_j) \in Q_1 \times Q_2$ y para todo $\sigma \in \Sigma$, por medio de

$$\delta((q_i, q_j), \sigma) = (\delta_1(q_i, \sigma), \delta_2(p_j, \sigma))$$

(a) Construir los AFD correspondientes a a^*b y ab^* sobre $\Sigma = \{a, b\}$.

(b) Usar esta técnica para construir directamente un AFD que acepte $a^*b \cap ab^*$.

2.9.8. Usar el lema de bombeo para probar que no es regular el lenguaje L_1 dado en el Ejemplo 2.9.2.

2.9.9. Probar que $\{ww \mid w \in \{a, b\}^*\}$ no es regular.

2.10 APLICACIONES DE LAS EXPRESIONES REGULARES Y LOS AUTÓMATAS FINITOS

Los autómatas finitos se usan frecuentemente en los problemas que implican el análisis de cadenas de caracteres. Tales problemas incluyen problemas de búsqueda e identificación, tales como la búsqueda de la existencia de una cadena en un fichero o el reconocimiento de cadenas de entrada que satisfagan ciertos criterios. Un autómata finito es, él mismo, un modelo de un procedimiento para reconocimiento de cadenas por medio de la expresión regular asociada. Por tanto, en la búsqueda de una cadena en un fichero, podemos aplicar el autómata finito

de forma sistemática a las cadenas del fichero hasta que se acepta la cadena o se termina el fichero.

Un problema común en la programación de computadoras es el de tener la seguridad de que los datos de entrada de un programa son correctos. Por ejemplo, si se espera un entero sin signo como dato de entrada y el usuario confunde uno de los dígitos con un carácter no numérico, se puede dar todo tipo de resultados impropios, desde una terminación anormal hasta el cálculo de resultados incorrectos (basura dentro, basura fuera). La programación cuidadosa pretende construir un programa a "prueba de balas", incluyendo unas rutinas de entrada que analicen la información introducida por el usuario y, de alguna forma, prevenir que se aplique información incorrecta al programa. Si pudiéramos construir un autómata finito que aceptara solamente las cadenas que representan información correcta, entonces tendríamos un modelo para dicha rutina de entrada. Puesto que los autómatas finitos se corresponden con las expresiones regulares, el problema se reduce a especificar la información correcta por medio de expresiones regulares.

En el caso de que la entrada esté formada por enteros sin signo, el lenguaje vendrá dado por $I = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \cdot \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*$. Es fácil construir un autómata finito que acepte I (véase Figura 2.53).

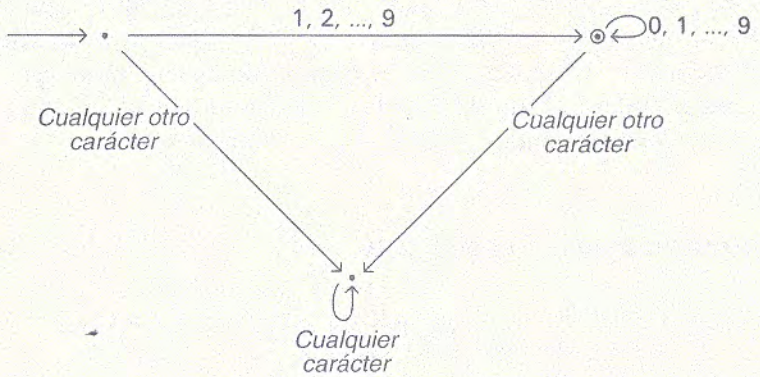


Figura 2.53

También es sencillo traducir el autómata finito a un código en un lenguaje de programación; sólo se necesita seguir el rastro de la posición actual en la cadena y del estado actual. A la vez que se recorre la cadena, se cambia de estado según corresponda y, cuando se acaba la cadena, se comprueba a qué estado se ha llegado y, según eso, se acepta o se rechaza la cadena.

Las expresiones regulares se pueden usar para especificar las unidades léxicas presentes en un lenguaje de programación. Los autómatas finitos asociados

se usan para reconocer dichas unidades (llamadas *componentes léxicos*). Dicho análisis es una etapa importante en la compilación de programas de ordenador. Por ejemplo, sea Σ el conjunto de caracteres de algún lenguaje de programación, es decir, todos los caracteres que debe reconocer un compilador para dicho lenguaje. Sea $L \subset \Sigma$ el subconjunto de todas las letras y $D \subset \Sigma$ el subconjunto de todos los dígitos. Supongamos que un comentario en dicho lenguaje comienza por los caracteres "- -" y termina con el símbolo de final de línea, el cual se denota mediante eol. Entonces una expresión regular para el componente léxico conocido como *comentario* es - -(Σ-eol)*eol.

Los identificadores del lenguaje pueden estar compuestos por letras, dígitos y subrayados, deben empezar con una letra y deben terminar con una letra o un dígito. Una expresión regular para estos componentes léxicos vendrá dada por $L\,(L \cup D \cup \_)^*\,(L \cup D)$. Se puede construir un autómata finito que reconozca este lenguaje regular y, por tanto, escribir un código apropiado para el reconocimiento de identificadores.

En la etapa de análisis léxico de un compilador, hay un aspecto que no estaba presente en el ejemplo de los enteros como entrada y es el hecho de que al tratar de reconocer un lexema se tienen distintas posibilidades. Por ejemplo, si la cadena actual fracasa como comentario, nos gustaría comprobar si puede ser un identificador y, si fracasa también aquí, si puede ser cualquier otro componente léxico. Generalmente, en un compilador, el autómata finito que reconoce todos los componentes léxicos está ordenado de alguna manera y, sistemáticamente, se aplica a la cadena hasta que se tiene éxito o falla en todo. Si falla, la cadena no puede formar parte de un programa construido correctamente.

Ejercicios de la Sección 2.10

2.10.1. Escriba un código, en su lenguaje de programación favorito, para implementar el autómata finito de la Figura 2.53.

2.10.2. Escriba un programa, en su lenguaje de programación favorito, para implementar el autómata finito que acepta enteros con signo y sin signo.

2.10.3. Escriba una rutina, en su lenguaje de programación favorito, que identifique números *reales* con y sin signo (para simplificar, suponemos que los números reales no vienen representados en notación exponencial, es decir, 1.23e-9).

2.10.4. Escriba una rutina, en su lenguaje de programación favorito, que identifique una cadena como un entero con y sin signo, un numero real con y sin signo, un comentario o un identificador (mire el ejemplo que precede a los ejercicios).

PROBLEMAS

2.1. El algoritmo de Moore. Sabemos que si L_1 y L_2 son lenguajes regulares sobre Σ, entonces $\Sigma^* - L_1$ y $\Sigma^* - L_2$ son lenguajes regulares y por consiguiente $L_1 \cap (\Sigma^* - L_2)$ y $L_2 \cap (\Sigma^* - L_1)$ también lo son.

Sea $L = (L_1 \cap (\Sigma^* - L_2)) \cup (L_2 \cap (\Sigma^* - L_1))$ y obsérvese que, puesto que L es regular, es aceptado por una autómata M. Por el Teorema 2.9.2, podemos determinar si M acepta alguna cadena (es decir, si $L = \emptyset$ o no). Pero obsérvese que, si M acepta una cadena, entonces L contiene una cadena y, por tanto, $L_1 \cap (\Sigma^* - L_2)$ y $L_2 \cap (\Sigma^* - L_1)$ no pueden ser ambos vacíos.

Supongamos que tenemos $L_1 \cap (\Sigma^* - L_2) \neq \emptyset$. Entonces existirá alguna cadena que esté en L_1 y no en L_2, y por tanto $L_1 \neq L_2$. Igualmente, si $L_2 \cap (\Sigma_* - - L_1) \neq \emptyset$, también llegaremos a que $L_1 \neq L_2$. Por otro lado, si M no acepta ninguna cadena, entonces $L = \emptyset$ y, por tanto, $L_1 \cap (\Sigma^* - L_2)$ y $L_2 \cap (\Sigma^* - L_1)$ serán ambos vacíos. Es decir, no hay ninguna cadena en L_1 que no lo esté en L_2, y viceversa, y así $L_1 = L_2$.

Hemos probado que hay un algoritmo para determinar si dos lenguajes regulares son el mismo. Sin embargo, nuestro algoritmo no es particularmente eficiente, ya que primero tenemos que construir el lenguaje L, obtener un AFD para él y después determinar si dicho AFD acepta alguna cadena. A continuación veremos un algoritmo mucho menos complejo que el de Moore.

Supongamos que M y M' son dos AFD sobre el alfabeto Σ. Para que la presentación sea lo más sencilla posible, supondremos que $\Sigma = \{a, b\}$. Primero renombraremos los estados de M y M' para que todos los estados sean distintos. Supongamos que q_1 y q_1' son los estados iniciales de M y M', respectivamente.

Construiremos una tabla de comparación que (en este caso) consta de tres columnas. Las entradas de cada columna son pares de estados (q, q'), uno de M y otro de M'. La entrada de la columna 1 indica el par de estados que será tratado en la fila correspondiente. La entrada de la columna 2 es el par de estados que sigue a los de la columna 1 mediante una transición con a. Del mismo modo, la entrada de la columna 3 es el estado siguiente por medio de una transición con b.

Por tanto, si (q, q') están en una entrada de la columna 1 y (p, p') y (r, r') son las entradas de la columna 2 y la columna 3 para la misma fila, entonces $\delta(q, a) = p$, $\delta'(q', a) = p'$, $\delta(q, b) = r$ y $\delta'(q', b) = r'$ son las transiciones de M y M'. Construiremos la tabla fila por fila empezando por (q_1, q_1') como entrada de la columna 1 y primera fila. En general, si (q, q') está en la columna 1 de cualquier fila, rellenaremos la columna 2 y la columna 3 de forma apropiada. Si cualquiera de las entradas de la columna 1 y la columna 2 no están ya en la columna 1 se añadirán antes de seguir con la fila siguiente.

Siempre que en la tabla (y para cualquier columna) encontremos un par (p, p') en el cual p es un estado final de M, pero p' no es un estado final de M' (o viceversa), se parará el proceso ya que habremos llegado a la conclusión de que

M y M' no son equivalentes. De otro modo, el proceso parará cuando no quede ninguna fila por completar. En este caso, M y M' son equivalentes.

Por ejemplo, los AFD de la Figura 2.54 no son equivalentes porque su tabla es (parcialmente) como sigue:

Columna 1	transición a columna 2	transición b columna 3
(q_1, q_1')	(q_1, q_1')	(q_2, q_2')
(q_2, q_2')	(q_3, q_4')	(q_1, q_3')

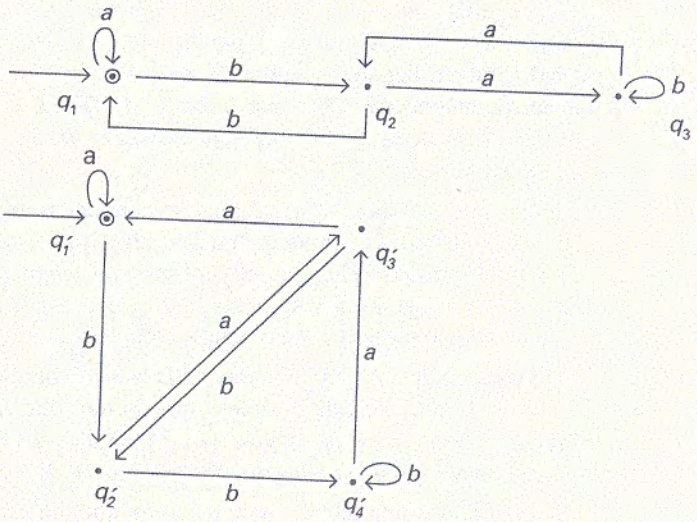


Figura 2.54

Aunque la tabla no se ha completado, el proceso termina porque q_1 es un estado final del primer AFD, pero q_3' no es un estado final del segundo. Por tanto, dichos AFD no son equivalentes.

El algoritmo de Moore se puede ampliar apropiadamente para cualquier alfabeto Σ. Simplemente se incluirá una columna para cada símbolo de Σ.

1. ¿Son equivalentes los AFD de la Figura 2.55 de la página 95?
2. ¿Son equivalentes los AFD de la Figura 2.56 de la página 95?

2.2. Considérese el AFD dado en la Figura 2.57 de la página 96. Obsérvese que ciertos estados se comportan de la misma forma para toda cadena de entrada. Por ejemplo, si estamos en el estado q_2 o q_8 y analizamos cualquier cadena de entrada no vacía, llegamos al mismo estado. De alguna forma la presencia de q_2 y q_8 es redundante.

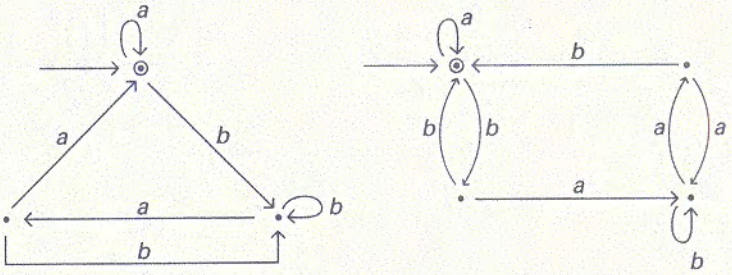
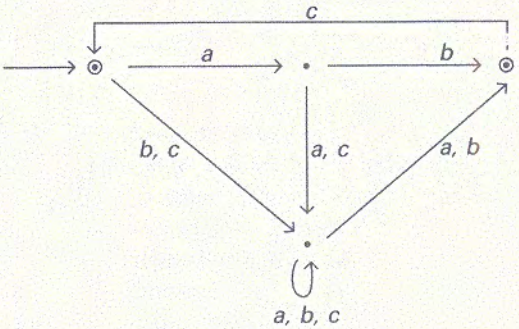


Figura 2.55



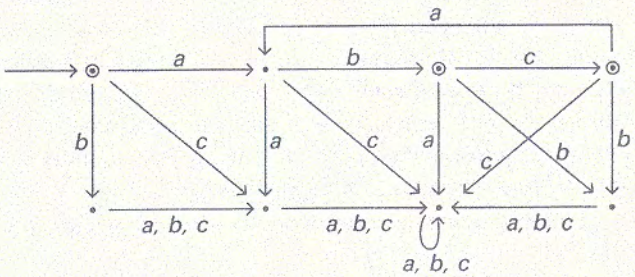


Figura 2.56

Conviene obtener un AFD para un lenguaje que sea el AFD *mínimo*, en el sentido de que tenga un número mínimo de estados. Esencialmente, lo que haremos será eliminar todos los estados redundantes (según vimos anteriormente).

Sea $M = (Q, \Sigma, s, F, \delta)$ un AFD. Los estados p y q son *distinguibles* si para alguna cadena x de Σ^*, se tiene que $\delta(p, x) \in F$ y $\delta(q, x) \notin F$, o viceversa. Si todos los pares de estados son distinguibles, M no tiene estados redundantes y, por tanto, ya es un AFD mínimo. Por otro lado, si M contiene uno o más conjuntos de estados no distinguibles, se puede eliminar la redundancia reemplazando cada conjunto por un único estado.

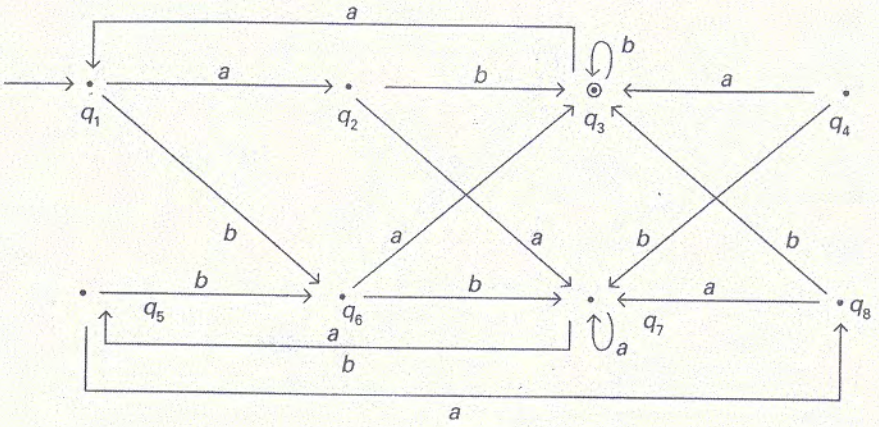


Figura 2.57

Los pares de estados equivalentes (no distinguibles) se pueden encontrar por medio de una tabla en la cual cada fila y columna corresponden a un estado. Inicialmente, se marcan como distinguibles las entradas correspondientes a un estado final y a un estado no final. Entonces, para cada par de estados que no se conocen como distinguibles, se considera $p_a = \delta(p, a)$ y $q_a = \delta(q, a)$ para todo $a \in \Sigma$. Si p_a y q_a son distinguibles por medio de la cadena x, entonces p y q son distinguibles por medio de la cadena ax.

Así, si la celda correspondiente a p_a y q_a está marcada para alguna a, marcaremos la celda para p y q. Si para todo $a \in \Sigma$, no está marcada la celda correspondiente a p_a y q_a, introduciremos (p, q) en una lista asociada con (p_a, q_a) para todo a. Si posteriormente se obtiene que p_a y q_a son distinguibles, se marcarán también p y q. Puesto que las celdas simétricas con respecto a la diagonal corresponden a los mismos pares de estados, necesitamos menos de la mitad de la tabla. Es más, las celdas correspondientes a la diagonal son no distinguibles.

La tabla siguiente corresponde al AFD del ejemplo precedente.

q_2	x						
q_3	x	x					
q_4	x	x	x				
q_5		x	x	x			
q_6	x	x	x		x		
q_7	x	x	x	x	x	x	
q_8	x		x	x	x	x	x
	q_1	q_2	q_3	q_4	q_5	q_6	q_7

Las colecciones de estados no distinguibles son $\{q_1, q_5\}$, $\{q_2, q_8\}$, $\{q_4, q_6\}$, $\{q_3\}$ y $\{q_7\}$. En el AFD reducido que obtenemos reemplazaremos cada colección de estados no distinguibles por un único estado. Por tanto, el AFD reducido para nuestro ejemplo es el de la Figura 2.58.

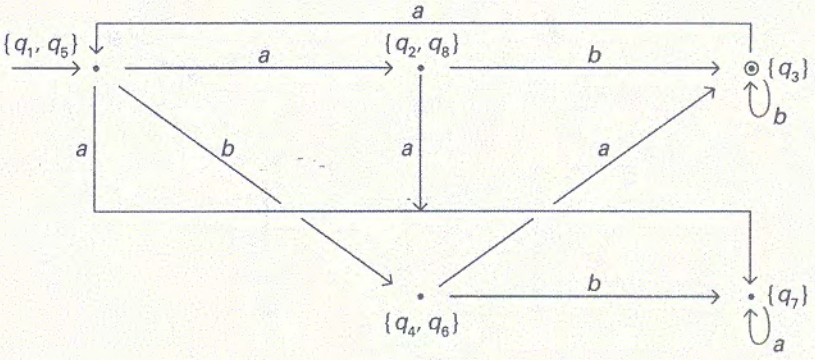


Figura 2.58

1. Obtener los AFD mínimos que correspondan a los AFD de la Figura 2.59.

2.3. Sea $\Sigma = \{(a, b, c) \mid a, b, c \in \{0, 1\}\}$ el alfabeto formado por todas las 3-tuplas de *ceros* y *unos*. Trataremos cada 3-tupla como un vector columna, es decir, $\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$. Entonces, una suma binaria tal como

$$\begin{array}{r} 0011 \\ +\ 0101 \\ \hline 1000 \end{array}$$

se puede interpretar como la cadena $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$ sobre Σ.

1. Probar que el lenguaje L_1 sobre Σ, compuesto por todas las cadenas que representan sumas binarias "correctas", es un lenguaje regular.

2. Usar el lema de bombeo (Lema 2.9.1) para probar que el lenguaje L_2 formado por todas las cadenas que representan productos binarios correctos no es un lenguaje regular.

   Hay muchas formas de tratar la adición. Cuando realizamos una suma, generalmente sumamos los pares de dígitos correspondientes y un valor previo para obtener un dígito resultante y un resto. Consideremos la suma binaria donde el valor de entrada es un 0 ó un 1. Si el valor previo es un 0, entonces se obtiene un resto sólo si el par de dígitos son dos *unos*. Si

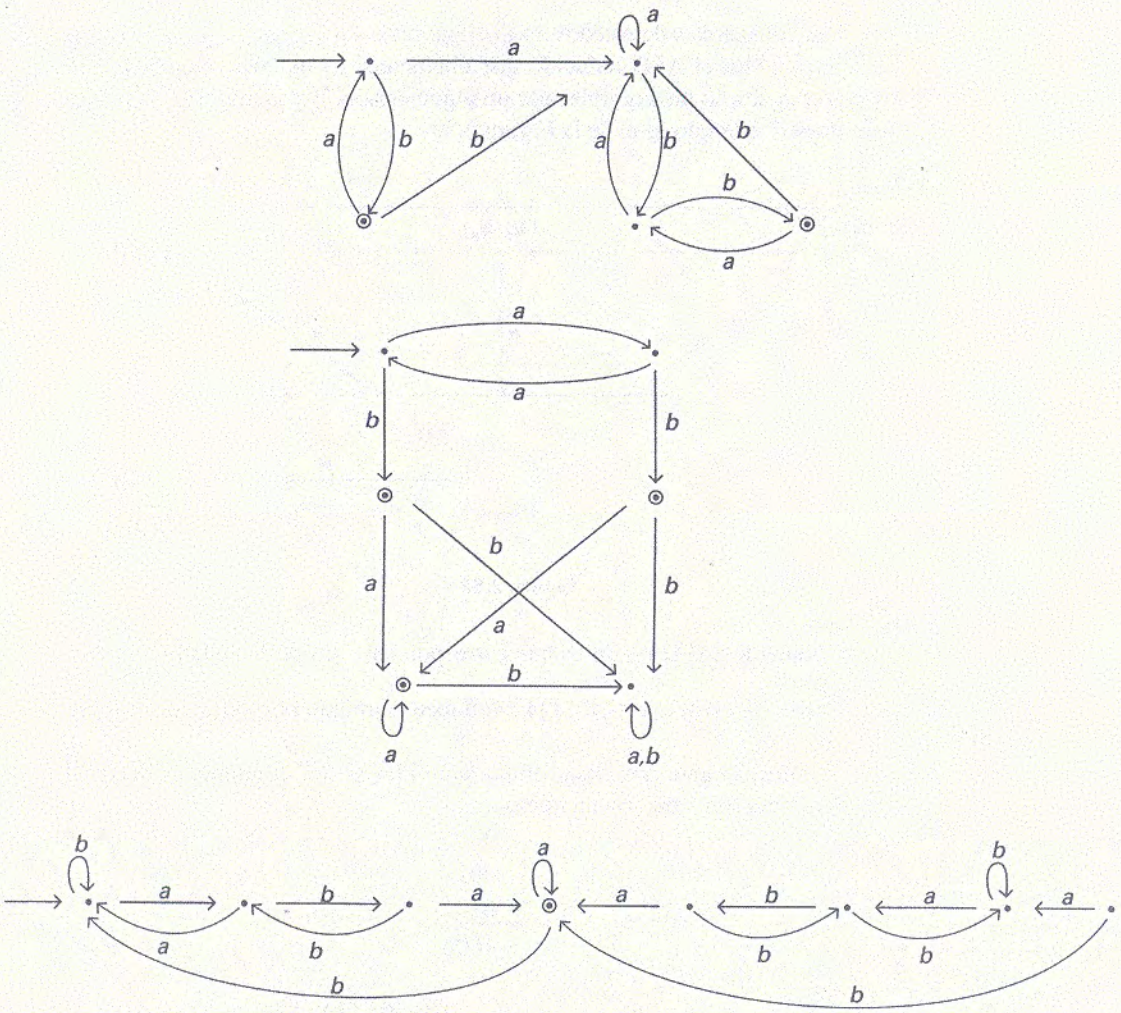


Figura 2.59

el valor previo es un 1, entonces la única forma de que un resto *no* sea un 1 es que el par de dígitos a sumar sean dos *ceros*.

Vamos a usar la notación $(x, y)/z$ para denotar el par de dígitos (x, y) que representan a los sumandos y z para indicar el dígito que resulta al sumar $x + y +$ (valor existente). Entonces, la suma binaria puede ser representada mediante el diagrama de la Figura 2.60. Obsérvese que este diagrama es muy semejante a un diagrama de transición de un autómata finito. De hecho, tendremos dos estados distintos que corresponden a los valores previos, y unas transiciones entre los estados que dependen de los dígitos a sumar (entrada), así como de los valores actuales (estado), y de un estado

inicial (el valor es inicialmente 0). La única diferencia entre este diagrama y el diagrama de transición de un autómata finito es que en éste la salida se representa como el resultado de una suma.

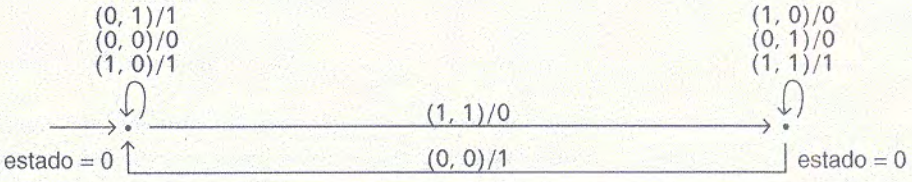


Figura 2.60

Un *transductor de estados finito determinista* (autómata que produce una salida) es una 6-tupla $M = (Q, \Sigma, \Gamma, s, \delta, \tau)$, donde Q es un conjunto finito de estados que contiene un estado inicial distinguible s; Σ y Γ son alfabetos, siendo Σ el alfabeto de *entrada* y Γ el alfabeto de *salida*; δ es la función de transición, donde $\delta: Q \times \Sigma \to Q$; y τ es la función de *salida*, donde $\tau: Q \times \Sigma \to \Gamma$.

Obsérvese que, al igual que en los autómatas finitos vistos en este capítulo, δ depende del estado actual y del símbolo de entrada actual. τ también depende del estado y la entrada actual y proporciona una salida.

Es importante notar que no hay ningún conjunto de estados finales. Los transductores no se ocupan de aceptar la entrada, sino de transformarla en una salida. En este sentido, los transductores transforman cadenas de entrada en cadenas de salida. Es decir, computan una *función* de Σ^* en Γ^*.

3. Considérese una máquina expendedora de latas de soda. Por sencillez, suponemos que hay un botón de selección y que la soda cuesta 0.30 dólares. También suponemos que, una vez que se han introducido 0.30 dólares en la máquina, cualquier moneda que se introduzca posteriormente será devuelta. Crear un transductor de estados finito determinista que modele el comportamiento de la máquina expendedora.

4. ¿Cómo se debería construir el transductor de estados finito determinista para que *acepte* un lenguaje?

2.4. Caracterización de los lenguajes regulares. $L \subseteq a^*$. En la Sección 2.9 y en sus ejercicios, vimos lenguajes no regulares de la forma

$$\{a^i \mid i \text{ satisface alguna condición}\}$$

Por otro lado, muchos lenguajes que son de esta forma *son* regulares, por ejemplo, los lenguajes $L_k = \{a^{k+i} \mid i \geq 0\}$ para $k = 0, 1, \ldots$ Es razonable preguntarse bajo qué condiciones es regular un lenguaje de estas características. En este problema obtendremos un resultado que responderá a dicha pregunta.

Una *progresión aritmética* es una secuencia de números naturales igualmente espaciados. Por ejemplo, $\{4, 7, 10, \ldots\}$ es una progresión aritmética. Toda progresión aritmética tiene dos parámetros que la determinan completamente: el punto de partida p y la diferencia común q. Se puede definir una progresión aritmética en términos de esos dos parámetros:

$$A_{pq} = \{x \mid x = p + nq \text{ para algún } n \in \mathbb{N}\}$$

Obsérvese que, según esta definición, un conjunto que contenga un único número natural es una progresión aritmética con $q = 0$. Así, por ejemplo, $\{3\}$ es la progresión aritmética A_{30}.

1. Sea A_{pq} una progresión aritmética. Probar que el lenguaje L dado a continuación es regular.

$$L = \{a^i \mid i \in A_{pq}\}$$

Un conjunto X de números naturales es *finalmente periódico* si X es finito o si hay dos números naturales $n_0 \geq 0$ y $t \geq 1$ para los cuales, si $x \geq n_0$, entonces $x \in X$ si y sólo si $x + t \in X$. Por ejemplo, el conjunto $\{2, 3, 7, 14, 103, 109, 115, 121, \ldots\}$ es finalmente periódico con $n_0 = 103$ y $t = 6$. Fíjese en que cualquier progresión aritmética A_{pq} es finalmente periódica ya que, si $x > p$, entonces $x \in A_{pq}$ si y sólo si $x = p + nq$ para algún n, si y sólo si $x + q = p + (n + 1)\, q \in A_{pq}$. Es decir, se puede tomar $n_0 = p$ y $t = q$.

2. Probar que la unión de dos progresiones aritméticas es un conjunto finalmente periódico.

3. Probar que la unión de un conjunto X finalmente periódico con una progresión aritmética A_{pq} es un conjunto finalmente periódico.

   Obsérvese que los Ejercicios 2 y 3 implican que la unión finita de progresiones aritméticas es un conjunto finalmente periódico.

4. Probar que todo conjunto finalmente periódico es la unión finita de progresiones aritméticas. *Indicaciones:* Obviamente, si X es finito queda probado. Supongamos que X es infinito. El conjunto $\{x \mid x < n_0\}$ es finito. ¿Qué relación existe entre el conjunto finito $\{x \mid n_0 \leq x < n_0 + t\}$ y el conjunto $\{x \mid x \geq n_0 + t\}$?

   El resultado de los Ejercicios 3 y 4 es que un conjunto X es finalmente periódico si y sólo si es unión finita de progresiones aritméticas.

5. Probar que si $L \subseteq a^*$ y $\{i \mid a^i \in L\}$ es finalmente periódico, entonces L es regular.

6. Probar que si $L \subseteq a^*$ es regular, entonces $\{i \mid a^i \in L\}$ es finalmente periódico. *Indicación:* Aplicar el Ejercicio 4 y el lema de bombeo.

7. Usar los Ejercicios 5 y 6 para probar que el lenguaje

$$L = \{a^{n^2} \mid n \geq 1\}$$

no es regular. *Indicación:* Considere el conjunto

$$X = \{n^2 \mid n \geq 1\}$$

¿Es finalmente periódico?

8. Aplicar los Ejercicios 5 y 6 para probar que los siguientes lenguajes no son regulares:

   (a) $\{a^{2^n} \mid n \geq 1\}$

   (b) $\{a^p \mid p \text{ es un primo}\}$

   (c) $\{a^{n!} \mid n \geq 1\}$

2.5. Homomorfismos y sustitución. El Teorema 2.8.1 demuestra algunas propiedades de cierre de lenguajes regulares: el conjunto de lenguajes regulares es cerrado respecto a la unión, concatenación y cerradura de estrella. En este problema estudiaremos otras dos propiedades de esta clase de lenguajes.

Sean los alfabetos Σ_1 y Σ_2. Una *sustitución* asocia cada símbolo $a \in \Sigma_1$ con un lenguaje $S \subseteq \Sigma_2^*$. Formalmente, definiremos una sustitución como una función $f: \Sigma_1 \to 2^{\Sigma_2^*}$ tal que $f(a_i) = S_i$, donde $a_i \in \Sigma_1$ y $S_i \in \Sigma_2^*$. Extender la sustitución a cadenas y lenguajes sobre Σ_1 de forma que

$$f(\varepsilon) = \varepsilon$$
$$f(wa) = f(w)f(a)$$

donde $w \in \Sigma_1^*$ y $a \in \Sigma_1$.

Por ejemplo, sea $\Sigma_1 = \{a, b\}$ y $\Sigma_2 = \{0, 1\}$. Se definen $f(a) = \{011\}^*$ y $f(b) = \{1001, 01101\}$. Entonces

$$f(aba) = \{011\}^* \{1001, 01101\} \{011\}^*$$

$$f(ab^*) = \bigcup_{i=0}^{\infty} f(ab^i) = \bigcup_{i=0}^{\infty} f(a)f(b^i) = \bigcup_{i=0}^{\infty} \{011\}^* \{1001, 01101\}^i$$

$$= \{011\}^* \bigcup_{i=0}^{\infty} \{1001, 01101\}^i = \{011\}^* \{1001, 01101\}^*$$

1. Sean los alfabetos Σ_1 y Σ_2 y f una sustitución, donde, para todo $a \in \Sigma_1$, $f(a) = R_a \subseteq \Sigma_2^*$ es un lenguaje regular. Sean a y b elementos de Σ_1.

   (a) Probar que $f(a \cup b) = f(a) \cup f(b)$.

   (b) Probar que $f(a^*) = f(a)^*$.

(c) Sea $R \subseteq \Sigma_1$ un lenguaje regular. Probar que $f(R)$ es un lenguaje regular. *Indicación:* Aplique inducción sobre el número de operadores en una expresión regular para R.

Supongamos que f es una sustitución en la cual, para todo $a \in \Sigma_1$, $f(a)$ contiene sólo una cadena. Dicha sustitución se llama *homomorfismo*. Si $L \subseteq \Sigma_1^*$, se dice que $f(L)$ es la *imagen homomórfica* del lenguaje L. Si $L \subseteq \Sigma_2^*$, se dice que $f^{-1}(L)$ es la *imagen homomórfica inversa* del lenguaje L.

Por ejemplo, sea $f: \{a, b, c\} \rightarrow \{a, b\}^*$ definida como $f(a) = a$, $f(b) = ba$ y $f(c) = a$. Entonces, si $L_1 = a^*(b \cup c)^*$, se obtiene que $f(L_1) = a^*(ba \cup a)^*$. Si $L_2 = (aba \cup a)^*$, entonces la imagen inversa es $f^{-1}(L_2) = ((a \cup c)\, a \cup (a \cup c))^*$. [Obsérvese que $f(a) = f(c) = a$ por lo que $f^{-1}(a) = a \cup c$.] En este caso, L_2 es un lenguaje regular y $f^{-1}(L_2)$ también lo es. Esto, como se verá en el Ejercicio 2, no es una coincidencia.

2. Probar que, si L es un lenguaje regular y f es un homomorfismo, entonces $f^{-1}(L)$ es un lenguaje regular.

La afirmación anterior se puede generalizar. Si f es una sustitución, entonces $f^{-1}(L)$ es regular si L es regular. Uniendo los ejercicios 1 y 2, se tiene que la clase de lenguajes regulares es cerrada con respecto a la imagen homomórfica y la imagen homomórfica inversa.

Puesto que los homomorfismos y sus inversas "preservan" la regularidad, se pueden usar para determinar si un lenguaje es regular o no. La idea es tomar un lenguaje que no se sabe si es regular y tratar de transformarlo por medio de homomorfismos en un lenguaje regular o no regular conocido.

Por ejemplo, se sabe que el lenguaje $\{a^n b^n \mid n \geq 1\}$ no es regular. Considere el lenguaje $L = \{a^n b a^n \mid n \geq 1\}$. Aunque el lema de bombeo se puede aplicar para deducir que L no es regular, por medio de homomorfismos llegaremos a la misma conclusión. Supongamos que L es regular. Sea $f: \{a, b, c\} \rightarrow \{a, b\}^*$ el homomorfismo definido previamente y consideremos $L_1 = f^{-1}(L) = f^{-1}(\{a^n b a^n \mid n \geq 1\})$. Ya que $f^{-1}(a) = a \cup c$, se obtiene que

$$f^{-1}(a^n b a^n) = \{a^i c^j a^k c^l b a^r c^s a^t c^u \mid i + j + k + l = n \quad \text{y} \quad r + s + t + u + 1 = n\}$$

Por tanto

$$f^{-1}(L) = \{a^i c^j a^k c^l b a^r c^s a^t c^u \mid i + j + k + l = r + s + t + u + 1\}$$

Entonces

$$f^{-1}(L) \cap a^* b c^* = \{a^n b c^{n-1} \mid n \geq 1\}$$

Ya que $a^* b c^*$ es, claramente, regular y ya que suponemos que L es regular y que $f^{-1}(L)$ es regular, debemos obtener que $\{a^n b c^{n-1} \mid n \geq 1\}$ tam-

bién es regular. Ahora sea g: $\{a, b, c\} \rightarrow \{a, b\}^*$ el homomorfismo definido mediante $g(a) = a$ y $g(b) = g(c) = b$. Entonces se tiene

$$g(f^{-1}(L) \cap a^*bc^*) = g(\{a^nbc^{n-1} \mid n \geq 1\}) = \{a^nb^n \mid n \geq 1\}$$

Puesto que g es un homomorfismo, tendremos que $\{a^nb^n \mid n \geq 1\}$ es regular, lo que es una contradicción. Por tanto, no puede ser que $\{a^nba^n \mid n \geq 1\}$ sea regular.

3. Probar que $\{a^ib^jc^i \mid i \geq j \geq 1\}$ no es regular.
4. Probar que $\{a^iba^j \mid i \neq j \text{ e } i, j \geq 1\}$ no es regular.

3

Lenguajes independientes del contexto

3.1 GRAMÁTICAS REGULARES

Las expresiones regulares y los autómatas finitos nos proporcionan dos medios para especificar o definir lenguajes. Las expresiones regulares nos proporcionan una plantilla o patrón para las cadenas del lenguaje. Todas las cadenas se corresponden con un patrón en particular y dichas cadenas serán las únicas que formarán dicho lenguaje. Igualmente, un autómata finito especifica un lenguaje como el conjunto de todas las cadenas que lo hacen pasar del estado inicial a uno de sus estados de aceptación. También se podría interpretar un autómata como un generador de cadenas del lenguaje, según se plantea a continuación. Un símbolo se genera al recorrer el camino etiquetado con dicho símbolo y que parte del estado actual al siguiente. Se empieza con la cadena vacía y se obtiene una cadena del lenguaje cuando el recorrido llega a un estado de aceptación.

Por ejemplo, se considera el autómata finito dado por el diagrama de transición de la Figura 3.1. Este autómata finito acepta el lenguaje regular $a\,(a^* \cup b^*)\,b$. Imaginemos que se comienza en el estado inicial y se atraviesa el diagrama de alguna forma. Cuando un camino va de un estado a otro, la "salida" es el símbolo que etiqueta dicho camino. Por tanto se podría obtener la cadena de salida aa^2b pasando por los estados q_1 - q_2 - q_3 - q_3 - q_3 - q_5. Se ve fácilmente que las cadenas generadas de esta forma serán aceptadas por este autómata finito. Es más, cualquier cadena aceptada por este autómata puede ser generada por este método.

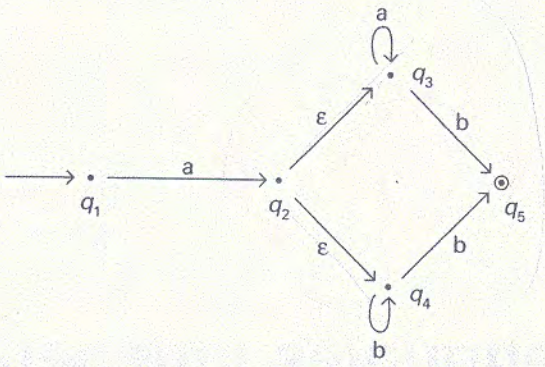


Figura 3.1

Obsérvese que todas las cadenas del lenguaje precedente estarán formadas por una *a* seguida de alguna "parte final". Si hacemos que *E* represente la parte final, lo dicho se puede representar simbólicamente mediante $S \rightarrow aE$. La flecha $\rightarrow$ se puede interpretar como "puede ser" o "se compone de". La parte final de una cadena estará formada por una de las dos listas de *aes* o *bes*. Por tanto, para indicar las múltiples posibilidades que hay para *E* podemos escribir $E \rightarrow A$ y $E \rightarrow B$. Las dos listas de *aes* y *bes* se pueden expresar como $A \rightarrow aA$ junto con $A \rightarrow b$ para indicar que una cadena de *aes* va seguida de una *b* o como $B \rightarrow bB$ junto con $B \rightarrow b$, para indicar que una cadena de *bes* va seguida de otra *b*.

En resumen, tendremos las siguientes expresiones

$$S \rightarrow aE$$
$$E \rightarrow A$$
$$E \rightarrow B$$
$$A \rightarrow b$$
$$A \rightarrow aA$$
$$B \rightarrow b$$
$$B \rightarrow bB$$

Estas expresiones pueden ser consideradas como *reglas de sustitución* para la generación de cadenas. El símbolo que se encuentra a la izquierda de la flecha se puede sustituir por la cadena de la derecha.

Por ejemplo, podemos generar *aab* empezando por *S*, sustituyéndola por *aE*, sustituyendo la *E* por *aA* y finalmente sustituyendo la *A* por *b*. Tendremos una secuencia de cadenas comenzando por *S* y terminando con *aab*. En cada paso, las letras mayúsculas (*S*, *E* y *A*) representan la parte de la cadena final que todavía no se ha generado. Bajo estas circunstancias tiene sentido interpretar la flecha en las expresiones precedentes como "es sustituido por".

Finalmente, introduciremos el símbolo | que será interpretado como "o". Si se usa este símbolo, las dos reglas $E \to A$ y $E \to B$, se pueden unir en $E \to A \mid B$. La colección precedente de reglas para generar cadenas puede volverse a escribir como sigue:

1. $S \to aE$
2. $E \to A \mid B$
3. $A \to aA \mid B$
4. $B \to bB \mid b$

La cadena a^3b se puede generar a partir de S aplicando primero la regla 1 para obtener aE. Entonces, aplicando la regla 2 se obtiene aA y aplicando la regla 3 se obtiene aaA y $aaaA$; finalmente, se puede aplicar la segunda parte de la regla 3 para obtener $aaab$. Podremos escribir una descripción del proceso de generación como

$$S \Rightarrow aE \Rightarrow aA \Rightarrow aaA \Rightarrow aaaA \Rightarrow aaab$$

donde la doble flecha $\Rightarrow$ se interpreta como "deriva", "produce" o "genera". Usaremos la notación $E \overset{*}{\Rightarrow} w$ para indicar que la cadena w se deriva a partir de S en 0 o más etapas.

Obsérvese que en este modelo hemos introducido una colección de nuevos símbolos para representar las porciones de cadena que no han sido generadas. Cuando las cadenas han sido completamente generadas, estarán formadas en su totalidad por símbolos del alfabeto Σ, pero antes de llegar a esto se obtendrán cadenas formadas por símbolos del alfabeto y por nuevos símbolos. Los nuevos símbolos se llaman *no terminales*, para indicar que deben ser sustituidos por símbolos del alfabeto antes de que la cadena haya sido totalmente generada. Por otro lado, los símbolos del alfabeto Σ se llaman *terminales*, para indicar que no es posible que sean sustituidos. Obsérvese, también, que el símbolo usado para representar a una cadena que no ha comenzado a generarse, debe ser necesariamente un no terminal. Finalmente, observemos que hemos generado las cadenas del lenguaje de izquierda a derecha —en las cadenas de las etapas intermedias por las que se pasa al aplicar las reglas, los no terminales deben aparecer solamente en el extremo derecho. Esto refleja la forma en la que un autómata finito reconocería una cadena del lenguaje.

Daremos la siguiente definición:

Definición 3.1.1. Una *gramática regular* G es una 4-tupla $G = (\Sigma, N, S, P)$, donde Σ es un alfabeto, N es una colección de símbolos no terminales, S es un no terminal llamado *símbolo inicial*, y P es una colección de reglas de sustitución, llamadas

producciones, y que son de la forma $A \rightarrow w$, donde $A \in N$ y w es una cadena sobre $\Sigma \cup N$ que satisface lo siguiente:

1. w contiene un no terminal como máximo.
2. Si w contiene un no terminal, entonces es el símbolo que está en el extremo derecho de w.

El *lenguaje generado* por la gramática regular G se denota por $L(G)$.

Por ejemplo, considérese la gramática regular $G = (\Sigma, N, S, P)$, donde

$$\begin{aligned} \Sigma &= \{a, b\} \\ N &= \{S, A\} \\ P : S &\rightarrow bA \\ A &\rightarrow aaA \mid b \mid \varepsilon \end{aligned}$$

Obsérvese que $L(G)$ contendrá todas las cadenas de la forma $ba^{2n}b$ y ba^{2n}. Es decir, $L(G) = b(a^2)^*(b \cup \varepsilon)$. Se puede demostrar, por inducción sobre n, que todas las cadenas de la forma $ba^{2n}b$ o ba^{2n} están en $L(G)$ y, por inducción sobre la longitud de una derivación, se demuestra que $L(G)$ está contenido en $b(a^2)^*(b \cup \varepsilon)$. (La etaba base es para una derivación de longitud 2).

De la definición se deduce que el lado derecho de una producción es una cadena de $\Sigma^*(N \cup \varepsilon)$. Obsérvese que ε puede ser el lado derecho de una producción. En el ejemplo precedente, la producción $A \rightarrow \varepsilon$ acaba con la generación de una cadena (al igual que la producción $A \rightarrow b$) ya que se "borra" el no terminal A.

Dado que las producciones emparejan no terminales de N con cadenas de $\Sigma^*(N \cup \varepsilon)$, conviene representarlas como pares ordenados de $N \times \Sigma^*(N \cup \varepsilon)$. Por tanto, el par (x, y) de $N \times \Sigma^*(N \cup \varepsilon)$ representa a la producción $x \rightarrow y$. Las producciones de P del ejemplo anterior se podrían representar mediante

$$P = \{(S, bA), (A, aaA), (A, b), (A, \varepsilon)\}$$

Si se llega al acuerdo de escribir los no terminales con letras mayúsculas y los terminales con letras minúsculas y además se conviene que S se use como símbolo inicial, entonces una gramática regular puede ser completamente especificada por medio de sus producciones. Por ejemplo, $S \rightarrow aS \mid b$ especifica completamente la gramática regular que genera el lenguaje a^*b.

Ejercicios de la Sección 3.1

3.1.1. Usar las reglas de la Figura 3.1 para derivar ab, ab^3, aa^3b. ¿Es posible derivar $abab$?

3.1.2. Supongamos que tenemos las reglas $S \to aS \mid bT$ y $T \to aa$. Dar una derivación para $abaa$, $aabaa$ y $aaabaa$. Probar como se deriva a^kba^2 para $k \geq 1$. ¿Es posible derivar las cadenas baa, b o aa?

3.1.3. Obtener una gramática regular para los siguientes lenguajes:

(a) $a^*b \cup a$

(b) $a^*b \cup b^*a$

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

3.1.4. La gramática regular dada por

$$S \to bA \mid aB \mid \varepsilon$$
$$A \to abaS$$
$$B \to babS$$

genera un lenguaje regular. Obtener una expresión regular para este lenguaje.

3.1.5. En nuestra definición de gramáticas regulares se dijo que si en el lado derecho de una producción hay un no terminal, éste debe estar situado en el extremo derecho. Esto corresponde a la generación de cadenas de izquierda a derecha. Por esta razón, una gramática regular también puede llamarse *gramática regular por la derecha*. Una *gramática regular por la izquierda* es aquella cuyas cadenas son generadas por la derecha, es decir, las producciones son pares de $N \times (N \cup \Sigma)\,\Sigma^*$.

(a) Obtener una gramática regular por la izquierda para el lenguaje $\{a^nbaa \mid n \geq 0\}$.

(b) Obtener las gramáticas regulares por la derecha y por la izquierda para

$$\{w \in \{a, b, c\}^* \mid w \text{ termina en } b \text{ y toda } c \text{ va seguida por una } a\}$$

(c) Para toda gramática $G = (\Sigma, N, S, P)$ que sea regular (por la izquierda o por la derecha), se puede definir la inversa de G como $G^I = (\Sigma, N, S, P')$, donde

$$P' = \{(A, x^I) \mid (A, x) \in P\}$$

Por tanto, si $A \to aB$ es una producción de G, entonces $A \to Ba$ es una producción de G^I.

Supongamos que G es una gramática regular por la derecha.

i. Probar que G^I es una gramática regular por la izquierda.

ii. Probar que $w \in L(G)$ si y sólo si $w^I \in L(G^I)$ por inducción sobre el número de producciones usadas para obtener w.

Se puede deducir de la parte (c) que la clase de los lenguajes generados por gramáticas regulares por la izquierda es la misma que la clase de los lenguajes generados por gramáticas regulares por la derecha. Por eso habitualmente, el término *gramática regular* se aplica para referirse a cualquier gramática ya sea regular por la izquierda o regular por la derecha.

3.2 GRAMÁTICAS REGULARES Y LENGUAJES REGULARES

Supongamos que L es un lenguaje regular. Se puede obtener una gramática regular que genere L por medio de un AFD $M = (Q, \Sigma, s, F, \delta)$ para el cual $L = L(M)$. Definimos $G = (N, \Sigma, S, P)$ por

$$N = Q$$
$$\Sigma = \Sigma$$
$$S = s$$
$$P = \{(q, ap) \mid \delta(q, a) \; p\} \cup \{(q, \varepsilon) \mid q \in F\}$$

Es decir, $q \to ap$ siempre que $\delta(q, a) = p$ y $q \to \varepsilon$ si q es un estado de aceptación del AFD.

Por ejemplo, el AFD dado en la Figura 3.2 acepta el lenguaje a^*b. La gramática regular correspondiente tiene las producciones

$$q_1 \to aq_1 \mid bq_2$$
$$q_2 \to aq_3 \mid bq_3 \mid \varepsilon$$
$$q_3 \to aq_3 \mid bq_3$$

En esta gramática los q_i son no terminales (una ruptura con la notación usual, la cual puede ser restablecida sólo con renombrar los q_i y q_1 como símbolo inicial).

Obsérvese que $w \in L(M)$ para $w = \sigma_1 \sigma_2 \ldots \sigma_n$ significa que

$$\delta(s, \sigma_1 \sigma_2 \ldots \sigma_n) = p$$

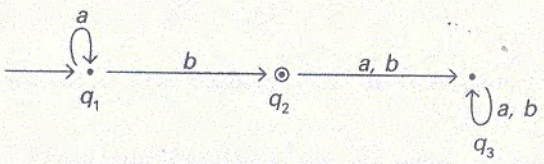


Figura 3.2

para algún $p \in F$. Si escribimos $q_{i+1} = \delta(q_i, \sigma_i)$ con $q_1 = s$, entonces se obtiene

$$\begin{aligned}\delta(s, \sigma_1 \sigma_2 \ldots \sigma_n) &= \delta(q_1, \sigma_1 \sigma_2 \ldots \sigma_n) \\ &= \delta(q_2, \sigma_2 \ldots \sigma_n) \\ &= \delta(q_3, \sigma_3 \ldots \sigma_n) \\ &\ldots \\ &= \delta(q_n, \sigma_n) \\ &= p \in F\end{aligned}$$

Ahora, puesto que $q_{i+1} = \delta(q_i, \sigma_i)$, se obtiene que $q_i \rightarrow \sigma_i q_{i+1}$ pertenece a G y, por tanto, (ya que $s = q_1$)

$$\begin{aligned}s = q_1 &\Rightarrow \sigma_1 q_2 \\ &\Rightarrow \sigma_1 \sigma_2 q_3 \\ &\ldots \\ &\Rightarrow \sigma_1 \sigma_2 \ldots \sigma_n p \\ &\Rightarrow \sigma_1 \sigma_2 \ldots \sigma_n\end{aligned}$$

Así que $w \in L(M)$ impliea que w es generada por G; es decir, $L(M) \subseteq L(G)$.

A la inversa, si w es generada por G, mediante la derivación siguiente

$$\begin{aligned}q_1 &\Rightarrow \sigma_1 q_2 \\ &\Rightarrow \sigma_1 \sigma_2 q_3 \\ &\ldots \\ &\Rightarrow \sigma_1 \sigma_2 \ldots \sigma_n p \\ &\Rightarrow \sigma_1 \sigma_2 \ldots \sigma_n\end{aligned}$$

entonces en M tendremos

$$\begin{aligned}\delta(s, \sigma_1 \sigma_2 \ldots \sigma_n) &= \delta(q_1, \sigma_1 \sigma_2 \ldots \sigma_n) \\ &= \delta(q_2, \sigma_2 \ldots \sigma_n) \\ &= \delta(q_3, \sigma_3 \ldots \sigma_n) \\ &\ldots \\ &= \delta(q_n, \sigma_n) \\ &= p \in F\end{aligned}$$

(ya que $s = q_1$). Asi, que $w \in L(G)$ implica que $w \in L(M)$, con lo que se tiene que $L(G) \subseteq L(M)$. Entonces se deduce que $L(G) = L(M)$.

También es posible partir de una gramática regular G y construir un AFN M de forma que $L(G) = L(M)$. Sea $G = (N, \Sigma, S, P)$ una gramática regular; se define $M = (Q, \Sigma, s, F, \Delta)$ mediante

$$Q = N \cup \{f\}, \quad \text{donde } f \text{ es un símbolo nuevo}$$
$$s = S$$
$$F = \{f\}$$

y Δ se construye como se indica a continuación, a partir de las producciones de P:

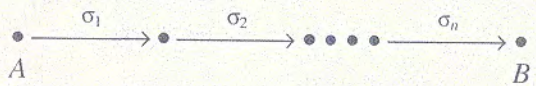
1. Si $A \to \sigma_1 \ldots \sigma_n B$ es una producción de P con A y B como no terminales, entonces se añadirán a Q los nuevos estados $q_1, q_2, \ldots, q_{n-1}$ y las transformaciones siguientes

$$\Delta(A, \sigma_1 \ldots \sigma_n) = \Delta(q_1, \sigma_2 \ldots \sigma_n) = \ldots = \Delta(q_{n-1}, \sigma_n) = B$$

2. Si $A \to \sigma_1 \ldots \sigma_n$ es una producción de P, entonces se añadirán a Q los nuevos estados $q_1, q_2, \ldots, q_{n-1}$ y a Δ, las transiciones siguientes

$$\Delta(A, \sigma_1 \ldots \sigma_n) = \Delta(q_1, \sigma_2 \ldots \sigma_n) = \ldots = \Delta(q_{n-1}, \sigma_n) = f$$

La construcción de Δ se puede concebir a partir de cómo estén etiquetadas las aristas del diagrama de transición correspondiente a M y entonces se añadirán los estados necesarios para cada uno de los símbolos de la cadena. Por tanto, si $A \to \sigma_1 \ldots \sigma_n B$, primero podríamos etiquetar las aristas entre A y B con $\sigma_1 \ldots \sigma_n$ y después añadir $n - 1$ estados nuevos, en la arista resultante:



Por ejemplo, la gramática regular

$$S \to aB \mid bA \mid \varepsilon$$
$$A \to abaS$$
$$B \to babS$$

daría como resultado el AFN cuyo diagrama de transición se muestra en la Figura 3.3.

Si G es una gramática regular y $w \in L(G)$ con $w = \sigma_1 \ldots \sigma_n$, entonces para los no terminales $A_1, A_2, \ldots, A_{n-1}$, se tiene la derivación

$$S \Rightarrow \sigma_1 A_1 \Rightarrow \ldots \Rightarrow \sigma_1 \ldots \sigma_{n-1} A_{n-1} \Rightarrow \sigma_1 \ldots \sigma_n$$

y entonces en el AFN resultante de esta construcción se tendrá

$$\Delta(s, \sigma_1 \ldots \sigma_n) = \Delta(A_1, \sigma_2 \ldots \sigma_n) = \ldots = \Delta(A_{n-1}, \sigma_n) = f$$

Por lo tanto, $w \in L(M)$. A la inversa, si $\Delta(s, \sigma_1 \ldots \sigma_n) = f$, entonces $S \overset{*}{\Rightarrow} \sigma_1 \ldots \sigma_n$, con lo que $w \in L(G)$. Luego $L(G) = L(M)$.

Aunque hemos demostrado las técnicas de construcción más usuales, en realidad hemos demostrado mucho más:

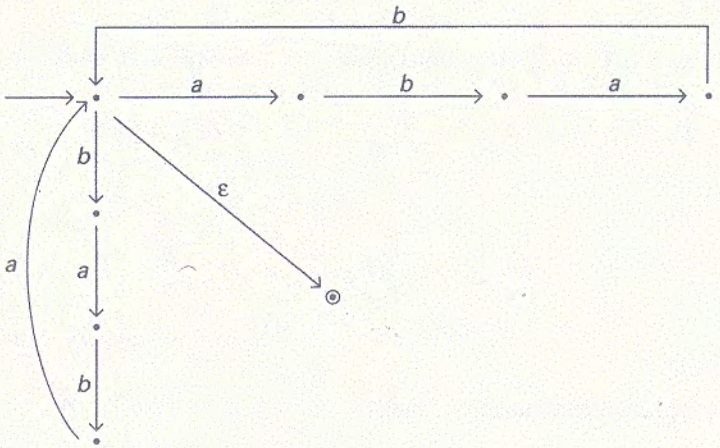


Figura 3.3

Teorema 3.2.1. *L* es regular si y sólo si es generado por una gramática regular.

Por tanto, tenemos tres métodos generales de especificación de lenguajes: las expresiones regulares, los autómatas finitos y las gramáticas regulares.

Ejercicios de la Sección 3.2.

3.2.1. Construir una gramática regular para el lenguaje regular aceptado por el autómata finito de la Figura 3.4.

3.2.2. Construir un AFN para la gramática regular

$$S \to aS \mid bB \mid b$$
$$B \to cC$$
$$C \to aS$$

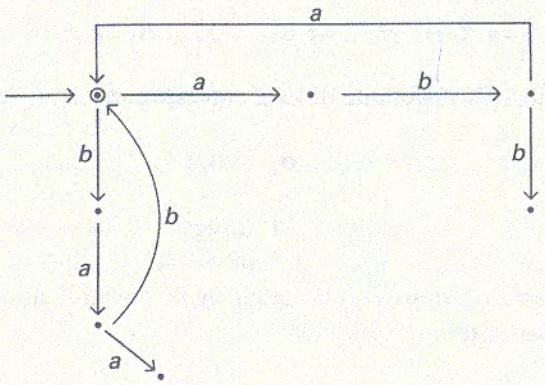


Figura 3.4

3.2.3. Construir un autómata finito para la gramática regular

$$S \rightarrow abA \mid B \mid baB \mid \varepsilon$$
$$A \rightarrow bS \mid b$$
$$B \rightarrow aS$$

3.2.4. Obtener una gramática regular para el lenguaje

$$L = \{w \in \{a, b\}^* \mid w \text{ no contiene la subcadena } aa\}$$

3.2.5. Obtener una gramática regular para $L = \{a^n b^n \mid n \geq 0\}$.

3.2.6. Una *producción regular por la izquierda* es una producción de la forma $A \rightarrow Bw$, donde A y B son no terminales y w es una cadena de terminales. Una *producción regular por la derecha* es una producción de la forma $A \rightarrow wB$. Por tanto, las gramáticas regulares por la izquierda (véase Ejercicio 3.1.5) y las gramáticas regulares por la derecha contienen solamente producciones regulares por la izquierda y producciones regulares por la derecha, respectivamente. Probar que una gramática regular no puede contener ambos tipos de producciones.

3.3 GRAMÁTICAS INDEPENDIENTES DEL CONTEXTO

Recordaremos que en nuestra definición de gramáticas regulares se requiere que el lado derecho de todas las producciones contenga al menos un no terminal. Es más, cuando un no terminal está presente, debe *aparecer* al final de la cadena (final izquierdo o final derecho, dependiendo de si es una gramática regular por la izquierda o por la derecha). Para expresar esto formalmente, se requiere que las producciones satisfagan $P \subseteq N \times \Sigma^* (N \cup \varepsilon)$ (o, en el caso de la regularidad por la izquierda, $P \subseteq N \times (N \cup \varepsilon) \Sigma^*$). Este requerimiento restringe en gran me-

dida la manera en la que se pueden formar las producciones y, en consecuencia, restringe las clases de lenguajes que se pueden especificar.

Supongamos que se permite que $P \subseteq N \times (N \cup \Sigma)^*$, de forma que las producciones puedan tener cero, uno o más no terminales que aparezcan en cualquier lugar del lado derecho de las mismas. Por ejemplo, la gramática dada por

$$S \rightarrow aB \mid bA$$
$$A \rightarrow a \mid aS \mid bAA$$
$$B \rightarrow b \mid bS \mid aBB$$

es una gramática de este tipo. Observe que, en definitiva, esta gramática *no* es una gramática regular. Por otro lado, todas las gramáticas regulares satisfacen este nuevo requerimiento en lo que respecta a la forma en la que se construyen las producciones y, por tanto, son gramáticas de este tipo. De esta forma, tendremos más de un tipo general de gramáticas.

Definición 3.3.1. Una *gramática independiente del contexto (GIC)* es una 4-tupla

$$G = (N, \Sigma, S, P)$$

donde N es una colección finita de no terminales, Σ es un alfabeto (también conocido como conjunto de terminales), S es un no terminal determinado que se llama símbolo inicial y $P \subseteq N \times (N \cup \Sigma)^*$ es un conjunto de producciones.

El *lenguaje generado por la GIC G* se denota por $L(G)$ y se llama *lenguaje independiente del contexto (LIC)*.

Por ejemplo, puesto que toda gramática regular es una GIC, se tiene que todo lenguaje regular es un LIC.

Al igual que una gramática regular, una GIC es una forma de probar cómo se generan cadenas en un lenguaje. Como con las gramáticas regulares, usaremos la notación $\Rightarrow$ para indicar el acto de generar como opuesto a $\rightarrow$, el cual es parte de una regla de producción. Cuando derivamos una cadena, los no terminales representan la parte de la cadena que todavía no se ha generado. En el caso de las gramáticas regulares, la parte de la cadena no generada siempre aparece al final. En las GIC que no son gramáticas regulares, puede haber más de un trozo no generado y pueden aparecer en cualquier lugar de la cadena. Cuando la derivación se completa, todos los trozos no generados habrán sido sustituidos por cadenas (posiblemente vacías) de símbolos terminales.

Consideremos la GIC dada por

$$S \rightarrow aSb \mid \varepsilon$$

La inducción sobre n prueba que esta gramática independiente del contexto genera el lenguaje independiente del contexto $\{a^n b^n \mid n \geq 0\}$. Por el Capítulo 2 sabemos que este lenguaje *no* es regular. Por tanto, hay lenguajes independientes del contexto que no son lenguajes regulares. Es decir, el conjunto de los lenguajes independientes del contexto contiene al conjunto de los lenguajes regulares.

Dedicaremos bastante tiempo al estudio de las gramáticas independientes del contexto y los lenguajes independientes del contexto. Sin embargo, antes de continuar, debemos mencionar otras formas de expresar las gramáticas regulares. Al generalizar, las gramáticas independientes del contexto, debemos eliminar todas las restricciones con respecto al lado derecho de las producciones, permitiendo que el mismo pueda estar formado por cualquier cadena sobre $N \cup \Sigma$. Lo único que debemos tener en cuenta en la generalización es la parte izquierda de las reglas de producción. Una *gramática de estructura de frase* es aquella en la que los lados izquierdos de las reglas de producción pueden estar formados por cualquier cadena no vacía sobre $N \cup \Sigma$, las cuales contienen algún no terminal. Por tanto, para una gramática de estructura de frase, la colección de reglas de producción P satisface

$$P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$$

Las gramáticas de estructura de frase se conocen como de *tipo 0* o gramáticas *no restringidas*.

El término independiente del contexto, cuando se aplica a gramáticas, sugiere que debería haber gramáticas que dependieran del contexto. Las *gramáticas dependientes del contexto* son gramáticas de estructura de frase, en las cuales las producciones se restringen a $\alpha \rightarrow \beta$, tal que $|\alpha| \leq |\beta|$. Hay una forma normal para estas gramáticas, en la cual toda producción es de la forma $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$ con $\beta \neq \varepsilon$. Tales producciones permiten que el no terminal A sea reemplazado por la cadena β, sólo cuando A aparezca en el "contexto" de α_1 y α_2.

Las gramáticas dependientes del contexto no pueden generar tantos lenguajes como las gramáticas de estructura de frase, aunque permiten que las derivaciones se realicen de forma predecible. Sin embargo, obsérvese que, puesto que $|S| = 1$ y $|\varepsilon| = 0$, es imposible derivar la cadena vacía en una gramática que sea verdaderamente dependiente del contexto. A menudo, los lenguajes de programación se crean para ser dependientes del contexto con el fin de simplificar el proceso de la compilación.

Ejercicios de la Sección 3.3

3.3.1. Dada la siguiente gramática independiente del contexto

$$S \rightarrow AA$$
$$A \rightarrow AAA \mid a \mid bA \mid Ab$$

(a) Obtener una derivación para la cadena b^2aba^2ba.

(b) Probar cómo puede obtenerse una derivación para $b^{m_1} ab^{m_2} a \ldots b^{m_{2n}} ab^{m_{2n+1}}$, para todo $n > 0$ y $m_1, m_2, \ldots, m_{2n+1} \geq 0$.

3.3.2. La gramática G independiente del contexto dada por

$$S \rightarrow aSb \mid aSa \mid bSa \mid bSb \mid \varepsilon$$

no es una gramática regular, aunque $L(G)$ es un lenguaje regular! Obtener una gramática regular G' tal que $L(G') = L(G)$.

3.3.3. Obtener una gramática independiente del contexto para cada uno de los siguientes lenguajes independientes de contexto:

(a) $\{a^m b^n \mid m \geq n\}$

(b) $\{w \in \{a, b\}^* \mid w$ tiene el doble de *aes* que de *bes*$\}$

(c) $\{a^m b^n \mid n \leq m \leq 2n\}$

(d) $\{a^m b^n c^p d^q \mid m + n \geq p + q\}$

3.4 ÁRBOLES DE DERIVACIÓN O DE ANÁLISIS Y AMBIGÜEDAD

Cuando una cadena se deriva mediante una gramática independiente del contexto, el símbolo inicial es sustituido por alguna cadena. Los no terminales de esta cadena son sustituidos uno tras otro por otra cadena, y así sucesivamente, hasta que se llega a una cadena formada sólo por símbolos terminales. No se puede realizar ninguna sustitución más, puesto que no hay no terminales que puedan ser sustituidos. A veces, es útil realizar un gráfico de la derivación, que indique de qué manera ha contribuido cada no terminal a formar la cadena final de símbolos terminales. Tal gráfico tiene forma de árbol y se llama *árbol de derivación* (o *árbol de análisis*).

Un árbol de derivación para una derivación dada se construye creando un nodo raíz que se etiqueta con el símbolo inicial. El nodo raíz tiene unos nodos hijos para cada símbolo que aparezca en el lado derecho de la producción usada para reemplazar el símbolo inicial. Todo nodo etiquetado con un no terminal también tiene unos nodos hijos etiquetados con los símbolos del lado derecho de

la producción usada para sustituir ese no terminal. Los nodos que no tienen hijos deben ser etiquetados con símbolos terminales.

Consideremos la gramática independiente del contexto

$$S \rightarrow AB$$
$$A \rightarrow aA \mid a$$
$$B \rightarrow bB \mid b$$

La cadena *aabbb* puede ser derivada mediante

$$S \Rightarrow AB \Rightarrow AbB \Rightarrow AbbB \Rightarrow Abbb \Rightarrow aAbbb \Rightarrow aabbb$$

En la Figura 3.5 se presenta un árbol de derivación para esta derivación. Comenzamos en la raíz S y generamos los hijos A y B. A y B son raíz del subárbol correspondiente a la parte de la cadena final que ellos generan. Obsérvese que todos los nodos hoja están etiquetados con símbolos terminales. Si se leen las hojas de izquierda a derecha, se obtiene la cadena *aabbb*.

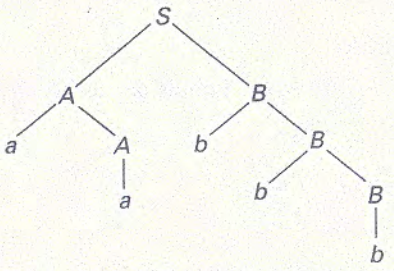


Figura 3.5

Finalmente, obsérvese que hay muchas derivaciones posibles para la cadena *aabbb*, las cuales también tienen el árbol de derivación anterior. Por ejemplo,

$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aaB \Rightarrow aabB \Rightarrow aabbB \Rightarrow aabbb$$

y

$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aAbB \Rightarrow aAbbB \Rightarrow aAbbb \Rightarrow aabbb$$

Para esta cadena y esta gramática, todas las derivaciones de *aabbb* tienen el mismo árbol de derivación. Sin embargo, no tiene porque cumplirse siempre. Para verlo, considérese esta gramática

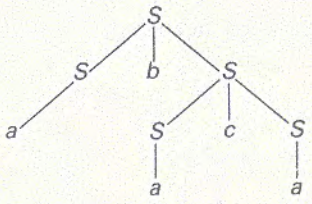
$$S \rightarrow SbS \mid ScS \mid a$$

Podemos derivar la cadena *abaca* de dos formas distintas como sigue:

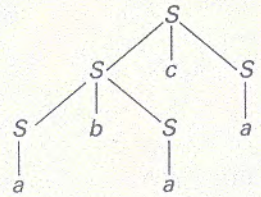
1. $S \Rightarrow SbS \Rightarrow SbScS \Rightarrow SbSca \Rightarrow Sbaca \Rightarrow abaca$

2. $S \Rightarrow ScS \Rightarrow SbScS \Rightarrow abScS \Rightarrow abacS \Rightarrow abaca$

El árbol de derivación para la derivación 1 es



mientras que el árbol para la derivación 2 es



Obsérvese que los dos árboles son distintos, aunque las cadenas producidas son la misma. (La cadena derivada corresponde a los nodos hoja y se llama *producto* del árbol de derivación).

Una gramática se dice que es *ambigua* si hay dos o más árboles de derivación distintos para la misma cadena. Una gramática en la cual, para toda cadena w, todas las derivaciones de w tienen el mismo árbol de derivación, es *no ambigua*.

La ambigüedad puede ser un problema para ciertos lenguajes en los que su significado depende, en parte, de su estructura, como ocurre con los lenguajes naturales y los lenguajes de programación. Si la estructura de un lenguaje tiene más de una descomposición y si la construcción parcial determina su significado, entonces el significado es ambiguo. Consideremos la sentencia "Juan vio a un hombre con un telescopio". El significado de esta sentencia es ambiguo debido a que "con un telescopio" puede describir al hombre que vio Juan o a la técnica que Juan empleó para ver al hombre.

Consideremos otro ejemplo de ambigüedad que obscurece el significado de las expresiones. Sea la siguiente gramática de asignaciones de expresiones:

$$A \rightarrow I := E$$
$$I \rightarrow a \mid b \mid c$$
$$E \rightarrow E + E \mid E * E \mid (E) \mid I$$

Los símbolos terminales han sido subrayados.

La cadena $a := b + c * a$ es una cadena de este lenguaje de sentencias de asignación. Hay dos árboles de derivación distintos para ella (véase Figura 3.6)

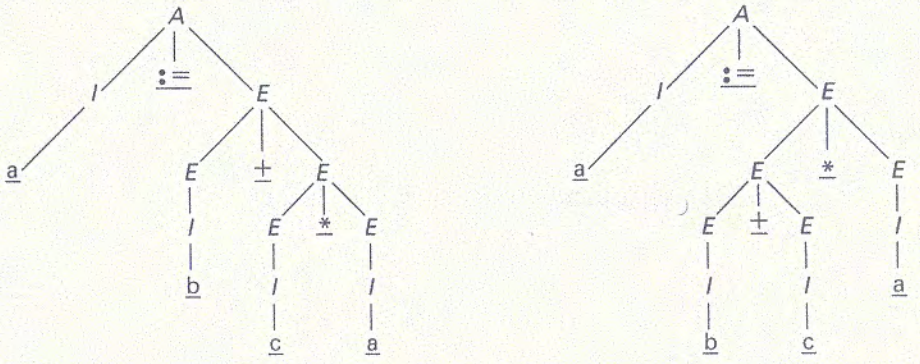


Figura 3.6

Si pretendemos determinar cómo se calcula el valor de la derecha del operador de asignación (el símbolo :=), se obtienen dos resultados posibles, $b + (c * a)$ o $(b + c) * a$. En general, estos resultados no son iguales.

En algunos casos, si la gramática es ambigua, se puede encontrar otra gramática que produzca el mismo lenguaje pero que no sea ambigua. Por ejemplo, la gramática

$$S \rightarrow A \mid B$$
$$A \rightarrow a$$
$$B \rightarrow a$$

es ambigua porque tiene dos árboles de derivación para la cadena a. Una gramática equivalente que no es ambigua es

$$S \rightarrow a$$

Si todas las gramáticas independientes del contexto para un lenguaje son ambiguas, se dice que el lenguaje es *un lenguaje independiente del contexto inherentemente ambiguo*. El lenguaje

$$L = \{a^i b^j c^k \mid i = j \text{ o } j = k\}$$

es inherentemente ambiguo. Intuitivamente, una gramática para L debe tener una clase de árbol de derivación para generar las cadenas para las cuales $i = j$ y otro para las cadenas en las cuales $j = k$. Si una cadena tiene $i = j = k$, tendrá dos derivaciones.

Vimos anteriormente que una cadena dada no puede tener más de una derivación igual en una gramática independiente del contexto no ambigua. Las derivaciones distintas corresponden a la elección de distintos no terminales a expandir. Por convención, dos formas de generar una cadena tienen una única salida. En una *derivación por la izquierda* el no terminal que se expande es, siempre, el del extremo izquierdo. Por tanto para la gramática

$$S \rightarrow SbS|ScS|a$$

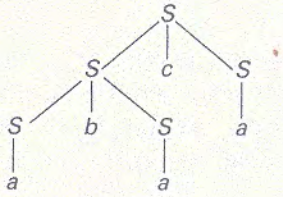
una derivación por la izquierda de *abaca* será

$$S \Rightarrow ScS \Rightarrow SbScS \Rightarrow abScS \Rightarrow abacS \Rightarrow abaca$$

Una *derivación por la derecha* es aquella en la cual el no terminal que se expande es el del extremo derecho. Por tanto

$$S \Rightarrow ScS \Rightarrow Sca \Rightarrow SbSca \Rightarrow Sbaca \Rightarrow abaca$$

es una derivación por la derecha de la cadena dada. Obsérvese que las dos derivaciones tienen el mismo árbol de derivación:



Este árbol de derivación también es compartido por otras derivaciones. En esta gramática

$$S \Rightarrow SbS \Rightarrow abS \Rightarrow abScS \Rightarrow abacS \Rightarrow abaca$$

es una derivación por la izquierda distinta de la precedente. La presencia de dos derivaciones por la izquierda distintas se corresponde con la existencia de dos árboles de derivación distintos. Por tanto, una gramática ambigua se caracteriza por tener dos (o más) derivaciones por la izquierda para la misma cadena.

Nos gustaría establecer las restricciones necesarias para que las producciones se formen de manera que el árbol de derivación resultante no sea necesariamente complejo o inútilmente sencillo. A la vez, no se pretende constreñir la formación de producciones hasta el punto de que no se pueda generar ningún lenguaje independiente del contexto a partir de los conjuntos de producciones que cumplan las restricciones. Pretendemos encontrar un modelo formal estándar (o una forma normal) para las producciones.

Como primer paso en el desarrollo del modelo, necesitamos limpiar las gramáticas para eliminar las producciones y símbolos inútiles. Consideremos la gramática independiente del contexto del ejemplo siguiente:

Ejemplo 3.5.1

$$S \to Aa \mid B \mid D$$
$$B \to b$$
$$A \to aA \mid bA \mid B$$
$$C \to abd$$

Obsérvese que C nunca formará parte de una derivación que parta del símbolo inicial, es decir, no existe $S \overset{*}{\Rightarrow} \alpha C \beta$ para toda cadena α y β de $(N \cup \Sigma)^*$. Por consiguiente, el símbolo C y la producción

$$C \to abd$$

son inútiles en el sentido de que nunca podrán contribuir a la generación de una cadena de $L(G)$. El símbolo D se obtiene a partir de S pero nunca deriva una cadena de símbolos terminales y, por tanto, nunca formará parte de una derivación de una cadena de terminales. Por tanto, D también es inútil, aunque por distinta razón. Por otro lado, el símbolo B es, en cierto modo, redundante, ya que deriva únicamente un símbolo terminal; por tanto, las derivaciones $S \Rightarrow B \Rightarrow b$ y $A \Rightarrow B \Rightarrow b$ podrían ser reducidas de forma que $S \Rightarrow b$ y $A \Rightarrow b$, y eliminándose B. Finalmente, obsérvese que si se elimina la producción $C \to abd$, el símbolo terminal d no puede aparecer en ninguna cadena de terminales generada mediante la gramática resultante.

Por tanto, hemos identificado muchos aspectos en los que las gramáticas independientes del contexto pueden ser depuradas. Cualquiera de esos problemas pueden ser eliminados sin afectar a la capacidad de generación de la gramática. Primero se eliminarán los no terminales que no deriven cadenas de terminales, tales como D en el Ejemplo 3.5.1.

Sea $G = (N, \Sigma, S, P)$ una gramática independiente del contexto. Transformaremos G en $G' = (N', \Sigma, S, P')$ de forma que $L(G) = L(G')$ y, para todo $A \in N'$,

se obtenga que $A \stackrel{*}{\Rightarrow} w$ para algún $w \in \Sigma^*$. Para realizarlo, construiremos iterativamente el nuevo conjunto de no terminales N' y el nuevo conjunto de producciones P' como sigue:

Algoritmo 3.5.1.

1. Inicializar N' con todos los no terminales A para los que $A \rightarrow w$, es una producción de G, con $w \in \Sigma^*$.
2. Inicializar P' con todas las producciones $A \rightarrow w$ para las cuales $A \in N'$ y $w \in \Sigma^*$.
3. Repetir

   Añadir a N' todos los no terminales A para los cuales $A \rightarrow w$, para algún $w \in (N' \cup \Sigma)^*$ que sea una producción de P y añadirla a P'.

   hasta que no se puedan añadir más no terminales a N'.

Obsérvese que el bucle del paso 3 termina, ya que N y P son finitos. Esencialmente, lo que estamos haciendo es recorrer hacia arriba todos los posibles árboles de análisis a partir de las cadenas de terminales, anotando los no terminales (y las producciones) que se encuentren. Todo no terminal (y producción) que no aparezca en N', no contribuirá a formar una subcadena de cualquier cadena "final" de terminales que sea generada por la gramática. Por tanto, su eliminación no altera el lenguaje generado.

Por ejemplo, en la gramática del Ejemplo 3.5.1, esperábamos eliminar el no terminal D. Después de aplicar el Algoritmo 3.5.1 a esta gramática, obtendremos la siguiente:

$$S \rightarrow Aa \mid B$$
$$A \rightarrow aA \mid bA \mid B$$
$$B \rightarrow b$$
$$C \rightarrow abd$$

Las gramáticas independientes del contexto se han definido de forma que se permite que existan producciones que tengan ε en el lado derecho. El Algoritmo 3.5.1 trata ε como una cadena de un terminal. Por tanto, al transformar la gramática

$$S \rightarrow aA \mid \varepsilon$$
$$A \rightarrow aA \mid bB \mid \varepsilon$$
$$B \rightarrow bB$$

se obtendrá la gramática

$$S \rightarrow aA \mid \varepsilon$$
$$A \rightarrow aA \mid \varepsilon$$

Las producciones de la forma $A \rightarrow \varepsilon$ se llaman *producciones* ε. A veces son necesarias, pero otras no son ni necesarias ni deseables. Podremos eliminarlas cuando no sean necesarias.

A menudo, tras la transformación de una gramática, nos quedan producciones como $C \rightarrow abd$ que no se usan. La razón por la cual dicha producción se encuentra en la gramática transformada es que, de la aplicación de dicha producción, se obtiene una cadena de terminales. Sin embargo, su presencia no es deseable ya que nunca se podrá derivar una cadena a partir del símbolo S que contenga el no terminal C. Es más, puesto que el símbolo terminal d sólo aparece en una cadena que se obtenga a partir de dicha producción, puede ser eliminado del alfabeto sin que el lenguaje generado sea alterado. El siguiente algoritmo elimina aquellos terminales y no terminales que no aparezcan en las cadenas que se deriven a partir de S. La gramática transformada resultante garantiza que un símbolo X será un terminal o no terminal de dicha gramática si y sólo si $S \overset{*}{\Rightarrow} \alpha X \beta$ para algunas cadenas α y β sobre $(N \cup \Sigma)^*$.

Sea $G = (N, \Sigma, S, P)$ una gramática independiente del contexto. Transformaremos G en la gramática $G' = (N', \Sigma', S, P')$ de forma que $L(G) = L(G')$ y para todo $X \in N' \cap \Sigma'$, se tenga que $S \overset{*}{\Rightarrow} \alpha X \beta$ para las cadenas α y β de $(N' \cap \Sigma')^*$. Para realizarlo, se construirán iterativamente los conjuntos de terminales, no terminales y producciones de la manera siguiente:

Algoritmo 3.5.2.

1. Inicializar N' de forma que contenga el símbolo inicial S, e inicializar P' y Σ' a $\emptyset$.
2. Repetir

   Para $A \in N'$, si $A \rightarrow w$ es una producción de P, entonces:

   1. Introducir $A \rightarrow w$ en P'.
   2. Para todo no terminal B de w, introducir B en N'.
   3. Para todo no terminal σ de w, introducir σ en Σ'.

   hasta que no se puedan añadir nuevas producciones.

Obsérvese que, puesto que P, N y Σ son finitos, el bucle de la etapa 2 siempre termina. El algoritmo ha sido diseñado para tener en cuenta todos los terminales y no terminales que sean accesibles desde S. Si un terminal o un no terminal no puede ser conseguido a partir de S, nunca será incluido en N' o Σ'. Si un

no terminal no es accesible, entonces todas las producciones que lo tengan en su lado izquierdo serán también excluidas.

Consideremos la gramática del Ejemplo 3.5.1 que fue transformada mediante el Algoritmo 3.5.1 en

$$S \to Aa \mid B$$
$$B \to b$$
$$A \to aA \mid bA \mid B$$
$$C \to abd$$

La gramática obtenida al aplicar el Algoritmo 3.5.2 es

$$S \to Aa \mid B$$
$$A \to aA \mid bA \mid B$$
$$B \to b$$

Obsérvese que la producción $C \to abd$ ha sido eliminada, junto con el no terminal C y el terminal d.

Hay que tener en cuenta que es importante el orden en el que los dos algoritmos precedentes son aplicados a una gramática. Consideremos la gramática

$$S \to AB \mid a$$
$$A \to a$$

Si aplicamos el Algoritmo 3.5.1 antes que el Algoritmo 3.5.2 obtendremos un resultado distinto al que obtendríamos si aplicamos primero el Algoritmo 3.5.2 y después el Algoritmo 3.5.1.

$$\left.\begin{array}{l} S \to AB \mid a \\ A \to a \end{array}\right\} \xrightarrow{\text{Alg. 3.5.1}} \left.\begin{array}{l} S \to a \\ A \to a \end{array}\right\} \xrightarrow{\text{Alg. 3.5.2}} S \to a$$

$$\left.\begin{array}{l} S \to AB \mid a \\ A \to a \end{array}\right\} \xrightarrow{\text{Alg. 3.5.2.}} \left.\begin{array}{l} S \to AB \mid a \\ A \to a \end{array}\right\} \xrightarrow{\text{Alg. 3.5.1}} \begin{array}{l} S \to a \\ A \to a \end{array}$$

Ahora dirigiremos nuestra atención a las producciones ε. Dichas producciones son de la forma $A \to \varepsilon$. Indudablemente, si $\varepsilon \in L(G)$, no podremos eliminar tales producciones para que ε pueda ser generado por la gramática. De esto se deduce que, si ε no está en $L(G)$, todas las producciones ε pueden ser eliminadas.

Se dice que un no terminal A es *anulable* si $A \overset{*}{\Rightarrow} \varepsilon$. Para la eliminación de las producciones ε, es crucial identificar los no terminales anulables. El siguien-

te algoritmo identifica el conjunto $\mathcal{N}$, de todos los no terminales anulables en una gramática independiente del contexto $G = (N, \Sigma, S, P)$.

Algoritmo 3.5.3.

1. Inicializar $\mathcal{N}$ con todos los no terminales A para los cuales existe una producción ε, $A \to \varepsilon$.
2. Repetir:

   Si $B \to w$ para algún $w \in (N \cup \Sigma)^*$ y todos los símbolos de w están en $\mathcal{N}$, añadir B a $\mathcal{N}$.

   hasta que no se añadan más no terminales a $\mathcal{N}$.

Por ahora, sólo nos ocuparemos de las gramáticas independientes del contexto $G = (N, \Sigma, S, P)$, para las que $L(G)$ no contiene a ε. Una vez que han sido identificados los no terminales anulables, se modifican las reglas de producción con el fin de poder eliminar las producciones ε. Esto se realiza sustituyendo producciones de la forma $B \to X_1 X_2 \ldots X_n$ por las producciones que se formen al eliminar los subconjuntos de X_i que son anulables. Se debe tener cuidado en no incluir $B \to \varepsilon$, incluso si todos los X_i son anulables.

Se crea el nuevo conjunto de producciones P' como sigue:

Si $B \to X_1 X_2 \ldots X_n$ es una producción de P, entonces en P' introduciremos todas las producciones de la forma $B \to Y_1 Y_2 \ldots Y_n$, donde las Y_i satisfagan:

$Y_i = X_i$ si X_i no es anulable.

$Y_i = X_i$ o ε si X_i es anulable.

Y_i no es ε para todo i (es decir, no se introduce en P' ninguna producción de la forma $B \to \varepsilon$)

Es importante señalar que a partir de una producción $B \to X_1 X_2 \ldots X_n$ de P, se pueden conseguir nuevas producciones en P'. Por ejemplo, si $B \to X_1 X_2$ y tanto X_1 como X_2 son anulables, se podrían obtener las producciones

$$B \to X_1 \,|\, X_2 \,|\, X_1 X_2$$

Consideremos la gramática G:

$$S \to aA$$
$$A \to aA \,|\, \varepsilon$$

Obsérvese que A es el único no terminal anulable (y que $\varepsilon \notin L(G)$). Si consideramos la producción $S \rightarrow aA$, tendremos que $X_1 = a$ y $X_2 = A$. Por tanto, añadiremos a la nueva colección las producciones $S \rightarrow a \mid aA$. La gramática que resulta, tras considerar todas las producciones originales, será

$$S \rightarrow aA \mid a$$
$$A \rightarrow aA \mid a$$

Obsérvese que se ha eliminado la producción ε, $A \rightarrow \varepsilon$.

En una gramática independiente del contexto, si $L(G)$ contiene ε, se pueden eliminar todas las producciones ε de G menos una. Primero, se eliminan todas las producciones ε de G. Esto transformará la gramática G en G' para la cual $L(G') = L(G) - \{\varepsilon\}$. Después se añade la producción $S \rightarrow \varepsilon$, la cual restituirá ε al lenguaje generado.

Las producciones de la forma $A \rightarrow B$, donde A y B son no terminales, se llaman producciones *unitarias* o *no generativas*. La presencia de producciones unitarias no indica, necesariamente, que un símbolo es inútil. Sin embargo, las producciones unitarias *hacen* que la gramática independiente del contexto sea innecesariamente compleja.

Por ejemplo, la aplicación de una producción de la forma $A \rightarrow B$ simplemente renombra un no terminal y añade un paso más a la derivación. Cualquier cadena que sea derivable a partir de B también lo será a partir de A. Por tanto, se puede eliminar ese paso extra saltando por encima de B. Por ejemplo, si las producciones de A y B son

$$A \rightarrow B$$
$$B \rightarrow w_1 \mid C$$

donde $C \in N$ y $w_1 \in (N \cup \Sigma)^*$, podemos eliminar la producción $A \rightarrow B$ e incluir la producción $A \rightarrow w_1 \mid C$. Obsérvese que, al eliminar la producción unitaria $A \rightarrow B$, se introduce la producción unitaria $A \rightarrow C$. Se podría repetir este proceso hasta que no existiera ninguna producción unitaria en la gramática, pero se podría realizar otro planteamiento con el fin de eliminar la circularidad de este proceso.

Obsérvese que, en el proceso precedente, la producción unitaria $A \rightarrow C$ se obtiene como resultado de las producciones $A \rightarrow B$ y $B \rightarrow C$ de la gramática original. Si conocemos todos los no terminales X tales que $A \overset{*}{\Rightarrow} X$, solamente mediante producciones unitarias, se podría entonces evitar introducirlas repetidamente e ir eliminando dichas producciones unitarias una a una. Para ver cómo se puede realizar este proceso, supongamos que tenemos las producciones

$$A \to B$$
$$B \to C \mid w_1$$
$$C \to D$$
$$D \to w_2$$

Entonces se tiene que $A \Rightarrow B \Rightarrow C \Rightarrow D$. Obsérvese que las producciones $A \to w_1 \mid w_2$ permiten que de A se deriven las mismas cadenas que se derivaban con las cinco producciones originales. Las nuevas producciones se obtienen a partir de las producciones no unitarias del conjunto de producciones original, creando una producción $A \to y$ para toda producción no unitaria $X \to y$, donde $X \in \{B, C, D\}$. A continuación presentaremos esta técnica de una forma más precisa.

Primero, para $A \in N$ se define

Unitario $(A) = \{B \in N \mid A \overset{*}{\Rightarrow} B$ usando solamente producciones unitarias$\}$

(Obsérvese que $A \in$ Unitario (A) puesto que $A \overset{*}{\Rightarrow} A$ mediante 0 producciones). Sea $G = (N, \Sigma, S, P)$ una gramática independiente del contexto que no tenga producciones ε. Construiremos una gramática independiente del contexto equivalente $G' = (N, \Sigma, S, P')$ en la que P' no contenga producciones unitarias, como se describe a continuación:

1. Inicializar P' de forma que contenga todos los elementos de P.
2. Para cada $A \in N$, obtener el conjunto Unitario (A).
3. Para cada A para el cual Unitario $(A) \neq \{A\}$
   Para cada $B \in$ Unitario (A)
   Para cada producción no unitaria $B \to w$ de P
   añadir $A \to w$ a P'.
4. Eliminar todas las producciones unitarias que haya en P'.

Por ejemplo, en la gramática

$$S \to A \mid Aa$$
$$A \to B$$
$$B \to C \mid b$$
$$C \to D \mid ab$$
$$D \to b$$

tenemos

$$\text{Unitario}(S) = \{S, A, B, C, D\}$$
$$\text{Unitario}(A) = \{A, B, C, D\}$$
$$\text{Unitario}(B) = \{B, C, D\}$$
$$\text{Unitario}(C) = \{C, D\}$$
$$\text{Unitario}(D) = \{D\}$$

El algoritmo introduce primero las producciones

$$S \to b \mid ab$$
$$A \to b \mid ab$$
$$B \to ab \mid b$$
$$C \to b \mid ab$$

y entonces se eliminan las producciones $S \to A$, $A \to B$, $B \to C$ y $C \to D$. La gramática resultante es

$$S \to b \mid ab \mid Aa$$
$$A \to b \mid ab$$
$$B \to ab \mid b$$
$$C \to b \mid ab$$
$$D \to b$$

Esta gramática puede simplificarse más, por medio de las otras técnicas.

En lo visto anteriormente, hemos realizado la eliminación de producciones en gramáticas independientes del contexto de una forma bastante incómoda. Como última etapa en la simplificación de gramáticas independientes del contexto, presentaremos un modelo o forma normal para las producciones. Se dice que una gramática independiente del contexto está en *forma normal de Chomsky* si no contiene producciones ε y si todas las producciones son de la forma $A \to a$, para $a \in \Sigma$, o de la forma $A \to BC$, donde B y C son no terminales. Es decir, en la forma normal de Chomsky el lado derecho de cada producción contiene un único símbolo terminal o una par de no terminales. Obsérvese que, para una gramática en forma normal de Chomsky, el árbol de derivación para cualquier derivación está bastante bien construido ya que, excepto en las hojas, ¡el árbol es binario!

Si G es una gramática independiente del contexto y $\varepsilon \notin L(G)$, G puede ser transformada en una gramática en forma normal de Chomsky. Para ello, primero se eliminan todas las producciones ε, los símbolos inútiles y las producciones unitarias de G. Obsérvese que, una vez que se ha realizado lo anterior, si $A \to w$ es una producción de G, se puede asegurar que $|w| \geq 1$. Es más, si $|w| = 1$, entonces w es un símbolo terminal de Σ, puesto que no hay producciones unitarias.

Por otro lado, si $|w| > 1$, entonces w puede contener tanto terminales como no terminales. Ahora transformaremos G convirtiendo tales w en cadenas que contengan sólo no terminales.

Supongamos que tenemos una producción de la forma $A \rightarrow w$, donde $w = X_1 X_2 \ldots X_n$. Si X_i es un símbolo terminal, llamado σ, sustituiremos X_i por un nuevo no terminal C_σ y añadiremos la producción $C_\sigma \rightarrow \sigma$. Una vez que se aplica a G está conversión, todas las producciones son de la forma $A \rightarrow w$, donde w es un símbolo terminal o una cadena formada sólo por no terminales.

La última etapa para la transformación de G en forma normal de Chomsky, consiste en eliminar las cadenas con más de dos no terminales que se encuentren en el lado derecho de una producción. Para ello, si $A \rightarrow B_1 B_2 \ldots B_n$ es una producción con $n \geq 2$, la reemplazaremos por $n - 1$ producciones

$$\begin{aligned} A &\rightarrow B_1 D_1 \\ D_1 &\rightarrow B_2 D_2 \\ &\vdots \\ D_{n-2} &\rightarrow B_{n-1} B_n \end{aligned}$$

En ellas, los D_i serán nuevos no terminales. En la gramática transformada resultante, el lado derecho de cada producción está compuesto por un único terminal o por una cadena de dos no terminales. Por lo tanto, todo lenguaje independiente del contexto que no contenga ε puede ser generado mediante una gramática independiente del contexto en forma normal de Chomsky.

Por ejemplo, consideremos la GIC

$$\begin{aligned} S &\rightarrow bA \mid aB \\ A &\rightarrow bAA \mid aS \mid a \\ B &\rightarrow aBB \mid bS \mid b \end{aligned}$$

Obsérvese que esta gramática no contiene producciones ε, producciones unitarias ni símbolos inútiles. Después de la primera transformación, la gramática se convierte en

$$\begin{aligned} S &\rightarrow C_b A \mid C_a B \\ A &\rightarrow C_b AA \mid C_a S \mid a \\ B &\rightarrow C_a BB \mid C_b S \mid b \\ C_a &\rightarrow a \\ C_b &\rightarrow b \end{aligned}$$

En esta versión, el lado derecho de todas las producciones está formado por un único símbolo terminal o por una cadena de dos o más no terminales. Después de la última conversión, la forma normal de Chomsky de la gramática será

$$
\begin{aligned}
S &\rightarrow C_b A \mid C_a A \\
A &\rightarrow C_b D_1 \mid C_a S \mid a \\
D_1 &\rightarrow AA \\
B &\rightarrow C_a D_2 \mid C_b S \mid b \\
D_2 &\rightarrow BB
\end{aligned}
$$

Si L es un lenguaje independiente del contexto que contiene ε, se puede obtener un gramática independiente del contexto en forma normal de Chomsky para $L - \{\varepsilon\}$ y después añadir a la misma, la producción $S \rightarrow \varepsilon$. La gramática resultante estará en forma normal de Chomsky exceptuando la producción ε.

Ejercicios de la Sección 3.5

3.5.1. Aplicar el Algoritmo 3.5.1 a las siguientes gramáticas:

(a)

$$
\begin{aligned}
S &\rightarrow aAb \mid cEB \mid CE \\
A &\rightarrow dBE \mid eeC \\
B &\rightarrow ff \mid D \\
C &\rightarrow gFB \mid ae \\
D &\rightarrow h
\end{aligned}
$$

(b)

$$
\begin{aligned}
S &\rightarrow aB \\
A &\rightarrow bcCCC \mid dA \\
B &\rightarrow e \\
C &\rightarrow fA \\
D &\rightarrow Dgh
\end{aligned}
$$

3.5.2. Aplicar el Algoritmo 3.5.1 a la siguiente gramática:

$$
\begin{aligned}
S &\rightarrow a \mid aA \mid B \mid C \\
A &\rightarrow aB \mid \varepsilon \\
B &\rightarrow Aa \\
C &\rightarrow bCD \\
D &\rightarrow ccc
\end{aligned}
$$

3.5.3. Aplicar el Algoritmo 3.5.2 a la siguiente gramática independiente del contexto:

$$S \to aAb$$
$$A \to ccC$$
$$B \to dd \mid D$$
$$C \to ae$$
$$D \to f$$
$$U \to gW$$
$$W \to h$$

3.5.4. Aplicar el Algoritmo 3.5.2 a la siguiente gramática independiente del contexto:

$$S \to a \mid aA \mid B$$
$$A \to aB \mid \varepsilon$$
$$B \to Aa$$
$$D \to ddd$$

3.5.5. Eliminar los símbolos inútiles de la siguiente gramática por medio de los Algoritmos 3.5.1 y 3.5.2:

$$S \to A \mid AA \mid AAA$$
$$A \to ABa \mid ACa \mid a$$
$$B \to ABa \mid Ab \mid \varepsilon$$
$$C \to Cab \mid CC$$
$$D \to CD \mid Cd \mid CEa$$
$$E \to b$$

3.5.6. Obtener la colección de no terminales anulables que pertenecen a la siguiente gramática:

$$S \to aA \mid bA \mid a$$
$$A \to aA \mid bAb \mid \varepsilon$$

3.5.7. Obtener, para la siguiente gramática, el número de no terminales anulables:

$$S \to ABaC$$
$$A \to AB$$
$$B \to b \mid \varepsilon$$
$$C \to D \mid \varepsilon$$
$$D \to d$$

3.5.8. Eliminar las producciones ε de la gramática:

$$S \to aA \mid bA \mid a$$
$$A \to aA \mid bAb \mid \varepsilon$$

3.5.9. Eliminar de la siguiente gramática las producciones ε:

$$S \to AB$$
$$A \to aA \mid abB \mid aCa$$
$$B \to bA \mid BB \mid \varepsilon$$
$$C \to \varepsilon$$
$$D \to dB \mid BCB \mid B \mid BB$$

3.5.10. Eliminar de la siguiente gramática las producciones ε:

$$S \to a \mid aA \mid B$$
$$A \to aB \mid \varepsilon$$
$$B \to Aa$$

3.5.11. Eliminar las producciones ε de la gramática siguiente:

$$S \to ABaC$$
$$A \to AB$$
$$B \to b \mid \varepsilon$$
$$C \to D \mid \varepsilon$$
$$D \to d$$

3.5.12. Simplificar la siguiente gramática tanto como sea posible

$$S \to aB \mid aaB$$
$$A \to \varepsilon$$
$$B \to bA$$
$$B \to \varepsilon$$

3.5.13. El lenguaje asociado con la siguiente gramática independiente del contexto contiene ε. Eliminar las producciones ε excepto $S \to \varepsilon$.

$$S \to AB \mid aB \mid \varepsilon$$
$$A \to BBB \mid aB \mid a \mid \varepsilon$$
$$B \to a \mid aA \mid \varepsilon$$

3.5.14. Realizar una algoritmo para construir Unitario (A) siendo A un no terminal de una GIC.

3.5.15. Eliminar todas las producciones unitarias de la siguiente gramática independiente del contexto:

$$S \to CBa \mid D$$
$$A \to bbC$$
$$B \to Sc \mid ddd$$
$$C \to eA \mid f \mid C$$
$$D \to E \mid SABC$$
$$E \to gh$$

3.5.16. Eliminar todas las producciones unitarias de la siguiente gramática independiente del contexto:

$$S \to Aa \mid Ba \mid B$$
$$A \to Aa \mid \varepsilon$$
$$B \to aA \mid BB \mid \varepsilon$$

[Obsérvese que $\varepsilon \in L(G)$].

3.5.17. Convertir las siguientes gramáticas a forma normal de Chomsky:

(a)
$$S \to AB \mid CA$$
$$A \to a$$
$$B \to BC \mid AB$$
$$C \to aB \mid b$$

(b)
$$S \to aAb \mid cHB \mid CH$$
$$A \to dBH \mid eeC$$
$$B \to ff \mid D$$
$$C \to gFB \mid ah$$
$$D \to i$$
$$E \to jF$$
$$F \to dcGGG \mid cF$$
$$G \to kF$$
$$H \to Hlm$$

3.5.18. Probar que, al realizar la conversión a forma normal de Chomsky, se puede elevar al cuadrado el número de las producciones de una gramática independiente del contexto.

3.6 PROPIEDADES DE LOS LENGUAJES INDEPENDIENTES DEL CONTEXTO

Las gramáticas en forma normal de Chomsky nos permiten obtener la relación que existe entre la longitud de una cadena y el número de pasos de su derivación. Si se deriva ε, se obtiene a partir de una única producción $S \rightarrow \varepsilon$. Se puede probar, mediante inducción, que si se puede derivar w y $|w| > 0$, entonces la derivación tiene exactamente $2|w|$ etapas. Por etapa, entendemos una sustitución. Escribir el símbolo inicial es una etapa, así como lo es sustituir a él o a cualquier otro terminal por el lado derecho de una producción.

Supongamos que G es una gramática independiente del contexto en forma normal de Chomsky y consideremos el árbol de derivación para una cadena cualquiera de $L(G)$. Si un nodo tiene dos hijos, entonces los nodos hijos serán etiquetados con un no terminal y podremos tener al menos dos hijos más del mismo. (Esto es debido a que G está en forma normal de Chomsky). Es decir, el nodo raíz puede tener dos hijos, cada uno de sus hijos puede tener dos hijos y así sucesivamente. En cada nivel se pueden duplicar el número de nodos con respecto al anterior. Por tanto, en el nivel k podremos tener 2^k nodos. (El nivel 0 es el nivel del nodo raíz).

Por otro lado, si un nodo tiene un único hijo, entonces dicho nodo hijo será etiquetado mediante un terminal, debido nuevamente, a que la gramática está en forma normal de Chomsky. Por tanto, todo símbolo terminal que etiquete a un nodo hoja corresponde a dos nodos, la hoja y su padre.

Supongamos que el camino más largo desde la raíz a las hojas consta de $m + 2$ nodos (y $m + 1$ arcos). Entonces, el árbol de derivación tiene $m + 2$ niveles (es decir, los niveles 0, 1, ..., $m + 1$) y los nodos del último nivel (las hojas) son hijos únicos de sus nodos padre. En el nivel m hay, como máximo, 2^m nodos padre. Por tanto, si el camino más largo de un árbol de derivación consta de $m + 2$ nodos, entonces 2^m es la longitud máxima de la cadena derivada. Otra forma de decirlo es que, si la cadena derivada tiene longitud mayor que 2^m, entonces el camino más largo debe contener más de $m + 2$ nodos.

Ahora, cada nodo del árbol de derivación de una cadena se corresponde con la aplicación de una producción. Por lo tanto, la relación que existe entre el número de producciones que comprende cada etapa y la longitud de la cadena de terminales resultante sugiere que una gramática independiente del contexto en forma normal de Chomsky, genera un promedio de un terminal por cada dos producciones.

El conocimiento de la relación existente entre la longitud de la cadena y su derivación en una gramática independiente del contexto, nos permite demostrar el siguiente *lema de bombeo* para los lenguajes independientes del contexto.

Lema 3.6.1. Sea L un lenguaje independiente del contexto que no contiene ε. Entonces existe un entero k para el cual, si $z \in L$ y $|z| > k$, entonces z se puede volver a escribir como $z = uvwxy$ con las propiedades siguientes:

1. $|vwx| \leq k$.
2. Al menos o v o x no es ε.
3. $uv^iwx^iy \in L$ para todo $i \geq 0$.

Demostración. Supongamos que $G = (N, \Sigma, S, P)$ es una gramática independiente del contexto en forma normal de Chomsky con $L = L(G)$. Sea n el número de no terminales de N y sea $k = 2^n$. Supongamos que $z \in L$ con $|z| > k$.

Vamos a considerar el camino más largo del árbol de derivación correspondiente a z. Puesto que $|z| > 2^n$, este camino debe contener más de $n+2$ nodos. De los $n+2$ últimos nodos del camino, el último corresponderá a un terminal. Por tanto $n+1$ de dichos nodos están etiquetados con no terminales. Puesto que sólo hay n no terminales en N, alguno se repetirá. Supongamos que A está repetido en el camino. Entonces se tiene que

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy \overset{*}{\Rightarrow} uvwxy = z$$

para algunas subcadenas u, v, w, x e y. Puesto que sólo hemos considerado los $n+2$ últimos nodos de este camino, el camino de A a vwx puede tener como máximo $n+2$ nodos. Por tanto

$$|vwx| \leq 2^n$$

según las observaciones previas a este teorema.

Además, puesto que $A \overset{*}{\Rightarrow} vAx$, debemos obtener que $A \overset{*}{\Rightarrow} v^iAx^i$ para algún $i \geq 0$. Por tanto, $S \overset{*}{\Rightarrow} uv^iwx^iy$ para todo $i \geq 0$.

Finalmente, obsérvese que, puesto que $A \overset{*}{\Rightarrow} vAx$ y puesto que G está en forma normal de Chomsky, se debe tener que

$$A \Rightarrow BC \overset{*}{\Rightarrow} vAx \overset{*}{\Rightarrow} vwx$$

para algunos no terminales B y C. Entonces $B \overset{*}{\Rightarrow} v$ y $C \overset{*}{\Rightarrow} Ax$ o si no $B \overset{*}{\Rightarrow} vA$ y $C \overset{*}{\Rightarrow} x$. Ya que v y x son cadenas de terminales y no hay producciones ε, en el primer caso se tiene que $|v| \geq 1$ con lo que $v \neq \varepsilon$ y, en el segundo caso, $|x| \geq 1$ con lo que $x \neq \varepsilon$. En cualquier caso al menos uno de ellos no es ε. □

Al igual que el lema de bombeo para lenguajes regulares, el lema de bombeo para lenguajes independientes del contexto nos proporciona la posibilidad de probar si ciertos lenguajes no son independientes del contexto. Para ello, usa-

remos esencialmente el mismo planteamiento que en el caso de los lenguajes regulares.

Por ejemplo, el lenguaje $L = \{a^i b^j | j = i^2\}$ no es independiente del contexto. Supongamos que L es independiente del contexto. Probaremos que es imposible. Si L es independiente del contexto, entonces se puede aplicar el lema de bombeo y por tanto habrá un k que satisfaga las condiciones del Lema 3.6.1. Consideremos $z = a^k b^{k^2}$. Desde luego, $z \in L$ y $|z| > k$. Por tanto, z se puede descomponer en

$$z = uvwxy$$

y se puede asegurar que $uv^i wx^i y \in L$ para todo $i \geq 0$, tal que $|vx| \geq 1$ y $|vwx| \leq k$. Obsérvese que, si $v = a^r b^s$ para algún r y s, entonces $v^i = (a^r b^s)^i$ y, por tanto, $uv^i wx^i y$ tiene *bes* antes de *aes* con lo que no puede pertenecer a L. De forma similar, si $x = a^r b^s$, tampoco pueden ser bombeados. Por lo que debemos obtener que

$$v = a^r \quad \text{y} \quad x = a^s$$

o

$$v = b^r \quad \text{y} \quad x = b^s$$

o también

$$v = a^r \quad \text{y} \quad x = b^s$$

para algún valor de r y s. En el primer caso se tiene

$$uv^2 wx^2 y = a^{k+r+s} b^{k^2}$$

En el segundo caso tenemos

$$uv^2 wx^2 y = a^k b^{k^2+r+s}$$

En ambos casos, cuando al menos uno de los dos r o s son mayores o iguales a 1 (como se requiere en el lema de bombeo), la cadena resultante no puede pertenecer a L. En el tercer caso, se obtendrá

$$uv^i wx^i y = a^{k+(i-1)r} b^{k^2+(i-1)s}$$

el cual no pertenece a L para toda i a excepción de un número finito. Por tanto, L no puede ser independiente del contexto puesto que para él no se cumple el lema de bombeo.

El lema de bombeo tiene otros usos además de demostrar que un lenguaje no es independiente del contexto. Primero vamos a considerar el siguiente problema: para una GIC G arbitraria, ¿$L(G)$ es finito?

Supongamos que $G = (N, \Sigma, S, P)$ es una GIC arbitraria que está en forma normal de Chomsky. Se supone que N tiene n elementos. Entonces, de la demostración del Lema 3.6.1, se sabe que, si una cadena de $L(G)$ tiene una longitud mayor que 2^{n-1}, entonces $L(G)$ es infinito. A la inversa, puesto que la colección de terminales de G es finita, si $L(G)$ es infinita, entonces alguna cadena de $L(G)$ contiene más de 2^{n-1} símbolos. Supongamos ahora que $z \in L(G)$ es la cadena más corta de $L(G)$ con la propiedad de que $|z| \geq 2^{n-1}$. Pediremos que

$$2^{n-1} \leq |z| \leq 2^{n-1} + 2^n$$

Para verlo, supongamos que $|z| > 2^{n-1} + 2^n$. Entonces, de la demostración del Lema 3.6.1, se deduce que $z = uvwxy$ y $uwy \in L(G)$, para las subcadenas u, v, w, x, e y apropiadas. Entonces, puesto que $|uwx| \leq 2^n$, se obtiene que $|uwy| \geq |z| - 2^n > 2^{n-1}$.

Por otro lado, $|uwy| < |uvwxy| = |z|$. Pero se había supuesto que z era la cadena más corta de longitud mayor que 2^{n-1} y, por tanto, se llega a una contradicción. Por consiguiente,

$$2^{n-1} \leq |z| \leq 2^{n-1} + 2^n$$

En consecuencia, $L(G)$ es infinito si y sólo si existe un $z \in L(G)$ para el cual

$$2^{n-1} \leq |z| \leq 2^{n-1} + 2^n$$

Hay un número finito de cadenas sobre Σ^* y, si es necesario, podremos comprobar si pertenecen a $L(G)$. De aquí que tengamos un algoritmo para comprobar si un lenguaje independiente del contexto es finito o no.

Obsérvese que, en la eliminación de estados inútiles, proporcionamos de forma inadvertida un algoritmo que responde a otras cuestiones sobre un lenguaje especificado por una gramática independiente del contexto. $L(G)$ es no vacío si y sólo si su símbolo inicial genera una cadena de terminales. Tenemos un algoritmo para eliminar los no terminales que no generan cadenas de terminales. Por tanto, $L(G) = \emptyset$ si el símbolo inicial no pertenece a la colección de no terminales de la gramática transformada.

Otra cuestión a tratar sobre los lenguajes independientes del contexto, hace referencia a sus miembros: Dado un lenguaje independiente del contexto L sobre el alfabeto Σ y una cadena $w \in \Sigma^*$, ¿$w \in L$ o no?

Lema 3.6.2. Sea $G = (N, \Sigma, S, P)$ una gramática independiente del contexto que no tiene producciones ε y que está en forma normal de Chomsky. Sea x una cadena de Σ^*. Se puede determinar, para cada $A \in N$ y para cada subcadena w de x, si $A \overset{*}{\Rightarrow} w$.

Demostración. Sea $n = |x|$. Puesto que hay muchas subcadenas de x, las nombraremos mediante su posición inicial y su longitud. Sea w_{ij} la subcadena que comienza en la posición i y tiene una longitud j. Probaremos que el lema se cumple para todo w_{ij}. Lo realizaremos por inducción sobre la longitud de la subcadena, es decir, sobre j.

Supongamos que $j = 1$. Entonces $|w_{ij}| = 1$ y, por tanto, w_{ij} es un símbolo terminal. Como la gramática está en forma normal de Chomsky, para algún no terminal A se tiene que $A \overset{*}{\Rightarrow} w_{ij}$ si y sólo si existe una producción $A \rightarrow w_{ij}$ en P. Es posible determinarlo, ya que P es finito.

Ahora supongamos que $j > 1$ y que la afirmación se cumple para toda subcadena de longitud menor que j. Obsérvese que $A \overset{*}{\Rightarrow} w_{ij}$ si y sólo si $A \rightarrow BC$ para algún par B y C de no terminales para los cuales $B \overset{*}{\Rightarrow} w_{ik}$ y $C \overset{*}{\Rightarrow} w_{i+k,j-k}$ para algún k entre 1 y $j-1$. Entonces tanto w_{ik} como $w_{i+k,j-k}$ tienen longitud menor que j y, por la hipótesis de inducción, es posible determinar si $B \overset{*}{\Rightarrow} w_{ik}$ y si $C \overset{*}{\Rightarrow} w_{i+k,j-k}$. Además podemos determinar si $A \overset{*}{\Rightarrow} w_{ij}$ para cada i entre 1 y n y cada j entre 1 y $n - i + 1$. □

En la demostración anterior se observa que si $j = n$ entonces se puede determinar si

$$S \overset{*}{\Rightarrow} w_{1j} = w_{1n} = x$$

Es decir, se puede determinar si $x \in L(G)$ para cada $x \in \Sigma^*$.

Se cumple que $x \in L(G)$ si y sólo si $S \overset{*}{\Rightarrow} x$ y, por el Lema 3.6.2, sabemos que es posible determinar si $S \overset{*}{\Rightarrow} x$. Naturalmente, no es lo mismo saber que algo es posible, que hacerlo. Se conocen varios algoritmos para determinar si $x \in L(G)$. Vamos a presentar uno llamado algoritmo de CYK y que se debe a Cocke, Younger y Kasami. El algoritmo de CYK simplemente construye conjuntos N_{ij} de no terminales que generan las subcadenas w_{ij} de x. Una vez hecho, si $S \in N_{1n}$, entonces $x \in L(G)$ (donde $|x| = n$). Obsérvese que se elimina mucho trabajo por el hecho de que no pueden existir subcadenas de longitud mayor que $n - i + 1$ que empiecen en la posición i.

El algoritmo de CYK, se enuncia como sigue:

1. Para cada $i = 1, 2, \ldots, n$, sea

$$N_{i1} = \{A \mid A \rightarrow w_{i1}\}$$

Es decir, N_{i1} es el conjunto de todos los no terminales que producen el i-ésimo símbolo de x.

2. Para $j = 2, 3, ..., n$, hacer lo siguiente:
   Para $i = 1, 2, ..., n - j + 1$, hacer lo siguiente:

   a. Inicializar $N_{ij} = \emptyset$.

   b. Para $k = 1, 2, ..., j - 1$, añadir a N_{ij} todos los no terminales A para los cuales $A \rightarrow BC$, con $B \in N_{ik}$ y $C \in N_{i+k, j-k}$.

3. Si $S \in N_{1n}$, entonces $x \in L(G)$.

Obsérvese en que el algoritmo de CYK requiere que se tenga una gramática independiente del contexto en forma normal de Chomsky. El algoritmo para la construcción de los conjuntos de no terminales N_{ij} sigue la idea de la demostración del Lema 3.6.2. En la etapa 2b, que es en la que se construyen los N_{ij}, se emparejan los no terminales de N_{ik} con los de $N_{i+k, j-k}$, y se trata de encontrarlos en el lado derecho de las producciones. Cuando se obtienen tales lados derechos, se añade a N_{ij} el no terminal que está en el lado izquierdo de la producción dada.

Ejemplo 3.6.1

Consideremos la gramática independiente del contexto

$$S \rightarrow AB \mid BC$$
$$A \rightarrow BA \mid a$$
$$B \rightarrow CC \mid b$$
$$C \rightarrow AB \mid a$$

Para la cadena $x = bbab$, se tiene la siguiente tabla donde cada casilla representa al conjunto N_{ij}:

		$j=1$	$j=2$	$j=3$	$j=4$
b	$j=1$	B	$\emptyset$	A	S, C
b	$j=2$	B	A, S	S, C	
a	$j=3$	A, C	S, C		
b	$j=4$	B			

Ya que S está en N_{14}, x se genera a partir del símbolo inicial S. Por tanto, x está en el lenguaje generado por esta gramática.

Supongamos que $G_1 = (N_1, \Sigma_1, S_1, P_1)$ y $G_2 = (N_2, \Sigma_2, S_2, P_2)$ son dos gramáticas independientes del contexto para las cuales N_1 y N_2 son disjuntos. Definiremos la gramática independiente del contexto $G = (N, \Sigma, S, P)$, donde

$$N = N_1 \cup N_2 \cup \{S\}$$
$$\Sigma = \Sigma_1 \cup \Sigma_2$$
$$P = P_1 \cup P_2 \cup \{(S, S_1), (S, S_2)\}$$

siendo S un nuevo símbolo. Es decir, P contiene las producciones de P_1 y P_2, además de dos nuevas producciones $S \rightarrow S_1 \mid S_2$. Se ve fácilmente que $L(G) = L(G_1) \cup L(G_2)$. Primero, si $w \in L(G_1)$, entonces se tiene que $S_1 \overset{*}{\Rightarrow} w$, con lo que $S \Rightarrow S_1 \overset{*}{\Rightarrow} w$ y, por tanto, se tiene que $w \in L(G)$. De forma similar, si $w \in L(G_2)$, entonces se tiene que $w \in L(G)$ y, por tanto, $L(G_1) \cup L(G_2) \subseteq L(G)$. Por otro lado, si $w \in L(G)$, entonces $S \overset{*}{\Rightarrow} w$. Pero $S \rightarrow S_1 \mid S_2$ son las únicas producciones que tienen en su lado izquierdo el símbolo S. Entonces o bien

$$S \Rightarrow S_1 \overset{*}{\Rightarrow} w$$

o

$$S \Rightarrow S_2 \overset{*}{\Rightarrow} w$$

Ahora bien, como $N_1 \cap N_2 = \emptyset$, sólo se pueden usar producciones de P_1 en $S_1 \overset{*}{\Rightarrow} w$, con lo que si $S \Rightarrow S_1 \overset{*}{\Rightarrow} w$, se deduce que $w \in L(G_1)$. De un modo similar, si $S \Rightarrow S_2 \overset{*}{\Rightarrow} w$, se obtiene que $w \in L(G_2)$. Por tanto, $L(G) \subseteq L(G_1) \cup L(G_2)$. Con lo que hemos demostrado el siguiente teorema:

Teorema 3.6.3. Si L_1 y L_2 son lenguajes independientes del contexto, entonces $L_1 \cup L_2$ es un lenguaje independiente del contexto.

Es decir, el conjunto de los lenguajes independientes del contexto es cerrado con respecto a la unión. Este conjunto también es cerrado con respecto a la cerradura de estrella.

Teorema 3.6.4. Si L es un lenguaje independiente del contexto, entonces L^* también lo es.

Demostración. Sea $L = L(G)$ para $G = (N, \Sigma, S, P)$. Construiremos una gramática independiente del contexto $G' = (N', \Sigma, S', P')$ que genere L^*, de forma que $N' = N \cup \{S', T\}$, donde S' y T son nuevos símbolos, y añadiremos las producciones $S' \rightarrow ST \mid \varepsilon$ y $T \rightarrow ST \mid \varepsilon$. Obsérvese que si $w_1, w_2, \ldots, w_n \in L(G)$, se tiene

$$
\begin{aligned}
S' &\Rightarrow ST \Rightarrow SST \\
&\overset{*}{\Rightarrow} S \ldots ST \\
&\qquad (n \text{ términos}) \\
&\Rightarrow S \ldots S\varepsilon \\
&\overset{*}{\Rightarrow} w_1 w_2 \ldots w_n
\end{aligned}
$$

Por tanto, $L^* \subseteq L(G')$. □

Teorema 3.6.5. La concatenación de lenguajes independientes del contexto es independiente del contexto.

Demostración. Véase el Ejercicio 3.6.5. □

En los Teoremas 3.6.3, 3.6.4 y 3.6.5 hemos enunciado tres *propiedades de cierre* de los lenguajes independientes del contexto. Desgraciadamente, los lenguajes independientes del contexto no son cerrados con respecto a la intersección. El lenguaje

$$L = \{a^i b^i c^i \mid i \geq 0\}$$

no es independiente del contexto, como se demostró en la aplicación del lema de bombeo (Lema 3.6.1). Obsérvese que $L_1 = \{a^i b^j c^j \mid i, j \geq 0\}$ es generado por la gramática independiente del contexto

$$
\begin{aligned}
S &\rightarrow AC \\
A &\rightarrow aA \mid \varepsilon \\
C &\rightarrow bCc \mid \varepsilon
\end{aligned}
$$

y el lenguaje $L_2 = \{a^i b^i c^j \mid i, j \geq 0\}$ es generado por

$$
\begin{aligned}
S &\rightarrow AC \\
A &\rightarrow aAb \mid \varepsilon \\
C &\rightarrow cC \mid \varepsilon
\end{aligned}
$$

y, por tanto, son lenguajes independientes del contexto. Pero se tiene que $L = L_1 \cap L_2$, y con lo que, la intersección de lenguajes independientes del contexto no es necesariamente un lenguaje independiente del contexto.

Además, los lenguajes independientes del contexto no son cerrados con respecto a la complementación. Si denotamos $\Sigma^* - L_i$ mediante $\overline{L}$, entonces

$$L_1 \cap L_2 = \overline{(\overline{L}_1 \cup \overline{L}_2)}$$

Ya que la unión de lenguajes independientes del contexto produce un lenguaje independiente del contexto, obsérvese que, si al complementar lenguajes independientes del contexto siempre se obtuvieran lenguajes independientes del contexto, la intersección anterior debería haber sido independiente del contexto.

Ejercicios de la Sección 3.6

3.6.1. Probar que cada uno de los siguientes lenguajes no son independientes del contexto.

(a) $\{a^i b^i c^i \mid i \geq 1\}$

(b) $\{a^i b^i c^j \mid j \geq i\}$

(c) $\{a^i b^j c^k \mid i \leq j \leq k\}$

(d) $\{a^i \mid i \text{ es primo}\}$

(e) $\{w \in \{a, b, c\}^* \mid w$ tiene el mismo número de *aes* que de *bes* y *ces*$\}$

(f) $\{a^n b^n c^m \mid n \leq m \leq 2n\}$

(g) $\{ww \mid w \in \{a, b\}^*\}$.

3.6.2. Determinar si el lenguaje generado por

$$S \rightarrow aaA \mid B$$
$$B \rightarrow aA \mid b$$
$$A \rightarrow aS \mid B \mid \varepsilon$$

es finito o infinito.

3.6.3. Determinar si *bba*, *bab* y *babba* están en $L(G)$ correspondiente a la gramática G del Ejemplo 3.6.1, usando el algoritmo de CYK.

3.6.4. Sea $G = (N, \Sigma, S, P)$ una GIC. Al igual que en la demostración del Teorema 3.6.4, construir $G' = (N', \Sigma, S, P')$. Demostrar que $L(G') \subseteq L^*$.

3.6.5. Demostrar el Teorema 3.6.5.

3.7 AUTÓMATA DE PILA

Hemos visto que las gramáticas independientes del contexto amplían la capacidad para especificar lenguajes al incluir algunos lenguajes que no son reconocidos por un autómata finito. En esta sección consideraremos un autómata que será capaz de reconocer todo lenguaje independiente del contexto.

Intuitivamente, el problema que se plantea con los autómatas finitos es que sólo tienen capacidad para una "memoria" finita. Lenguajes independientes del contexto tan sencillos como $\{a^n b^n \mid n \geq 0\}$ necesitan guardar gran cantidad de in-

formación; se debe verificar no solamente que todas las *aes* preceden a las *bes*, sino que además se tienen que contar las *aes*. Puesto que el número de *aes* es arbitrario, necesitamos establecer una limitación sobre el número de *aes* que se pueden contar.

El lenguaje independiente del contexto $\{wcw^I | w \in \{a, b, c\}^*\}$ necesita algo más que una capacidad sin límites para contar los símbolos. Además se deben guardar los símbolos de la cadena w para compararlos con los símbolos de la cadena w^I.

En esta sección definiremos un autómata que cuenta con un mecanismo que permite almacenamiento ilimitado y opera como una pila. Este autómata se llama *autómata de pila*. En la siguiente sección, se estudiará la conexión que existe entre los autómatas de pila y los lenguajes independientes del contexto.

Un autómata de pila se comporta de forma similar a como lo hacen los autómatas finitos del Capítulo 2. En todo momento se encuentran en un estado y el cambio de estado depende del estado actual y de una información adicional. En el caso de los autómatas finitos del Capítulo 2, la "información adicional" consiste en el símbolo de entrada actual. En el caso de los autómatas de pila, se incluyen el símbolo de entrada actual y el símbolo que está en ese momento en la cima de la pila. Además de cambiar de estado, el autómata de pila cambia, también, la cima de la pila.

Definición 3.7.1. Un *autómata de pila no determinista* (ADPND) es una 7-tupla, $M = (Q, \Sigma, \Gamma, \Delta, s, F, z)$ donde

Q es un conjunto finito de *estados*

Σ es un *alfabeto de entrada*

Γ es un alfabeto llamado *alfabeto de la pila*

Δ es una *regla de transición*

$s \in Q$ es el *estado inicial* o *de partida*

$z \in \Gamma$ es el *símbolo inicial de la pila*

$F \subseteq Q$ es el conjunto de *estados finales* o *de aceptación*.

De nuestra descripción anterior se deduce que una regla de transición, para obtener el siguiente estado y la acción a realizar sobre la pila, debe conocer el estado actual, el símbolo de entrada actual y la cima actual de la pila. Por consiguiente, Δ se define por medio de ternas de la forma (q, σ, γ), donde q es un estado de Q, σ es un símbolo de $\Sigma \cup \{\varepsilon\}$ y $\gamma \in \Gamma$. El resultado de aplicar Δ a dicha terna es una colección de pares (p, w), donde $p \in Q$ es el estado siguiente y $w \in \Gamma^*$ es la cadena que se introducirá (o apilará) en la pila *en lugar del* símbolo γ que estaba antes allí. Por tanto, se puede definir la relación Δ como

$$\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^*$$

Ya que estamos definiendo un autómata de pila *no determinista*, cabe esperar que Δ no tiene por qué ser una función. Por tanto, si aplicamos Δ a la terna (q_1, a, b) se obtiene el conjunto $\{(q_2, cd), (q_2, dce), (q_3, efc)\}$, del cual se elige de forma no determinista uno de los pares, de manera que el autómata de pila cambia para reflejar dicha elección. Obsérvese que, puesto que los pares resultantes pertenecen a $Q \times \Gamma^*$ y ya que $\varepsilon \in \Gamma^*$, se puede tener que de $\Delta(q, a, b)$ se obtenga (p, ε). Esto indica que el estado siguiente es p y que el símbolo b se elimina (o se desapila) de la cima de la pila.

Al definir $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^*$, se fuerza a que Δ deba considerar un símbolo de la pila en *todos* sus movimientos. Es decir, no es posible ningún movimiento si la pila está vacía. Por esta razón, se supone que inicialmente la pila contiene algún símbolo z, que es el símbolo inicial de la pila. Por otro lado, un movimiento tal como

$$\Delta(q, \varepsilon, a) = \{(p, aa)\}$$

indica que el ADPND puede cambiar a un estado p y apilar una a en la pila sin consumir ningún símbolo de la entrada. Finalmente, obsérvese que si $\Delta(q, \sigma, \gamma) = \emptyset$, no es posible ningún movimiento desde el estado q con el símbolo de entrada σ y con el símbolo de la pila γ. En este caso, el ADPND parará su ejecución.

Consideremos el autómata de pila no determinista definido por

$$\begin{aligned} Q &= \{q_1, q_2, q_3, q_4\} \\ \Sigma &= \{a, b\} \\ \Gamma &= \{A, B\} \\ z &= A \\ F &= \{q_4\} \\ s &= q_1 \end{aligned}$$

y Δ dado por la siguiente tabla:

Δ	(a, A)	(b, A)	(ε, A)	(a, B)	(b, B)	(ε, B)
q_1	$\{(q_2, BA), (q_4, A)\}$		$\{(q_4, \varepsilon)\}$			
q_2				$\{(q_2, BB)\}$	$\{(q_3, \varepsilon)\}$	
q_3			$\{(q_4, A)\}$		$\{(q_3, \varepsilon)\}$	

Ya que Δ depende del estado actual, el símbolo de entrada actual y el símbolo actual de la cima de la pila, la tabla tiene filas que corresponden a los estados y columnas que corresponden a los pares de símbolos de entrada y de la pila.

Obsérvese que no hay transiciones para todas las ternas posibles de estado, símbolo de entrada y símbolo de la pila. Por tanto, si el ADPND pasa a un estado para el cual no se especifica un estado siguiente y una acción de la pila para los símbolos actuales de la pila y la entrada, el ADPND no puede volver a realizar ningún movimiento. En otras palabras, su actividad termina. En particular, cuando el autómata está en el estado q_4, que es el estado de aceptación, no hay ninguna transición sea cual sea el símbolo de la cima y de la entrada.

Las acciones que realiza un ADPND son sencillas. La transición $\Delta(q_2, a, B) = \{(q_2, BB)\}$ apila B sobre la pila y no cambia el estado. La transición $\Delta(q_2, b, B) = \{(q_3, \varepsilon)\}$ desapila la B y cambia de estado. La transición $\Delta(q_1, a, A) = \{(q_2, BA), (q_4, \varepsilon)\}$ indica que se realizará la elección de una acción de forma no determinista. Como, para esta transición, el ADPND empieza en el estado q_1 con A en la cima de la pila, esta transición indica que o bien se apila B y se pasa al estado q_2 o si no se desapila A y se pasa al estado de aceptación q_4. Si se realiza esto último, el ADPND parará.

Por otro lado, si el ADPND se mueve al estado q_2, entonces obsérvese que cada vez que a aparece en la entrada se apila una B en la pila. Es más, puesto que el ADPND permanece en el estado q_2 hasta que se encuentra la primera b y entonces se mueve al estado q_3, ninguna b puede preceder a una a. Finalmente, en el estado q_3 sólo se consideran las *bes* y, cuando se encuentra cualquier b, se desapila B de la pila. (Sólo pueden desapilarse las *Bes* que fueron apiladas, debido a encontrarse una a en la entrada).

El lector podría verificar que las únicas cadenas que pertenecen al lenguaje $\{a^n b^n \mid n \geq 0\} \cup \{a\}$, son las únicas cadenas de entrada que, una vez que han sido consumidas, causan que el ADPND termine en el estado final q_4.

El autómata de pila no determinista del ejemplo anterior *acepta* el lenguaje $\{a^n b^n \mid n \geq 0\} \cup \{a\}$ puesto que las cadenas que una vez consumidas causan que el ADPND se mueva de su configuración inicial a su estado final son exactamente las cadenas que pertenecen al lenguaje anterior. Procederemos a formalizar qué significa que un ADPND acepte un lenguaje pero primero introduciremos una notación usual.

Durante el procesamiento de una cadena, se pueden describir las sucesivas configuraciones por las que pasa el ADPND en función de su estado actual, de los contenidos de la pila y de la entrada no leída. La terna (q, w, u), donde q es el estado actual, w es la cadena de entrada restante y u el contenido de la pila (con el símbolo de la cima en el extremo de la izquierda), se llama *descripción instantánea* (DI) del autómata. Describe la configuración del autómata de pila en

un instante en particular. Indicaremos un movimiento de una configuración a otra situando el símbolo $\vdash$ entre dos descripciones instantáneas

$$(q_1, aw, bx) \vdash (q_2, w, yx)$$

representa un movimiento que resulta de $(q_2, y) \in \Delta\,(q_1, a, b)$. Podemos denotar los movimientos con un número arbitrario de pasos por medio de $\overset{*}{\vdash}$ y $\overset{+}{\vdash}$ (donde * indica "cero o más pasos" y + indica "uno o más pasos").

Ahora definiremos formalmente lo que significa que un autómata de pila no determinista acepte un lenguaje.

Definición 3.7.2. Sea $M = (Q, \Sigma, \Gamma, s, z, F, \Delta)$ un autómata de pila no determinista. El lenguaje aceptado por M se denota por $L\,(M)$ y es el conjunto

$$L\,(M) = \{w \in \Sigma^* \mid (s, w, z) \overset{*}{\vdash} (p, \varepsilon, u) \text{ para } p \in F \text{ y } u \in \Gamma^*\}$$

Obsérvese que la aceptación requiere que M se mueva a un estado final cuando la cadena w se agote. M puede terminar o no con la pila vacía. (Sin embargo, obsérvese que cuando la pila se vacía el ADPND *debe* parar, ya que todas las transiciones requieren un símbolo de la pila).

Ejemplo 3.7.1

Supongamos que queremos construir un autómata de pila no determinista que acepte el lenguaje

$$L = \{w \in \{a, b\}^* \mid w \text{ contiene la misma cantidad de } aes \text{ que de } bes\}$$

Debemos contar el número de ocurrencias de *aes* y *bes*. Esto puede realizarse introduciendo símbolos en la pila cuando se lee algún carácter de entrada y extrayéndolo cuando se lee otro. Sea M el ADPND dado por

$$\begin{aligned} Q &= \{q_1, q_2\} \\ \Sigma &= \{a, b\} \\ \Gamma &= \{A, B, Z\} \\ s &= q_1 \\ z &= Z \\ F &= \{q_2\} \end{aligned}$$

y la regla de transición Δ dada por la lista

$$\Delta(q_1, \varepsilon, Z) = \{(q_2, Z)\}$$
$$\Delta(q_1, a, Z) = \{(q_1, AZ)\}$$
$$\Delta(q_1, b, Z) = \{(q_1, BZ)\}$$
$$\Delta(q_1, a, A) = \{(q_1, AA)\}$$
$$\Delta(q_1, b, A) = \{q_1, \varepsilon)\}$$
$$\Delta(q_1, a, B) = \{(q_1, \varepsilon)\}$$
$$\Delta(q_1, b, B) = \{(q_1, BB)\}$$

Para procesar la cadena *abba*, M realiza los siguientes movimientos:

$$\begin{aligned}(q_1, abba, Z) &\vdash (q_1, bba, AZ)\\ &\vdash (q_1, ba, Z)\\ &\vdash (q_1, a, BZ)\\ &\vdash (q_1, \varepsilon, Z)\\ &\vdash (q_2, \varepsilon, Z)\end{aligned}$$

y el ADPND para en el estado de aceptación q_2. Luego el *abba* está en $L(M)$.

Ejemplo 3.7.2

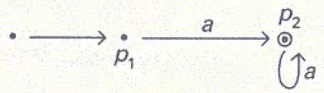
Podemos hacer uso del hecho de que los símbolos pueden ser recuperados de la pila en orden inverso a como fueron introducidos. Consideremos el lenguaje $L = \{wcw^I | w \in \{a, b\}^*\}$. Un ADPND que acepta L debería introducir los caracteres de entrada en la pila hasta que se encuentra una c y, entonces, compara los caracteres de la entrada con la cima de la pila, desapilando la pila si concuerdan. Un ADPND M para el mismo puede ser dado mediante

$$Q = \{q_1, q_2, q_3\}$$
$$\Sigma = \{a, b\}$$
$$\Gamma = \{a, b, z\}$$
$$s = q_1$$
$$z = \text{ símbolo inicial de la pila}$$
$$F = \{q_3\}$$

y Δ viene dado por la lista siguiente:

$$\Delta(q_1, a, z) = \{(q_1, az)\} \qquad \Delta(q_1, c, z) = \{(q_2, z)\}$$
$$\Delta(q_1, a, a) = \{(q_1, aa)\} \qquad \Delta(q_1, c, a) = \{(q_2, a)\}$$
$$\Delta(q_1, a, b) = \{(q_1, ab)\} \qquad \Delta(q_1, c, b) = \{(q_2, b)\}$$
$$\Delta(q_1, b, z) = \{(q_1, bz)\} \qquad \Delta(q_2, a, a) = \{(q_2, \varepsilon)\}$$
$$\Delta(q_1, b, a) = \{(q_1, ba)\} \qquad \Delta(q_2, b, b) = \{(q_2, \varepsilon)\}$$
$$\Delta(q_1, b, b) = \{(q_1, bb)\} \qquad \Delta(q_2, \varepsilon, z) = \{(q_3, z)\}$$

Cada autómata finito no determinista se puede interpretar como un ADPND que nunca opera sobre la pila. Por ejemplo, consideremos el AFN cuyo diagrama de transición es



Sea el ADPND M dado por

$$Q = \{q_1, q_2\}$$
$$\Sigma = \{a\}$$
$$\Gamma = \{z\}, \text{el símbolo inicial de la pila}$$
$$s = q_1$$
$$F = \{q_2\}$$

y Δ dado por

Δ	(a, z)
q_1	$\{(q_2, z)\}$
q_2	$\{(q_2, z)\}$

Está claro que M acepta el mismo lenguaje que el autómata finito.

Ejercicios de la Sección 3.7

3.7.1. Obtener un ADPND que acepte $\{a^n b^n \mid n \geq 0\}$.

3.7.2. Describir el proceso realizado por el ADPND M del Ejemplo 3.7.1 sobre las cadenas *abaababb* y *abaa*. ¿Son aceptadas por M?

3.7.3. Describir el proceso que realiza el ADPND M del Ejemplo 3.7.2 sobre las cadenas *c*, *abcba*, *abcab* y *babbcbbab*.

3.7.4. El lenguaje $L = \{ww^I \mid w \in \{a, b\}^+\}$ es similar al lenguaje del Ejemplo 3.7.2, excepto porque no hay ningún carácter que marque el límite entre w y w^I. Por tanto, para que un autómata de pila no determinista acepte L tendrá que existir una elección no determinista en un punto medio que pasa de apilar caracteres en la

pila a compararlos y desapilarlos. Usar el ADPND del Ejemplo 3.7.2 como punto de partida y construir un ADPND que acepte L añadiendo o eliminando las transiciones apropiadas.

3.7.5. Obtener los ADPND para los siguientes lenguajes:

(a) $\{a^n b^{2n} | n \geq 0\}$

(b) $\{a^n b^m c^{n+m} | n \geq 0 \text{ y } m \geq 0\}$

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

(d) $\{w \in \{a, b\}^* | w \text{ contiene una } a \text{ más que } bes\}$

(e) $\{a^n b^m | n \geq 0 \text{ y } m \neq n\}$

(f) $\{w_1 c w_2 | w_1, w_2 \in \{a, b\}^* \text{ y } w_1 \neq w_2^I\}$

3.7.6. ¿Qué lenguaje será aceptado por el M dado a continuación

$Q = \{q_1, q_2, q_3\}$
$\Sigma = \{a, b\}$
$\Gamma = \Sigma \cup \{z\}$, donde z es el símbolo inicial de la pila
$s = q_1$
$F = \{q_3\}$

y Δ viene dado por la tabla

Δ	(a, z)	(a, b)	(b, a)	(b, b)
q_1	$\{(q_2, a), (q_3, \varepsilon)\}$			
q_2		$\{(q_3, \varepsilon)\}$	$\{(q_2, b)\}$	$\{(q_2, b)\}$

¿Cuál será el lenguaje aceptado si se cambia F por $F = \{q_1, q_2, q_3\}$?

3.7.7. Obtener un ADPND que acepte aa^*ba. La presencia de un dispositivo de memoria en un autómata puede permitir que se economicen estados. Un autómata finito no determinista para el lenguaje precedente, necesariamente tiene al menos cuatro estados y, por tanto, un ADPND para este lenguaje que ignore su pila deberá tener, al menos, cuatro estados. Obtener un ADPND para aa^*ba que sólo tenga dos estados.

3.8 AUTÓMATAS DE PILA Y LENGUAJES INDEPENDIENTES DEL CONTEXTO

El principal resultado de esta sección es que los autómatas de pila no deterministas aceptan los lenguajes independientes del contexto. Esto requiere que, para cada lenguaje independiente del contexto, debamos poder encontrar un autómata

de pila no determinista que lo acepte y a la inversa, para cada lenguaje aceptado por un autómata de pila no determinista, obtener una gramática independiente del contexto. Primero demostraremos que cualquier lenguaje independiente del contexto es aceptado por un autómata de pila no determinista presentando un método de construcción para dicho ADPND.

Sea $G = (N, \Sigma, S, P)$ una gramática independiente del contexto. Desearíamos construir un autómata de pila no determinista que acepte $L(G)$. El ADPND que construiremos debería poder verificar que todas las cadenas de $L(G)$ son derivables. La idea principal es construir un ADPND que pueda realizar una derivación por la izquierda para cualquier cadena del lenguaje. Representaremos la derivación guardando en la pila los no terminales del extremo derecho de la derivación, mientras que la parte izquierda (compuesta totalmente por terminales) es idéntica que la cadena de entrada que se ha leído.

Empezaremos por introducir en la pila el símbolo inicial (de G). En cada una de las etapas posteriores realizaremos una de estas acciones:

1. Si el símbolo que está en la cima de la pila es un no terminal A, lo sustituiremos por el lado derecho de la producción para A, $A \rightarrow w$, o
2. Si el símbolo de la cima de la pila es un terminal y se corresponde con el siguiente símbolo de la entrada, los desapilaremos de la pila.

Estas acciones imitan una derivación por la izquierda para la cadena de entrada. Si se agota la cadena de entrada y en la cima de la pila se encuentra el símbolo inicial de la pila, aceptaremos la cadena puesto que es posible la derivación de la misma. Obsérvese que el no determinismo se presenta en la etapa 1 cuando se elige el lado derecho de alguna producción para que sea introducido en la pila.

Para definir dicho autómata de pila no determinista M, sean

$$Q = \{q_1, q_2, q_3\}$$
$$\Gamma = N \cup \Sigma \cup \{z\}, \quad \text{donde } z \text{ es un símbolo inicial de la pila (y es distinto de todos los símbolos de } N \cup \Sigma)$$
$$F = \{q_3\}$$
$$s = q_1$$

y la regla de transición Δ está compuesta por cuatro tipos de transiciones:

1. $\Delta(q_1, \varepsilon, z) = \{(q_2, Sz)\}$, la cual se corresponde con la introducción del símbolo inicial en la pila.

2. $\Delta(q_2, \varepsilon, A) = \{(q_2, w) | A \to w$ es una producción de $P\}$ para cada no terminal A de N.
3. $\Delta(q_2, a, a) = \{(q_2, \varepsilon)\}$ para cada símbolo terminal a de Σ.
4. $\Delta(q_2, \varepsilon, z) = \{(q_3, z)\}$.

Ejemplo 3.8.1

Sea G una gramática independiente del contexto cuyas producciones son

$$S \to aSa | bSb | c$$

Obsérvese que $L(G) = \{wcw^I | w \in \{a, b\}^*\}$. El autómata de pila no determinista correspondiente tiene la siguiente regla de transición

$$\Delta(q_1, \varepsilon, z) = \{(q_2, Sz)\}$$
$$\Delta(q_2, \varepsilon, S) = \{(q_2, aSa), (q_2, bSb), (q_2, c)\}$$
$$\Delta(q_2, a, a) = \Delta(q_2, b, b) = \Delta(q_2, c, c) = \{(q_2, \varepsilon)\}$$
$$\Delta(q_2, \varepsilon, z) = \{(q_3, \varepsilon)\}$$

La cadena *abcba* es aceptada mediante la siguiente secuencia:

$$\begin{aligned}
(q_1, abcba, z) &\vdash (q_2, abcba, Sz) \\
&\vdash (q_2, abcba, aSaz) \\
&\vdash (q_2, bcba, Saz) \quad \text{se extrae un símbolo de la pila} \\
&\vdash (q_2, bcba, bSbaz) \\
&\vdash (q_2, cba, Sbaz) \quad \text{otra extracción} \\
&\vdash (q_2, cba, cbaz) \\
&\vdash (q_2, ba, baz) \quad \text{otra extracción} \\
&\vdash (q_2, a, az) \quad \text{otra extracción} \\
&\vdash (q_2, \varepsilon, z) \quad \text{otra extracción} \\
&\vdash (q_3, \varepsilon, z) \quad \text{se acepta}
\end{aligned}$$

Supongamos que se ha construido un ADPND a partir de una gramática independiente del contexto, de la forma vista anteriormente. Obsérvese que si se tiene que

$$(q_2, x, A\alpha) \overset{*}{\vdash} (q_2, x, \beta\alpha)$$

entonces, en la gramática es posible que $A \overset{*}{\Rightarrow} \beta$. Por tanto, si $w = a_1 a_2 \ldots a_n$ es aceptada por el ADPND, debemos tener que

$$\begin{aligned}(q_2, a_1a_2 \ldots a_n, Sz) &\overset{*}{\vdash} (q_2, a_1a_2 \ldots a_n, a_1\beta_1 z)\\ &\vdash (q_2, a_2 \ldots a_n, \beta_1 z)\\ &\overset{*}{\vdash} (q_2, a_n, a_n z)\\ &\vdash (q_2, \varepsilon, z)\\ &\vdash (q_3, \varepsilon, z)\end{aligned}$$

y por tanto se obtienen las derivaciones siguientes

$$S \overset{*}{\Rightarrow} a_1\beta_1 \overset{*}{\Rightarrow} a_1a_2\beta_2 \overset{*}{\Rightarrow} \ldots \overset{*}{\Rightarrow} a_1a_2 \ldots a_n = w$$

Por consiguiente, si w es aceptada por un ADPND, entonces w se deriva de la gramática.

A la inversa, supongamos que tenemos una gramática en forma normal de Chomsky. Si $S \overset{*}{\Rightarrow} a_1a_2 \ldots a_n$, entonces tendremos una derivación de $a_1a_2 \ldots a_n$, por la izquierda y de la forma

$$S \overset{*}{\Rightarrow} A_1\alpha_1 \Rightarrow a_1\alpha_1 \overset{*}{\Rightarrow} a_1A_2\alpha_2 \Rightarrow a_1a_2\alpha_2 \overset{*}{\Rightarrow} a_1a_2 \ldots a_n$$

Por tanto, en un ADPND derivado de esta gramática, se puede tener la secuencia

$$\begin{aligned}(q_2, a_1a_2 \ldots a_n, Sz) &\overset{*}{\vdash} (q_2, a_1a_2 \ldots a_n, a_1\alpha_1 z)\\ &\vdash (q_2, a_2 \ldots a_n, \alpha_1 z)\\ &\overset{*}{\vdash} (q_2, \varepsilon, z)\\ &\vdash (q_3, \varepsilon, z)\end{aligned}$$

Es decir, el ADPND acepta la cadena $w = a_1a_2 \ldots a_n$. Por tanto, se deduce el siguiente teorema:

Teorema 3.8.1. Si L es un lenguaje independiente del contexto, entonces existirá un ADPND para el cual $L = L(M)$.

El Teorema 3.8.1 representa la mitad de nuestro objetivo. Nuestra tarea pendiente es demostrar su inverso, es decir, que cualquier lenguaje aceptado por un ADPND es independiente del contexto. Para generalizar, necesitaremos tomar un ADPND arbitrario y probar cómo se obtendrá una gramática independiente del contexto que genere el lenguaje aceptado por él. Con el objeto de simplificar el problema, trataremos con uno equivalente, desde el principio, en el cual el ADPND se comporte correctamente.

Sea M un ADPND arbitrario. El *lenguaje aceptado por la pila vacía* de M se define como

$$N(M) = \{w \mid (q_1, w, z) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)\}$$

En este caso, q_1 es el estado inicial de M y p es cualquier estado de Q. $N(M)$ es, sencillamente, el conjunto de todas las cadenas que conducen a M desde el estado inicial, a cualquier estado en el cual la pila esté vacía. Puesto que todas las transiciones se definen en función de un símbolo de la pila, tal configuración es una configuración de parada de M.

Obsérvese que para un ADPND, no se cumple necesariamente que $N(M) = L(M)$. Por ejemplo sea M, cuyas transiciones son

$$\Delta(q_1, a, z) = \{(q_1, az)\}$$
$$\Delta(q_1, b, z) = \{(q_2, \varepsilon)\}$$
$$\Delta(q_1, a, a) = \{(q_3, a)\}$$

donde $Q = \{q_1, q_2, q_3\}$, $F = \{q_3\}$, z es el símbolo inicial de la pila y q_1 es el estado inicial. Obsérvese que $N(M) = \{b\}$ mientras que $L(M) = \{a^2\}$.

Sea M una ADPND. Podemos transformar M en un ADPND M' tal que $L(M) = N(M')$. Resultará que M' tendrá un único estado final.

Primero observemos que si M nunca desapila z de la pila, se realiza por medio de una transición de la forma $(q', \varepsilon) \in \Delta(q, \tau, z)$ para algunos estados q y q' y algún $\tau \in \Gamma \cup \{\varepsilon\}$. Puede ser que $q' \in F$ o no. Se sustituyen todas las transiciones donde q' no esté en F por nuevas transiciones (p_1, z) para un nuevo estado p_1. Se sustituyen todas las transiciones en las que q' esté en F por (p_2, z) para un nuevo estado p_2 y se incluye p_2 en F. Después de realizar esto, $N(M) = \emptyset$ pero $L(M)$ no ha variado. Entonces se añaden estados y transiciones, de forma que, una vez que M entre en un estado final, vacíe su pila. Para todo $q \in F$, se añaden las transiciones

$$\Delta(q, \varepsilon, \gamma) = \{(p_3, \gamma)\}, \qquad \text{para todo } \gamma \in \Gamma$$
$$\Delta(p_3, \varepsilon, \gamma) = \{(p_3, \varepsilon)\}, \qquad \text{para todo } \gamma \in \Gamma - \{z\}$$
$$\Delta(p_3, \varepsilon, z) = \{(p_4, \varepsilon)\}$$

Después de hecho esto, se obtiene que $F = \{p_4\}$. El ADPND resultante, M', acepta cadenas en un único estado final con la pila vacía. Por tanto, $L(M') = N(M') = L(M)$.

Además se requiere que todas las transiciones sean de la forma

$$\Delta(q, a, A) = \{c_1, c_2, \ldots, c_n\}$$

donde cada $c_i = (p, \varepsilon)$ o $c_i = (p, BC)$. Es decir, cada transición incrementa o decrementa el contenido de la pila en un único símbolo.

Para construir un gramática independiente del contexto para un ADPND, invertiremos el proceso de la última sección para que una derivación simule los movimientos que un ADPND realiza para aceptar una cadena. Para ello, en cada etapa, la parte correspondiente a la cadena parcialmente generada formada por no terminales, reflejará el contenido de la pila, mientras que el prefijo que forman los terminales es la parte de la cadena que ya ha sido procesada. Una configuración de un ADPND contiene un estado además del contenido de la pila. Ya que la derivación que simulará el movimiento de un ADPND depende del estado actual, queremos que nuestra gramática independiente del contexto lleve también el rastro de los estados por los que se pasa. Una solución de este problema nos lleva a usar no terminales de la forma $[qAp]$, donde interpretaremos $[qAp] \stackrel{*}{\Rightarrow} w$ como la acción del ADPND correspondiente, que saca A de la pila y se mueve del estado q al estado p mientras consume la cadena de entrada w.

Ya que se requieren ciertos criterios para formar las transiciones, se simplifica el proceso de enumeración de las producciones correspondientes. Si $(q_j, \varepsilon) \in \Delta(q_i, a, A)$, la producción correspondiente es $[q_i A q_j] \rightarrow a$, ya que el ADPND pasa del estado q_i al estado q_j y desapila A de la pila sobre el símbolo de entrada a.

Por otro lado, si $(q_j, BC) \in \Delta(q_i, a, A)$, obsérvese que la entrada a produce que A sea eliminado de la pila, pero entonces B y C serían rechazados. Ya que el ADPND acepta cuando está la pila vacía, B y C deben ser eliminados necesariamente. Su eliminación se obtiene por medio de los cambios de estado apropiados. Por tanto, si $(q_j, BC) \in \Delta(q_i, a, A)$, incluiremos todas las producciones de la forma $[q_i A q_m] \rightarrow a\,[q_j B q_n]\,[q_n C q_m]$, donde q_n y q_m pueden ser cualquiera de los estados de Q.

Finalmente, tendremos como símbolo inicial $[szq_f]$, donde s es el estado inicial, z es el símbolo inicial de la pila y q_f es el (único) estado de aceptación del ADPND.

Ejemplo 3.8.2

Consideremos el ADPND cuyas transiciones son las siguientes

1. $\Delta(q_1, a, z) = \{(q_1, Az)\}$
2. $\Delta(q_1, a, A) = \{(q_1, AA)\}$
3. $\Delta(q_1, b, A) = \{(q_2, \varepsilon)\}$
4. $\Delta(q_2, b, A) = \{(q_2, \varepsilon)\}$
5. $\Delta(q_2, \varepsilon, A) = \{(q_2, \varepsilon)\}$
6. $\Delta(q_2, \varepsilon, z) = \{(q_3, \varepsilon)\}$

donde z es el símbolo inicial de la pila, $F = \{q_3\}$ y q_1 es el estado inicial. Obsérvese que este ADPND sólo pasa al estado de aceptación cuando la pila está va-

cía. Además, todas las transiciones son de la forma requerida. En una derivación de la gramática asociada, tendremos como símbolo inicial $[q_1zq_3]$ y las transiciones 3, 4, 5 y 6 se traducen fácilmente como las siguientes producciones:

$$[q_1Aq_2] \rightarrow b$$
$$[q_2Aq_2] \rightarrow b \mid \varepsilon$$
$$[q_2zq_3] \rightarrow \varepsilon$$

Las transiciones 1 y 2 generan las siguientes producciones:

De 1: $[q_1zq_1] \rightarrow a\,[q_1Aq_1]\,[q_1zq_1] \mid a\,[q_1Aq_2]\,[q_2zq_1] \mid a\,[q_1Aq_3]\,[q_3zq_1]$
$[q_1zq_2] \rightarrow a\,[q_1Aq_1]\,[q_1zq_2] \mid a\,[q_1Aq_2]\,[q_2zq_2] \mid a\,[q_1Aq_3]\,[q_3zq_2]$
$[q_1zq_3] \rightarrow a\,[q_1Aq_1]\,[q_1zq_3] \mid a\,[q_1Aq_2]\,[q_2zq_3] \mid a\,[q_1Aq_3]\,[q_3zq_3]$

De 2: $[q_1Aq_1] \rightarrow a\,[q_1Aq_1]\,[q_1Aq_1] \mid a\,[q_1Aq_2]\,[q_2Aq_1] \mid a\,[q_1Aq_3]\,[q_3Aq_1]$
$[q_1Aq_2] \rightarrow a\,[q_1Aq_1]\,[q_1Aq_2] \mid a\,[q_1Aq_2]\,[q_2Aq_2] \mid a\,[q_1Aq_3]\,[q_3Aq_2]$
$[q_1Aq_3] \rightarrow a\,[q_1Aq_1]\,[q_1Aq_3] \mid a\,[q_1Aq_2]\,[q_2Aq_3] \mid a\,[q_1Aq_3]\,[q_3Aq_3]$

La cadena *aabb* es aceptada por este autómata con la siguiente secuencia de configuraciones:

$$\begin{aligned}(q_1, aabb, z) &\vdash (q_1, abb, Az)\\ &\vdash (q_2, \varepsilon, z)\\ &\vdash (q_1, bb, AAz)\\ &\vdash (q_2, b, Az)\\ &\vdash (q_3, \varepsilon, \varepsilon)\end{aligned}$$

La derivación correspondiente en la gramática precedente es

$$\begin{aligned}[q_1zq_3] &\Rightarrow a\,[q_1Aq_2]\,[q_2zq_3]\\ &\Rightarrow aa[q_1Aq_2]\,[q_2Aq_2]\,[q_2zq_3]\\ &\Rightarrow aab\,[q_2Aq_2][q_2zq_3]\\ &\Rightarrow aabb\,[q_2zq_3]\\ &\Rightarrow aabb\end{aligned}$$

(Obsérvese que el "primer q" en cada cadena perteneciente a la derivación, corresponde al estado actual en la secuencia del ADPND. Además, las secuencias de símbolos "centrales" se corresponde con el contenido de la pila cuando es aceptada *aabb*).

A menudo, esta construcción crea gramáticas complicadas. Por ejemplo, en el caso precedente, los símbolos no terminales $[q_2zq_1]$, $[q_3zq_1]$, $[q_3zq_2]$, $[q_2Aq_1]$, $[q_3Aq_1]$, $[q_3Aq_2]$ y $[q_3zq_3]$, nunca aparecen en el lado izquierdo de una produc-

ción. Por tanto, dichos símbolos nunca podrán aparecer en una derivación de una cadena de terminales. A pesar de haber introducido una complejidad no deseable, esta técnica de construcción se puede usar para cualquier ADPND cuyas reglas de transición satisfacen las condiciones. Se puede aplicar las técnicas de simplificación vistas en la Sección 3.5 a la gramática resultante, con el fin de eliminar las partes inútiles. Obsérvese que, de la forma de construir una gramática a partir de un ADPND que satisfaga las condiciones dadas, se obtiene que

$$(q_i, uv, Ax) \overset{*}{\vdash} (q_j, v, x)$$

por medio de las operaciones que realiza un autómata de pila, entonces en la gramática resultante se tendrá

$$[q_i A q_j] \overset{*}{\Rightarrow} u$$

Es decir, si el ADPND elimina A de la pila cuando se lee la cadena u y pasa del estado q_i al estado q_j, entonces el no terminal $[q_i A q_j]$ puede derivar u.

A la inversa, obsérvese que, si en alguna derivación se tiene

$$[q_i A q_k] \overset{*}{\Rightarrow} a\,[q_j B q_r]\,[q_r C q_k]$$

entonces A puede ser sustituido por BC en la pila cuando se lea a y se pasará del estado q_i al estado q_j.

Por otro lado, si $[q_i A q_j] \Rightarrow a$ es una derivación con un único paso, entonces $\Delta(q_i, a, A)$ contiene (q_j, ε), con lo que A podrá ser desapilado al consumirse la cadena de entrada a. Por tanto, si $[q_i A q_j] \overset{*}{\Rightarrow} u$, la secuencia de cadenas de terminales y no terminales que forman la derivación se corresponde con la secuencia de movimientos que realiza un ADPND para el cual $(q_i, uv, Ax) \overset{*}{\vdash} (q_j, v, x)$. Por tanto, cualquier cadena de terminales generada por la gramática es aceptada por el ADPND, y viceversa. Enunciaremos el siguiente teorema:

Teorema 3.8.2. Si L es aceptado por un ADPND, entonces L es un lenguaje independiente del contexto.

Los autómatas de pila no deterministas nos proporcionan otra alternativa para caracterizar los lenguajes independientes del contexto. Es tan fácil probar que un lenguaje es un lenguaje independiente del contexto, demostrando que es aceptado por un ADPND, como probar que es independiente del contexto si es generado por una gramática independiente del contexto. A veces incluso es más fácil describir un ADPND que una gramática. Además, el modelo del ADPND

nos facilita la tarea de probar ciertas propiedades de los lenguajes independientes del contexto.

Teorema 3.8.3. Si L_1 es un lenguaje independiente del contexto y L_2 es un lenguaje regular, entonces $L_1 \cap L_2$ es un lenguaje independiente del contexto.

Demostración. Sea $M_1 = (Q_1, \Sigma_1, \Gamma, s_1, z, F_1, \Delta_1)$ un ADPND que acepta el lenguaje L_1. Sea $M_2 = (Q_2, \Sigma_2, s_2, F_2, \delta)$ un autómata finito *determinista* que acepta el lenguaje regular L_2. Construiremos un ADPND que acepte $L_1 \cap L_2$ mediante la combinación de M_1 y M_2. Este ADPND simulará paralelamente las acciones de M_1 y M_2 y aceptará una cadena cuando y sólo cuando M_1 y M_2 acepten una. Definiremos el ADPND $M = (Q, \Sigma, \Gamma, s, z, F, \Delta)$ como sigue:

$$Q = Q_{p1} \times Q_2$$
$$s = (s_1, s_2)$$
$$F = F_1 \times F_2$$
$$\Sigma = \Sigma_1 \cup \Sigma_2$$

y la regla de transición Δ se define de forma que se tiene que

$$((q_k, p_l), x) \in \Delta((q_i, q_j), a, b)$$

si y sólo si $\delta(p_j, a) = p_l$ y $(q_k, x) \in \Delta_1(q_i, a, b)$. Es decir,

$$\Delta((q_i, p_j), a, b) = \{((q_k, p_l), x) \mid (q_k, x) \in \Delta_1(q_i, a, b) \quad \text{y} \quad \delta(p_j, a) = \{p_l\}\}$$

Obsérvese que los estados de M se etiquetan mediante pares que representan los estados en las que estarán M_1 y M_2, respectivamente, tras procesar la entrada. Una cadena se acepta si provoca que M termine en un estado (q, p), donde $q \in F_1$ y $p \in F_2$. Es decir, se acepta si M_1 acepta la cadena acabando en el estado q y M_2 acepta la cadena acabando en el estado p. La inducción sobre el número de pasos de una computación prueba que $(s, w, z) \overset{*}{\vdash} ((q, p), \varepsilon, u)$ con $q \in F_1$ y $p \in F_2$, si y sólo si $(s_1, w, z) \overset{*}{\vdash} (q, \varepsilon, u)$ y $\delta(s_2, w) = p$. □

En la Sección 3.7, descubrimos que la intersección de lenguajes independientes del contexto no tenía que ser necesariamente un lenguaje independiente del contexto. Es decir, la familia de los lenguajes independientes del contexto no es cerrada bajo la intersección. El Teorema 3.8.3 prueba que esta familia es cerrada bajo la *intersección regular*.

Por ejemplo, el lenguaje $L = \{a^n b^n \mid n \geq 0 \text{ y } n \neq 100\}$ es un lenguaje independiente del contexto. Aunque dicha propiedad puede ser probada mediante la construcción de una gramática independiente del contexto que lo genere o de un

ADPND que lo acepte, es más elegante demostrarlo aplicando el teorema. Obsérvese que $L_1 = \{a^n b^n \mid n \geq 0\}$ es un lenguaje independiente del contexto. Además, sabemos que $L_2 = a^* b^*$ y $L_3 = \{a^{100} b^{100}\}$ son regulares. Por tanto, $L_4 = L_2 - L_3$ también es regular, ya que los lenguajes regulares son cerrados con respecto a la diferencia de lenguajes. Entonces tendremos que $L = L_1 \cap L_4$ y, por tanto, según el teorema, L es independiente del contexto.

El Teorema 3.8.3 también se puede usar para demostrar que algunos lenguajes no son independientes del contexto. Por ejemplo, sabemos que $\{a^n b^n c^n \mid n \geq 0\}$ no es independiente del contexto. Sea

$$L = \{w \in \{a, b, c\}^* \mid w \text{ contiene el mismo número de } aes, bes \text{ y } ces\}$$

Entonces L no es independiente del contexto, porque si lo fuera, entonces

$$\{a^n b^n c^n \mid n \geq 0\} = L \cap (a^* b^* c^*)$$

también lo sería.

Ejercicios de la Sección 3.8

3.8.1. Obtener una secuencia de descripciones instantáneas para el ADPND del Ejemplo 3.8.1, que acepte la cadena *babcbab*. ¿En qué punto fallaría el intento de aceptar la cadena *abcab*?

3.8.2. Obtener un ADPND que acepte el lenguaje generado por la siguiente gramática independiente del contexto:

$$S \rightarrow aAA$$
$$A \rightarrow bS \mid aS \mid a$$

3.8.3. Obtener un ADPND que acepte el lenguaje generado por la gramática independiente del contexto

$$S \rightarrow aSb \mid aSbb \mid \varepsilon$$

3.8.4. Construir una ADPND que acepte el lenguaje generado por la gramática independiente del contexto

$$S \rightarrow aABB \mid aAA$$
$$A \rightarrow aBB \mid \varepsilon$$
$$B \rightarrow bBB \mid A$$

3.8.5. Demostrar cómo se convierte un ADPND arbitrario en uno para el cual todas las transiciones son de la forma

$$\Delta(q, a, A) = \{c_1, c_2, \ldots, c_n\}, \text{ donde cada } c_i = (p, \varepsilon) \text{ o } c_i = (p, BC).$$

3.8.6. Obtener una derivación de a^3b^3 mediante la gramática del Ejemplo 3.8.2.

3.8.7. Aplicar la construcción precedente al ADPND dado por $M = (Q, \Sigma, \Gamma, s, z, F, \Delta)$, donde $Q = \{q_1, q_2, q_3, q_4\}$, $F = \{q_3\}$, $s = q_1$, $\Sigma = \{a, b\}$, $\Gamma = \{A, B, z\}$ y Δ como se muestra a continuación

$$\Delta(q_1, a, z) = \{(q_1, Az)\} \qquad \Delta(q_1, b, A) = \{(q_2, \varepsilon)\}$$
$$\Delta(q_4, \varepsilon, z) = \{(q_1, Az)\} \qquad \Delta(q_2, \varepsilon, z) = \{(q_3, \varepsilon)\}$$
$$\Delta(q_1, a, A) = \{(q_4, \varepsilon)\}$$

3.8.8. Obtener una gramática independiente del contexto que genere el mismo lenguaje que el ADPND $M = (Q, \Sigma, \Gamma, s, z, F, \Delta)$, donde $Q = \{q_1, q_2\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, z\}$, $s = q_1$, $F = \{q_2\}$ y Δ dado mediante

$$\Delta(q_1, a, z) = \{(q_1, az)\}$$
$$\Delta(q_1, b, a) = \{(q_1, aa)\}$$
$$\Delta(q_1, a, a) = \{(q_2, \varepsilon)\}$$

3.8.9. Obtener una grámatica independiente del contexto para el lenguaje aceptado por el ADPND $M = (Q, \Sigma, \Gamma, s, z, F, \Delta)$, donde $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $\Gamma = \{A, z\}$, z es el símbolo inicial de la pila, $F = \{q_3\}$, y Δ dado por

$$\Delta(q_1, a, z) = \{(q_1, Az)\}$$
$$\Delta(q_1, a, A) = \{(q_1, A)\}$$
$$\Delta(q_1, b, A) = \{(q_2, \varepsilon)\}$$
$$\Delta(q_2, \varepsilon, z) = \{(q_3, \varepsilon)\}$$

Obsérvese que este ADPND no satisface las dos condiciones requeridas para la construcción de una GIC.

3.8.10. Sea $M = (Q, \Sigma, \Gamma, s, z, F, \Delta)$, donde $Q = \{q_1, q_2, q_3, q_4\}$, $\Sigma = \{a, b\}$, $\Gamma = \{X, z\}$, $s = q_1$, z es el símbolo inicial de la pila, $F = \emptyset$ (es decir, este ADPND sólo acepta cadenas cuando la pila se vacía), y Δ dado por

$$\Delta(q_1, b, z) = \{(q_1, Xz)\} \qquad \Delta(q_1, \varepsilon, z) = \{(q_1, \varepsilon)\}$$
$$\Delta(q_1, b, X) = \{(q_1, XX)\} \qquad \Delta(q_2, b, X) = \{(q_2, \varepsilon)\}$$
$$\Delta(q_1, a, X) = \{(q_2, X)\} \qquad \Delta(q_2, a, z) = \{(q_1, z)\}$$

Después de realizar los cambios necesarios para que este ADPND satisfaga las condiciones requeridas para construir una gramática independiente del contexto, obtener una gramática independiente del contexto para el lenguaje aceptado por este autómata.

3.8.11. Obtener una gramática independiente del contexto que genere el lenguaje aceptado por el ADPND

$$M = (Q, \Sigma, \Gamma, s, z, F, \Delta) \quad \text{donde}$$
$$Q = \{q_1, q_2\}$$
$$\Sigma = \{a, b\}$$
$$\Gamma = \{A, z\}, \quad \text{y } z \text{ es el símbolo inicial de la pila}$$
$$s = q_1$$
$$F = \{q_2\}$$

y Δ viene dado por

$$\Delta(q_1, a, z) = \{(q_1, Az)\}$$
$$\Delta(q_1, b, A) = \{(q_1, AA)\}$$
$$\Delta(q_1, a, A) = \{(q_2, \varepsilon)\}$$

3.8.12. Demostrar que

$$\{a^n b^n \mid n \geq 0 \text{ y } n \text{ no es múltiplo de } 5\}$$

es un lenguaje independiente del contexto.

3.8.13. Sea $L = \{a^i b^j c^k \mid i \neq j \text{ o } j \neq k\}$

(a) Probar que L es un lenguaje independiente del contexto.

(b) Probar que si Σ es cualquier alfabeto para el cual $L \subseteq \Sigma^*$, entonces $\Sigma^* - L$ no es independiente del contexto. (*Indicación:* Considérese el lenguaje $a^* b^* c^*$).

3.8.14. Considérese el lenguaje formado por todas las cadenas sobre $\{a, b\}$ que contienen el mismo número de *aes* que de *bes*, pero que no contienen la subcadena *aab*. ¿Es un lenguaje independiente del contexto?

3.8.15. Probar que el conjunto de los lenguajes independientes del contexto no es cerrado con respecto a la complementación. Es decir, en general no es cierto que L y $\Sigma^* - L$ sean lenguajes independientes del contexto.

3.9 FORMA NORMAL DE GREIBACH

La forma normal de Chomsky es una de las formas normales que se usan para las gramáticas independientes del contexto. Otra forma normal que tiene una gran importancia teórica y práctica es la *forma normal de Greibach*. En la forma norma de Greibach, se restringe la posición en la que pueden aparecer los terminales y los no terminales. Empezaremos presentando dos resultados usuales.

Teorema 3.9.1. Si $A \to \alpha B \gamma$ es una producción de una gramática independiente del contexto y si $B \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$ son todas las producciones que tienen a B en

su lado izquierdo, entonces la producción $A \rightarrow \alpha B \gamma$ se puede reemplazar por $A \rightarrow \alpha\beta_1\gamma | \alpha\beta_2\gamma | \ldots | \alpha\beta_m\gamma$ sin que varíe el lenguaje generado por la gramática.

Demostración. Sea G la gramática original y sea G' la gramática que resulta de la transformación. Tenemos que probar que $L(G) = L(G')$. Primero demostraremos que $L(G') \subseteq L(G)$. Supongamos que $w \in L(G')$. Si la derivación de w no contiene ninguna de las producciones $A \rightarrow \alpha\beta_i\gamma$, entonces $w \in L(G)$ y queda probado. Por otro lado, si la derivación de w usa una producción $A \rightarrow \alpha\beta_i\gamma$, entonces en G se puede usar $A \Rightarrow \alpha B\gamma \Rightarrow \alpha\beta_i\gamma$ para derivar w. Por tanto, $w \in L(G)$. Entonces se tiene $L(G') \subseteq L(G)$.

Para demostrar que $L(G) \subseteq L(G')$, consideremos que $w \in L(G)$. Si $A \rightarrow \alpha B\gamma$ no se usa en la derivación de w, entonces, puesto que todas las otras producciones de G están en G', $w \in L(G')$ y quedará probado. Si $A \rightarrow \alpha B\gamma$ se usa en la derivación de w, entonces en algún momento, B debe ser reemplazado por uno de los β_i. Esto se puede hacer de forma inmediata unificando ambos pasos en uno $A \Rightarrow \alpha\beta_i\gamma$. Por tanto, w es derivable en G', con lo que $w \in L(G')$ y se obtiene que $L(G) \subseteq L(G')$. □

Por ejemplo, consideremos la gramática independiente del contexto cuyas producciones son

$$S \rightarrow a | aaB | BAb | abBc$$
$$B \rightarrow baabS | bba$$

Si realizamos la sustitución presentada en el Teorema 3.9.1, obtendremos la gramática

$$S \rightarrow a | aabaabS | aabba | baabSab | bbaab$$
$$| abbaabSc | abbbac$$
$$B \rightarrow baabS | bba$$

Obsérvese que las producciones de B, $B \rightarrow baabS | bba$ no desaparecen, pero no podrán formar parte de ninguna derivación. Sin embargo, con el fin de simplificar la gramática resultante, una de las técnicas de simplificación vistas en la Sección 3.5 podría eliminar dichas producciones inútiles.

Una producción de la forma $A \rightarrow \alpha A$, donde $\alpha \in (N \cup \Sigma)^*$, se conoce como *recursiva por la derecha*. De forma semejante, una producción *recursiva por la izquierda* será de la forma $A \rightarrow A\alpha$. Las producciones recursivas por la derecha producen árboles que se expanden por la derecha, mientras que los árboles correspondientes a las producciones recursivas por la izquierda se expanden por la izquierda. En muchas aplicaciones correspondientes a las gramáticas, no es deseable que exista recursividad por la izquierda. El siguiente teorema proporciona

una forma de eliminar la recursividad por la izquierda en las gramáticas independientes del contexto.

Teorema 3.9.2. Sea G una gramática independiente del contexto y A un no terminal de G. Si $A \to A\alpha_1 | A\alpha_2 | \ldots | A\alpha_n$ son todas las producciones para A, que son recursivas por la izquierda, y si $A \to \beta_1 | \beta_2 | \ldots | \beta_m$ son las restantes producciones para A, entonces se puede construir una gramática equivalente introduciendo un nuevo no terminal Z y reemplazando *todas* las producciones precedentes por

$$A \to \beta_1 | \beta_2 | \ldots | \beta_m | \beta_1 Z | \beta_2 Z | \ldots | \beta_m Z$$
$$Z \to \alpha_1 | \alpha_2 | \ldots | \alpha_n | \alpha_1 Z | \alpha_2 Z | \ldots | \alpha_n Z$$

Demostración. Obsérvese que en ambos casos las cadenas derivables de A mediante el uso de una o más producciones, forman un lenguaje regular

$$\{\beta_1, \beta_2, \ldots, \beta_m\} \{\alpha_1, \alpha_2, \ldots, \alpha_n\}^*. \quad \square$$

Considérese la gramática independiente del contexto dada mediante

$$S \to Sa | Sb | cA$$
$$A \to Aa | a | \varepsilon$$

Obsérvese que hay producciones recursivas por la izquierda con S y A en su lado izquierdo. Aplicando el teorema al no terminal S e introduciendo un no terminal nuevo Z_1, se obtiene la siguiente gramática transformada que es independiente del contexto

$$S \to cA | cAZ_1$$
$$Z_1 \to a | b | aZ_1 | bZ_1$$
$$A \to Aa | a | \varepsilon$$

Entonces, aplicando el teorema al no terminal A e introduciendo un no terminal nuevo Z_2, se obtienen las producciones

$$S \to cA | cAZ_1$$
$$Z_1 \to a | b | aZ_1 | bZ_1$$
$$A \to a | aZ_2 | \varepsilon | Z_2$$
$$Z_2 \to a | aZ_2$$

Obsérvese que al eliminar las producciones recursivas por la izquierda se introducen nuevos terminales y producciones recursivas por la derecha.

Definición 3.9.3. Una gramática independiente del contexto está en *forma normal de Greibach* (FNG) si todas las producciones son de la forma $A \to a\alpha$, donde a es un símbolo terminal y $\alpha \in (\Sigma \cup N)^*$.

Obsérvese que esta formal normal requiere que toda producción tenga un símbolo del alfabeto como primer símbolo del lado derecho de las producciones. Por tanto una gramática en FNG no puede tener producciones recursivas por la izquierda. Es más, puesto que cada producción requiere que haya al menos un símbolo del alfabeto, una gramática independiente del contexto en FNG sólo puede generar lenguajes no vacíos que no contengan ε.

Es posible construir una gramática independiente del contexto en forma normal de Greibach para cualquier gramática independiente del contexto que no contenga ε. El algoritmo para realizar esto consiste en diferentes etapas.

Supongamos que L es un lenguaje independiente del contexto no vacío que no contiene ε. Primero, sea $G = (\Sigma, N, S, P)$ una gramática independiente del contexto en forma normal de Chomsky que genera L. Supongamos que $N = \{A_1, A_2, ..., A_n\}$, donde $A_1 = S$. Entonces modificamos las producciones de forma que si $A_r \to A_s\alpha$ es una producción, entonces $r < s$. Supongamos que hemos modificado las producciones de forma que para $1 \leq i \leq k$, si $A_i \to A_j\alpha$, entonces $i < j$. Demostraremos como modificar las producciones para A_{k+1}.

Si $A_{k+1} \to A_j\alpha$ es una producción con $k+1 > j$, se genera un nuevo conjunto de producciones para reemplazar las A_j que aparecen en el lado derecho de las producciones, por el lado derecho de todas las producciones de la forma $A_j \to \beta$, como se vio en el Teorema 3.9.1. Cuando se realicen dichas sustituciones, se obtendrán producciones de la forma $A_{k+1} \to A_r\alpha$. Puede ocurrir que $k+1 \leq r$ o que $r < k+1$. Si $k+1 \leq r$, se tendrá una producción de la forma deseada. Si $r < k+1$, deberemos repetir el proceso. Puesto que solamente hay k índices menores que $k+1$, después de repetir el proceso $k-1$ veces, se habrán eliminado todas las producciones de la forma $A_{k+1} \to A_r\alpha$ para las cuales $r < k+1$. Por tanto, las producciones que quedarán, serán todas de la forma $A_{k+1} \to A_r\alpha$, con $k+1 \leq r$. Obsérvese que las producciones para las cuales $k+1 = r$ son producciones recursivas por la izquierda que pueden ser eliminadas mediante el Teorema 3.9.2 introduciendo un no terminal nuevo Z_{k+1}.

Repetiremos el proceso para cada uno de los no terminales originales desde A_1 hasta A_n. Por tanto sólo se tendrán producciones de las tres formas siguientes:

1. $A_k \to A_j\alpha$, con $k < j$
2. $A_k \to a\alpha$, para $a \in \Sigma$
3. $Z_k \to \alpha$, para $\alpha \in (N \cup \{Z_1, Z_2, ..., Z_n\})^*$

Obsérvese que, puesto que A_n es el no terminal con mayor índice, todas las producciones para A_n deben ser del tipo 2. Es decir, en el extremo izquierdo del lado derecho de una producción para A_n, debe haber un símbolo terminal. También, toda producción para A_{n-1} debe tener en el extremo izquierdo del lado derecho, un terminal o A_n. Si tiene A_n, puede ser reemplazado por el lado derecho de una de sus producciones como se hace en el Teorema 3.9.1. Dichas producciones comenzarán con un terminal. Luego procederemos a transformar las producciones correspondientes a A_{n-2}, A_{n-3} y así sucesivamente, hasta que el lado derecho de todas las producciones correspondientes a los no terminales originales comiencen con un terminal.

Finalmente, consideraremos las producciones para $Z_1, Z_2, ..., Z_n$. Puesto que inicialmente teníamos una gramática independiente del contexto en forma normal de Chomsky y sólo hemos aplicado los Teoremas 3.9.1 y 3.9.2, ninguna de las producciones $Z_i \rightarrow \alpha$ tendrá otro Z_j en el extremo izquierdo de su lado derecho. Por tanto, todas la producciones correspondientes a los Z_i tendrán terminales o A_i al principio de su lado derecho. Para los que tengan al principio del lado derecho alguna A_k, aplicaremos una vez más, el Teorema 3.9.1 y todas las producciones resultantes estarán en la forma deseada.

La transformación vista nos da pie para enunciar el siguiente teorema:

Teorema 3.9.4. Todo lenguaje independiente del contexto no vacío, que no contiene la palabra vacía ε, se puede generar mediante una gramática independiente del contexto $G = (N, \Sigma, S, P)$ en la cual todas las producciones son de la forma $A \rightarrow aw$ para $A \in N$, $a \in \Sigma$ y $w \in N^*$.

Consideremos la gramática independiente del contexto en forma normal de Chomsky (cuyos no terminales han sido etiquetados convenientemente)

$$A_1 \rightarrow A_2 A_2 \mid a$$
$$A_2 \rightarrow A_1 A_2 \mid b$$

Obsérvese que las producciones $A_1 \rightarrow A_2 A_2 \mid a$ ya se encuentran en la forma necesaria para realizar la primera etapa. (Las producciones deben satisfacer $A_i \rightarrow A_j \alpha$ para $i < j$ sólo cuando hay un no terminal en el lado derecho. Por tanto $A_1 \rightarrow a$ es aceptable). Consideraremos las producciones correspondientes a A_2. La producción $A_2 \rightarrow b$ se acepta, pero $A_2 \rightarrow A_1 A_2$ no. Debemos substituir A_1, obteniendo las producciones $A_2 \rightarrow A_2 A_2 A_2 \mid a A_2$. Eliminamos la recursividad por la izquierda y obtenemos el conjunto de producciones

$$A_1 \rightarrow A_2 A_2 \mid a$$
$$A_2 \rightarrow aA_2 \mid aA_2 Z \mid b \mid bZ$$

Finalmente, substituimos A_2 de forma apropiada para que el lado derecho de todas las producciones comience con un terminal. Esto produce

$$\begin{aligned} A_1 &\to aA_2A_2 \mid aA_2ZA_2 \mid bA_2 \mid bZA_2 \mid a \\ A_2 &\to aA_2 \mid aA_2Z \mid b \mid bZ \\ Z &\to aA_2A_2 \mid aA_2ZA_2 \mid bA_2 \mid bZA_2 \mid aA_2A_2Z \\ &\quad \mid aA_2ZA_2Z \mid bA_2Z \mid bZA_2Z \end{aligned}$$

Obsérvese que todas las producciones de la forma dada en el Teorema 3.9.4 son necesariamente de la forma apropiada para la FNG. Por tanto, se obtiene el siguiente corolario:

Corolario 3.9.5. Todo lenguaje L independiente del contexto y no vacío, que no contenga ε, puede ser generado mediante una gramática independiente del contexto en forma normal de Greibach.

Una definición alternativa para la FNG requiere que todas las producciones sean de la forma $A \to aw$ para $A \in N$, $a \in \Sigma$ y $w \in N^*$. Las dos definiciones de FNG son equivalentes (Ejercicio 3.9.5).

Ejercicios de la Sección 3.9

3.9.1. Eliminar la recursividad por la izquierda de

$$\begin{aligned} S &\to Sa \mid aAc \mid c \\ A &\to Ab \mid ba \end{aligned}$$

3.9.2. Eliminar la recursividad por la izquierda de $S \to aSb \mid SS \mid \varepsilon$.

3.9.3. En las observaciones anteriores al Teorema 3.9.3, la afirmación "ninguna de las producciones $Z_i \to \alpha$, tienen otro Z_j en el extremo izquierdo del lado derecho de la producción", depende de que el lado derecho de todas las producciones correspondientes a A_i empiece con un terminal o con $A_j A_k$ para algún j y algún k. Probarlo por inducción sobre el número de aplicaciones del Teorema 3.9.1 y 3.9.2.

3.9.4. Pasar a forma normal de Greibach

(a) $S \to aSb \mid ab$

(b) $S \to AA \mid a$

$A \to SS \mid b$

(c) $S \to Sa \mid Sb \mid cA$

$A \to Aa \mid a \mid \varepsilon$

3.9.5. Probar que la siguiente definición de forma normal es equivalente a la Definición 3.9.3: Una gramática independiente del contexto $G = (N, \Sigma, S, P)$ está en forma normal de Greibach si toda producción de P es de la forma $A \rightarrow aw$ para $A \in N, a \in \Sigma$ y $w \in N^*$.

PROBLEMAS

3.1. Consideremos el lenguaje $L = \{a^i b^i c^i \mid i \geq 1\}$ del Ejercicio 3.6.1(a). Este lenguaje no es un lenguaje independiente del contexto puesto que, para un k dado, podemos elegir $z = a^k b^k c^k$ y demostrar que ninguna elección posible de las subcadenas u, v, w, x e y satisface el lema de bombeo (3.6.1). Este lenguaje presenta uno de los mayores inconvenientes que tiene el lema de bombeo: no hay forma de decir qué parte de la cadena puede considerarse como la parte uwx. Esta incertidumbre nos lleva a considerar una variedad de subcasos que pueden resultar tediosos, a pesar de usar una técnica efectiva. En este primer ejercicio trataremos de presentar una versión más fuerte del lema de bombeo que nos permita controlar el enfoque que se dé a las partes de la cadena que son bombeadas.

Sea $G = (N, \Sigma, S, P)$ una gramática independiente del contexto en forma normal de Chomsky con $|N| = k$, y sea $n = 2^k + 1$. Supongamos que $z \in L(G)$ con $|z| > n$, y que "marcamos" al menos n posiciones de z (es decir, queremos distinguir las partes marcadas de las que no lo están). Queremos encontrar un camino en el árbol de derivación para w, análogo al visto en el lema de bombeo. Construiremos el camino partiendo de la raíz y descendiendo por la izquierda o por la derecha. Descenderemos por el subárbol izquierdo o por el subárbol derecho para determinar que subárbol genera el mayor número de posiciones de z que están marcadas. El camino resultante se extiende desde la raíz hasta una de las posiciones marcadas.

Obsérvese que cualquiera de los nodos del árbol de análisis (y que están presentes en el camino que hemos encontrado), tienen cero, uno o dos hijos, y cada uno de los hijos es, a su vez, raíz de un subárbol que puede generar o no posiciones marcadas. Si un nodo del árbol tiene dos hijos que son raíz de subárboles que generan posiciones marcadas, se llamará *punto de rama*.

1. Probar que, para cada punto de rama presente en el camino que hemos construido, tenemos al menos la mitad de descendientes marcados que para el punto de rama anterior.

2. Probar que el camino construido contiene al menos $k + 1$ puntos de rama.

El resultado visto es un resultado más débil que el que se debe a W. Ogden. Nos referiremos a él como "lema de Ogden".

Lema de Ogden. Sea L un lenguaje independiente del contexto. Entonces existe una constante n para la cual, si $z \in L$ y tenemos al menos n posiciones de z marcadas, podemos escribir que $z = uvwxy$, de forma que

1. vx tiene al menos una posición marcada.
2. vwx tiene como máximo n posiciones marcadas.
3. Para todo $i \geq 0$, $uv^i wx^i y \in L$.

3. Demostrar el lema de Ogden. *Indicación:* Por el Ejercicio 2, hay al menos $k+1$ puntos de rama en un camino elegido convenientemente. Aplíquese la demostración del lema de bombeo para el último de los $k+1$.

3.2. En los problemas siguientes se puede usar el lema de Ogden.

1. Aplicar el lema de Ogden para demostrar que el lenguaje

$$\{a^i b^i c^i \mid i \geq 1\}$$

no es independiente del contexto.

2. Usar el lema de Ogden para demostrar que

$$\{a^i b^j c^k d^l \mid i = 0 \text{ o } j = k = l\}$$

no es un lenguaje independiente del contexto.

3. Usar el lema de Ogden para demostrar que

$$\{a^i b^j c^k \mid i < j < k\}$$

no es independiente del contexto.

4. ¿Es un lenguaje independiente del contexto el lenguaje

$$\{a^i b^j c^k \mid i \neq j,\ j \neq k \text{ e } i \neq k\}?$$

¿Por qué o por qué no? ¿Qué ocurre con

$$\{a^i b^j c^k \mid i \neq j \text{ y } j \neq k\}?$$

3.3. Sea $L = \{ww \mid w \in \{a, b\}^*\}$. En el Ejercicio 3.6.1 (g) se propuso que se demostrará que L no era independiente del contexto. Demostrar que $\overline{L}$, el complemento de L en $\{a, b\}^*$, es independiente del contexto. *Indicación:* Obsérvese que cualquier cadena de longitud impar sobre $\{a, b\}^*$, automáticamente pertenece a L. Por otro lado, cualquier cadena de longitud par de $\overline{L}$ se puede escribir como $u_1 x_1 v_1 u_2 x_2 v_2$, donde x_1 y x_2 están en $\{a, b\}$, $x_1 \neq x_2$ y $|u_1| = |u_2|$ y $|v_1| = |v_2|$. Considérese la unión de dos lenguajes independientes del contexto, uno de los cuales contenga sólo las cadenas de longitud impar sobre $\{a, b\}^*$ y el otro contenga las cadenas de la forma $u_1 x_1 v_1 u_2 x_2 v_2$.

3.4. Un tipo de autómata similar al ADPND, es el *autómata de pila con dos pilas* y es una 7-tupla $M = (Q, \Sigma, \Gamma, \Delta, s, F, z)$. En éste, Q, Σ, Γ, s y F son los mismos que en los ADPND, pero la relación de transición Δ es de la forma

$$\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times \Gamma \times Q \times \Gamma^* \times \Gamma^*$$

Obsérvese que, cuando describimos una transición de Δ, se tiene en cuenta un símbolo de cada una de las pilas. Por ejemplo, una transición de la forma $\Delta(q, \sigma, \tau_1, \tau_2) = \{(p, \alpha, \varepsilon)\}$ cambiaría del estado q al estado p, sustituiría el símbolo τ_1 de la primera pila por el símbolo (o cadena) α, y desapilaría el símbolo τ_2 de la segunda pila.

1. Probar que cualquier lenguaje independiente del contexto es aceptado por un autómata de pila con dos pilas.
2. Probar que el lenguaje $L = \{a^i b^i c^i \mid i \geq 0\}$ es aceptado por un ADP con dos pilas.
3. Describir una técnica por medio de la cual el lenguaje L sea reconocido por un ADP con dos pilas
$$L = \{a^i b^i c^i d^i \mid i \geq 0\}$$

3.5. Se dice que el determinismo está presente en un autómata cuando no se puede elegir cómo se comportará en cada estado. Una condición que debe cumplirse para que un autómata de pila sea determinista es que no puede haber más de un elemento en los conjuntos $\Delta(q, \sigma, \tau)$. Sin embargo, esto no es suficiente para garantizar el determinismo. Considérese el ADP que contiene las transiciones

$$\Delta(q, a, A) = \{(p, w)\}$$
$$\Delta(q, \varepsilon, A) = \{(p', w')\}$$

Obsérvese que este ADP tiene que elegir cómo comportarse una vez que se encuentra en el estado q. Por tanto, incluso si todos los conjuntos $\Delta(q_i, \sigma, \tau)$ contienen un único elemento, puede seguir habiendo no determinismo. Definimos un *autómata de pila determinista* (ADPD) como un ADP en el cual no hay ninguna configuración para la cual el ADP tenga que elegir entre más de un movimiento. En otras palabras, un ADP es determinista si (a) cada $\Delta(q, a, A)$ tiene un elemento como máximo, y (b) si $\Delta(q, a, A) \neq \emptyset$, entonces $\Delta(q, \varepsilon, A) = \emptyset$. Un lenguaje independiente del contexto es un *lenguaje independiente del contexto determinista* (LICD) si es aceptado por un ADPD.

1. Probar que todo lenguaje regular es un LICD.
2. Probar que no todo LICD es regular.
3. Obviamente, cualquier LICD es un LIC ya que todo ADPD es un ADPND. Usar esto para demostrar que no todo LIC es un LICD. *Indicación:* En el Problema 3.3, se obtuvo un lenguaje independiente del contexto cuyo complemento no era independiente del contexto.

4

Máquinas de Turing

4.1 DEFINICIONES BÁSICAS

Recordemos que la colección de los lenguajes regulares constituye un subconjunto propio de los lenguajes independientes del contexto. Por tanto, se puede decir que los autómatas finitos son menos potentes que los autómatas de pila, con respecto a la capacidad que tienen para aceptar lenguajes. Por otro lado, hay muchos lenguajes que no son independientes del contexto, algunos de los cuales son bastante sencillos, tales como

$$\{a^n b^n c^n \mid n \geq 0\}$$

y

$$\{ww \mid w \in \Sigma^*\}$$

En este capítulo vamos a estudiar un tercer tipo de dispositivo para reconocimiento de lenguajes, la máquina de Turing. Las máquinas de Turing son más generales que cualquier autómata finito y cualquier autómata de pila, debido a que ellas pueden reconocer tanto los lenguajes regulares como los lenguajes independientes del contexto y, además, muchos otros tipos de lenguajes. Aunque sean más potentes que los dos tipos de autómatas anteriores, todos ellos son bastante similares con respecto a los componentes y las acciones que realizan.

Cuando pasamos de los autómatas finitos a los autómatas de pila, introdujimos un dispositivo para almacenamiento de memoria, la pila, que proporcionó la

posibilidad de "recordar" la cantidad de información necesaria para el reconocimiento de los lenguajes independientes del contexto. Una pila es bastante restrictiva debido a que el acceso a la información que tenemos está limitado a la cima de la pila. Para poder acceder a los datos que se encuentran debajo de la cima, necesitamos sacar el dato de la cima —dato que puede que no deseemos que se pierda.

Por ejemplo, si pretendemos reconocer $\{a^n b^n c^n \mid n \geq 0\}$ por medio de un autómata de pila no determinista, deberíamos contar el numero de *aes* que aparecen en la cadena de entrada introduciendo algunos símbolos en la pila. Después, para contar las *bes*, deberíamos desapilar los símbolos de la pila. Y, cuando llegue el momento de contar las *ces*, la pila estará vacía —nuestro recuento se habrá perdido. El problema subyacente no es la carencia de memoria, sino la forma en la que está organizada.

Supongamos que permitimos que la pila sea "recorrida" sin necesidad de desapilar nada. Es decir, igual que antes, se puede apilar y desapilar pero, además, permitimos que los datos que se encuentran bajo la cima puedan ser consultados. Podríamos reconocer $\{a^n b^n c^n \mid n \geq 0\}$ apilando un símbolo en la pila por cada a, como hacíamos antes. Para reconocer las *bes* podríamos situar un apuntador que apunte a la cima de la pila, de forma que por cada b el apuntador descienda por la pila. De esta forma, cuando leamos la parte de las *ces* que contiene la cadena de entrada, podremos usar el recuento realizado anteriormente, ya que no ha sido destruido. La capacidad de recorrer la pila añade una capacidad substancial al proceso de reconocimiento.

Aunque se debería experimentar con distinto tipos de organización de la memoria y sus operaciones, la organización introducida por las máquinas de Turing es bastante sencilla. Consiste en una colección de celdas de almacenamiento que se extiende infinitamente en ambas direcciones —esencialmente es una *cinta* infinita. Cada celda es capaz de almacenar un único símbolo. La colección no tiene una celda primera ni última y, por tanto, tiene una capacidad de almacenamiento ilimitada. A los contenidos de las celdas se puede acceder en cualquier orden. Además, tendrá, asociada con la cinta, una cabeza de lectura/escritura que puede moverse sobre la cinta y por cada movimiento leerá o escribirá un símbolo.

Veamos la siguiente definición:

Definición 4.1.1. Una *máquina de Turing* es una 7-tupla $M = (Q, \Sigma, \Gamma, s, b, F, \delta)$, donde

Q es un conjunto finito de estados

Σ es un alfabeto de entrada

Γ es una alfabeto llamado *alfabeto de la cinta*

$s \in Q$ es el estado inicial

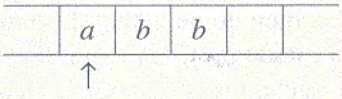
$b \in \Gamma$ es el símbolo *blanco* (y no está en Σ)

$F \subseteq Q$ es el conjunto de estados finales o de aceptación

$\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ es una *función parcial* que se llama función de transición

En esta definición se supone que el valor inicial de todas las celdas de la cinta es el símbolo b. La definición requiere que $b \notin \Sigma$. Generalmente, permitimos que $\Sigma \subseteq \Gamma - \{b\}$. La función de transición δ transforma pares (q, σ) formados por el estado actual y los símbolos de la cinta en ternas de la forma (p, t, X), donde p es el estado siguiente, t es el símbolo escrito en la cinta y X es un movimiento de lectura/escritura de la cabeza, que puede ser L o R, según que el movimiento sea hacia la izquierda o hacia la derecha (nos imaginamos que la cinta se extiende de izquierda a derecha).

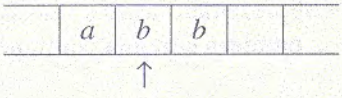
Por ejemplo, la transición $\delta(q_1, a) = (q_5, b, R)$ provoca que la máquina de Turing pase de una configuración



Posición actual de la cabeza *l/e*

Estado interno q_1

a la configuración



Posición actual de la cabeza *l/e*

Estado interno q_1

Haremos hincapié en que una función parcial no está necesariamente definida para todo elemento del conjunto *del que* se realiza la transformación. Por tanto, puede que δ no tenga una imagen para algún par $Q \times \Gamma$.

Obsérvese que las transiciones dependen únicamente del estado actual y del contenido de la celda sobre la que se encuentre la cabeza de lectura/escritura. Por tanto, cualquier cadena de entrada se debe presentar a la máquina sobre su

cinta. Esto es debido a que se requiere que $\Sigma \subseteq \Gamma - \{b\}$ (y que b no es un símbolo de entrada).

Consideremos la máquina de Turing definida mediante

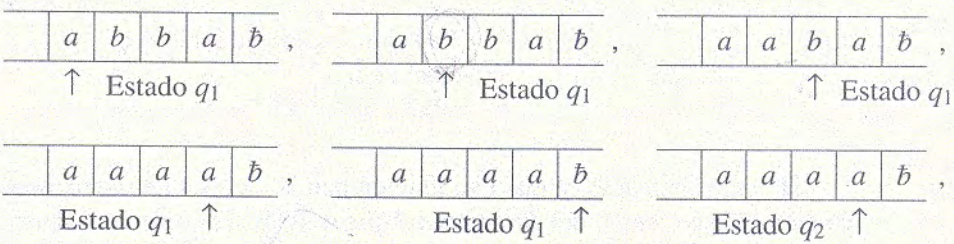
$$Q = \{q_1, q_2\}$$
$$\Sigma = \{a, b\}$$
$$\Gamma = \{a, b, b\}$$
$$F = \{q_2\}$$
$$s = q_1$$

y δ dado por

$$\delta(q_1, a) = (q_1, a, R)$$
$$\delta(q_1, b) = (q_1, a, R)$$
$$\delta(q_1, b) = (q_2, b, L)$$

Esta maquina empieza sus operaciones en el estado q_1. Si el contenido de la celda de la cinta sobre la que se encuentra la cabeza de lectura/escritura es a, la transición que se puede aplicar es $\delta(q_1, a) = (q_1, a, R)$. La máquina de Turing sobreescribirá la a que está en la cinta con otra a (es decir, no se produciría ningún cambio en el contenido de la cinta), se moverá una celda hacia la derecha y permanecerá en el estado q_1. Cualquiera de las siguientes *aes* que haya en la cinta, ya sea en esta nueva posición como en cualquier otra que esté a la derecha, no se cambiará, sin embargo las *bes* serán sustituidas por *aes* (por medio de la transición $\delta(q_1, b) = (q_1 a, R)$). Si la máquina de Turing encuentra un blanco (el símbolo b), se moverá una celda hacia la izquierda y pasará al estado final q_2. No hay ninguna transición desde el estado q_2, con lo que la máquina de Turing parará una vez que llega a ese estado.

Estos gráficos representan las distintas etapas del proceso para la máquina de Turing dada, que empieza con una configuración inicial muy sencilla:



Representar las configuraciones de una máquina de Turing de esta forma es bastante pesado. Cualquier configuración viene determinada por el estado actual, el contenido de la cinta y la posición de la cabeza de lectura/escritura sobre

la cinta. A continuación veremos las dos notaciones que más se emplean para representar esta información de una manera más conveniente. La primera representa una configuración como un par $(q_i, w_1\underline{\sigma}w_2)$, donde q_i es el estado actual, w_1 es la cadena de la cinta que precede a la celda sobre la que se encuentra la cabeza de entrada/salida, $\underline{\sigma}$ es el símbolo de la cinta sobre el que se encuentra la cabeza de entrada/salida y w_2 es la cadena que hay a continuación de la cabeza de entrada/salida. Por tanto, en el ejemplo anterior la configuración inicial de nuestra maquina de Turing podría ser $(q_1, \underline{b}abba)$, la segunda sería $(q_1, a\underline{b}ba)$ y así sucesivamente. Otra notación alternativa viene dada por una cadena $a_1a_2 \ldots a_{k-1}\, q_i\, a_k \ldots a_n$ que representa a la configuración (q_i, wa_ku); es decir, la cabeza de entrada/salida se coloca sobre la celda que contiene a_k y el estado actual es q_i. Obsérvese que la cadena $a_1a_2 \ldots a_{k-1}q_i\, a_k \ldots a_n$ indica que la cabeza de entrada/salida se encuentra sobre el símbolo de la cinta que aparece *siguiendo* al estado. Por tanto, la primera de las dos configuraciones del ejemplo anterior se puede representar como q_1abba y aq_1bba. Usaremos estas dos notaciones indistintamente. Las configuraciones de una máquina de Turing se conocen como *descripciones instantáneas* (DIS)

Sea cual sea la notación que utilicemos, denotaremos el paso de una configuración a otra por medio del símbolo ya familiar $\vdash$. Por tanto, en el ejemplo anterior se tiene que

$$(q_1, \underline{a}bba) \vdash (q_1, a\underline{b}ba) \vdash (q_1, aa\underline{b}a) \vdash (q_1, aaa\underline{a})$$
$$\vdash (q_1, aaaa\underline{b}) \vdash (q_2, aaa\underline{a})$$

o

$$q_1abba \vdash aq_1bba \vdash aaq_1ba \vdash aaaq_1a \vdash aaaaq_1b$$
$$\vdash aaaq_2a$$

Las notaciones $\overset{*}{\vdash}$ y $\overset{+}{\vdash}$ tienen el significado usual "cero o más" o "una o más", respectivamente.

Veamos otro ejemplo en el que consideraremos la siguiente máquina de Turing

$$Q = \{q_1, q_2, q_3\}$$
$$\Sigma = \{a, b\}$$
$$\Gamma = \{a, b, \underline{b}\}$$
$$F = \{q_3\}$$
$$s = q_1$$

y la función de transición δ, dada por

$$\delta(q_1, a) = (q_1, a, L)$$
$$\delta(q_1, b) = (q_1, b, L)$$
$$\delta(q_1, \flat) = (q_2, \flat, R)$$
$$\delta(q_2, a) = (q_3, a, L)$$
$$\delta(q_2, b) = (q_3, b, L)$$
$$\delta(q_2, \flat) = (q_3, \flat, L)$$

Esta máquina de Turing examinará la cinta hacia la izquierda hasta que se encuentre con la primera celda en blanco. Entonces parará y se colocará sobre el blanco. Por tanto, deberíamos tener

$$\begin{aligned}(q_1, aab\underline{a}bb) &\vdash (q_1, aa\underline{b}abb) \vdash (q_1, a\underline{a}babb)\\ &\vdash (q_1, \underline{a}ababb) \vdash (q_1, \underline{\flat}aababb)\\ &\vdash (q_2, \underline{a}ababb) \vdash (q_3, \underline{\flat}aababb)\end{aligned}$$

o

$$\begin{aligned}aabq_1abb &\vdash aaq_1babb \vdash aq_1ababb \vdash q_1aababb\\ &\vdash q_1\flat aababb \vdash q_2aababb \vdash q_3\flat aababb\end{aligned}$$

Obsérvese que, cuando $\delta(q, a)$ está indefinido y la configuración de la máquina de Turing es $(q, w_1\underline{a}w_2)$, es imposible que pase a otra. Entonces se dice que la máquina de Turing está *parada*. Puede que $q \in F$, siendo F el conjunto de estados finales, o puede que no. De cualquier forma, muchas veces nos gustaría dotar de significado a la parada en un estado de F. De hecho, para simplificar, supondremos que no se definirá ninguna transición para cualquier estado de F, por lo que la máquina de Turing se parará siempre que llegue a un estado final. Sin embargo, en cualquier caso, la secuencia de todos los movimientos que conducen a una configuración de parada se llama *computación*.

Consideremos la máquina de Turing dada por

$$Q = \{q_1, q_2\}$$
$$\Sigma = \{a, b\}$$
$$\Gamma = \{a, b, \flat\}$$
$$s = q_1$$
$$F = \emptyset$$

y δ definida por

$$\delta(q_1, a) = (q_2, a, R)$$
$$\delta(q_1, b) = (q_2, b, R)$$
$$\delta(q_1, \flat) = (q_2, \flat, R)$$
$$\delta(q_2, a) = (q_1, a, L)$$

$$\delta(q_2, b) = (q_1, b, L)$$
$$\delta(q_2, \not b) = (q_1, \not b, L)$$

Si en esta máquina de Turing se comienza con la cabeza de lectura/escritura sobre la a de una cadena de la forma abw, se tiene la siguiente secuencia de movimientos:

$$q_1abw \vdash aq_2bw \vdash q_1abw \vdash aq_2bw \vdash \ldots$$

La máquina de Turing se moverá por tiempo indefinido con la cabeza de lectura/escritura desplazándose de derecha a izquierda alternativamente. Éste es un ejemplo de máquina de Turing que nunca parará (se dice, siguiendo la terminología de los programas de computadoras, que la máquina se encuentra en un "bucle infinito"). Esta situación es fundamental en la teoría de las máquinas de Turing y se representará por $(q, w_1\underline{\sigma}w_2) \vdash \infty$ o $w_1q\underline{\sigma}w_2 \overset{*}{\vdash} \infty$. Ello indica que la máquina de Turing, que empezó con la configuración inicial $w_1q\sigma w_2$, nunca se detendrá.

Ejercicios de la Sección 4.1

4.1.1. Construir una máquina de Turing que analice una cadena sobre $\{a, b\}^+$ desplazándose por la cinta de izquierda a derecha y que reemplace todas las *bes* que aparezcan por una c. La máquina de Turing debería comenzar con la cabeza sobre el primer símbolo (el que está más a la izquierda) de la cadena y terminar con la cabeza sobre el blanco final (el blanco que sigue a la a o la c que esté más a la derecha en la cadena transformada).

4.1.2. Construir una máquina de Turing que pare cuando se le presente una cadena de

$$L = \{a^n b^m \mid n, m \geq 0 \text{ y los dos no son 0 a la vez}\}$$

Comenzar el procesamiento con la cabeza situada sobre el primer símbolo (el que está más a la izquierda) de la cadena.

4.1.3. Construir una máquina de Turing que enumere todos los enteros binarios, en orden numérico sobre su cinta cuando comience con $(q_1, \underline{0}\not b)$. Es decir, la maquina de Turing podría ejecutarse de esta forma:

$$(q_1, \underline{0}\not b) \overset{*}{\vdash} (q_1, \underline{1}\not b) \overset{*}{\vdash} (q_1, 1\underline{0}\not b) \overset{*}{\vdash} (q_1, 1\underline{1}\not b) \overset{*}{\vdash} \ldots$$

Obsérvese que esta máquina nunca parará.

4.1.4. Construir una máquina de Turing que enumere sobre su cinta, todos los enteros binarios en orden numérico separados por blancos $\not b$ y que comience con $(q_1, \underline{0}\not b)$.

4.2 MÁQUINAS DE TURING COMO ACEPTADORES DE LENGUAJES

Una máquina de Turing se puede comportar como un aceptador de un lenguaje, de la misma forma que lo hace un autómata finito o un autómata de pila. Colocamos una cadena w en la cinta, situamos la cabeza de lectura/escritura sobre el símbolo del extremo izquierdo de la cadena w y ponemos en marcha la máquina a partir de su estado inicial. Entonces w es aceptada si, después de una secuencia de movimientos, la máquina de Turing llega a un estado final y para. Por tanto, w es aceptada si $qw \vdash^* w_1 p w_2$ para algún estado final p y unas cadenas w_1 y w_2. (Haremos hincapié en que se ha supuesto que la máquina de Turing para cuando llega a un estado final). Entonces, se obtiene la siguiente definición:

Definición 4.2.1. Sea $M = (Q, \Sigma, \Gamma, s = q_1, \flat, F, \delta)$ una máquina de Turing. Entonces el *lenguaje aceptado por M* es

$$L(M) = \{w \in \Sigma^* \mid q_1 w \vdash^* w_1 p w_2 \text{ para } p \in F \text{ y } w_i \in \Gamma^*\}$$

Ejemplo 4.2.1

Consideremos que se diseña una máquina de Turing que acepta el lenguaje regular a^* sobre $\Sigma = \{a, b\}$. Comenzando con el símbolo que está más a la izquierda en una cadena, realizaremos un análisis hacia la derecha, leyendo cada símbolo y comprobando que es una a; si lo es, realizamos un desplazamiento hacia la derecha. Si encontramos un blanco ($\flat$) sin que se haya leído ningún símbolo que no fuera a, paramos y aceptamos la cadena. Si, por otro lado, encontramos un símbolo que no es ni a ni $\flat$, podemos parar en un estado que no es de aceptación.

Sea $Q = \{q_1, q_2\}$, $s = q_1$ y $F = \{q_2\}$, y sea δ definida por

$$\delta(q_1, a) = (q_1, a, R)$$
$$\delta(q_1, \flat) = (q_2, \flat, R)$$

Obsérvese que esta máquina de Turing para en el estado q_2, sólo si se analiza una cadena de 0 ó más *aes*.

Para rechazar una cadena que no es aceptable, lo único que hay que hacer es evitar que se llegue a un estado final. En el Ejemplo 4.2.1, las cadenas que no son aceptables, provocan que la máquina pare en un estado que no es final. Obsérvese que si la máquina de Turing entra en un bucle infinito, no se puede llegar a un estado final (se sigue suponiendo que no hay transiciones definidas para los estados finales). Otra alternativa para rechazar una cadena es entrar en un bucle infinito. El lenguaje del Ejemplo 4.2.1 podría ser aceptado por

$$M = (\{q_1, q_2, q_3\}, q_1, \{a, b\}, \{a, b, \flat\}, \flat, \{q_3\}, \delta)$$

donde δ está definida mediante

$$\begin{aligned} \delta(q_1, a) &= (q_1, a, R), & \delta(q_2, a) &= (q_2, a, R) \\ \delta(q_1, b) &= (q_2, b, R), & \delta(q_2, b) &= (q_2, b, R) \\ \delta(q_1, \flat) &= (q_3, \flat, R), & \delta(q_2, \flat) &= (q_2, \flat, R) \end{aligned}$$

Obsérvese que si se encuentra una b, la máquina de Turing pasa al estado q_2. En el estado q_2, la máquina de Turing siempre se mueve hacia la derecha.

Ejemplo 4.2.2

Consideremos el lenguaje $\{a^n b^n \mid n \geq 1\}$. Para reconocer este lenguaje, no sólo se debe contar el número de *aes* y *bes* sino que también se tiene que verificar que todas las *aes* aparezcan a la izquierda de todas las *bes*. Una forma de abordar este problema es eliminar una por una las *aes* y las *bes* que están en sus posiciones correspondientes. Es decir, empezar con la a que está más a la izquierda y convertirla en algún otro símbolo; entonces, nos desplazamos hacia la derecha hasta que encontramos la primera b. Entonces la convertimos en otro símbolo y nos desplazamos hasta la a que está más a la izquierda. Repetimos este proceso hasta que no queden *aes* y *bes*.

Sea q_1 el estado inicial y supongamos que usamos una c para reemplazar a una a y una d para reemplazar a una b. Las transiciones

$$\begin{aligned} \delta(q_1, a) &= (q_2, c, R), & \delta(q_2, d) &= (q_2, d, R) \\ \delta(q_2, a) &= (q_2, a, R), & \delta(q_2, b) &= (q_3, d, L) \end{aligned}$$

provocan que la máquina de Turing reemplace la a que está más a la izquierda por una c y analice la cadena hacia la derecha, hasta encontrar una b. Esta b se sustituye por una d.

Las transiciones

$$\begin{aligned} \delta(q_3, d) &= (q_3, d, L) \\ \delta(q_3, a) &= (q_3, a, L) \\ \delta(q_3, c) &= (q_1, c, R) \end{aligned}$$

provocan que la máquina de Turing retroceda hacia la izquierda hasta la a que esté más a la izquierda. Puesto que la máquina de Turing sólo puede pasar al estado q_3 mediante las transiciones del primer conjunto, se puede asegurar que una c precede a la a que se encuentra situada más a la izquierda.

Si se acaban todas las *aes*, la máquina de Turing estará en el estado q_1 y situada sobre la celda de la cinta que contiene una d como resultado de la última

transición presente en el conjunto precedente. Se debe realizar una comprobación final para ver si todas las *bes* han sido convertidas en *des*. Las transiciones

$$\delta(q_1, d) = (q_4, d, R)$$
$$\delta(q_4, d) = (q_4, d, R)$$
$$\delta(q_4, b) = (q_5, b, L)$$

llevan a cabo dicha comprobación y, si dicha comprobación tiene éxito (es decir, si no quedan *bes*), deja a la máquina de Turing en el estado q_5 y situada sobre la última *d*. Por tanto, la máquina de Turing que acepta $\{a^n b^n \mid n \geq 1\}$ será

$$M = (\{q_1, q_2, q_3, q_4, q_5\}, \{a, b\}, \{a, b, c, d\}, q_1, b, \{q_5\}, \delta)$$

donde δ está representada por las 10 transiciones precedentes.

El Ejercicio 4.2.6 prueba que el lenguaje $\{a^n b^n c^n \mid n \geq 0\}$ es aceptado por una máquina de Turing. Aunque se sabe que este lenguaje no es independiente del contexto, las máquinas de Turing pueden aceptar algunos lenguajes que un autómata de pila no determinista no puede aceptar.

Un lenguaje que es aceptado por una máquina de Turing se conoce como lenguaje *recursivamente enumerable* (a menudo se abrevia como lenguaje R.E.). El término enumerable proviene de que dichos lenguajes son aquellos cuyas cadenas pueden ser listadas (enumeradas) por una máquina de Turing. Esta clase de lenguajes es bastante grande, incluyendo los lenguajes independientes del contexto.

Recuérdese que, para que una máquina de Turing acepte un lenguaje, no necesita parar sobre cualquier cadena de entrada. Sólo necesita parar en un estado de aceptación para aquellas cadenas que pertenezcan al lenguaje. De hecho hay lenguajes R.E. para los cuales ninguna máquina de Turing que los acepte para con todas las entradas (naturalmente, cualquier máquina de Turing para dichos lenguajes debe parar para toda cadena que pertenezca realmente al lenguaje). La subclase de los lenguajes recursivamente enumerables que son aceptados por al menos una máquina de Turing que para con toda cadena de entrada (dependiendo de si la cadena es aceptada o no), se conoce por la clase de los lenguajes *recursivos*. Más tarde volveremos a hablar sobre los lenguajes recursivos y recursivamente enumerables.

Puesto que las máquinas de Turing pueden leer y escribir sobre su cinta pueden convertir la entrada en salida. Hemos hecho uso de esto en el reconocimiento del lenguaje $\{a^n b^n \mid n \geq 1\}$; cuando transformábamos las cadenas de *aes* y *bes* en cadenas de *ces* y *des*. La transformación de la entrada en salida es el primer propósito de las computadoras digitales; por tanto, una máquina de Turing se considera como un modelo abstracto de una computadora. Se supone que

la entrada para la máquina de Turing está formada por todos los símbolos de la cinta que no son blancos. La salida está formada por cualquiera de los símbolos que queden en la cinta cuando la computación termina.

Por ejemplo, considérese la máquina de Turing $M = (Q, \Sigma, \Gamma, s, \flat, F, \delta)$ donde

$$Q = \{q_1, q_2, q_3\}$$
$$\Sigma = \{a, b\}$$
$$\Gamma = \{a, b, \flat\}$$
$$s = q_1$$
$$F = \{q_3\}$$

y δ dado por

$$\delta(q_1, a) = (q_1, b, R), \qquad \delta(q_2, a) = (q_2, a, L)$$
$$\delta(q_1, b) = (q_1, a, R), \qquad \delta(q_2, b) = (q_2, b, L)$$
$$\delta(q_1, \flat) = (q_2, \flat, L), \qquad \delta(q_2, \flat) = (q_3, \flat, R)$$

Esta máquina de Turing *complementa* las cadenas sobre el alfabeto Σ. Es decir, convierte las *aes* en *bes* y viceversa. Si se comenzó con la configuración $(q_1, \underline{\sigma_1}\sigma_2 \ldots \sigma_n)$, para con la configuración $(q_3, \underline{\alpha_1}\alpha_2 \ldots \alpha_n)$, donde α_i es a si σ_i es b y viceversa.

Obsérvese que las máquinas de Turing pueden ser consideradas como la implementación de una función de cadena f definida mediante $f(w) = u$ cuando se cumple $q_s w \overset{*}{\vdash} q_f u$, donde q_s es el estado inicial y q_f es un estado final. Por conveniencia y claridad, se requiere que la cabeza de lectura/escritura empiece y termine, respectivamente, sobre el símbolo de las cadenas de entrada y salida que está situado más a la izquierda.

Daremos la siguiente definición:

Definición 4.2.2. Se dice que una función de cadena f es *Turing computable* si existe una máquina de Turing $M = (Q, \Sigma, \Gamma, q_1, \flat, F, \delta)$ para la cual $q_1 w \overset{*}{\vdash} q_f u$ para algún $q_f \in F$, cuando $f(w) = u$.

Aunque la computabilidad de Turing se ha definido sólo para funciones de cadena, se puede extender fácilmente esta definición a las funciones integrables, como se muestra en el siguiente ejemplo.

Ejemplo 4.2.3

Supongamos que tenemos $\Sigma = \{a, b\}$ y que representamos los enteros positivos mediante cadenas de *aes*. Así, el entero positivo n estaría representado por a^n. La función suma $f(n, m) = n + m$ podría ser implementada mediante la trans-

formación de $a^n b a^m$ en $a^{n+m}b$. Podríamos obtener una máquina de Turing apropiada para la suma, que estaría representada por $M = (Q, \Sigma, \Gamma, s, b, F, \delta)$, donde

$$Q = \{q_1, q_2, q_3, q_4, q_5\}$$
$$F = \{q_5\}$$

y δ dada por las siguientes transformaciones:

$$\delta(q_1, a) = (q_1, a, R), \qquad \delta(q_3, a) = (q_4, b, L)$$
$$\delta(q_1, b) = (q_2, a, R), \qquad \delta(q_4, a) = (q_4, a, L)$$
$$\delta(q_2, a) = (q_2, a, R), \qquad \delta(q_4, b) = (q_5, b, R)$$
$$\delta(q_2, b) = (q_3, b, L)$$

Esta máquina de Turing simplemente desplaza la b hacia el final, a la derecha de a^{n+m}. Para ello, se crea una a extra. La máquina de Turing "recordará" que se ha creado una a al pasar al estado q_2 una vez que se ha encontrado la b, y entonces se escribirá una b sobre la a que está al final de la cadena. Obsérvese, también, que cuando termina, la máquina de Turing sitúa su cabeza de lectura/escritura sobre la a que se encuentra más a la izquierda.

Ejercicios de la Sección 4.2

4.2.1. Transformar la máquina de Turing del Ejemplo 4.2.1 para que, cuando reciba una cadena que deba aceptar, pare en un estado de aceptación con la cabeza de lectura/escritura sobre el primer blanco que tenga la cadena.

4.2.2. Construir una máquina de Turing que acepte el lenguaje $\{a^{2n} | n \geq 0\}$ sobre $\Sigma = \{a, b\}$.

4.2.3. Mostrar la ejecución de la máquina de Turing del Ejemplo 4.2.2 cuando se parte de cada una de las siguientes configuraciones: $(q_1, \underline{a}abb)$, $(q_1, \underline{a}ab)$ y $(q_1, \underline{a}aabbb)$.

4.2.4. Diseñar una máquina de Turing que acepte el lenguaje $\{a^n b^n | n \geq 0\}$. Obsérvese que, además de tener en cuenta lo visto en el Ejemplo 4.2.2, debemos comprobar la cadena vacía. Transformar la máquina de Turing para que no pare si encuentra una cadena no aceptable.

4.2.5. Diseñar una máquina de Turing que acepte el lenguaje $\{a^n b^n | n \geq 1\}$ por medio de la eliminación de las *aes* y *bes* que están en los *extremos* opuestos de la cadena. Es decir, usando nuevamente *c* y *d*, la cadena *aaabbb* sería primero transformada en *caabbd*, después en *ccabdd* y, por último, en *cccddd*.

4.2.6. Diseñar una máquina de Turing que acepte $\{a^n b^n c^n | n \geq 0\}$.

4.2.7. Construir las máquinas de Turing que acepten los siguientes lenguajes sobre $\Sigma = \{a, b\}$:

(a) aba^*b

(b) $\{w|$ la longitud de w es par$\}$

(c) $\{a^n b^m | n \geq 1 \text{ y } m \neq n\}$

(d) $\{w|w$ contiene el mismo número de *aes* que de *bes*$\}$

(e) $\{a^n b^m a^{n+m} | n \geq 0 \text{ y } m \geq 1\}$

(f) $\{a^n b^n a^n b^m | n \geq 0 \text{ y } m \geq 0\}$

(g) $\{a^n b^{2n} | n \geq 1\}$

(h) $\{ww | w \in \{a, b\}^+\}$

(i) $\{w | w = w^I\}$

(j) $\{a^{n^2} | n \geq 1\}$

(k) $\{a^{2^n} | n \geq 0\}$

4.2.8. Ejecutar la máquina de Turing del Ejemplo 4.2.3 sobre las entradas a^2ba^3 y a^2b. ¿Cómo se comportará esta máquina de Turing cuando se sume $2 + 0$?

4.2.9. Una forma de abordar el método del Ejemplo 4.2.3 que suma $n + m$ es añadir $a^n b$ a a^m cuando la cadena de entrada viene dada por $a^n ba^m$. Entonces la cadena original $a^n b$ se debe eliminar de la cinta. Construir una máquina de Turing que implemente la suma de esta forma.

4.2.10. Para todo número natural, ya sea par o impar, construir una máquina de Turing que calcule la *función de paridad* de los números naturales, es decir, que compute

$$f(n) = \begin{cases} 0, & \text{si } n \text{ es par} \\ 1, & \text{si } n \text{ es impar} \end{cases}$$

4.2.11. En el Ejercicio 4.2.10 hemos transformado cadenas de símbolos en cadenas formadas por un único símbolo. Hay algunos lenguajes que pueden ser reconocidos por medio de esta técnica. Si L es un lenguaje (de un tipo apropiado como se verá más tarde), la *función característica* de L, χ_L, transforma una cadena w sobre el alfabeto en una cadena con un único símbolo 0 ó 1, dependiendo de si w pertenece a L o no. Es decir, χ_L se define como

$$\chi_L(w) = \begin{cases} 1, & \text{si } w \in L \\ 0, & \text{si } w \notin L \end{cases}$$

Construir una máquina de Turing que compute $\chi_L(w)$ para el siguiente lenguaje sobre $\Sigma = \{a, b\}$:

(a) $L = aba^* b$

(b) $L = \{w |$ la longitud de w es par$\}$

(c) $L = \{a^n b^{2n} | n \geq 0\}$

¿Son recursivos estos lenguajes? ¿Son recursivamente enumerables?

4.2.12. La mayoría de las tareas diarias pueden ser interpretadas como transformaciones de cadenas. Por ejemplo, para comprobar la paridad se requiere que las cadenas de n bits se transformen en cadenas de $n + 1$ bits en las cuales, o bien hay un número par de bits 1 (paridad par) o un número impar de bits 1 (paridad impar). Construir una máquina de Turing que transforme cadenas de ocho *ceros* y *unos* en cadenas de nueve *ceros* y *unos*, en las cuales el número de *unos* sea siempre impar. Iniciar la máquina de Turing en la celda que contiene el bit del extremo izquierdo de la cadena. Se supone que el bit de paridad es el noveno bit por la izquierda.

4.2.13. Dada la k-tupla de números naturales $(n_1, n_2, ..., n_k)$, la *proyección* p_i de $(n_1, n_2, ..., n_k)$ es n_i. Es decir, p_i devuelve el i-ésimo componente de la k-tupla. Construir una máquina de Turing P_i que compute la función proyección p_i.

4.3 CONSTRUCCIÓN DE MÁQUINAS DE TURING

Como el lector puede deducir de las observaciones y ejercicios previos, las máquinas de Turing pueden desempeñar muchas actividades además del reconocimiento de lenguajes. De hecho, las máquinas de Turing se toman como modelos teóricos de las computadoras, un tema que trataremos más tarde. Por ahora nos centraremos en la forma de simplificar la construcción de una máquina de Turing. La idea básica es construir una colección de máquinas de Turing sencillas y combinarlas de diferentes formas con el fin de crear una más compleja.

Podemos combinar dos máquinas de Turing permitiendo que compartan la misma cinta y, que cuando una termine su ejecución, la otra empiece. El contenido de la cinta cuando comienza la ejecución de la segunda máquina de Turing, está formado por todo lo que dejó la primera máquina de Turing, y la cabeza de lectura/escritura de la segunda se situará, al comienzo de la ejecución, sobre la celda de la cinta sobre la que terminó la primera.

Ejemplo 4.3.1

Sea M_1 dada por

$$Q_1 = \{q_1, q_2, q_3, q_4\}$$
$$\Sigma = \{a\}$$
$$\Gamma = \{a, b\}$$

$$s_1 = q_1$$
$$F_1 = \{q_4\}$$

con δ_1 de la forma siguiente:

$$\delta_1(q_1, a) = (q_2, a, R)$$
$$\delta_1(q_1, b) = (q_2, b, R)$$
$$\delta_1(q_2, a) = (q_2, a, R)$$
$$\delta_1(q_2, b) = (q_3, b, L)$$
$$\delta_1(q_3, b) = (q_4, b, R)$$
$$\delta_1(q_3, a) = (q_4, a, R)$$

Sea M_2 dada por

$$Q_2 = \{p_1, p_2\}$$
$$\Sigma \text{ y } \Gamma \quad \text{los mismos de } M_1$$
$$s_2 = p_1$$
$$F_2 = \{p_2\}$$

con δ_2 como se escribe a continuación:

$$\delta_2(p_1, a) = (p_2, a, R)$$
$$\delta_2(p_1, b) = (p_2, a, R)$$

Obsérvese que M_1 busca el primer blanco que haya a la derecha de donde ha comenzado, mientras que M_2 escribe una a y para. (La a se escribe independientemente del contenido de la celda actual). Al combinar estas dos máquinas de Turing de forma que una computación de M_1 vaya seguida por una de M_2, obtenemos un dispositivo que primero busca hacia la derecha el primer b y después escribe una a en todas las celdas. Representaremos la combinación de estas dos máquinas de Turing mediante M_1M_2 para indicar que la computación de M_1 va seguida por la computación de M_2.

Vamos a definir formalmente la combinación o composición de máquinas de Turing como sigue:

Definición 4.3.1. Sean M_1 y M_2 dos máquinas de Turing sobre el mismo alfabeto de entrada Σ y el mismo alfabeto de la cinta Γ, donde

$$M_1 = (Q_1, \Sigma, \Gamma, s_1, b, F_1, \delta_1)$$

y

$$M_2 = (Q_2, \Sigma, \Gamma, s_2, b, F_2, \delta_2)$$

Se supone que $Q_1 \cap Q_2 = \emptyset$. La *composición* de las máquinas de Turing M_1 y M_2 es la máquina de Turing $M = (Q, \Sigma, \Gamma, s, \flat, F, \delta)$, que se denota M_1M_2, donde

$$Q = Q_1 \cup Q_2$$
$$s = s_1$$
$$F = F_2$$

$$\delta(q, \sigma) = \begin{cases} \delta_1(q, \sigma), & \text{si } q \in Q_1 \text{ y } \delta_1(q, \sigma) \neq (p, \tau, X) \\ & \text{para todo } p \in F_1 \\ \delta_2(q, \sigma), & \text{si } q \in Q_2 \\ (s_2, \tau, X), & \text{si } q \in Q_1 \text{ y } \delta_1(q, \sigma) = (p, \tau, X) \\ & \text{para algún } p \in F_1 \end{cases}$$

Obsérvese que la composición M_1M_2 se comporta como M_1 hasta que M_1 llega a un estado final. En ese momento cambia al estado inicial de M_2 y se comporta como M_2 hasta que termina.

En nuestro ejemplo previo, M_1M_2 tendría las transiciones dadas mediante

$$\delta(q_1, a) = (q_2, a, R), \qquad \delta(q_3, a) = (p_1, a, R)$$
$$\delta(q_1, \flat) = (q_2, \flat, R), \qquad \delta(q_3, \flat) = (p_1, \flat, R)$$
$$\delta(q_2, a) = (q_2, a, R), \qquad \delta(p_1, a) = (p_2, a, R)$$
$$\delta(q_2, \flat) = (q_3, \flat, L), \qquad \delta(p_1, \flat) = (p_2, \flat, R)$$

con $s = q_1$ y $F = \{p_2\}$.

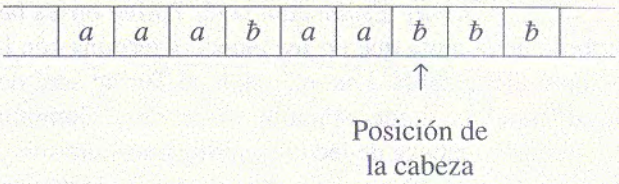
Nos vamos a referir a la máquina de Turing M_1 del Ejemplo 4.3.1 como $R_\flat$. Es decir, $R_\flat$ busca el primer blanco que haya a la derecha de la posición actual de la cabeza. Consideremos una cinta

	a	a	a	$\flat$	a	a	$\flat$	$\flat$	$\flat$

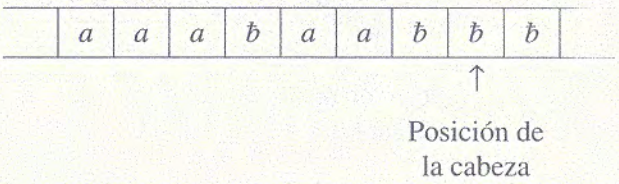
↑

Posición actual de la cabeza

La máquina de Turing compuesta $R_\flat R_\flat$ terminaría en la posición



mientras que la máquina de Turing compuesta $R_bR_bR_b$ terminaría en la posición



Otra forma de especificar las transiciones es el uso de una tabla. La tabla de transiciones correspondiente a R_b sería

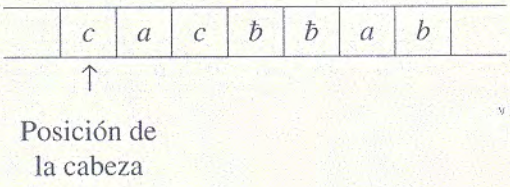
$\delta(q, \sigma)$	$\sigma \neq b$	$\sigma = b$
q_1	(q_2, σ, R)	(q_2, b, R)
q_2	(q_2, σ, R)	(q_3, b, L)
q_3	(q_4, σ, R)	(q_4, b, R)

Obsérvese que una de las columnas se especifica para un símbolo de la cinta en particular y la otra especifica el resto de símbolos de la cinta. La transición $\delta(q_2, \sigma) = (q_2, \sigma, R)$ de la columna etiquetada con $\sigma \neq b$ indica que, para todo símbolo de la cinta σ que sea distinto de b, la máquina de Turing escribe σ en la cinta, se mueve a la derecha y permanece en el estado q_2. Aunque al principio habíamos especificado R_b sólo para el alfabeto de la cinta $\Gamma = \{a, b\}$, ahora hemos especificado R_b para todo alfabeto de cinta que incluya b como el blanco.

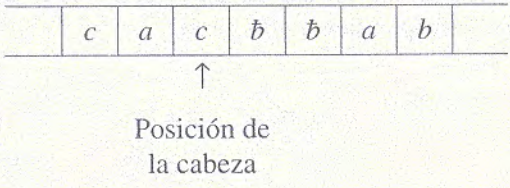
Consideremos la máquina de Turing cuya tabla de transición es

δ	$\sigma = b$	$\sigma \neq b$
q_1	(q_2, b, L)	(q_2, σ, L)
q_2	(q_2, b, L)	(q_3, σ, R)
q_3	(q_4, b, L)	(q_4, σ, L)

donde $F = \{q_4\}$ y $s = q_1$. Esta máquina de Turing busca hacia la izquierda el primer símbolo de la cinta que no sea blanco y termina con la cabeza de lectura/escritura sobre dicha celda. Esta máquina de Turing será denotada por $L_{\overline{b}}$ (la $\overline{b}$ se usa para denotar "cualquier símbolo excepto b"). Combinando $L_{\overline{b}}$ y R_b, se obtiene $R_b L_{\overline{b}}$, donde la cabeza de lectura/escritura se sitúa sobre el símbolo de la cinta que precede al primer b que hay a la derecha de la posición actual. Por tanto, si empezamos sobre



la máquina de Turing combinada terminaría en



En el Ejemplo 4.3.1 vimos una máquina de Turing (M_2) que escribía un único símbolo como salida. Será conveniente especificar una máquina de Turing que escriba como salida un único símbolo, a, y que permanezca sobre dicha celda. Vamos a especificar esta máquina de Turing donde $s = q_1$ y $F = \{q_3\}$ mediante la siguiente tabla:

$\delta(q, \sigma)$	$\sigma \in \Gamma$
q_1	(q_2, a, R)
q_2	(q_3, σ, L)

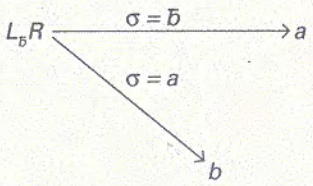
Si denotamos esta máquina de Turing mediante el símbolo a, que escribe como salida, podemos componerla con otras. Por ejemplo, $R_b aR$ buscaría el primer b que hubiera a la derecha de la posición actual de la cabeza, escribiría una a en la celda y se movería a la siguiente celda por la derecha.

Se desearía poder combinar las máquinas de Turing sencillas de modo que se consiguiera una máquina de Turing sencilla bajo unas condiciones determinadas. Por ejemplo, supongamos que $L_{\overline{b}}R$ va seguida por una máquina que escriba una a si la celda es un blanco o escriba una b si la celda contiene una a. Se nece-

sita bifurcar el camino de ejecución. Consideremos la máquina de Turing cuya tabla de transición es

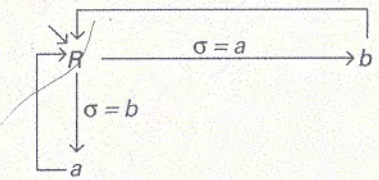
$\delta(q, \sigma)$	$\sigma = \flat$	$\sigma = a$
q_1	$(q_2, \flat, L)$	(q_4, a, L)
q_2	$(q_3, \flat, R)$	(q_3, a, R)
q_4	$(q_5, \flat, R)$	(q_5, a, R)

donde $F = \{q_3, q_5\}$ y $s = q_1$ (obsérvese que no hay transiciones para q_3 o q_5). Supongamos que componemos esta máquina de Turing con las máquinas de Turing que escriben a y b de forma que, si termina en el estado q_3, se inicia la ejecución de la máquina de Turing que escribe a y, si termina en el estado q_5, la ejecución que comienza será la de la máquina de Turing que escribe b. Hemos provocado una bifurcación en la ejecución y se podría denotar mediante el diagrama



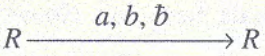
hemos representado la máquina de Turing que causa la bifurcación mediante las flechas.

Por ejemplo, una máquina de Turing que analiza una cadena convirtiendo toda a en b y toda b en a, se podría representar

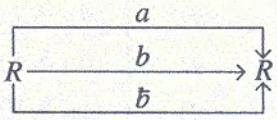


Obsérvese que hemos marcado con una flecha la máquina de Turing sencilla que comienza la secuencia de ejecución, al igual que hicimos con el estado inicial de un autómata finito. Es más, puesto que no hay ningún estado siguiente a R cuando el símbolo actual de la cinta es $\flat$, entonces en ese caso, la máquina compuesta se debería parar.

Las flechas múltiples pueden ser eliminadas de varias formas. Por ejemplo, si $\Gamma = \{a, b, c, \flat\}$, la máquina compuesta denotada por

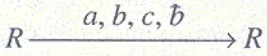


realiza lo mismo que la máquina compuesta

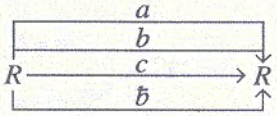


Es decir, se mueve una celda hacia la derecha y entonces, si el símbolo que hay en ese lugar es a, b o $\dot{b}$ se mueve nuevamente hacia la derecha.

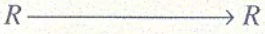
Igualmente, la máquina de Turing denotada por



o por

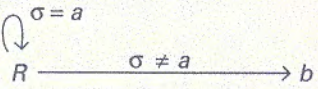


o por

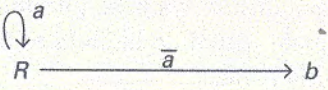


o incluso por RR o R^2, simplemente mueve la cabeza de lectura/escritura dos celdas hacia la derecha (obsérvese que, puesto que Γ está formado solamente por los símbolos a, b, c y $\dot{b}$, todos los símbolos de la cinta producen el segundo R).

Un caso muy común es que se tenga una ramificación para un símbolo específico de la cinta y otra para todos los demás. Esto se puede denotar de varias formas. Por ejemplo, si $\Gamma = \{a, b, c, \dot{b}\}$, entonces

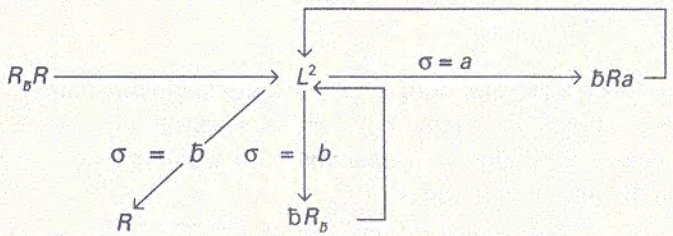


busca hacia la derecha la primera celda que contenga un símbolo que no sea a y escribe una b en ese lugar. El diagrama

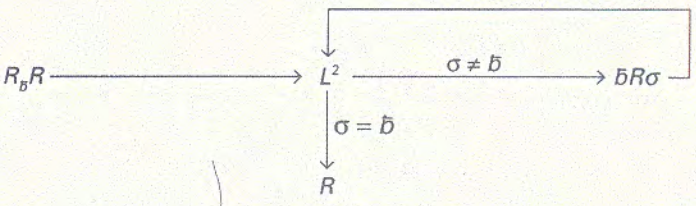


denota lo mismo.

Consideremos el problema del desplazamiento de una cadena sobre la cinta una celda a la derecha. Supongamos que se requiere que la cadena a desplazar sea precedida y seguida por blancos. Por tanto, desearíamos transformar $\underline{\flat}w\flat$ en $\flat\flat\underline{w}$ (el símbolo donde estará situada la cabeza de lectura/escritura se indica mediante el símbolo de subrayado). Si suponemos que el alfabeto de la cinta, Γ, es $\{a, b, \flat\}$, dicha máquina de Turing podría ser construida como se muestra a continuación:

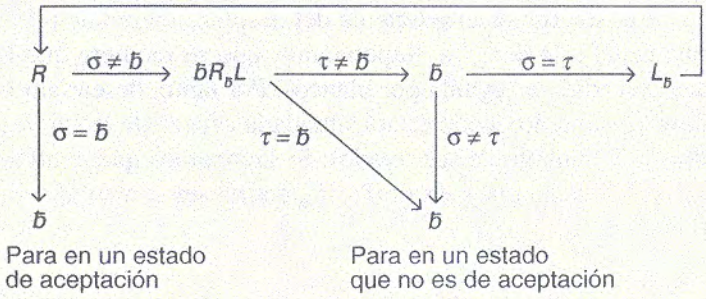


Esta máquina de Turing mueve, uno a uno, todos los símbolos de la cadena hacia la derecha hasta que encuentra el $\flat$ que precedía originalmente a la cadena. Entonces, se mueve hacia la derecha y para. Dos de los caminos pueden ser abreviados como



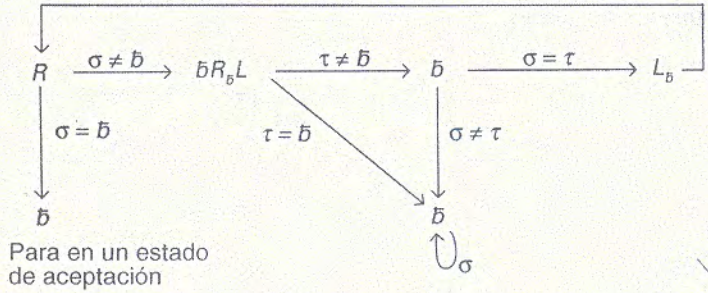
El símbolo σ en $\flat R\sigma$ significa que la máquina compuesta "recuerda" el símbolo que ha sido sobrescrito con el $\flat$. (Originalmente se realizó por medio de dos caminos separados). Esta máquina de desplazamiento hacia la derecha es relativamente sencilla y útil. La denotaremos mediante S_R.

Consideremos el proceso de reconocimiento de $\{ww^I \mid w \in \Sigma\}$. Una forma sería comparar los símbolos que están en los extremos derecho e izquierdo. Si concuerdan, entonces los eliminamos y repetimos el proceso. La cadena será aceptada cuando todos los símbolos hayan sido eliminados. Por otro lado, si dos símbolos fallan al ser comparados, terminamos de forma inmediata en un estado que no es de aceptación. Empezamos con $\flat u\flat$ y esperamos descubrir que $u = ww^I$. Consideremos la máquina de Turing compuesta



Obsérvese que aquí estamos abreviando los caminos posibles al recordar dos símbolos de la cinta σ y τ. Los estados en los que termina esta máquina de Turing son estados de aceptación o no y son denotados por las tres trayectorias con distintas condiciones.

Recordemos que la aceptación de un lenguaje requiere solamente que la máquina de Turing pare en un estado de aceptación si la cadena facilitada está en el lenguaje. Por tanto, una alternativa a la máquina compuesta precedente, podría ser



Esta máquina se detiene en un estado de aceptación para las cadenas de la forma ww^I y, para todas las cadenas que no son de esta forma, no parará, sino que escribirá $\bar{b}$ de forma infinita.

Ejercicios de la Sección 4.3

4.3.1. ¿Cómo se podría realizar la máquina de Turing compuesta M_2M_1 del Ejemplo 4.3.1?

4.3.2. Construir formalmente la máquina de Turing compuesta M_2M_1 del Ejemplo 4.3.1 (dar Q, F, s y δ).

4.3.3. Construir las siguientes máquinas de Turing:

- $L_{\flat}$ Sitúa la cabeza de lectura/escritura sobre el primer $\flat$ que haya a la izquierda de la posición actual de la cabeza.
- $R_{\overline{\flat}}$ Sitúa la cabeza de lectura y escritura sobre el primer símbolo no $\flat$ que haya a la derecha de la posición actual de la cabeza.
- R Mueve la cabeza de lectura/escritura una celda hacia la derecha a partir de su posición actual.
- L Mueve la cabeza de lectura/escritura una celda hacia la izquierda a partir de su posición actual.

4.3.4. ¿Qué efecto produce S_R sobre $\underline{\flat}aa\flat bbb\flat$? ¿Qué efecto produce S_R^2 sobre $\underline{\flat}aa\flat bbb\flat cc\flat$?

4.3.5. Construir una máquina de desplazamiento hacia la izquierda S_L por medio de la composición de máquinas de Turing. Por ejemplo, S_L podría transformar $\underline{\flat}ab\flat$ en $\underline{a}bb\flat$.

4.3.6. Construir una máquina de copia C. C debería transformar $\underline{\flat}w\flat$ en $\underline{\flat}w\flat w\flat$.

4.3.7. Usar la C obtenida en el Ejercicio 4.3.6 (igual que otras máquinas de Turing elementales) para construir una máquina de Turing que compute $f(n, m) = nm$ (es decir, la multiplicación de enteros).

4.3.8. La sustracción de números naturales no está definida para todos los pares puesto que la diferencia puede ser negativa. Definir la operación *resta modificada* como

$$n - m = \begin{cases} n - m, & \text{si } n \geq m \\ 0, & \text{si } n < m \end{cases}$$

Construir una máquina de Turing que compute $n - m$ (la máquina de Turing debería tomar $\underline{\flat}a^n\flat a^m\flat$ y devolver $\underline{\flat}a^{n-m}\flat$).

4.3.9. Construir una máquina de Turing que acepte $\{wcw \mid w \in \{a, b\}^*\}$.

4.3.10. Sea $L(n, m) = \{a^n b^{n-m} \mid n, m \geq 0\}$ (el operador resta modificada, se definió en el Ejercicio 4.3.8). Para unos n y m dados, construir una máquina de Turing que acepte $L(n, m)$.

4.3.11. Construir una máquina de Turing que acepte $\{w \mid w = w^I\}$.

4.3.12. (a) Construir una máquina de Turing que compute la función entera $f(n) = \lfloor n/2 \rfloor$, donde $\lfloor x \rfloor$ representa a la función cuyo resultado es el mayor entero menor o igual que x.

(b) Construir una máquina de Turing que, dados una cadena w y un entero $n \leq |w|$, sitúe la cabeza de lectura/escritura sobre el n-ésimo símbolo de w (a partir de la izquierda).

(c) Construir una variante de S_R, S'_R, que no requiera que la cadena vaya precedida por un símbolo $\flat$. Por ejemplo, S'_R transformaría $\underline{a}bbab\flat$ en $\underline{\flat}babbab\flat$.

(d) Construir una máquina de Turing que acepte $\{ww | w \in \Sigma^*\}$, haciendo uso de las máquinas de Turing obtenidas desde el apartado (a) al (c).

4.4 MODIFICACIONES DE LAS MÁQUINAS DE TURING

Hay otras definiciones de las máquinas de Turing que son equivalentes a la nuestra. Algunos de esos modelos alternativos son mucho más complicados aunque todos tienen la misma potencia computacional (o de cálculo). Muchas de ellas dotan de mayor flexibilidad al diseño de una máquina de Turing que resuelva un problema en particular. Vamos a ver primero unas pequeñas variaciones de nuestra definición original.

Recuérdese que la máquina de Turing sencilla $L_\flat$ sitúa la cabeza de lectura/escritura sobre el primer $\flat$ que haya a la izquierda de la posición actual. Para hacerlo, buscamos fuera de la celda actual y retrocedemos. Esto es debido a nuestra definición que requiere que por cada transición se mueva la cabeza de la cinta.

La función de transición estaba definida como

$$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, L\}$$

y puede ser modificada como

$$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, L, S\}$$

donde S significa "permanecer", es decir *no* mover la cabeza de lectura/escritura. Por tanto $\delta(q, \sigma) = (p, \sigma', S)$ significa que se pasa del estado q al p, se escribe σ' en la celda actual y la cabeza se queda sobre la celda actual. Obsérvese que en esta definición está contenida la nuestra, puesto que ésta es una extensión de la máquina de Turing que hemos definido.

Por otro lado, una máquina de Turing, para la cual esté definida $\delta(q, \sigma) = (p, \sigma', S)$, se puede simular por medio de una máquina que corresponda a nuestra definición original, añadiéndole simplemente los estados y movimientos de la forma $\delta(q, \sigma) = (p', \sigma', R)$ y $\delta(p', \tau) = (p, \tau, L)$ y/o de la forma $\delta(q, \sigma) = (p', \sigma', L)$ y $\delta(p', \tau) = (p, \tau, R)$ para todo $\tau \in \Gamma$.

Por ejemplo, sea M_1 una máquina cuyas transiciones están definidas mediante la tabla

$\delta(q, \sigma)$	$\sigma \neq b$	$\sigma = b$
q_1	(q_2, σ, L)	(q_2, σ, L)
q_2	(q_2, σ, L)	(q_3, σ, S)

donde $s = q_1$ y $F = \{q_3\}$. Obsérvese que esto es la implementación de una máquina de Turing que tiene la capacidad de no moverse. Consideremos M_2 como una máquina de Turing que responde a nuestra definición y, que se deriva de M_1 por medio de esta transformación, cuyas transiciones se presentan en la tabla

$\delta(q, \sigma)$	$\sigma \neq b$	$\sigma = b$
q_1	(q_2, σ, L)	(q_2, σ, L)
q_2	(q_2, σ, L)	(q_4, b, L)
q_4	(q_3, σ, R)	(q_3, b, R)

donde $s = q_1$ y $F = \{q_3\}$.

Otra modificación sencilla de nuestra máquina de Turing básica es aquella mediante la cual cada celda de la cinta se divide en subceldas. Cada subcelda es capaz de contener un símbolo de la cinta. La cinta

	b	b	b	
...	a	a	b	...
	a	a	b	

tiene cada celda subdividida en tres subceldas. Se dice que esta cinta tiene *múltiples pistas*. Puesto que cada celda de esta máquina de Turing contiene múltiples caracteres, el contenido de las celdas de la cinta puede ser representado mediante n-uplas ordenadas. En el ejemplo anterior, las celdas de la cinta contienen (b, a, a), (b, a, a) y (b, b, b). Por tanto, los movimientos que realice esta máquina dependerán de su estado actual y de la n-tupla que represente el contenido de la celda actual.

Supongamos que Γ es un alfabeto de cinta. Una máquina de Turing que tiene una cinta de k pistas, cada una de las cuales contiene un símbolo de Γ, puede interpretarse como una máquina de Turing cuyo alfabeto de cinta estuviera formado por todas las k-tuplas sobre Γ. Por ejemplo, si $\Gamma = \{a, b, b\}$ y M es una máquina de Turing de 2 pistas cuyas celdas contienen pares de símbolos de Γ, se puede considerar que su alfabeto de cinta es $\Gamma \times \Gamma$. Viéndolo de esta forma está

claro que una máquina de Turing multipista no tiene más potencia que nuestra máquina de Turing original. Sin embargo, hace que sea más fácil la construcción de máquinas de Turing que resuelvan ciertos problemas.

Ejemplo 4.4.1

Supongamos que queremos construir una máquina de Turing que sume dos números binarios. Para ello, podemos construir una máquina de Turing de tres pistas. Se supone que la entrada serán dos números binarios que ocupen las dos pista superiores de la cinta. Suponemos que sus dígitos se alinean por la derecha, que sus representaciones binarias son de la misma longitud (lo que se puede conseguir rellenándolas con tantos ceros como sea necesario) y que la cabeza de lectura/escritura se sitúa sobre la celda del extremo izquierdo de la cadena. Por tanto, si tuviésemos que sumar 101 y 10, la cinta debería contener

	b	1	0	1	b	·
···	b	0	1	0	b	···
	b	b	b	b	b	·
			↑			

Cabeza *ell*

La máquina de Turing realizará la suma en la tercera pista. Por tanto, el alfabeto de cinta estará formado por las ternas:

(b, b, b)	$(1, 1, b)$	$(1, 1,0)$	$(1, 1, 1)$
$(0, 0, b)$	$(0, 0, 0)$	$(0, 0, 1)$	$(b, b, 1)$
$(0, 1, b)$	$(0, 1, 0)$	$(0, 1, 1)$	
$(1, 0, b)$	$(1, 0, 0)$	$(1, 0, 1)$	

Esta máquina de Turing primero buscará hacia la derecha el extremo derecho de los números que van a ser sumados. Entonces sumará pares de dígitos, desde la derecha hacia la izquierda, llevando la cuenta de los resultados que se obtengan y sumando a quienes corresponda. Por tanto, se obtiene (suponiendo que q_1 es el estado inicial):

$$\delta(q_1, \sigma) = \begin{cases} (q_1, \sigma, R), & \text{si } \sigma \neq (b, b, b) \\ (q_2, \sigma, L), & \text{si } \sigma = (b, b, b) \end{cases}$$

$\delta(q_2, (0, 0, b)) = (q_2, (0, 0, 0), L)$ $\quad$ $\delta(q_3, (0, 0, b)) = (q_2, (0, 0, 1), L)$
$\delta(q_2, (0, 1, b)) = (q_2, (0, 1, 1), L)$ $\quad$ $\delta(q_3, (0, 1, b)) = (q_3, (0, 1, 0), L)$
$\delta(q_2, (1, 0, b)) = (q_2, (1, 0, 1), L)$ $\quad$ $\delta(q_3, (1, 0, b)) = (q_3, (1, 0, 0), L)$
$\delta(q_2, (1, 1, b)) = (q_3, (1, 1, 0), L)$ $\quad$ $\delta(q_3, (1, 1, b)) = (q_3, (1, 1, 1), L)$
$\delta(q_2, (b, b, b)) = (q_4, (b, b, 0), S)$ $\quad$ $\delta(q_3, (b, b, b)) = (q_4, (b, b, 1), S)$

Obsérvese que se necesita que esta máquina de Turing tenga la posibilidad de no moverse. La máquina de Turing transformará

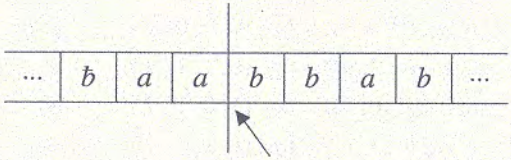
	b	1	0	1	b	
...	b	0	1	0	b	...
	b	b	b	b	b	

en

	b	b	1	0	1	b	
...	b	b	0	1	0	b	...
	b	0	1	1	1	b	

Otra modificación sencilla (y bastante común) que puede realizarse en nuestra definición de máquina de Turing es que se use una cinta que se extienda infinitamente en *una* única dirección. Generalmente, se tiene una cinta que se extiende infinitamente hacia la derecha. No está permitido realizar ningún movimiento hacia la izquierda a partir de la celda del extremo izquierdo. Desde luego, cualquier máquina de Turing de esta forma, puede ser simulada por una de las que responden a nuestra definición original. Para cada computación, simplemente marcaremos una de las celdas de nuestra cinta infinita por los dos lados, como la celda que se encuentra en el límite izquierdo.

Una máquina de Turing con una cinta finita en un sentido puede simular una máquina de Turing con la cinta infinita en los dos sentidos pero con dos pistas. Sea M una máquina de Turing con una cinta infinita en los dos sentidos. La máquina de Turing M', que tiene una cinta infinita en un sentido, puede simular a M si tiene una cinta con dos pistas. La cinta superior contiene la información correspondiente a la parte derecha de la cinta de M, a partir de un punto de referencia dado. La pista inferior contiene la parte izquierda de la cinta de M (en orden inverso). Por tanto, si la cinta de M contenía



la cinta de M' podría ser como

*	b	b	a	b	···
*	a	a	b	b	

Hemos usado un símbolo especial, *, para marcar el límite izquierdo de la cinta. Cuando M tuviera que pasar el punto de referencia, M' tendría que encontrarse con la celda marcada con *. Si M está trabajando sobre las celdas que están a la derecha del punto de referencia, M' está trabajando sobre la pista superior. Cuando M trabaja sobre las celdas que están a la izquierda del punto de referencia, M' trabaja sobre la pista inferior. Cuando M pasa el punto de referencia, M' se encuentra con los *, cambia de dirección y cambia de pista sobre la que trabajar.

Una modificación de nuestra definición, que es más complicada, es la máquina de Turing *multicinta*. Esta máquina de Turing tiene varias cintas, cada una de las cuales tiene su propia cabeza de lectura/escritura. Las cabezas de lectura/escritura se controlan independientemente (es decir, al mismo tiempo, no tienen que moverse en la misma dirección, ni realizar el mismo número de movimientos, ni incluso, hacer nada a la vez). En un sólo movimiento, esta máquina de Turing

1. Cambia de estado dependiendo del estado actual y del contenido de las celdas de *todas* las cintas, que están analizando actualmente las cabezas de lectura/escritura.
2. Escribe un nuevo símbolo en cada una de las celdas barridas por sus cabezas de lectura/escritura.
3. Mueve cada una de sus cabezas hacia la izquierda o hacia la derecha (de forma independiente al resto de las cabezas).

Por tanto, la función de transición para una máquina de Turing con n cintas, es de la forma

$$\delta: Q \times \Gamma^n \rightarrow Q \times \Gamma^n \times \{L, R\}^n$$

donde una transición de la forma

$$\delta\,(q, (\sigma_1, \sigma_2, ..., \sigma_n)) = (p, (\tau_1, \tau_2, ..., \tau_n), (X_1, X_2, ..., X_n))$$

significa que cambia del estado q a p, reemplaza σ_i por τ_i en la cinta i y mueve la cabeza de la cinta i en la dirección X_i.

Ejemplo 4.4.2

Las máquinas de Turing multicinta simplifican en gran medida alguna de las actividades que nos gustaría hacer con máquinas de Turing. Consideremos el proceso de reconocimiento de $\{a^n b^n \mid n \geq 1\}$. Éste es bastante laborioso en una máquina de Turing con una única cinta. Es mucho más fácil realizarlo con una máquina de Turing con dos cintas. Supongamos que, inicialmente, colocamos la cadena a analizar en la cinta 1 y que q_1 es el estado inicial. Si la cabeza de lectura/escritura de la cinta 1 está situada inicialmente sobre el carácter del extremo izquierdo de la cadena, las cuatro transiciones siguientes son fundamentales para el reconocimiento (cualquier otra transición sería para cadenas mal formadas y se puede suponer que llega a un estado que no es de aceptación):

$$\delta\,(q_1, (a, b)) = (q_1, (a, a), (R, R))$$
$$\delta\,(q_1, (b, b)) = (q_2, (b, b), (S, L))$$
$$\delta\,(q_2, (b, a)) = (q_2, (b, a), (R, L))$$
$$\delta\,(q_2, (b, b)) = (q_3, (b, b), (R, L))$$

Aunque una máquina de Turing multicinta parece bastante distinta y posiblemente más potente que nuestra máquina de Turing definida originalmente, las dos son equivalentes en el sentido de que cada una de ellas puede ser simulada por la otra.

Sea M_1 una máquina de Turing con k cintas. M_1 puede ser simulada mediante una máquina de Turing, M_2, que tenga una única cinta dividida en $2k + 1$ pistas. Cada una de las cintas de M_1 corresponde a dos pistas de M_2. Una pista almacena el contenido de la cinta de M_1 correspondiente, mientras que la otra se usa para guardar un marcador que afecta a la posición de la cabeza correspondiente a esta cinta de M_1. La última de todas las pistas de la cinta de M_2 se usa para guardar los marcadores de final. Los marcadores de final indican las posiciones de los marcadores de la cabeza que están más a la izquierda y más a la derecha. Cada movimiento de M_1 es simulado por M_2, realizando un barrido con su cabeza de lectura/escritura, primero de izquierda a derecha y después de derecha a izquierda. La cabeza de M_2 comienza en la celda marcada como extremo izquierdo (por el marcador de final izquierdo). Para simular un movimiento de la máquina M_1 de k cintas, M_2 hace un barrido hacia la derecha almacenando (por medio de estados), para cada una de las cabezas, los símbolos marcados como principio del análisis. Si la cabeza de M_2 encuentra el marcador de final derecho,

quiere decir que la máquina de Turing ha recorrido todos los símbolos que todas las cabezas de M_1 analizan y, por tanto, ya tiene bastante información como para determinar un movimiento. Después realiza un barrido hacia la izquierda, actualizando el contenido de la cinta y ajustando los marcadores de la cabeza como corresponda, hasta que encuentre el marcador de final izquierdo. Obsérvese que cuando M_2 realiza este barrido necesita tener la posibilidad de mover los marcadores a la izquierda o la derecha de sus posiciones actuales. De esto se deduce que el patrón de movimiento que realiza la cabeza de lectura/escritura durante el barrido consiste en dos desplazamientos a la izquierda seguidos de dos desplazamientos a la derecha y, por último, un desplazamiento hacia la izquierda. Finalmente, M_2 pasa al estado correspondiente al estado siguiente de M_1.

Otra modificación que se puede hacer en nuestra máquina de Turing original es permitir que la cinta tenga muchas dimensiones. Por ejemplo, una cinta de dos dimensiones que se extienda hacia abajo y hacia arriba, al igual que hacia la derecha y hacia la izquierda. Dependiendo del estado actual de la máquina de Turing y del símbolo analizado, cambia de estado, escribe un símbolo en la celda actual y se mueve a la izquierda, a la derecha, hacia arriba o hacia abajo. Por tanto, la función de transición para esta máquina de Turing será de la forma

$$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, U, D\}$$

Una máquina de Turing multidimensional simula, fácilmente, una máquina de Turing estándar. Simplemente realiza todas sus computaciones en una única dimensión. Una máquina de Turing estándar también puede simular una máquina de Turing multidimensional y, por tanto, la complejidad y flexibilidad adicional que se debe a la múltiple dimensión, no es una capacidad real. Para simular una máquina de Turing de dos dimensiones mediante una máquina de Turing estándar, primero asociaremos una dirección a todas las celdas de la cinta. Una forma de hacerlo es fijar, de forma arbitraria, un lugar en la cinta a partir del cual se asignarán las coordenadas a las celdas de la misma forma que se realiza en el plano de coordenadas.

Por ejemplo,

	–1, 1		1, 1		
	–1, 0	0, 0	1, 0	2, 0	
		0, –1			

Entonces, usaremos una cinta de dos pistas para simular la máquina de Turing. Una pista se encargará de almacenar el contenido de las celdas y la otra las coordenadas. Según esto, si la celda (1, 2) contiene una a y la celda (−3, 12) contiene una b, la cinta de la máquina de Turing que realiza la simulación será como

	a				b					
	1	*	2	*	−	3	*	1	2	

Obsérvese que se ha introducido un nuevo símbolo para separar los valores de las coordenadas. Para simular un movimiento de una máquina de Turing de dos dimensiones, esta máquina calcula la dirección de la celda a la que se movería la máquina de Turing de dos dimensiones. Entonces, localiza en la pista inferior la celda con dicha dirección y cambia el contenido de la celda en la pista superior.

Finalmente, una modificación importante que se puede introducir en nuestra definición original es la eliminación del requerimiento de que la regla de transición sea una función. Esta máquina de Turing se llamará máquina de Turing *no determinista*, ya que para un estado actual y el símbolo actual de la cinta, puede haber un número finito de movimientos a elegir. Por tanto, la regla de transición, Δ, de dicha máquina, satisface

$$\Delta(q, \sigma) \subseteq Q \times \Gamma \times \{L, R\}$$

Por ejemplo, si la máquina de Turing tiene una transición

$$\Delta(q_1, a) = \{(q_1, b, R), (q_2, a, L)\}$$

entonces los movimientos

$$(q_1, abb\underline{a}b) \vdash (q_1, abbb\underline{b}) \quad \text{y} \quad (q_1, abb\underline{a}b) \vdash (q_2, ab\underline{b}ab)$$

son posibles.

Ya que cualquier máquina de Turing determinista es también no determinista, es lógico que una máquina de Turing determinista se puede simular mediante una no determinista. También una máquina de Turing determinista puede simular una no determinista. Por tanto, no se gana ninguna potencia adicional a causa del no determinismo.

Para ver como se realizaría la simulación, sea M_1 una máquina de Turing no determinista que acepta algún lenguaje. Describiremos una máquina de Turing (determinista) M_2, que será de tres cintas, que simule M_1. Téngase en cuenta que, para cualesquiera estado actual y símbolo de la cinta de M_1, hay un número

finito de movimientos para realizar a continuación. Numeraremos dichos movimientos $1, 2, ..., k$. Puesto que tanto Q_1 como Γ_1 de M_1 son conjuntos finitos, sólo existirá un número finito de pares de estado actual y símbolo de la cinta. Por tanto, se puede obtener el par que tenga el mayor número de posibles movimientos a realizar a continuación. Sea este número n. Entonces cualquier computación para M_1 se puede representar mediante una secuencia finita de números elegidos entre 1 y n, donde cada número representará el movimiento elegido para ser realizado a continuación en esta etapa de la computación. No toda secuencia finita de números entre 1 y n representan computaciones válidas puesto que puede que haya menos de n elecciones para algún par estado-símbolo de la cinta.

En la máquina de Turing de tres cintas, M_2, la primera cinta almacenará la entrada y la segunda generará, de forma sistemática, secuencias finitas de números entre 1 y n. Para cada secuencia, M_2 copiará la cadena de entrada en la tercera cinta y simulará M_1 sobre esta cinta usando la secuencia almacenada en la segunda cinta como guía para realizar la computación. Si una de las secuencias elegidas hace que M_1 acepte la cadena, entonces dicha secuencia será generada por M_2 y la cadena será aceptada. Si ninguna de las secuencias elegidas provoca la aceptación por parte de M_1, entonces tampoco M_2 la aceptará.

Obsérvese que este razonamiento se redacta en términos de la aceptación de lenguajes. Las máquinas de Turing no deterministas se interpretan como aceptadores de lenguaje. Además se puede generalizar este razonamiento fácilmente, para simular una máquina de Turing multicinta no determinista.

En esta sección hemos descrito varias modificaciones a realizar sobre nuestra máquina de Turing original, ninguna de las cuales tiene mayor o menor potencia que la original. Entonces, cabe preguntarse por qué nos hemos molestado en estudiarlas si ninguna de ellas aporta ninguna ganancia en cuanto a capacidad computacional, o por qué no nos hemos centrado en la más sencilla de todas ellas. Una de las razones es que, a partir de ahora, conocemos formas distintas de resolver problemas mediante las máquinas de Turing. Si una de ellas resolviera un problema, de forma más fácil que la original entonces, sin temor a perder generalidad, usaremos dicha variante.

Otra razón aún más profunda para el estudio de dichas modificaciones y sus equivalencias con el original, es que con esto se comprende aún más la actualidad de las máquinas de Turing. Cualquier computación que se pueda realizar por medio de una de las nuevas máquinas cae dentro de la categoría de "computable por una máquina de Turing" y, por tanto, es mecánicamente computable.

Esto apenas ha sido un recorrido exhaustivo por las modificaciones que se pueden realizar a las máquinas de Turing. Más adelante nos extenderemos sobre otras variaciones de la definición original tratada en este capítulo.

Ejercicios de la Sección 4.4

4.4.1. Eliminar del Ejemplo 4.4.1 la restricción de que el más corto de los dos números binarios deba ser rellenado con ceros.

4.4.2. Construir una máquina de Turing de tres pistas que reste el número binario de la segunda pista del número binario de la primera y deje el resultado en la tercera pista.

Hacer otra, suponiendo que la máquina de Turing es de dos pistas y que el resultado se deja sobre la segunda.

Hacerlo, también, para que el resultado quede en la primera.

4.4.3. Construir una máquina de Turing de tres pistas que determine si el número binario que está en la primera pista es menor que el de la segunda. Si es menor, escribir el carácter S sobre la tercera pista y si no lo es, escribir los caracteres GE sobre la tercera pista. ¿Cómo se podría incluir una forma de comprobar la igualdad?

4.4.4. Una máquina de Turing puede comprobar si un número binario mayor que 2, es primo. Para ello se escribirá dicho número en la pista 1. La máquina de Turing escribirá el número 2 en binario en la segunda pista y copiará la primera pista en la tercera. Entonces restará la segunda pista de la tercera tantas veces como sea posible (dejando el resultado en la tercera). En realidad, este proceso lo que hace es dividir el número de la pista uno entre el de la pista dos y deja el resto en la tercera. Si el resto es 0, entonces el número no es primo. Si no es 0, entonces el número de la pista dos se incrementa. Si el número de la primera pista es igual al de la segunda, entonces el número es primo ¿por qué? Si el número de la pista 2, es menor que el de la 1, se repite el proceso anterior para el nuevo número de la pista 2. Construir una máquina de Turing que realice este proceso de comprobación.

4.4.5. Las múltiples pistas sirven para realizar el reconocimiento de las frases de un lenguaje de forma no destructiva. La cadena en cuestión se sitúa sobre la primera pista y se usa una pista adicional para realizar el análisis de los símbolos (como contrapartida a la eliminación o sobreescritura de los mismos). Esto se realiza situando marcadores en la segunda pista sobre o bajo los símbolos de la primera pista que han sido ya considerados. Construir una máquina de Turing que reconozca el lenguaje $\{wcw \mid w \in \{a, b\}^*\}$.

4.4.6. Sea M una máquina de Turing con cinta infinita en los dos sentidos cuyas transiciones son

$\delta(q, \sigma)$	$\sigma = a$	$\sigma = b$	$\sigma = b$
q_1	(q_1, c, R)	(q_1, b, R)	(q_2, b, L)
q_2	(q_2, a, L)	(q_2, d, L)	(q_3, b, S)

Supongamos que q_1 es el estado inicial de M. Construir una máquina de Turing con cinta infinita en un sentido que simule las acciones de M para una cadena de entrada situada en cualquier posición de la cinta de M.

4.4.7. Diseñar una máquina de Turing con cinta infinita en un sentido que, cuando comience con la configuración (q_1, wb), aceptará

$$L = \{w \in \{a, b\}^* | w \text{ contiene al menos una } a\}$$

(*Nota:* Téngase en cuenta que, para que una cadena sea aceptada, la máquina de Turing tiene que parar en un estado de aceptación).

4.4.8. Describir la ejecución de la máquina de Turing de dos cintas del Ejemplo 4.4.2 sobre la cadena a^2b^2.

4.4.9. Construir una máquina de Turing de dos cintas que reconozca el lenguaje $L = \{ww^I | w \in \{a, b\}^+\}$. Supóngase que, inicialmente, las cadenas a analizar están situadas en la cinta 1.

4.4.10. Supóngase que M es una máquina de Turing de dos cintas con un alfabeto de cinta $\Gamma = \{a, b\}$. Definir cada una de las partes de una máquina de Turing de cinco pistas M' que simule el movimiento $\delta(q_1, (a, b)) = (q_2, (b, a), (R, L))$ de M.

4.4.11. Supóngase que se define una máquina de Turing *multicabeza*, como una que tiene una única cinta pero varias cabezas de lectura/escritura sobre la cinta. Cada movimiento depende tanto del estado actual como de todos los símbolos reconocidos por las cabezas. En cada movimiento, si más de una cabeza está sobre la misma celda, se decide según lo convenido previamente.

(a) Construir una "maquina de copia" con dos cabezas que, en la cinta, transforme bwb en $bwbwb$.

(b) Obtener, de forma similar a como se simuló una máquina de Turing con k cintas mediante una máquina de Turing con una cinta, una simulación de una máquina de Turing multicabeza mediante una máquina de Turing con una sola cinta de dos sentidos.

4.4.12. ¿Cómo se puede simular una máquina de Turing que responda a nuestra definición, mediante una máquina de Turing multicinta? ¿Cómo puede una máquina de Turing multicabeza, simular una máquina de Turing que cumpla la definición original?

4.4.13. Construir una máquina de Turing con una cinta de dos dimensiones que "dibuje" una tabla 2×3 y que empiece por la celda de la esquina inferior izquierda.

4.4.14. Construir una máquina de Turing, según nuestra descripción original, que genere todas las secuencias de los números 1, 2 y 3 sobre su cinta, de forma que las secuencias menores aparezcan primero y las que tengan la misma longitud aparezcan en orden lexicográfico. ¿Cómo se podría generalizar para las secuencias de los números 1, 2, ..., n?

4.4.15. Construir una máquina de Turing no determinista que acepte el lenguaje $L = \{ww | w \in \{a, b\}^+\}$.

4.4.16. Construir una máquina de Turing no determinista que acepte

$$L = \{xww^I y | x, y, w \in \{a, b\}^+ \text{ y } |x| \geq |y|\}$$

¿Cómo se podría resolver este problema de forma determinista?

4.4.17. Explicar con cuidado por qué la sentencia "Si ninguna de las secuencias elegidas nos lleva a una aceptación por parte de M_1, entonces M_2 tampoco aceptará" es verdad con respecto a la descripción de la simulación de una máquina de Turing no determinista mediante una máquina de Turing (determinista).

4.5 MÁQUINAS DE TURING UNIVERSALES

La máquina de Turing *universal* es una máquina de Turing que, a partir de una descripción adecuada de una máquina de Turing M y una cadena de entrada w, simula el comportamiento de M sobre la cadena w. En esta sección describiremos dicha máquina de Turing. Necesitaremos este desarrollo posteriormente, cuando estudiemos la jerarquía de lenguajes y la resolubilidad, pero también nos servirá para fortalecer la conexión casi intuitiva, entre las máquinas de Turing y las computadoras digitales modernas.

Lo primero que debemos hacer es encontrar la manera de describir una máquina de Turing arbitraria $M = (Q, \Sigma, \Gamma, s, b, F, \delta)$ ya que su descripción sirve de introducción a la máquina de Turing universal. Esto requiere que la descripción sea codificada a partir de un alfabeto finito. Estableciendo unas sencillas convenciones, una codificación de ese tipo es algo realmente sencillo.

Para empezar, se requiere que M tenga un único estado de aceptación. La transformación de una máquina de Turing arbitraria en una que tenga un único estado final es un proceso bastante sencillo. Simplemente se añaden las transiciones y los estados necesarios para que la máquina de Turing pueda pasar de cada estado de aceptación q al nuevo estado de aceptación p que será único, y la cabeza de lectura/escritura se dejará en la posición en la que estaba cuando se había llegado a q.

Entonces se supone que $Q = \{q_1, q_2, ..., q_n\}$, donde q_1 es el estado inicial y q_2 es el único estado final de M. Es más, suponemos $\Gamma = \{\sigma_1, \sigma_2, ..., \sigma_m\}$, donde σ_1 es el blanco. Partiendo de estas suposiciones, M estará completamente determinado por medio de su función de transición. Luego, para codificar M, sólo hay que codificar δ. Para ello, representamos q_1 por 1, q_2 por 11 y así sucesivamente. De forma similar, representamos σ_i por una cadena de i *unos*. Finalmente, representamos las directivas para la cabeza de lectura/escritura de forma que L será 1

y R será 11. Si usamos los ceros como separadores, podemos codificar una transición tal como

$$\delta(q_3, \sigma_1) = (q_4, \sigma_3, L)$$

mediante la cadena 0111010111110111010. De ello se sigue que M tiene una codificación representada por una cadena finita de *ceros* y *unos*. Es más, dada una codificación, podemos decodificarla correctamente.

Una máquina de Turing universal M_u se puede implementar como una máquina de Turing de tres cintas cuyo alfabeto de entrada contenga *ceros* y *unos*. La primera cinta contiene la codificación de M con su cabeza situada sobre el *cero* inicial de la cadena de *ceros* y *unos*. La segunda cinta contiene la codificación del contenido de la cinta de M con su cabeza situada sobre el *uno* que pertenece a la codificación del símbolo actual. La tercera cinta se usa para guardar el estado actual de M. Inicialmente, esta última cinta contiene la versión codificada del estado inicial de M (estado q_1, al que le corresponde un *uno*) rodeado por blancos. La cabeza de lectura/escritura se sitúa sobre el primer *uno* de la cadena codificada.

M_u analiza y compara el contenido de las cintas segunda y tercera con el de la primera cinta, hasta que encuentra una transición para la configuración codificada o hasta que agota todas las posibilidades. Si no se encuentra una transición correspondiente a la configuración, M_u para, como debería hacerlo M. En otro caso, M_u se comporta como lo haría M.

Si M con la cadena w, para, entonces M_u parará cuando tenga en sus cintas la codificación de M y la codificación de w. Es más, la cadena final que quede en la cinta de M_u será la codificación de la cadena que hubiera quedado en M. Cuando M para, M_u puede decir si está en el único estado de aceptación y, según esto, moverse a uno de sus estados de aceptación o no.

Ejercicios de la Sección 4.5

4.5.1. Sea M dado por

$$Q = \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\}$$
$$\Sigma = \{a, b\},$$
$$\Gamma = \{a, b, b\},$$
$$s = q_1$$
$$F = \{q_6, q_7\}$$

con δ definida como sigue:

$$\delta(q_1, b) = (q_1, b, R) \qquad \delta(q_3, b) = (q_4, b, L)$$
$$\delta(q_1, a) = (q_2, b, R) \qquad \delta(q_3, a) = \delta(q_2, b) = (q_8, b, L)$$
$$\delta(q_1, b) = (q_6, b, R) \qquad \delta(q_4, a) = (q_4, a, L)$$
$$\delta(q_2, a) = (q_2, a, R) \qquad \delta(q_4, b) = (q_4, b, L)$$
$$\delta(q_2, b) = (q_2, b, R) \qquad \delta(q_4, b) = (q_5, b, R)$$
$$\delta(q_2, b) = (q_3, b, L) \qquad \delta(q_5, a) = (q_1, a, L)$$
$$\delta(q_5, b) = (q_7, b, R) \qquad \delta(q_5, b) = (q_6, b, R)$$
$$\delta(q_8, a) = \delta(q_8, b) = \delta(q_8, b) = (q_8, b, L)$$

Convertir M en una máquina de Turing con un único estado.

4.5.2. Obtener una codificación completa de la máquina de Turing dada por

$$\delta(q_1, \sigma_1) = (q_1, \sigma_1, R)$$
$$\delta(q_1, \sigma_2) = (q_3, \sigma_1, L)$$
$$\delta(q_3, \sigma_1) = (q_2, \sigma_2, L)$$

4.5.3. Obtener un algoritmo que determine si una cadena $w \in \{0, 1\}^+$ es una máquina de Turing codificada.

4.5.4. Hacer un bosquejo de un procedimiento que enumere las codificaciones de todas las máquinas de Turing. *Indicación:* En el Ejercicio 4.1.4. se podía enumerar, en orden, todos los números enteros binarios. Combinar esto con el Ejercicio 4.5.3.

El Ejercicio 4.5.4. tiene una consecuencia importante que será la siguiente:

4.5.5. Demostrar que el conjunto de las máquinas de Turing es numerable.

PROBLEMAS

En este capítulo hemos visto que nos es posible aumentar la capacidad de computación de una máquina de Turing estándar, por mucho que se complique la estructura de la cinta. Se puede probar que es posible limitar su potencia si se restringe la manera en la que se usa la cinta. Una restricción posible es limitar el número de celdas de la cinta que pueden ser usadas por la máquina de Turing, basándose en la longitud de la cadena de entrada. Por tanto, podría haber más espacio disponible para las computaciones de cadenas largas que para las cortas.

Definiremos una *autómata linealmente acotado* (ALA) como una máquina de Turing no determinista $M = (Q, \Sigma, \Gamma, s, b, F, \Delta)$, en la cual el alfabeto de cinta contenga dos símbolos especiales < y >. M comienza con la configuración $(q_1, \leq w >)$ (donde q_1 es el estado inicial de M). No se permite que M reemplace los símbolos < o >, ni que mueva su cabeza a la izquierda de < o a la derecha de >. Obsérvese que un autómata linealmente acotado tiene que realizar su computación en las únicas celdas de la cinta que estaban originalmente ocupadas por la cadena de entrada.

Por ejemplo, consideremos el ALA definido por

$$Q = \{q_1, q_2\}$$
$$\Sigma = \{a, b\}$$
$$\Gamma = \{a, b, <, >\}$$
$$s = q_1$$
$$F = \{q_2\}$$

y Δ dado por

$$\Delta(q_1, <) = (q_1, <, R)$$
$$\Delta(q_1, a) = (q_1, b, R)$$
$$\Delta(q_1, b) = (q_1, a, R)$$
$$\Delta(q_1, >) = (q_2, >, S)$$

Este ALA complementa sus cadenas de entrada convirtiendo las *aes* en *bes* y viceversa. Obsérvese que, aunque puede reconocer y trabajar sobre los símbolos < y >, no puede reemplazarlos o moverse más allá de ellos.

Supongamos que un ALA comienza siempre con su cabeza situada sobre el símbolo <.

4.1. Obtener un ALA que acepte el lenguaje $\{a^n b^n c^n \mid n \geq 1\}$.

4.2. Obtener un ALA que determine si sus cadenas de entrada tienen longitud impar. Suponer que las cadenas de entrada pertenecen a $\{a, b\}^*$.

4.3. Obtener un ALA que determine si sus cadenas de entrada, tomadas de $\{a, b\}^*$ son un palíndromo.

Un autómata linealmente acotado puede tener cintas con múltiples pistas. En el caso de las pistas múltiples, las celdas disponibles para la computación siguen restringidas a las que originalmente ocupó la cadena de entrada. Por ejemplo, si $w = abaa$ es la cadena de entrada para un ALA con dos pistas, la cinta podría ser como

		<	*a*	*b*	*a*	*a*	>		
		<					>		

(donde se supone que las celdas en blanco contienen *b*).

4.4. Construir un ALA con dos pistas que acepte el lenguaje

$$\{a^{n!} \mid n \geq 0\}$$

donde $n!$ es la función factorial. (*Indicación:* Dividir la cadena repetidamente entre 2, 3, 4, ... situando el divisor en la segunda pista. ¿Cuándo y cómo se podrá determinar si hay $n!$ *aes* en la primera pista?)

4.5. Sea $k > 0$ un entero fijado. Construir un ALA que, cuando se le den dos enteros n y m (como a^n y a^m), determine si $n - m$ es divisible por k.

5

Máquinas de Turing y lenguajes

5.1 LENGUAJES ACEPTADOS POR MÁQUINAS DE TURING

En el Capítulo 4 hemos introducido las máquinas de Turing y estudiado algunas de sus propiedades. En particular, el desarrollo de una máquina de Turing universal sugiere que una máquina de Turing puede ser concebida de forma análoga a un programa de una computadora y, por tanto, las máquinas de Turing podrían modelar los mecanismos de computación. Estableceremos esta conexión a lo largo del Capítulo 6, donde estudiaremos los límites de un mecanismo de computación mediante el estudio de la resolubilidad. En este capítulo, estudiaremos las clases de lenguajes que son aceptados por máquinas de Turing. Al realizarlo, llegaremos a un problema general en todas las máquinas de Turing. Este problema se llama *problema de parada*. El problema de parada será la clave para tratar la resolubilidad. Terminaremos con un teorema (el teorema de la jerarquía) que relacionará todos los tipos de lenguajes estudiados.

Recordaremos que una cadena w sobre algún alfabeto es aceptada por una máquina de Turing M, cuando una computación que empezó con w en la cinta de M y con M en su estado inicial, termina con M en un estado final. Por otro lado, w puede ser rechazada de dos formas, ya sea porque M para en algún estado que no es de aceptación o porque M nunca para. Los dos métodos de rechazo de cadenas no son equivalentes. Por tanto, hemos definido los lenguajes *recursivamente enumerables* como los lenguajes que son aceptados por una máquina de Turing (de alguna forma) y los lenguajes *recursivos* como los lenguajes acepta-

dos por alguna máquina de Turing que siempre para sobre alguna entrada. Formalmente, daremos la siguiente definición:

Definición 5.1.1. Un lenguaje L sobre un alfabeto Σ se dice que es *recursivamente enumerable* si es aceptado por alguna máquina de Turing. Es decir, L es recursivamente enumerable si para alguna máquina de Turing M tenemos que

$$L = \{w \in \Sigma^* \mid qw \overset{*}{\vdash} upv \text{ para } p \in F \text{ y } u, v \in \Gamma^*\}$$

(donde q es el estado inicial de M y F es el conjunto de estados finales de M).

Un lenguaje L es *recursivo* si L es recursivamente enumerable y hay alguna máquina de Turing que para sobre *todas* las entradas que acepta L.

De esta definición se deduce que todo lenguaje que es recursivo, también es recursivamente enumerable. En la Sección 5.4 veremos que hay lenguajes que son recursivamente enumerables, que no son recursivos.

Ejercicios de la Sección 5.1

5.1.1. Probar que L es recursivo si y sólo si la función característica de L, χ_L, es una función Turing computable (véase Ejercicio 4.2.11). (Por esta razón, si L es un lenguaje recursivo y T es una máquina de Turing que acepta L y para sobre todas las entradas, entonces se dice que T *decide* L, puesto que para toda cadena w, T decide si w está en L o no.)

5.1.2. Supongamos que M es una máquina de Turing que para sobre todas las entradas y que acepta el lenguaje L. Transformar M en M', donde $L = L(M')$ y M' rechaza las cadenas porque no para.

5.2 LENGUAJES REGULARES, INDEPENDIENTES DEL CONTEXTO, RECURSIVOS Y RECURSIVAMENTE ENUMERABLES

Sea $M = (Q, \Sigma, s, F, \delta)$ un autómata finito determinista. Se puede construir una máquina de Turing $M' = (Q', \Sigma', \Gamma, s', b, F', \delta')$ para la cual $L(M) = L(M')$ por medio de:

$$\begin{aligned} Q' &= Q \cup \{q'\}, \quad \text{donde } q' \text{ es un nuevo estado que no está en } Q \\ \Sigma' &= \Sigma \\ \Gamma &= \Sigma \cup \{b\} \\ F' &= \{q'\} \end{aligned}$$

$\delta'(q, \sigma) = (\delta(q, \sigma), \sigma, R)$, para $q \in Q$ y $\sigma \in \Sigma'$

$\delta'(q, \flat) = (q', \flat, S)$, para todo $q \in F$, donde S es la directiva de cinta "no moverse"

La máquina de Turing resultante analiza sus cadenas de entrada de izquierda a derecha. Con cada símbolo de entrada cambia su estado para reflejar el correspondiente cambio de estado en el autómata finito. Cuando se encuentra un $\flat$ en el extremo derecho de la cadena de entrada, pasa a un estado de aceptación sólo si el autómata finito aceptara la cadena.

Obsérvese que, ya que nunca puede haber en M pares de la forma $(q, \flat)$, definiendo δ' igual a δ en lo que sea posible y $\delta'(q, \flat) = (q', \flat, S)$ para el estado $q \in F$, δ' se convierte en una función de transición bien definida.

Puesto que todo lenguaje regular puede ser aceptado por un autómata finito determinista, tendremos el siguiente teorema:

Teorema 5.2.1. Si L es un lenguaje regular, entonces L es también un lenguaje recursivo.

Por el Ejemplo 4.2.2 sabemos que hay una máquina de Turing que para sobre todas las entradas y que acepta $\{a^n b^n \mid n \geq 1\}$. Por el Ejemplo 2.9.1 sabemos que este lenguaje no es regular. Por tanto, hay lenguajes recursivos que no son regulares.

Sea M un autómata de pila no determinista. Podemos construir una máquina de Turing que emule el comportamiento de M en lo que se refiere a a la aceptación o rechazo de cadenas. Para esto debemos tener en cuenta los cambios de estado de M y los cambios que se producen en la pila de M cuando está reconociendo una cadena. Para simplificar, usaremos una máquina de Turing con dos cintas. La cinta 1 almacenará la cadena de entrada y la cinta 2 almacenará el contenido de la pila de M. Un apilamiento corresponde a un movimiento hacia la derecha seguido por una actualización de la cinta 2. Por tanto, la cabeza de lectura/escritura de la cinta 2 siempre analizará el símbolo de la cima de la pila. Los cambios de estado de la máquina de Turing reflejan los cambios de estado de M. Consideremos un ADPND M dado por

$$Q = \{q_1, q_2, q_3\}, \qquad s = q_1$$
$$\Sigma = \{a, b\}, \qquad F = \{q_3\}$$
$$\Gamma = \{a, b, z\},$$
$$\Delta(q_1, \varepsilon, z) = \{(q_3, \varepsilon)\}, \qquad \Delta(q_1, a, z) = \{(q_1, az)\}$$
$$\Delta(q_1, a, a) = \{(q_1, aa)\}, \qquad \Delta(q_1, b, a) = \{(q_2, \varepsilon)\}$$
$$\Delta(q_2, b, a) = \{(q_2, \varepsilon)\}, \qquad \Delta(q_2, \varepsilon, z) = \{(q_3, \varepsilon)\}$$

Este ADPND acepta el lenguaje independiente del contexto $\{a^n b^n \mid n \geq 0\}$.

La máquina de Turing que construiremos empezará en el estado q'. La cabeza de lectura/escritura de la cinta 1 comienza sobre el símbolo del extremo izquierdo de la cadena de entrada y la cinta 2 está vacía (es decir, todos los símbolos son blancos). Primero, marcaremos el "fondo" de la pila en la cinta 2. Por tanto, tendremos las transiciones

$$\delta(q', (a, \not b)) = (q_1, (a, z), (S, S))$$
$$\delta(q', (\not b, \not b)) = (q_1, (\not b, z), (S, S))$$

(La directiva S de la cinta indica, de nuevo, que la correspondiente cabeza de la cinta no se mueve). Los dos casos en los cuales este ADPND realiza un apilamiento se ejecuten primero mediante un movimiento hacia la derecha de la cabeza de la cinta de la pila (cinta 2) y escribiendo un nuevo carácter sobre ella. En este momento, el carácter de entrada es "consumido" mediante el movimiento de la cabeza de la cinta 1 hacia la derecha. Por tanto, tendremos las transiciones

$$\delta(q_1, (a, z)) = (p_1, (a, z), (S, R))$$
$$\delta(q_1, (a, a)) = (p_1, (a, a), (S, R))$$
$$\delta(p_1, (a, \not b)) = (q_1, (a, a), (R, S))$$

las cuales apilan *aes* sobre la pila. Además p_1 es un nuevo estado. Cuando se desapila el ADPND, en la cinta 2 la cabeza realiza un movimiento hacia abajo, es decir, hacia la izquierda. Por tanto, tenemos las transiciones

$$\delta(q_1, (b, a)) = (q_2, (b, a), (R, L))$$
$$\delta(q_2, (b, a)) = (q_2, (b, a), (R, L))$$

Finalmente, la cadena se acepta cuando ha sido totalmente "consumida" y se ha vaciado la pila. La transición del ADPND $\Delta(q_2, \varepsilon, z) = \{(q_3, \varepsilon)\}$ corresponde al movimiento

$$\delta(q_2, (\not b, z)) = (q_3, (\not b, z), (S, L))$$

de la máquina de Turing.

Dado que los lenguajes independientes del contexto son los que acepta un autómata de pila no determinista, tendremos el siguiente teorema:

Teorema 5.2.2. Si L es un lenguaje independiente del contexto, entonces L es un lenguaje recursivo.

Si realizamos la intersección de dos lenguajes recursivos, obtenemos un lenguaje recursivo. Para probarlo, supongamos que M_1 y M_2 son dos máquinas de

Turing que paran sobre toda cadena. Entonces, si $w \in L(M_i)$, sabemos que M_i parará en un estado de aceptación y, si $w \notin L(M_i)$, M_i parará en un estado que no es de aceptación para cada $i = 1, 2$. Sea M una máquina de Turing que tiene dos cintas. La cadena de entrada w estará en la cinta 1 y se hará una copia de ella en la cinta 2. La máquina M emulará a M_1 mediante la cinta 1 hasta que M_1 pare. Si M_1 para en un estado que no es de aceptación, entonces M también para en un estado que no es de aceptación. Por otro lado, si M_1 aceptara w, entonces M emularía a M_2 mediante la cinta 2 (con la copia de w). Si M_2 parara en un estado que no es de aceptación, entonces M también para en un estado que no es de aceptación. Sin embargo, si M_2 aceptara w, entonces M acepta w. Por tanto $w \in L(M)$ si y sólo si $w \in L(M_1)$ y $w \in L(M_2)$, con lo que obtenemos que $L(M) = L(M_1) \cap L(M_2)$. Finalmente, obsérvese que para toda entrada, M para, ya que M_1 y M_2 también lo hacen. Por tanto, $L(M)$ es un lenguaje recursivo. Veamos el siguiente teorema:

Teorema 5.2.3. Si L_1 y L_2 son lenguajes recursivos, entonces $L_1 \cap L_2$ también lo es.

Por el Capítulo 3 sabemos que $L = \{a^i b^i c^i \mid i \geq 0\}$ es la intersección de dos lenguajes independientes del contexto, pero que él mismo no es independiente del contexto. Por medio de los Teoremas 5.2.2 y 5.2.3 se obtiene que L es un lenguaje recursivo. Por tanto, el inverso del Teorema 5.2.2 no se cumple. Es decir, hay lenguajes recursivos que no son independientes del contexto.

Ejercicios de la Sección 5.2

5.2.1. Si L es regular, ¿L es recursivamente enumerable? ¿Por qué?

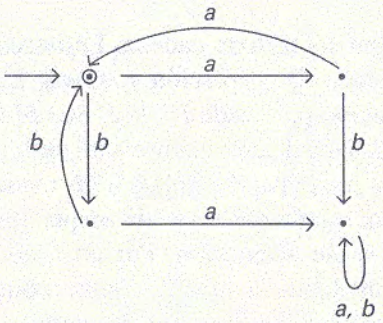
5.2.2. El autómata finito determinista M acepta las cadenas de longitud par sobre $\Sigma = \{a, b\}$. Construir una máquina de Turing, a partir de este autómata, que acepte $L(M)$.

$$M = (Q, \Sigma, s, F, \delta)$$

donde

$$Q = \{q_1, q_2\}$$
$$\Sigma = \{a, b\}$$
$$s = q_1$$
$$F = \{q_1\}$$
$$\delta(q_1, a) = \delta(q_1, b) = q_2$$
$$\delta(q_2, a) = \delta(q_2, b) = q_1$$

5.2.3. Usar la técnica vista en esta sección para construir una máquina de Turing que acepte el lenguaje aceptado por el siguiente autómata finito determinista:



5.2.4. Usar la técnica de construcción de esta sección para construir una máquina de Turing que acepte las mismas cadenas que el ADPND dado por

$$Q = \{q_1, q_2\} \qquad s = q_1$$
$$\Sigma = \{a, b\} \qquad F = \{q_2\}$$
$$\Gamma = \{a, b, z\}$$

y Δ dado por

$$\Delta(q_1, \varepsilon, z) = \{(q_2, z)\} \qquad \Delta(q_1, a, a) = \{(q_1, aa)\}$$
$$\Delta(q_1, a, z) = \{(q_1, az)\} \qquad \Delta(q_1, b, a) = \{(q_1, \varepsilon)\}$$
$$\Delta(q_1, b, z) = \{(q_1, bz)\} \qquad \Delta(q_1, a, b) = \{(q_1, \varepsilon)\}$$
$$\Delta(q_1, b, b) = \{(q_1, bb)\}$$

5.2.5. (a) ¿Cómo se podría tratar la transición

$$\Delta(q, a, b) = \{(p, baaba)\}$$

a la hora de construir una máquina de Turing que corresponda a un ADPND?

(b) ¿Como podría tratarse una transición de la forma

$$\Delta(q, a, b) = \{(q_1, b), (q_2, a)\}$$

al construir una máquina de Turing que corresponda a un ADPND? Obsérvese que, puesto que los ADPND tienen muhas posibilidades de ser no deterministas, se nos puede plantear un problema de estas características. (*Indicación:* ¿Qué sabemos acerca de las máquinas de Turing no deterministas?)

(c) Usar las respuestas a los apartados (a) y (b), además de la técnica de construcción de máquinas de Turing para ADPND, con el objeto de construir una máquina de Turing que acepte el lenguaje aceptado por el ADPND siguiente:

$Q = \{q_1, q_2, q_3, q_4, q_5, q_6\}$ $\quad s = q_1$
$\Sigma = \{a, b\}$ $\quad F = \{q_6\}$
$\Gamma \{a, b, z\}$

y Δ dado por:

$$\Delta(q_1, \varepsilon, z) = \{(q_6, \varepsilon)\}$$
$$\Delta(q_1, a, z) = \{(q_2, babz)\}$$
$$\Delta(q_2, a, b) = \{(q_2, babb), (q_3, babb)\}$$
$$\Delta(q_3, b, b) = \{(q_4, \varepsilon)\}$$
$$\Delta(q_4, a, a) = \{(q_5, \varepsilon)\}$$
$$\Delta(q_5, b, b) = \{(q_3, \varepsilon)\}$$
$$\Delta(q_5, \varepsilon, z) = \{(q_6, \varepsilon)\}$$

[Nótese que el lenguaje aceptado por este autómata es $\{a^n (bab)^n | n \geq 0\}$.]

5.2.6. Definir formalmente la técnica de construcción del Ejercicio 5.2.5, que construye una máquina de Turing M' que acepta el mismo lenguaje que un ADPND $M = (Q, \Sigma, \Gamma, s, F, z, \Delta)$.

5.2.7. En la construcción de una máquina de Turing a partir de un ADPND, se asegura que la máquina de Turing construida para con todas las cadenas, por tanto ¿es posible afirmar que los lenguajes independientes del contexto son recursivos (en vez de sólo recursivamente enumerable)?

5.2.8. Probar que la unión de dos lenguajes recursivos es un lenguaje recursivo. ¿La unión de dos lenguajes recursivamente enumerables arbitrarios es también un lenguaje recursivamente enumerable?

5.2.9. Probar que la intersección de lenguajes recursivamente enumerables es un lenguaje recursivamente enumerable.

5.2.10. Probar que el siguiente lenguaje L sobre el alfabeto $\Sigma = \{a, b\}$ es un lenguaje recursivo. (*Indicación:* Encontrar un lenguaje independiente del contexto apropiado para realizar la intersección).

$L = \{w | w$ contiene el mismo número de *aes* que de *bes* y no contiene ninguna subcadena $aab\}$

5.3 LENGUAJES RECURSIVOS Y RECURSIVAMENTE ENUMERABLES

Supongamos que L es un lenguaje recursivo sobre el alfabeto Σ. Entonces, hay una máquina de Turing $M = (Q, \Sigma, \Gamma, s, b, F, \delta)$ que para sobre toda entrada y acepta L. Considérese la máquina de Turing $M' = (Q, \Sigma, \Gamma, s, b, Q - F, \delta)$. Obsérvese que cualquier cadena de L hace que M' pare en algún estado de F y, por tanto, M' rechaza todas las cadenas de L. Por otro lado, si $w \notin L$, M parará en algún estado que no es de F, cuando w esté en su cinta al principio de la computa-

ción. Es decir, M para en algún estado de $Q - F$. Pero entonces, M' también parará y, por tanto, si $w \notin L$, entonces $w \in L(M')$. Así, $L(M') = \Sigma^* - L$. Es más, ya que M para sobre todas las cadenas, entonces M' también lo hace. Finalmente, tendremos el siguiente lema:

Lema 5.3.1. Si L es un lenguaje recursivo, entonces $\Sigma^* - L$ es un lenguaje recursivo.

Ésta es una propiedad de los lenguajes recursivos que, en general, no se cumple para los lenguajes recursivamente enumerables. Para verlo, supongamos que Σ es un alfabeto. Ya que Σ^* es numerable, podemos enumerarlo como $\Sigma^* = \{w_1, w_2, w_3, \ldots\}$. En la sección de las máquinas de Turing universales indicamos que todas las máquinas de Turing sobre Σ podían ser enumeradas (por medio de las cadenas de símbolos generadas para representar las máquinas de Turing codificadas). Por tanto, podremos listar todas las máquinas de Turing sobre Σ como $M_1, M_2, \ldots$ Sea

$$L = \{w_i \mid w_i \text{ es aceptado por } M_i\}$$

Este lenguaje es recursivamente enumerable, pero su complemento no.

Primero, veremos que L es recursivamente enumerable, obteniendo una máquina de Turing M que acepte L. M será una composición de varias máquinas de Turing. Si se tiene una cadena w sobre Σ, obsérvese que $w = w_i$ para algún i. M primero generará $w_1, w_2, \ldots$ hasta que encuentre un i para el cual $w = w_i$. Entonces M generará la (codificada) i-ésima máquina de Turing M_i, codificará w_i, y pasará de la codificación de M_i y w_i a la máquina de Turing universal U que emule M_i sobre w_i. Si M_i para y acepta w_i, entonces U para en un estado de aceptación y, por tanto, M para y acepta w_i.

Por otro lado, M_i puede que pare sin aceptar w_i o puede que no pare. En ambos casos, M no para y no acepta w_i. Así, $w_i \in L(M)$ si y sólo si $w_i \in L(M_i)$.

Segundo, observemos que $\Sigma^* - L$ no es recursivamente enumerable. Para ello, supongamos que $\Sigma^* - L$ es recursivamente enumerable. Entonces debe ser aceptado por una máquina de Turing, que llamaremos M_j. Consideremos w_j. Si $w_j \in L(M_j)$, tendremos que $w_j \in L = \Sigma^* - L(M_j)$ y, por tanto, $w_j \notin L(M_j)$. A la inversa, si $w_j \notin L(M_j)$, entonces $w_j \notin L$ y, por tanto, se tiene que $w_j \in \Sigma^* - L = L(M_j)$. En ambos casos se llega a una contradicción, luego el lenguaje $\Sigma^* - L$ no es aceptado por ninguna máquina de Turing. Veamos el siguiente teorema:

Teorema 5.3.2. Hay un lenguaje recursivamente enumerable L para el cual $\Sigma^* - L$ no es recursivamente enumerable.

Obsérvese que el lenguaje L descrito aquí no puede ser recursivo porque, si lo fuera, su complemento también lo sería y, por tanto, L sería recursivamente

enumerable. Luego tendremos un lenguaje recursivamente enumerable que no es recursivo. Por consiguiente, la inversa de la sentencia que dice que todo lenguaje recursivo también es recursivamente enumerable no es cierta.

Como ya hemos comentado, las máquinas de Turing pueden considerarse como modelos de computación mecánica. Una máquina de Turing que realiza una tarea —ya sea reconocer un lenguaje o realizar un patrón sobre una cinta multidimensional— modela un proceso. Los procesos que siempre terminan se llaman algoritmos y, por tanto, una máquina de Turing que para sobre cualquier cadena, es un modelo de algoritmo. Esto significa que, para los lenguajes recursivos, hay un algoritmo que determina si una cadena w está en un lenguaje L. Este algoritmo será modelado por una máquina de Turing que para sobre cualquier entrada y acepta L.

Por otro lado, si L es recursivamente enumerable pero no recursivo, no hay una máquina de Turing que pare sobre todas las entradas y acepte L. De esto se deduce que no hay ningún modelo de algoritmo que determine si $w \in L$, para una cadena arbitraria w. El problema de los miembros (¿está w en L?) para los lenguajes recursivamente enumerables arbitrarios que no son recursivos es un ejemplo del problema de irresolubilidad, que es un problema para el cual no hay ningún algoritmo que pueda dar una respuesta al mismo, ya sea negativa o positiva. Trataremos estas cuestiones en el Capítulo 6.

El Teorema 5.2.3 junto con los Ejercicios 5.2.8 y 5.2.9 muestran que el conjunto de los lenguajes recursivos es cerrado con respecto a la unión y la intersección. También los lenguajes recursivamente enumerables son cerrados con respecto a la unión y la intersección. La demostración de que la intersección de lenguajes recursivamente enumerables es recursivamente enumerable es similar a la de los lenguajes recursivos y fue vista en el Ejercicio 5.2.8. La demostración de que la unión de lenguajes recursivamente enumerables es también un lenguaje recursivamente enumerable proporciona una técnica de demostración muy útil e interesante.

Teorema 5.3.3. Si L_1 y L_2 son lenguajes recursivamente enumerables, entonces $L_1 \cup L_2$ es también recursivamente enumerable.

Demostración. Si L_1 y L_2 son recursivos, entonces sabemos que su unión también es recursiva y, por tanto, recursivamente enumerable. Luego para este caso quedaría probado.

Supongamos que, al menos uno de los dos, es recursivamente enumerable pero no recursivo. Sea $L_i = L(M_i)$, donde $M_i = (Q_i, \Sigma, \Gamma_i, s_i, \flat, F_i, \delta_i)$ son máquinas de Turing para $i = 1$ e $i = 2$. Vamos a hacer un esbozo de la construcción de una máquina de Turing $M = (Q, \Sigma, \Gamma, s, \flat, F, \delta)$ que acepte $L_1 \cup L_2$. En otras palabras, M acepta w si y sólo si w es aceptada por al menos una de las M_i, M será

una máquina de Turing con dos cintas que simulará simultáneamente M_1 en la cinta 1 y M_2 en la cinta 2. Si M_1 o M_2 para en un estado de aceptación, M también para y acepta la cadena de entrada. Si una de la dos para en un estado que no es de aceptación antes de que la otra pare, M centrará su atención en la máquina de Turing que todavía no paró. Si, para la cadena w, ninguna de las máquinas de Turing para (y por tanto $w \notin L_1 \cup L_2$), M nunca parará y, por tanto, nunca aceptará w.

Esta simulación se realiza incluyendo en Q los pares de $Q_1 \times Q_2$ y permitiendo todo movimiento de la forma

$$\delta((q_1, q_2), (\sigma_1, \sigma_2)) = (p_1, p_2), (\tau_1, \tau_2), (X_1, X_2))$$

donde para cada $i = 1, 2$,

$$\delta_i(q_i, \sigma_i) = (p_i, \tau_i, X_i)$$

es una transición de M_i.

Además, para incluir las situaciones en las que una de las máquinas de Turing para en un estado que no es de aceptación antes de que la otra pare, necesitamos incluir las transiciones que permitan que M siga simulando la otra máquina de Turing. Por ejemplo, si M_1 parara en el estado q_3, que no es de aceptación (en este caso, M_1 no tendría ninguna transición desde q_3 sobre cualquier símbolo de la cinta), deberíamos incluir en M todas las transiciones de la forma

$$\delta((q_3, q), (\sigma_1, \sigma_2)) = ((q_3, p), (\sigma_1, \tau_2), (S, X_2))$$

donde $\delta_2(q, \sigma_2) = (p, \tau_2, X_2)$ es una transición de M_2. Si hiciéramos esto para todos los estados en los cuales M_1 o M_2 para sin aceptar ninguna cadena, entonces M no pararía en todas ellas cuando M_1 y M_2 parasen sin aceptar "una cadena" ¿por qué se considera aceptable este comportamiento?

Finalmente para cada par (q_1, q_2) en los cuales q_1 ó q_2 es un estado de aceptación de su correspondiente máquina de Turing, incluiremos las transiciones de la forma

$$\delta((q_1, q_2), (\sigma_1, \sigma_2)) = (q, (\sigma_1, \sigma_2), (S, S))$$

donde $q \in Q$ es un nuevo estado que es el único estado de aceptación de M. □

La demostración del Teorema 5.3.3 sugiere una técnica que puede ser usada para probar que cualquier lenguaje recursivamente enumerable cuyo complemento es recursivamente enumerable debe ser un lenguaje recursivo. Para probarlo, supongamos que L y $\Sigma^* - L$ son ambos recursivamente enumerables. Usaremos el método de construcción usado en la demostración del Teorema 5.3.3 para construir una máquina de Turing que acepte $L \cup (\Sigma^* - L)$. Naturalmente, esta unión es Σ^*, pero puesto que $L \cap (\Sigma^* - L) = \emptyset$, cualquier cadena $w \in \Sigma^*$ es-

tará solamente en uno de los dos. Así, si tenemos las máquinas de Turing M_1 y M_2 que aceptan L y $\Sigma^* - L$, respectivamente, entonces para cualquier $w \in \Sigma^*$ solamente una de las dos máquinas de Turing parará y aceptará w. Modificaremos la máquina de Turing M de la demostración precedente puesto que, si M_1 parase y aceptase w, entonces M la aceptaría, mientras que si M_2 aceptase w, entonces M pararía y la rechazaría. Luego tenemos una máquina de Turing que para todas la entradas, para y acepta L. Por tanto, se puede deducir el siguiente teorema:

Teorema 5.3.4. Si L es un lenguaje recursivamente enumerable para el cual $\Sigma^* - L$ también es recursivamente enumerable, entonces L es un lenguaje recursivo.

Un lenguaje es *enumerado* por una máquina de Turing M si, sobre su cinta, genera las cadenas del lenguaje separándolas mediante algún marcador. Obsérvese que, si el lenguaje es finito, la máquina de Turing puede parar. Por otro lado, si el lenguaje es infinito, entonces la máquina de Turing no parará.

Si L es enumerado por la máquina de Turing M, entonces podemos construir una máquina de Turing M' que acepte L. M' tiene una cinta más que M, la cual usa como cinta de entrada. Para analizar la cadena w, la sitúa sobre esta cinta. Entonces, M' simulará M sobre la(s) otra(s) cinta(s), excepto cuando M' genere una cadena, ya que entonces M' parará y comparará la cadena que acaba de ser generada con la cadena w. Si coinciden, M' para en un estado de aceptación, en otro caso, M' continúa. De esto se deduce que todo lenguaje enumerado por una máquina de Turing es recursivamente enumerable.

Por otro lado, si L es recursivamente enumerable, entonces podemos construir una máquina de Turing que los enumere. Para ello, supongamos que M es una máquina de Turing que acepta a L. Podemos construir una máquina de Turing M' con tres cintas, que enumere L. La idea básica es la siguiente: Las cadenas de L serán enumeradas sobre la cinta 1. Sobre la cinta 2, M' generará las cadenas de Σ^* siguiendo un cierto orden. Sobre la cinta 3, M' simulará las acciones que realiza M sobre las cadenas generadas. Si una cadena fuera aceptada por M, entonces M' la escribiría en la cinta 1, por tanto enumeraría L. Nótese que, ya que sólo se requiere que L sea recursivamente enumerable y *no* necesariamente recursiva, la simulación de las acciones de M sobre una cadena puede producir problemas si M nunca para sobre esa cadena.

Este problema se puede evitar consiguiendo que M' simule secuencias (finitas) de movimientos de M cada vez más largas, mas que intentando procesar las cadenas completamente. M' genera las cadenas en orden lexicográfico (alfabético). Supongamos que $\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_n\}$. Primero, M' genera la cadena vacía, ε. Entonces M' genera todas las cadenas de longitud 1 en el orden $\sigma_1, \sigma_2, ..., \sigma_n$. Después M' genera todas las cadenas de longitud 2; $\sigma_1\sigma_1, \sigma_1\sigma_2, ...,$

$\sigma_1\sigma_n$, $\sigma_2\sigma_1$, $\sigma_2\sigma_2$, ..., $\sigma_2\sigma_n$, ..., $\sigma_n\sigma_n$, y así sucesivamente. (Obsérvese que el orden en el que son listadas, determina el orden lexicográfico de las cadenas).

Para organizar las simulaciones de secuencias de movimientos de M sobre dichas cadenas, M' genera ε primero y simula un movimiento sobre ella. En un segundo paso, M' simula dos movimientos sobre ε (si es posible), genera σ_1 y simula un movimiento sobre ella. En tercer lugar, M' simulará tres movimientos sobre ε, dos sobre σ_1, y entonces genera σ_2 y simula un movimiento sobre ella. La cinta 2 se usa para llevar cuenta de las cadenas generadas y del número de movimientos realizados sobre ellas. Por tanto, después de i pasos, la cinta 2 contendrá i cadenas sobre Σ^* así como el total de movimientos realizados sobre ellas. En los siguientes pasos, M' procesa cada una de las i cadenas incrementando además el contador de movimientos para cada una de ellas, copiándola en la cinta 3 y simulando M sobre ella con el número de movimientos especificado. Si M acepta la cadena, entonces M' la copia en la cinta 1.

Nótese que, de esta forma, se elimina el problema de que M nunca pare sobre una cadena específica. Cualquier cadena de L se genera sobre la cinta 2 y se acepta durante la simulación de M. Por tanto, cualquier cadena de L aparece en la cinta 1. Veamos el siguiente teorema:

Teorema 5.3.5. Un lenguaje L es recursivamente enumerable si y sólo si L es enumerado por alguna máquina de Turing.

Por tanto, la capacidad de ser enumerado por una máquina de Turing, caracteriza completamente a los lenguajes recursivamente enumerables.

Ejercicios de la Sección 5.3

5.3.1. En la construcción de la demostración del Teorema 5.3.3, si M_1 y M_2 rechazaran una cadena al pararse en un estado que no es de aceptación, la máquina de Turing M rechazaría la cadena al no pararse. Realizar un esbozo de la forma en la que se debe construir M para que también pare en un estado que no es de aceptación.

5.3.2. Usar la misma construcción anterior para probar que $L_1 \cap L_2$ es recursivamente enumerable si L_1 y L_2 son recursivamente enumerables.

5.3.3. Esbozar la máquina de Turing descrita mediante las observaciones anteriores al Teorema 5.3.4. ¿Cómo se determina si la que acepta la cadena es M_1 o M_2?

5.3.4. Supongamos que $L_1, L_2, ..., L_n$ son lenguajes que forman una partición de Σ^*. Si todos los L_i son recursivamente enumerables, probar que cada uno de ellos también es recursivo.

5.3.5. La técnica descrita en esta sección para enumerar un lenguaje recursivamente enumerable mediante una máquina de Turing no es muy eficiente. Las cadenas se escriben repetidamente sobre la cinta 1, puesto que una vez que su contador de movimientos tiene una valor mayor o igual que el número de movimientos, M necesita aceptarlas y serán escritas en la cinta 1. Sugerir una técnica para eliminar esta redundancia (teniendo en cuenta que puede que sea necesario que las cadenas se queden en la cinta 2 para que pueda continuar el proceso de generación de Σ^*, en orden lexicográfico).

5.3.6. Supongamos que L es recursivamente enumerable pero no recursivo, Probar que, para toda máquina de Turing M que acepte L, hay un número infinito de cadenas que pueden hacer que M no pare.

5.4 GRAMÁTICAS NO RESTRINGIDAS Y LENGUAJES RECURSIVAMENTE ENUMERABLES

A través del estudio de lenguajes que hemos realizado, hemos tratado dos técnicas para especificarlos. Podemos especificar un lenguaje describiendo un procedimiento para reconocer sus cadenas. Estos procedimientos tomaron la forma de autómatas de varios tipos. Alternativamente, se puede especificar un lenguaje por medio de una técnica que genere sus cadenas. Las gramáticas nos proporcionan este método.

Tanto en las gramáticas regulares como en las que son independientes del contexto, se restringió la manera de formar las producciones. ¿Hasta qué punto se pueden relajar estas restricciones y hacer que todavía tengan sentido los lenguajes resultantes? Es más, ¿cómo se pueden relacionar dichos lenguajes con los que son aceptados por las máquinas de Turing?

Si representamos las producciones de las gramáticas regulares por medio del producto cartesiano de conjuntos, éstas serán pares de $N \times \Sigma^* (N \cup \varepsilon)$ (para gramáticas regulares por la derecha), que indican que el lado izquierdo de la producción debe estar formado por un único no terminal, mientras que el lado derecho de la producción debe estar compuesto por cualquier cadena de terminales seguida por un no terminal. Al pasar a las gramáticas independientes del contexto, la estructura de la parte derecha de las producciones se hizo menos restrictiva; es decir, se permitió que fueran pares del producto $N \times (N \cup \Sigma)^*$. Ahora, el lado derecho podrá contener cualquier número de no terminales, mientras que el lado izquierdo sigue restringido a un único no terminal.

Aunque en una gramática independiente del contexto la estructura del lado derecho de las producciones parece ser lo más general posible, podemos dar al menos dos sugerencias con respecto al lado izquierdo. Se debería poder permitir que en el lado izquierdo hubiera cadenas no vacías de no terminales e incluso cadenas de no terminales y terminales. De las dos, las cadenas no vacías de ter-

minales y no terminales a la vez, es la forma más general. Según esto, nuestras producciones podrían ser pares del producto $(N \cup \Sigma)^+ \times (N \cup \Sigma)^*$. De esta manera hemos eliminado todas las restricciones en cuanto a la manera de formar las producciones. (Obsérvese que, si se permitiera que el lado izquierdo de una producción fuera la cadena vacía, se tendría la posibilidad de tener más de un punto de partida para las derivaciones, lo que no es deseable. Por tanto, no permitiremos ε como lado izquierdo de una producción).

Veamos la siguiente definición:

Definición 5.4.1. Una *gramática no restringida* (que también se conoce como una gramática estructurada por frases) es una 4-tupla $G = (N, \Sigma, S, P)$, donde

N es un alfabeto de símbolos no terminales
Σ es un alfabeto de símbolos terminales con $N \cap \Sigma = \emptyset$
$S \in N$ es el símbolo inicial
P es un conjunto finito de producciones de la forma $\alpha \to \beta$, donde $\alpha \in (N \cup \Sigma)^+$ y $\beta \in (N \cup \Sigma)^*$ (es decir, $P \subset (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$ y es un conjunto finito).

Obsérvese que cualquier gramática regular o independiente del contexto es además una gramática no restringida. Como cabría esperar, tenemos una potencia generativa mayor que la que tienen las gramáticas independientes del contexto o las regulares, debido a que la gramática es menos restrictiva con respecto a la formación de las producciones.

Por ejemplo, consideremos la gramática no restringida dada por

$$\begin{aligned} S &\to aSBC \mid aBC \\ CB &\to BC \\ aB &\to ab \\ bB &\to bb \\ bC &\to bc \\ cC &\to cc \end{aligned}$$

(En estas gramáticas mantenemos las convenciones adoptadas al estudiar las gramáticas regulares y las gramáticas independientes del contexto. Es decir, S sigue siendo el símbolo inicial, las minúsculas los terminales y las mayúsculas los no terminales).

En la cadena que resulta al sustituir el símbolo inicial, siempre hay el mismo número de *aes*, *Bes* y *Ces*. Es más, todas las *aes* preceden a las *Bes* y a las *Ces*. La producción $CB \to BC$ nos permite intercambiar los no terminales C y B, obteniendo una cadena de la forma $a^n B^n C^n$. Las producciones $aB \to ab$ y

$bB \rightarrow bb$, transforman toda B en b, obteniéndose $a^n b^n C^n$. Finalmente, las producciones $bC \rightarrow bc$ y $cC \rightarrow cc$ transforman la cadena en $a^n b^n c^n$.

Por otro lado, si $n = 1$, esta gramática puede derivar abc por medio de $S \Rightarrow aBC \Rightarrow abC \Rightarrow abc$. Si $n \geq 1$, entonces la cadena $a^n b^n c^n$ se deriva por medio de

$$S \overset{n-1}{\Rightarrow} a^{n-1} S (BC)^{n-1} \Rightarrow a^n (BC)^n \overset{n}{\Rightarrow} a^n B^n C^n \overset{n}{\Rightarrow} a^n b^n C^n \overset{n}{\Rightarrow} a^n b^n c^n$$

Por tanto, en G se deriva cualquier cadena de la forma $a^n b^n c^n$ para $n \geq 1$.

De esto se deduce que $L(G) = \{a^n b^n c^n \mid n \geq 1\}$. Luego con esta gramática podemos derivar lenguajes que no son independientes del contexto.

Veamos otro ejemplo de la potencia generativa de las gramáticas no restringidas. Consideremos la gramática

$$\begin{aligned} &S \rightarrow ACaB, && Ca \rightarrow aaC \\ &CB \rightarrow DB \mid E, && aD \rightarrow Da \\ &AD \rightarrow AC, && aE \rightarrow Ea \\ &AE \rightarrow \varepsilon \end{aligned}$$

Esta gramática genera $L = \{a^{2^k} \mid k > 0\}$. Para hacernos una idea de cómo se generan cadenas en L, consideremos la derivación $a^{2^2} = a^4$. Tendremos

$$\begin{aligned} S &\Rightarrow ACaB \Rightarrow AaaCB \Rightarrow AaaDB \Rightarrow AaDaB \Rightarrow ADaaB \Rightarrow ACaaB \\ &\Rightarrow AaaCaB \Rightarrow AaaaaCB \Rightarrow AaaaaE \Rightarrow AaaaEa \Rightarrow AaaEaa \\ &\Rightarrow AaEaaa \Rightarrow AEaaaa \Rightarrow aaaa \end{aligned}$$

A y B actúan como marcadores de final de la cadena de *aes* que están siendo generadas. C se desplaza hacia la derecha al duplicarse el número de *aes* hasta que está junto a B, entonces se transforma en una D. D se desplaza hacia la derecha hasta que encuentra una A, y entonces se convierte en C. Cuando CB es reemplazada por E, termina la generación de *aes*. Entonces la E se desplaza hacia la izquierda hasta que encuentra la A, momento en el cual se elimina AE.

Supongamos que G es una gramática no restringida. Se pueden listar todas las w para las cuales $S \Rightarrow w$, es decir, todas las cadenas que son derivadas en un sólo paso. El número de cadenas de este tipo que se puede obtener será finito, puesto que la colección de producciones de la gramática es finita. También podremos hacer una lista de las cadenas que se derivan en dos pasos, en tres pasos, y así sucesivamente. Cada una de estas colecciones será también finita. De lo que se deduce que $L(G)$ puede ser listada de esta forma. Lo que quiere decir que podría haber una máquina de Turing que enumerase $L(G)$. Esto no nos sorprende puesto que las gramáticas generan cadenas en un orden determinado.

Aplicando el Teorema 5.3.5 a las observaciones anteriores podemos deducir el siguiente teorema:

Teorema 5.4.2. Si G es una gramática no restringida, entonces $L(G)$ es un lenguaje recursivamente enumerable.

También se cumple que todo lenguaje recursivamente enumerable es generado por una gramática no restringida. Para verlo, supongamos que L es un lenguaje recursivamente enumerable que es aceptado por una máquina de Turing $M = (Q, \Sigma, \Gamma, s = q_1, \flat, F, \delta)$. Vamos a construir una gramática no restringida, G con base en M. Si $w \in L(M)$, entonces $q_1 w \overset{*}{\vdash} xq_f y$, donde $q_f \in F$ es un estado de aceptación de M y x e y son cadenas de $(\Gamma - \{\flat\})^*$. La gramática G produce la cadena w, mediante una derivación hacia atrás, desde $xq_f y$ a $q_1 w$, eliminando después, q_1. G se define como $G = (N, \Sigma, S, P)$, donde N es la coleción de símbolos $N = (\Gamma - \Sigma) \cup Q \cup \{A_1, A_2, S\}$, siendo A_1, A_2 y S son unos símbolos nuevos.

Las producciones de P son de cuatro tipos. Primero queremos que G derive una configuración final de la forma $xq_f y$. Esto se consigue mediante las producciones de la forma

$$S \to \flat S \mid S\flat \mid A_1 A_2$$
$$A_2 \to aA_2 \mid A_2 a \mid q$$

para toda $a \in \Gamma - \{\flat\}$ y toda $q \in F$. (Obsérvese que $\flat \in \Gamma \subseteq N$ con lo que $\flat S$ y $S\flat$ pertenecen a $(N \cup \Sigma)^*$.) Introducimos A_1 como un marcador, que solamente se elimina al final de la derivación. Mediante dichas producciones se obtienen derivaciones de la forma

$$S \overset{*}{\Rightarrow} \flat \ldots \flat A_1 x q_f y \flat \ldots \flat$$

para todo x e y pertenecientes a $(\Gamma - \{\flat\})^*$ y los estados finales q_f de F. Puesto que estamos realizando un proceso hacia atrás para que M acepte una cadena, debemos pasar de la configuración $xq_f y$ a $q_1 w$, lo que significa que necesitamos reemplazar $A_1 x q_f y$ por $A_1 q_1 w$. Para ello añadimos a P las producciones de la forma $bp_j \to q_i a$, donde $\delta(q_i, a) = (q_j, b, R)$ es una transición de M, y las producciones de la forma $q_j cb \to cq_i a$ para cualquier símbolo c de Γ, donde $\delta(q_i, a) = (q_j, b, L)$ es una transición de M. El primer tipo de producciones (correspondientes a los movimientos hacia la derecha), "deshacen" lo hecho por la transición de M, al sustituir la b por la a y desplazarse al no terminal que se encuentre a la izquierda de a. En las producciones correspondientes a los movimientos hacia la izquierda, se reemplazan las cadenas de la forma $q_j cb$ por las cadenas de la forma $cq_i a$, donde c es cualquier símbolo que se encontraba originalmente a la izquierda de la cabeza de M. Por tanto, para la transición

$\delta(q_i, a) = (q_j, b, L)$ correspondiente a un movimiento hacia la izquierda, debemos añadir todas las producciones de la forma $q_j cb \rightarrow cq_i a$, donde $c \in \Gamma$.

Finalmente, puesto que pueden quedar blancos $\flat$ y algunos símbolos A_1 y q_1, limpiaremos la cadena por medio de las producciones

$$A_1 q_1 \rightarrow \varepsilon$$
$$\flat \rightarrow \varepsilon$$

Como un ejemplo muy sencillo de la construcción de dicha gramática, consideremos la máquina de Turing $M = (Q, \Sigma, \Gamma, s = q_1, \flat, F, \delta)$, teniendo

$$Q = \{q_1, q_2, q_3\}$$
$$\Sigma = \{a\}$$
$$\Gamma = \{a, \flat\}$$
$$F = \{q_3\}$$
$$\delta(q_1, a) = (q_1, a, R)$$
$$\delta(q_1, \flat) = (q_2, \flat, R)$$
$$\delta(q_2, \flat) = (q_3, \flat, L)$$

Esta máquina de Turing acepta a^*. Aplicando la construcción anterior, obtendremos los siguientes grupos de producciones:

$$\left.\begin{array}{r} S \rightarrow \flat S \mid S\flat \mid A_1 A_2 \\ A_2 \rightarrow aA_2 \mid A_2 a \mid q_3 \end{array}\right\} \quad \text{para el primer paso}$$

$$\left.\begin{array}{r} aq_1 \rightarrow q_1 a \\ \flat q_1 \rightarrow q_1 \flat \end{array}\right\} \quad \begin{array}{l} \text{para las transiciones que mueven} \\ \text{la cabeza de } M \text{ hacia la derecha} \end{array}$$

$$\left.\begin{array}{r} q_3 a\flat \rightarrow aq_2\flat \\ q_3\flat\flat \rightarrow \flat q_2\flat \end{array}\right\} \quad \begin{array}{l} \text{para las transiciones que mueven} \\ \text{la cabeza de } M \text{ hacia la izquierda} \end{array}$$

$$\left.\begin{array}{r} A_1 q_1 \rightarrow \varepsilon \\ \flat \rightarrow \varepsilon \end{array}\right\} \quad \text{para el último paso}$$

Para comprender la conexión que existe entre la forma en la que M acepta una cadena y la forma en la que G genera la misma cadena, consideremos $w = a^2$. M acepta w por medio de las siguientes transiciones:

$$q_1 a^2 \vdash aq_1 a \vdash a^2 q_1 \flat \vdash a^2 \flat q_2 \flat \vdash a^2 q_3 \flat\flat$$

Esto corresponde a que G primero genera $A_1a^2q_3\flat\flat$ y después retrocede desde esta configuración de M a $A_1q_1a^2$, posteriormente borra la subcadena A_1q_1. Finalmente, tendremos la siguiente derivación

$$\begin{aligned} S &\Rightarrow S\flat \Rightarrow S\flat\flat \Rightarrow A_1A_2\flat\flat \Rightarrow A_1aA_2\flat\flat \Rightarrow A_1a^2A_2\flat\flat \\ &\Rightarrow A_1a^2q_3\flat\flat \Rightarrow A_1a^2\flat q_2\flat \Rightarrow A_1a^2q_1\flat\flat \Rightarrow A_1aq_1a\flat\flat \\ &\Rightarrow A_1q_1a^2\flat\flat \overset{*}{\Rightarrow} A_1q_1a^2 \Rightarrow a^2 \end{aligned}$$

Teorema 5.4.3. Si $L = L(M)$ es un lenguaje recursivamente enumerable y G se construye de la forma que hemos descrito anteriormente, entonces $L = L(G)$.

Demostración. Tenemos que demostrar que si $w \in L(M)$, entonces $w \in L(G)$ y viceversa.

Primero, supongamos $w \in L(M)$. Entonces se debe cumplir que $q_1w \overset{*}{\vdash} xq_fy$ para las cadenas x e y de $(\Gamma - \{\flat\})^*$. A partir de la construcción de G que hemos visto, obtendremos

$$S \overset{*}{\Rightarrow} \flat \ldots \flat A_1xq_fy\flat \ldots \flat$$

Esto siempre es posible, puesto que de la primera colección de producciones construida, podríamos derivar cualquier cadena de esta forma.

Ahora supongamos que M aplica la transición $\delta(q_i, b_1) = (q_j, c, L)$, durante la computación que acepta w, obteniéndose las siguientes configuraciones

$$a_1 \ldots a_nq_ib_1 \ldots b_m \vdash a_1 \ldots a_{n-1}q_jcb_2 \ldots b_m$$

Puesto que la transición $\delta(q_i, b_1) = (q_j, c, L)$ produce $q_ja_nc \rightarrow a_nq_ib_1$ de G, tendremos

$$a_1 \ldots a_nq_ib_1 \ldots b_m \Rightarrow a_1 \ldots a_{n-1}q_jcb_2 \ldots b_m$$

que es correcto en una derivación de G. Aplicaremos un razonamiento similar para los movimientos de M hacia la derecha. Mediante un razonamiento inductivo, obtendremos que

$$\flat \ldots \flat A_1xq_fy\flat \ldots \flat \overset{*}{\Rightarrow} \flat \ldots \flat A_1q_1w\flat \ldots \flat$$

Finalmente, aplicando la "limpieza" de producciones para eliminar A_1q_1 y todos los blancos, tendremos $S \overset{*}{\Rightarrow} w$. Es decir, $w \in L(G)$.

A la inversa, si $w \in L(G)$, entonces $S \overset{*}{\Rightarrow} w$, y por tanto se tiene

$$\begin{aligned} S &\overset{*}{\Rightarrow} \flat \ldots \flat A_1 x q_f y \flat \ldots \flat \\ &\overset{*}{\Rightarrow} \flat \ldots \flat q_1 w \flat \ldots \flat \\ &\overset{*}{\Rightarrow} w \end{aligned}$$

con lo que obtenemos que $xq_fy \overset{*}{\Rightarrow} q_1w$. Puesto que q_1w corresponde a la configuración inicial de M cuando comienza el procesamiento de w, un razonamiento inductivo sencillo probará que $q_1w \overset{*}{\vdash} xq_fy$ y, por lo tanto, $w \in L(M)$. □

Uniendo los Teoremas 5.4.2 y 5.4.3, obtendremos el siguiente resultado:

Teorema 5.4.4. Un lenguaje L es recursivamente enumerable si y sólo si $L = L(G)$ para alguna gramática G, no restringida.

Los lenguajes que se obtienen a partir de las gramáticas no restringidas son exactamente los mismos que son aceptados por máquinas de Turing.

Ejercicios de la Sección 5.4

5.4.1. ¿Qué lenguaje genera esta gramática?

$$\begin{aligned} S &\to Ba \mid a, & Sa &\to Bb \\ Sb &\to \varepsilon, & B &\to bS \mid BB \mid b \end{aligned}$$

¿Es un lenguaje independiente del contexto? ¿Es regular?

5.4.2. Obtener una gramática no restringida para cada uno de estos lenguajes

(a) $\{a^nb^na^nb^n \mid n \geq 0\}$

(b) $\{a^ib^jc^k \mid i < j < k\}$

(c) $\{ww \mid w \in \{a, b\}^*\}$

(d) $\{www \mid w \in \{a, b\}^*\}$

5.4.3. Algunos textos definen una gramática no restringida como aquella en la cual todas las producciones son de la forma $\alpha \to \beta$, donde $\alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*$ y β igual que en nuestra definición. Es decir, α contiene al menos un no terminal. Ésta fue la definición que dimos en la Sección 4.3, de manera informal. Probar por qué las gramáticas de este tipo no tienen ni mayor ni menor potencia generativa que las que corresponden a nuestra definición. (*Indicación:* ¿Cómo se podría transformar una gramática de este tipo en una del otro? ¿A qué afecta dicha transformación?)

5.4.4. Hacer un esbozo del diseño de una máquina de Turing que enumere $L(G)$ para una gramática no restringida G (*Indicación:* Considérese una máquina de Turing que tenga al menos tres cintas. La cinta 1 es la cinta de salida sobre la cual

se enumera $L(G)$, la cinta 2 contiene la lista de las producciones de G y la cinta 3 es la cinta de trabajo. Buscar una forma para que dicha máquina de Turing genere, en primer lugar, las cadenas que se derivan de G en un sólo paso, ordenadas de alguna manera sobre la cinta 3 y que luego las copie sobre la cinta 1; después genere las cadenas que se derivan de G en dos pasos, y así sucesivamente).

5.4.5. En las observaciones anteriores al Teorema 5.4.3 y en su demostración se plantean, implícitamente, dos requerimientos que deben cumplir las máquinas de Turing que se transforman en gramáticas no restringidas. Primero, la máquina de Turing debe ser *no supresora*. Es decir, no puede tener ninguna transición de la forma $\delta(q, \sigma) = (p, b, X)$ para cualquier $\sigma \neq b$. Segundo, se supone que la máquina de Turing, cuando pare (al aceptar una cadena), lo hará con su cabeza de lectura/escritura sobre el primer blanco seguido de la cadena que no sea de blancos, el cual estará a la izquierda de la cinta.

(a) Probar que si L es aceptado por una máquina de Turing M arbitraria, entonces L será aceptada por una máquina de Turing M' no supresora.

(b) Construir una máquina de Turing no supresora que acepte el lenguaje $\{a^n b^m \mid n, m \geq 0\}$, y que pare sobre el primer blanco que vaya seguido de la cadena que queda sobre su cinta.

(c) Usar la técnica vista en esta sección para obtener la gramática no restringida asociada a este lenguaje.

(d) Obtener una derivación (usando la gramática de la parte c) y la secuencia de las DI de la máquina de Turing de la parte b que se obtienen al aceptar la cadena a^2ba.

5.4.6. Probar que, si L_1 y L_2 son lenguajes recursivamente enumerables, entonces L_1L_2 y L_1^* son lenguajes recursivamente enumerables.

5.5 LENGUAJES SENSIBLES AL CONTEXTO Y LA JERARQUÍA DE CHOMSKY

Entre las gramáticas no restringidas y la forma restringida de las gramáticas independientes del contexto, se pueden definir varias gramáticas con diferentes niveles de restricción. No todas producen una clase de lenguajes interesante, pero unas que si lo hacen son las que forman el conjunto de gramáticas sensibles al contexto. Las gramáticas sensibles al contexto producen una clase de lenguajes que está estrictamente situada entre los lenguajes independientes del contexto y los lenguajes recursivos.

Definición 5.5.1. Una gramática $G = (N, \Sigma, S, P)$ es una *gramática sensible al contexto* si todas las producciones son de la forma $\alpha \rightarrow \beta$, donde $\alpha, \beta \in (N \cup \Sigma)^+$ y $|\alpha| \leq |\beta|$.

Por ejemplo, la gramática dada por

$$\begin{aligned} S &\to abc \mid aAbc \\ Ab &\to bA \\ Ac &\to Bbcc \\ bB &\to Bb \\ aB &\to aa \mid aaA \end{aligned}$$

es una gramática sensible al contexto. Esta gramática genera el lenguaje $\{a^n b^n c^n \mid n \geq 1\}$, con lo que tenemos un ejemplo de un lenguaje sensible al contexto que *no* es independiente del contexto.

Recordemos que toda gramática independiente del contexto se puede poner en forma normal de Chomsky, en la cual las producciones son de la forma $A \to \alpha$ o también $A \Rightarrow BC$. Puesto que las producciones de esta forma satisfacen la definición de gramáticas sensibles al contexto, se deduce que toda gramática independiente del contexto es también una gramática sensible al contexto. Por tanto podremos enunciar el siguiente lema:

Lema 5.5.2. El conjunto de los lenguajes sensibles al contexto contiene el conjunto de los lenguajes independientes del contexto.

La restricción de que el lado derecho de las producciones en una gramática sensible al contexto sea al menos tan largo como el lado izquierdo hace que la gramática sea *no contráctil*. Puesto que la cadena vacía tiene longitud 0, podemos deducir de la definición que $\varepsilon \notin L(G)$, para cualquier gramática G sensible al contexto. A veces es conveniente que la cadena pertenezca al lenguaje que ha sido generado por una gramática sensible al contexto. Luego extenderemos nuestra definición con el fin de permitir producciones de la forma $S \to \varepsilon$ (siendo S el símbolo inicial), con tal de que S no aparezca en el lado derecho de cualquier producción.

Lema 5.5.3. Sea $G = (N, \Sigma, S, P)$ una gramática sensible al contexto. Entonces, existe una gramática $G_1 = (N_1, \Sigma, S_1, P_1)$ sensible al contexto, para la cual $L(G) = L(G_1)$ y S_1 nunca aparece en el lado derecho de una producción de P_1.

Demostración. Sea $N_1 = \{S_1\} \cup N$, donde S_1 es un símbolo nuevo. Sea P_1 el conjunto definido por $P_1 = P \cup \{S_1 \to \alpha \mid S \to \alpha$ es una producción de $G\}$. Obsérvese que P_1 satisface la condición de que S_1 nunca aparece en el lado derecho de una producción. Es más, si $S \overset{*}{\Rightarrow} w$ es una derivación en G, entonces alguna producción $S \to \alpha$ es el primer paso de la misma (es decir, se tiene que $S \Rightarrow \alpha \overset{*}{\Rightarrow} w$). Pero

entonces $S_1 \rightarrow \alpha$ es una producción de P_1 y, por tanto, $S_1 \Rightarrow \alpha \Rightarrow w$ es una derivación de G_1. Así $w \in L(G)$ si y solo si $w \in L(G_1)$. □

Supongamos que $G = (N, \Sigma, S, P)$ es una gramática sensible al contexto y que $w \in L(G)$. Si $w = \varepsilon$, entonces se tiene que $S \rightarrow \varepsilon$. Si $w \neq \varepsilon$, entonces habrá alguna derivación tal que $S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_m = w$. Puesto que una gramática sensible al contexto es no contráctil, se debe cumplir que

$$|\alpha_1| \leq |\alpha_2| \leq \ldots \leq |\alpha_m| = |w|$$

Si $m > |w|$ (es decir, la derivación se realiza en más pasos que símbolos tiene w), entonces existirá algún i, j y p para los cuales

$$|\alpha_i| = |\alpha_{i+1}| = \ldots = |\alpha_{i+j}| = p$$

Es decir, durante $j + 1$ pasos de la derivación, la cadena que se está generando no crece en longitud. Ahora bien, puesto que N y Σ son conjuntos finitos, tendremos que $|N \cup \Sigma| = k$ para algún k, y por tanto hay k^p cadenas posibles de longitud p. Si el número de pasos de la derivación donde la cadena parcialmente derivada tiene longitud p es mayor que el número de cadenas posibles de dicha longitud (es decir, si $j + 1 > k^p$), entonces al menos dos de los $\alpha_i, \alpha_{i+1}, \ldots, \alpha_{i+j}$ deben ser el mismo. Entonces, en este caso, podemos eliminar de la derivación al menos un paso. Es decir, si $\alpha_r = \alpha_s$, entonces

$$S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_r \Rightarrow \alpha_{s+1} \Rightarrow \ldots \Rightarrow \alpha_m = w$$

eliminando $\alpha_{r+1} \Rightarrow \ldots \Rightarrow \alpha_s$, con lo que se obtiene una derivación más corta.

La idea es que si w es derivable en G entonces hay alguna derivación de w que "no es demasiado larga". Lo que indica que hay un algoritmo para determinar si $w \in L(G)$. Es decir, podemos obtener una máquina de Turing que para sobre toda entrada y que acepta $L(G)$.

Lema 5.5.4. Sea $G = (N, \Sigma, S, P)$ una gramática sensible al contexto. Entonces existe una máquina de Turing T, que para sobre toda entrada y acepta $L(G)$.

Demostración. Vamos a describir una máquina de Turing, M, no determinista con tres cintas que acepta $L(G)$. La cinta 1 es la cinta de entrada. La cinta 3 guarda la derivación de la cadena de entrada w, si existe tal derivación. La cadena 2 contiene las producciones de G, donde la producción $u \rightarrow v$ se representa mediante $u\#v$.

Una computación de M con una cadena de entrada w consiste en las tres etapas siguientes:

1. Se escribe la cadena Sb en la cinta 3.
2. M genera las producciones de G sobre la cinta 2.
3. Hasta que M para, se repiten los pasos siguientes:
   3a. Se elige una producción ubv de la cinta 2.
   3b. Si bxb es la cadena que se encuentra más a la derecha en la cinta 2 (la que se derivó más recientemente), se elige, si existe, un ejemplo de una subcadena u de x. Si no hay una subcadena u de x, entonces M para.
   3c. Se reemplaza por v la subcadena u de la cadena bxb anterior y se escribe sobre la cinta 3 la cadena x' que se obtiene, seguida por bxb. (Por tanto, si bxb es la cadena bx_1ux_2b, escribiríamos $x' = x_1vx_2$ sobre la cinta 3, y a continuación se pondría bxb.)
   3d. Si $x' = w$, entonces M para en un estado de aceptación.
   3e. Si x' está en la cinta 3, pero en otro lugar, entonces M para y rechaza w.
   3f. Si $|x'| > |w|$, entonces M para en un estado que no es de aceptación.

Ya que sólo hay un número finito de cadenas de $(N \cup \Sigma)^*$ cuya longitud es menor o igual que $|w|$, toda derivación para, entra en un ciclo o deriva una cadena de longitud mayor que $|w|$. Todas estas posibilidades se contemplan en (3b), (3d), (3e) o (3f). Por tanto M para sobre todas las cadenas. Como se vio en las observaciones que preceden a este teorema, toda cadena de $L(G)$ se puede derivar mediante una derivación no cíclica. Si w se deriva mediante tal derivación, M simula la derivación y para. □

El Teorema 5.5.5 se obtiene a partir del Lema 5.5.4.

Teorema 5.5.5. Si L es un lenguaje sensible al contexto, entonces L es recursivo.

Consideremos el conjunto de todos los lenguajes sensibles al contexto con $\Sigma = \{a, b\}$. Dado que todo conjunto de no terminales es finito, supongamos que los renombramos, con lo que obtenemos $N \subseteq \{A_1, A_2, \ldots\}$ si $G = (N, \Sigma, S, P)$ es cualquiera de las gramáticas sensibles al contexto. Suponemos que A_1 es, siempre, el símbolo inicial. Vamos a codificar todas las gramáticas sensibles al contexto sobre $\{a, b\}$ como cadenas de 0 y 1.

Primero codificaremos los símbolos a y b como 00 y 001. Después codificaremos cada no terminal A_i como 01^i. Representaremos la flecha ($\rightarrow$) de las producciones mediante 0011 y la coma (,) mediante 00111. Toda gramática sensible

al contexto, se puede describir como una cadena de producciones separadas por comas (usando la flecha, $\rightarrow$, para separar los lados izquierdo y derecho de las producciones) y, por tanto, se puede codificar como una cadena de ceros y unos, formadas como hemos visto.

Obsérvese que cualquier gramática sensible al contexto puede ser descodificada fácilmente. Nótese, también, que no todas las cadenas de ceros y unos representan gramáticas sensibles al contexto, pero que dada una cadena, se puede decir fácilmente si representa a una gramática sensible al contexto sobre $\{a, b\}$.

Si generamos cadenas de ceros y unos en orden lexicográfico (alfabético), podemos comprobar si cada una de ellas representa a una gramática sensible al contexto codificada. Por tanto, podemos encontrar la i-ésima gramática G_i sensible al contexto codificada mediante la generación de cadenas en este orden hasta que se genera la i-ésima cadena que es una gramática sensible al contexto codificada. Luego podremos indexar las gramáticas sensibles al contexto sobre $\{a, b\}$, de forma significativa, $G_1, G_2, \ldots$

Ahora, supongamos que enumeramos $\{a, b\}^*$ en orden alfabético, como $w_1, w_2, \ldots$ Definimos $L = \{w_i | w_i \notin L(G_i)\}$. Podemos probar de forma bastante sencilla que L es recursivo. Dado $w \in \{a, b\}^*$, determinamos i para el cual $w = w_i$. Entonces generamos G_i y determinamos si $w = w_i \in L(G_i)$ usando el algoritmo anterior (Lema 5.5.4). Componemos las máquinas de Turing apropiadas para obtener una máquina de Turing M que pare sobre todas las entradas con $L = L(M)$.

Probaremos que ninguna gramática sensible al contexto genera L, realizando una sencilla diagonalización. Supongamos que $L = L(G)$ para alguna gramática sensible al contexto G sobre $\{a, b\}$. Entonces $G = G_i$ para algún i. Si $w_i \in L = L(G_i)$, entonces mediante la definición de L obtenemos que $w_i \notin L(G_i)$, lo que es una contradicción. A la inversa, si $w_i \notin L = L(G_i)$, entonces, nuevamente por la definición de L, $w_i \in L$. Lo que de nuevo es una contradicción. Por tanto concluimos que ninguna gramática sensible al contexto sobre $\{a, b\}$ genera L. De ello se deduce que L es un lenguaje recursivo que no es sensible al contexto.

Luego tendremos el siguiente lema:

Lema 5.5.6. Hay lenguajes recursivos que no son lenguajes sensibles al contexto.

Obtendremos el siguiente teorema, a partir de los Lemas 5.5.2 y 5.5.6. y del Teorema 5.5.5.

Teorema 5.5.7. Los lenguajes sensibles al contexto contienen, *propiamente*, a los lenguajes independientes del contexto A su vez, los lenguajes recursivos contienen *propiamente* a los lenguajes sensibles al contexto.

En 1959, Noam Chomsky clasificó las gramáticas en cuatro familias. Las gramáticas no restringidas, sensibles al contexto, independientes del contexto y regulares, que se conocen como gramáticas de tipo 0, de tipo 1, de tipo 2 y de tipo 3, respectivamente. Los lenguajes que resultan de dichas gramáticas también se identifican como lenguajes de tipo 0, 1, 2 y 3. Como ya hemos visto, todo lenguaje independiente del contexto que contiene la cadena vacía se genera mediante una gramática independiente del contexto en la cual $S \rightarrow \varepsilon$ es la única producción ε. Igualmente, hemos demostrado que todo lenguaje sensible al contexto que contiene ε se deriva de una gramática "limpia" en la cual $S \rightarrow \varepsilon$ es la única producción ε. Dejando a un lado los problemas que se derivan de la palabra vacía, vimos que todo lenguaje de tipo i es también de tipo $i - 1$, teniendo un tipo de inclusión propia.

Esta jerarquía de lenguajes se conoce como la *jerarquía de Chomsky*. La Figura 5.1 ilustra las relaciones que existen.

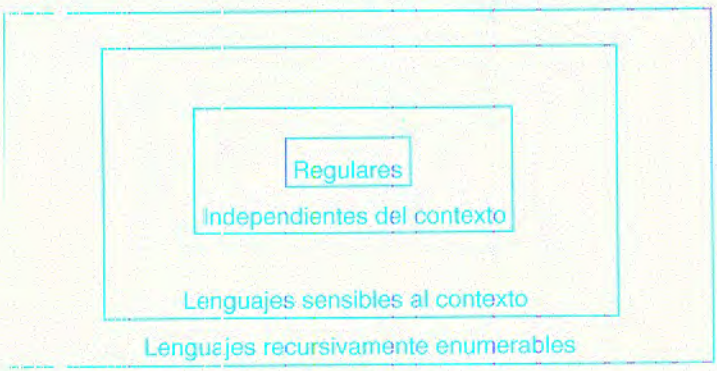


Figura 5.1

Además de los tipos de lenguajes/gramáticas estudiados por Chomsky, también hemos estudiado los lenguajes/gramáticas recursivas que se encuentran entre los lenguajes recursivamente enumerables y los lenguajes sensibles al contexto. Podemos definir otras familias de lenguajes y su lugar dentro de la jerarquía que hemos presentado.

Concluimos esta sección y capítulo con el *teorema de la jerarquía*, el cual resume todas las relaciones entre lenguajes/gramáticas que hemos descrito.

Teorema 5.3. Sobre un alfabeto dado, el conjunto de los lenguajes recursivamente enumerables contiene propiamente al conjunto de los lenguajes recursivos que contiene propiamente al conjunto de los lenguajes sensibles al contexto que contiene propiamente al conjunto de lenguajes independientes del contexto, que a su

vez contiene propiamente a los lenguajes regulares. Resumiendo, sea $\mathcal{L}_x$ la representación del conjunto de lenguajes x. Entonces se tiene

$$\mathcal{L}_{\text{regular}} \subset \mathcal{L}_{\text{i.c.}} \mathcal{L}_{\text{d.c.}} \subset \mathcal{L}_{\text{recursivo}} \subset \mathcal{L}_{\text{r.e.}}$$

Ejercicios de la Sección 5.5.

5.5.1. Si una gramática sensible al contexto puede contener la producción $S \to \varepsilon$, ¿por qué se requiere que S nunca aparezca en el lado derecho de una producción?

5.5.2. En la máquina de Turing descrita en la demostración del Lema 5.5.4, ¿dónde se presenta el no determinismo?

5.5.3. En la máquina de Turing M descrita en la demostración del Lema 5.5.4, ¿por qué M debe generar las reglas de producción de G sobre la cinta 2?

5.5.4. Aunque la máquina de Turing vista en el Lema 5.5.4 proporciona un modelo de un algoritmo para los elementos de una gramática sensible al contexto, éste no es eficiente. Este ejercicio estudia un algoritmo más eficiente.

Sea $G = (N, \Sigma, S, P)$ una gramática sensible al contexto. Para determinar si $w \in L(G)$, lo podemos hacer, fácilmente, comprobando si $\varepsilon \in L(G)$. Supongamos que $w \in \Sigma^*$ y que $|w| = n > 0$. Sea $T_0 = \{S\}$ y, en general, sea

$$T_m = \{\alpha \in (N \cup \Sigma)^+ \mid S \overset{k}{\Rightarrow} \alpha \text{ para } k \leq m \text{ y } |\alpha| \leq n\}$$

(a) Probar que $T_m = T_{m-1} \cup \{\alpha \mid \beta \overset{*}{\Rightarrow} \alpha \text{ para } \beta \in T_{m-1} \text{ y } |\alpha| \leq n\}$

(b) Probar que $T_m = A \cap B$ para los conjuntos

$$A = \{\alpha \in (N \cup \Sigma)^* \mid |\alpha| \leq n\}$$

$$B = \{\alpha \in (N \cup \Sigma)^* \mid S \overset{*}{\Rightarrow} \alpha \text{ en no más de } m \text{ pasos}\}$$

Obsérvese que:

i. Si $S \overset{*}{\Rightarrow} \alpha$ y $|\alpha| \leq n$, entonces se obtiene que $\alpha \in T_m$, para algún m.

ii. Si S no deriva α o si $|\alpha| > n$, entonces $\alpha \notin T_m$ para todo m.

iii. $T_{m-1} \subseteq T_m$.

iv. Si $T_{m-1} = T_m$, entonces $T_m = T_{m+1} = \ldots$

(c) Probar que (iv) es cierto

Por tanto, para determinar si $w \in L(G)$, calcularemos T_0, T_1 y así sucesivamente, hasta que obtengamos que $T_{m-1} = T_m = \ldots$ En este momento, ya se han generado todas las cadenas de longitud menor o igual que n, con lo que basta con que comprobemos T_{m-1} para w. Si $w \in T_{m-1}$, entonces $w \in L(G)$; en otro caso, $w \notin L(G)$.

(d) Sea G la gramática dada por las producciones

$$\begin{aligned} S &\to aSBC \mid aBC, & bB &\to bb \\ CB &\to BC, & bC &\to bc \\ aB &\to ab, & cC &\to cc \end{aligned}$$

1. Usar el algoritmo para probar que $abac \notin L(G)$.
2. Usar el algoritmo para determinar si $abaa$ y $aabbcc$ son cadenas de $L(G)$.

5.5.5. Dada una cadena de *ceros* y *unos*, proponer una técnica que determine si dicha cadena representa a una gramática sensible al contexto sobre $\{a, b\}$ codificada de la forma descrita en el Lema 5.5.6.

5.5.6. Codificar

$$\begin{aligned} A_1 &\to aA_2A_1 \mid b \\ A_2a &\to aA_2 \mid ab \\ bA_2 &\to bb \end{aligned}$$

mediante el método descrito en el Lema 5.5.6.

5.5.7. Sea $G = (N, \Sigma, S, P)$ una gramática no restringida. Sea c un símbolo terminal que no pertenece a Σ. Construir una gramática sensible al contexto G' basada en G, con $G' = (N, \Sigma \cup \{c\}, S, P')$ para la cual $w \in L(G)$ si y sólo si $wc^k \in L(G')$ para algún $k \geq 0$.

5.5.8. Sea C un mecanismo de alguna clase. Supongamos que podemos enumerar C mediante $M_1, M_2 \ldots$ y que hay un algoritmo que, dado un mecanismo M de C y una entrada w, determina si M acepta w. Probar que no todos los lenguajes recursivos sobre el alfabeto Σ son aceptados por un mecanismo de C.

5.5.9. ¿El conjunto de los lenguajes sensibles al contexto es cerrado con respecto a la unión, la concatenación o la cerradura de estrella? ¿Por qué?

PROBLEMAS

5.1. Si L_1, L_2 son lenguajes recursivamente enumerables sobre Σ, ¿Es $\bigcup_{i=1}^{\infty} L_i$ recursivamente enumerable? ¿Por qué?

5.2. Probar que el complemento de un lenguaje independiente del contexto es recursivo.

5.3. Si L_1 es recursivo y L_2 es recursivamente enumerable, probar que $L_1 - L_2$ es recursivamente enumerable. ¿$L_2 - L_1$ es recursivo? ¿Es recursivamente enumerable?

5.4. Si L_1 y L_2 son lenguajes recursivamente enumerables sobre Σ, esbozar la construcción de unamáquina de Turing que acepte L_1L_2 y L_1^*. (*Indicación:* considerar máquinas de Turing no deterministas).

5.5. Nuestro razonamiento de que los lenguajes generados por gramáticas no restringidas son recursivamente enumerables (Teorema 5.4.2) se basaba en la capacidad de poder obtener una máquina de Turing que enumerase las cadenas del lenguaje de forma ordenada. Se puede obtener un razonamiento alternativo mediante la construcción de una máquina de Turing no determinista que acepte las cadenas del lenguaje.

Sea G una gramática no restringida. La máquina de Turing M que construyamos elegirá de forma no determinista una derivación de G y la simulará sobre su cinta. Entonces comparará la cadena derivada con la cadena de entrada y, si coinciden, la aceptará. Si no coinciden, M simulará otra derivación, y así sucesivamente. Si la cadena de entrada está en $L(G)$, entonces M simulará una derivación de la misma; en otro caso M nunca parará.

Para realizar la simulación, primero se escribe el símbolo inicial S sobre la cinta. Entonces M entra en un bucle, el cual puede que termine después de un cierto numero de pasos. Cada paso corresponde a una elección de una producción de G realizada de forma no determinista, que se tratará de aplicar, y de una posición en la cadena que está en la cinta, y a la cual se pretende aplicar la producción. Una vez que se selecciona la posición, M comprueba si dicha posición comienza con una secuencia de símbolos que se corresponden con el lado izquierdo de la producción seleccionada y, si es así, se reemplaza la secuencia por el lado derecho de la producción. Este bucle se repite hasta que M elija, de forma no determinista, que quiere salir del mismo. Entonces será cuando se compare la cadena generada con la cadena de entrada para comprobar si coinciden.

1. Obsérvese que el no determinismo está presente en tres momentos de la simulación que M realiza: en la elección de la producción a usar, en la elección de la posición inicial y al tomar la decisión de salirse del bucle. ¿Por qué es necesario que exista no determinismo cuando se decide la salida del bucle?

2. Describir la máquina de Turing M.

   a. ¿Cómo se pueden representar las producciones de G de forma que se pueda elegir entre ellas de forma no determinista?

   b. ¿Qué cinta y qué alfabeto se necesitan incluir?

   c. Describa la mayoría de los componentes de M.

3. Describir con cuidado todos los pasos necesarios para realizar una vuelta del bucle.

   a. ¿En qué afecta el no determinismo a la elección de una producción? ¿En la posición inicial? ¿En la salida del bucle?

   b. ¿Cómo se aplica una producción que se haya elegido y para cuya parte izquierda se encuentre una concordancia?

5.6. En el Problema 4.1, se vio que el lenguaje $L = \{a^n b^n c^n \mid n \geq 1\}$ era aceptado por una autómata linealmente acotado. Esto sugiere que existe una conexión entre los lenguajes sensibles al contexto y el autómata linealmente acotado.

1. Probar que si G es una gramática independiente del contexto entonces existe un autómata linealmente acotado que acepta $L(G)$. Para simplificar, supongamos que $L(G)$ no contiene la cadena vacía. (*Indicaciones:* Considérese un autómata linealmente acotado con dos pistas en el cual las producciones de G se codifican mediante los estados y las transiciones. Revisar la demostración del Lema 5.5.4).

   También se cumple el inverso de lo anterior. Es decir, si M es un autómata linealmente acotado, entonces $L(M)$ es un lenguaje sensible al contexto y, por tanto, existe una gramática sensible al contexto que lo genera. Las observaciones que preceden al Teorema 5.4.3. presentan una técnica para la construcción de una gramática basada en una máquina de Turing arbitraria. Obsérvese que todas las producciones que se obtienen son no contráctiles excepto $A_1 q_1 \to \varepsilon$ y $b \to \varepsilon$. En el caso de que la máquina de Turing fuera un autómata linealmente acotado, no sería necesaria la producción $b \to \varepsilon$ porque un autómata linealmente acotado nunca puede pasar sus marcadores de final. Si se omite la producción $A_1 q_1 \to \varepsilon$, entonces el lenguaje que se genera será $\{A_1 q_1 w \mid w \in L(M)\}$. Obsérvese que si M es un autómata linealmente acotado, dicho lenguaje difiere de $L(M)$ en que toda palabra tendrá un prefijo no deseable. Puesto que en una gramática independiente del contexto no podemos borrar símbolos, no se tiene la capacidad para eliminar dichos prefijos.

   Sea $M = (Q, \Sigma, \Gamma, q_1, b, F, \Delta)$ un autómata linealmente acotado y supongamos que $\varepsilon \notin L(M)$. Definiremos una gramática sensible al contexto $G = (N, \Sigma, S, P)$ para la cual $L(G) = L(M)$. La gramática sensible al contexto que construyamos generará dos copias de una cadena y simularemos las acciones de M sobre una de las copias. Si M acepta la cadena, G elimina la cadena que se obtiene de la ejecución de M, dejando solamente la cadena que se acepta. Para que G sea sensible al contexto, vamos a evitar la necesidad de tener producciones que eliminen símbolos, construyendo no terminales que contengan una gran cantidad de información. Si a y b pertenecen a Σ y $q \in Q$, N tendría (como símbolos no terminales) pares de la forma

$$(*) \begin{cases} (a, a), \quad (a, <b), \quad (a, b>), \quad (a, <b>), \quad (a, qb), \quad (a, q<b) \\ (a, <qb), \quad (a, qb>), \quad (a, bq>), \quad (a, q<b>), \quad (a, <qb>), \quad (a, <bq>) \end{cases}$$

   N contiene todos esos pares para todo $a, b \in \Sigma$ y para todo $q \in Q$. [Obsérvese que si se sustituye un no terminal $(a, bq>)$ por el terminal a no violamos la condición de contráctil.] En todos los casos, el primer elemento representa el símbolo terminal que se encuentra originalmente en dicha

posición y que no será alterado durante la simulación de M. El segundo elemento se usa en la simulación de M, como describiremos brevemente.

N también contiene los símbolos S y A. S es el símbolo inicial de G y A se usa para generar cadenas de pares que representan la cadena de entrada para M. Las producciones para S y A son

$S \rightarrow (a, q_1 < a) A | (a, q_1 < a >)$, para todo $a \in \Sigma$ (q_i es el estado inicial de M)

$A \rightarrow (a, a) A | (a, a >)$, para todo $a \in \Sigma$

Dichas producciones pueden generar cadenas de la forma

$$(a_{i_1}, q_1 < a_{i_1}) (a_{i_2}, a_{i_2}) \ldots (a_{i_k}, a_{i_k} >)$$

Obsérvese cómo es la cadena formada por los segundos elementos

$$q_1 < a_{i_1} a_{i_2} \ldots a_{i_k} >$$

la cual representa una configuración inicial de M sobre la cadena de entrada formada por los primeros elementos.

2. Proponer una forma para construir las producciones que simulan la computación de M sobre la cadena formada por los segundos elementos de una cadena de pares ordenados. *Indicaciones:* Dichas producciones se formarían con los no terminales que pertenecen al conjunto anterior (*). Para ver un ejemplo, consideremos el funcionamiento de un ALA M (con un estado inicial q_1 y un estado de aceptación q_n) que acepta el lenguaje formado por las cadenas de *aes* y *bes* en las que al menos hay una a y una b. M acepta *aba* de esta forma:

$$q_1 < aba > \vdash < q_1 aba > \vdash < a q_2 ba > \vdash < ab q_3 a > \vdash < aba q_3 > \vdash < aba q_4 >$$

Si la cadena generada por G se corresponde con una cadena de $L(M)$, la simulación de M llegará a un estado de aceptación y terminará, cuando dicha simulación se realice sobre la cadena formada por los segundos elementos. Será en este momento, cuando deseemos transformarla en una cadena formada por los primeros elementos que sea una cadena de símbolos terminales. Esto se realiza mediante las producciones de la forma

(1) $(a, q < b) \rightarrow a$, $(a, bq >) \rightarrow a$,
$(a, < qb) \rightarrow a$, $(a, q < b) \rightarrow a$, para todo $q \in F$
$(a, qb) \rightarrow a$, $(a, < qb) \rightarrow a$, y toda $b \in \Sigma$
$(a, qb >) \rightarrow a$, $(a, < bq) \rightarrow a$

y de la forma

(2) $(a, x)\, b \rightarrow ab$,
$b\,(a, x) \rightarrow ba$, para toda $b \in \Sigma$ y $(a, x) \in N$

3. ¿Por qué se necesita que las producciones de (1) dependan de q, pero las de (2) no?

4. a. Obtener un ALA que acepte el lenguaje de los palíndromos sobre $\{a, b\}$.

   b. Usar la técnica anterior para transformar un ALA en una gramática sensible al contexto.

5.7. Sea $\alpha \rightarrow \beta$ una producción no contráctil (es decir, $|\alpha| \leq |\beta|$).

1. Construir una secuencia de producciones no contráctiles, de forma que el lado derecho de cada una tenga una longitud menor o igual a 2 y que produzca β a partir de α (es decir, $\alpha \overset{*}{\Rightarrow} \beta$ por medio de dichas producciones).

2. Construir una secuencia de producciones de la forma $uAv \rightarrow uwv$ (es decir, $\alpha \rightarrow \beta$ donde

$$\alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^* \quad \text{y} \quad \beta \in (N \cup \Sigma)^* \Sigma^+ (N \cup \Sigma)^*)$$

   lo cual produce los mismos resultados que $AB \rightarrow CD$.

3. Demostrar que todo lenguaje sensible al contexto se genera mediante una gramática sensible al contexto en la cual cada producción es de la forma $uAv \rightarrow uwv$ para $w \in \Sigma^+$ y $u, v \in (N \cup \Sigma)^*$.

6

Resolubilidad*

6.1 EL PROBLEMA DE PARADA

En la Definición 4.2.2 definimos que una función f es *Turing computable*, o *computable*, si existe una máquina de Turing que computa $f(w)$ para toda w perteneciente al dominio de f. Un ejemplo de función computable es la función característica de un lenguaje recursivo. [Recuérdese que la función característica, χ_L, es la función que vale 1 (sí) ó 0 (no) dependiendo de si el argumento pertenece al lenguaje o no]. De forma intuitiva, sabemos por el Capítulo 5, que la función característica de un lenguaje recursivamente enumerable que no es recursivo, *no* es computable, ya que existen algunas cadenas de dicho lenguaje para las que la máquina de Turing que lo acepta, no para. Por lo tanto tenemos ejemplos de funciones computables y no computables.

Por otro lado, en este capítulo estudiaremos ciertas cuestiones acerca de la computabilidad, en función de las situaciones en las cuales el resultado de la computación es sí o no (o lo que es igual, 1 ó 0). Los problemas de este tipo se llaman *problemas de decisión* y dicha clase de computabilidad se conoce como *resolubilidad.*

Por ejemplo, el problema "dada una gramática independiente del contexto G, ¿$L(G)$ es vacío?" es un problema del tipo mencionado anteriormente. Obsérvese que hay un número infinito de *casos* para este problema —uno por cada gramática independiente del contexto. Además, para cada caso, se puede determinar una respuesta afirmativa o negativa (véase Sección 2.6).

---

* NOTA: En la jerga matemática e informática, se usa el término "decidibilidad" como traducción del término "decidability". Nosotros no lo hemos adoptado debido a no figurar en el DRAE.

Otro ejemplo sería la pregunta "dada una gramática sensible al contexto y una cadena w, ¿$w \in L(G)$?" Nuevamente obsérvese que hay infinitos casos de este problema. Cada uno de ellos se determina mediante una gramática sensible al contexto y una cadena. Es más, cada uno tiene una respuesta sí o no (véase Sección 5.5).

Se dice que los problemas de decisión son *resolubles** si existe un algoritmo que es capaz de responder sí o no a cada uno de dichos casos. Si el algoritmo no existe, se dice que el problema es *irresoluble*. Los problemas de los ejemplos anteriores son ambos resolubles puesto que existe un algoritmo para ellos.

Quizás el problema irresoluble más conocido es el *problema de parada* para máquinas de Turing. El problema de parada es el siguiente:

> Sea M una máquina de Turing arbitraria con un alfabeto de entrada Σ. Sea $w \in \Sigma^*$. ¿Parará M con la cadena w como cadena de entrada?

Un *caso* del problema de parada está formado por la pareja máquina de Turing-cadena de entrada. Una solución a dicho problema sería un algoritmo que, para cualquier caso, respondiera sí o no de forma correcta, es decir, un algoritmo lo bastante general como para responder correctamente al problema de parada para cualquier combinación de máquina de Turing y cadena de entrada.

Al final de nuestro estudio sobre las máquinas de Turing pusimos de relieve la estrecha relación que existe entre ellas y los algoritmos. En el Capítulo 5 especificamos dicha relación: nuestro modelo de algoritmo es una máquina de Turing que para sobre todas las entradas. Por tanto, podemos buscar una solución al problema de parada, buscando, entre la información que tenemos sobre máquinas de Turing y cadenas de entrada, una máquina de Turing que pare sobre todas las entradas y proporcione una respuesta positiva o negativa.

En la Sección 4.5 hablamos de la máquina de Turing universal y, en particular, presentamos una técnica para codificar las máquinas de Turing sobre cualquier alfabeto de entrada Σ como cadenas de *ceros* y *unos*. Hay una cantidad numerable de cadenas de ceros y unos y, desde luego, no todas tienen por qué ser una codificación de una máquina de Turing. Por tanto como máximo hay un número numerable de máquinas de Turing con un alfabeto de entrada Σ. Se pueden enumerar las máquinas de Turing sobre Σ, como $M_1, M_2, \ldots$ Además, puesto que Σ^* también es numerable, podemos enumerarlo como $\Sigma^* = \{w_1, w_2, \ldots\}$.

Consideremos el lenguaje

$$L = \{w_i | w_i \text{ no es aceptada por } M_i\}$$

* Ver nota de la página anterior.

Se puede afirmar que L no es un lenguaje recursivamente enumerable. Para demostrarlo, supongamos que L es un recursivamente enumerable. Entonces L debe ser aceptado por alguna máquina de Turing, llamada M_k. Consideremos w_k. Obsérvese que, si $w_k \in L$, entonces w_k no debe ser aceptada por M_k, con lo que $w_k \notin L(M_k) = L$, lo que es una contradicción. Por otro lado, si $w_k \notin L$, entonces ya que $L = L(M_k)$, se obtiene que $w_k \notin L(M_k)$ y, por tanto, w_k debe estar en L, lo que es otra contradicción. De esto se deduce que no hay ninguna máquina de Turing que acepte L y que, por tanto, no puede ser recursivamente enumerable.

Ahora supongamos que el problema de parada tiene solución. Es decir, que hay una máquina de Turing que para sobre todas las cadenas de entrada y, que ante una descripción de una máquina de Turing y una cadena de entrada (ambas codificadas), se puede determinar si M para sobre la cadena de entrada. Entonces, se puede construir una máquina de Turing M_L que acepte el lenguaje anterior. Sea w una cadena. Primero, M_L enumera w_1, w_2, ... hasta que encuentra el k para el cual $w = w_k$. Entonces M genera M_k y pasa el código correspondiente a w_k y M_k a la supuesta máquina de Turing, la cual determina si M_k para sobre la entrada w_k. Si se obtiene que M_k no para sobre la entrada w_k, entonces M_L para y acepta $w_k = w$ (¿por qué?). Por otro lado, si se obtiene que M_k para sobre la entrada w_k, entonces se pasan los códigos correspondientes a M_k y w_k a la máquina de Turing universal que simula M_k sobre w_k. En este caso, la máquina de Turing universal parará, determinando si M_k acepta w_k. Si M_k acepta w_k, entonces M_L para y rechaza w_k. Si M_k no acepta w_k, entonces M_L para y acepta w_k.

Por tanto, $w \in L(M_L)$ si y sólo si $w \in L$ y, entonces, se obtiene que $L = L(M_L)$. Esto, a su vez, contradice el hecho de que L no es recursivamente enumerable. Por tanto, no puede haber un algoritmo general que dé respuesta al problema de parada para una combinación arbitraria máquina de Turing-cadena de entrada. Luego podemos enunciar el siguiente teorema:

Teorema 6.1.1. El problema de parada para las máquinas de Turing es irresoluble.

Podemos usar la irresolubilidad del problema de parada para demostrar que otros problemas también son irresolubles. Una forma de hacerlo es demostrando que, si un determinado problema se puede resolver, entonces el problema de parada también es resoluble.

Por ejemplo, el *problema de la cinta en blanco* consiste en el problema de decidir si una máquina de Turing parará cuando comience con una cinta en blanco. Para demostrar que el problema de la cinta en blanco es irresoluble, demostraremos que, si fuera resoluble, el problema de parada también lo sería. Entonces, consideremos que el problema de la cinta en blanco es resoluble. Sean M una máquina de Turing y w cualquier cadena. Sea M' la máquina de Turing que comienza con una cinta en blanco, luego escribe w sobre la cinta y se ejecuta

como si hubiera comenzado con la configuración q_1w (donde q_1 es el estado inicial de M). Obsérvese que M' es una máquina de Turing que al principio tenía la cinta en blanco. Si obtenemos un algoritmo que determine si una máquina de Turing arbitraria que comienza con una cinta en blanco para, podremos determinar si M' para. Pero M' para si y sólo si la máquina de Turing M original, para con la cadena w como entrada. Por tanto podríamos obtener una solución para el problema de parada si existiera un algoritmo general para el problema de la cinta en blanco. Esto contradice el Teorema 6.1.1 y, por tanto, el problema de la cinta en blanco es también irresoluble.

La técnica que relaciona el problema de parada con el problema de la cinta en blanco de forma que nos permite deducir a partir de la irresolubilidad del problema de parada la irresolubilidad del problema de la cinta vacía, se llama *reducción*. Se dice que el problema de parada se *reduce* al problema de la cinta vacía porque la resolubilidad del problema de la cinta vacía nos permite deducir que la resolubilidad del problema de parada (Un problema X se reduce al problema Y si, al obtener la solución de Y, se puede obtener la solución de X.)

Ejercicios de la Sección 6.1

6.1.1. Demostrar que si el problema de parada se puede resolver, entonces todo lenguaje recursivamente enumerable es recursivo.

6.1.2. Sea M una máquina de Turing con un alfabeto de entrada Σ. Sea w_m una codificación de M sobre $\{0, 1\}$. Sea

$$L = \{w_m w \mid M \text{ para sobre la cadena de entrada codificada como } w\}$$

Demostrar que L es recursivamente enumerable. Demostrar que L no es recursivo.

6.1.3. El *problema de la entrada en un estado* para máquinas de Turing se puede enunciar como sigue:

Para una máquina de Turing arbitraria $M = (Q, \Sigma, \Gamma, s, b, F, \delta)$, el estado q y la cadena $w \in \Sigma^*$, ¿entrará M en el estado q cuando comience con la cadena w? Demostrar que este problema es irresoluble aplicando la reducción correspondiente al problema de parada.

6.1.4. El *problema de la vacuidad*, "es $L(M) = \emptyset$" para una máquina de Turing arbitraria, también es una problema irresoluble. Demostrar que este problema es irresoluble mediante la reducción del problema de la cinta en blanco al mismo.

6.1.5. Mediante reducciones apropiadas, demostrar que cada uno de los problemas siguientes es irresoluble:

(a) Para una máquina de Turing M con un alfabeto de entrada Σ, ¿es $L(M) = \Sigma^*$?

(b) Para las máquinas de Turing M_1 y M_2 arbitrarias ¿son $L(M_1) = L(M_2)$?

(c) Para una máquina de Turing M arbitraria con el alfabeto de cinta Γ y $a \in \Gamma$, si M comienza con la cinta en blanco, ¿escribirá el símbolo a en la cinta alguna vez?

6.2 EL PROBLEMA DE CORRESPONDENCIA DE POST

Los problemas irresolubles que hemos visto hasta ahora atañen a las propiedades de las máquinas de Turing. El hecho de que el problema de parada sea irresoluble también tiene consecuencias en otra áreas. En muchos casos, es difícil obtener dichas consecuencias a partir del problema de parada. En esta sección vamos a obtener la irresolubilidad del problema de correspondencia de Post (al que nos referiremos como PCP).

Un caso del PCP se llama *sistema de correspondencia de Post* y está compuesto por tres elementos: un alfabeto Σ y dos conjuntos A y B, de cadenas de Σ^+, donde ambos tienen el mismo número de cadenas. Supongamos que $A = \{u_1, u_2, ..., u_k\}$ y $B = \{v_1, v_2, ..., v_k\}$. Una *solución* para este caso (es decir, una solución para el problema de correspondencia de Post) es una secuencia de índices $i_1, i_2, ..., i_n$, para los cuales $u_{i_1}u_{i_2} \ldots u_{i_n} = v_{i_1}v_{i_2} \ldots v_{i_n}$.

Por ejemplo, si $\Sigma = \{a, b\}$, $A = \{a, abaaa, ab\}$ y $B = \{aaa, ab, b\}$, la solución a este sistema de correspondencia de Post viene dada mediante $i_1 = 2$, $i_2 = i_3 = 1$, e $i_4 = 3$ ya que $u_2u_1u_1u_3 = abaaaaaab = v_2v_1v_1v_3$.

Conviene interpretar el sistema de correspondencia de Post como una colección de bloques de la forma

u_i
v_i

Por tanto, el sistema de correspondencia visto anteriormente será

a
aaa

$i = 1$,

$abaaa$
ab

$i = 2$,

ab
b

$i = 3$

Una solución se corresponde con la forma en la que se colocan los bloques uno al lado del otro, de manera que la cadena formada con las celdas superiores se corresponda con la cadena formada con las celdas inferiores. Por tanto, la solución anterior se representa

$abaaa$	a	a	ab
ab	aaa	aaa	b
$i_1 = 2$	$i_2 = 1$	$i_3 = 1$	$i_4 = 3$

Consideremos el sistema de correspondencia de Post dado por

ab	baa	aba
aba	aa	baa
$i = 1$	$i = 2$	$i = 3$

Obsérvese que cualquiera de las soluciones debe empezar con $i_1 = 1$, ya que éste es el único bloque donde ambas cadenas empiezan por la misma letra. El siguiente bloque para esta solución debe comenzar con una a en la celda superior; de aquí que o bien $i_2 = 1$ o $i_2 = 3$. Pero $i_2 = 1$ no sirve ya que obtendríamos

ab	ab
aba	aba
$i_1 = 1$	$i_2 = 1$

donde las cadenas superior e inferior no pueden ser iguales. Por tanto, i_2 debe ser 3, obteniéndose

ab	aba
aba	baa
$i_1 = 1$	$i_2 = 3$

De forma similar se puede demostrar que i_3 debe ser 3, obteniéndose

ab	aba	aba
aba	baa	baa
$i_1 = 1$	$i_2 = 3$	$i_3 = 3$

Pero este razonamiento es infinito, ya que nunca podremos elegir un índice que consiga que la cadena superior llegue a tener la misma longitud que la inferior. Luego este sistema de correspondencia de Post no tiene solución.

El problema de determinar si un sistema de correspondencia de Post arbitrario tiene solución es el *problema de correspondencia de Post* (PCP).

En los ejemplos precedentes, fue posible obtener un razonamiento (o una construcción especifica) que resolviera cada uno de los casos de PCP existentes. En general, pensamos que no hay ningún algoritmo que decida si un sistema de correspondencia de Post tiene solución. Probaremos esto, demostrando que si

PCP fuera resoluble entonces se podría resolver el problema de parada para las máquinas de Turing; es decir, el problema de parada para las máquinas de Turing se reduciría al PCP. Primero modificaremos el PCP y demostraremos que si el PCP fuera resoluble entonces el PCP *modificado* también lo sería.

En el PCP *modificado* (PCPM), buscaremos una solución para el sistema de correspondencia de Post en la cual la secuencia de índices debe comenzar por 1 (es decir, se debe tener que $i_1 = 1$). De esta forma, buscaremos una secuencia de índices $1, i_2, ..., i_n$ para la cual $u_1 u_{i_2} \ldots u_{i_n} = v_1 v_{i_2} \ldots v_{i_n}$. Primero demostraremos la conexión necesaria entre PCPM y PCP.

Lema 6.2.1. Si el PCP es resoluble, entonces el PCPM también es resoluble.

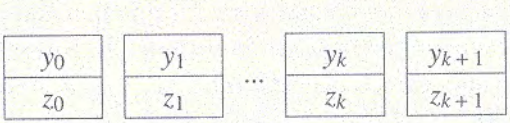
Demostración. Sean $A = \{u_1, u_2, ..., u_k\}$ y $B = \{v_1, v_2, ..., v_k\}$ una muestra del PCPM con el alfabeto Σ. Supongamos que cada $u_i = a_{i_1} a_{i_2} \ldots a_{i_{m_i}}$ y cada $v_i = b_{i_1} b_{i_2} \ldots b_{i_{n_i}}$, donde las *aes* y las *bes* son símbolos del alfabeto Σ. Para cada i, sea

$$y_i = a_{i_1} \$ a_{i_2} \$ \ldots \$ a_{i_{m_i}} \$ \quad \text{y} \quad z_i = \$ b_{i_1} \$ b_{i_2} \ldots \$ b_{i_{n_i}}$$

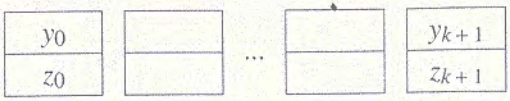
donde \$ es un nuevo símbolo que no está en Σ. Obsérvese que y_i es justamente u_i añadiendo \$ a cada uno de sus símbolos, y z_i es v_i con \$ como prefijo de cada uno de sus símbolos. Sea % otro símbolo que no pertenece a Σ y sea

$$y_0 = \$ y_1, \qquad z_0 = z_1$$
$$y_{k+1} = \%, \qquad z_{x+1} = \$\%$$

Consideremos un ejemplo del PCP dado por el siguiente conjunto de bloques:



Podemos afirmar que esta muestra del PCP tiene una solución si y solo si el original de PCPM dado por A y B tiene solución. Para probar esto, obsérvese que, si $i_1, i_2, ..., i_r$ proporciona una solución para esta muestra de PCP, entonces ya que todos los z_i empiezan por \$ y sólo y_0 empieza por \$, debemos considerar que $i_1 = 0$. Es más, i_r debe ser $k + 1$ puesto que sólo y_{k+1} y z_{k+1} coinciden en su último símbolo. Por tanto, si hay una solución para esta muestra de PCP, debe estar formada por $0, i_2, i_3, ..., k + 1$. Es decir, debemos tener



o, de forma equivalente,

$$\$a_{1_1}\,\$a_{1_2}\,\$ \ldots \$a_{1_{m_1}}\,\$ \ldots \$\% = \$b_{1_1}\,\$b_{1_2}\,\$ \ldots \$b_{1_n}\,\$ \ldots \$\%$$

Si se ignoran los signos \$, se obtiene

$$u_1 u_{i_2} \ldots u_{i_r} = v_1 v_{i_2} \ldots v_{i_r}$$

lo que es una solución de la muestra de PCPM anterior. Por tanto, si el PCP es resoluble, entonces el PCPM es resoluble. □

El Lema 6.2.1 nos da la capacidad de demostrar que un PCP es irresoluble mediante la demostración de que el PCPM es irresoluble. Esto simplifica nuestro problema puesto que el PCPM es más estructurado. Demostraremos que si el PCPM fuera resoluble, entonces el problema de parada para máquinas de Turing también lo sería, y, por tanto, el PCPM no puede ser resoluble.

Supongamos que el PCPM es resoluble, es decir, que existe un algoritmo general que se puede usar para determinar si cualquier muestra del PCPM tiene solución. Entonces, nuestro objetivo es demostrar que hay un algoritmo que puede determinar si una máquina de Turing M arbitraria, parará cuando empiece con una cadena arbitraria w sobre su cinta.

Para realizarlo, observaremos que cualquier máquina de Turing puede convertirse en una máquina que sólo para en un estado de aceptación. Es fácil de hacer, puesto que si la máquina de Turing para en un estado que no es de aceptación, se le pueden añadir transiciones que provoquen que la máquina entre en un bucle infinito. Obsérvese que el lenguaje aceptado por la máquina de Turing transformada es el mismo que el que acepta la máquina de Turing original. Luego vamos a suponer que cualquier máquina de Turing es de esta forma.

Ahora supongamos que $M = (Q, \Sigma, \Gamma, s, b, F, \delta)$ es una máquina de Turing y w es una cadena sobre Σ. Vamos a demostrar cómo construir una muestra del PCPM para la que la capacidad de determinar si existe una solución, implique que M tenga la capacidad de parar sobre la entrada w [y, por tanto, determinar si $w \in L(M)$].

Sea \$ un símbolo que no pertenece a Γ. Construiremos las listas $A = \{u_1, u_2, ..., u_n\}$ y $B = \{v_1, v_2, ..., v_n\}$ del PCPM mediante cinco grupos. Para simplificar, representaremos dichas listas como bloques.

El primer grupo estará formado por un único bloque

$$\begin{array}{|c|}\hline u_1 \\ \hline v_1 \\ \hline \end{array} = \begin{array}{|c|}\hline \$ \\ \hline \$q_1 w \$ \\ \hline \end{array}$$

en el que q_1 es el estado inicial de M.

El segundo grupo de bloques está formado por el bloque

$
$

y todos los bloques de la forma

σ
σ

donde σ es un símbolo no blanco de Γ.

El tercer grupo se deriva de las transiciones de M. Si $\delta(q, \sigma) = (p, \tau, R)$, añadimos el bloque

$q\sigma$
τp

Si $\delta(q, \sigma) = (p, \tau, L)$, incluimos el bloque de la forma

$\gamma q\sigma$
$p\gamma\tau$

para $\gamma \in \Gamma - \{b\}$. Si $\delta(q, b) = (p, \tau, R)$, añadimos el bloque

$q\$$
$\tau p\$$

y si $\delta(q, b) = (p, \tau, L)$, añadimos todos los bloques de la forma

$\gamma q\$$
$p\gamma\tau\$$

para toda $\gamma \in \Gamma - \{b\}$.

El cuarto grupo se deriva del conjunto de estados de aceptación de M. Para cada $q \in F$ y para todas σ y τ pertenecientes a $\Gamma - \{b\}$, añadimos los bloques

$\sigma q\tau$
q

,

$\sigma q\$$
$q\$$

,

$\$q\tau$
$\$q$

, y

$q\$\$$
$\$$

Ejemplo 6.2.1

Supongamos que M tiene el estado inicial q_1 y el estado de aceptación q_3 con las transiciones dadas por la tabla

$\delta(q_i, \sigma)$	$\sigma = a$	$\sigma = b$	$\sigma = b$
q_1	(q_2, b, R)	(q_2, a, L)	(q_2, b, L)
q_2	(q_3, a, L)	(q_1, a, R)	(q_2, a, R)

Sea $w = ab$. La muestra del PCPM dada por la máquina de Turing y esta cadena estaría representada por los siguientes bloques:

Grupo 1:

$\$$
$\$q_1ab\$$

$i = 1$

Grupo 2:

a
a

$i = 1$

b
b

$i = 3$

$\$$
$\$$

$i = 4$

Grupo 3:

q_1a
bq_2

$i = 5$ para $\delta(q_1, a) = (q_2, b, R)$

aq_1b
q_2aa

$i = 6$

bq_1b
q_2ba

$i = 7$ para $\delta(q_1, b) = (q_2, a, L)$

$aq_1\$$
$q_2ab\$$

$i = 8$

$bq_1\$$
$q_2bb\$$

$i = 9$ para $\delta(q_1, b) = (q_2, b, L)$

aq_2a
q_3aa

$i = 10$

bq_2a
q_3ba

$i = 11$ para $\delta(q_2, a) = (q_3, a, L)$

q_2b
aq_1

$i = 12$ para $\delta(q_2, b) = (q_1, a, R)$

$q_2\$$
$aq_2\$$

$i = 13$ para $\delta(q_2, b) = (q_2, a, R)$

Grupo 4:

aq_3a
q_3

$i = 14$

aq_3b
q_3

$i = 15$

bq_3a
q_3

$i = 16$

bq_3b
q_3

$i = 17$

$aq_3\$$
$q_3\$$

$i = 18$

$bq_3\$$
$q_3\$$

$i = 19$

$\$q_3a$
$\$q_3$

$i = 20$

$\$q_3b$
$\$q_3$

$i = 21$

$q_3\$\$$
$\$$

$i = 22$

Supongamos que tratamos de encontrar una solución para dicha muestra de PCPM. Debemos empezar con

u_1
v_1

Esto deja un "resto" de $q_1ab\$$ en los v_i que se debe ajustar a los u_i. Eligiendo $u_5u_3u_4$, compensamos dicho resto e introducimos un nuevo resto de $bq_2b\$$. Este resto puede ser cubierto mediante $u_3u_{12}u_4$, lo que introduce un nuevo resto $baq_1\$$. Si seguimos realizando composiciones y generando restos (realizando en algún caso un retroceso cuando se realiza un mala elección entre todas las opciones disponibles), se obtiene la solución:

	$\$$	q_1a	b	$\$$	b	q_2b	$\$$	b	$aq_1\$$	bq_2a	b	$\$q_3b$	a	b	$\$q_3q$	b	$\$q_3b$	$\$$	$q_3\$\$$
	$\$q_1ab\$$	bq_2	b	$\$$	b	aq_1	$\$$	b	$q_2ab\$$	q_3ba	b	$\$q_3$	a	b	$\$q_3$	b	$\$q_3$	$\$$	$\$$
Índice	1	5	3	4	3	12	4	3	8	11	3	21	2	3	20	3	21	4	22

La cadena formada mediante esta solución es:

$$\$q_1ab\$bq_2b\$baq_1\$bq_2ab\$q_3bab\$q_3ab\$q_3b\$q_3\$\$$$

Obsérvese $ab \in L(M)$, por lo que será aceptada mediante la computación

$$q_1ab \vdash bq_2b \vdash baq_1 \vdash bq_2ab \vdash q_3bab$$

Las configuraciones individuales de esta computación se separan mediante los signos \$ y en el orden en que aparecen en la cadena formada en la solución del PCPM.

Si repasamos la forma en la que se definieron las u_i y las v_i en la construcción de la muestra del PCPM para las máquinas de Turing M, nos vemos obligados a elegir el bloque

$$\begin{array}{|c|}\hline u_1 \\ \hline v_1 \\ \hline \end{array} = \begin{array}{|c|}\hline \$ \\ \hline \$q_1w\$ \\ \hline \end{array}$$

para empezar una solución (si es que existe alguna). La parte q_1w que aparece entre los \$ de v_1 representa la configuración inicial de M como comienzo de la computación de w. Entonces, si w empieza por σ y si $\delta(q_1, \sigma) = (q_k, \tau, R)$, construimos el par (u_i, v_i) como

$$\begin{array}{|c|}\hline q_1\sigma \\ \hline \tau q_k \\ \hline \end{array}$$

de forma que, para compensar el resto que le falta a la cadena del cuadrado superior, debemos construir la siguiente configuración de M en el cuadro inferior. Entonces el resto se rellena añadiendo los símbolos restantes que faltan de w en las cadenas superior e inferior. Por tanto, añadiremos todos los pares de la forma

$$\begin{array}{|c|}\hline \sigma_i \\ \hline \sigma_i \\ \hline \end{array} \text{ y } \begin{array}{|c|}\hline \$ \\ \hline \$ \\ \hline \end{array}$$

Cuando en los cuadrados superiores se consigue la cadena que hay en el cuadrado inferior de la primera caja, la cadena inferior debe crecer para representar la siguiente configuración de M. Extenderemos la cadena de la misma forma. En cada etapa, la cadena inferior va un paso por delante de la cadena superior, por tanto, cada vez que la cadena superior completa una configuración de M, la cadena inferior tiene que construir la siguiente configuración. Cada uno de los restos es necesario para incluir los pares (u_i, v_i) apropiados, de forma que la cadena superior concuerde con la inferior una vez que se llega a una configuración M que sea de aceptación.

Primero obsérvese que, si la cadena de las u_i es de la forma $\alpha\$$ y la cadena de las v_i es de la forma $\alpha\$x\$$, entonces podemos ampliar las u_i a $\alpha\$x\$$ y las v_i a $\alpha\$x\$y\$$, donde y representa la configuración de M un movimiento después. Además, la cadena de las v_i es la única cadena que se puede corresponder con la nueva cadena de las u_i.

Para probarlo, supongamos que la cadena de las u_i es $\alpha\$$ mientras que la cadena de las v_i es $\alpha\$\sigma_1\sigma 2 \ldots \sigma_k q\sigma_{k+1} \ldots \sigma_{k+m}\$$ para algún $m > 0$. Además, supongamos que $\delta(q, \sigma_{k+1}) = (q', \tau, R)$. Las otras posibilidades son similares. Los bloques que nos permiten ampliar la cadena de las u_i son

$$\boxed{\begin{array}{|c|}\hline \sigma_i \\ \hline \sigma_i \\ \hline \end{array}}, \quad \text{para } i = 1, 2, \ldots, k$$

el bloque

$$\boxed{\begin{array}{|c|}\hline q\sigma_{k+1} \\ \hline \tau p \\ \hline \end{array}}$$

bloques de la forma

$$\boxed{\begin{array}{|c|}\hline \sigma_i \\ \hline \sigma_i \\ \hline \end{array}}, \quad \text{para } i = k+2, \ldots, k+m$$

y finalmente el bloque

$$\boxed{\begin{array}{|c|}\hline \$ \\ \hline \$ \\ \hline \end{array}}$$

La cadena que se obtiene en las v_i es

$$\alpha\$\sigma_1 \ldots \sigma_k q\sigma_{k+1} \ldots \sigma_{k+m}\$\sigma_1 \ldots \sigma_k \tau p\sigma_{k+2} \ldots \sigma_{k+m}\$$$

Obsérvese que $y = \sigma_1 \ldots \sigma_k \tau p\sigma_{k+2} \ldots \sigma_{k+m}$ es exactamente la configuración de M que resulta de esta transición. Es más, puesto que en este proceso no hay ningún par (u, v) que se pueda elegir, no hay otra cadena de v_i que se pueda obtener.

Recuérdese que hemos supuesto que una máquina de Turing M para sólo cuando acepta una cadena de entrada.

Lema 6.2.2. M para sobre la entrada w si y sólo si hay una solución de la muestra derivada de PCPM.

Demostración. Supongamos que M para sobre la entrada w. En ese caso hay una secuencia de configuraciones de M que comienza con $q_1 w$ y termina en un estado

de aceptación. Supongamos que dicha secuencia se representa mediante las cadenas $x_1, x_2, ..., x_k$ para algún k. Según lo visto anteriormente, podemos construir una secuencia de u_i de la forma $\$x_1\$x_2 \ldots \$x_{k-1}\$$ y una secuencia de v_i de la forma $\$x_1\$x_2 \ldots \$x_{k-1}\$x_k\$$. Ya que M para en la configuración x_k, debemos tener que $x_k = yqz$ para algún estado $q \in F$ y las cadenas y y z pertenecientes a Γ^*. Si al menos una de las dos cadenas no es una cadena vacía, podemos ampliar la cadena u_i y la cadena v_i por medio de uno de los pares (u, v) del cuarto grupo y posiblemente distintos de

σ_i
σ_i

o

$
$

lo que hace que la cadena de las u_i sea como $\$x_1\$x_2\$ \ldots \$x_{k-1}\$x_k\$$ y la cadena de las v_i como $\$x_1\$x_2\$ \ldots \$x_{k-1}\$x_k\$x_{k'}$ donde $x_{k'}$ tiene al menos un símbolo menos que x_k. Repetimos esto, reduciendo la cadena que se encuentra entre el último par de $ de la v_i hasta obtener cadenas de la forma:

$$u_i: \ \$x_1\$x_2\$ \ldots \$x_k\$ \ldots \$$$
$$v_i: \ \$x_1\$x_2\$ \ldots \$x_k\$x_{k'}\$ \ldots \$q\$$$

El último grupo de pares (u, v) contiene un bloque de la forma

$q$$$
$

el cual se puede añadir para obtener cadenas iguales. Por tanto, si M para sobre la entrada w, podemos obtener una solución para la muestra derivada del PCPM.

A la inversa, supongamos que M no para sobre la entrada w. Por lo anterior, las cadenas de las u_i y las v_i de la muestra derivada del PCPM representan configuraciones de M consecutivas. Ya que M nunca pasa a un estado de parada (todos los cuales son de aceptación), no se puede añadir ninguno de los pares (u_i, v_i) pertenecientes al cuarto grupo. Por inducción se demuestra que el desarrollo de las cadenas u_i y v_i siempre tiene un número distinto de signos $ y, por tanto, esta muestra del PCPM no tiene solución. □

Entonces podemos deducir el siguiente teorema:

Teorema 6.2.3. El PCP no es resoluble.

Demostración. Sea M una máquina de Turing arbitraria y w una cadena arbitraria (suponemos que M para sólo cuando acepta una cadena). Si el PCP es resoluble, en-

tonces por el Lema 6.2.1 podemos determinar si tiene solución la muestra del PCPM derivada de M. Por tanto, por el Lema 6.2.2 se puede determinar si M para sobre la entrada w. Ya que M era una máquina de Turing arbitraria y w una cadena arbitraria, existe un algoritmo para resolver una muestra arbitraria del problema de parada, lo que contradice el Teorema 6.1.1. Por tanto el PCP no es resoluble. □

Ejercicios de la Sección 6.2

6.2.1. El sistema de correspondencia de Post representado por

aaa	baa
aa	$abaaa$
$i=1$	$i=2$

tiene solución. Encontrarla.

6.2.2. ¿Tiene solución este sistema de correspondencia de Post?

ab	bba	aba
aba	aa	bab
$i=1$	$i=2$	$i=3$

6.2.3. Para cada uno de los siguientes sistemas de correspondencia de Post, obtener una solución o demostrar que no existe.

(a)

a	bb	a
aa	b	bb
$i=1$	$i=2$	$i=3$

(b)

a	aab	$abaa$
aaa	b	ab
$i=1$	$i=2$	$i=3$

(c)

ab	ba	b	ba
a	bab	aa	ab
$i=1$	$i=2$	$i=3$	$i=4$

(d)

ab	baa	aba
aba	aa	baa
$i=1$	$i=2$	$i=3$

(e)

aa	bb	abb
aab	ba	b
$i=1$	$i=2$	$i=3$

(f)

ab	aa	ab	bb
bb	ba	abb	bab
$i=1$	$i=2$	$i=3$	$i=4$

6.2.4. Aunque el PCP es irresoluble, se puede modificar el problema fácilmente para que sea resoluble. Demostrar que hay un algoritmo de decisión para el PCP correspondiente al sistema de correspondencia de Post con un alfabeto con un único símbolo.

6.2.5. Demostrar que, en la demostración del Lema 6.2.1, si hay una solución para la muestra del PCPM dada por A y B, entonces la instancia derivada del PCP también tiene solución.

6.2.6. Según lo tratado después del Lema 6.2.1, se pretende que la conversión de una máquina de Turing M en una máquina que sólo para cuando acepta una cadena, sea una conversión fácil. De hecho, la conversión consiste en dos pasos.

(a) Obtener una técnica para identificar los estados que no son de aceptación en los cuales $M = (Q, \Sigma, \Gamma, s, b, F, \delta)$ pararía si entrara en ellos.

(b) Supongamos que q es un estado que no es de aceptación desde el cual no hay transiciones. Demostrar de qué forma se pueden construir transiciones que provocarían que M nunca parase en el caso de que entrara en q.

6.2.7. Para la máquina de Turing del Ejemplo 6.2.1 y la cadena $w = abb$, ¿tiene solución la muestra derivada del PCPM? ¿Qué ocurriría si $w = a$?

6.2.8. Un método alternativo para demostrar que el PCPM es irresoluble se obtiene a partir de las gramáticas no restringidas.

(a) Demostrar que el *problema de los elementos* de un lenguaje recursivamente enumerable es irresoluble. (El problema de los elementos de lenguajes recursivamente enumerables se puede enunciar como sigue: "¿Hay un algoritmo para decidir si $w \in L$ para un lenguaje L recursivamente enumerable y una cadena arbitraria w?")

(b) Sea $G = (N, \Sigma, S, P)$ una gramática no restringida y $w \in \Sigma^+$. Construir una muestra de PCPM de la siguiente manera: Sea $u_1 = F$ y $v_1 = FS \Rightarrow$, donde F es un símbolo que no pertenece a $N \cup \Sigma$. Por tanto, obtenemos el bloque

F
$FS \Rightarrow$

$i = 1$

Para cada $x \in N \cup \Sigma$, añadiremos el bloque

x
x

Sea E un símbolo que no pertenece a $N \cup \Sigma$, y añadimos el bloque

$\Rightarrow wE$
E

Para cada producción $\alpha \rightarrow \beta$ de P, añadir un bloque

α
β

Finalmente, añadir el bloque

$\Rightarrow$
$\Rightarrow$

i. Para la siguiente gramática no restringida y la cadena a^3b^3, construir una muestra del PCPM:

$$S \rightarrow aASB \mid aBb$$
$$aA \rightarrow aa$$
$$aB \rightarrow ab$$
$$bB \rightarrow bb$$

ii. Obtener una derivación de a^3b^3 mediante esta gramática.

iii. Obtener una solución para la muestra del PCPM que hemos construido.

(c) Esbozar una demostración de la siguiente sentencia: Si G es una gramática no restringida y $w \in \Sigma^+$ es una cadena, la muestra del PCPM construida como en la parte (b) tiene solución si y sólo si $w \in L(G)$.

(d) Probar que si el PCPM es resoluble entonces el problema de los elementos de un lenguaje recursivamente enumerable es también resoluble (demostrando que el PCPM es irresoluble).

6.2.9. Demostrar que el PCP sigue siendo irresoluble incluso si los sistemas de correspondencia de Post se restringen sobre alfabetos de dos símbolos.

6.2.10. Usar el PCP para demostrar que el problema de la vacuidad para las gramáticas sensibles al contexto es irresoluble. [El problema de la vacuidad para gramáticas sensibles al contexto es, "Para una GSC arbitraria G, ¿$L(G) = \emptyset$?"]. *Indicación:* Una consecuencia del Problema 5.6 es que, para cualquier autómata linealmente acotado M, podemos construir una gramática sensible al contexto que genere $L(M)$. Queremos demostrar que PCP es resoluble si el problema de la vacuidad para gramáticas sensibles al contexto es resoluble, para ello consideraremos un ALA que acepte sólo las cadenas y para las cuales $u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k} = y$.

6.3 IRRESOLUBILIDAD Y LENGUAJES INDEPENDIENTES DEL CONTEXTO

En el Capítulo 3 mostramos algoritmos para algunos problemas de decisión para gramáticas y lenguajes independientes del contexto. En particular, obtuvimos una solución para el problema de los elementos de lenguajes independientes del contexto, para el problema de si una gramática independiente del contexto genera alguna cadena de terminales (el problema de la vacuidad), y para el problema de si una gramática independiente del contexto genera o no, un lenguaje infinito. Además dedujimos que muchas de las preguntas que se pueden responder para lenguajes regulares se convierten en preguntas irresolubles para gramáticas independientes del contexto. La irresolubilidad del PCP proporciona una herramienta útil para demostrar la irresolubilidad de algunos problemas.

Nuestro primer planteamiento para demostrar que una pregunta acerca de gramáticas o lenguajes independientes del contexto es irresoluble es demostrar que, si fuera decidible, entonces el PCP sería decidible. Para cada una de las preguntas demostraremos cómo se construye una gramática independiente del contexto a partir de un sistema de correspondencia de Post arbitrario, de forma que, si la pregunta acerca de las gramáticas fuera decidible, entonces una muestra arbitraria del PCP sería también decidible.

Supongamos que Σ, A y B es una muestra del PCP, arbitraria, donde A es $u_1, u_2, \ldots, u_k$ y B es $v_1, v_2, \ldots, v_k$, listas de cadenas sobre Σ. Supongamos que $C = \{a_1, a_2, \ldots, a_k\}$ son símbolos que no pertenecen a Σ. Vamos a construir dos gramáticas independientes del contexto G_A y G_B que se derivan de A, B y C.

Sea $G_A = (\{S_A\}, \Sigma \cup C, S_A, P_A)$ y $G_B = (\{S_B\}, \Sigma \cup C, S_B, P_B)$, donde S_A y S_B son símbolos nuevos. El conjunto de producciones P_A contiene todas las producciones de la forma $S_A \rightarrow u_i S_A a_i \mid u_i a_i$ para $i = 1, 2, \ldots, k$. Igualmente, P_B contiene todas las producciones de la forma $S_B \rightarrow v_i S_B a_i \mid v_i a_i$ para $i = 1, 2, \ldots, k$.

Obsérvese que el lenguaje generado por G_A se compone de todas las cadenas no vacías de la forma

$$u_{i_1}u_{i_2} \ldots u_{i_{n-1}}u_{i_n}a_{i_n}a_{i_{n-1}} \ldots a_{i_2}a_{i_1}$$

De forma similar, G_B genera todas las cadenas no vacías de la forma

$$v_{i_1}v_{i_2} \ldots v_{i_{n-1}}v_{i_n}a_{i_n}a_{i_{n-1}} \ldots a_{i_2}a_{i_1}$$

Si la muestra anterior del PCP tiene una solución, donde

$$u_{i_1}u_{i_2} \ldots u_{i_{n-1}}u_{i_n} = v_{i_1}v_{i_2} \ldots v_{i_{n-1}}v_{i_n}$$

entonces se obtiene que la cadena

$$u_{i_1}u_{i_2} \ldots u_{i_{n-1}}u_{i_n}a_{i_n}a_{i_{n-1}} \ldots a_{i_2}a_{i_1} \in L(G_A) \cap L(G_B)$$

A la inversa, si para una cadena w se tiene que $w \in L(G_A) \cap L(G_B)$, entonces w debe ser de la forma

$$w = w'a_{i_n}a_{i_{n-1}} \ldots a_{i_2}a_{i_1}$$

para un $w' \in \Sigma^+$ y un sufijo $a_{i_n}a_{i_{n-1}} \ldots a_{i_2}a_{i_1}$. Pero la única forma en que se puede generar este sufijo es generando el prefijo $u_{i_1}u_{i_2} \ldots u_{i_{n-1}}u_{i_n}$ de G_A y el prefijo $v_{i_1}v_{i_2} \ldots v_{i_{n-1}}v_{i_n}$ de G_B; por tanto

$$\begin{aligned} w &= u_{i_1}u_{i_2} \ldots u_{i_{n-1}}u_{i_n}a_{i_n}a_{i_{n-1}} \ldots a_{i_2}a_{i_1} \\ &= v_{i_1}v_{i_2} \ldots v_{i_{n-1}}v_{i_n}a_{i_n}a_{i_{n-1}} \ldots a_{i_2}a_{i_1} \end{aligned}$$

y entonces la cadena $u_{i_1}u_{i_2} \ldots u_{i_{n-1}}u_{i_n}$ es una solución del ejemplo del PCP. De lo que se deduce que dicha instancia del PCP tiene una solución si y sólo si $L(G_A) \cap L(G_B) \neq \emptyset$.

Teorema 6.3.1. El problema de la intersección vacía para las gramáticas independientes del contexto es irresoluble.

Demostración. Supongamos que este problema es resoluble. Entonces hay un algoritmo para decidir si $L(G_1) \cap L(G_2) = \emptyset$. Sean A, B y Σ una muestra del PCP. Por la construcción anterior, se derivan las dos gramáticas G_A y G_B. Usaremos el algoritmo para decidir si $L(G_A) \cap L(G_B) = \emptyset$. Esta muestra del PCP tiene solución si y sólo si $L(G_A) \cap L(G_B) \neq \emptyset$. Por tanto, podemos determinar si una muestra arbitraria del PCP es resoluble, lo que contradice el Teorema 6.2.3. □

Recuérdese que una gramática independiente del contexto es ambigua si hay dos derivaciones por la izquierda que son distintas para la misma cadena. La gramática independiente del contexto

$$S \to aSB \mid SS \mid \varepsilon$$
$$B \to b$$

es ambigua, ya que hay dos derivaciones por la izquierda de a^2b^2.

Por desgracia, en general no es posible determinar si una gramática independiente del contexto es ambigua. Es decir, la *cuestión de la ambigüedad de gramáticas independientes del contexto*, "¿es G una gramática independiente del contexto ambigua?", es irresoluble. Para probarlo, vamos a suponer que la pregunta es resoluble; es decir, que hay un algoritmo que decide sobre ella, para una gramática independiente del contexto. Demostraremos que el PCP debería ser resoluble.

Una vez más, sean Σ, A y B una muestra arbitraria del PCP y construyamos, como antes, las gramáticas G_A y G_B. Definimos una nueva gramática G basada en G_A y G_B. Sea $G = (\{S, S_A, S_B\}, \Sigma, S, P)$, en la cual el conjunto de producciones P viene dado mediante $P_A \cup P_B \cup \{S \to S_A, S \to S_B\}$. Obsérvese que G genera $L(G_A) \cup L(G_B)$. Es más, debido a la estructura de las producciones de G_A y G_B, todas las derivaciones de G son derivaciones por la izquierda.

Se ve fácilmente que las gramáticas G_A y G_B no son ambiguas. Por ejemplo, en $L(G_A)$ cualquier cadena que termina con a_i se debe derivar a partir de la producción $S_A \to u_i S_A a_i$. Igualmente podríamos decir qué producciones se aplicaron en etapas posteriores. Entonces se deduce que si G es una gramática independiente del contexto ambigua, entonces lo es porque hay una cadena w para la cual

$$S \Rightarrow S_A \overset{*}{\Rightarrow} w = u_{i_1} u_{i_2} \ldots u_{i_n} a_{i_n} \ldots a_{i_2} a_{i_1}$$

y también

$$S \Rightarrow S_B \overset{*}{\Rightarrow} w = v_{i_1} v_{i_2} \ldots v_{i_n} a_{i_n} \ldots a_{i_2} a_{i_1}$$

Es decir, $w \in L(G_A) \cap L(G_B)$. Pero $L(G_A) \cap L(G_B) \neq \emptyset$ es equivalente a tener una solución de esta muestra del PCP.

Por tanto, si hay un algoritmo para determinar si una gramática independiente del contexto es ambigua, entonces podemos determinar si existe una solución para una muestra arbitraria del PCP. Esto es una contradicción y por tanto podemos enunciar el siguiente teorema:

Teorema 6.3.2. El problema de la ambigüedad para gramáticas independientes del contexto es irresoluble.

Aunque el PCP es útil para demostrar la irresolubilidad de muchos problemas sobre lenguajes y gramáticas de contexto libre, también hay otros métodos. Este método implica el problema de la vacuidad para máquinas de Turing (véase Sección 6.2). Supongamos que $M = (Q, \Sigma, \Gamma, s = q_1, b, F, \delta)$ es una máquina de Turing y supongamos que $q_1 w \vdash x_1 \vdash \ldots \vdash x_n$ es una computación de M [con $w \in L(M)$]. Entonces la cadena $q_1 w \$ x_1^R \$ x_2 \$ x_3^R \$ \ldots$ se llamará *computación válida* (en la misma se supone que \$ es un símbolo que no pertenece a Γ). Obsérvese que cada configuración x_i es una cadena de $\Gamma^* Q \Gamma^*$ y que no empieza ni termina con b. Necesariamente x_n también es una cadena de $\Gamma^* F \Gamma^*$.

Obsérvese que toda computación válida de M se puede considerar como una cadena sobre $\Gamma \cup Q \cup \{\$\}$. Definimos el conjunto de las *computaciones no válidas* de M como el complemento del conjunto de las computaciones válidas con respecto a $\Gamma \cup Q \cup \{\$\}$.

Es lógico preguntarse cómo sería una cadena que consta de una computación no válida. Si la cadena w es una computación no válida, entonces w satisface una de las siguientes condiciones:

1. w no es de la forma $y_0 \$ y_1 \$ y_2 \$ \ldots \$ y_k \$$, donde y_i es una configuración de M si i es par o sino y_i^R es una configuración de M si i es impar.
2. y_0 no es de la forma $q_1 x$ para algún $x \in \Sigma^*$.
3. y_k no es una configuración de aceptación de M; es decir, $y_k \notin \Gamma^* F \Gamma^*$.
4. No se cumple que $y_i \vdash y_{i+1}^R$ para algún i par.
5. No se cumple que $y_i^R \vdash y_{i+1}$ para algún i impar.

El conjunto de las cadenas que satisfacen la condición 1 forma un lenguaje regular, al igual que los conjuntos que satisfacen las condiciones 2 y 3. Por tanto, las cadenas que satisfacen las condiciones 1 ó 2 ó 3 constituyen un lenguaje regular (Teorema 2.8.1). Por consiguiente, se puede construir un autómata finito para este lenguaje y entonces obtener una gramática regular (y, por lo tanto, independiente del contexto).

Los conjuntos de cadenas que satisfacen las condiciones 4 y 5 son lenguajes independientes del contexto. Para probarlo, construiremos un autómata de pila no determinista M' que acepte el lenguaje de la condición 4. M' selecciona de forma no determinista una i para la cual y_i va precedido por un número par de signos \$. Entonces analiza y_i y, cuando lo hace, introduce x en la pila, donde $y_i \vdash x$ está en M. Una vez que M' encuentra el \$ del final derecho de y_i, compara y_{i+1} con el contenido de la pila, estrayendo los símbolos que coinciden. Obsérvese que se obtendrá una coincidencia sólo si $y_{i+1} = x^R$. Si en algún momento un símbolo no coincide, M' analiza la cadena restante y la acepta. Obsérvese que

sería sencillo hacer que la pila también se vaciara cuando se hace una aceptación. La construcción para el lenguaje que cumple la condición 5 es similar.

Por tanto, el conjunto de las computaciones no válidas es la unión de dos lenguajes independientes del contexto y un lenguaje regular, y además es un lenguaje independiente del contexto. (Teorema 3.6.3). Es más, podemos construir una gramática independiente del contexto para este conjunto a partir de los dos ADPND y la gramática regular.

Entonces, hemos demostrado lo siguiente:

Lema 6.3.3. El conjunto de las computaciones no válidas de la máquina de Turing M es un lenguaje independiente del contexto.

Entonces se puede demostrar el Teorema 6.3.4.

Teorema 6.3.4. Para una gramática independiente del contexto arbitraria, la pregunta $¿L(G) = \Sigma^*?$ es irresoluble.

Demostración. Supongamos que este problema es resoluble. Demostraremos que el problema de la vacuidad para máquinas de Turing es resoluble, con lo que se contradice el Ejercicio 6.1.4.

Sea $M = (Q, \Sigma, \Gamma, q_1, b, F, \delta)$ una máquina de Turing arbitraria. Por el Lema 3.3, podemos construir una gramática independiente del contexto G que genere todas las computaciones no válidas de M. Obsérvese que $L(M) = \emptyset$ si y sólo si $L(G) = \Sigma^*$. Por tanto, si esta pregunta fuera decidible, entonces la pregunta de la vacuidad también lo sería. □

A partir del Teorema 6.3.4 se deducen varios resultados. Por ejemplo, si G_1 es una gramática independiente del contexto y G_2 es una gramática que genera Σ^* (donde Σ es el alfabeto de terminales de G_1), entonces la pregunta "$¿L(G_1) = L(G_2)?$" es equivalente a la del teorema. Por tanto, no se puede decidir si $L(G_1) = L(G_2)$ para dos gramáticas G_1 y G_2 arbitrarias, independientes del contexto.

Si R es un lenguaje regular arbitrario y G_1 es una gramática independiente del contexto, la sentencia "¿Es $L(G_1) = R$?" no es resoluble. Para demostrarlo, consideramos $R = \Sigma^*$, donde Σ es el alfabeto de los terminales de G_1. Una vez más se puede observar que "¿Es $L(G_1) = R$?" es equivalente a la sentencia del Teorema 6.3.4.

Ejercicios de la Sección 6.3

6.3.1. ¿Es decidible el siguiente problema? Sea G_1 una gramática regular arbitraria y G_2 una gramática independiente del contexto arbitraria. $¿L(G_1) \cap L(G_2) = \emptyset?$

6.3.2. Los lenguajes $L(G_A)$ y $L(G_B)$ tienen una propiedad interesante. Si $L(G_A) \cap L(G_B)$ es regular, entonces es vacía. Demostrar esta propiedad. Para ello utilice el hecho de que para una gramática independiente del contexto G, el problema "¿$L(G)$ es regular?" es irresoluble.

6.3.3. Supóngase que no hay ninguna computación válida de M. ¿En qué afecta esto a $L(M)$?

6.3.4. Construir expresiones regulares para cada uno de los lenguajes de las condiciones 1, 2 y 3, vistos antes del Lema 6.3.3.

6.3.5. Demostrar que $L(G_1) \subseteq L(G_2)$ es un problema irresoluble para las gramáticas independientes del contexto G_1 y G_2.

6.3.6. Demostrar que es irresoluble que $R \subseteq L(G)$ para un lenguaje regular R arbitrario y una gramática independiente del contexto G.

6.3.7. Para una gramática G, independiente del contexto, arbitraria y para un lenguaje R regular arbitrario, la pregunta "¿Está $L(G) \subseteq R$?" ¿es una pregunta decidible? [*Indicación:* $L(G) \subseteq R$ si y sólo si $L(G) \cap \overline{R} = \emptyset$ ¿Qué sabemos sobre el lenguaje $L(G) \cap \overline{R}$? ¿Qué sabemos sobre la resolubilidad del problema de la vacuidad para $L(G) \cap \overline{R}$?

PROBLEMAS

6.1. En este problema vamos a usar una técnica distinta para llegar a la conclusión del Teorema 6.3.1, es decir, el problema de la intersección vacía para las gramáticas independientes del contexto es irresoluble.

Sea $M = (Q, \Sigma, \Gamma, s = q_1, b, F, \delta)$ una máquina de Turing para la cual el símbolo $\$ \notin F$. Recuérdese que una manera de representar configuraciones de una máquina de Turing consiste en hacerlo mediante las cadenas de la forma $\sigma_1\sigma_2 \ldots \sigma_k q \sigma_{k+1} \ldots \sigma_n$, donde los $\sigma_i \in \Gamma$ y $q \in Q$. Es decir, una configuración es una cadena sobre $\Gamma^* Q \Gamma^*$. Sean

$$A = \{y\$z^R \mid y, z \in \Gamma^* Q \Gamma^* \text{ e } y \vdash z \text{ está en } M\}$$
$$B = \{y^R \$z \mid y, z \in \Gamma^* Q \Gamma^* \text{ e } y \vdash z \text{ está en } M\}$$

1. Demostrar que A y B son lenguajes independientes del contexto. [*Indicación:* Esbozar un ADPND que acepte A (o B) y aplicar el Teorema 3.8.2.]
2. Demostrar que los lenguajes L_1 y L_2 son lenguajes independientes del contexto, donde
$$L_1 = (A\$)^* (\varepsilon \cup \Gamma^* F \Gamma^* \$)$$
$$L_2 = q_1\Sigma^* \$ (B\$)^* (\varepsilon \cup \Gamma^* F \Gamma^* \$)$$
3. Demostrar que $L_1 \cap L_2$ es exactamente el conjunto de la computaciones válidas de M.

4. Supóngase que el problema de la intersección vacía para gramáticas independientes del contexto fuera resoluble. Construir un algoritmo que responda al problema ¿Es vacío $L(M)$ para una máquina de Turing M arbitraria? Deducir que el problema de la intersección vacía para lenguajes independientes del contexto no es resoluble.

6.2. 1. Sea M una máquina de Turing que, para cada cadena de entrada, realiza al menos tres movimientos. (Obsérvese que cualquier máquina de Turing puede transformarse en una de este tipo). Demostrar que el conjunto de las computaciones válidas de M es un lenguaje independiente del contexto si y sólo si $L(M)$ es finito. [*Indicación:* Si $L(M)$ es finito, entonces el conjunto de las computaciones válidas también es finito, con lo que quedaría probado. Por otro lado, si $L(M)$ es infinito y las computaciones válidas forman un lenguaje independiente del contexto, entonces hay una computación válida de la forma $w_1\$w_2\$w_3\$ \ldots$ con $|w_2|$ de un tamaño para el que se pueda aplicar el lema de Ogden (véase Problemas 3.1 y 3.2).]

2. Demostrar que para una gramática G arbitraria, independiente del contexto, el hecho de saber si $\overline{L(G)}$ es un lenguaje independiente de contexto, es irresoluble.

3. Demostrar que es irresoluble saber si $L(G_1) \cap L(G_2)$ es un lenguaje independiente del contexto, para las gramáticas independientes del contexto G_1 y G_2.

7

Introducción a la complejidad computacional

En la Sección 4.4 demostramos que las distintas definiciones de una máquina de Turing no incrementaban la potencia computacional del modelo básico. Hicimos esto para demostrar que una máquina de Turing de un tipo puede simular una máquina de Turing de otro tipo.

Cuando se simulan las acciones de una máquina de Turing de un tipo más complejo mediante una que corresponde a nuestra definición básica, es habitual que se consuma la mayoría del espacio y del tiempo. El modelo de máquina de Turing que se use no afecta a la potencia computacional o a la capacidad para decidir un problema. Sin embargo, los requerimientos de espacio y tiempo para una computación se ven claramente afectados por el modelo elegido. La complejidad de una computación mide los requerimientos de la misma en cuanto a espacio y tiempo. En este capítulo haremos una breve introducción de la *teoría de la complejidad computacional* con respecto a los autómatas y lenguajes.

7.1 COMPLEJIDAD ESPACIAL

La complejidad de una computación se mide por la cantidad de espacio y tiempo que consume. Las computaciones eficientes tienen unas exigencias de recursos pequeñas (considerando "pequeñas" de una manera relativa). Los recursos que necesita una computación que acepta cadenas de algún lenguaje, puede depender

del tamaño (longitud) de la cadena de entrada. Empezaremos por considerar los recursos espaciales.

Definición 7.1.1. Sea M una máquina de Turing con k cintas. Supongamos que, sobre cualquier entrada de longitud n, la cabeza de lectura/escritura de M consulta al menos $S(n)$ celdas de cualquiera de las cintas, donde S: $\mathbb{N} \rightarrow \mathbb{N}$ es una función. Entonces se dice que M tiene una *complejidad espacial* $S(n)$ o que es una máquina de Turing espacialmente acotada por $S(n)$. También se dice que $L(M)$ es un lenguaje con complejidad espacial $S(n)$.

Ejemplo 7.1.1

Consideremos el lenguaje

$$L = \{xyzy^I \mid x, y \in \Sigma^*, z \in \Sigma^+\}$$

Una máquina de Turing no determinista con dos cintas, que acepta el lenguaje L tiene la entrada w sobre la cinta 1. Cuando recorre w, averigua (de forma no determinista) dónde empieza y, lo copia en la cinta 2. Continúa el recorrido después de copiar y, ignorando z, y entonces debe averiguar (de forma no determinista) donde empieza y^R. Mientras recorre y^R, compara los símbolos de la cinta 1 con los de la cinta 2. Suponiendo que la máquina de Turing comienza con la cabeza de la cinta 1 situada sobre el primer símbolo de la entrada, obsérvese que se recorre w en su totalidad, al igual que los b que siguen a w. Es más, en la cinta 2 se analizan tanto y como los blancos anteriores y posteriores. Por tanto, una cota espacial para esta máquina de Turing es

$$S(n) = \max\{n+1, n+2\} = n+2$$

Obsérvese que, cualquier S': $\mathbb{N} \rightarrow \mathbb{N}$ para la cual $S(n) \leq S'(n)$ también es una cota espacial.

Ya que una máquina de Turing recorre al menos una celda de cada cinta, la cota espacial debe ser al menos 1. Para tratar la complejidad espacial, a veces conviene suponer una máquina de Turing que tenga una única cinta de lectura y una o más cintas de trabajo. Vamos a modificar la definición de complejidad espacial para que signifique que las cabezas de lectura/escritura de las cintas de trabajo recorran como máximo $S(n)$ celdas. Debido a esto, no contamos las celdas recorridas sino que simplemente analizamos la cadena de entrada. Por lo tanto en algunas circunstancias puede darse el hecho de que la complejidad espacial sea menor que la lineal.

Recuérdese que en la Sección 4.4 simulamos una máquina de Turing de k cintas mediante una máquina de Turing de una cinta con $2k+1$ pistas. Luego

podemos usar la misma simulación para nuestra máquina de Turing con k cintas de trabajo y una cinta de entrada. Es más, si $S(n)$ es una cota espacial para la máquina de Turing con k cintas, la simulación no usará más de $S(n)$ celdas de las $2k+1$ pistas de sus cintas de trabajo, ya que $S(n)$ también es una cota espacial para la simulación. Por tanto tenemos el siguiente teorema:

Teorema 7.1.2. Si una máquina de Turing con k cintas de trabajo y cuya cota espacial es $S(n)$ acepta el lenguaje L, entonces una máquina de Turing con una cinta de trabajo y cota espacial $S(n)$ también lo acepta.

El Teorema 7.1.2 dice que el número de cintas de trabajo usadas al aceptar un lenguaje, no afecta a la complejidad espacial de L. De hecho, podemos comprimir mediante un factor constante la cantidad de espacio de cinta que se usa al aceptar un lenguaje, codificando distintos símbolos de la cinta en uno. Por ejemplo, si el alfabeto de la cinta está formado por a, b y b, podríamos considerar una codificación de pares de símbolos. Esto podría aumentar el tamaño del alfabeto de cinta, pero también reduciría el espacio de cinta a la mitad.

Teorema 7.1.3. Sea L un lenguaje aceptado por una máquina de Turing M con k cintas de trabajo, cuya cota espacial es $S(n)$. Para todo $c > 0$, hay una máquina de Turing espacialmente acotada por $cS(n)$ que acepta L.

Demostración. (*esbozo*) Sea r un entero que satisface que $rc \geq 1$. Codificaremos el alfabeto de la cinta de M como bloques de r símbolos. Usaremos las transiciones de M para definir las nuevas transiciones basadas en los bloques de r símbolos. □

Como resultado del Teorema 7.1.3, podemos comparar el comportamiento asintótico de las cotas espaciales cuando comparemos los requerimientos espaciales de las máquinas de Turing. Por tanto, la máquina de Turing M_1 con cota espacial $S_1(n) = n^2 + 2n + 1$ y la máquina de Turing M_2 con cota espacial $S_2(n) = 3n^2 - 1$, tienen ambas la misma complejidad espacial $S(n) = n^2$. Por otro lado, una complejidad espacial de n^3 es menor que las complejidades espaciales n^4 o 2^n.

Recuerde que una DI de una máquina de Turing tiene en cuenta tanto el estado actual como el contenido actual de la cinta. Así, una cota espacial para una máquina de Turing proporciona una cota sobre el tamaño de una DI. Si unimos la información que aporta una cota espacial con lo que sabemos acerca del tamaño de los conjuntos de estados y de símbolos de cinta, obtendremos una cota para el número de movimientos que se realizan en una secuencia de aceptación. Por ejemplo, una máquina de Turing con k cintas de trabajo con un conjunto de estados Q, un alfabeto de cinta Γ y una cota espacial $S(n)$ tiene $n+2$ posiciones

para su cabeza de entrada, $S(n)$ posiciones para las cabezas de lectura/escritura de cada una de las k cintas de trabajo, $|\Gamma|^{S(n)}$ contenidos posibles para cada una de las cintas de trabajo, y $|Q|$ elecciones a realizar para obtener el estado actual. Por tanto, hay un máximo de $|Q|\ (n+2)\ |\Gamma|^{kS(n)}\ (S(n))^k$ posibles DI. Entonces, si M acepta su entrada en más de $|Q|\ (n+2)\ |\Gamma|^{kS(n)}\ (S(n))^k$ movimientos, entonces hay alguna DI repetida. Esta computación de aceptación contiene un bucle que puede ser omitido. De lo que se deduce que, si M acepta una entrada de longitud n, lo hará en $|Q|\ (n+2)\ |\Gamma|^{kS(n)}\ (S(n))^k$ movimientos como máximo. Obsérvese que si $S(n) \geq \log n$ entonces se puede encontrar una constante c para la cual $|Q|\ (n+2)\ |\Gamma|^{kS(n)}\ (S(n))^k \leq c^{S(n)}$ (véase Ejercicio 7.1.2). Tendremos el siguiente lema:

Lema 7.1.4. Sea M una máquina de Turing de k cintas con complejidad espacial $S(n)$, donde $S(n) \geq \log n$. Entonces existe una contante c para la cual, si w es una entrada cualquiera con longitud $|w| = n$, de modo que:

1. M tiene como máximo $c^{S(n)}$ descripciones instantáneas.
2. Si M acepta w, entonces lo hace en $c^{S(n)}$ movimientos como máximo.

Para eliminar los bucles (es decir, la secuencia de DI repetidas) de una computación de aceptación de M, tenemos el siguiente corolario:

Corolario 7.1.5. Sea M una máquina de Turing de k cintas con complejidad espacial $S(n)$, donde $S(n) \geq \log n$. Entonces existe una constante c para la cual, si M acepta la entrada w, entonces existe una computación $\alpha_1 \vdash \alpha_2 \vdash \ldots \vdash \alpha_m$ donde los α_i son distintos y $m \leq c^{S(n)}$.

En la Sección 4.4 se demostró que las máquinas de Turing deterministas y no deterministas son iguales en cuanto a capacidad de aceptación. La simulación de una máquina de Turing no determinista mediante una determinista realizará una búsqueda exhaustiva de todas las secuencias finitas de movimientos hasta obtener una computación de aceptación (si existe). De hecho, vamos a ver que es posible que bajo ciertas circunstancias, se pueda construir una máquina de Turing determinista que acepte el mismo lenguaje que una no determinista y cuya complejidad espacial sea el cuadrado de la complejidad de la no determinista.

Primero, observemos que, si M es una máquina de Turing de k cintas no determinista, podemos construir una máquina de Turing determinista M_1 de forma que, dadas dos DI de M, I_1 e I_2, compruebe si $I_1 \vdash I_2$. M_1 intentará obtener sistemáticamente, una transición de M que produzca I_2 a partir de I_1. Si encuentra una, la acepta o devuelve "verdad". Si no encuentra ninguna, la rechaza o devuelve "falso".

Supongamos que I_1 e I_2 son DI de M y $m \geq 0$. Definimos el predicado PRODUCE (I_1, I_2, m) que devuelve la respuesta a la pregunta ¿$I_1 \overset{(t)}{\vdash} I_2$ para algún $t \leq m$?

Obsérvese que PRODUCE (I_1, I_2, m) se comporta como sigue:

1. Si $m = 0$ e $I_1 = I_2$, devuelve "verdadero".
2. Si $m = 1$ e $I_1 \vdash I_2$, devuelve "verdadero".
3. Si $m > 1$ y PRODUCE $(I_1, I', \lceil \frac{m}{2} \rceil)$ y PRODUCE $(I', I_2, \lfloor \frac{m}{2} \rfloor)$ devuelven "verdadero", entonces devuelve "verdadero".
4. En otro caso, devuelve "falso".

Añadiendo una cinta de trabajo que funcione como una pila para la máquina de Turing determinista M_1, podemos calcular el predicado PRODUCE (I_1, I_2, m). En cada llamada recursiva, M_1 introduce I_1, I_2 e I' en la pila. Todos tienen una longitud máxima de $S(n)$. En este caso, para determinar si M acepta la entrada w, podemos evaluar PRODUCE (I_1, I_f, m), donde I_1 es una DI inicial de M, I_f es una DI de aceptación y $m = c^{S(n)}$. [Obsérvese que m puede ser codificado en modo binario usando como máximo $S(n) \log c$ bits o celdas. Luego, en cada apilamiento se podrían introducir $(3 + \log c)\, S(n)$ símbolos en la pila.] Ya que el parámetro m es dividido por dos en cada llamada recursiva, se hacen como máximo $1 + \lceil \log c^{S(n)} \rceil = 1 + \lceil S(n) \log c \rceil$ llamadas. Por tanto, la pila necesita un espacio igual a $(1 + \lceil S(n) \log c \rceil)\,(3 + S(n) \log c)$. Es decir, M_1 tiene una cota espacial $c_1 (S(n))^2$, donde $S(n)$ es la cota espacial de M y c_1 es una constante. Ya que M_1 es determinista, podemos usar una máquina de Turing determinista con complejidad espacial $c_1 (S(n))^2$ que acepta el mismo lenguaje que la máquina de Turing no determinista M, y la cual usa un espacio de $S(n)$ para aceptar una cadena.

Obsérvese que lo anterior depende totalmente del valor de $S(n)$. Para la entrada w de longitud $|w| = n$, si sabemos el valor de $S(n)$, podemos calcular $m = c^{S(n)}$ y determinar las DI I_1, I_2 e I' con una longitud apropiada, para determinar si M puede aceptar w de forma determinista en un espacio $c_1 (S(n))^2$. Si no sabemos el valor de $S(n)$, no podemos determinar la longitud máxima de una DI o la longitud máxima de una computación de aceptación.

Definición 7.1.6. Se dice que una función $S(n)$ es totalmente construible* en espacio si hay una máquina de Turing M_S que tiene una cota espacial $S(n)$ y, para toda entrada de longitud n, M_S usa exactamente las $S(n)$ celdas de su cinta de trabajo.

---

* NOTA: A pesar de que este término no se encuentra en la última edición del DRAE, lo acuñamos debido a que es el adjetivo que mejor expresa el hecho de que una función se construye.

Obsérvese que si $S(n)$ es totalmente construible en espacio entonces, dada una entrada w de longitud $|w| = n$, podemos usar M_S para distinguir las $S(n)$ celdas de la cinta de trabajo. La máquina de Turing M_1 precedente, puede usar dichas $S(n)$ celdas para generar la DI y calcular $m = c^{S(n)}$. Nótese que M_S es necesariamente una máquina de Turing determinista y, por tanto, la máquina de Turing compuesta M', construida a partir de M_1 y M_S también es determinista. Finalmente, obsérvese que por el Teorema 7.1.2, M' tiene una cota espacial de $(S(n))^2$. Luego hemos demostrado el siguiente teorema:

Teorema 7.1.7. (Teorema de Savitch) Si $S(n)$ es totalmente construible en espacio y M es una máquina con Turing no determinista de complejidad espacial $S(n)$, entonces hay una máquina de Turing M' determinista, con complejidad espacial $(S(n))^2$ para la cual $L(M) = L(M')$.

Los lenguajes acotados espacialmente forman una jerarquía en función del espacio requerido para que sean aceptados.

Definición 7.1.8. La familia de los lenguajes aceptados por máquinas de Turing deterministas con complejidad espacial $S(n)$ es ESPACIOD $(S(n))$. La familia de los lenguajes aceptados por máquinas de Turing no deterministas con complejidad espacial $S(n)$ es ESPACION $(S(n))$. Estas clases se conocen como *clases de complejidad espacial*.

Resumiremos en el Teorema 7.1.9, las relaciones que, según hemos visto, existen entre estas clases de complejidad.

Teorema 7.1.9. Sean S_1, S_2 y S funciones de $\mathbb{N}$ en $\mathbb{N}$. Supongamos que $S_1(n) \leq S_2(n)$ para todo n, y que $c > 0$. Entonces

1. ESPACIOD $(S_1(n)) \subseteq$ ESPACIOD $(S_2(n))$.
2. ESPACION $(S_1(n)) \subseteq$ ESPACION $(S_2(n))$.
3. ESPACIOD $(S(n)) \subseteq$ ESPACION $(S(n))$.
4. ESPACIOD $(S(n)) =$ ESPACIOD $(c\,S(n))$.
5. ESPACION $(S(n)) \subseteq$ ESPACIOD $((S(n))^2)$.

La inclusión de la parte (1) del Teorema 7.1.9 es propia bajo determinadas circunstancias. Vamos a ofrecer el siguiente teorema, el cual no demostraremos:

Teorema 7.1.10. Si $S_2(n)$ es totalmente contruible en espacio y

$$\lim_{n \to \infty} \frac{S_1(n)}{S_2(n)} = 0$$

y si $S_1(n) \geq \log n$ y $S_2(n) \geq \log n$, entonces existe un lenguaje

$$L \in \text{ESPACIOD}(S_2(n)) - \text{ESPACIOD}(S_1(n))$$

Obsérvese que el Teorema 7.1.10 no requiere que $S_1(n) \leq S_2(n)$ para todo n. Si, además de los requisitos que necesita el Teorema 7.1.10, también tenemos que $S_1(n) \leq S_2(n)$ se cumple para todo n, entonces ESPACIOD $(S_1(n)) \subset$ ESPACIOD $(S_2(n))$, es decir, se cumple la inclusión propia.

Definición 7.1.11. Si una máquina de Turing determinista con complejidad espacial $S(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots a_0$, acepta el lenguaje L, entonces se dice que L es un lenguaje de la clase ESPACIOP. Si la máquina de Turing que acepta L es no determinista y tiene cota espacial polinómica, entonces L es un lenguaje de la clase ESPACIONP.

Obsérvese que, puesto que

$$a_k n^k + a_{k-1} n^{k-1} + \ldots a_0 \leq (|a_k| + |a_{k-1}| + \ldots + |a_0|) n^k$$

por el Teorema 7.1.3 se obtiene que

$$\text{ESPACIOD}(a_k n^k + a_{k-1} n^{k-1} + \ldots a_0) \subseteq \text{ESPACIOD}(n^k)$$

Por tanto, podemos caracterizar ESPACIOP $= \bigcup_{k=1}^{\infty}$ ESPACIOD (n^k). De manera similar, ESPACIONP $= \bigcup_{k=1}^{\infty}$ ESPACION (n^k). Obsérvese que el teorema de Savitch (Teorema 7.1.7) dice que ESPACIONP = ESPACIOP ya que ESPACION $(n^k) \subseteq$ ESPACIOD(n^k).

Ejercicios de la Sección 7.1

7.1.1. Sea M una máquina de Turing de una cinta, definida mediante las transiciones que damos más abajo y en la que q_3 es el único estado de aceptación y q_1 es el estado inicial. Supongamos que el alfabeto de la cinta es $\{a, b, \dot{b}\}$. Obsérvese que $L(M) = \{a^{2k+1} b \mid k \geq 0\}$. Construir una máquina de Turing M' para la cual $L(M') = L(M)$, pero de forma que M' tenga como símbolos de la cinta codificaciones de pares de símbolos de la cinta de M (véase Teorema 7.1.3).

$$\Delta(q_1, a) = \{(q_2, a, R)\}, \qquad \Delta(q_2, a) = \{(q_1, a, R)\}$$
$$\Delta(q_2, b) = \{(q_3, b, R)\}$$

7.1.2. En este problema derivamos la constante c del Lema 7.1.4, haciendo referencia a las observaciones que preceden al Lema 7.1.4.

(a) Para $\log n \leq S(n)$, demostrar que $\log_{|\Gamma|}(n+2) \leq (k+1)\,S(n)$ con lo que $(n+2) \leq |\Gamma|^{(k+1)\,S(n)}$.

(b) Usar la parte (a) para demostrar que $(n+2)\,(S(n))^k \leq |\Gamma|^{(2k+1)\,S(n)}$.

(c) Obtener una constante c para la cual $|Q|\,(n+2)\,|\Gamma|^{kS(n)}\,(S(n))^k \leq c^{S(n)}$.

7.1.3. ¿A cuál de las clases de complejidad espacial pertenecen los siguientes tipos de lenguajes? Lenguajes regulares, lenguajes independientes del contexto, lenguajes sensibles al contexto.

7.1.4. Demostrar que, si una máquina de Turing con cota espacial $S(n) \geq \log n$ acepta el lenguaje L, entonces existe una máquina de Turing con una cota espacial $S(n)$ que acepta el lenguaje L y que para sobre todas las cadenas. (Por tanto, los lenguajes espacialmente acotados son lenguajes recursivos).

7.2 COMPLEJIDAD TEMPORAL

Aunque el espacio es un recurso importante de cualquier máquina de Turing, el tiempo de computación también es importante. En esta sección consideraremos que la complejidad temporal de las computaciones de máquinas de Turing, depende del tamaño de la cadena de entrada. El tiempo se mide por el número de movimientos que hace una máquina de Turing.

Definición 7.2.1. Sea M una máquina de Turing de k cintas. Supongamos que M realiza como máximo $T(n)$ movimientos sobre toda cadena de longitud n y para una función T: $\mathbb{N} \to \mathbb{N}$. Entonces se dice que M tiene *complejidad temporal* $T(n)$ o que es una máquina de Turing *con cota temporal* $T(n)$. Además se dice que $L(M)$ es un lenguaje temporalmente acotado por $T(n)$ o con complejidad temporal $T(n)$.

Obsérvese que, para que una máquina de Turing lea su entrada, debe realizar al menos $n+1$ movimientos. Por tanto, $T(n) \geq n+1$ para cualquier cota temporal $T(n)$.

Ejemplo 7.2.1

Consideremos el lenguaje $L = \{xcx^I \mid x \in \Sigma^*\}$. Para reconocer el lenguaje L con una máquina de Turing de una cinta, podemos recorrer la entrada de atrás

hacia adelante comparando los símbolos del comienzo con los símbolos del final hasta que encontremos la c central. Cada recorrido completo a través de la entrada requiere $2t$ movimientos, donde t es la longitud actual de la cadena de entrada "no comparada". En cada recorrido completo, t se reduce en 1. Esto continúa hasta que $t = 1$ (la c está en el centro). Por tanto, tenemos que

$$T(n) = n^2 + n - 2$$

También podríamos reconocer L usando una máquina de Turing de dos cintas. La máquina de Turing copia los símbolos de la entrada en la segunda cinta hasta que encuentra la c central. Entonces, en cada paso mueve la cabeza de la segunda cinta hacia la izquierda, comparando los símbolos de la segunda cinta con los de la primera. Esta máquina de Turing requiere $n + 1$ movimientos para reconocer una entrada de longitud n. Por tanto, para ésta, $T(n) = n + 1$.

El Ejemplo 7.2.1 sugiere que la complejidad temporal está relacionada con el número de cintas disponibles para realizar la computación.

El Teorema 7.1.3 decía que si se trataran los símbolos de las múltiples cintas de forma ingeniosa, como un único símbolo, podríamos reducir linealmente el espacio requerido para la computación. Se puede aplicar lo mismo en el caso del tiempo; teniendo en cuenta ciertas restricciones, se puede incrementar linealmente la velocidad de una computación.

Supongamos que M es una máquina de Turing de una cinta. Consideramos M', la cual se deriva de M por medio de la unión de m símbolos de la cinta en uno sólo. (El alfabeto de la cinta de M' debería ser $\Gamma' = \Gamma^m$, donde Γ es el alfabeto de la cinta de M). Entonces, cuando la cabeza de lectura/escritura de M entra en un bloque de m celdas de la cinta, entra o por el lado derecho o por el izquierdo. Al principio, la cabeza de lectura/escritura de M se encuentra sobre la celda del extremo izquierdo del primer bloque de m celdas. M' puede usar los estados para guardar en qué lugar del bloque de m celdas, que corresponde a una única celda de la cinta de M', se encuentra la cabeza de M. Una vez que la cabeza de M entra en un bloque de m celdas, realiza una secuencia de movimientos antes de salir. Dichos movimientos pueden cambiar el contenido del bloque de m celdas. Es decir, cuando la cabeza de M' se mueve desde una celda de la cinta, debemos escribir en dicha celda el bloque de m símbolos que se corresponde con el contenido de la cinta de M cuando su cabeza se mueve desde el bloque de m celdas correspondiente. Podemos determinar, qué tiene el bloque de m celdas mirando simplemente el comportamiento de M sobre la cinta que contiene el bloque de m símbolos. Por tanto, dado un bloque de m símbolos, podemos determinar qué bloque de m símbolos lo sustituirá. Es decir, podemos determinar que escribirá M' sobre su cinta cuando retira su cabeza del símbolo actual.

Consideremos la máquina de Turing M que calcula la función de cadena $f((ab)^n) = (ab)^{n+1}$, para la cual se tienen las siguientes transiciones:

$$\Delta(q_1, a) = (q_2, a, R) \qquad \Delta(q_1, b) = (q_5, b, S)$$
$$\Delta(q_2, b) = (q_1, b, R) \qquad \Delta(q_2, a) = (q_5, a, S)$$
$$\Delta(q_1, \not{b}) = (q_3, a, R) \qquad \Delta(q_3, a) = (q_5, a, S)$$
$$\Delta(q_3, \not{b}) = (q_4, b, R) \qquad \Delta(q_3, b) = (q_5, b, S)$$

(aquí, el estado inicial es q_1, y q_4 es el único estado final). Para construir una máquina de Turing M', que una las dos cintas de celdas de M en una sola celda, necesitamos un alfabeto

$$\Gamma' = \{\text{'}aa\text{'}, \text{'}ab\text{'}, \text{'}ba\text{'}, \text{'}bb\text{'}, \text{'}\not{b}\not{b}\text{'}, \text{'}a\not{b}\text{'}, \text{'}\not{b}a\text{'}, \text{'}b\not{b}\text{'}, \text{'}\not{b}b\text{'}\}$$

Obsérvese que esto es esencialmente $\Gamma \times \Gamma$. Las transiciones de M' tienen en cuenta las acciones que realiza M sobre cada bloque de símbolo de las dos cintas. Por tanto, M' tiene las siguientes transiciones:

$$\Delta'(p_1, \text{'}ab\text{'}) = (p_1, \text{'}ab\text{'}, R) \qquad \Delta'(p_1, \text{'}\not{b}\not{b}\text{'}) = (p_2, \text{'}ab\text{'}, R)$$
$$\Delta'(p_1, \text{'}a\not{b}\text{'}) = (p_3, \text{'}a\not{b}\text{'}, R) \qquad \Delta'(p_1, \text{'}aa\text{'}) = (p_3, \text{'}aa\text{'}, R)$$
$$\Delta'(p_1, \text{'}\not{b}a\text{'}) = (p_3, \text{'}\not{b}a\text{'}, R) \qquad \Delta'(p_1, \text{'}b\not{b}\text{'}) = (p_3, \text{'}b\not{b}\text{'}, R)$$
$$\Delta'(p_1, \text{'}\not{b}b\text{'}) = (p_3, \text{'}\not{b}b\text{'}, R) \qquad \Delta'(p_1, \text{'}bb\text{'}) = (p_3, \text{'}bb\text{'}, R)$$

Obsérvese que, para una computación de M sobre la entrada $(ab)^n$, se cumple que $T(n) = 2n + 2$, mientras que para la computación correspondiente a M' sobre la entrada $(ab)^n$ se cumple $T'(n) = n + 1$.

Esto, todavía se puede mejorar. Si sabemos el contenido de los bloques de m símbolos que ocupan las celdas que se encuentran a la izquierda y a la derecha del bloque actual de m celdas de M, podemos determinar sus contenidos después de que M mueva su cabeza hacia la izquierda o hacia la derecha del bloque actual de m celdas. Por tanto, podemos determinar el contenido de las celdas de las dos cintas que se encuentran a la izquierda o a la derecha de la celda de la cinta actual de M'. Puesto que M requiere al menos m movimientos para salir de la región de $3m$ celdas, podemos simular al menos m movimientos de M ajustando las celdas de las tres cintas de M' y moviendo su cabeza.

Supongamos que M' mueve su cabeza una celda hacia la izquierda, dos celdas hacia la derecha y después una celda a la izquierda, guardando (por medio de los estados) el contenido de las celdas que se encuentran a la izquierda y la derecha de la celda actual. En este momento, tenemos información suficiente como para determinar cuál será el contenido de esas tres celdas, cuando M mueva su cabeza fuera de las $3m$ celdas correspondientes. En cuatro movimientos, M' puede actualizar el contenido de las celdas derecha, izquierda y actual de forma adecuada y mover su cabeza al lugar que corresponde a la posición de la cabeza de M cuando ha llegado al final de dichos movimientos. Por tanto, en ocho

movimientos M' simula al menos m movimientos de M. Si M hace $T(n)$ movimientos, M' hace como máximo $8\lceil \frac{T(n)}{m} \rceil$ movimientos.

Aunque hemos centrado nuestra atención sobre una máquina de Turing de una cinta, no se plantea ningún problema si extendemos lo visto a una máquina de Turing de k cintas. Sin embargo, antes se suponía que la entrada se le proporciona a M' en forma codificada. Si M es un máquina de Turing de k cintas con $k > 1$, podemos construir una máquina de Turing M' como antes pero añadiéndole la capacidad de recibir la misma cadena de entrada de M y codificarla. M' simplemente recorre la cinta sobre la que se encuentra la entrada, codificándola y uniendo m símbolos en uno, escribiendo el resultado en la segunda cinta. Esta segunda cinta se convierte en la cinta de entrada, mientras que la primera pasa a ser la cinta de trabajo. Obsérvese que realiza n movimientos para recorrer la entrada y $\lceil \frac{n}{m} \rceil$ movimientos para mover la cabeza de la nueva cinta de trabajo a la posición de comienzo de la cadena de entrada codificada. Por tanto, en $n + \lceil \frac{n}{m} \rceil + 8\lceil \frac{T(n)}{m} \rceil$ movimientos, M' puede realizar la misma computación que M hace en $T(n)$ movimientos.

Consideremos la cantidad $n + \lceil \frac{n}{m} \rceil + 8\lceil \frac{T(n)}{m} \rceil$. Primero observemos que, para cualquier número x, $\lceil x \rceil < x + 1$. Segundo, obsérvese que, cuando $n \geq 6$, $T(n) \geq 7$. Finalmente, supongamos que $\inf_{n \to \infty} \frac{T(n)}{n} = \infty$, por lo tanto, para cualquier constante d existe un N tal que, para $n \geq N$, tenemos $\frac{T(n)}{n} \geq d$. Suponiendo que $n \geq N \geq 6$, obtenemos:

$$n + \lceil \frac{n}{m} \rceil + 8\lceil \frac{T(n)}{m} \rceil \leq n + \frac{n}{m} + 1 + 8\left(\frac{T(n)}{m} + 1 \right) =$$

$$= (n+2) + \frac{n}{m} + 8\frac{T(n)}{m} + 7 \leq$$

$$\leq 2n + \frac{n}{m} + 8\frac{T(n)}{m} + T(n) \leq$$

$$\leq \frac{2T(n)}{d} + \frac{T(n)}{dm} + T(n)\left(\frac{8}{m} + 1 \right) =$$

$$= T(n)\left(\frac{2}{d} + \frac{1}{dm} + \frac{8+m}{m} \right)$$

Por tanto, dado un $c > 0$, podemos ajustar m y d de forma que el número de movimientos de M' sea como máximo $cT(n)$. Ésta es la aceleración lineal que buscábamos.

Teorema 7.2.2. Sea $k > 1$. Supongamos que $\inf_{n \to \infty} \frac{T(n)}{n} = \infty$. Si la máquina de Turing de k cintas con complejidad temporal $T(n)$, acepta el lenguaje L, entonces, para $c > 0$, la máquina de Turing de k cintas con complejidad temporal $cT(n)$ también acepta L.

Obsérvese que si L es aceptado por una máquina de Turing de una cinta con cota temporal $T(n)$, entonces también es aceptado por una máquina de Turing de k cintas con complejidad temporal $T(n)$ para $k > 1$. Por tanto, obtenemos el siguiente corolario:

Corolario 7.2.3. Sea $\inf_{n \to \infty} \frac{T(n)}{n} = \infty$. Si una máquina de Turing que tiene complejidad temporal $T(n)$ acepta L, entonces, para todo $c > 0$, una máquina de Turing con complejidad temporal $cT(n)$ acepta L.

Si $T_1(n) \leq T_2(n)$ para todo n y L es un lenguaje con complejidad temporal $T_1(n)$, entonces L también tiene complejidad temporal $T_2(n)$. En particular, si L es un lenguaje con complejidad temporal

$$T_1(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$$

obsérvese que $T_1(n) \leq (|a_k| + |a_{k-1}| + \ldots + |a_0|) n^k$ para todo n. Aplicando lo anterior con

$$c = \frac{1}{|a_k| + |a_{k-1}| + \ldots + |a_0|}$$

se obtiene que L tiene complejidad temporal n^k.

En el Ejemplo 7.2.1, vimos cuál era la complejidad temporal de $L = \{xcx^I | x \in \Sigma^*\}$. Cuando usábamos una máquina de Turing con una cinta, la complejidad temporal era de n^2. Sin embargo, con dos cintas, L tiene complejidad temporal n. De esto se obtiene el siguiente teorema:

Teorema 7.2.4. Si L es aceptado por una máquina de Turing de k cintas con complejidad temporal $T(n)$ y si $\inf_{n \to \infty} \frac{T(n)}{n} = \infty$, entonces L es aceptado por una máquina de Turing de una cinta que tiene complejidad temporal $(T(n))^2$.

Demostración. Si una máquina de Turing de una cinta con complejidad temporal $T(n)$, acepta el lenguaje L, entonces, puesto que $T(n) \leq (T(n))^2$ para todo n, L también es aceptado por una máquina de Turing de una cinta con complejidad $(T(n))^2$. Supongamos que L se acepta mediante una máquina de Turing con $k > 1$ cintas y cota temporal $T(n)$. Consideramos una simulación de una máquina de Turing de k cintas por medio de una máquina de Turing de una cinta, como vimos en la Sección 4.4. Después de t movimientos, los marcadores de cabeza pueden estar

$2t$ celdas más allá. Cada movimiento de la máquina de Turing de k cintas corresponde a un barrido de todas las celdas marcadas que se analizan mediante las k cabezas, seguido por un barrido hacia atrás para actualizar los marcadores de cabeza y el contenido de las celdas. El barrido hacia atrás realiza cinco movimientos por celda. Por tanto, t movimientos de la máquina de Turing de k cintas corresponden a $12t$ movimientos de la simulación. Luego $T(n)$ movimientos pueden requerir

$$\sum_{t=1}^{T(n)} 12t = 6\,(T(n))^2 + 6T(n)$$

movimientos en la simulación. La simulación se puede acelerar vía el Teorema 7.2.2 a $(T(n))^2$. □

A continuación definiremos unas clases análogas a las clases de complejidad espacial ESPACIOD y ESPACION:

Definición 7.2.5. La familia de los lenguajes aceptados por máquinas de Turing deterministas con complejidad temporal $T(n)$ es TIEMPOD $(T(n))$. La familia de los lenguajes que aceptan las máquinas de Turing no deterministas con complejidad temporal $T(n)$ es TIEMPON $(T(n))$. Dichas clases se llaman *clases de complejidad temporal*.

Teorema 7.2.6.

1. Si $T_1(n) \leq T_2(n)$, entonces TIEMPOD $(T_1(n)) \subseteq$ TIEMPOD $(T_2(n))$.
2. Si $T_1(n) \leq T_2(n)$, entonces TIEMPON $(T_1(n)) \subseteq$ TIEMPON $(T_2(n))$.
3. TIEMPOD $(T(n)) \subseteq$ TIEMPON $(T(n))$.
4. Si $\inf_{n\to\infty} \frac{T(n)}{n} = \infty$, entonces TIEMPOD $(T(n))$ = TIEMPOD $(cT(n))$ para todo $c \geq 0$.

Las partes 1 y 2 del Teorema 7.2.6 dicen que todo lenguaje con complejidad temporal $T_1(n)$ también tiene complejidad temporal $T_2(n)$ para $T_2(n)$ "mayor" que $T_1(n)$. Es lógico preguntarse cuánto más grande debe ser $T_2(n)$, ya que hay lenguajes con complejidad temporal $T_2(n)$ que no tienen complejidad temporal $T_1(n)$. La parte 4 del Teorema 7.2.6 dice que si $T_1(n)$ satisface $\inf_{n\to\infty} \frac{T(n)}{n} = \infty$, entonces $T_2(n)$ debe ser mayor que una constante multiplicada por $T_1(n)$. El Teorema 7.2.8 el cual presentaremos sin realizar su demostración, nos proporciona las condiciones suficientes para que existan lenguajes que pertenezcan a TIEMPOD $(T_2(n))$ que no estén en TIEMPOD $(T_1(n))$. El Teorema 7.2.8 depende de la siguiente definición, la cual es análoga a la Definición 7.1.6.

Definición 7.2.7. Una función $T(n)$ es *totalmente contruible en tiempo* si existe una máquina de Turing acotada temporalmente mediante $T(n)$ que tenga exactamente una duración de $T(n)$ sobre todas las cadenas de longitud n.

Teorema 7.2.8. Si $T_2(n)$ es totalmente construible en tiempo y

$$\inf_{n \to \infty} \frac{T_1(n) \log (T_1(n))}{T_2(n)} = 0$$

entonces existe un lenguaje $L \in$ TIEMPOD $(T_2(n))$ – TIEMPOD $(T_1(n))$.

El Teorema 7.2.9 presenta la relación que existe entre tiempo y espacio.

Teorema 7.2.9. Si $L \in$ TIEMPOD $(f(n))$, entonces $L \in$ ESPACIOD $(f(n))$.

Demostración. Supongamos que la máquina de Turing que realiza como máximo $f(n)$ movimientos sobre la cadena de longitud n, acepta el lenguaje L. Entonces M puede inspeccionar $1 + f(n)$ celdas de la cinta como máximo, y por tanto $L \in$ ESPACIOD $(f(n) + 1)$. Pero $f(n) + 1 \leq 2f(n)$, ya que las cotas temporales son al menos $n + 1$, luego por el Teorema 7.1.3, $L \in$ ESPACIOD $(f(n))$. □

Hay una forma importante de relacionar el tiempo determinista y el tiempo no determinista. Supongamos que M es una máquina de Turing no determinista con una cota temporal $T(n)$. Vamos a contar (y limitar) el número de posibles DI de M. Si M tiene k cintas, entonces M puede inspeccionar $T(n) + 1$ celdas como máximo durante el tiempo $T(n)$. Por tanto, en $T(n)$ movimientos una cinta puede llegar a contener como máximo una cadena de longitud $T(n) + 1$. Si Γ es el alfabeto de la cinta de M, hay $|\Gamma|^{T(n)+1}$ cadenas de longitud $T(n) + 1$ posibles. Éste es el número de cadenas que M puede dejar sobre su cinta. Puesto que hay k cintas, existen $|\Gamma|^{k(T(n)+1)}$ cadenas posibles que describen el contenido de las cintas de M después de realizar $T(n)$ movimientos. Es más, cada cabeza de lectura/escritura puede estar sobre una de las $T(n) + 1$ celdas posibles de cada una de las k cintas. Si Q es el conjunto de estados de M, entonces el número de DI posibles sobre una cadena de entrada n es

$$|Q| \ (T(n) + 1)^k \ |\Gamma|^{k(T(n)+1)} \leq c^{T(n)}$$

para una constante c elegida de forma apropiada. Sistemáticamente, vamos a buscar una DI de aceptación que sea accesible desde la DI inicial de M mediante $T(n)$ movimientos y usando una máquina de Turing determinista que genere y compruebe dichas DI. Primero buscaremos las DI a las que se pueden acceder en un único movimiento, luego las que son accesibles en dos y así sucesivamente.

hasta llegar a $T(n)$ movimientos. Sea r la longitud de una DI. Suponiendo que generamos las DI una tras otra sobre la cinta, realizaremos como máximo $3r$ movimientos para generar y comprobar cada DI. En cada etapa hay como máximo $c^{T(n)}$ DI siguientes. Por lo tanto, cada etapa requiere como máximo $3rc^{T(n)}$ movimientos. En esta búsqueda hay como máximo $T(n)$ etapas, de modo que dicha búsqueda requiere $3rT(n)\,c^{T(n)}$ movimientos. La longitud de una DI es como máximo $r = 1 + k(T(n) + 2)$; por tanto, dada una constante d elegida de forma apropiada, la búsqueda requiere $d^{T(n)}$ movimientos como máximo. Veamos el siguiente teorema:

Teorema 7.2.10. Si una máquina de Turing no determinista con complejidad temporal $T(n)$ acepta L, entonces existe una máquina de Turing determinista con complejidad temporal $d^{T(n)}$, para alguna contante d, que también acepta L. Es decir, si $L \in$ TIEMPON $(T(n))$, entonces hay una constante d para la cual $L \in$ TIEMPOD $(d^{T(n)})$.

En el Lema 7.1.4 se observó que si M tiene complejidad espacial $S(n) \geq \log n$, entonces podemos limitar el número de DI distintos que M tiene, por $c^{S(n)}$ para alguna constante c. Si M es una máquina de Turing determinista, entonces que la misma DI aparezca dos veces, indica que M entra en un bucle y, por tanto, nunca para o no acepta la cadena. De esto se deduce que se puede construir una máquina de Turing M', determinista, que simule M y lleve la cuenta de los movimientos sobre una cinta adicional. Si la cuenta supera $c^{S(n)}$, la simulación tiene DI de M repetidas, y en consecuencia M' para y rechaza la cadena. Está claro que M' tiene una cota temporal de $2c^{S(n)}$. Si aplicamos el Teorema 7.2.2 se obtiene el siguiente teorema:

Teorema 7.2.11. Si $L \in$ ESPACIOD $(S(n))$ y $S(n) \geq \log n$, entonces $L \in$ TIEMPOD $(c^{S(n)})$ para una constante c.

En la Definición 7.1.11, agrupamos en una clase todos los lenguajes que aceptaban las máquinas de Turing deterministas con cota espacial polinómica y todos los que aceptaban las máquinas de Turing no deterministas con cota espacial polinómica, en otra. El teorema de Savitch demuestra que estas dos clases de lenguajes son la misma puesto que se pasa de espacio no determinista a espacio determinista sólo con elevar al cuadrado la cota espacial. Del Teorema 7.2.10 se puede deducir que no se puede obtener un resultado tan bueno para las clases de complejidad temporal ya que el pasar del no determinismo al determinismo, eleva a un exponente la cota temporal.

Definición 7.2.12. La clase $\mathcal{P}$ se compone de todos los lenguajes que acepta una máquina de Turing determinista que tiene una cota temporal polinómica. La clase $\mathcal{NP}$ se compone de todos los lenguajes que aceptan máquinas de Turing no deterministas que tengan una cota temporal polinómica.

Obsérvese que, pára todo n

$$a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \leq (|a_k| + |a_{k-1}| + \ldots + |a_0|) n^k$$

así que, aplicando el Corolario 7.2.3, se obtiene

$$\text{TIEMPOD}\ (a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0) \subseteq \text{TIEMPOD}\ (n^k)$$

De lo que se deduce que podemos caracterizar $\mathcal{P}$ como

$$\mathcal{P} = \bigcup_{n \geq 1} \text{TIEMPOD}\ (n^k)$$

De forma similar,

$$\mathcal{NP} = \bigcup_{n \geq 1} \text{TIEMPON}\ (n^k)$$

Finalmente, puesto que TIEMPOD $(n^k) \subseteq$ TIEMPON (n^k) para todo k, obtenemos que $\mathcal{P} \subseteq \mathcal{NP}$. Desgraciadamente, esto queda un poco lejos de los conocimientos que tenemos. Uno de los problemas más importantes que todavía no ha resuelto la ciencia de la computación es la cuestión de la igualdad de $\mathcal{P}$ y $\mathcal{NP}$. Responder afirmativamente implica tener que demostrar que todo lenguaje de $\mathcal{NP}$ también es de $\mathcal{P}$. Esto requeriría encontrar alguna forma de transformar una máquina de Turing no determinista con cota temporal polinómica en una máquina de Turing determinista con cota temporal polinómica (los polinomios no tienen por qué ser del mismo grado). El Teorema 7.2.10 dice que podemos realizar la transformación de una máquina de Turing no determinista a una determinista e incrementamos la cota temporal en una cantidad exponencial. Esto no quiere decir que la transformación que necesitamos sea imposible, sólo que no sabemos cómo realizarla.

Por otro lado, demostrar que $\mathcal{P} \neq \mathcal{NP}$ requiere que encontremos un lenguaje que esté en $\mathcal{NP}$ pero no en $\mathcal{P}$. Si L es dicho lenguaje, debemos demostrar que no hay ninguna máquina de Turing con cota temporal polinómica que acepte L. De todas formas, hay lenguajes de $\mathcal{NP}$ que no sabemos si están en $\mathcal{P}$ o no lo están.

Ejercicios de la Sección 7.2

7.2.1. Demostrar que, si una máquina de Turing de k cintas con cota temporal $T(n) = cn$ para $k > 1$ y una constante c acepta L, entonces, para todo $\varepsilon > 0$, hay una máquina de Turing con cota temporal $(1 + \varepsilon)\, n$ que acepta L.

7.2.2. ¿Es cierto que TIEMPOD $(2^{2^n}) \subseteq$ TIEMPOD $(2^{2^n + n})$? ¿Por qué?

7.3 INTRODUCCIÓN A LA TEORÍA DE LA COMPLEJIDAD

En la Definición 4.2.2, definíamos que una función de cadena f es Turing computable si existe una máquina de Turing M que, dada una cadena de entrada w, calcula u siempre que $f(w) = u$. Una función de cadena que es Turing computable se dice que es *computable en tiempo polinómico* si hay una máquina de Turing que la calcula y tiene una cota temporal polinómica.

Definición 7.3.1. Se dice que un lenguaje L_1 es *reducible en tiempo polinómico* a un lenguaje L_2 si hay una función de cadena computable en tiempo polinómico para la cual $f(u) \in L_2$ si y sólo si $u \in L_1$.

Las reducciones en tiempo polinómico son importantes para determinar a cual de las clases ($\mathcal{P}$ o $\mathcal{NP}$) pertenece un lenguaje, como indica el siguiente teorema.

Teorema 7.3.2. Si L_1 es reducible en tiempo polinómico a L_2, entonces

a. Si $L_2 \in \mathcal{P}$ entonces $L_1 \in \mathcal{P}$.

b. Si $L_2 \in \mathcal{NP}$ entonces $L_1 \in \mathcal{NP}$.

Demostración. (b) Supongamos que $L_2 \in \mathcal{NP}$ y que f es una función que reduce L_1 a L_2 en tiempo polinómico. Para comprobar si una cadena w está en L_1, podemos calcular $f(w)$ en tiempo polinómico y entonces usar una máquina de Turing no determinista con una cota temporal polinómico para L_2, para comprobar si $f(w) \in L_2$. Puesto que la composición de polinómicos con polinómicos es también polinómico, queda demostrado.

La demostración de (a) es similar. □

La notación $L_1 <_p L_2$ se usa para indicar que L_1 es reducible a L_2 en tiempo polinómico. Obsérvese que si $L_1 <_p L_2$, entonces determinar si $w \in L_1$ no es más difícil que determinar si $f(w) \in L_2$, donde f es la función que reduce L_1 a L_2 en tiempo polinómico.

Definición 7.3.3. Para toda clase C de lenguajes, un lenguaje L se dice que es *C-difícil* (o difícil para la clase C) si, para todo $L' \in C$, $L' <_p L$. Es decir, todos los lenguajes se reducen a L en tiempo polinómico.

En particular L es $\mathcal{NP}$-difícil si para todo lenguaje $L' \in \mathcal{NP}$, $L' <_p L$.

Obsérvese que la definición del lenguaje C-difícil L, no dice nada sobre dónde está L. Es decir, L puede estar o no estar en C.

Definición 7.3.4. Si L es C-difícil y $L \in C$, entonces se dice que L es *C-completo*.

En particular, si L es $\mathcal{NP}$-difícil y L también es un lenguaje que pertenece a $\mathcal{NP}$, entonces L es $\mathcal{NP}$-completo.

Teorema 7.3.5. Si L es un lenguaje $\mathcal{NP}$-completo y $L \in \mathcal{P}$, entonces $\mathcal{P} = \mathcal{NP}$.

Demostración. Sea el lenguaje $L_1 \in \mathcal{NP}$. Entonces, puesto que L es $\mathcal{NP}$-difícil, tenemos que $L_1 <_p L$. Ya que $L \in \mathcal{P}$, aplicamos el Teorema 7.3.2, con lo que $L_1 \in \mathcal{P}$. □

El Teorema 7.3.5 nos proporciona un medio para establecer la igualdad entre $\mathcal{P}$ y $\mathcal{NP}$. Es decir, podemos demostrar que $\mathcal{P} = \mathcal{NP}$ demostrando que un lenguaje $\mathcal{NP}$-completo está en $\mathcal{P}$. Naturalmente, esto requiere que tengamos un lenguaje $\mathcal{NP}$-completo. Vamos a construir un lenguaje $\mathcal{NP}$-completo y demostraremos que lo es. El Teorema 7.3.8 y su corolario nos muestran una forma, relativamente fácil, para obtener lenguajes $\mathcal{NP}$-completos.

El primer lenguaje $\mathcal{NP}$-completo que encontramos es el lenguaje llamado L_{sat}. Se compone de todas las expresiones booleanas codificadas para las que una asignación de valores de verdad, hace que sean evaluadas como verdaderas.

Una *variable booleana* es un variable que toma los valores de verdadero o falso. Vamos a representar verdadero mediante 1 y falso por medio del 0. Se llama *literal* a una variable o a su negación. Recordemos las conectivas lógicas $\wedge$ (y/conjunción), $\vee$ (o/disyunción) y la negación vistas en el Capítulo 0. Dichas conectivas se usan para formar *expresiones booleanas*. Una *cláusula* es un tipo de expresión booleana que consiste en la disyunción de literales. Por tanto, $y \vee \overline{z} \vee \overline{x}$ es una cláusula. Una *asignación de valores de verdad* para un conjunto de variables booleanas es una asignación de los valores 0 y 1 a las variables booleanas del conjunto. Una cláusula X es *satisfactible** si hay una asignación de valores de verdad para sus variables que hacen que X sea verdadera. Una colec-

* NOTA: Acuñamos este término por considerar que expresa, mejor que cualquiera que aparece en el DRAE, el significado de lo que se quiere definir.

ción de cláusulas es satisfactible si hay una asignación de valores de verdad para las variables de las cláusulas que satisface todas las cláusulas simultáneamente.

Por ejemplo, sea

$$C = \{x_1 \vee \bar{x}_2, \bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3, \bar{x}_1 \vee x_2 \vee x_3\}$$

Una asignación de valores de verdad que sea satisfactible para C es $x_1 = 1$, $x_2 = 0$ y $x_3 = 1$. Todas las colecciones de cláusulas No son satisfactibles. Por ejemplo, el conjunto

$$D = \{\bar{x}_1, x_1 \vee x_2, x_1 \vee \bar{x}_2\}$$

no es satisfactible. El *problema de la satisfactibilidad** (SAT) es el problema de decidir si es satisfactible una colección de cláusulas arbitraria.

Obsérvese que hemos descrito un problema de decisión: "Dado un conjunto de cláusulas, ¿Es satisfactible dicho conjunto de cláusulas?" Podemos transformar SAT en un lenguaje L_{sat} por medio de la codificación de las muestras de SAT. Supongamos que una muestra de SAT contiene las variables $x_1, x_2, ..., x_n$. podemos codificar x_i como el símbolo '&' seguido por i escrita como una cadena binaria (de ceros y unos). Para representar un ejemplo de SAT necesitamos un alfabeto $\Sigma = \{',', '\vee', '-', '\&', '0', '1'\}$, donde '–' se usa para representar la negación. Sea

$$L_{sat} = \{w \in \Sigma^* \mid w \text{ representa un conjunto satisfactible de cláusulas}\}$$

Dada una cláusula codificada y una asignación de valores de verdad para las variables, podemos evaluar la cláusula en tiempo polinómico. Por tanto, dada una asignación de valores de verdad podemos determinar, en tiempo polinómico, si se satisface un conjunto codificado de cláusulas. Por tanto L_{sat} está en $\mathcal{NP}$ puesto que una máquina de Turing no determinista puede determinar si $w \in L_{sat}$, buscando primero una asignación de valores de verdad y después evaluando las cláusulas en tiempo polinómico. Así, tendremos el siguiente lema:

Lema 7.3.6. $L_{sat} \in \mathcal{NP}$

Por tanto, si podemos demostrar que L_{sat} es $\mathcal{NP}$-difícil, tendremos un lenguaje $\mathcal{NP}$-completo. Para demostrar que L_{sat} es $\mathcal{NP}$-difícil debemos buscar un lenguaje $L \in \mathcal{NP}$ y reducirlo a L_{sat} en tiempo polinómico. Ya que L es arbitrario, no no depende de ninguna de las características de L, sólo de que está en $\mathcal{NP}$. Es decir, sólo sabemos que L es aceptado por una máquina de Turing no determi-

* Ver nota de la página anterior.

nista con cota temporal polinómica. ¡Tenemos la suficiente información como para realizar una reducción en tiempo polinómico!

Supongamos que $L \in \mathcal{NP}$ es aceptado por una máquina de Turing M no determinista que tiene una cota temporal polinómica $T(n)$. Dada una entrada $w = \sigma_1\sigma_2 \ldots \sigma_n$ para M, la vamos a transformar en una muestra de SAT, E_w, de forma que E_w es satisfactible si y sólo si $w \in L$. Obsérvese que E_w es, necesariamente, una colección de cláusulas; por tanto, la transformación debe incluir las variables booleanas y la disyunción.

Si M acepta una cadena de longitud n, hay una computación, que lleva a una aceptación que requiere como máximo $T(n)$ pasos. Por el Teorema 7.2.4, podemos suponer que M tiene una única cinta. Supongamos que numeramos las celdas de la cinta de M, que ocupa la entrada, como 1, 2, ..., n y las celdas que se encuentran a la izquierda del primer símbolo se numeran como 0, –1, ... Obsérvese que en $T(n)$ movimientos, M sólo puede consultar las celdas con los números $-T(n)$, $-T(n) + 1$, ..., $T(n) + 1$. Por tanto, las posiciones que ocupa la cabeza de M durante la computación se pueden especificar mediante variables booleanas H_{ij} para $0 \leq i \leq T(n)$, $-T(n) \leq j \leq T(n) + 1$, donde $H_{ij} = 1$ significa que la cabeza de M analiza la celda j después del movimiento i.

Supongamos que M tiene los estados $q_0, q_1, \ldots, q_p$, donde q_0 es el estado inicial de M y q_p es el único estado de aceptación. Vamos a definir unas variables booleanas Q_{ij} asociadas a la computación, de forma que serán verdaderas si, después del movimiento i, M pasa al estado q_j, donde $0 \leq i \leq T(n)$ y $0 \leq j \leq p$.

Finalmente, supongamos que el alfabeto de la cinta de M es $\Gamma = \{\tau_0, \tau_1, \ldots, \tau_m\}$, donde τ_0 es el blanco (es decir, $\tau_0 = b$). Definimos las variables booleanas S_{ijk} para $0 \leq i \leq T(n)$, $-T(n) \leq j \leq T(n) + 1$, y $0 \leq k \leq m$, cuyo valor será verdadero si y sólo si después del movimiento i, la celda j contiene τ_k. Luego toda computación de M corresponde a una asignación de valores de verdad para las variables H_{ij}, Q_{ij} y S_{ijk}.

Obsérvese que, aunque una computación de M induce una asignación de valores de verdad para las variables, lo inverso no es cierto. Es decir, una asignación de valores de verdad arbitraria para las variables, no necesita corresponder a una computación de M. Podemos transformar el problema de determinar si M acepta w en uno en el cual el conjunto de cláusulas E_w sobre H_{ij}, Q_{ij} y S_{ijk} es satisfactible si y sólo si M acepta w en $T(n)$ movimientos.

Hay cuatro situaciones necesarias para que una asignación de valores de verdad corresponda a una computación de aceptación de M. La asignación de valores de verdad debe reflejar lo siguiente:

1. La cabeza de lectura/escritura sólo puede analizar una celda, cada celda puede contener un único símbolo y M puede estar en un único estado.

(Aquí $d = -1$ si $X = L$ y $d = 1$ si $X = R$). Las cláusulas

$$\overline{H_{ij}} \vee \overline{Q_{is}} \vee \overline{S_{ijt}} \vee H_{i+1,j+d}$$
$$\overline{H_{ij}} \vee \overline{Q_{is}} \vee \overline{S_{ijt}} \vee Q_{i+1,s'}$$
$$\overline{H_{ij}} \vee \overline{Q_{is}} \vee \overline{S_{ijt}} \vee S_{i+1,j,t'}$$

se satisfacen cuando la expresión booleana anterior es verdadera. El caso especial de que repetimos una DI de aceptación se expresa como

$$H_{ij} \wedge Q_{ip} \wedge S_{ijt} \wedge H_{i+1,j} \wedge Q_{i+1,p} \wedge S_{i+1,j,t}$$

(Aquí q_p es un estado de aceptación de M. La siguientes cláusulas son verdaderas cuando esta expresión es verdadera:

$$\overline{H_{ij}} \vee \overline{Q_{ip}} \vee \overline{S_{ijt}} \vee H_{i+1,j}$$
$$\overline{H_{ij}} \vee \overline{Q_{ip}} \vee \overline{S_{ijt}} \vee Q_{i+1,p}$$
$$\overline{H_{ij}} \vee \overline{Q_{ip}} \vee \overline{S_{ijt}} \vee S_{i+1,j,t}$$

Añadiremos tales cláusulas a E_w. Obsérvese que el número de cláusulas de E_w depende de $T(n)$ y del número de estados y símbolos de la cinta de M. Puesto que el número de estados y símbolos de la cinta no depende de la longitud de la cadena de entrada, el número de cláusulas es polinómico en $T_{(n)}$, es decir $T'_{(n)}$. Es fácil darse cuenta de que las restricciones que se obtienen a partir de las cláusulas de E_w, fuerzan a que las asignaciones de verdad para las variables booleanas correspondan a computaciones de aceptación de M. Por lo tanto, $w \in L$ si y sólo si E_w es satisfactible.

Obsérvese que tenemos

$$(T(n) + 1)(p + 1) + 2(T(n) + 1)^2 + 2(m + 1)(T(n) + 1)^2$$

variables. Ya que las cláusulas pueden contener una variable o su negación, hay al menos $c(T(n) + 1)^2$ variables distintas y sus negaciones para una constante c elegida apropiadamente. Por tanto, el número máximo de literales que puede haber en una cláusula es $c(T(n) + 1)^2$. De lo que se deduce que el tamaño de E_w es como máximo $T(n)^r c(T(n) + 1)^2$, lo cual estará acotado por algún polinómico en $T(n)$, llamado $T'_{(n)}$. La codificación de cada variable requiere un espacio que estará acotado por el logaritmo del número de variables, ya que codificamos los índices de las variables en binario. Sin embargo, esto requiere que las variables tengan un único índice y, por lo tanto, se deben volver a indexar antes de codificarlas. Esto se realiza en tiempo polinómico en la longitud de E_w, lo que a su vez

es polinómico en $T(n)$. La reducción de L a L_{sat} requiere tiempo polinómico y, de esa forma, L_{sat} es $\mathcal{NP}$-difícil.

Vamos a demostrar el teorema de Cook.

Teorema 7.3.7. **(Cook)** L_{sat} es $\mathcal{NP}$-completo.

Una vez que se ha obtenido un lenguaje $\mathcal{NP}$-completo, la reducción de lenguajes nos proporciona una técnica para demostrar que otros lenguajes son $\mathcal{NP}$-completos.

Lema 7.3.8. Si L_1 es $\mathcal{NP}$-completo y $L_1 <_p L_2$, entonces L_2 es $\mathcal{NP}$-difícil.

Demostración. Supongamos que $L \in \mathcal{NP}$. Entonces, puesto que L_1 es $\mathcal{NP}$-completo, hay una función f_1, computable en tiempo polinómico, de forma que $f_1: L \to L_1$ con la propiedad de que $w \in L$ si y sólo si $f_1(w) \in L_1$. Puesto que $L_1 <_p L_2$, hay también una función computable en tiempo polinómico que es $f_2: L_1 \to L_2$, con lo que $u \in L_1$ si y sólo si $f_2(u) \in L_2$. Obsérvese que $w \in L$ si y sólo si $f_1(w) \in L_1$ si y sólo si $f_2(f_1(w)) \in L_2$. Es más, puesto que f_1 y f_2 son computables en tiempo polinómico en la longitud de sus argumentos, $f_2(f_1(w))$ es computable en tiempo polinómico en la longitud de $f_1(w)$, lo que por medio del Teorema 7.2.9 es polinómico en la longitud de w. Por tanto, la función $f_2(f_1(w))$ compuesta, es computable en tiempo polinómico. De lo que se deduce que $L <_p L_2$, y, por tanto, L_2 es $\mathcal{NP}$-difícil. □

Corolario 7.3.9. Si L_1 es $\mathcal{NP}$-completo y $L_2 \in \mathcal{NP}$ con $L_1 <_p L_2$, entonces L_2 es $\mathcal{NP}$-completo.

Este corolario nos muestra una técnica para obtener otros lenguajes que sean $\mathcal{NP}$-completos. Dado un lenguaje L de $\mathcal{NP}$, si podemos reducir a L, en tiempo polinómico, un lenguaje que sea $\mathcal{NP}$-completo, entonces L también es $\mathcal{NP}$-completo. Cuando nos encontramos con un lenguaje que se sospecha que es $\mathcal{NP}$-completo, podemos aplicar fácilmente este corolario y demostrar que lo es. Otras veces, L puede ser similar a un lenguaje L, que hemos demostrado que es $\mathcal{NP}$-completo. A veces la similitud permite que la demostración de que L' es $\mathcal{NP}$-completo pueda ser adaptada para demostrar lo mismo con respecto a L.

Ejercicios de la Sección 7.3

7.3.1. Hemos tratado las cláusulas codificadas como cadenas sobre el alfabeto $\Sigma = \{$',', '$\vee$', '–', '&', '0', '1'$\}$. Dada una asignación de valores de verdad, ¿se podría codificar de forma que pudiera ser analizada por una máquina de Turing que pudiera evaluar la cláusula codificada?

7.3.2. Describir cómo una máquina de Turing puede evaluar un conjunto de cláusulas en tiempo polinómico.

7.3.3. Describir una técnica para volver a indexar los literales de E_w, en tiempo polinómico en el tamaño de E_w.

7.3.4. Demostrar que, si $T(n)$ es un polinómico en n, entonces la siguiente expresión también lo es:

$$a_k T(n)^k + a_{k-1} T(n)^{k-1} + \ldots + a_1 T(n) + a_0$$

PROBLEMAS

7.1. Hemos mostrado muchos lenguajes que son $\mathcal{NP}$-completos. En este problema vamos a estudiar un lenguaje derivado de un problema llamado 3SAT. 3SAT es un problema de satisfactibilidad como SAT, excepto porque las cláusulas implicadas en el mismo están formadas por tres literales (en SAT no se restringió el número de literales).

Obviamente, una máquina de Turing que acepte L_{sat}, se puede adaptar para que acepte $L_{3\text{sat}}$. La modificación que se necesita consiste en una "submáquina" que comprueba que la muestra codificada está formada por cláusulas que contienen exactamente tres literales. El resto de la máquina L_{sat} no se tiene por qué cambiar. Por tanto, $L_{3\text{sat}} \in \mathcal{NP}$.

Para realizar la reducción en tiempo polinómico de L_{sat} a $L_{3\text{sat}}$, se necesita tomar cadenas que potencialmente pertenecen a L_{sat} y hacer que parezcan cadenas de $L_{3\text{sat}}$. En vez de tratar este problema a nivel de lenguaje, lo vamos a tratar a nivel de cláusula. Es decir, cogeremos una muestra de SAT y la transformaremos en una muestra de 3SAT.

Supongamos que tenemos un conjunto de cláusulas C: es decir, tenemos una muestra de SAT. Si todas las cláusulas están formadas por tres literales, quedará probado. Si todas las cláusulas no están formadas por tres literales, debemos convertir C en C', de forma que todas las cláusulas de C' estén formadas por tres literales, por lo tanto C' es satisfactible sólo si C lo es y, por tanto, cualquier asignación de valores de verdad que satisfaga C', también satisfará C. Supongamos que c es una cláusula de C que no contiene tres literales. Podemos considerar tres casos. La cláusula c puede contener uno, dos, o más de tres literales. Supongamos que c contiene dos literales, entonces $c = x_1 \vee x_2$. Podemos construir dos cláusulas c_1 y c_2 con tres literales a partir de c y con la propiedad de que una asignación de valores de verdad que sea satisfactible para el conjunto $\{c_1, c_2\}$ induce una asignación de valores de verdad para las variables de c, y viceversa. Sea α una nueva variable booleana (la cual no aparece en ninguna de las cláusulas de C), Obsérvese que $c_1 = x_1 \vee x_2 \vee \alpha$ y $c_2 = x_1 \vee x_2 \vee \overline{\alpha}$ son las dos cláusulas que se querían.

1. Supongamos que c es una cláusula que está formada por un único literal. Demostrar cómo se construye un conjunto D de cláusulas que tengan la

propiedad de que ninguna asignación de valores de verdad para D, que lo satisfaga, induce una asignación de valores de verdad para la variable x, y viceversa. *Indicación:* Se puede realizar con cuatro cláusulas.

En el caso de que c sea una clase compuesta por más de tres literales, podemos construir un conjunto de cláusulas que tengan la propiedad deseada. Para ello, supongamos que c es la cláusula $x_1 \vee x_2 \vee \ldots \vee x_k$. Sean $\alpha_1, \alpha_2, \ldots, \alpha_{k-3}$ unas nuevas variables booleanas. Sea D la cláusula formada por $x_1 \vee x_2 \vee \alpha_1$, $x_2 \vee \overline{\alpha}_1 \vee \alpha_2$, $x_3 \vee \overline{\alpha}_2 \vee \overline{\alpha}_3$, ..., $x_{k-1} \vee x_k \vee \overline{\alpha}_{k-3}$. Obsérvese que D cumple la propiedad deseada. Es decir, una asignación de valores de verdad para las variables $x_1, x_2, \ldots, x_k$ que haga que c sea verdadera, induce una asignación de valores de verdad que satisface D, y viceversa.

Por tanto, una colección de cláusulas se puede transformar en una colección de cláusulas que estén formadas por tres literales con la propiedad de que una asignación de valores de verdad que satisface el conjunto original, induce una asignación de valores de verdad para el conjunto transformado y viceversa.

2. Demostrar que la transformación anterior se puede realizar en tiempo polinómico.

3. Demostrar que $L_{3\text{sat}}$ es $\mathcal{NP}$-completo.

Referencias y bibliografía

El material para realizar este libro proviene de muchas fuentes, al igual que la forma en la que se presenta. En este pequeño comentario vamos a mencionar ciertos temas que provienen de fuentes particularmente importantes. En la bibliografía que aparece a continuación, se enumeran muchas de las fuentes que el autor usó para preparar este texto. Sin embargo, no se pretende que este comentario y la bibliografía sean particularmente exhaustivos o completos.

El Problema 1.8 se debe a Salomaa [11]. El Lema 2.8.2 es una modificación de un teorema de Arden [1]. Este lema se usa para desarrollar el Lema 2.8.3, el cual se basa en un trabajo realizado por Brzozowski [2]. Éste también proporciona una técnica elegante para derivar un autómata finito a partir de una expresión regular. El Problema 2.4 procede del Harrison [4]. La Sección 4.3 debe sus orígenes a Lewis y Papadimitriou [7]. Su libro trata de la construcción de máquinas de Turing, más de lo que nosotros lo hemos hecho.

Los libros mencionados anteriormente, así como los libros restantes, se usaron para gran parte del desarrollo y la presentación. Los excelentes libros de Lewis y Papadimitriou [7], Harrison [4] y Hopcroft y Ullman [5] son clásicos en esta materia. Los otros tres son apropiados para ser estudiados después de este libro, como es Wood [13]. El primero de los libros de Hopcroft y Ullman [5] es un compendio sobre lenguajes formales, teoría de autómatas y complejidad computacional, que se desarrollan aún más en [6]. Los libros de Linz [8], Carroll y Long [3], Rayward-Smith [10], Martin [9] y Sudkamp [12] son introducciones sobre este tema y son similares a este libro. Los cinco son una excelente referencia durante la lectura de este libro.

REFERENCIAS

[1] ARDEN, D. N. "Delayed Logic and Finite State Machines", in *Theory of Computing Machine Design*. Ann Arbor, MI: University of Michigan Press, 1960.

[2] BRZOZOWSKI, J. A. "Derivatives of Regular Expression", *Journal of the Association for Computing Machinery*, 11, no. 4 (October 1964), pp. 481-494.

[3] CARROLL, JOHN, and DARRELL LONG. *Theory of Finite Automata with an Introduction to Formal Languages*, Englewood Cliffs, NJ: Prentice Hall, 1989.

[4] HARRISON, MICHAEL. *Introduction to Formal Language Theory*, Addison-Wesley Series in Computer Science. Reading, MA: Addison-Wesley, Inc., 1978.

[5] HOPCROFT, JOHN E., and JEFFREY D. ULLMAN. *Formal Languages and Their Relation to Automata*, Addison-Wesley Series in Computer Science. Reading, MA: Addison-Wesley, Inc., 1969.

[6] HOPCROFT, JOHN E., and JEFFREY D. ULLMAN. *Introduction to Automata Theory and Computation*, Addison-Wesley Series in Computer Science. Reading, MA: Addison-Wesley, Inc., 1979.

[7] LEWIS, H. R., and C. H. PAPADIMITRIOU. *Elements of the Theory of Computation*, Englewood Cliffs, NH: Prentice Hall, 1981.

[8] LINZ, PETER. *An Introduction to Formal Languages and Automata*, Lexington, MA: D. C. Heath and Company, 1990.

[9] MARTIN, JOHN C. *Introduction to Languages and the Theory of Computation*, New York: McGraw-Hill, Inc., 1991.

[10] RAYWARD-SMITH, V. J. *A First Course in Formal Language Theory*, Computer Science Texts Series. Oxford, UK: Blackwell Scientific Publications, 1983.

[11] SALOMAA, ARTO. "Morphisms on Free Monoids and Language Theory", in *Formal Language Theory: Perspectives and Open Problems*, ed. Ronald V. Book. New York: Academic Press, Inc.

[12] SUDKAMP, THOMAS A. *Languages and Machines, An Introduction to the Theory of Computer Science*, Addison-Wesley Series in Computer Science. Reading, MA: Addison-Wesley, Inc., 1988.

[13] WOOD, DERICK. *Theory of Computation*, New York: John Wiley & Sons, Inc., 1987.

Índice analítico

D

E

F

G

K

L

M

Q

R

S

T

U

V

W