

Facultad de Ciencias

notas *de clase*

Teoría de la
Computación
lenguajes, autómatas,
gramáticas.

Rodrigo De Castro



Biología

Estadística

Farmacia

Física

Geología

Instituto de Ciencias Naturales

Matemáticas

Observatorio Astronómico

Química



UNIVERSIDAD
NACIONAL
DE COLOMBIA

Sede Bogotá

Teoría de la Computación

Lenguajes, autómatas, gramáticas

Rodrigo De Castro Korgi

Ph.D. en Matemáticas
University of Illinois, U.S.A.

Departamento de Matemáticas
Universidad Nacional de Colombia, Bogotá

Teoría de la Computación
Lenguajes, autómatas, gramáticas

© UNIVERSIDAD NACIONAL DE COLOMBIA
FACULTAD DE CIENCIAS

Juan Manuel Tejeiro, Decano
Natalia Ruiz, Vicedecana Académica

Gustavo Rubiano, Director de Publicaciones

© Rodrigo De Castro Korgi
Profesor Asociado
Departamento de Matemáticas

Primera edición, 2004
Diagramación en L^AT_EX realizada por el autor

Impresión:
UNIBIBLOS
Universidad Nacional de Colombia
Bogotá D.C., 2004

Índice general

Prólogo	1
Introducción. ¿Qué es la Teoría de la Computación?	3
1. Alfabetos, cadenas y lenguajes	5
1.1. Alfabetos y cadenas	5
1.2. Concatenación de cadenas	7
1.3. Potencias de una cadena	8
1.4. Longitud de una cadena	8
1.5. Reflexión o inversa de una cadena	9
1.6. Subcadenas, prefijos y sufijos	9
1.7. Lenguajes	10
1.8. Operaciones entre lenguajes	11
1.9. Concatenación de lenguajes	12
1.10. Potencias de un lenguaje	14
1.11. La clausura de Kleene de un lenguaje	14
1.12. Reflexión o inverso de un lenguaje	17
1.13. Lenguajes regulares	18
1.14. Expresiones regulares	19
2. Autómatas finitos	25
2.1. Autómatas finitos deterministas (AFD)	25
2.2. Diagrama de transiciones de un autómata	28
2.3. Diseño de autómatas	30
2.4. Autómatas finitos no-deterministas (AFN)	33
2.5. Equivalencia computacional entre los AFD y los AFN	38
2.6. Autómatas con transiciones λ (AFN- λ)	43

2.7. Equivalencia computacional entre los AFN- λ y los AFN	47
2.8. Teorema de Kleene. Parte I	49
2.9. Ejemplos de la parte I del Teorema de Kleene	52
2.10. Lema de Arden	55
2.11. Teorema de Kleene. Parte II	57
2.12. Ejemplos de la parte II del Teorema de Kleene	58
3. Otras propiedades de los lenguajes regulares	63
3.1. Lema de bombeo	63
3.2. Propiedades de clausura	67
3.3. Propiedades de clausura para autómatas	69
3.4. Homomorfismos \bowtie	72
3.5. Imagen inversa de un homomorfismo \bowtie	74
3.6. Algoritmos de decisión	75
4. Lenguajes y gramáticas independientes del contexto	81
4.1. Gramáticas generativas	81
4.2. Gramáticas independientes del contexto	82
4.3. Árbol de una derivación	88
4.4. Gramáticas ambiguas	91
4.5. Gramáticas para lenguajes de programación	94
4.6. Gramáticas para lenguajes naturales \bowtie	96
4.7. Gramáticas regulares	98
4.8. Eliminación de las variables inútiles	102
4.9. Eliminación de las producciones λ	107
4.10. Eliminación de las producciones unitarias	110
4.11. Forma Normal de Chomsky (FNC)	113
4.12. Forma Normal de Greibach (FNG) \bowtie	120
4.13. Lema de bombeo para LIC	125
4.14. Propiedades de clausura de los LIC	130
4.15. Algoritmos de decisión para GIC	135
5. Autómatas con pila	143
5.1. Autómatas con Pila Deterministas (AFPD)	143
5.2. Autómatas con pila no-deterministas (AFPN)	150
5.3. Aceptación por pila vacía	154
5.4. Autómatas con pila y LIC. Parte I.	157
5.5. Autómatas con pila y LIC. Parte II. \bowtie	160

6. Máquinas de Turing	167
6.1. Máquinas de Turing como aceptadoras de lenguajes	167
6.2. Subrutinas o macros	174
6.3. Máquinas de Turing como calculadoras de funciones	176
6.4. Máquinas de Turing como generadoras de lenguajes	179
6.5. Variaciones del modelo estándar de MT	180
6.5.1. Estado de aceptación único	180
6.5.2. Máquina de Turing con cinta dividida en pistas	181
6.5.3. Máquina de Turing con múltiples cintas	181
6.5.4. Máquinas de Turing no-deterministas (MTN)	183
6.6. Simulación de autómatas por medio de máquinas de Turing	186
6.6.1. Simulación de autómatas	186
6.6.2. Simulación de autómatas con pila	186
6.7. Autómatas con dos pilas (AF2P) \oplus	188
6.8. Propiedades de clausura de los lenguajes RE y de los lenguajes recursivos	193
6.9. Máquinas de Turing, computadores, algoritmos y la tesis de Church-Turing	197
6.9.1. Máquinas de Turing y algoritmos	198
6.9.2. Máquinas de Turing y computadores	199
7. Problemas indecidibles	201
7.1. Codificación y enumeración de máquinas de Turing	201
7.2. Máquina de Turing universal	206
7.3. Algoritmos de aceptación para lenguajes RE	209
7.4. Lenguajes que no son RE	211
7.5. Lenguajes RE no recursivos	212
7.6. Problemas indecidibles o irresolubles	215
Bibliografía	221

Prólogo

Este libro contiene lo *mínimo* que los estudiantes de las carreras de ingeniería de sistemas y de matemáticas deberían saber sobre los fundamentos matemáticos de la teoría de la computación. Está basado en el material de clase utilizado por el autor durante los últimos años en la Universidad Nacional de Colombia, sede de Bogotá.

A estudiantes y profesores

El libro está escrito tanto para estudiantes de matemáticas —quienes, es de suponer, tienen más experiencia con razonamientos abstractos y demostraciones— como para estudiantes de ingeniería. Es el profesor quien debe establecer el tono del curso, enfatizando ya sea el rigor matemático o una presentación más intuitiva y práctica. Los resultados están presentados en forma de teoremas, corolarios y lemas, con sus respectivas demostraciones; éstas pueden omitirse, si así lo estima el profesor. En los cursos dirigidos a estudiantes de ingeniería de sistemas, el énfasis debe residir —tanto por parte del profesor como por parte del estudiante— en los ejemplos y ejercicios prácticos; hay que resaltar más el *significado* de los enunciados que sus demostraciones formales. El libro contiene gran cantidad de ejemplos y problemas resueltos, con aplicaciones o ilustraciones directas de la teoría.

Como prerequisito, es imprescindible que el estudiante haya tomado al menos un curso de matemáticas discretas en el que se haya familiarizado con las nociones básicas y la notación de la teoría intuitiva de conjuntos, grafos, inducción matemática y lógica elemental. La experiencia previa en programación es muy útil pero, de ninguna manera, necesaria.

El material se presenta en secciones relativamente cortas, lo que permite alguna flexibilidad en la selección de los tópicos del curso. Así, si el tiempo

disponible no es holgado, o no es posible avanzar con la velocidad suficiente, podrían suprimirse las secciones demarcadas con el símbolo **✗**.

Casi todas las secciones poseen ejercicios, de variada dificultad; los más difíciles están precedidos de un símbolo de admiración ! y podrían ser considerados opcionales. Es responsabilidad del estudiante resolver los ejercicios que sean asignados por el profesor. La única manera de aprender y asimilar las ideas y técnicas presentadas en la clase es trabajar seria y completamente los ejercicios.

Material de apoyo en la red

Versiones preliminares de estas notas aparecieron, de forma incompleta, en el curso virtual de Teoría de la Computación perteneciente al programa *Universidad Virtual* de la Universidad Nacional de Colombia. Es la intención del autor mantener y actualizar permanentemente la versión virtual interactiva de este curso, con material de apoyo como temas y ejercicios nuevos, corrección de errores, software, enlaces a otra páginas Web, etc. Se puede acceder libremente al curso virtual en el portal

<http://www.virtual.unal.edu.co/>

siguiendo los enlaces: Cursos–Facultad de Ciencias–Matemáticas–Teoría de la Computación.

Agradecimientos

Durante la elaboración de estas notas he recibido por parte de estudiantes atentos muchas observaciones útiles que han ayudado a mejorar sustancialmente la presentación. Quiero expresarles mis agradecimientos a todos ellos, demasiado numerosos para mencionarlos individualmente.

La primera versión del curso virtual fue realizada con la ayuda del estudiante de posgrado Adolfo Reyes, a quien expreso mi gratitud y reconocimiento. Para la preparación de la presente versión tuve la suerte de contar con la colaboración del estudiante de matemáticas Camilo Cubides, con quien estoy muy agradecido por la calidad y seriedad de su trabajo.

Finalmente, quiero agradecer a Gustavo Rubiano, Director de las oficina de publicaciones de la Facultad de Ciencias, por su continuo apoyo y su cooperación desinteresada.

Introducción

¿Qué es la Teoría de la Computación?

La Teoría de la Computación estudia modelos abstractos de los dispositivos concretos que conocemos como computadores, y analiza lo que se puede y no se puede hacer con ellos. Este estudio teórico se inició varias décadas antes de la aparición de los primeros computadores reales y continúa creciendo, a medida que la computación incrementa su sofisticación.

Entre los muchos tópicos que conforman la teoría de la computación, sólo tendremos la oportunidad de tratar someramente los dos siguientes:

Modelos de computación. Las investigaciones en este campo comenzaron en la década de los 30 del siglo XX con el trabajo del lógico norteamericano Alonzo Church (1903–1995) y del matemático británico Alan Turing (1912–1954). Church introdujo el formalismo conocido como cálculo- λ y enunció la tesis —hoy conocida como tesis de Church— de que las funciones efectivamente computables, es decir, computables por *cualquier* método computacional concebible, son exactamente las funciones λ -computables. En contraste con el enfoque más abstracto de Church, Turing (quien fue alumno doctoral de Church en la universidad de Princeton) propuso un modelo concreto de máquina computadora, hoy conocida como la *máquina de Turing*, capaz de simular las acciones de *cualquier* otro dispositivo físico de computación secuencial.

Curiosamente, las propuestas de Church y Turing se publicaron exactamente en el mismo año: 1936. Los dos formalismos resultaron ser equivalentes y, desde entonces, se han propuesto muchos otros modelos de computación. Como todos han resultado ser equivalentes entre sí, ha ganado

aceptación universal la tesis de Church-Turing: no hay modelo de computación más general ni poderoso que la máquina de Turing.

En los años 40 y 50 del siglo XX se adelantaron investigaciones sobre máquinas de Turing con capacidad restringida, surgiendo así la noción de *máquina de estado finito* o *autómata finito* (“autómata” es sinónimo de “máquina de cómputo automático”). Los autómatas han resultado ser modelos muy útiles para el diseño de diversos tipos de software y hardware.

Lenguajes y gramáticas formales. Una línea investigativa, aparentemente alejada de los modelos de computación, surgió con los estudios del lingüista norteamericano Noam Chomsky¹. Chomsky introdujo en 1956 la noción de *gramática generativa* con el propósito de describir los lenguajes naturales como el español, el inglés, el francés, etc. Chomsky clasificó las gramáticas en cuatro tipos, dependiendo de la forma de sus *producciones*, que son las reglas que utiliza una gramática para generar palabras o cadenas de símbolos. Pocos años después se estableció que hay una estrecha relación entre autómatas y gramáticas: los lenguajes de la llamada *jerarquía de Chomsky* corresponden a los lenguajes que pueden ser reconocidos por tipos especiales de autómatas.

La interacción entre los autómatas (mecanismos para *procesar* cadenas de símbolos) y las gramáticas (mecanismos para *generar* cadenas de símbolos) es una fuente de resultados profundos y significativos. Desde la aparición de los influyentes textos de Hopcroft y Ullman ([HU1], 1969) y ([HU2], 1979), un curso semestral básico de teoría de la computación se ha centrado en el estudio de autómatas y gramáticas. Estas notas de clase reflejan esa tradición.

¹En el año 2002, la Universidad Nacional de Colombia otorgó el doctorado *Honoris Causa* a Noam Chomsky.

Alfabetos, cadenas y lenguajes

De manera muy amplia podría decirse que la computación es la manipulación de secuencias de símbolos. Pero el número de símbolos disponibles en cualquier mecanismo de cómputo es finito y todos los objetos usados como entradas o salidas (inputs/outputs) deben ser identificados en un tiempo finito. Desde el punto de vista teórico esto impone dos restricciones básicas: el conjunto de símbolos (alfabeto) debe ser finito y se deben considerar únicamente cadenas (secuencias de símbolos) de longitud finita. Surgen así los ingredientes esenciales de una teoría abstracta de la computación: *alfabetos* y *cadenas*. Los conjuntos de cadenas (ya sean finitos o infinitos) se denominarán *lenguajes*.

1.1. Alfabetos y cadenas

Un **alfabeto** es un conjunto finito no vacío cuyos elementos se llaman **símbolos**. Denotamos un alfabeto arbitrario con la letra Σ .

Una **cadena** o **palabra** sobre un alfabeto Σ es cualquier sucesión (o secuencia) finita de elementos de Σ . Admitimos la existencia de una única cadena que no tiene símbolos, la cual se denomina **cadena vacía** y se denota con λ . La cadena vacía desempeña, en la teoría de la computación, un papel similar al del conjunto vacío \emptyset en la teoría de conjuntos.

Ejemplo Sea $\Sigma = \{a, b\}$ el alfabeto que consta de los dos símbolos a y b . Las siguientes son cadenas sobre Σ :

aba
 $ababaaa$
 $aaaab.$

Obsérvese que $aba \neq aab$. El orden de los símbolos en una cadena es significativo ya que las cadenas se definen como *sucesiones*, es decir, conjuntos *secuencialmente ordenados*.

Ejemplo El alfabeto $\Sigma = \{0, 1\}$ se conoce como *alfabeto binario*. Las cadenas sobre este alfabeto son secuencias finitas de ceros y unos, llamadas *secuencias binarias*, tales como

$$\begin{array}{l} 001 \\ 1011 \\ 00100001. \end{array}$$

Ejemplo $\Sigma = \{a, b, c, \dots, x, y, z, A, B, C, \dots, X, Y, Z\}$, el alfabeto del idioma castellano. Las palabras oficiales del castellano (las que aparecen en el diccionario DRA) son cadenas sobre Σ .

Ejemplo El alfabeto utilizado por muchos de los llamados *lenguajes de programación* (como Pascal o C) es el conjunto de caracteres ASCII (o un subconjunto de él) que incluye, por lo general, las letras mayúsculas y minúsculas, los símbolos de puntuación y los símbolos matemáticos disponibles en los teclados estándares.

El conjunto de *todas* las cadenas sobre un alfabeto Σ , incluyendo la cadena vacía, se denota por Σ^* .

Ejemplo Sea $\Sigma = \{a, b, c\}$, entonces

$$\Sigma^* = \{\lambda, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, abc, baa, \dots\}.$$

En la siguiente tabla aparece la notación corrientemente utilizada en la teoría de la computación. De ser necesario, se emplean subíndices.

Notación usada en la teoría de la computación	
Σ, Γ	denotan alfabetos.
Σ^*	denota el conjunto de todas las cadenas que se pueden formar con los símbolos del alfabeto Σ .
a, b, c, d, e, \dots	denotan símbolos de un alfabeto.
u, v, w, x, y, z, \dots	denotan cadenas, es decir, sucesiones finitas de símbolos de un alfabeto.
$\alpha, \beta, \gamma, \dots$	denotan lenguajes (definidos más adelante).
λ	denota la cadena vacía, es decir, la única cadena que no tiene símbolos.
$A, B, C, \dots, L, M, N, \dots$	denotan lenguajes (definidos más adelante).

- ☞ Algunos autores denotan la cadena vacía con la letra griega ε . Preferimos denotarla con λ porque ε tiende a confundirse con el símbolo \in usado para la relación de pertenencia.
- ☞ Si bien un alfabeto Σ es un conjunto finito, Σ^* es siempre un conjunto infinito (enumerable). En el caso más simple, Σ contiene solo un símbolo, $\Sigma = \{a\}$, y $\Sigma^* = \{\lambda, a, aa, aaa, aaaa, aaaaa, \dots\}$.
- ☞ Hay que distinguir entre los siguientes cuatro objetos, que son todos diferentes entre sí: \emptyset , λ , $\{\emptyset\}$ y $\{\lambda\}$.
- ☞ La mayor parte de la teoría de la computación se hace con referencia a un alfabeto Σ fijo (pero arbitrario).

1.2. Concatenación de cadenas

Dado un alfabeto Σ y dos cadenas $u, v \in \Sigma^*$, la **concatenación de u y v** se denota como $u \cdot v$ o simplemente uv y se define descriptivamente así:

1. Si $v = \lambda$, entonces $u \cdot \lambda = \lambda \cdot u = u$. Es decir, la concatenación de cualquier cadena u con la cadena vacía, a izquierda o a derecha, es igual a u .
2. Si $u = a_1a_2 \cdots a_n$, $v = b_1b_2 \cdots b_m$, entonces

$$u \cdot v = a_1a_2 \cdots a_n b_1 b_2 \cdots b_m.$$

Es decir, $u \cdot v$ es la cadena formada escribiendo los símbolos de u y a continuación los símbolos de v .

La concatenación de cadenas se puede definir inductiva o recursivamente de la siguiente manera. Si $u, v \in \Sigma^*$, $a \in \Sigma$, entonces

1. $u \cdot \lambda = \lambda \cdot u = u$.
2. $u \cdot (va) = (u \cdot v)a$.

Propiedad. *La concatenación de cadenas es una operación asociativa. Es decir, si $u, v, w \in \Sigma^*$, entonces*

$$(uv)w = u(vw).$$

Demostración. Se puede hacer escribiendo explícitamente las cadenas u , v , w y usando la definición descriptiva de concatenación. También se puede dar una demostración inductiva usando la definición recursiva de concatenación (ejercicio opcional). □

1.3. Potencias de una cadena

Dada $u \in \Sigma^*$ y $n \in \mathbb{N}$, se define (descriptivamente) u^n en la siguiente forma

$$\begin{aligned} u^0 &= \lambda, \\ u^n &= \underbrace{uu \cdots u}_{n \text{ veces}}. \end{aligned}$$

Como ejercicio, el estudiante puede dar una definición recursiva de u^n .

1.4. Longitud de una cadena

La **longitud** de una cadena $u \in \Sigma^*$ se denota $|u|$ y se define como el número de símbolos de u (contando los símbolos repetidos). Es decir,

$$|u| = \begin{cases} 0, & \text{si } u = \lambda, \\ n, & \text{si } u = a_1 a_2 \cdots a_n. \end{cases}$$

Ejemplo $|aba| = 3$, $|baaa| = 4$.

Ejemplo Si $w \in \Sigma^*$, $n, m \in \mathbb{N}$, demostrar que $|w^{n+m}| = |w^n| + |w^m|$. Esta no es una propiedad realmente importante (¡no la usaremos nunca en este libro!); la presentamos aquí para enfatizar los conceptos involucrados e ilustrar los razonamientos estrictos.

Solución. Caso $n, m \geq 1$. $|w^{n+m}| = |\underbrace{ww \cdots w}_{n+m \text{ veces}}| = (n+m)|w|$. Por otro lado,

$$|w^n| + |w^m| = |\underbrace{ww \cdots w}_{n \text{ veces}}| + |\underbrace{ww \cdots w}_{m \text{ veces}}| = n|w| + m|w|.$$

Caso $n = 0, m \geq 1$. $|w^{n+m}| = |w^{0+m}| = |w^m|$. Por otro lado,

$$|w^n| + |w^m| = |w^0| + |w^m| = |\lambda| + |w^m| = 0 + |w^m| = |w^m|.$$

Caso $m = 0, n \geq 1$. Similar al caso anterior.

Caso $n = 0, m = 0$. $|w^{n+m}| = |w^{0+0}| = |\lambda| = 0$. Por otro lado,

$$|w^n| + |w^m| = |w^0| + |w^0| = |\lambda| + |\lambda| = 0 + 0 = 0.$$

1.5. Reflexión o inversa de una cadena

La **reflexión** o **inversa** de una cadena $u \in \Sigma^*$ se denota u^R y se define descriptivamente así:

$$u^R = \begin{cases} \lambda, & \text{si } u = \lambda, \\ a_n \cdots a_2 a_1, & \text{si } u = a_1 a_2 \cdots a_n. \end{cases}$$

De la definición se observa claramente que la reflexión de la reflexión de una cadena es la misma cadena, es decir,

$$(u^R)^R = u, \quad \text{para } u \in \Sigma^*.$$

Algunos autores escriben u^{-1} en lugar de u^R para denotar la reflexión de una cadena u .

Ejercicios de la sección 1.5

- ① Dar una definición recursiva de u^R .
- ② Si $u, v \in \Sigma^*$, demostrar que $(uv)^R = v^R u^R$. Generalizar esta propiedad a la concatenación de n cadenas.

1.6. Subcadenas, prefijos y sufijos

Una cadena v es una **subcadena** o una **subpalabra** de u si existen cadenas x, y tales que $u = xvy$. Nótese que x o y pueden ser λ y, por lo tanto, la cadena vacía es una subcadena de cualquier cadena y toda cadena es subcadena de sí misma.

Un **prefijo** de u es una cadena v tal que $u = vw$ para alguna cadena $w \in \Sigma^*$. Se dice que v es un **prefijo propio** si $v \neq u$.

Similarmente, un **sufijo** de u es una cadena v tal que $u = wv$ para alguna cadena $w \in \Sigma^*$. Se dice que v es un **sufijo propio** si $v \neq u$.

Obsérvese que λ es un prefijo y un sufijo de toda cadena u ya que $u\lambda = \lambda u = u$. Por la misma razón, toda cadena u es prefijo y sufijo de sí misma.

Ejemplo Sean $\Sigma = \{a, b, c, d\}$ y $u = bcbaadb$.

Prefijos de u :

λ
 b
 bc
 bcb
 $bcba$
 $bcbaa$
 $bcbaad$
 $bcbaadb$

Sufijos de u :

λ
 b
 db
 adb
 $aadb$
 $baadb$
 $cbaadb$
 $bcbaadb$

1.7. Lenguajes

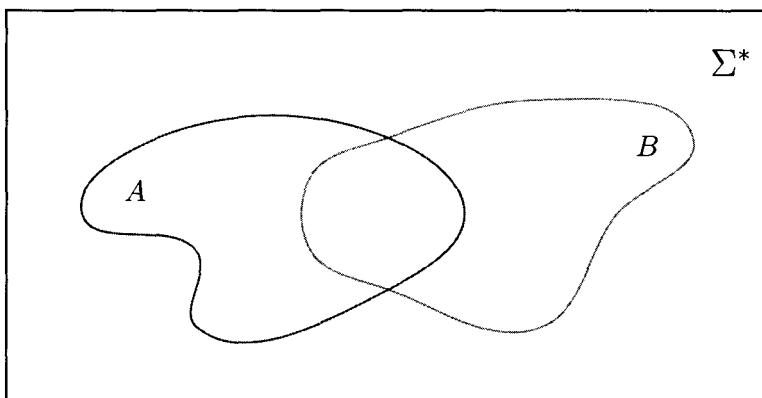
Un lenguaje L sobre un alfabeto Σ es un subconjunto de Σ^* , es decir $L \subseteq \Sigma^*$.

Casos extremos:

$L = \emptyset$, lenguaje vacío.

$L = \Sigma^*$, lenguaje de todas las cadenas sobre Σ .

Todo lenguaje L satisface $\emptyset \subseteq L \subseteq \Sigma^*$, y puede ser finito o infinito. Los lenguajes se denotan con letras mayúsculas $A, B, C, \dots, L, M, N, \dots$. En la siguiente gráfica se visualizan dos lenguajes A y B sobre Σ .



Ejemplos Los siguientes son ejemplos de lenguajes sobre los alfabetos especificados.

- $\Sigma = \{a, b, c\}$. $L = \{a, aba, aca\}$.
- $\Sigma = \{a, b, c\}$. $L = \{a, aa, aaa, \dots\} = \{a^n : n \geq 1\}$.

- $\Sigma = \{a, b, c\}$. $L = \{\lambda, aa, aba, ab^2a, ab^3a, \dots\} = \{ab^n a : n \geq 0\} \cup \{\lambda\}$.
- $\Sigma = \{a, b, c, \dots, x, y, z, A, B, C, \dots, X, Y, Z\}$. $L = \{u \in \Sigma^* : u \text{ aparece en el diccionario español DRA}\}$. L es un lenguaje finito.
- $\Sigma = \{a, b, c\}$. $L = \{u \in \Sigma^* : u \text{ no contiene el símbolo } c\}$. Por ejemplo, $abbaab \in L$ pero $abbcaa \notin L$.
- $\Sigma = \{0, 1\}$. L = conjunto de todas las secuencias binarias que contienen un número impar de unos.
- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. El conjunto \mathbb{N} de los números naturales se puede definir como un lenguaje sobre Σ , en la siguiente forma:

$$\mathbb{N} = \{u \in \Sigma^* : u = 0 \text{ ó } 0 \text{ no es un prefijo de } u\}.$$

Como ejercicio, el estudiante puede definir el conjunto de los enteros $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ como un lenguaje sobre un alfabeto adecuado.

-  El concepto abstracto de “lenguaje”, tal como se ha definido, no es exactamente la misma noción utilizada en la expresión “lenguaje de programación”. Para precisar la relación entre estos conceptos, consideremos el alfabeto Σ de los caracteres ASCII. Un programa en C o en Pascal, por ejemplo, es simplemente una cadena de símbolos de Σ y, por lo tanto, un conjunto de programas es un lenguaje (en el sentido formal definido en esta sección).

1.8. Operaciones entre lenguajes

Puesto que los lenguajes sobre Σ son subconjuntos de Σ^* , las operaciones usuales entre conjuntos son también operaciones válidas entre lenguajes. Así, si A y B son lenguajes sobre Σ (es decir $A, B \subseteq \Sigma^*$), entonces los siguientes también son lenguajes sobre Σ :

$A \cup B$	Unión
$A \cap B$	Intersección
$A - B$	Diferencia
$\bar{A} = \Sigma^* - A$	Complemento

Estas operaciones entre lenguajes se llaman *operaciones conjuntistas* o *booleanas* para distinguirlas de las *operaciones lingüísticas* (concatenación, potencia, inverso, clausura) que son extensiones a los lenguajes de las operaciones entre cadenas.

1.9. Concatenación de lenguajes

La **concatenación** de dos lenguajes A y B sobre Σ , notada $A \cdot B$ o simplemente AB se define como

$$AB = \{uv : u \in A, v \in B\}.$$

En general, $AB \neq BA$.

Ejemplo Si $\Sigma = \{a, b, c\}$, $A = \{a, ab, ac\}$, $B = \{b, b^2\}$, entonces

$$AB = \{ab, ab^2, ab^2, ab^3, acb, acb^2\}.$$

$$BA = \{ba, bab, bac, b^2a, b^2ab, b^2ac\}.$$

Ejemplo Si $\Sigma = \{a, b, c\}$, $A = \{ba, bc\}$, $B = \{b^n : n \geq 0\}$, entonces

$$AB = \{bab^n : n \geq 0\} \cup \{bcb^n : n \geq 0\}.$$

$$BA = \{b^nba : n \geq 0\} \cup \{b^nb^c : n \geq 0\}$$

$$= \{b^{n+1}a : n \geq 0\} \cup \{b^{n+1}c : n \geq 0\}$$

$$= \{b^n a : n \geq 1\} \cup \{b^n c : n \geq 1\}.$$

Propiedades de la concatenación de lenguajes. Sean A, B, C lenguajes sobre Σ , es decir $A, B, C \subseteq \Sigma^*$. Entonces

1. $A \cdot \emptyset = \emptyset \cdot A = \emptyset$.
2. $A \cdot \{\lambda\} = \{\lambda\} \cdot A = A$.
3. Propiedad Asociativa,

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

4. Distributividad de la concatenación con respecto a la unión,

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

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

5. Propiedad distributiva generalizada. Si $\{B_i\}_{i \in I}$ es una familia cualquiera de lenguajes sobre Σ , entonces

$$A \cdot \bigcup_{i \in I} B_i = \bigcup_{i \in I} (A \cdot B_i),$$

$$\left(\bigcup_{i \in I} B_i \right) \cdot A = \bigcup_{i \in I} (B_i \cdot A).$$

Demostración.

1. $A \cdot \emptyset = \{uv : u \in A, v \in \emptyset\} = \emptyset$.
2. $A \cdot \{\lambda\} = \{uv : u \in A, v \in \{\lambda\}\} = \{u : u \in A\} = A$.
3. Se sigue de la asociatividad de la concatenación de cadenas.
4. Caso particular de la propiedad general, demostrada a continuación.
5. Demostración de la igualdad $A \cdot \bigcup_{i \in I} B_i = \bigcup_{i \in I} (A \cdot B_i)$:

$$\begin{aligned} x \in A \cdot \bigcup_{i \in I} B_i &\iff x = u \cdot v, \quad \text{con } u \in A \text{ & } v \in \bigcup_{i \in I} B_i \\ &\iff x = u \cdot v, \quad \text{con } u \in A \text{ & } v \in B_j, \text{ para algún } j \in I \\ &\iff x \in A \cdot B_j, \quad \text{para algún } j \in I \\ &\iff x \in \bigcup_{i \in I} (A \cdot B_i). \end{aligned}$$

La igualdad $\left(\bigcup_{i \in I} B_i\right) \cdot A = \bigcup_{i \in I} (B_i \cdot A)$ se demuestra de forma similar. \square

- ☞ La propiedad asociativa permite escribir concatenaciones de tres o más lenguajes sin necesidad de usar paréntesis.
- ☞ En general, no se cumple que $A \cdot (B \cap C) = A \cdot B \cap A \cdot C$. Es decir, la concatenación no es distributiva con respecto a la intersección. Contraejemplo: $A = \{a, \lambda\}$, $B = \{\lambda\}$, $C = \{a\}$. Se tiene:

$$A \cdot (B \cap C) = \{a, \lambda\} \cdot \emptyset = \emptyset.$$

Por otro lado,

$$A \cdot B \cap A \cdot C = \{a, \lambda\} \cdot \{\lambda\} \cap \{a, \lambda\} \cdot \{a\} = \{a, \lambda\} \cap \{a^2, a\} = \{a\}.$$

Ejercicios de la sección 1.9

- ① Dar un ejemplo de un alfabeto Σ y dos lenguajes diferentes A, B sobre Σ tales que $AB = BA$.
- ② Una de las dos contenencias siguientes es verdadera y la otra es falsa. Demostrar o refutar, según sea el caso:
 - (i) $A \cdot (B \cap C) \subseteq A \cdot B \cap A \cdot C$.
 - (ii) $A \cdot B \cap A \cdot C \subseteq A \cdot (B \cap C)$.

1.10. Potencias de un lenguaje

Dado un lenguaje A sobre Σ , ($A \subseteq \Sigma^*$), y un número natural $n \in \mathbb{N}$, se define A^n en la siguiente forma

$$A^0 = \{\lambda\},$$

$$A^n = \underbrace{AA \cdots A}_{n \text{ veces}} = \{u_1 \cdots u_n : u_i \in A, \text{ para todo } i, 1 \leq i \leq n\}.$$

De esta forma, A^2 es el conjunto de las concatenaciones dobles de cadenas de A , A^3 está formado por las concatenaciones triples y, en general, A^n es el conjunto de todas las concatenaciones de n cadenas de A , de todas las formas posibles. Como ejercicio, el estudiante puede dar una definición recursiva de A^n .

1.11. La clausura de Kleene de un lenguaje

La clausura de Kleene o estrella de Kleene o simplemente la **estrella** de un lenguaje A , $A \subseteq \Sigma^*$, es la unión de todas las potencias de A y se denota por A^* .

(Descripción 1)

$$A^* = \bigcup_{i \geq 0} A^i = A^0 \cup A^1 \cup A^2 \cup \cdots \cup A^n \cdots$$

Según la definición de las potencias de una lenguaje, A^* consta de todas las concatenaciones de cadenas de A consigo mismas, de todas las formas posibles. Tenemos así una útil descripción de A^* :

(Descripción 2)

$\begin{aligned} A^* &= \text{conjunto de } \textit{todas} \text{ las concatenaciones} \\ &\quad \text{de cadenas de } A, \text{ incluyendo } \lambda \\ &= \{u_1 \cdots u_n : u_i \in A, n \geq 0\} \end{aligned}$

De manera similar se define la **clausura positiva** de un lenguaje A , $A \subseteq \Sigma^*$, denotada por A^+ .

$$A^+ = \bigcup_{i \geq 1} A^i = A^1 \cup A^2 \cup \cdots \cup A^n \cdots$$

A^+ se puede describir de la siguiente manera

$\begin{aligned} A^+ &= \text{conjunto de } \textit{todas} \text{ las concatenaciones de cadenas de } A \\ &= \{u_1 \cdots u_n : u_i \in A, n \geq 1\} \end{aligned}$

Obsérvese que $A^* = A^+ \cup \{\lambda\}$ y que $A^* = A^+$ si y solamente si $\lambda \in A$.

Propiedades de * y +. Sea A un lenguaje sobre Σ , es decir, $A \subseteq \Sigma^*$.

1. $A^+ = A^* \cdot A = A \cdot A^*$.
2. $A^* \cdot A^* = A^*$.
3. $(A^*)^n = A^*$, para todo $n \geq 1$.
4. $(A^*)^* = A^*$.
5. $A^+ \cdot A^+ \subseteq A^+$.
6. $(A^*)^+ = A^*$.
7. $(A^+)^* = A^*$.
8. $(A^+)^+ = A^+$.
9. Si A y B son lenguajes sobre Σ^* , entonces $(A \cup B)^* = (A^* B^*)^*$.

Demostración.

$$\begin{aligned} 1. \quad A \cdot A^* &= A \cdot (A^0 \cup A^1 \cup A^2 \cup \dots) \\ &= A^1 \cup A^2 \cup A^3 \cup \dots \\ &= A^+. \end{aligned}$$

Similarmente se demuestra que $A^* \cdot A = A^+$.

2. Si $x \in A^* \cdot A^*$, entonces $x = u \cdot v$, con $u \in A^*$, $v \in A^*$. De modo que, $x = u \cdot v$, con $u = u_1 u_2 \cdots u_n$, $u_i \in A$, $n \geq 0$ y $v = v_1 v_2 \cdots v_m$, $v_i \in A$, $m \geq 0$.

De donde

$$x = u \cdot v = u_1 \cdot u_2 \cdots u_n \cdot v_1 \cdot v_2 \cdots v_m.$$

con $u_i \in A$, $v_i \in A$, $n \geq 0$. Por lo tanto, x es una concatenación de $n+m$ cadenas de A . Así que $x \in A^*$.

Recíprocamente, si $x \in A^*$, entonces $x = x \cdot \lambda \in A^* \cdot A^*$. Esto prueba la igualdad de los conjuntos $A^* \cdot A^*$ y A^* .

3. Se sigue de la propiedad anterior.

$$\begin{aligned} 4. \quad (A^*)^* &= (A^*)^0 \cup (A^*)^1 \cup (A^*)^2 \cup \dots \\ &= \{\lambda\} \cup A^* \cup A^* \cup A^* \cup \dots \\ &= A^*. \end{aligned}$$

5. La demostración de esta propiedad es similar a la de la propiedad 2, pero con la restricción $m, n \geq 1$. En general, no se tiene la igualdad $A^+ \cdot A^+ = A^+$; más adelante se mostrará un contraejemplo.

$$\begin{aligned} 6. \quad (A^*)^+ &= (A^*)^1 \cup (A^*)^2 \cup (A^*)^3 \cup \dots \\ &= A^* \cup A^* \cup A^* \cup \dots \\ &= A^*. \end{aligned}$$

$$\begin{aligned} 7. \quad (A^+)^* &= (A^+)^0 \cup (A^+)^1 \cup (A^+)^2 \cup \dots \\ &= \{\lambda\} \cup A^+ \cup A^+ A^+ \cup \dots \\ &= A^* \cup (\text{conjuntos contenidos en } A^+) \\ &= A^*. \end{aligned}$$

$$\begin{aligned} 8. \quad (A^+)^+ &= (A^+)^1 \cup (A^+)^2 \cup (A^+)^3 \cup \dots, \\ &= A^+ \cup (\text{conjuntos contenidos en } A^+) \\ &= A^+. \end{aligned}$$

9. Según la Descripción 2, el lenguaje $(A \cup B)^*$ está formado por las concatenaciones de cadenas de A consigo mismas, las concatenaciones de cadenas de B consigo mismas y las concatenaciones de cadenas de A con cadenas de B , de todas las formas posibles y en cualquier orden. Si se usa también la Descripción 2, se observa que eso mismo se obtiene al efectuar $(A^* B^*)^*$. \square

Contraejemplo de $A^+ \cdot A^+ = A^+$. Sea $\Sigma = \{a, b\}$, $A = \{a\}$. Se tiene

$$A^+ = A^1 \cup A^2 \cup \dots = \{a\} \cup \{aa\} \cup \{aaa\} \cup \dots = \{a^n : n \geq 1\}.$$

Por otro lado,

$$\begin{aligned} A^+ \cdot A^+ &= \{a, a^2, a^3, \dots\} \cdot \{a, a^2, a^3, \dots\} = \{a^2, a^3, a^4, \dots\} \\ &= \{a^n : n \geq 2\}. \end{aligned}$$

Según las definiciones dadas, Σ^* tiene dos significados:

Σ^* = conjunto de las cadenas sobre el alfabeto Σ .

Σ^* = conjunto de todas las concatenaciones de cadenas de Σ .

No hay conflicto de notaciones porque las dos definiciones anteriores de Σ^* dan lugar al mismo conjunto.

1.12. Reflexión o inverso de un lenguaje

Dado A un lenguaje sobre Σ , se define A^R de la siguiente forma:

$$A^R = \{u^R : u \in A\}.$$

A^R se denomina la **reflexión** o el **inverso** de A .

Propiedades. Sean A y B lenguajes sobre Σ (es decir, $A, B \subseteq \Sigma^*$).

1. $(A \cdot B)^R = B^R \cdot A^R$.
2. $(A \cup B)^R = A^R \cup B^R$.
3. $(A \cap B)^R = A^R \cap B^R$.
4. $(A^R)^R = A$.
5. $(A^*)^R = (A^R)^*$.
6. $(A^+)^R = (A^R)^+$.

Demostración. Demostraremos las propiedades 1 y 5; las demás se dejan como ejercicio para el estudiante.

1. $x \in (A \cdot B)^R \iff x = u^R, \text{ donde } u \in A \cdot B$
 $\iff x = u^R, \text{ donde } u = vw, v \in A, w \in B$
 $\iff x = (vw)^R, \text{ donde } v \in A, w \in B$
 $\iff x = w^R v^R, \text{ donde } v \in A, w \in B$
 $\iff x \in B^R \cdot A^R.$
5. $x \in (A^*)^R \iff x = u^R, \text{ donde } u \in A^*$
 $\iff x = (u_1 \cdot u_2 \cdots u_n)^R, \text{ donde los } u_i \in A, n \geq 0$
 $\iff x = u_n^R \cdot u_{n-1}^R \cdots u_1^R, \text{ donde los } u_i \in A, n \geq 0$
 $\iff x \in (A^R)^*.$

Ejercicios de la sección 1.12

- ① Demostrar las propiedades 2, 3, 4 y 6 de la reflexión de cadenas.
- ② ¿Se pueden generalizar las propiedades 2 y 3 anteriores para uniones e intersecciones arbitrarias, respectivamente?

1.13. Lenguajes regulares

Los **lenguajes regulares** sobre un alfabeto dado Σ son todos los lenguajes que se pueden formar a partir de los lenguajes básicos \emptyset , $\{\lambda\}$, $\{a\}$, $a \in \Sigma$, por medio de las operaciones de unión, concatenación y estrella de Kleene.

A continuación presentamos una definición recursiva de los lenguajes regulares. Sea Σ un alfabeto.

1. \emptyset , $\{\lambda\}$ y $\{a\}$, para cada $a \in \Sigma$, son lenguajes regulares sobre Σ . Estos son los denominados lenguajes regulares básicos.
2. Si A y B son lenguajes regulares sobre Σ , también lo son

$$\begin{array}{ll} A \cup B & (\text{unión}), \\ A \cdot B & (\text{concatenación}), \\ A^* & (\text{estrella de Kleene}). \end{array}$$

Obsérvese que tanto Σ como Σ^* son lenguajes regulares sobre Σ . La unión, la concatenación y la estrella de Kleene se denominan **operaciones regulares**.

Ejemplos Sea $\Sigma = \{a, b\}$. Los siguientes son lenguajes regulares sobre Σ :

1. El lenguaje A de todas las cadenas que tienen exactamente una a :

$$A = \{b\}^* \cdot \{a\} \cdot \{b\}^*.$$

2. El lenguaje B de todas las cadenas que comienzan con b :

$$B = \{b\} \cdot \{a, b\}^*.$$

3. El lenguaje C de todas las cadenas que contienen la cadena ba :

$$C = \{a, b\}^* \cdot \{ba\} \cdot \{a, b\}^*.$$

4. $(\{a\} \cup \{b\}^*) \cdot \{a\}$.

5. $[(\{a\}^* \cup \{b\}^*) \cdot \{b\}]^*$.

Es importante observar que *todo lenguaje finito $L = \{w_1, w_2, \dots, w_n\}$ es regular* ya que L se puede obtener con uniones y concatenaciones:

$$L = \{w_1\} \cup \{w_2\} \cup \dots \cup \{w_n\},$$

y cada w_i es la concatenación de un número finito de símbolos, $w_i = a_1 a_2 \dots a_k$; por lo tanto, $\{w_i\} = \{a_1\} \cdot \{a_2\} \dots \{a_k\}$.

1.14. Expresiones regulares

Con el propósito de simplificar la descripción de los lenguajes regulares se definen las llamadas expresiones regulares.

La siguiente es la definición recursiva de las **expresiones regulares** sobre un alfabeto Σ dado.

1. Expresiones regulares básicas:

- \emptyset es una expresión regular que representa al lenguaje \emptyset .
- λ es una expresión regular que representa al lenguaje $\{\lambda\}$.
- a es una expresión regular que representa al lenguaje $\{a\}$, $a \in \Sigma$.

2. Si R y S son expresiones regulares sobre Σ , también lo son:

$$\begin{aligned}(R)(S) \\ (R \cup S) \\ (R)^*\end{aligned}$$

$(R)(S)$ representa la concatenación de los lenguajes representados por R y S ; $(R \cup S)$ representa su unión, y $(R)^*$ representa la clausura de Kleene del lenguaje representado por R . Los paréntesis () son símbolos de agrupación y se pueden omitir si no hay peligro de ambigüedad.

Para una expresión regular R cualquiera se utiliza en ocasiones la siguiente notación:

$$L(R) := \text{lenguaje representado por } R.$$

Utilizando esta notación y la definición recursiva de expresión regular podemos escribir las siguientes igualdades en las que R y S son expresiones regulares arbitrarias:

$$\begin{aligned}L(\emptyset) &= \emptyset. \\ L(\lambda) &= \{\lambda\}. \\ L(a) &= \{a\}, \quad a \in \Sigma. \\ L(RS) &= L(R)L(S). \\ L(R \cup S) &= L(R) \cup L(S). \\ L(R^*) &= L(R)^*. \end{aligned}$$

Ejemplo Dado el alfabeto $\Sigma = \{a, b, c\}$,

$$(a \cup b^*)a^*(bc)^*$$

es una expresión regular que representa al lenguaje

$$(\{a\} \cup \{b\}^*) \cdot \{a\}^* \cdot \{bc\}^*.$$

Ejemplo Dado el alfabeto $\Sigma = \{a, b\}$,

$$(\lambda \cup a)^* (a \cup b)^* (ba)^*$$

es una expresión regular que representa al lenguaje

$$(\{\lambda\} \cup \{a\})^* \cdot (\{a\} \cup \{b\})^* \cdot \{ba\}^*.$$

Ejemplos Podemos representar los tres primeros lenguajes de la sección 1.13 con expresiones regulares:

1. El lenguaje A de todas las cadenas que tienen exactamente una a :

$$A = b^* ab^*.$$

2. El lenguaje B de todas las cadenas que comienzan con b :

$$B = b(a \cup b)^*.$$

3. El lenguaje C de todas las cadenas que contienen la cadena ba :

$$C = (a \cup b)^* ba (a \cup b)^*.$$

La representación de lenguajes regulares por medio de expresiones regulares no es única. Es posible que haya varias expresiones regulares diferentes para el mismo lenguaje. Por ejemplo, $b(a \cup b)^*$ y $b(b \cup a)^*$ representan el mismo lenguaje.

Otro ejemplo: las dos expresiones regulares $(a \cup b)^*$ y $(a^*b^*)^*$ representan el mismo lenguaje por la propiedad 9 de la sección 1.11.

Ejemplos Encontrar expresiones regulares que representen los siguientes lenguajes, definidos sobre el alfabeto $\Sigma = \{a, b\}$:

1. Lenguaje de todas las cadenas que comienzan con el símbolo b y terminan con el símbolo a .

Solución. $b(a \cup b)^* a.$

2. Lenguaje de todas las cadenas que tienen un número par de símbolos (cadenas de longitud par).

Solución. $(aa \cup ab \cup ba \cup bb)^*$. Otra expresión regular para este lenguaje es $[(a \cup b)(a \cup b)]^*$.

3. Lenguaje de todas las cadenas que tienen un número par de *aes*.

Soluciones:

$$\begin{aligned} & b^*(b^*ab^*ab^*)^*. \\ & (ab^*a \cup b)^*. \\ & (b^*ab^*ab^*)^* \cup b^*. \end{aligned}$$

Ejemplos Encontrar expresiones regulares que representen los siguientes lenguajes, definidos sobre el alfabeto $\Sigma = \{0, 1\}$:

1. Lenguaje de todas las cadenas que tienen exactamente dos ceros.

Solución. $1^*01^*01^*$.

2. Lenguaje de todas las cadenas cuyo penúltimo símbolo, de izquierda a derecha, es un 0.

Solución. $(0 \cup 1)^*0(0 \cup 1)$. Usando la propiedad distributiva obtenemos otra expresión regular para este lenguaje: $(0 \cup 1)^*00 \cup (0 \cup 1)^*01$.

Ejemplo Sea $\Sigma = \{0, 1\}$. Encontrar una expresión regular que represente el lenguaje de todas las cadenas que no contienen dos ceros consecutivos.

Solución. La condición de que no haya dos ceros consecutivos implica que todo cero debe estar seguido necesariamente de un uno, excepto un cero al final de la cadena. Por lo tanto, las cadenas de este lenguaje se obtienen concatenando unos con bloques 01, de todas las formas posibles. Hay que tener en cuenta, además, que la cadena puede terminar ya sea en 1 o en 0. A partir de este análisis, llegamos a la expresión regular

$$(1 \cup 01)^* \cup (1 \cup 01)^*0.$$

Usando la propiedad distributiva, obtenemos otra expresión para este lenguaje: $(1 \cup 01)^*(\lambda \cup 0)$.

Ejemplo Sea $\Sigma = \{a, b, c\}$. Encontrar una expresión regular que represente el lenguaje de todas las cadenas que no contienen la cadena *bc*.

Solución. Una b puede estar seguida solamente de otra b o de una a , mientras que las a es y las c es pueden estar seguidas de cualquier símbolo. Teniendo en cuenta todas las restricciones y posibilidades, arribamos a la siguiente expresión regular:

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

La condición de que no aparezca la cadena bc significa que una c puede estar precedida solamente de una a y de otra c . Siguiendo esta descripción, obtenemos otra expresión regular para el lenguaje en cuestión:

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

Ejercicios de la sección 1.14

- ① Encontrar expresiones regulares para los lenguajes descritos a continuación:
 - (i) $\Sigma = \{0, 1, 2\}$. Lenguaje de todas las cadenas que comienzan con 2 y terminan con 1.
 - (ii) $\Sigma = \{a, b, c\}$. Lenguaje de todas las cadenas que tienen un número par de símbolos.
 - (iii) $\Sigma = \{a, b\}$. Lenguaje de todas las cadenas que tienen un número impar de símbolos.
 - (iv) $\Sigma = \{a, b, c\}$. Lenguaje de todas las cadenas que tienen un número impar de símbolos.
 - (v) $\Sigma = \{a, b\}$. Lenguaje de todas las cadenas que tienen un número impar de a es.
 - (vi) $\Sigma = \{a, b\}$. Lenguaje de todas las cadenas que tienen la cadena ab un número par de veces.
 - (vii) $\Sigma = \{a, b\}$. Lenguaje de todas las cadenas que tienen un número par de a es o un número impar de b es.
 - (viii) $\Sigma = \{0, 1, 2\}$. Lenguaje de todas las cadenas que no contienen dos unos consecutivos.

- ② Encontrar expresiones regulares para los siguientes lenguajes definidos sobre el alfabeto $\Sigma = \{0, 1\}$:
 - (i) Lenguaje de todas las cadenas que tienen por lo menos un 0 y por lo menos un 1.

- (ii) Lenguaje de todas las cadenas que tienen a lo sumo dos ceros consecutivos.
- (iii) Lenguaje de todas las cadenas cuyo quinto símbolo, de izquierda a derecha, es un 1.
- (iv) Lenguaje de todas las cadenas de longitud par ≥ 2 formadas por ceros y unos alternados.
- (v) Lenguaje de todas las cadenas cuya longitud es ≥ 4 .
- (vi) Lenguaje de todas las cadenas de longitud impar que tienen unos únicamente en las posiciones impares.
- (vii) Lenguaje de todas las cadenas cuya longitud es un múltiplo de tres.
- (viii) Lenguaje de todas las cadenas que no contienen tres ceros consecutivos.
- (ix) Lenguaje de todas las cadenas que no contienen cuatro ceros consecutivos.
- !(x) Lenguaje de todas las cadenas que no contienen la subcadena 101.



No todos los lenguajes sobre un alfabeto dado Σ son regulares. Más adelante se mostrará que el lenguaje

$$L = \{\lambda, ab, aabb, aaabbb, \dots\} = \{a^n b^n : n \geq 0\}$$

sobre $\Sigma = \{a, b\}$ no se puede representar por medio de una expresión regular, y por lo tanto, no es un lenguaje regular.

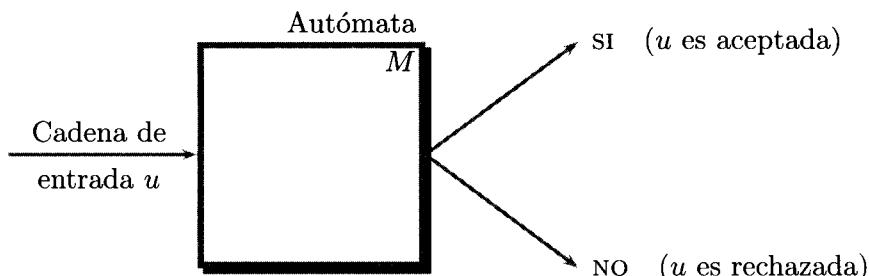
Capítulo 2

Autómatas finitos

Los autómatas son máquinas abstractas con capacidad de computación. Históricamente, su estudio se originó con la llamada “máquina de Turing”, que fue propuesta en 1936 por el matemático británico Alan Turing (1912–1954) con el propósito de precisar las características y las limitaciones de un dispositivo de computación mecánica. En los años 40 y 50 del siglo XX se adelantaron investigaciones sobre máquinas de Turing con capacidad restringida, lo que dio lugar a la noción de autómata finito.

2.1. Autómatas finitos deterministas (AFD)

Los **autómatas finitos** son máquinas abstractas que procesan cadenas de entrada, las cuales son aceptadas o rechazadas:



El autómata actúa leyendo los símbolos escritos sobre una cinta semi-infinita, dividida en celdas o casillas, sobre la cual se escribe una cadena de entrada u , un símbolo por casilla. El autómata posee una **unidad de control** (también llamada **cabeza lectora**, **control finito** o **unidad de**

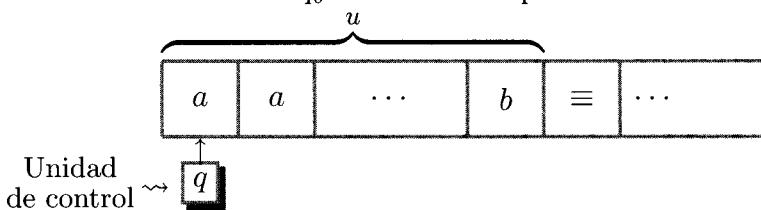
memoria) que tiene un número finito de configuraciones internas, llamadas **estados del autómata**. Entre los estados de un autómata se destacan el **estado inicial** y los **estados finales o estados de aceptación**.

Formalmente, un autómata finito M está definido por cinco parámetros o componentes, $M = (\Sigma, Q, q_0, F, \delta)$, a saber:

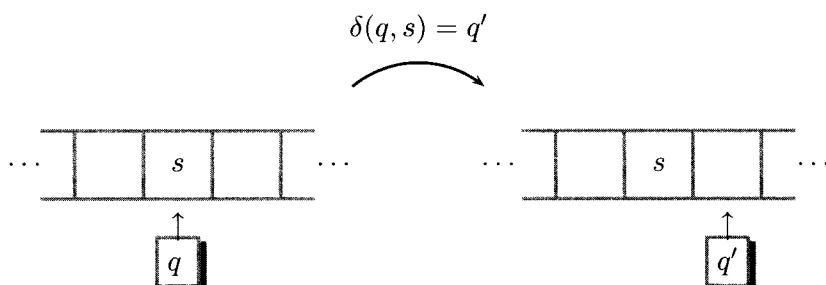
1. Un alfabeto Σ , llamado alfabeto de cinta. Todas las cadenas que procesa M pertenecen a Σ^* .
2. $Q = \{q_0, q_1, \dots, q_n\}$, conjunto de estados internos del autómata.
3. $q_0 \in Q$, estado inicial.
4. $F \subseteq Q$, conjunto de estados finales o de aceptación. $F \neq \emptyset$.
5. La función de transición del autómata

$$\begin{array}{rcl} \delta : Q \times \Sigma & \longrightarrow & Q \\ (q, s) & \longmapsto & \delta(q, s) \end{array}$$

Una cadena de entrada u se coloca en la cinta de tal manera que el primer símbolo de u ocupa la primera casilla de la cinta. La unidad de control está inicialmente en el estado q_0 escaneando la primera casilla:



La función de transición δ indica el estado al cual pasa el control finito, dependiendo del símbolo escaneado y de su estado actual. Así, $\delta(q, s) = q'$ significa que, en presencia del símbolo s , la unidad de control pasa del estado q al estado q' y se desplaza hacia la derecha. Esta acción constituye un **paso computacional**:



Puesto que la función δ está definida para toda combinación estado-símbolo, una cadena de entrada cualquiera es procesada completamente, hasta que la unidad de control encuentra la primera casilla vacía.

La unidad de control de un autómata siempre se desplaza hacia la derecha; no puede retornar ni escribir símbolos sobre la cinta.

Ejemplo Consideremos el autómata definido por los siguientes cinco componentes:

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

$$Q = \{q_0, q_1, q_2\}.$$

q_0 : estado inicial.

$F = \{q_0, q_2\}$, estados de aceptación.

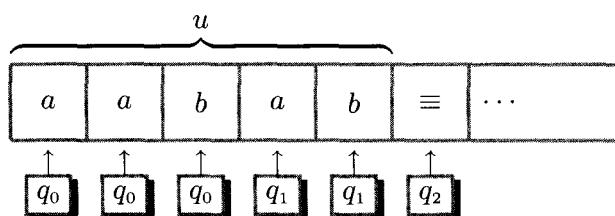
Función de transición δ :

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

$$\begin{array}{ll} \delta(q_0, a) = q_0 & \delta(q_0, b) = q_1 \\ \delta(q_1, a) = q_1 & \delta(q_1, b) = q_2 \\ \delta(q_2, a) = q_1 & \delta(q_2, b) = q_1. \end{array}$$

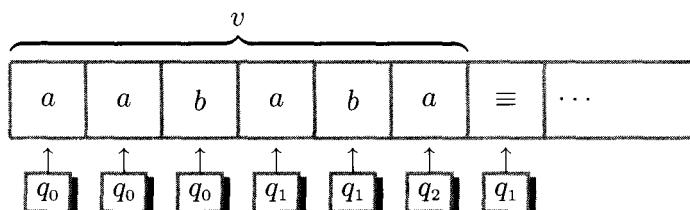
Vamos a ilustrar el procesamiento de dos cadenas de entrada.

1. $u = aabab$.



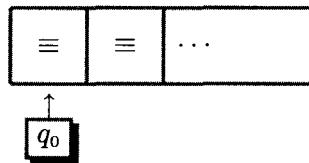
Como q_2 es un estado de aceptación, la cadena de entrada u es aceptada.

2. $v = aababa$.



Puesto que q_1 no es un estado de aceptación, la entrada v es rechazada.

Caso especial: la cadena λ es la cadena de entrada.



Como q_0 es un estado de aceptación, la cadena λ es aceptada.

En general se tiene lo siguiente: la cadena vacía λ es aceptada por un autómata M si y solamente si el estado inicial q_0 de M también es un estado de aceptación.

Los autómatas finitos descritos anteriormente se denominan **autómatas finitos deterministas** (AFD) ya que para cada estado q y para cada símbolo $a \in \Sigma$, la función de transición $\delta(q, a)$ siempre está definida. Es decir, la función de transición δ *determina completa y unívocamente* la acción que el autómata realiza cuando la unidad de control se encuentra en un estado q leyendo un símbolo s sobre la cinta.

Dado un autómata M , el lenguaje **aceptado o reconocido** por M se denota $L(M)$ y se define por

$$L(M) := \{u \in \Sigma^* : M \text{ termina el procesamiento de la cadena de entrada } u \text{ en un estado } q \in F\}.$$

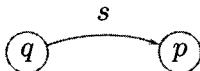
2.2. Diagrama de transiciones de un autómata

Un autómata finito se puede representar por medio de un grafo dirigido y etiquetado. Recuérdese que un **grafo** es un conjunto de vértices o nodos unidos por arcos o conectores; si los arcos tienen tanto dirección como etiquetas, el grafo se denomina **grafo dirigido y etiquetado** o **digrafo etiquetado**.

El digrafo etiquetado de un autómata se obtiene siguiendo las siguientes convenciones:

- Los vértices o nodos son los estados del autómata.
- El estado q se representa por: (q)
- El estado inicial q_0 se representa por: $>(q_0)$

- Un estado final q se representa por:
- La transición $\delta = (q, s) = p$ se representa en la forma



Dicho grafo se denomina **diagrama de transiciones del autómata** y es muy útil para hacer el seguimiento completo del procesamiento de una cadena de entrada. Una cadena u es aceptada si existe una trayectoria etiquetada con los símbolos de u , que comienza en el estado q_0 y termina en un estado de aceptación.

Ejemplo Diagrama de transiciones del autómata presentado en la sección anterior.

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

$$Q = \{q_0, q_1, q_2\}.$$

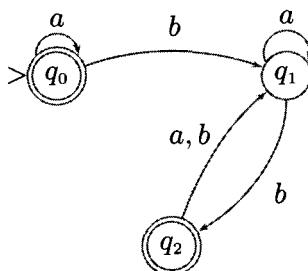
q_0 : estado inicial.

$F = \{q_0, q_2\}$, estados de aceptación.

Función de transición δ :

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

$$\begin{array}{ll} \delta(q_0, a) = q_0 & \delta(q_0, b) = q_1 \\ \delta(q_1, a) = q_1 & \delta(q_1, b) = q_2 \\ \delta(q_2, a) = q_1 & \delta(q_2, b) = q_1 \end{array}$$



Examinando el diagrama de transiciones podemos observar fácilmente que la entrada $aaababbb$ es aceptada mientras que $aabaaba$ es rechazada.

2.3. Diseño de autómatas

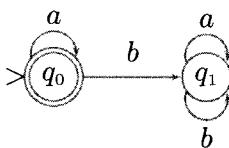
Para autómatas deterministas se adopta la siguiente convención adicional con respecto a los diagramas de transiciones: se supone que los arcos no dibujadas explícitamente conducen a un estado “limbo” de no-aceptación. Es decir, en el diagrama de transiciones se indican únicamente los arcos que intervengan en trayectorias de aceptación. Esto permite simplificar considerablemente los diagramas.

En este capítulo abordaremos dos tipos de problemas:

1. Dado un lenguaje regular L diseñar un autómata finito M que acepte o reconozca a L , es decir, tal que $L(M) = L$.
2. Dado un autómata M determinar el lenguaje aceptado por M .

Más adelante se demostrará, en toda su generalidad, que estos problemas *siempre* tienen solución. Consideremos inicialmente problemas del primer tipo.

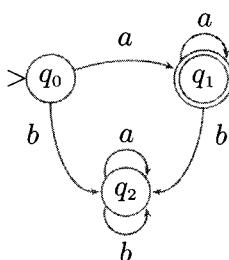
Ejemplo $L = a^* = \{\lambda, a, a^2, a^3, \dots\}$. AFD M tal que $L(M) = L$:



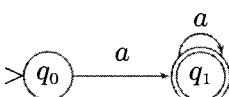
Versión simplificada:



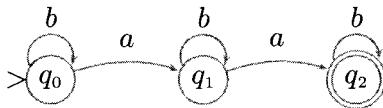
Ejemplo $L = a^+ = \{a, a^2, a^3, \dots\}$. AFD M tal que $L(M) = L$:



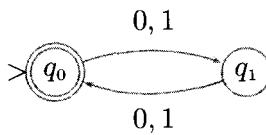
Versión simplificada:



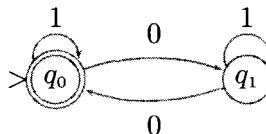
Ejemplo $\Sigma = \{a, b\}$. L = lenguaje de las cadenas que contienen exactamente dos a s = $b^*ab^*ab^*$. AFD M tal que $L(M) = L$:



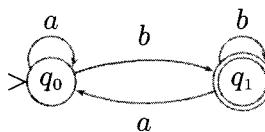
Ejemplo $\Sigma = \{0, 1\}$. L = lenguaje de las cadenas sobre Σ que tienen un número par de símbolos (cadenas de longitud par). AFD M tal que $L(M) = L$:



Ejemplo $\Sigma = \{0, 1\}$. L = lenguaje de las cadenas sobre Σ que contienen un número par de ceros. AFD M tal que $L(M) = L$:



Ejemplo $\Sigma = \{a, b\}$. L = lenguaje de las cadenas sobre Σ que terminan en b . AFD M tal que $L(M) = L$:



Ejercicios de la sección 2.3

- ① Diseñar autómatas finitos deterministas que acepten los siguientes lenguajes:
 - (i) $\Sigma = \{0, 1\}$. L = lenguaje de las cadenas sobre Σ de longitud impar.

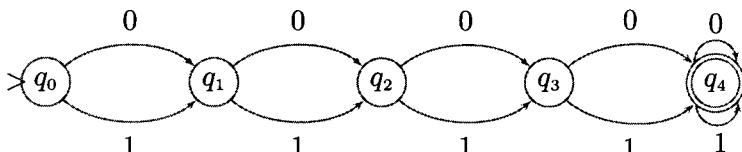
- (ii) $\Sigma = \{0, 1\}$. L = lenguaje de las cadenas sobre Σ que contienen un número impar de unos.
- (iii) $\Sigma = \{a, b\}$. $L = ab^+$.
- (iv) $\Sigma = \{a, b\}$. $L = ab^* \cup ab^*a$.
- (v) $\Sigma = \{0, 1\}$. $L = (0 \cup 10)^*$.
- (vi) $\Sigma = \{0, 1\}$. $L = (01 \cup 10)^*$.
- (vii) $\Sigma = \{0, 1\}$. Lenguaje de todas las cadenas que no contienen dos unos consecutivos.
- (viii) $\Sigma = \{a, b\}$. $L = \{a^{2i}b^{3j} : i, j \geq 0\}$.
- (ix) $\Sigma = \{a, b\}$. L = lenguaje de las cadenas sobre Σ que contienen un número par de aes y un número par de bes. Ayuda: utilizar 4 estados.
- (x) $\Sigma = \{a, b\}$. Para cada combinación de las condiciones “par” e “impar” y de las conectivas “o” e “y”, diseñar un AFD que acepte el lenguaje L definido por

L = lenguaje de las cadenas con un número par/impar de aes y/o un número par/impar de bes.

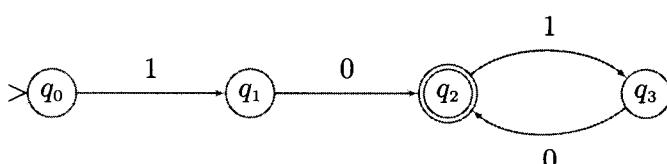
Ayuda: utilizar el autómata de 4 estados diseñado en el ejercicio anterior, modificando adecuadamente el conjunto de estados finales.

- ② Determinar los lenguajes aceptados por los siguientes AFD. Describir los lenguajes ya sea por medio de una propiedad característica o de una expresión regular.

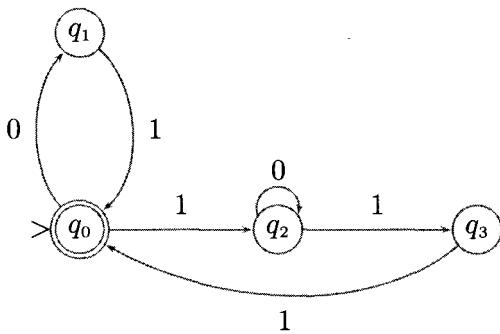
(i)



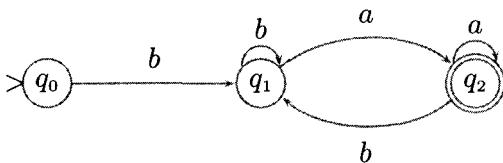
(ii)



(iii)



(iv)



2.4. Autómatas finitos no-deterministas (AFN)

Los **autómatas finitos no-deterministas** (AFN) se asemejan a los AFD, excepto por el hecho de que para cada estado $q \in Q$ y cada $a \in \Sigma$, la transición $\delta(q, a)$ puede consistir en más de un estado o puede no estar definida. Concretamente, un AFN está definido por $M = (\Sigma, Q, q_0, F, \Delta)$ donde:

1. Σ es el alfabeto de cinta.
2. Q es un conjunto (finito) de estados internos.
3. $q_0 \in Q$ es el estado inicial.
4. $\emptyset \neq F \subseteq Q$ es el conjunto de estados finales o estados de aceptación.
- 5.

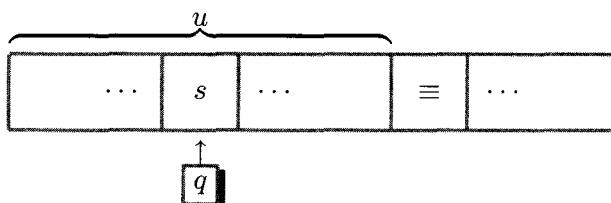
$$\begin{aligned} \Delta : Q \times \Sigma &\longrightarrow \wp(Q) \\ (q, s) &\longmapsto \Delta(q, s) = \{q_{i_1}, q_{i_2}, \dots, q_{i_k}\} \end{aligned}$$

donde $\wp(Q)$ es el conjunto de subconjunto de Q .

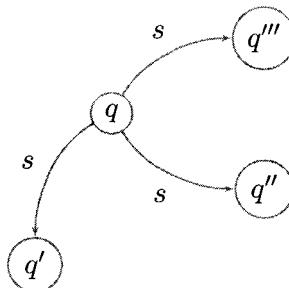
El significado de $\Delta(q, s) = \{q_1, q_2, \dots, q_k\}$ es el siguiente: estando en el estado q , en presencia del símbolo s , la unidad de control puede pasar (aleatoriamente) a uno cualquiera de los estados q_1, q_2, \dots, q_k , después de lo cual se desplaza a la derecha.

Puede suceder que $\Delta(q, s) = \emptyset$, lo cual significa que, si durante el procesamiento de una cadena de entrada u , M ingresa al estado q leyendo sobre la cinta el símbolo s , el cómputo se aborta.

Cómputo abortado:



La noción de diagrama de transiciones para un AFN se define de manera análoga al caso AFD, pero puede suceder que desde un mismo nodo (estado) salgan dos o más arcos con la misma etiqueta:



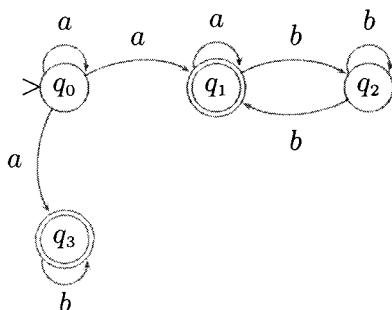
Un AFN M puede procesar una cadena de entrada $u \in \Sigma^*$ de varias maneras. Sobre el diagrama de transiciones del autómata, esto significa que pueden existir varias trayectorias, desde el estado q_0 , etiquetadas con los símbolos de u .

La siguiente es la noción de aceptación para autómatas no-deterministas:

$L(M)$ = lenguaje aceptado o reconocido por M = $\{u \in \Sigma^* : \text{existe por lo menos un cómputo completo de } u \text{ que termina en un estado } q \in F\}$
--

Es decir, para que una cadena u sea aceptada, debe existir algún cómputo en el que u sea procesada completamente y que finalice estando M en un estado de aceptación.

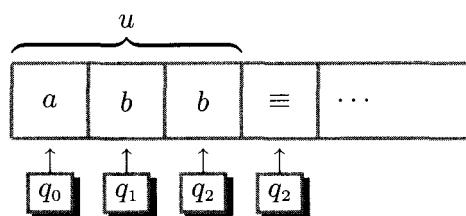
Ejemplo Sea M el siguiente AFN:



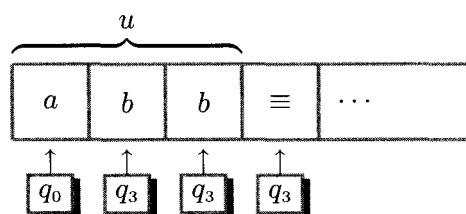
Δ	a	b
q_0	$\{q_0, q_1, q_3\}$	\emptyset
q_1	$\{q_1\}$	$\{q_2\}$
q_2	\emptyset	$\{q_1, q_2\}$
q_3	\emptyset	$\{q_3\}$

Para la cadena de entrada $u = abb$, existen cómputos que conducen al rechazo, cómputos abortados y cómputos que terminan en estados de aceptación. Según la definición de lenguaje aceptado, $u \in L(M)$.

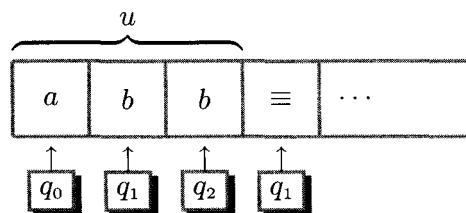
Cómputo de rechazo:



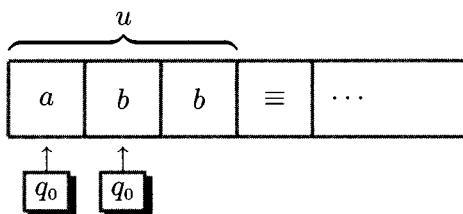
Cómputo de aceptación:



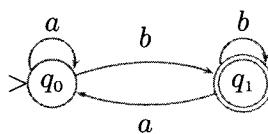
Otro cómputo de aceptación:



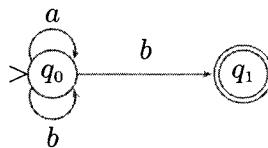
Cómputo abortado:



Ejemplo En el último ejemplo de la sección 2.3 se diseñó el siguiente AFD que acepta el lenguaje de las cadenas sobre $\Sigma = \{a, b\}$ que terminan en b :

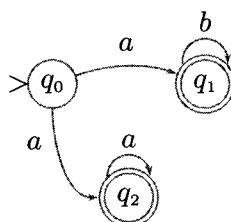


Un AFN que acepta el mismo lenguaje y que es, tal vez, más fácil de concebir, es el siguiente:

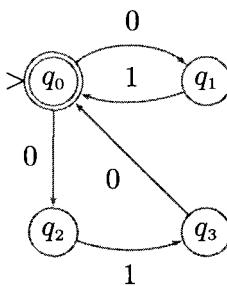


Este autómata se asemeja a la expresión regular $(a \cup b)^*b$.

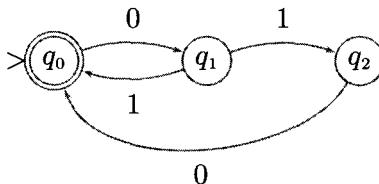
Ejemplo Considérese el lenguaje $L = ab^* \cup a^+$ sobre el alfabeto $\Sigma = \{a, b\}$. El siguiente AFN M satisface $L(M) = L$.



Ejemplo $\Sigma = \{0, 1\}$, $L = (01 \cup 010)^*$. El siguiente AFN acepta a L .



Otro AFN que acepta el mismo lenguaje y que tiene sólo tres estados es el siguiente:



Ejercicios de la sección 2.4

Diseñar autómatas AFD o AFN que acepten los siguientes lenguajes:

- ① $\Sigma = \{a, b\}$. $L = ab^+a^*$.
- ② $\Sigma = \{a, b\}$. $L = a(a \cup ab)^*$.
- ③ $\Sigma = \{a, b, c\}$. $L = a^*b^*c^*$.
- ④ $\Sigma = \{0, 1, 2\}$. L = lenguaje de las cadenas sobre Σ que comienzan con 0 y terminan con 2.
- ⑤ $\Sigma = \{0, 1\}$. Lenguaje de las cadenas de longitud par ≥ 2 formadas por ceros y unos alternados.
- ⑥ $\Sigma = \{0, 1\}$. Lenguaje de las cadenas que tienen a lo sumo dos ceros consecutivos.
- ⑦ $\Sigma = \{0, 1\}$. Lenguaje de las cadenas de longitud impar que tienen unos únicamente en las posiciones impares.
- ⑧ $\Sigma = \{a, b, c\}$. L = lenguaje de las cadenas sobre Σ que contienen la cadena bc .
- ⑨ $\Sigma = \{a, b, c\}$. L = lenguaje de las cadenas sobre Σ que no contienen la cadena bc . En el último ejemplo de la sección 1.14 se presentaron dos expresiones regulares para L . Nota: ¡se puede construir un AFD con sólo dos estados para aceptar este lenguaje!

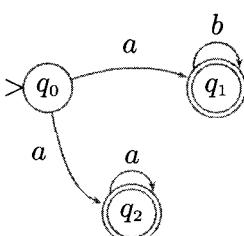
2.5. Equivalencia computacional entre los AFD y los AFN

En esta sección se mostrará que los modelos AFD y AFN son computacionalmente equivalentes. En primer lugar, es fácil ver que un AFD $M = (\Sigma, Q, q_0, F, \delta)$ puede ser considerado como un AFN $M' = (\Sigma, Q, q_0, F, \Delta)$ definiendo $\Delta(q, a) = \{\delta(q, a)\}$ para cada $q \in Q$ y cada $a \in \Sigma$. Para la afirmación recíproca tenemos el siguiente teorema.

2.5.1 Teorema. *Dado un AFN $M = (\Sigma, Q, q_0, F, \Delta)$ se puede construir un AFD M' equivalente a M , es decir, tal que $L(M) = L(M')$.*

Este teorema, cuya demostración se dará en detalle más adelante, establece que el no-determinismo se puede eliminar. Dicho de otra manera, los autómatas deterministas y los no-deterministas aceptan los mismos lenguajes. La idea de la demostración consiste en considerar cada conjunto de estados $\{p_1, \dots, p_j\}$, que aparezca en la tabla de la función Δ del autómata no-determinista, como un único estado del nuevo autómata determinista. La tabla de Δ se completa hasta que no aparezcan nuevas combinaciones de estados. Los estados de aceptación del nuevo autómata son los conjuntos de estados en los que aparece *por lo menos* un estado de aceptación del autómata original. El siguiente ejemplo ilustra el procedimiento.

Ejemplo Consideremos el AFN M , presentado en la sección 2.4, que acepta el lenguaje $L(M) = ab^* \cup a^+$ sobre $\Sigma = \{a, b\}$:

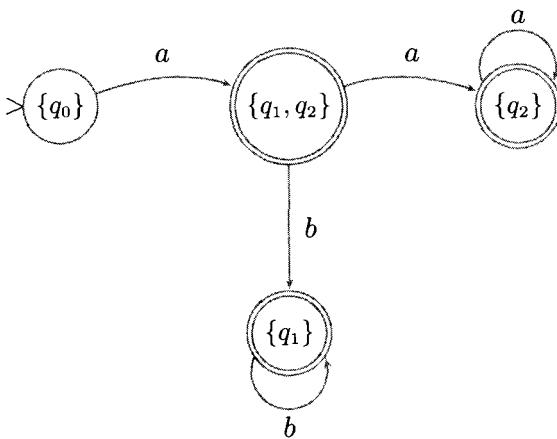


Δ	a	b
q_0	$\{q_1, q_2\}$	\emptyset
q_1	\emptyset	$\{q_1\}$
q_2	$\{q_2\}$	\emptyset

El nuevo AFD M' construido a partir de M tiene un estado más, $\{q_1, q_2\}$, y su función de transición δ tiene el siguiente aspecto:

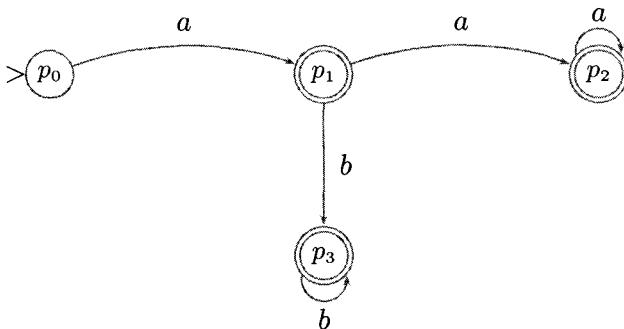
δ	a	b
q_0	$\{q_1, q_2\}$	\emptyset
q_1	\emptyset	$\{q_1\}$
q_2	$\{q_2\}$	\emptyset
$\{q_1, q_2\}$	$\{q_2\}$	$\{q_1\}$

El diagrama de transiciones de este autómata es:

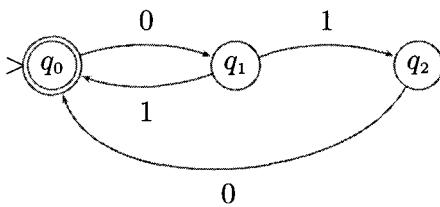


Los estados de aceptación son aquéllos en los que aparezcan q_1 ó q_2 , que son los estados de aceptación del autómata original.

Para mayor simplicidad, podemos cambiar los nombres de los estados del nuevo autómata:



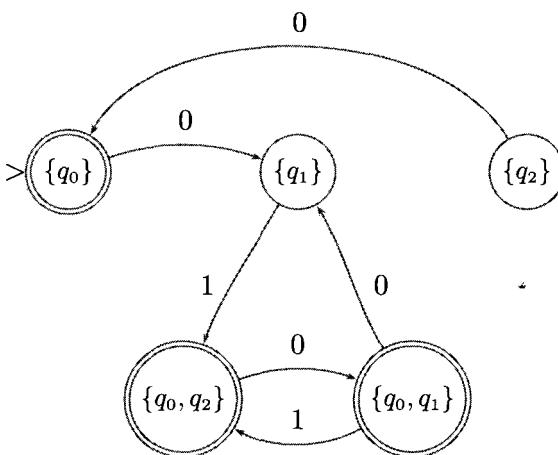
Ejemplo El siguiente AFN M , presentado en la sección 2.4, acepta el lenguaje $L = (01 \cup 010)^*$ sobre $\Sigma = \{0, 1\}$.



La tabla de la función de transición de M se extiende para completar la función δ del nuevo AFN:

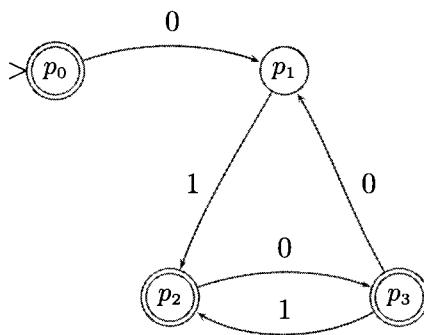
δ	0	1
q_0	$\{q_1\}$	\emptyset
q_1	\emptyset	$\{q_0, q_2\}$
q_2	$\{q_0\}$	\emptyset
$\{q_0, q_2\}$	$\{q_0, q_1\}$	\emptyset
$\{q_0, q_1\}$	$\{q_1\}$	$\{q_0, q_2\}$

El diagrama de transiciones del nuevo autómata es:



Los estados de aceptación son aquéllos en los que aparezca q_0 ya que q_0 es el único estado de aceptación del autómata original.

Puesto que el nuevo estado $\{q_2\}$ no interviene en la aceptación de cadenas, el autómata se puede simplificar en la siguiente forma:



Para la demostración del Teorema 2.5.1, conviene extender la definición de la función de transición, tanto de los autómatas deterministas como de los no-deterministas.

2.5.2 Definición. Sea $M = (\Sigma, Q, q_0, F, \delta)$ un AFD. La función de transición $\delta : Q \times \Sigma \rightarrow Q$ se extiende a una función $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$ por medio de la siguiente definición recursiva:

$$\begin{cases} \hat{\delta}(q, \lambda) = q, & q \in Q, \\ \hat{\delta}(q, a) = \delta(q, a), & q \in Q, a \in \Sigma, \\ \hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a), & q \in Q, a \in \Sigma, w \in \Sigma^*. \end{cases}$$

Según esta definición, para una cadena de entrada $w \in \Sigma^*$, $\hat{\delta}(q_0, w)$ es el estado en el que el autómata termina el procesamiento de w . Por lo tanto, podemos describir el lenguaje aceptado por M de la siguiente forma:

$$L(M) = \{w \in \Sigma^* : \hat{\delta}(q_0, w) \in F\}.$$

Notación. Sin peligro de ambigüedad, la función extendida $\hat{\delta}(q, w)$ se notará simplemente $\delta(q, w)$.

2.5.3 Definición. Sea $M = (\Sigma, Q, q_0, F, \Delta)$ un AFN. La función de transición $\Delta : Q \times \Sigma \rightarrow \wp(Q)$ se extiende inicialmente a conjuntos de estados. Para $a \in \Sigma$ y $S \subseteq F$ se define

$$\Delta(S, a) := \bigcup_{q \in S} \Delta(q, a).$$

Luego se extiende Δ a una función $\widehat{\Delta} : Q \times \Sigma^* \longrightarrow \mathcal{P}(Q)$, de manera similar a como se hace para los AFD. Recursivamente,

$$\begin{cases} \widehat{\Delta}(q, \lambda) = \{q\}, & q \in Q, \\ \widehat{\Delta}(q, a) = \Delta(q, a), & q \in Q, a \in \Sigma, \\ \widehat{\Delta}(q, wa) = \Delta(\widehat{\Delta}(q, w), a) = \bigcup_{p \in \widehat{\Delta}(q, w)} \Delta(p, a), & q \in Q, a \in \Sigma, w \in \Sigma^*. \end{cases}$$

Según esta definición, para una cadena de entrada $w \in \Sigma^*$, $\widehat{\Delta}(q_0, w)$ es el conjunto de los posibles estados en los que terminan los cómputos *completos* de w . Si el cómputo se aborta durante el procesamiento de w , se tendría $\widehat{\Delta}(q_0, w) = \emptyset$. Usando la función extendida $\widehat{\Delta}$, el lenguaje aceptado por M se puede describir de la siguiente forma:

$$L(M) = \{w \in \Sigma^* : \widehat{\Delta}(q_0, w) \text{ contiene un estado de aceptación}\}.$$

Notación. Sin peligro de ambigüedad, la función extendida $\widehat{\Delta}(q, w)$ se notará simplemente $\Delta(q, w)$.

Demostración del Teorema 2.5.1:

Dado el AFN $M = (\Sigma, Q, q_0, F, \Delta)$, construimos el AFD M' así:

$$M' = (\Sigma, \mathcal{P}(Q), \{q_0\}, F', \delta)$$

donde

$$\begin{aligned} \delta : \mathcal{P}(Q) \times \Sigma &\longrightarrow \mathcal{P}(Q) \\ (S, a) &\longmapsto \delta(S, a) := \Delta(S, a). \end{aligned}$$

$$F' = \{S \subseteq \mathcal{P}(Q) : S \cap F \neq \emptyset\}.$$

Se demostrará que $L(M) = L(M')$ probando que, para toda cadena $w \in \Sigma^*$, $\delta(\{q_0\}, w) = \Delta(q_0, w)$. Esta igualdad se demostrará por inducción sobre w . Para $w = \lambda$, claramente se tiene $\delta(\{q_0\}, \lambda) = \Delta(q_0, \lambda) = \{q_0\}$. Para $w = a$, $a \in \Sigma$, se tiene

$$\delta(\{q_0\}, a) = \Delta(\{q_0\}, a) = \Delta(q_0, a).$$

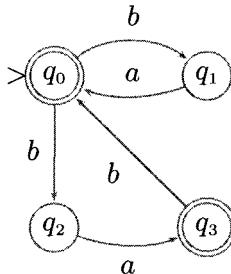
Supóngase (hipótesis de inducción) que $\delta(\{q_0\}, w) = \Delta(q_0, w)$, y que $a \in \Sigma$.

$$\begin{aligned} \delta(\{q_0\}, wa) &= \delta(\delta(\{q_0\}, w), a) && (\text{definición de la extensión de } \delta) \\ &= \delta(\Delta(\{q_0\}, w), a) && (\text{hipótesis de inducción}) \\ &= \Delta(\Delta(\{q_0\}, w), a) && (\text{definición de } \delta) \\ &= \Delta(\{q_0\}, wa) && (\text{definición de la extensión de } \Delta) \\ &= \Delta(q_0, wa) && (\text{definición de la extensión de } \Delta). \quad \square \end{aligned}$$

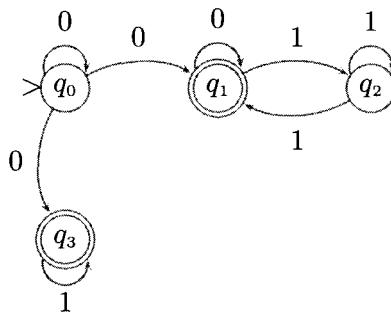
Ejercicios de la sección 2.5

Diseñar AFD equivalentes a los siguientes AFN:

①



②



2.6. Autómatas con transiciones λ (AFN- λ)

Un **autómata finito con transiciones λ** (AFN- λ) es un autómata no-determinista $M = (\Sigma, Q, q_0, F, \Delta)$ en el que la función de transición está definida como:

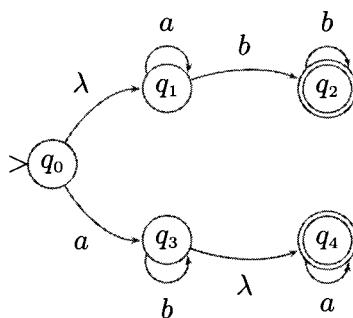
$$\Delta : Q \times (\Sigma \cup \{\lambda\}) \rightarrow \mathcal{P}(Q).$$

La transición $\Delta(q, \lambda) = \{p_{i_1}, \dots, p_{i_n}\}$, llamada **transición λ** , **transición nula** o **transición espontánea**, tiene el siguiente significado computacional: estando en el estado q , el autómata puede cambiar a uno cualquiera de los estados p_{i_1}, \dots, p_{i_n} , independientemente del símbolo leído y sin mover la unidad de control. Dicho de otra manera, las transiciones λ permiten al autómata cambiar internamente de estado sin procesar o “consumir” el símbolo leído sobre la cinta.

En el diagrama del autómata, las transiciones λ dan lugar a arcos con etiquetas λ . Una cadena de entrada w es aceptada por un AFN- λ si existe por lo menos una trayectoria, desde el estado q_0 , cuyas etiquetas son exactamente los símbolos de w , intercalados con cero, uno o más λ s.

En los autómatas AFN- λ , al igual que en los AFN, puede haber múltiples cómputos para una misma cadena de entrada, así como cómputos abortados.

Ejemplo M :



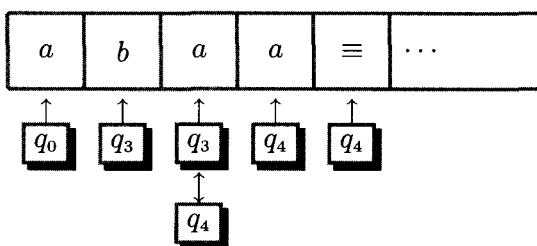
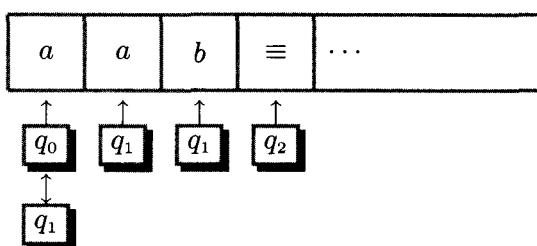
Ejemplos de cadenas aceptadas por M :

$$u = aab$$

$$v = abaa$$

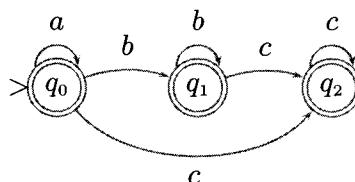
$$w = abbaa$$

Cómputos de aceptación de $u = aab$ y $v = abaa$:

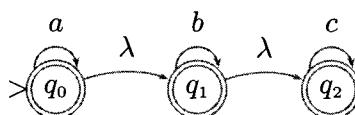


Los AFN- λ permiten aún más libertad en el diseño de autómatas, especialmente cuando hay numerosas concatenaciones.

Ejemplo $\Sigma = \{a, b, c\}$. $L = a^*b^*c^*$. AFD que acepta a L :



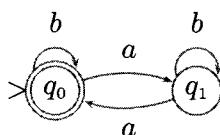
AFN- λ que acepta a L :



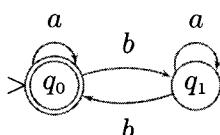
Este autómata se asemeja a la expresión regular $a^*b^*c^*$: las concatenaciones han sido reemplazadas por transiciones λ .

Ejemplo $\Sigma = \{a, b\}$. L = lenguaje de todas las cadenas sobre Σ que tienen un número par de aes o un número par de bes.

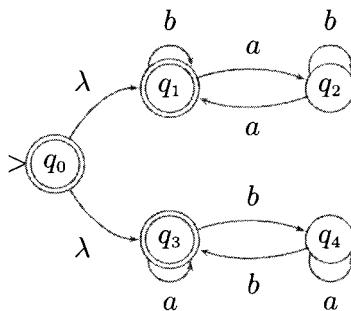
AFD que acepta el lenguaje de las cadenas con un número par de aes:



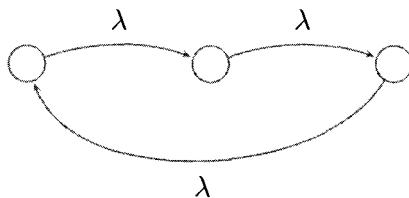
AFD que acepta el lenguaje de las cadenas con un número par de bes:



AFN- λ que acepta el lenguaje de las cadenas con un número par de aes o un número par de bes:



A diferencia de los AFD y los AFN, en los AFN- λ pueden existir “cómputos infinitos”, es decir cómputos que nunca terminan. Esto puede suceder si el autómata ingresa a un estado desde el cual haya varias transiciones λ encadenadas que retornen al mismo estado, como por ejemplo:



Ejercicios de la sección 2.6

Diseñar AFN- λ que acepten los siguientes lenguajes:

- ① $(ab \cup b)^*ab^*$, sobre $\Sigma = \{a, b\}$.
- ② $a(a \cup c)^*b^+$, sobre $\Sigma = \{a, b, c\}$.
- ③ $ab^* \cup ba^* \cup b(ab \cup ba)^*$, sobre $\Sigma = \{a, b\}$.
- ④ $ab^*ba^*b(ab \cup ba)^*$, sobre $\Sigma = \{a, b\}$.
- ⑤ $(0 \cup 010)^*0^*(01 \cup 10)^*$, sobre $\Sigma = \{0, 1\}$.
- ⑥ $0^+1(010)^*(01 \cup 10)^*1^+$, sobre $\Sigma = \{0, 1\}$.
- ⑦ $\Sigma = \{a, b\}$. L = lenguaje de todas las cadenas sobre Σ que tienen un número par de aes y un número par de bes.

2.7. Equivalencia computacional entre los AFN- λ y los AFN

En esta sección se mostrará que el modelo AFN- λ es computacionalmente equivalente al modelo AFN. O dicho más gráficamente, las transiciones λ se pueden eliminar, añadiendo transiciones que las simulen, sin alterar el lenguaje aceptado.

En primer lugar, un AFN $M = (\Sigma, Q, q_0, F, \Delta)$ puede ser considerado como un AFN- λ en el que, simplemente, hay *cero* transiciones λ . Para la afirmación recíproca tenemos el siguiente teorema.

2.7.1 Teorema. *Dado un AFN- λ $M = (\Sigma, Q, q_0, F, \Delta)$, se puede construir un AFN M' equivalente a M , es decir, tal que $L(M) = L(M')$.*

Bosquejo de la demostración. Para construir M' a partir de M se requiere la noción de **λ -clausura de un estado**. Para un estado $q \in Q$, la λ -clausura de q , notada $\lambda[q]$, es el conjunto de estados de M a los que se puede llegar desde q por 0, 1 o más transiciones λ . Nótese que, en general, $\lambda[q] \neq \Delta(q, \lambda)$. Por definición, $q \in \lambda[q]$. La λ -clausura de un conjunto de estados $\{q_1, \dots, q_k\}$ se define por:

$$\lambda[\{q_1, \dots, q_k\}] := \lambda[q_1] \cup \dots \cup \lambda[q_k].$$

Además, $\lambda[\emptyset] := \emptyset$. Sea $M' = (\Sigma, Q, q_0, F', \Delta')$ donde

$$\begin{aligned} \Delta' : Q \times \Sigma &\longrightarrow \wp(Q) \\ (q, a) &\longmapsto \Delta'(q, a) := \lambda[\Delta(\lambda[q], a)]. \end{aligned}$$

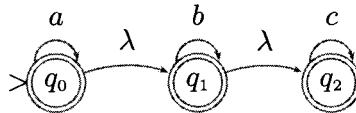
M' simula así las transiciones λ de M teniendo en cuenta todas las posibles trayectorias. F' se define como:

$$F' = \{q \in Q : \lambda[q] \text{ contiene al menos un estado de aceptación}\}.$$

Es decir, los estados de aceptación de M' incluyen los estados de aceptación de M y aquellos estados desde los cuales se puede llegar a un estado de aceptación por medio de una o más transiciones λ . \square

Como se puede apreciar, la construcción de M' a partir de M es puramente algorítmica.

Ejemplo Vamos a ilustrar el anterior algoritmo con el AFN- λ M , presentado en el segundo ejemplo de la sección 2.6.



$L(M) = a^*b^*c^*$. Las λ -clausuras de los estados vienen dadas por:

$$\lambda[q_0] = \{q_0, q_1, q_2\}.$$

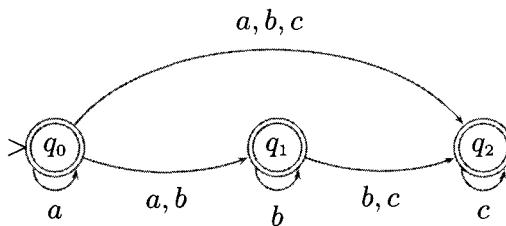
$$\lambda[q_1] = \{q_1, q_2\}.$$

$$\lambda[q_2] = \{q_2\}.$$

La función de transición $\Delta': Q \times \{a, b, c\} \rightarrow \wp(\{q_0, q_1, q_2\})$ es:

$$\begin{aligned}\Delta'(q_0, a) &= \lambda[\Delta(\lambda[q_0], a)] = \lambda[\Delta(\{q_0, q_1, q_2\}, a)] = \lambda[\{q_0\}] = \{q_0, q_1, q_2\}. \\ \Delta'(q_0, b) &= \lambda[\Delta(\lambda[q_0], b)] = \lambda[\Delta(\{q_0, q_1, q_2\}, b)] = \lambda[\{q_1\}] = \{q_1, q_2\}. \\ \Delta'(q_0, c) &= \lambda[\Delta(\lambda[q_0], c)] = \lambda[\Delta(\{q_0, q_1, q_2\}, c)] = \lambda[\{q_2\}] = \{q_2\}. \\ \Delta'(q_1, a) &= \lambda[\Delta(\lambda[q_1], a)] = \lambda[\Delta(\{q_1, q_2\}, a)] = \lambda[\emptyset] = \emptyset. \\ \Delta'(q_1, b) &= \lambda[\Delta(\lambda[q_1], b)] = \lambda[\Delta(\{q_1, q_2\}, b)] = \lambda[\{q_1\}] = \{q_1, q_2\}. \\ \Delta'(q_1, c) &= \lambda[\Delta(\lambda[q_1], c)] = \lambda[\Delta(\{q_1, q_2\}, c)] = \lambda[\{q_2\}] = \{q_2\}. \\ \Delta'(q_2, a) &= \lambda[\Delta(\lambda[q_2], a)] = \lambda[\Delta(\{q_2\}, a)] = \lambda[\emptyset] = \emptyset. \\ \Delta'(q_2, b) &= \lambda[\Delta(\lambda[q_2], b)] = \lambda[\Delta(\{q_2\}, b)] = \lambda[\emptyset] = \emptyset. \\ \Delta'(q_2, c) &= \lambda[\Delta(\lambda[q_2], c)] = \lambda[\Delta(\{q_2\}, c)] = \lambda[\{q_2\}] = \{q_2\}.\end{aligned}$$

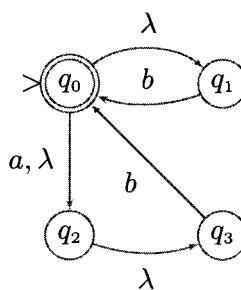
El autómata M' así obtenido es el siguiente:



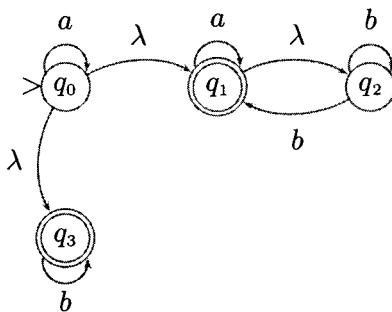
Ejercicios de la sección 2.7

Construir AFN equivalentes a los siguientes AFN- λ :

①



②



2.8. Teorema de Kleene. Parte I

En las secciones anteriores se ha mostrado la equivalencia computacional de los modelos AFD, AFN y AFN- λ , lo cual puede ser descrito en la forma:

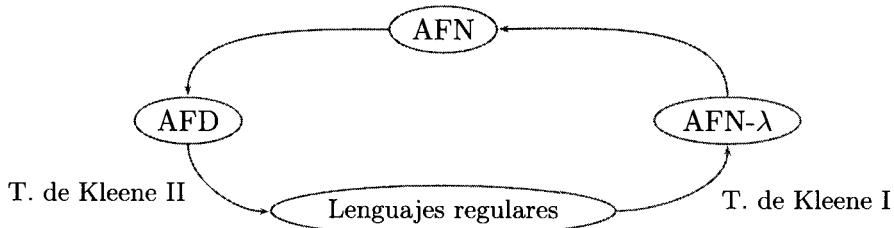
$$\boxed{\text{AFD} \equiv \text{AFN} \equiv \text{AFN-}\lambda}$$

Esto quiere decir que para cada autómata de uno de estos tres modelos se pueden construir autómatas equivalentes en los otros modelos. Por lo tanto, los modelos AFD, AFN y AFN- λ aceptan exactamente los mismos lenguajes. El Teorema de Kleene establece que tales lenguajes son precisamente los lenguajes regulares.

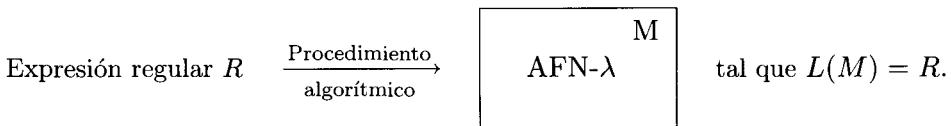
2.8.1. Teorema de Kleene. *Un lenguaje es regular si y sólo si es aceptado por un autómata finito (AFD o AFN o AFN- λ).*

Para demostrar el teorema consideraremos las dos direcciones por separado. Primero demostraremos que para un lenguaje regular L dado existe un AFN- λ tal que $L(M) = L$. En la sección 2.11 demostraremos que, a partir de un AFD M , se puede encontrar una expresión regular R tal que $L(M) = R$. En ambas direcciones las demostraciones son constructivas.

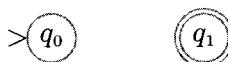
Las construcciones de este capítulo se pueden presentar así:



Parte I. Dada una expresión regular R sobre un alfabeto Σ , se puede construir un AFN- λ M tal que el lenguaje aceptado por M sea exactamente el lenguaje representado por R .



Demostración. Puesto que la definición de las expresiones regulares se hace recursivamente, la demostración se lleva a cabo razonando por inducción sobre R . Para las expresiones regulares básicas, podemos construir fácilmente autómatas que acepten los lenguajes representados. Así, el autómata



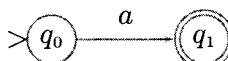
acepta el lenguaje \emptyset , es decir, el lenguaje representado por la expresión regular $R = \emptyset$.

El autómata



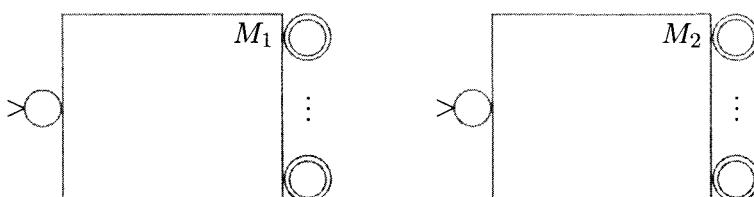
acepta el lenguaje $\{\lambda\}$, es decir, el lenguaje representado por la expresión regular $R = \lambda$.

El autómata



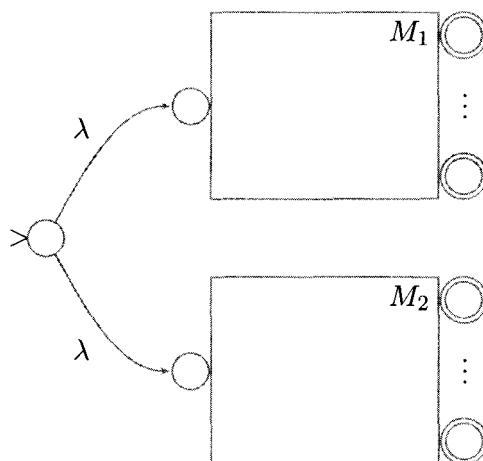
acepta el lenguaje $\{a\}$, $a \in \Sigma$, es decir, el lenguaje representado por la expresión regular $R = a$.

Paso inductivo: supóngase que para las expresiones regulares R y S existen AFN- λ M_1 y M_2 tales que $L(M_1) = R$ y $L(M_2) = S$. Esquemáticamente vamos a presentar los autómatas M_1 y M_2 en la siguiente forma:

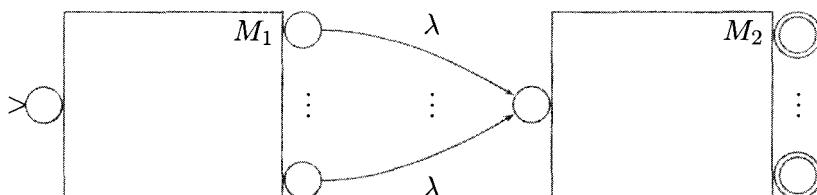


Los estados finales o de aceptación se dibujan a la derecha, pero cabe advertir que el estado inicial puede ser también un estado de aceptación. Obviando ese detalle, podemos ahora obtener AFN- λ que acepten los lenguajes $R \cup S$, RS y R^* .

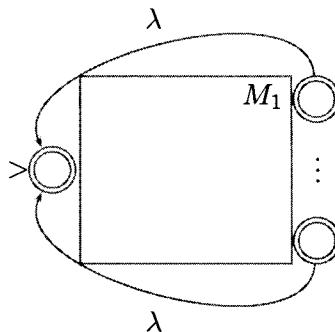
- Autómata que acepta $R \cup S$. Los autómatas M_1 y M_2 se conectan en paralelo y los estados finales del nuevo autómata son los estados finales de M_1 junto con los de M_2 :



- Autómata que acepta RS . Los autómatas M_1 y M_2 se conectan en serie y los estados finales del nuevo autómata son únicamente los estados finales de M_2 :



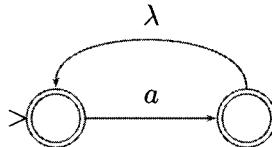
- Autómata que acepta R^* . Los estados finales del nuevo autómata son los estados finales de M_1 junto con el estado inicial.



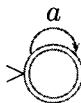
Esto concluye la demostración de la parte I del Teorema de Kleene. En la siguiente sección se presentan ejemplos concretos del procedimiento utilizado en la demostración. \square

2.9. Ejemplos de la parte I del Teorema de Kleene

De acuerdo con las construcciones presentadas en la demostración de la parte I del Teorema de Kleene, un AFN- λ que acepta el lenguaje a^* es:



Para simplificar las próximas construcciones utilizaremos, en su lugar, el bucle de un estado:



Ejemplo Vamos a utilizar el procedimiento del teorema para construir un AFN- λ que acepte el lenguaje $a^*(ab \cup ba)^* \cup a(b \cup a^*)$ sobre el alfabeto $\{a, b\}$.

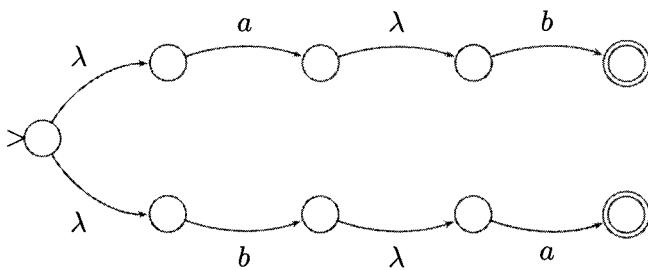
Autómata que acepta ab :



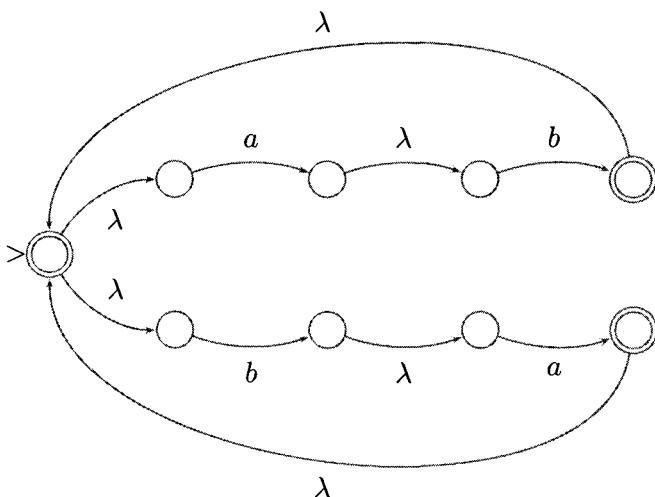
Autómata que acepta ba :



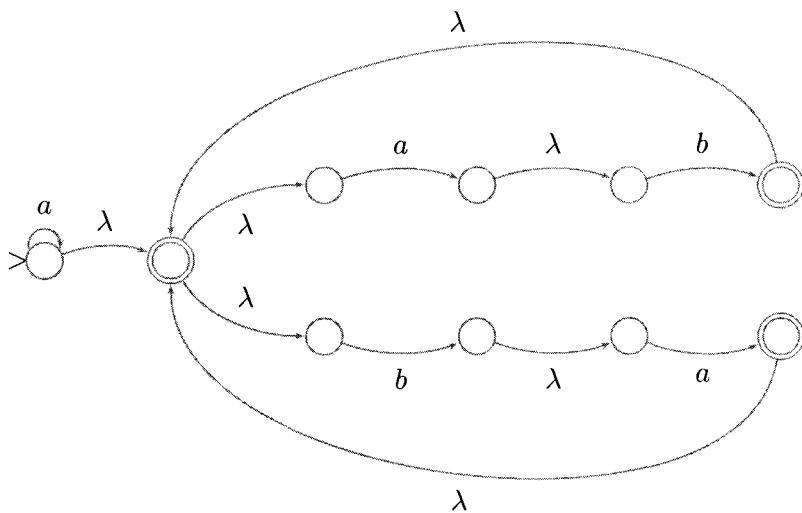
Autómata que acepta $ab \cup ba$:



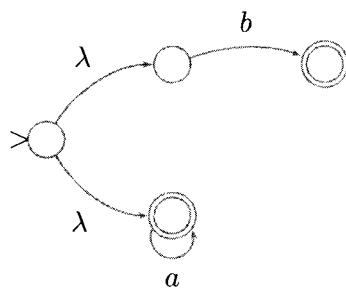
Autómata que acepta $(ab \cup ba)^*$:



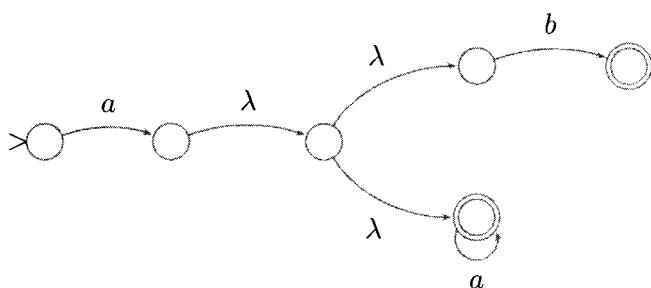
Autómata que acepta $a^*(ab \cup ba)^*$:



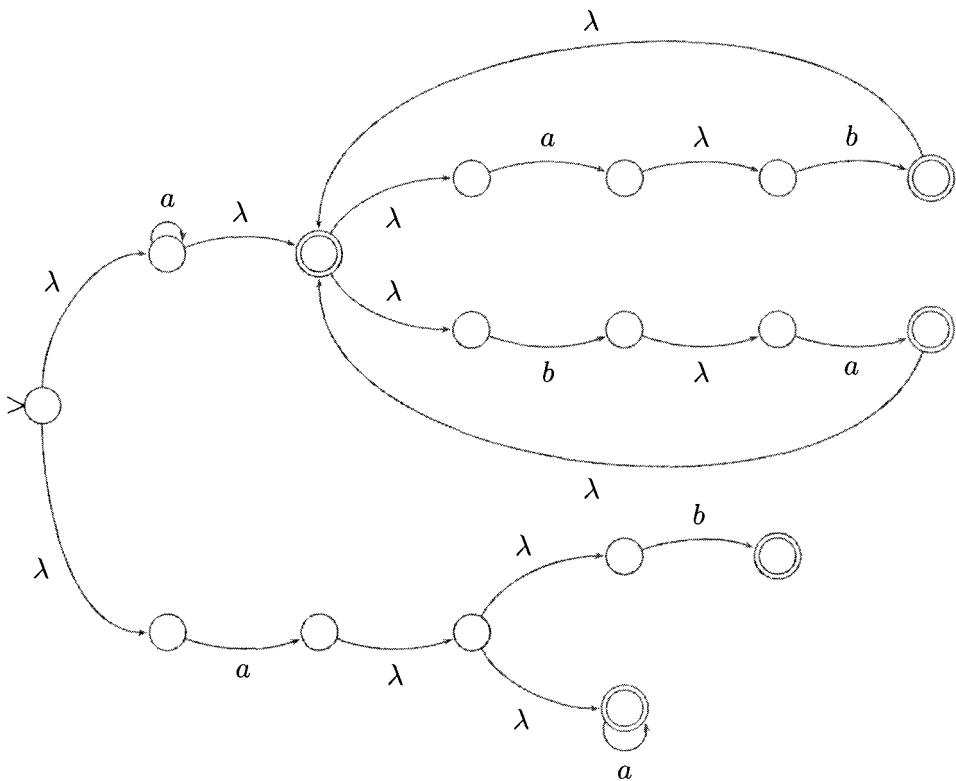
Autómata que acepta $b \cup a^*$:



Autómata que acepta $a(b \cup a^*)$:



Autómata que acepta $a^*(ab \cup ba)^* \cup a(b \cup a^*)$:



Ejercicios de la sección 2.9

Diseñar autómatas AFN- λ que acepten los siguientes lenguajes sobre el alfabeto $\Sigma = \{a, b, c\}$:

- ① $a^*(b \cup ab^* \cup ab^*a)c^* \cup (a \cup b)(a \cup ac)^*$.
- ② $c^*a(a \cup ba)^*(abc)^* \cup c^*(a \cup cb^*c)$.
- ③ $(ac)^* \cup a(a \cup ab^*a) \cup (abc)^*(cba)^* \cup (c \cup ab \cup ba \cup ca)^*(ca \cup cb)^*$.

2.10. Lema de Arden

Vamos a utilizar el siguiente resultado, conocido como “lema de Arden”, para demostrar la segunda parte del Teorema de Kleene.

2.10.1. Lema de Arden. Si A y B son lenguajes sobre un alfabeto Σ y $\lambda \notin A$, entonces la ecuación $X = AX \cup B$ tiene una única solución dada por $X = A^*B$.

Demostración. Si X es una solución de $X = AX \cup B$, entonces $B \subseteq AX \cup B = X$. También se tiene $AX \subseteq X$; a partir de esta contenencia y usando inducción sobre n , se puede demostrar que $A^nX \subseteq X$ para todo $n \in \mathbb{N}$. Por lo tanto

$$A^nB \subseteq A^nX \subseteq X$$

para todo $n \in \mathbb{N}$. Así que

$$A^*B = \left(\bigcup_{n \geq 0} A^n \right) B = \bigcup_{n \geq 0} A^nB \subseteq X.$$

Esto muestra que toda solución de $X = AX \cup B$ contiene a A^*B y es fácil verificar que, de hecho, A^*B es una solución:

$$A(A^*B) \cup B = A^+B \cup B = (A^+ \cup \lambda)B = A^*B.$$

Para la unicidad, demostraremos que si $A^*B \cup C$, con $C \cap A^*B = \emptyset$, es una solución de la ecuación, entonces $C = \emptyset$.

$$\begin{aligned} A^*B \cup C &= A(A^*B \cup C) \cup B \\ &= A^+B \cup AC \cup B \\ &= (A^+ \cup \lambda)B \cup AC \\ &= A^*B \cup AC. \end{aligned}$$

Intersectando con C ambos lados de la anterior igualdad, se tiene:

$$\begin{aligned} (A^*B \cap C) \cup C &= (A^*B \cap C) \cup (AC \cap C), \\ C &= AC \cap C. \end{aligned}$$

Por lo tanto, $C \subseteq AC$. Si se tuviera $C \neq \emptyset$, existiría una cadena $u \in C$ de longitud mínima. Entonces $u = vw$, con $v \in A$, $w \in C$. Como $\lambda \notin A$, $v \neq \lambda$; por consiguiente $|w| < |u|$. Esta contradicción muestra que necesariamente $C = \emptyset$, tal como se quería. \square

Ejemplo La ecuación $X = aX \cup b^*ab$ tiene solución única $X = a^*b^*ab$.

Ejemplo La ecuación $X = a^2X \cup b^+X \cup ab$ se puede escribir en la forma $X = (a^2 \cup b^+)X \cup ab$. Por el lema de Arden la ecuación tiene solución única $X = (a^2 \cup b^+)^*ab$.

Ejemplo La ecuación $X = ab^2X \cup aX \cup a^*b \cup b^*a$ se puede escribir como $X = (ab^2 \cup a)X \cup (a^*b \cup b^*a)$. Por lema de Arden la ecuación tiene solución única $X = (ab^2 \cup a)^*(a^*b \cup b^*a)$.

Ejercicios de la sección 2.10

① Encontrar las soluciones (únicas) de las siguientes ecuaciones:

- (i) $X = aX \cup bX$.
- (ii) $X = aX \cup b^*ab \cup bX \cup a^*$.

!② Demostrar de si $\lambda \in A$, entonces Y es una solución de la ecuación $X = AX \cup B$ si y solo si $Y = A^*(B \cup D)$ para algún $D \subseteq \Sigma^*$.

2.11. Teorema de Kleene. Parte II

En esta sección demostraremos que para todo AFN $M = (\Sigma, Q, q_0, F, \Delta)$ existe una expresión regular R tal que $L(M) = R$.

Un autómata tiene un único estado inicial pero cambiando dicho estado surgen nuevos autómatas. Para cada $q_i \in Q$, sea M_i el autómata que coincide con M pero con estado inicial q_i ; más precisamente, $M_i = (\Sigma, Q, q_i, F, \Delta)$. Al lenguaje aceptado por M_i lo denotaremos A_i ; es decir, $L(M_i) = A_i$. En particular, $A_0 = L(M)$. Puesto que los estados de aceptación no se han alterado, se tiene que

$$A_i = \{w \in \Sigma^* : \Delta(q_i, w) \cap F \neq \emptyset\}.$$

Cada A_i se puede escribir como

$$(2.1) \quad A_i = \begin{cases} \bigcup_{a \in \Sigma} \{aA_j : q_j \in \Delta(q_i, a)\}, & \text{si } q_i \notin F, \\ \bigcup_{a \in \Sigma} \{aA_j : q_j \in \Delta(q_i, a)\} \cup \lambda. & \text{si } q_i \in F. \end{cases}$$

Si $Q = \{q_0, q_1, \dots, q_n\}$, las igualdades de la forma (2.1) dan lugar a un sistema de $n + 1$ ecuaciones con $n + 1$ incógnitas (los A_i):

$$\begin{cases} A_0 = C_{01}A_0 \cup C_{02}A_1 \cup \dots \cup C_{0n}A_n & (\cup \lambda) \\ A_1 = C_{11}A_0 \cup C_{12}A_1 \cup \dots \cup C_{1n}A_n & (\cup \lambda) \\ \vdots \\ A_n = C_{n1}A_0 \cup C_{n2}A_1 \cup \dots \cup C_{nn}A_n & (\cup \lambda) \end{cases}$$

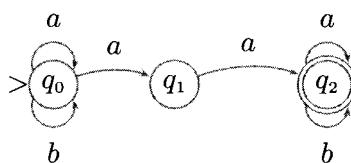
donde cada coeficiente C_{ij} o es \emptyset o es un símbolo de Σ . El término λ se añade a una ecuación solamente si el estado correspondiente es un estado de aceptación.

Utilizando sucesivas veces el lema de Arden, se puede mostrar que este sistema de ecuaciones siempre se puede solucionar y su solución es única. En efecto, comenzando con la última ecuación, se escribe A_n en términos de los demás A_i , para lo cual se usa el lema de Arden si es necesario. Este valor de A_n se reemplaza en las demás ecuaciones y el sistema se reduce a n ecuaciones con n incógnitas. Similarmente, A_{n-1} se escribe en términos de los demás A_i , usando el lema de Arden si es necesario, y tal valor se reemplaza en las ecuaciones restantes. Prosiguiendo de esta manera, el sistema original se reduce a una sola ecuación cuya incógnita es precisamente A_0 . Esta ecuación se soluciona recurriendo una vez más al lema de Arden. Puesto que los coeficientes C_{ij} diferentes de \emptyset son símbolos de Σ , se obtiene una expresión regular R tal que $L(M) = A_0 = R$.

2.12. Ejemplos de la parte II del Teorema de Kleene

A continuación ilustraremos el procedimiento de la sección 2.11 para encontrar $L(M)$ a partir de un AFN $M = (\Sigma, Q, q_0, F, \Delta)$ dado.

Ejemplo Considérese el siguiente AFN M :



Por simple inspección sabemos que $L(M) = (a \cup b)^*a^2(a \cup b)^*$, pero utilizaremos el método descrito para encontrar explícitamente $L(M)$.

El sistema de ecuaciones asociado con el autómata M es:

$$\begin{cases} (1) & A_0 = aA_0 \cup bA_0 \cup aA_1 \\ (2) & A_1 = aA_2 \\ (3) & A_2 = aA_2 \cup bA_2 \cup \lambda. \end{cases}$$

La ecuación (3) se puede escribir como

$$(4) \quad A_2 = (a \cup b)A_2 \cup \lambda.$$

Aplicando el lema de Arden en (4):

$$(5) \quad A_2 = (a \cup b)^*\lambda = (a \cup b)^*.$$

Reemplazando (5) en (2):

$$(6) \quad A_1 = a(a \cup b)^*.$$

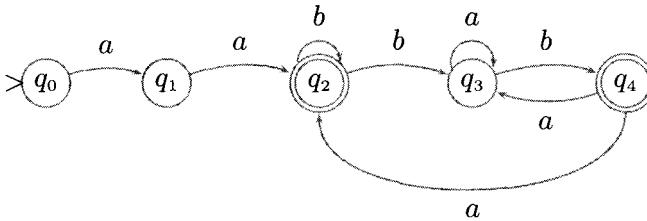
Reemplazando (6) en (1):

$$(7) \quad A_0 = (a \cup b)A_0 \cup a^2(a \cup b)^*.$$

Aplicando el lema de Arden en (7) concluimos:

$$A_0 = (a \cup b)^*a^2(a \cup b)^*$$

Ejemplo Encontrar una expresión regular para el lenguaje aceptado por el siguiente AFN M :



El sistema de ecuaciones asociado con el autómata M es:

$$\begin{cases} (1) & A_0 = aA_1 \\ (2) & A_1 = aA_2 \\ (3) & A_2 = bA_2 \cup bA_3 \cup \lambda \\ (4) & A_3 = aA_3 \cup bA_4 \\ (5) & A_4 = aA_2 \cup aA_3 \cup \lambda \end{cases}$$

Reemplazando (5) en (4):

$$(6) \quad A_3 = aA_3 \cup baA_2 \cup baA_3 \cup b = (a \cup ba)A_3 \cup baA_2 \cup b.$$

Aplicando el lema de Arden en (6):

$$(7) \quad A_3 = (a \cup ba)^*(baA_2 \cup b) = (a \cup ba)^*baA_2 \cup (a \cup ba)^*b.$$

Reemplazando (7) en (3):

$$(8) \quad A_2 = bA_2 \cup b(a \cup ba)^*baA_2 \cup b(a \cup ba)^*b \cup \lambda.$$

El sistema original de cinco ecuaciones y cinco incógnitas se reduce al sistema de tres ecuaciones y tres incógnitas formado por (1), (2) y (8).

La ecuación (8) se puede escribir como

$$(9) \quad A_2 = [b \cup b(a \cup ba)^*ba]A_2 \cup b(a \cup ba)^* \cup \lambda.$$

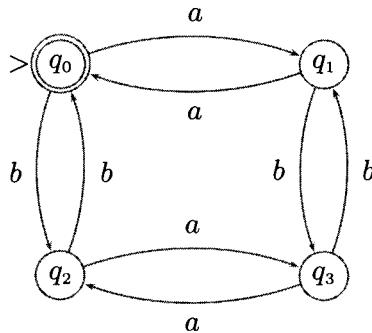
Aplicando el lema de Arden en (9):

$$(10) \quad A_2 = [b \cup b(a \cup ba)^*ba]^*[b(a \cup ba)^*b \cup \lambda].$$

Si se sustituye (10) en (2) y luego el valor de A_1 obtenido se sustituye en (1), se obtiene finalmente:

$$A_0 = a^2[b \cup b(a \cup ba)^*ba]^*[b(a \cup ba)^*b \cup \lambda]$$

Ejemplo Encontrar una expresión regular para el lenguaje L de todas las cadenas sobre $\Sigma = \{a, b\}$ que tienen un número par de aes y un número par de bes. El siguiente autómata acepta el lenguaje L :



Este autómata da lugar al siguiente sistema de ecuaciones:

$$\begin{cases} (1) & A_0 = aA_1 \cup bA_2 \cup \lambda \\ (2) & A_1 = aA_0 \cup bA_3 \\ (3) & A_2 = aA_3 \cup bA_0 \\ (4) & A_3 = aA_2 \cup bA_1 \end{cases}$$

Reemplazando (4) en (3):

$$(5) \quad A_2 = a^2A_2 \cup abA_1 \cup bA_0.$$

Reemplazando (4) en (2):

$$(6) \quad A_1 = aA_0 \cup baA_2 \cup b^2A_1.$$

El sistema original de cuatro ecuaciones y cuatro incógnitas se reduce a un sistema de tres ecuaciones y tres incógnitas, a saber:

$$\begin{cases} (1) & A_0 = aA_1 \cup bA_2 \cup \lambda \\ (6) & A_1 = aA_0 \cup baA_2 \cup b^2A_1 \\ (5) & A_2 = a^2A_2 \cup abA_1 \cup bA_0 \end{cases}$$

Aplicando el lema de Arden en (5):

$$(7) \quad A_2 = (a^2)^*(abA_1 \cup bA_0) = (a^2)^*abA_1 \cup (a^2)^*bA_0.$$

Reemplazando (7) en (6):

$$(8) \quad A_1 = aA_0 \cup ba(a^2)^*abA_1 \cup ba(a^2)^*bA_0 \cup b^2A_1.$$

Reemplazando (7) en (1):

$$(9) \quad A_0 = aA_1 \cup b(a^2)^*abA_1 \cup b(a^2)^*bA_0 \cup \lambda.$$

El sistema se reduce ahora a dos ecuaciones:

$$\begin{cases} (9) \quad A_0 = aA_1 \cup b(a^2)^*abA_1 \cup b(a^2)^*bA_0 \cup \lambda \\ (8) \quad A_1 = aA_0 \cup ba(a^2)^*abA_1 \cup ba(a^2)^*bA_0 \cup b^2A_1 \\ \qquad\qquad\qquad = (ba(a^2)^*ab \cup b^2)A_1 \cup aA_0 \cup ba(a^2)^*bA_0. \end{cases}$$

Aplicando el lema de Arden en (8):

$$(10) \quad \begin{aligned} A_1 &= (ba(a^2)^*ab \cup b^2)^*(aA_0 \cup ba(a^2)^*bA_0) \\ &= (ba(a^2)^*ab \cup b^2)^*aA_0 \cup (ba(a^2)^*ab \cup b^2)^*ba(a^2)^*bA_0. \end{aligned}$$

Haciendo $R = (ba(a^2)^*ab \cup b^2)^*$, (10) se puede escribir como

$$(11) \quad A_1 = RaA_0 \cup Rba(a^2)^*bA_0.$$

Aplicando el lema de Arden en (9):

$$(12) \quad \begin{aligned} A_0 &= (b(a^2)^*b)^*(aA_1 \cup b(a^2)^*abA_1 \cup \lambda) \\ &= (b(a^2)^*b)^*aA_1 \cup (b(a^2)^*b)^*b(a^2)^*abA_1 \cup (b(a^2)^*b)^*. \end{aligned}$$

Haciendo $S = (b(a^2)^*b)^*$, (12) se puede escribir como:

$$(13) \quad A_0 = SaA_1 \cup Sb(a^2)^*abA_1 \cup S.$$

Al sustituir (11) en (13), el sistema original se reduce a una sola ecuación:

$$(14) \quad A_0 = Sa[RaA_0 \cup Rba(a^2)^*bA_0] \cup Sb(a^2)^*ab[RaA_0 \cup Rba(a^2)^*bA_0] \cup S.$$

Agrupando los términos en los que aparece A_0 y factorizando, se obtiene

$$(15) \quad A_0 = [SaRa \cup SaRba(a^2)^*b \cup Sb(a^2)^*abRa \cup Sb(a^2)^*abRba(a^2)^*b]A_0 \cup S.$$

Aplicando lema de Arden en (15):

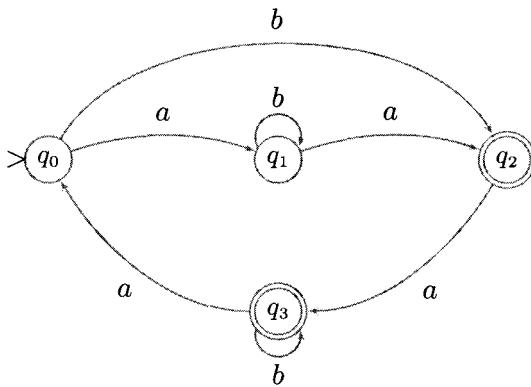
$$(16) \quad A_0 = [SaRa \cup SaRba(a^2)^*b \cup Sb(a^2)^*abRa \cup Sb(a^2)^*abRba(a^2)^*b]^*S.$$

Si sustituimos R y S en (16) obtenemos una expresión regular para L .

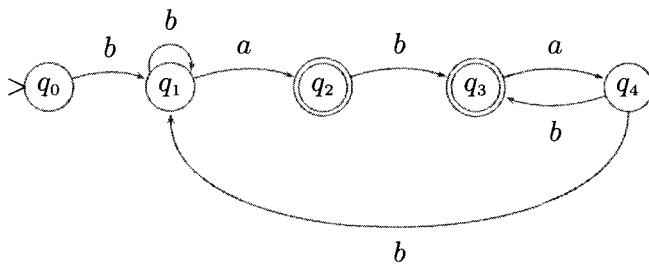
Ejercicios de la sección 2.12

- ① Utilizando el lema de Arden, encontrar expresiones regulares para los siguientes lenguajes sobre $\Sigma = \{a, b\}$:
 - (i) El lenguaje L de todas las cadenas que tienen un número par de a s y un número impar de b s.
 - (ii) El lenguaje L de todas las cadenas que tienen un número par de a s o un número impar de b s.
- ② Utilizando el lema de Arden, encontrar expresiones regulares para los lenguajes aceptados por los siguientes AFN:

(i)



(ii)



Capítulo 3

Otras propiedades de los lenguajes regulares

En los dos capítulos anteriores hemos presentado las propiedades básicas de los lenguajes regulares pero no hemos visto cómo se puede demostrar que un lenguaje no es regular. El llamado “lema de bombeo”, expuesto en este capítulo, sirve para tal propósito. También veremos que la regularidad es una propiedad que se preserva por las operaciones booleanas usuales, por homomorfismos y por las imágenes inversas de homomorfismos. Finalmente, analizaremos ciertos problemas de decisión referentes a autómatas y a lenguajes regulares.

3.1. Lema de bombeo

El llamado “lema de bombeo” (*pumping lemma*, en inglés) es una propiedad de los lenguajes regulares que es muy útil para demostrar que ciertos lenguajes no son regulares.

3.1.1. Lema de bombeo. *Para todo lenguaje regular L (sobre un alfabeto dado Σ) existe una constante $n \in \mathbb{N}$, llamada constante de bombeo para L , tal que toda cadena $w \in L$, con $|w| \geq n$, satisface la siguiente propiedad:*

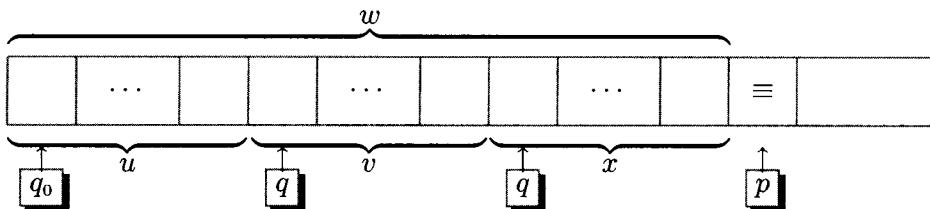
$$(B) \quad \left\{ \begin{array}{l} w \text{ se puede descomponer como } w = uvx, \text{ con } |uv| \leq n, v \neq \lambda, \\ \text{y para todo } i \geq 0 \text{ se tiene } uv^i x \in L. \end{array} \right.$$

Demostración. Por el Teorema de Kleene y por los teoremas de equivalencia de los modelos AFD, AFN y AFN- λ , existe un AFD M tal que $L(M) = L$.

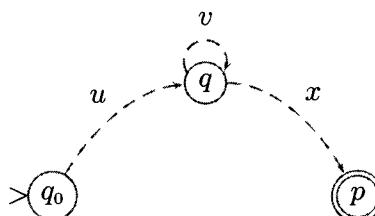
Sea

$$n = \# \text{ de estados de } M.$$

Si $w \in L$ y $|w| \geq n$, entonces durante el procesamiento completo de w , hay por lo menos un estado que se repite. Sea q el primer estado que se repite. Tal como se muestra en la siguiente gráfica, w se puede descomponer como $w = uvx$, donde $|uv| \leq n$, $v \neq \lambda$.



Nótese que tanto u como x pueden ser la cadena vacía λ , pero $v \neq \lambda$. Además, la cadena v se puede “bombeo”, en el sentido de que $uv^i x$ es aceptada por M para todo $i \geq 0$. En el diagrama de estados, se puede visualizar esta propiedad de bombeo de v :



Uso del lema de bombeo. El lema de bombeo se puede usar para concluir que un cierto lenguaje dado L no es regular, recurriendo a un razonamiento por contradicción (o reducción al absurdo). El razonamiento utilizado tiene la siguiente forma:

1. Si L fuera regular, existiría una constante de bombeo n para L .
2. Se escoge una cadena “adecuada” $w \in L$ y se aplica la propiedad (B) del lema de bombeo, descomponiendo w como

$$w = uvx, \quad v \neq \lambda, \quad |uv| \leq n.$$
3. Se llega a la siguiente contradicción:
 - (I) Por el lema de bombeo, $uv^i x \in L$, para todo $i \geq 0$.

(II) $uv^i x$ no puede estar en L , para algún $i \in I$. Por lo general, basta escoger valores pequeños de i como $i = 0$ ó $i = 2$.

Ejemplo Usar el lema de bombeo para demostrar que el lenguaje $L = \{a^i b^i : i \geq 0\}$ no es regular.

Solución. Si L fuera regular, existiría una constante de bombeo n para L . Sea $w = a^n b^n \in L$. Entonces w se puede descomponer como $w = uvx$, con $|v| \geq 1$ y $|uv| \leq n$. Por lo tanto, u y v constan únicamente de aes:

$$\begin{aligned} u &= a^r, && \text{para algún } r \geq 0, \\ v &= a^s, && \text{para algún } s \geq 1. \end{aligned}$$

Entonces,

$$x = a^{n-(r+s)} b^n = a^{n-r-s} b^n.$$

Por el lema de bombeo, $uv^i x \in L$ para todo $i \geq 0$. En particular, si $i = 0$, $ux \in L$. Pero $ux = a^r a^{n-r-s} b^n = a^{n-s} b^n$. Como $n - s \neq n$, la cadena $ux \notin L$ lo cual es una contradicción. Se concluye entonces que L no puede ser regular.

Tomando $i = 2$ también se llega a una contradicción: por un lado, $uv^2 x \in L$, pero

$$uv^2 x = a^r a^s a^s a^{n-r-s} b^n = a^{r+2s+n-r-s} b^n = a^{n+s} b^n.$$

Como $s \geq 1$, $a^{n+s} b^n$ no está en L .

El argumento anterior también sirve para demostrar que el lenguaje $L = \{a^i b^i : i \geq 1\}$ no es regular.

Ejemplo Demostrar que el lenguaje de los palíndromos sobre $\{a, b\}$ no es un lenguaje regular. Recuérdese que un **palíndromo** es una cadena w tal que $w = w^R$.

Solución. Si L fuera regular, existiría una constante de bombeo n para L . Sea $w = a^n b a^n \in L$. Entonces w se puede descomponer como $w = uvx$, con $|v| \geq 1$, $|uv| \leq n$, y para todo $i \geq 0$, $uv^i x \in L$. Por lo tanto, u y v constan únicamente de aes:

$$\begin{aligned} u &= a^r, && \text{para algún } r \geq 0, \\ v &= a^s, && \text{para algún } s \geq 1. \end{aligned}$$

Entonces,

$$x = a^{n-(r+s)} b a^n = a^{n-r-s} b a^n.$$

Tomando $i = 0$, se concluye que $ux \in L$, pero

$$ux = a^r a^{n-r-s} ba^n = a^{n-s} ba^n.$$

Como $s \geq 1$, $a^{n-s} ba^n$ no es un palíndromo. Esta contradicción muestra que L no puede ser regular.

Ejemplo Demostrar que el lenguaje $L = \{a^{i^2} : i \geq 0\}$ no es regular. L está formado por cadenas de aes cuya longitud es un cuadrado perfecto.

Solución. Si L fuera regular, existiría una constante de bombeo n para L . Sea $w = a^{n^2} \in L$. Entonces w se puede descomponer como $w = uvx$, con $|v| \geq 1$, $|uv| \leq n$. Por el lema de bombeo, $uv^2x \in L$, pero por otro lado,

$$n^2 < n^2 + |v| = |uvx| + |v| = |uv^2x| \leq n^2 + |uv| \leq n^2 + n < (n+1)^2.$$

Esto quiere decir que el número de símbolos de la cadena uv^2x no es un cuadrado perfecto y, por consiguiente, $uv^2x \notin L$. En conclusión, L no es regular.

Ejercicios de la sección 3.1

- ① Usar el lema de bombeo para demostrar que los siguientes lenguajes no son regulares:
 - (i) $L = \{w \in \{a, b\}^* : w$ tiene el mismo número de aes que de bes $\}$.
 - (ii) $L = \{a^i b a^i : i \geq 1\}$, sobre $\Sigma = \{a, b\}$.
 - (iii) $L = \{a^i b^j a^i : i, j \geq 0\}$, sobre $\Sigma = \{a, b\}$.
 - (iv) $L = \{0^i 1^{2i} : i \geq 0\}$, sobre $\Sigma = \{0, 1\}$.
 - (v) $L = \{1^i 0 1^i 0 : i \geq 1\}$, sobre $\Sigma = \{0, 1\}$.
 - (vi) $L = \{a^i b^j c^{i+j} : i, j \geq 0\}$, sobre $\Sigma = \{a, b, c\}$.
 - (vii) $L = \{a^i b^j : j > i \geq 0\}$, sobre $\Sigma = \{a, b\}$.
 - (viii) $L = \{ww : w \in \Sigma^*\}$, siendo $\Sigma = \{a, b\}$.
 - (ix) $L = \{ww^R : w \in \Sigma^*\}$, siendo $\Sigma = \{a, b\}$.
 - (x) $L = \{a^i : i$ es un número primo $\}$, sobre $\Sigma = \{a\}$.
- ② ¿Es $L = \{(ab)^i : i \geq 0\}$ un lenguaje regular?
- ③ Encontrar la falacia en el siguiente argumento: “Según la propiedad (B) del enunciado del lema de bombeo, se tiene que $uv^i x \in L$ para todo $i \geq 0$. Por consiguiente, L posee infinitas cadenas y, en conclusión, todo lenguaje regular es infinito.”

3.2. Propiedades de clausura

Las propiedades de clausura afirman que a partir de lenguajes regulares se pueden obtener otros lenguajes regulares por medio de ciertas operaciones entre lenguajes. Es decir, la regularidad es preservada por ciertas operaciones entre lenguajes; en tales casos se dice que *los lenguajes regulares son cerrados bajo las operaciones*.

El siguiente teorema establece que la colección $\mathcal{R} \subseteq \mathcal{P}(\Sigma^*)$ de los lenguajes regulares sobre un alfabeto Σ es cerrada bajo todas las operaciones booleanas.

3.2.1 Teorema. *Si L , L_1 y L_2 son lenguajes regulares sobre un alfabeto Σ , también lo son:*

- | | | |
|-----|-------------------------------|------------------------|
| (1) | $L_1 \cup L_2$ | (unión) |
| (2) | $L_1 L_2$ | (concatenación) |
| (3) | L^* | (estrella de Kleene) |
| (4) | L^+ | (clausura positiva) |
| (5) | $\overline{L} = \Sigma^* - L$ | (complemento) |
| (6) | $L_1 \cap L_2$ | (intersección) |
| (7) | $L_1 - L_2$ | (diferencia) |
| (8) | $L_1 \triangleleft L_2$ | (diferencia simétrica) |

Demostración.

(1), (2) y (3) se siguen de la definición de los lenguajes regulares.

(4) Por (2), (3) y $L^+ = L^* L$.

(5) Por el Teorema de Kleene y por los teoremas de equivalencia de los modelos AFD, AFN y AFN- λ , existe un AFD $M = (\Sigma, Q, q_0, F, \delta)$ tal que $L(M) = L$. Para construir un AFD que acepte el complemento de L basta intercambiar los estados finales con los no finales. Si M' es el autómata $(\Sigma, Q, q_0, Q - F, \delta)$, entonces $L(M') = \overline{L}$.

(6) Se sigue de (1) y (5) teniendo en cuenta que $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.

(7) Se sigue de (5) y (6) teniendo en cuenta que $L_1 - L_2 = L_1 \cap \overline{L_2}$.

(8) Puesto que

$$L_1 \triangleleft L_2 = (L_1 \cup L_2) - (L_1 \cap L_2) = (L_1 - L_2) \cup (L_2 - L_1)$$

el resultado se sigue de (1), (6), (7). □

Recuérdese que un **álgebra booleana de conjuntos** es una familia $\mathcal{A} \subseteq \mathcal{P}(X)$ cerrada bajo las operaciones de unión, intersección y complemento, tal que $\emptyset \in \mathcal{A}$, $X \in \mathcal{A}$.

3.2.2 Corolario. *La colección $\mathcal{R} \subseteq \mathcal{P}(\Sigma^*)$ de todos los lenguajes regulares sobre un alfabeto Σ es un álgebra booleana de conjuntos.*

Demostración. Se sigue del Teorema 3.2.1 y del hecho de que \emptyset y Σ^* son lenguajes regulares sobre Σ . \square

- ☞ Hemos visto que un lenguaje finito es regular y que la unión finita de lenguajes regulares es regular. Pero una unión infinita de lenguajes regulares no necesariamente es regular; considérese, por ejemplo,

$$L = \{a^n b^n : n \geq 1\} = \bigcup_{i \geq 1} \{a^i b^i\}.$$

Cada conjunto $\{a^i b^i\}$ es regular (porque posee sólo una cadena) pero L no lo es.

- ☞ Un sublenguaje (subconjunto) de un lenguaje regular no es necesariamente regular, es decir, la familia de los lenguajes regulares no es cerrada para subconjuntos. Dicho de otra forma, un lenguaje regular puede contener sublenguajes no-regulares. Por ejemplo, $L = \{a^n b^n : n \geq 1\}$ es un sublenguaje del lenguaje regular $a^* b^*$, pero L mismo no es regular.

Las propiedades de clausura permiten concluir, razonando por contradicción, que ciertos lenguajes no son regulares. Esto se ilustra en los siguientes ejemplos en los que se usa el hecho de que los lenguajes $L = \{a^i b^i : i \geq 0\}$ y $L = \{a^i b^i : i \geq 1\}$ no son regulares.

Ejemplo $L = \{a^i b^j : i, j \geq 0, i \neq j\}$ no es regular. Si lo fuera, $a^* b^* - L$ también lo sería, pero $a^* b^* - L = \{a^i b^i : i \geq 0\}$.

Ejemplo El lenguaje $L = \{wb^n : w \in \Sigma^*, |w| = n, n \geq 1\}$ sobre $\Sigma = \{a, b\}$ no es regular. Si L fuera regular, también lo sería $L \cap a^* b^*$ pero $L \cap a^* b^* = \{a^n b^n : n \geq 1\}$.

Ejercicios de la sección 3.2

- ① Demostrar que $a^* b^*$ es la unión de dos lenguajes disyuntos no-regulares.

- ② Sea L un lenguaje no-regular y N un subconjunto finito de L . Demostrar que $L - N$ tampoco es regular.
- ③ Demostrar o refutar las siguientes afirmaciones:
- (i) Un lenguaje no-regular debe ser infinito.
 - (ii) Si el lenguaje $L_1 \cup L_2$ es regular, también lo son L_1 y L_2 .
 - (iii) Si los lenguajes L_1 y L_2 no son regulares, el lenguaje $L_1 \cap L_2$ tampoco puede ser regular.
 - (iv) Si el lenguaje L^* es regular, también lo es L .
 - (v) Si L es regular y N es un subconjunto finito de L , entonces $L - N$ es regular.
 - (vi) Un lenguaje regular L es infinito si y sólo si en cualquier expresión regular de L aparece por lo menos una $*$.
- ④ Utilizar las propiedades de clausura para concluir que los siguientes lenguajes no son regulares:
- (i) $L = \{1^i 0 1^j 0 : i, j \geq 1, i \neq j\}$, sobre $\Sigma = \{0, 1\}$. Ayuda: utilizar el ejercicio 1(v) de la sección 3.1.
 - (ii) $L = \{uvu^R : u, v \in \{a, b\}^+\}$, sobre $\Sigma = \{a, b\}$. Ayuda: utilizar el ejercicio 1(ii) de la sección 3.1.
 - (iii) $L = \{u : |u| \text{ es un cuadrado perfecto}\}$, sobre $\Sigma = \{a, b, c\}$. Ayuda: utilizar el último ejemplo de la sección 3.1.

3.3. Propiedades de clausura para autómatas

Las propiedades de clausura del Teorema 3.2.1 se pueden enunciar como procedimientos algorítmicos para la construcción de autómatas finitos.

3.3.1 Teorema. Sean M , M_1 y M_2 autómatas finitos (ya sean AFD o AFN o AFN- λ) y $L(M) = L$, $L(M_1) = L_1$, $L(M_2) = L_2$. Se pueden construir autómatas finitos que acepten los siguientes lenguajes:

- | | | | |
|-----|------------------|-----|----------------------------|
| (1) | $L_1 \cup L_2$. | (5) | $\bar{L} = \Sigma^* - L$. |
| (2) | $L_1 L_2$. | (6) | $L_1 \cap L_2$. |
| (3) | L^* . | (7) | $L_1 - L_2$. |
| (4) | L^+ . | (8) | $L_1 \triangleleft L_2$. |

Demostración. La construcción de autómatas para $L_1 \cup L_2$, $L_1 L_2$, L^* y L^+ se presentó en la demostración de la parte I del Teorema de Kleene. En el

numeral (5) del Teorema 3.2.1 se vio cómo se puede construir un AFD M' que acepte \overline{L} a partir de un AFD M que acepte L .

Los procedimientos de construcción de (1), (2), (3), (4) y (5) se pueden combinar para obtener autómatas que acepten los lenguajes $L_1 \cap L_2$, $L_1 - L_2$ y $L_1 \triangleleft L_2$. \square

Para diseñar un autómata que acepte $L_1 \cap L_2$, según el argumento del Teorema 3.3.1, hay que usar la igualdad $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ y los procedimientos para unión, complemento, eliminación de transiciones λ y eliminación del no-determinismo. El siguiente teorema muestra que existe una construcción más sencilla para el caso $L_1 \cap L_2$.

3.3.2 Teorema. Sean $M_1 = (\Sigma, Q_1, q_1, F_1, \delta_1)$ y $M_2 = (\Sigma, Q_2, q_2, F_2, \delta_2)$ dos AFD. Entonces el AFD

$$M = (\Sigma, Q_1 \times Q_2, (q_1, q_2), F_1 \times F_2, \delta)$$

donde

$$\begin{aligned} \delta : (Q_1 \times Q_2) \times \Sigma &\longrightarrow Q_1 \times Q_2 \\ \delta((q_i, q_j), a) &= (\delta_1(q_i, a), \delta_2(q_j, a)) \end{aligned}$$

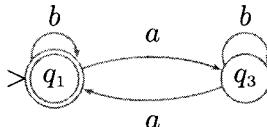
satisface $L(M) = L(M_1) \cap L(M_2)$.

Demostración. Sea $w \in \Sigma^*$. La conclusión del teorema se sigue demostrando primero por inducción sobre w que $\delta((q_1, q_2), w) = (\delta_1(q_1, w), \delta_2(q_2, w))$ para toda cadena $w \in \Sigma^*$ (aquí se usan las funciones extendidas de δ , δ_1 y δ_2 , según la definición 2.5.2). Finalmente, se observa que:

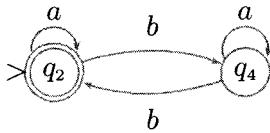
$$\begin{aligned} w \in L(M) &\iff \delta((q_1, q_2), w) \in F_1 \times F_2 \\ &\iff (\delta_1(q_1, w), \delta_2(q_2, w)) \in F_1 \times F_2 \\ &\iff \delta_1(q_1, w) \in F_1 \quad \& \quad \delta_2(q_2, w) \in F_2 \\ &\iff w \in L(M_1) \quad \& \quad w \in L(M_2) \\ &\iff w \in L(M_1) \cap L(M_2). \quad \square \end{aligned}$$

Ejemplo Utilizar el Teorema 3.3.2 para construir un AFD que acepte el lenguaje L de todas las cadenas sobre $\Sigma = \{a, b\}$ que tienen un número par de aes y un número par de bes.

AFD M_1 que acepta las cadenas con un número par de aes:



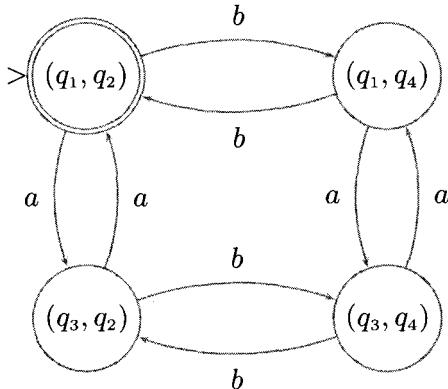
AFD M_2 que acepta las cadenas con un número par de b s:



Entonces $L = L(M_1) \cap L(M_2)$. El nuevo autómata tiene 4 estados: (q_1, q_2) , (q_1, q_4) , (q_3, q_2) y (q_3, q_4) ; el único estado de aceptación es (q_1, q_2) . Su función de transición δ es

$$\begin{aligned}\delta((q_1, q_2), a) &= (\delta_1(q_1, a), \delta_2(q_2, a)) = (q_3, q_2) \\ \delta((q_1, q_2), b) &= (\delta_1(q_1, b), \delta_2(q_2, b)) = (q_1, q_4) \\ \delta((q_1, q_4), a) &= (\delta_1(q_1, a), \delta_2(q_4, a)) = (q_3, q_4) \\ \delta((q_1, q_4), b) &= (\delta_1(q_1, b), \delta_2(q_4, b)) = (q_1, q_2) \\ \delta((q_3, q_2), a) &= (\delta_1(q_3, a), \delta_2(q_2, a)) = (q_1, q_2) \\ \delta((q_3, q_2), b) &= (\delta_1(q_3, b), \delta_2(q_2, b)) = (q_3, q_4) \\ \delta((q_3, q_4), a) &= (\delta_1(q_3, a), \delta_2(q_4, a)) = (q_1, q_4) \\ \delta((q_3, q_4), b) &= (\delta_1(q_3, b), \delta_2(q_4, b)) = (q_3, q_2)\end{aligned}$$

El diagrama de estados del autómata así obtenido es:



Ejercicios de la sección 3.3

- ① Utilizar el Teorema 3.3.2 para construir AFD que acepten los siguientes lenguajes sobre el alfabeto $\{a, b, c\}$:
 - (i) El lenguaje L de todas las cadenas que tienen longitud par y terminan en a .

- (ii) El lenguaje L de todas las cadenas de longitud par que tengan un número impar de *b*s.
 - (iii) El lenguaje L de todas las cadenas de longitud impar que tengan un número par de *c*s.
 - (iv) El lenguaje L de todas las cadenas de longitud impar que tengan exactamente dos *a*s.
- ② Utilizar el procedimiento del Teorema 3.3.1 para construir un autómata que acepte el lenguaje L de todas las cadenas sobre $\Sigma = \{a, b\}$ que tienen un número par de *a*s y un número par de *b*s. Compárese con el AFD construido en el último ejemplo de esta sección.

3.4. Homomorfismos \mathbb{X}

Un homomorfismo es una sustitución h que reemplaza cada símbolo a de un alfabeto de Σ por una cadena $h(a) \in \Gamma^*$, donde Γ es otro alfabeto (por supuesto, Σ y Γ pueden ser el mismo alfabeto). Más precisamente, un homomorfismo es una función $h : \Sigma^* \rightarrow \Gamma^*$ tal que $h(\lambda) = \lambda$ y para toda cadena $u = a_1a_2 \cdots a_n$, con $a_i \in \Sigma$, se tiene

$$h(a_1a_2 \cdots a_n) = h(a_1)h(a_2) \cdots h(a_n).$$

Un homomorfismo h está completamente determinado por sus imágenes en los símbolos de Σ , es decir, por los valores $h(a)$, con $a \in \Sigma$.

Ejemplo Sean $\Sigma = \{a, b, c\}$, $\Gamma = \{0, 1\}$ y $h : \Sigma^* \rightarrow \Gamma^*$ el homomorfismo definido por $h(a) = 0$, $h(b) = 1$ y $h(c) = 010$. El homomorfismo h convierte cadenas de Σ^* en cadenas de Γ^* siguiendo las siguientes reglas: cada a se reemplaza por 0, cada b por 1 y cada c se reemplaza por la cadena 010. Así,

$$\begin{aligned} h(a^2b^2) &= h(aabb) = h(a)h(a)h(b)h(b) = 0011. \\ h(acb^2c) &= h(a)h(c)h(b)^2h(b) = 001011010. \end{aligned}$$

También se deduce fácilmente que $h(a^*b^*c^*) = 0^*1^*(010)^*$.

El siguiente teorema afirma que los homomorfismos preservan lenguajes regulares; dicho de otra manera, la imagen homomorfa de un lenguaje regular es un lenguaje regular.

3.4.1 Teorema. *Sea $h : \Sigma^* \rightarrow \Gamma^*$ un homomorfismo.*

(1) Para cadenas $u, v \in \Sigma^*$ y lenguajes $A, B \subseteq \Sigma^*$ se tiene

$$\begin{aligned} h(uv) &= h(u)h(v), \\ h(A \cup B) &= h(A) \cup h(B), \\ h(AB) &= h(A)h(B), \\ h(A^*) &= [h(A)]^*. \end{aligned}$$

(2) Si L es un lenguaje regular sobre Σ , entonces $h(L)$ es un lenguaje regular sobre Γ .

Demostración.

- (1) Se sigue directamente de la definición de homomorfismo; los detalles se dejan al estudiante.
- (2) Por inducción sobre el número de operandos (uniones, concatenaciones y estrellas) en la expresión regular que representa a L . La base de la inducción es inmediata ya que $h(\lambda) = \lambda$, para cada $a \in \Sigma$, $h(a)$ es una cadena determinada en Γ^* y $h(\emptyset) = \emptyset$.

La hipótesis de inducción afirma que si L es regular también lo es $h(L)$. Para el paso inductivo se supone, en tres casos, que $L = A \cup B$, $L = AB$ o $L = A^*$. Utilizando la hipótesis de inducción y la parte (1) del presente teorema se llega a la conclusión deseada. \square

La parte (2) del Teorema 3.4.1 es muy útil para demostrar que ciertos lenguajes no son regulares, razonando por contradicción. En los siguientes ejemplos ilustramos la técnica que se usa en estas situaciones.

Ejemplo Sabiendo que $\{a^i b^i : i \geq 1\}$ no es regular (sección 3.1), podemos concluir que $L = \{0^i 1^i : i \geq 1\}$ tampoco lo es. Razonamos de la siguiente manera: si L fuera regular, lo sería también $h(L)$ donde h es el homomorfismo $h(0) = a$, $h(1) = b$. Pero $h(L) = \{h(0)^i h(1)^i : i \geq 1\} = \{a^i b^i : i \geq 1\}$. Por consiguiente, L no es regular.

Ejemplo $L = \{0^n 21^n : n \geq 1\}$ no es regular; si lo fuera, $h(L)$ también lo sería, donde h es el homomorfismo $h(0) = 0$, $h(1) = 1$, $h(2) = \lambda$. Pero $h(L) = \{0^n 1^n : n \geq 1\}$ no es regular, como se dedujo en el ejemplo anterior.

Ejercicios de la sección 3.4

- ① Llenar los detalles de la demostración de la parte (1) del Teorema 3.4.1.

- ② Utilizar homomorfismos y el hecho de que los lenguajes $\{a^i b^i : i \geq 1\}$ y $\{0^i 1^i : i \geq 1\}$ no son regulares para concluir que los siguientes lenguajes tampoco lo son:
- $L = \{a^i b a^j : i, j \geq 1, i \neq j\}$, sobre $\Sigma = \{a, b\}$.
 - $L = \{a^i b^i c^i : i \geq 1\}$, sobre $\Sigma = \{a, b, c\}$.
 - $L = \{a^i b^j c^i : i, j \geq 1\}$, sobre $\Sigma = \{a, b, c\}$.
 - $L = \{0^i 1^j 2^k : i, j, k \geq 0, i + j = k\}$, sobre $\Sigma = \{a, b, c\}$.

3.5. Imagen inversa de un homomorfismo

Dado un homomorfismo $h : \Sigma^* \rightarrow \Gamma^*$ y un lenguaje $B \subseteq \Gamma^*$, la **imagen inversa de A por h** es

$$h^{-1}(B) := \{u \in \Sigma^* : h(u) \in B\}.$$

La regularidad es también preservada por imágenes inversas de homomorfismos, tal como lo afirma el siguiente teorema.

3.5.1 Teorema. *Sea $h : \Sigma^* \rightarrow \Gamma^*$ un homomorfismo y $B \subseteq \Gamma^*$ un lenguaje regular sobre Γ . La imagen inversa $h^{-1}(B)$ es un lenguaje regular sobre Σ .*

Demostración. Comenzando con un AFD $M = (\Gamma, Q, q_0, F, \delta)$ que acepte a B podemos construir un AFD $M' = (\Sigma, Q, q_0, F, \delta')$ que acepte $h^{-1}(B)$. La función de transición δ' se define mediante $\delta'(q, a) = \delta(q, h(a))$ (aquí se usa la función extendida δ , según la definición 2.5.2). No es difícil demostrar por inducción sobre w que $\delta'(q_0, w) = \delta(q_0, h(w))$, para toda cadena $w \in \Sigma^*$. Como los estados de aceptación de M y M' coinciden, M' acepta a w si y sólo si M acepta a $h(w)$. Por lo tanto,

$$w \in L(M') \iff h(w) \in L(M) = B \iff w \in h^{-1}(B)$$

Esto quiere decir que $L(M') = h^{-1}(B)$. □

Con la ayuda del Teorema 3.5.1 y de las demás propiedades de clausura también podemos concluir que ciertos lenguajes no son regulares.

Ejemplo El lenguaje $L = \{0^n 10^n : n \geq 1\}$ no es regular ya que si lo fuera, también lo sería $h^{-1}(L)$ donde h es el homomorfismo $h(0) = h(1) = 0, h(2) = 1$. Pero

$$h^{-1}(L) = \left\{ \{0, 1\}^n 2 \{0, 1\}^n : n \geq 1 \right\}.$$

Entonces $h^{-1}(L) \cap 0^*21^*$ sería regular; este último lenguaje es igual a $\{0^n21^n : n \geq 1\}$. Finalmente, por medio del homomorfismo $g(0) = 0$, $g(1) = 1$, $g(2) = \lambda$ se concluiría que $g(\{0^n21^n : n \geq 1\}) = \{0^n1^n : n \geq 1\}$ es regular, lo cual sabemos que no es cierto.

Ejercicios de la sección 3.5

Por medio de un razonamiento similar al del ejemplo de esta sección, demostrar que los siguientes lenguajes sobre $\Sigma = \{0, 1\}$ no son regulares:

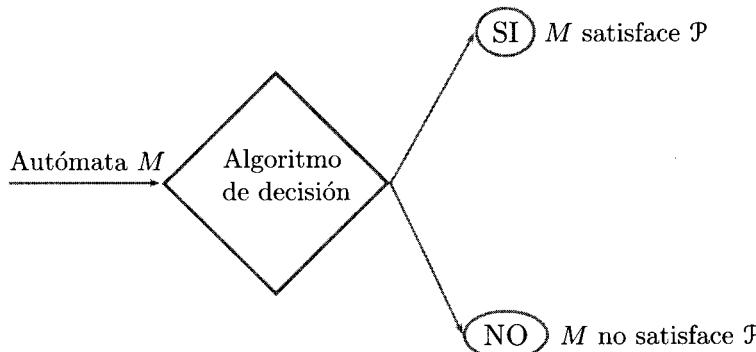
- ① $L = \{ww : w \in \Sigma^*\}$. Ayuda: intersectar primero L con 0^*10^*1 .
- ② $L = \{ww^R : w \in \Sigma^*\}$. Ayuda: intersectar primero L con 0^*110^* .

3.6. Algoritmos de decisión

La noción de algoritmo que consideraremos en esta sección es la corriente:

Un **algoritmo** es un procedimiento sistemático, claro y preciso, que termina en un número finito de pasos para cualquier entrada dada.

Dada una propiedad \mathcal{P} , referente a autómatas sobre un alfabeto Σ , un **problema de decisión para \mathcal{P}** consiste en buscar un algoritmo, aplicable a un autómata arbitrario M , que responda SI o NO a la pregunta: ¿satisface M la propiedad \mathcal{P} ? El siguiente diagrama ilustra la situación.



Si existe un algoritmo de decisión, se dice que el problema \mathcal{P} es **decidible** o **resoluble**; en caso contrario, el problema \mathcal{P} es **indecidible** o **irresoluble**.

Es importante tener presente que, para que un problema \mathcal{P} sea decidable, no basta responder SI o NO a la pregunta “¿satisface M la propiedad $\mathcal{P}?$ ” para uno o varios autómatas aislados M ; es necesario exhibir un algoritmo aplicable a *todos* los autómatas. Es posible que en algunos casos concretos se pueda decidir afirmativa o negativamente sobre la satisfabilidad de una propiedad \mathcal{P} y, sin embargo, el problema \mathcal{P} sea indecidible.

El primer problema de decisión que consideraremos es el problema de decidir si un autómata M acepta o no *alguna* cadena; es decir, decidir si $L(M) = \emptyset$ ó $L(M) \neq \emptyset$. Este problema, llamado *problema de la vacuidad*, difiere del siguiente problema

$$\mathcal{P}: \begin{cases} \text{Dado un autómata (AFD o AFN o AFN-}\lambda\text{) } M \text{ y} \\ \text{una cadena } w \in \Sigma^*, \text{ ¿acepta } M \text{ la cadena } w? \end{cases}$$

En el problema \mathcal{P} anterior las entradas son autómatas M y cadenas $w \in \Sigma^*$; para resolverlo es suficiente el siguiente algoritmo: convertir M en un AFD M' (utilizando los algoritmos ya conocidos para tal efecto) y luego procesar w con M' . Sólo habrá dos salidas: M' acepta a w o M' no acepta a w . El problema de la vacuidad *no* se puede resolver usando esta misma idea porque no podemos examinar *todas* las entradas $w \in \Sigma^*$ (hay infinitas cadenas w). Se requiere entonces una idea diferente, como la presentada en el siguiente teorema.

3.6.1 Teorema. *Sea Σ un alfabeto dado. Existe un algoritmo para el siguiente problema de decisión referente a autómatas sobre Σ :*

Dado un autómata (AFD o AFN o AFN- λ) M , ¿Es $L(M) \neq \emptyset$? (es decir, ¿acepta M alguna cadena?).

Demostración. Podemos diseñar un algoritmo sencillo para encontrar los estados que son accesibles (o alcanzables) desde el estado inicial de M , es decir, los estados para los cuales existen trayectorias desde el estado inicial. Si algún estado de aceptación es alcanzable, se tendrá $L(M) \neq \emptyset$; en caso contrario $L(M) = \emptyset$. En el recuadro de la parte superior de la página siguiente aparece el algoritmo para encontrar el conjunto **ALC** de estados alcanzables.

Claramente, el autómata M acepta alguna cadena (o sea, $L(M) \neq \emptyset$) si y solo si **ALC** contiene algún estado de aceptación. \square

3.6.2 Corolario. *Sea Σ un alfabeto dado. Existen algoritmos para los siguientes problemas de decisión referentes a autómatas sobre Σ :*

INICIALIZAR:

ALC := $\{q_0\}$, donde q_0 es el estado inicial de M .

REPETIR:

Para cada $q \in \text{ALC}$ buscar los arcos que salen de q y añadir a **ALC** los estados p para los cuales haya un arco de q a p con cualquier etiqueta (puede ser λ).

HASTA:

No se añaden nuevos estados a **ALC**.

- (1) *Dados dos autómatas M_1 y M_2 (AFD o AFN o AFN- λ), $\delta L(M_1) \subseteq L(M_2)$?*
- (2) *Dados dos autómatas M_1 y M_2 (AFD o AFN o AFN- λ), $\delta L(M_1) = L(M_2)$?*

Demostración.

- (1) Sea $L_1 = L(M_1)$ y $L_2 = L(M_2)$. Se tiene

$$L_1 \subseteq L_2 \iff L_1 - L_2 = \emptyset.$$

Algoritmo: construir un AFD M' que acepte el lenguaje $L_1 - L_2$ (esto se puede hacer en razón de la parte (7) del Teorema 3.3.1). Utilizar luego el procedimiento del Teorema 3.6.1 para decidir si $L_1 - L_2$ es o no vacío.

- (2) $L(M_1) = L(M_2) \iff L(M_1) \subseteq L(M_2)$ y $L(M_2) \subseteq L(M_1)$. Por lo tanto, basta aplicar dos veces el algoritmo de la parte (1). También se puede encontrar un algoritmo de decisión teniendo en cuenta que $L_1 = L_2 \iff L_1 \triangleleft L_2 = \emptyset$ junto con la parte (8) del Teorema 3.3.1. \square

El argumento utilizado en la demostración del lema de bombeo sirve para establecer un criterio que permite decidir si el lenguaje aceptado por un autómata es o no infinito. El criterio aparece en el siguiente teorema.

3.6.3 Teorema. *Sea M un AFD con n estados y sea $L = L(M)$. L es infinito si y solo si M acepta una cadena w tal que $n \leq |w| < 2n$.*

Demostración. Si $w \in M$ y $n \leq |w| < 2n$, entonces por la demostración del lema de bombeo, w se puede descomponer como $w = uvx$, donde $|uv| \leq n$, $v \neq \lambda$. M acepta infinitas cadenas: $uv^i x$ para todo $i \geq 0$.

Recíprocamente, si L es infinito, existe $w \in L$ con $|w| \geq n$. Por la demostración del lema de bombeo, $w = uvx$ donde $|uv| \leq n$, $v \neq \lambda$. Si $|w| < 2n$ la demostración termina. Si $|w| \geq 2n$, puesto que $|v| \leq |uv| \leq n$, se tendrá $|ux| \geq n$ y $ux \in L$. De nuevo, si $|ux| < 2n$, la demostración termina; en caso contrario, se prosigue de esta forma hasta encontrar una cadena en L cuya longitud ℓ satisfaga $n \leq \ell < 2n$. \square

3.6.4 Corolario. *Sea Σ un alfabeto dado. Existe un algoritmo para el siguiente problema de decisión referente a autómatas sobre Σ :*

Dado un autómata M (AFD o AFN o AFN- λ), ¿Es $L(M)$ infinito?

Demostración. Algoritmo: construir un AFD M' que acepte el lenguaje $L(M)$ y chequear la aceptación o rechazo de todas las cadenas $w \in \Sigma^*$ tales que $n \leq |w| < 2n$, donde $n = \#$ de estados de M' . \square

Hemos trabajado con problemas de decisión referentes a autómatas, pero también podemos considerar problemas sobre lenguajes regulares. Dada una propiedad \mathcal{P} , referente a lenguajes regulares sobre un alfabeto Σ , un problema de decisión para \mathcal{P} consiste en buscar un algoritmo, aplicable a todo lenguaje regular L (es decir, a toda expresión regular R), que responda SI o NO a la pregunta: ¿satisface L la propiedad \mathcal{P} ? Puesto que conocemos algoritmos de conversión entre la representación por expresiones regulares y la representación por autómatas, los problemas decidibles sobre autómatas corresponden a problemas decidibles sobre lenguajes regulares y viceversa. Así por ejemplo, en razón del Corolario 3.6.4, el siguiente problema es decidible: “Dada una expresión regular R , ¿es $L(R)$ infinito?”

Ejercicios de la sección 3.6

- ① Sea Σ un alfabeto dado. Encontrar algoritmos para los siguientes problemas de decisión referentes a autómatas sobre Σ :
 - Dado un autómata M (AFD o AFN o AFN- λ), ¿es $L(M) = \Sigma^*$?
 - Dado un autómata M (AFD o AFN o AFN- λ), ¿acepta M todas las cadenas de longitud impar?
 - Dado un autómata M (AFD o AFN o AFN- λ) y una cadena $w \in \Sigma^*$, ¿acepta M la cadena ww^R ?
 - Dados dos autómatas M_1 y M_2 (AFD o AFN o AFN- λ), ¿existe alguna cadena que no sea aceptada por ninguno de los autómatas M_1 y M_2 ?

-
- (v) Dado un autómata M (AFD o AFN o AFN- λ), ¿es $L(M)$ cofinito? (Un conjunto es **cofinito** si su complemento es finito).
 - (vi) Dado un autómata M (AFD o AFN o AFN- λ), ¿acepta M alguna cadena de longitud 1250? Ayuda: hay un número finito de cadenas con longitud 1250.
 - (vii) Dado un autómata M (AFD o AFN o AFN- λ), ¿acepta M alguna cadena de longitud mayor que 1250? Ayuda: habría un número infinito de cadenas por examinar; usar el Teorema 3.6.3.
 - (viii) Dado un autómata M (AFD o AFN o AFN- λ), ¿acepta M por lo menos 1250 cadenas? Ayuda: usar la misma idea del problema (vii).
- !② Encontrar algoritmos de decisión, que no utilicen autómatas, para resolver los siguientes problemas:
- (i) Dada una expresión regular R , ¿es $L(R) \neq \emptyset$? Ayuda: puesto que las expresiones regulares se definen recursivamente, el algoritmo requiere descomponer la expresión R y utilizar criterios de vacuidad para la estrella de Kleene, la unión y la concatenación de lenguajes.
 - (ii) Dada una expresión regular R , ¿contiene $L(R)$ por lo menos 150 cadenas?
- ☞ Al considerar problemas de decisión lo importante es la *existencia* o no de algoritmos de decisión, no tanto la *eficiencia* de los mismos.
- ☞ En el Capítulo 7 se mostrará que existen problemas indecidibles, es decir, problemas para los cuales no hay algoritmos de decisión.

Capítulo

4

Lenguajes y gramáticas independientes del contexto

Como se ha visto, los autómatas son dispositivos que procesan cadenas de entrada. En capítulos posteriores consideraremos modelos de autómatas con mayor poder computacional que el de los modelos AFD, AFN y AFN- λ . En el presente capítulo estudiaremos una noción completamente diferente, aunque relacionada, la de gramática generativa, que es un mecanismo para generar cadenas a partir de un símbolo inicial.

- ☞ Los autómatas *procesan* cadenas
- ☞ Las gramáticas *generan* cadenas

4.1. Gramáticas generativas

Las gramáticas generativas fueron introducidas por Noam Chomsky en 1956 como un modelo para la descripción de los lenguajes naturales (español, inglés, etc). Chomsky clasificó las gramáticas en cuatro tipos: 0, 1, 2 y 3. Las gramáticas de tipo 2, también llamadas gramáticas independientes del contexto, se comenzaron a usar en la década de los sesenta para presentar la sintaxis de lenguajes de programación y para el diseño de analizadores sintácticos en compiladores. En este curso únicamente tendremos oportunidad de estudiar gramáticas de tipos 2 y 3. El estudiante puede pasar directamente a la sección 4.2; el resto de la presente sección es material (opcional) de referencia.

Una **gramática generativa** es una cuádrupla, $G = (V, \Sigma, S, P)$ formada

por dos alfabetos disyuntos V (alfabeto de *variables* o *no-terminales*) y Σ (alfabeto de *terminales*), una variable especial $S \in V$ (llamada *símbolo inicial*) y un conjunto finito $P \subseteq (V \cup \Sigma)^* \times (V \cup \Sigma)^*$ de *producciones* o *reglas de re-escritura*. Una producción $(v, w) \in P$ se denota por $v \rightarrow w$ y se lee “ v produce w ”; v se denomina la *cabeza* y w el *cuerpo* de la producción. Se exige que la cabeza de la producción tenga por lo menos una variable.

El significado de la producción $v \rightarrow w$ es: la cadena v se puede reemplazar (sobre-escribir) por la cadena w . Comenzando con el símbolo inicial S y aplicando las producciones de la gramática, en uno o más pasos, se obtienen cadenas de terminales y/o no-terminales. Aquellas cadenas que sólo tengan terminales conforman lo que se denomina el *lenguaje generado por G* .

Las gramáticas se clasifican de acuerdo con el tipo de sus producciones:

Gramáticas de tipo 0. No tienen restricciones. También se llaman *gramáticas no-restringidas* o *gramáticas con estructura de frase* en razón de su origen lingüístico.

Gramáticas de tipo 1. Las producciones son de la forma $u_1 A u_2 \rightarrow v_1 w v_2$, donde A es una variable y $w \neq \lambda$. También se llaman *gramáticas sensibles al contexto* o *gramáticas contextuales*.

Gramáticas de tipo 2. Las producciones son de la forma $A \rightarrow w$ donde A es una variable. También se llaman *gramáticas independientes del contexto* o *gramáticas no-contextuales*.

Gramáticas de tipo 3. Las producciones son de la forma $A \rightarrow a$ o de la forma $A \rightarrow aB$, donde A y B son variables y a es un símbolo terminal. También se llaman *gramáticas regulares*.

Se dice que un lenguaje es de tipo i si es generado por una gramática de tipo i . Esta clasificación de lenguajes se conoce como la *jerarquía de Chomsky*.

4.2. Gramáticas independientes del contexto

Una **gramática independiente del contexto** (GIC), también llamada **gramática no-contextual** o **gramática de tipo 2**, es una cuádrupla, $G = (V, \Sigma, S, P)$ formada por:

1. Un alfabeto V cuyos elementos se llaman **variables** o **símbolos no-terminales**.

2. Un alfabeto Σ cuyos elementos se llaman **símbolos terminales**. Se exige que los alfabetos Σ y V sean disyuntos.
3. Una variable especial $S \in V$, llamada **símbolo inicial** de la gramática.
4. Un conjunto finito $P \subseteq V \times (V \cup \Sigma)^*$ de **producciones** o **reglas de re-escritura**. Una producción $(A, w) \in P$ de G se denota por $A \rightarrow w$ y se lee “ A produce w ”; su significado es: la variable A se puede reemplazar (sobre-escribir) por la cadena w . En la producción $A \rightarrow w$, A se denomina la **cabeza** y w el **cuerpo** de la producción.

Notación y definiciones. Las variables se denotan con letras mayúsculas A, B, C, \dots . Los elementos de Σ o símbolos terminales se denotan con letras minúsculas a, b, c, \dots . Si $u, v \in (V \cup \Sigma)^*$ y $A \rightarrow w$ es una producción, se dice que uvv se **deriva directamente** de uAv , lo cual se denota por

$$uAv \implies uvv.$$

Si se quiere hacer referencia a la gramática G , se escribe

$$uAv \xrightarrow{G} uvv \quad \text{ó} \quad uAv \implies_G uvv.$$

Si u_1, u_2, \dots, u_n son cadenas en $(V \cup \Sigma)^*$ y hay una sucesión de derivaciones directas

$$u_1 \xrightarrow{G} u_2, \quad u_2 \xrightarrow{G} u_3, \dots, u_{n-1} \xrightarrow{G} u_n$$

se dice que u_n se deriva de u_1 y se escribe $u_1 \xrightarrow{*} u_n$. La anterior sucesión de derivaciones directas se representa como

$$u_1 \implies u_2 \implies u_3 \implies \cdots \implies u_{n-1} \implies u_n$$

y se dice que es una **derivación** o una **generación** de u_n a partir de u_1 . Para toda cadena w se asume que $w \xrightarrow{*} w$; por lo tanto, $u \xrightarrow{*} v$ significa que v se obtiene de u utilizando cero, una o más producciones de la gramática. Análogamente, $u \xrightarrow{+} v$ significa que v se obtiene de u utilizando una o más producciones.

El **lenguaje generado por una gramática** G se denota por $L(G)$ y se define como

$$L(G) := \{w \in \Sigma^* : S \xrightarrow{+} w\}.$$

Un lenguaje L sobre un alfabeto Σ se dice que es un **lenguaje independiente del contexto** (LIC) si existe una GIC G tal que $L(G) = L$. Dos GIC G_1 y G_2 son **equivalentes** si $L(G_1) = L(G_2)$.

La denominación “independiente del contexto” proviene del hecho de cada producción o regla de re-escritura $A \rightarrow w$ se aplica a la variable A independientemente de los caracteres que la rodean, es decir, independientemente del contexto en el que aparece A .

Ejemplo Sea G una gramática (V, Σ, S, P) dada por:

$$\begin{aligned} V &= \{S\} \\ \Sigma &= \{a\} \\ P &= \{S \rightarrow aS, S \rightarrow \lambda\} \end{aligned}$$

Se tiene $S \implies \lambda$ y

$$S \implies aS \xrightarrow{*} a \cdots aS \implies a \cdots a.$$

Por consiguiente, $L(G) = a^*$.

De manera más simple, podemos presentar una gramática GIC listando sus producciones y separando con el símbolo $|$ las producciones de una misma variable. Se supone siempre que las letras mayúsculas representan variables y las letras minúsculas representan símbolos terminales. Así la gramática del ejemplo anterior se presenta simplemente como:

$$S \rightarrow aS | \lambda.$$

Ejemplo La gramática $G = (V, \Sigma, S, P)$ dada por:

$$\begin{aligned} V &= \{S, A\} \\ \Sigma &= \{a, b\} \\ P &= \{S \rightarrow aS, S \rightarrow bA, S \rightarrow \lambda, A \rightarrow bA, A \rightarrow b, A \rightarrow \lambda\} \end{aligned}$$

se puede presentar como

$$\begin{cases} S \rightarrow aS | bA | \lambda \\ A \rightarrow bA | b | \lambda \end{cases}$$

Se tiene $S \implies \lambda$. Todas las demás derivaciones en G comienzan ya sea con la producción $S \rightarrow aS$ o con $S \rightarrow bA$. Por lo tanto, tenemos

$$S \implies aS \xrightarrow{*} a \cdots aS \implies a \cdots a.$$

$$S \implies bA \xrightarrow{*} b \cdots bA \implies b \cdots b.$$

$$S \implies aS \xrightarrow{*} a \cdots aS \implies a \cdots abA \xrightarrow{*} a \cdots ab \cdots bA \implies a \cdots ab \cdots b.$$

Por consiguiente $L(G) = a^*b^*$.

Las siguientes gramáticas también generan el lenguaje a^*b^* y son, por lo tanto, equivalentes a G :

$$\begin{cases} S \rightarrow AB \\ A \rightarrow aA \mid \lambda \\ B \rightarrow bB \mid \lambda \end{cases} \quad \begin{cases} S \rightarrow AB \mid \lambda \\ A \rightarrow aA \mid a \mid \lambda \\ B \rightarrow bB \mid b \mid \lambda \end{cases} \quad \begin{cases} S \rightarrow aS \mid A \\ A \rightarrow bA \mid \lambda \end{cases}$$

Ejemplo La gramática

$$\begin{cases} S \rightarrow aS \mid aA \\ A \rightarrow bA \mid b \end{cases}$$

genera el lenguaje a^+b^+ . Otra gramática equivalente es:

$$\begin{cases} S \rightarrow AB \\ A \rightarrow aA \mid a \\ B \rightarrow bB \mid b \end{cases}$$

Ejemplo La gramática

$$\begin{cases} S \rightarrow 1A \mid 0 \\ A \rightarrow 0A \mid 1A \mid \lambda \end{cases}$$

genera el lenguaje de los números naturales en numeración binaria. Nótese que la única cadena que comienza con 0, generable con esta gramática, es la cadena 0.

Ejemplo Encontrar una GIC que genere el lenguaje $0^*10^*10^*$ sobre $\Sigma = \{0, 1\}$, es decir, el lenguaje de todas las cadenas con exactamente dos unos.

Solución.

$$G : \quad \begin{cases} S \rightarrow A1A1A \\ A \rightarrow 0A \mid \lambda \end{cases}$$

Una gramática equivalente es

$$\begin{cases} S \rightarrow 0S \mid 1A \\ A \rightarrow 0A \mid 1B \\ B \rightarrow 0B \mid \lambda \end{cases}$$

Ejemplo Encontrar una GIC que genere el lenguaje $L = \{a^i b^i : i \geq 0\}$ sobre $\Sigma = \{a, b\}$, el cual no es un lenguaje regular.

Solución.

$$S \rightarrow aSb \mid \lambda.$$

Ejemplo Encontrar una GIC que genere el lenguaje de todos los palíndromos sobre $\Sigma = \{a, b\}$, el cual no es lenguaje regular.

Solución.

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \lambda.$$

Ejemplo Encontrar una GIC que genere el lenguaje de todas las cadenas sobre $\Sigma = \{a, b\}$ que tienen un número par de símbolos.

Solución.

$$S \rightarrow aSa \mid bSb \mid aSb \mid bSa \mid \lambda$$

Dos gramáticas equivalentes son:

$$\left\{ S \rightarrow aaS \mid bbS \mid abS \mid baS \mid \lambda \right. \quad \left. \begin{cases} S \rightarrow AAS \mid \lambda \\ A \rightarrow a \mid b \end{cases} \right.$$

Ejemplo Encontrar una GIC que genere el lenguaje $(ab \cup ba)^*$ sobre $\Sigma = \{a, b\}$.

Solución.

$$S \rightarrow abS \mid baS \mid \lambda.$$

Ejemplo Demostrar que la gramática G dada por:

$$S \rightarrow (S)S \mid \lambda$$

genera el lenguaje de todas las cadenas de paréntesis anidados y equilibrados; es decir, cadenas como $(()), ((())(), ((())((()))))$.

Solución. Es fácil ver que G genera cadenas de paréntesis anidados y equilibrados ya que cada aplicación de la producción $S \rightarrow (S)S$ da lugar a un par de paréntesis equilibrados.

Para establecer la dirección recíproca demostraremos por inducción sobre n la siguiente afirmación: “Toda cadena con $2n$ paréntesis anidados y equilibrados se puede generar en G ”.

$$n = 1: \text{ } () \text{ se genera con } S \xrightarrow{} (S)S \xrightarrow{2} () .$$

$$n = 2: \text{ } ()() \text{ se genera con } S \xrightarrow{} (S)S \xrightarrow{} (S)(S)S \xrightarrow{3} ()().$$

$$()() \text{ se genera con } S \xrightarrow{} (S)S \xrightarrow{} ((S)S)S \xrightarrow{3} ()().$$

Paso inductivo: una cadena con $2n$ paréntesis anidados y equilibrados es de la forma (u) ó $(u)v$ donde u y v tienen estrictamente menos de $2n$ paréntesis anidados y equilibrados. Por hipótesis de inducción $S \xrightarrow{+} u$ y $S \xrightarrow{+} v$. Por lo tanto,

$$S \xrightarrow{} (S)S \xrightarrow{+} (u)S \xrightarrow{} (u).$$

$$S \xrightarrow{} (S)S \xrightarrow{+} (u)S \xrightarrow{+} (u)v.$$

Ejercicios de la sección 4.2

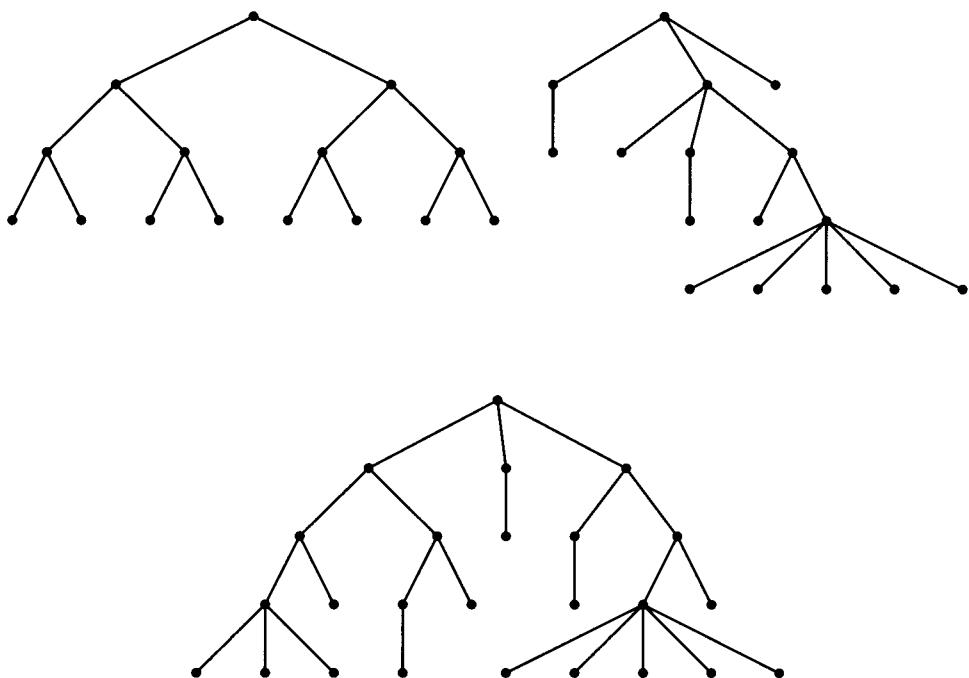
- ① Encontrar GIC que generen los siguientes lenguajes sobre $\Sigma = \{a, b\}$:
 - (i) El lenguaje de las cadenas que tienen un número par de *b*s.
 - (ii) El lenguaje de las cadenas que comienzan con *b* y terminan con *ba*.
 - (iii) $a^*b \cup a$.
 - (iv) $a^*b \cup b^*a$.
 - (v) $(ab^* \cup b^*a)^*$.
 - (vi) $\{a^i b^{2i} : i \geq 0\}$.
 - (vii) $\{ab^i ab^i : i \geq 1\}$.
- ② Encontrar GIC que generen los siguientes lenguajes sobre $\Sigma = \{a, b, c, d\}$:
 - (i) $\{a^i b^j c^j d^i : i, j \geq 1\}$.
 - (ii) $\{a^i b^i c^j d^j : i, j \geq 1\}$.
 - (iii) $\{a^i b^j c^j d^i : i, j \geq 1\} \cup \{a^i b^i c^j d^j : i, j \geq 1\}$.
 - (iv) $\{a^i b^j c^{i+j} : i \geq 0, j \geq 1\}$.
- ③ Demostrar que la gramática $S \rightarrow SS \mid (S) \mid \lambda$ también genera el lenguaje de todas las cadenas de paréntesis anidados y equilibrados.
- ④ Encontrar una gramática que genere el lenguaje de todas las cadenas de paréntesis circulares y/o angulares, anidados y equilibrados. Es decir cadenas como $((\))$, $([\]) [(\)]$, $[(\)([[(\)])])$, etc. Cadenas como $([])$ ó $[([])]$ tienen paréntesis equilibrados pero no anidados y, por lo tanto, no pertenecen a este lenguaje.
- !⑤ Sea $\Sigma = \{0, 1\}$. Encontrar una GIC que genere el lenguaje de las cadenas que tienen igual número de ceros que de unos.

4.3. Árbol de una derivación

Un **árbol con raíz** es un tipo muy particular de grafo no-dirigido; está formado por un conjunto de vértices o nodos conectados entre sí por arcos o aristas, con la siguiente propiedad: existe un nodo especial, llamado la **raíz** del árbol, tal que hay una única trayectoria entre cualquier nodo y la raíz. De esta manera, la raíz se ramifica en nodos, llamados **descendientes inmediatos**, cada uno de los cuales puede tener, a su vez, descendientes inmediatos, y así sucesivamente.

La propiedad que caracteriza a un árbol garantiza que un nodo puede tener 0, 1, o más descendientes inmediatos pero un único antecesor inmediato. El único nodo que no tiene antecesores es la raíz. Los nodos que tienen descendientes, excepto la raíz, se denominan **nodos interiores**. Los nodos que no tienen descendientes se llaman **hojas** del árbol. En la terminología usualmente utilizada, los descendientes de un nodo también se denominan **hijos** y los antecesores **padres o ancestros**. El número de vértices (y por lo tanto, el número de arcos) de un árbol puede ser infinito.

Ejemplo Algunos árboles con raíz:



El **árbol de una derivación** $S \xrightarrow{*} w$, $w \in \Sigma^*$, en una GIC es un árbol con raíz que se forma de la siguiente manera:

1. La raíz está etiquetada con el símbolo inicial S .
2. Cada nodo interior está etiquetado con un no terminal.
3. Cada hoja está etiquetada con terminal o con λ .
4. Si en la derivación se utiliza la producción $A \rightarrow s_1s_2 \dots s_k$, donde $s_i \in (V \cup \Sigma)^*$, el nodo A tiene k descendientes inmediatos: s_1, s_2, \dots, s_k , escritos de izquierda a derecha.

De esta forma, las hojas del árbol de derivación de $S \xrightarrow{*} w$ son precisamente los símbolos de la cadena w , leídos de izquierda a derecha y, posiblemente, algunos λ .

La búsqueda de un árbol de derivación para una cadena $w \in \Sigma^*$ se denomina **análisis sintáctico** de w . Los árboles de derivación también se suelen llamar árboles sintácticos o árboles de análisis sintáctico.

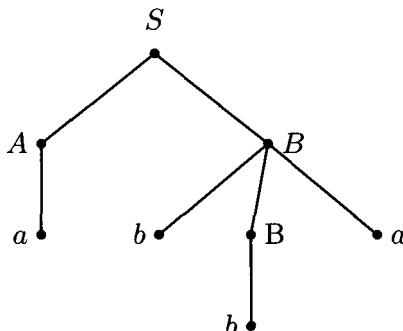
Ejemplo Sea G la gramática:

$$\begin{cases} S \rightarrow AB \mid AaB \\ A \rightarrow aA \mid a \\ B \rightarrow bBa \mid b \end{cases}$$

El árbol de la derivación

$$S \implies AB \implies AbBa \implies abBa \implies abba$$

es



El anterior es también el árbol de la derivación

$$S \implies AB \implies aB \implies abBa \implies abba.$$

Esto muestra que dos derivaciones diferentes pueden tener el mismo árbol de derivación ya que en el árbol aparecen las producciones utilizadas pero no el orden en que han sido aplicadas.

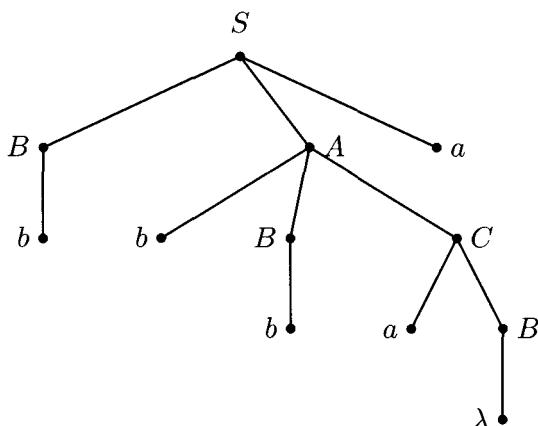
Ejemplo Sea G la gramática:

$$\begin{cases} S \rightarrow BAa \\ A \rightarrow bBC \mid a \\ B \rightarrow bB \mid b \mid \lambda \\ C \rightarrow aB \mid aa \end{cases}$$

El árbol de la derivación

$$S \implies BAa \implies bAa \implies bbBCa \implies bbbCa \implies bbaBa \implies bbaa$$

es



Ejercicios de la sección 4.3

① Sea G siguiente gramática:

$$G : \begin{cases} S \rightarrow aS \mid AaB \\ A \rightarrow aA \mid a \\ B \rightarrow bBbB \mid b \end{cases}$$

Encontrar una derivación de la cadena $aaaabbbb$ y hallar el árbol de tal derivación.

② Sea G la siguiente gramática:

$$G : \begin{cases} S \rightarrow ABC \mid BaC \mid aB \\ A \rightarrow Aa \mid a \\ B \rightarrow BAB \mid bab \\ C \rightarrow cC \mid \lambda \end{cases}$$

Encontrar derivaciones de las cadenas $w_1 = abab$, $w_2 = babacc$, $w_3 = ababababc$ y hallar los árboles de tales derivaciones.

4.4. Gramáticas ambiguas

La noción de ambigüedad se puede presentar en términos de árboles sintácticos o en términos de ciertas derivaciones “estándares”: las llamadas derivaciones a izquierda.

4.4.1 Definición. Una derivación se llama **derivación a izquierda** (o derivación más a la izquierda) si en cada paso se aplica una producción a la variable que está más a la izquierda.

Una derivación cualquiera se puede transformar siempre en una única derivación a izquierda aplicando, en cada paso, producciones a la variable que esté más a la izquierda.

4.4.2 Definición. Una GIC G es **ambigua** si existe una cadena $w \in \Sigma^*$ para la cual hay dos derivaciones a izquierda diferentes. Equivalentemente, una GIC G es **ambigua** si existe una cadena $w \in \Sigma^*$ con dos árboles de derivación diferentes.

Ejemplo Considérese el alfabeto $\Sigma = \{0, 1, +, *, (,)\}$. La siguiente gramática G_{sp} para generar números naturales, sumas y productos, en numeración binaria, es ambigua:

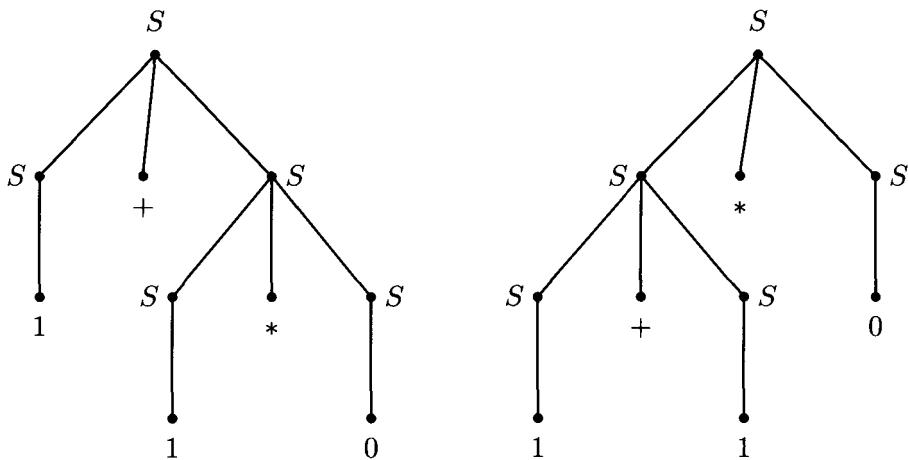
$$S \rightarrow S + S \mid S * S \mid (S) \mid 0S \mid 1S \mid 0 \mid 1$$

La cadena $1 + 1 * 0$ tiene dos derivaciones a izquierda diferentes:

$$S \Rightarrow S + S \Rightarrow 1 + S \Rightarrow 1 + S * S \Rightarrow 1 + 1 * S \Rightarrow 1 + 1 * 0.$$

$$S \Rightarrow S * S \Rightarrow S + S * S \Rightarrow 1 + S * S \Rightarrow 1 + 1 * S \Rightarrow 1 + 1 * 0.$$

Los árboles de derivación correspondientes a las anteriores derivaciones son:



En la gramática G_{sp} la ambigüedad se puede eliminar introduciendo paréntesis:

$$S \rightarrow (S + S) \mid (S * S) \mid (S) \mid 0S \mid 1S \mid 0 \mid 1$$

Aunque la introducción de paréntesis elimina la ambigüedad, las expresiones generadas tienen un excesivo número de paréntesis lo que dificulta el análisis sintáctico (en un compilador, por ejemplo). Lo más corriente en estos casos es utilizar gramáticas ambiguas como G_{sp} siempre y cuando se establezca un orden de precedencia para los operadores. Lo usual es establecer que $*$ tenga una mayor orden de precedencia que $+$, es decir, por convención $*$ actúa antes que $+$. Por ejemplo, la expresión $1 * 1 + 0$ se interpreta como $(1 * 1) + 0$ y la expresión $10 + 11 * 110 + 1$ se interpreta como $10 + (11 * 110) + 1$.

Ejemplo La siguiente gramática es ambigua:

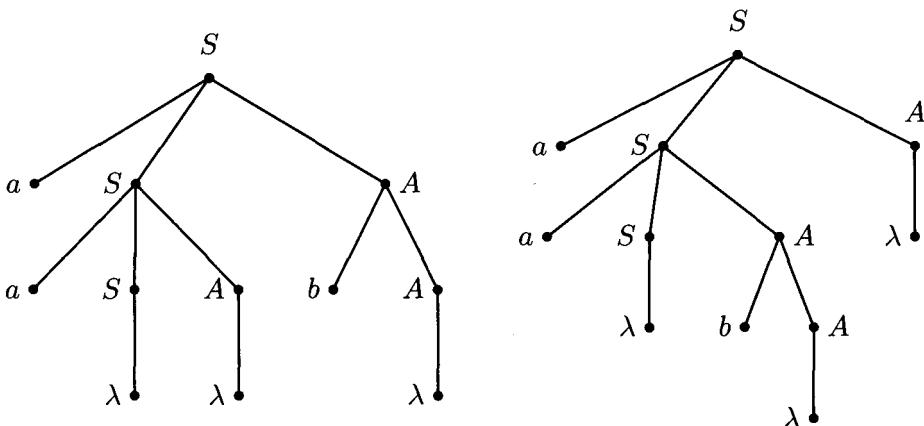
$$G : \begin{cases} S \rightarrow aSA \mid \lambda \\ A \rightarrow bA \mid \lambda \end{cases}$$

G es ambigua porque para la cadena aab hay dos derivaciones a izquierda diferentes.

$$S \Rightarrow aSA \Rightarrow aaSAA \Rightarrow aaAA \Rightarrow aaA \Rightarrow aabA \Rightarrow aab.$$

$$S \Rightarrow aSA \Rightarrow aaSAA \Rightarrow aaAA \Rightarrow aabAA \Rightarrow aabA \Rightarrow aab.$$

Los árboles de derivación para estas dos derivaciones son:



El lenguaje generado por esta gramática es $a^+b^* \cup \lambda$. Se puede construir una gramática no-ambigua que genere el mismo lenguaje:

$$G' : \begin{cases} S \rightarrow AB \mid \lambda \\ A \rightarrow aA \mid a \\ B \rightarrow bB \mid \lambda \end{cases}$$

Para ver que la gramática G' no es ambigua se puede razonar de la siguiente manera: la cadena λ se puede generar de manera única con la derivación $S \Rightarrow \lambda$. Una derivación de una cadena no vacía debe comenzar aplicando la producción $S \rightarrow AB$; la variable A genera cadenas de a s de manera única y B genera cadenas de b s también de manera única. Por consiguiente, toda cadena tiene una única derivación a izquierda.

- ☞ Existen lenguajes independientes del contexto para los cuales toda gramática es ambigua. Tales lenguajes se llaman **inherentemente ambiguos**. Un ejemplo es el lenguaje

$$L = \{a^i b^i c^j d^j : i, j \geq 1\} \cup \{a^i b^j c^i d^j : i, j \geq 1\}.$$

sobre el alfabeto $\{a, b, c, d\}$. En el Ejercicio ② (iii) de la sección 4.2 se pidió mostrar que L es un LIC. La demostración de que L es inherentemente ambiguo es bastante intrincada y puede encontrarse en la referencia [HU1].

- ☞ No existe ningún algoritmo que permita determinar si una GIC dada es o no ambigua. Con la terminología de la sección 3.6, esto quiere decir que el problema de la ambigüedad para GIC es decidible. Éste no es un resultado trivial; para su demostración remitimos al estudiante a la referencia [HMU].

Ejercicios de la sección 4.4

① Mostrar que las siguientes gramáticas son ambiguas:

- (i) $S \rightarrow aSbS \mid bSaS \mid \lambda$.
- (ii) $S \rightarrow abS \mid abScS \mid \lambda$.

② Mostrar que las gramáticas G_1 y G_2 siguientes son ambiguas. En cada caso, encontrar el lenguaje generado por la gramática y construir luego una gramática no-ambigua que genere el mismo lenguaje.

$$G_1 : \begin{cases} S \rightarrow AaSbB \mid \lambda \\ A \rightarrow aA \mid a \\ B \rightarrow bB \mid \lambda \end{cases}$$

$$G_2 : \begin{cases} S \rightarrow ASB \mid AB \\ A \rightarrow aA \mid a \\ B \rightarrow bB \mid \lambda \end{cases}$$

③ Encontrar una GIC no-ambigua que genere el lenguaje $a^*b(a \cup b)^*$.

4.5. Gramáticas para lenguajes de programación

La sintaxis de los lenguajes de programación, o al menos una gran porción de ésta, se presenta usualmente por medio de gramáticas GIC. En tales casos se dice que el lenguaje está en la **forma Backus-Naur** o, simplemente, en la **forma BNF**. Los lenguajes que están en BNF ofrecen ventajas significativas para el diseño de analizadores sintácticos en compiladores.

Ejemplo A continuación se exhibe una gramática para generar los números reales sin signo, similar a la utilizada en muchos lenguajes de programación. Las variables aparecen encerradas entre paréntesis $\langle \rangle$ y $\langle real \rangle$ es la variable inicial de la gramática. El alfabeto de terminales es $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., +, -, E\}$. En el contexto de los lenguajes de programación los terminales se denominan también **componentes léxicos**, **lexemas** o **tokens**.

$$\begin{array}{lcl} \langle real \rangle & \rightarrow & \langle dígitos \rangle \langle decimal \rangle \langle exp \rangle \\ \langle dígitos \rangle & \rightarrow & \langle dígitos \rangle \langle dígitos \rangle \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \langle decimal \rangle & \rightarrow & . \langle dígitos \rangle \mid \lambda \\ \langle exp \rangle & \rightarrow & E \langle dígitos \rangle \mid E+ \langle dígitos \rangle \mid E- \langle dígitos \rangle \mid \lambda \end{array}$$

Esta gramática genera expresiones como 47.236, 321.25E+35, 0.8E9 y 0.8E+9. Las partes decimal y exponencial son “opcionales” debido a las producciones $\langle\text{decimal}\rangle \rightarrow \lambda$ y $\langle\text{exp}\rangle \rightarrow \lambda$, pero no se generan expresiones como .325, E125, 42.5E ni 0.1E+.

Ejemplo Gramática para generar los **identificadores** en lenguajes de programación, es decir, cadenas cuyo primer símbolo es una letra que va seguida de letras y/o dígitos. Las variables aparecen encerradas entre paréntesis $\langle \rangle$ e $\langle\text{identificador}\rangle$ es la variable inicial de la gramática. La variable $\langle\text{lsds}\rangle$ representa “letras o dígitos”. Los terminales en esta gramática son las letras, minúsculas o mayúsculas, y los dígitos.

$\langle\text{identificador}\rangle$	\rightarrow	$\langle\text{letra}\rangle\langle\text{lsds}\rangle$
$\langle\text{lsds}\rangle$	\rightarrow	$\langle\text{letra}\rangle\langle\text{lsds}\rangle \mid \langle\text{dígito}\rangle\langle\text{lsds}\rangle \mid \lambda$
$\langle\text{letra}\rangle$	\rightarrow	a b c … x y z A B C … Y Z
$\langle\text{dígito}\rangle$	\rightarrow	0 1 2 3 4 5 6 7 8 9

Ejercicios de la sección 4.5

- ① Con la gramática del primer ejemplo de esta sección hacer derivaciones y árboles de derivación para las cadenas 235.101E+25 y 0.01E-12.
- ② Encontrar una GIC, con una sola variable, para generar los identificadores, es decir, el lenguaje del segundo ejemplo de esta sección.
- ③ Una gramática similar a la siguiente se usa en muchos lenguajes de programación para generar expresiones aritméticas formadas por sumas y productos de números en binario:

$\langle\text{expresión}\rangle$	\rightarrow	$\langle\text{término}\rangle \mid \langle\text{expresión}\rangle + \langle\text{término}\rangle$
$\langle\text{término}\rangle$	\rightarrow	$\langle\text{factor}\rangle \mid \langle\text{término}\rangle * \langle\text{factor}\rangle$
$\langle\text{factor}\rangle$	\rightarrow	$\langle\text{número}\rangle \mid (\langle\text{expresión}\rangle)$
$\langle\text{número}\rangle$	\rightarrow	1⟨número⟩ 0⟨número⟩ 0 1

El alfabeto de terminales de esta gramática es {0, 1, +, *, (,)}.

- (i) Hacer derivaciones y árboles de derivación para las siguientes cadenas:

10+101*10
 (101+10*10)
 (10+111)*(100+10*101+1111)

- (ii) Demostrar que esta gramática no es ambigua.

4.6. Gramáticas para lenguajes naturales

Para los lenguajes naturales, como el español, se pueden presentar GIC que generen las frases u oraciones permitidas en la comunicación hablada o escrita. Una GIC para generar una porción de las oraciones del idioma español se presenta a continuación; las variables aparecen encerradas entre paréntesis $\langle \rangle$ y $\langle Oración \rangle$ es la variable inicial de la gramática. Los terminales son las palabras propias del idioma.

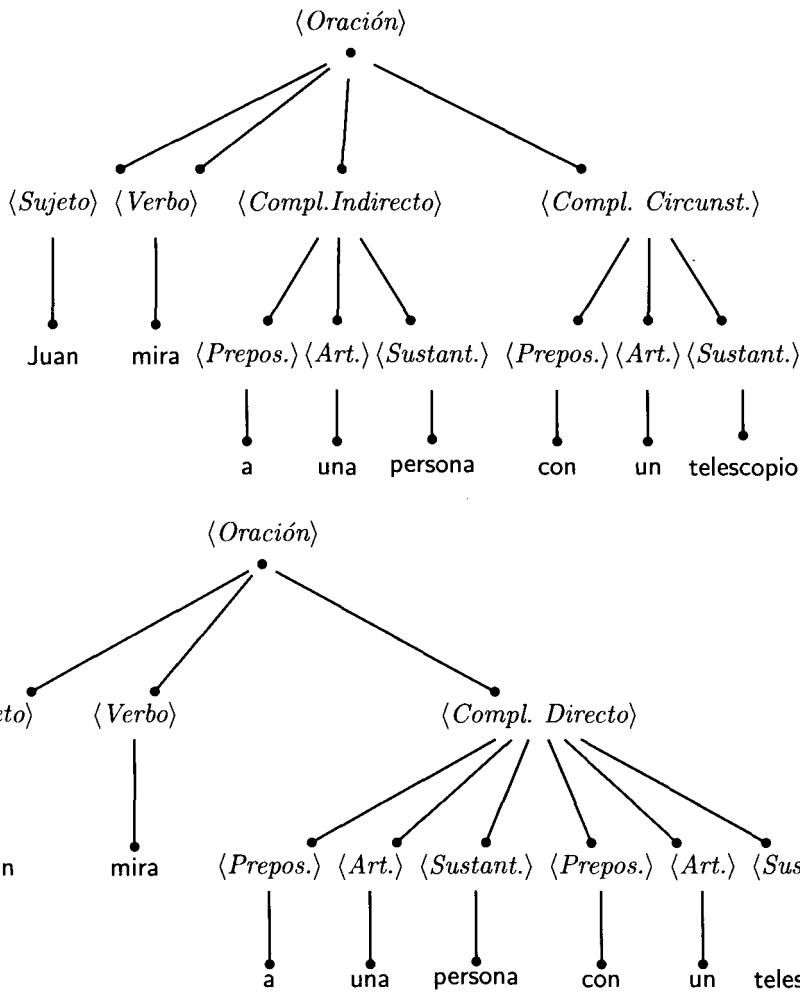
$\langle Oración \rangle$	$\rightarrow \langle Sujeto \rangle \langle Verbo \rangle \langle Compl. Directo \rangle $ $\qquad \langle Sujeto \rangle \langle Verbo \rangle \langle Compl. Directo \rangle \langle Compl. Circunst. \rangle $ $\qquad \langle Sujeto \rangle \langle Verbo \rangle \langle Compl. Indirecto \rangle \langle Compl. Circunst. \rangle$
$\langle Sujeto \rangle$	$\rightarrow \langle Sustant. \rangle Juan Pedro María \dots$
$\langle Compl. Directo \rangle$	$\rightarrow \langle Prepos. \rangle \langle Artículo \rangle \langle Sustant. \rangle $ $\qquad \langle Prepos. \rangle \langle Artículo \rangle \langle Sustant. \rangle \langle Prepos. \rangle \langle Artículo \rangle \langle Sustant. \rangle $ $\qquad \langle Prepos. \rangle \langle Artículo \rangle \langle Sustant. \rangle \langle Prepos. \rangle \langle Sustant. \rangle \langle Adjetivo \rangle$
$\langle Compl. Indirecto \rangle$	$\rightarrow \langle Prepos. \rangle \langle Artículo \rangle \langle Sustant. \rangle $ $\qquad \langle Prepos. \rangle \langle Artículo \rangle \langle Sustant. \rangle \langle Adjetivo \rangle $ $\qquad \langle Prepos. \rangle \langle Sustant. \rangle \langle Prepos. \rangle \langle Sustant. \rangle$
$\langle Compl. Circunst. \rangle$	$\rightarrow \langle Prepos. \rangle \langle Artículo \rangle \langle Sustant. \rangle $ $\qquad \langle Prepos. \rangle \langle Artículo \rangle \langle Sustant. \rangle \langle Adjetivo \rangle \langle Adverbio \rangle $ $\qquad \langle Prepos. \rangle \langle Artículo \rangle \langle Sustant. \rangle \langle Prepos. \rangle \langle Artículo \rangle \langle Sustant. \rangle$
$\langle Sustant. \rangle$	$\rightarrow \text{casa} \text{perro} \text{libro} \text{lápiz} \text{mesa} \lambda \dots$
$\langle Adjetivo \rangle$	$\rightarrow \text{rojo} \text{azul} \text{inteligente} \text{malvado} \text{útil} \lambda \dots$
$\langle Prepos. \rangle$	$\rightarrow \text{a} \text{ante} \text{bajo} \text{con} \text{contra} \text{de} \text{desde} \text{en} \text{entre} \text{hacia} $ $\qquad \text{hasta} \text{para} \text{por} \text{según} \text{sin} \text{so} \text{sobre} \text{tras} \lambda \dots$
$\langle Artículo \rangle$	$\rightarrow \text{el} \text{la} \text{lo} \text{las} \text{los} \text{un} \text{uno} \text{una} \text{unas} \text{unos} \lambda$
$\langle Adverbio \rangle$	$\rightarrow \text{muy} \text{bastante} \text{poco} \text{demasiado} \text{lento} \text{ lentamente} $ $\qquad \text{rápido} \text{ rápidamente} \lambda \dots$
$\langle Verbo \rangle$	$\rightarrow \text{escribir} \text{escribo} \text{escribe} \text{escribes} \text{escriben} \text{escribí} $ $\qquad \text{escribiste} \text{escribieron} \dots$

Los lenguajes naturales son casi siempre ambiguos porque existen muchas reglas de producción, lo que da lugar a múltiples árboles de derivación para ciertas oraciones.

Ejemplo La oración

Juan mira a una persona con un telescopio

es ambigua. La ambigüedad surge porque las producciones permiten dos árboles de derivación:



Ejercicios de la sección 4.6

Usando la gramática exhibida en la presente sección, mostrar que las siguientes oraciones del idioma español son ambiguas. En cada caso hacer los árboles de derivación.

- ① María escucha la conversación tras la puerta.
- ② Pedro ve la televisión en la mesa.
- ③ Manuel observa a la chica con vestido rojo.
- ④ Ana soñó con un gato en pijama.

4.7. Gramáticas regulares

4.7.1 Definición. Una GIC se llama **regular** si sus producciones son de la forma

$$\begin{cases} A \rightarrow aB, & a \in \Sigma, B \in V. \\ A \rightarrow \lambda. \end{cases}$$

Los siguientes teoremas establecen la conexión entre los lenguajes regulares y las gramáticas regulares.

4.7.2 Teorema. *Dado un AFD $M = (Q, \Sigma, q_0, F, \delta)$, existe una GIC regular $G = (V, \Sigma, S, P)$ tal que $L(M) = L(G)$.*

Demostración. Sea $V = Q$ y $S = q_0$. Las producciones de G están dadas por

$$\begin{cases} q \rightarrow ap & \text{si y sólo si } \delta(q, a) = p. \\ q \rightarrow \lambda & \text{si y sólo si } q \in F. \end{cases}$$

Demostraremos primero que para toda $w \in \Sigma^*$, $w \neq \lambda$ y para todo $p, q \in Q$ se tiene

(1) Si $\delta(q, w) = p$ entonces $q \xrightarrow{*} wp$.

La demostración de (1) se hace por inducción sobre w . Si $w = a$ y $\delta(q, a) = p$, entonces $q \rightarrow ap$ es una producción de G y obviamente se concluye $q \xrightarrow{*} ap$. Para el paso inductivo, sea $\delta(q, wa) = p'$. Entonces

$$p' = \delta(q, wa) = \delta(\delta(q, w), a) = \delta(p, a)$$

donde $\delta(q, w) = p$. Por hipótesis de inducción $q \xrightarrow{*} wp$ y como $\delta(p, a) = p'$, entonces $p \xrightarrow{*} ap'$. Por lo tanto,

$$q \xrightarrow{*} wp \xrightarrow{*} wap'$$

que era lo que se quería demostrar.

A continuación demostraremos el recíproco de (1): para toda $w \in \Sigma^*$, $w \neq \lambda$ y para todo $p, q \in Q$ se tiene

(2) Si $q \xrightarrow{*} wp$ entonces $\delta(q, w) = p$.

La demostración de (2) se hace por inducción sobre la longitud de la derivación $q \xrightarrow{*} wp$, es decir, por el número de pasos o derivaciones directas que hay en $q \xrightarrow{*} wp$. Si la derivación tiene longitud 1, necesariamente $q \xrightarrow{*} ap$ lo cual significa que $\delta(q, a) = p$. Para el paso inductivo, supóngase

que $q \xrightarrow{*} wp$ tiene longitud $n + 1$, $w = w'a$ y en el último paso se aplica la producción $p' \rightarrow ap$. Entonces

$$q \xrightarrow{*} w'p' \Rightarrow w'ap = wp.$$

Por hipótesis de inducción, $\delta(q, w') = p'$ y por consiguiente

$$\delta(q, w) = \delta(q, w'a) = \delta(\delta(q, w'), a) = \delta(p', a) = p,$$

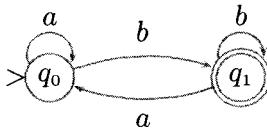
que era lo que se quería demostrar.

Como consecuencia de (1) y (2) se puede ahora demostrar que

(3) Para toda cadena $w \in \Sigma^*$, $\delta(q_0, w) \in F$ si y sólo si $S \xrightarrow{*G} w$, lo cual afirma que $L(M) = L(G)$. En efecto, si $w = \lambda$, $\delta(q_0, w) \in F$ si y sólo si $q_0 \in F$. Por lo tanto, $q_0 \rightarrow \lambda$ es una producción de G . Así que $S \xrightarrow{} \lambda$. Recíprocamente, si $S \xrightarrow{*} \lambda$, necesariamente $S \xrightarrow{} \lambda$, $q_0 \in F$ y $\delta(q_0, \lambda) \in F$.

Sea ahora $w \neq \lambda$. Si $\delta(q_0, w) = p \in F$, por (1) se tiene $q_0 \xrightarrow{*} w$, o sea, $S \xrightarrow{*} w$. Recíprocamente, si $S \xrightarrow{*G} w$, entonces $q_0 \xrightarrow{*G} wp \xrightarrow{} w$ donde $p \rightarrow \lambda$. Utilizando (2), se tiene $\delta(q_0, w) = p \in F$. \square

Ejemplo El siguiente AFD M , presentado en el último ejemplo de la sección 2.3, acepta las cadenas que terminan en b :



M induce la gramática regular

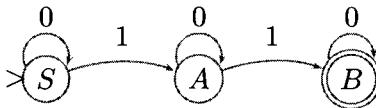
$$G : \begin{cases} q_0 \rightarrow aq_0 \mid bq_1 \\ q_1 \rightarrow bq_1 \mid aq_0 \mid \lambda \end{cases}$$

que cumple $L(M) = L(G)$. Las variables de G son q_0 y q_1 , siendo q_0 la variable inicial.

Ejemplo Para el lenguaje regular $0^*10^*10^*$, sobre $\Sigma = \{0, 1\}$ (el lenguaje de todas las cadenas con exactamente dos unos), vimos en la sección 4.2 una gramática que lo genera:

$$\begin{cases} S \rightarrow A1A1A \\ A \rightarrow 0A \mid \lambda \end{cases}$$

Esta gramática no es regular, pero por medio del AFD



y el Teorema 4.7.2 se puede obtener una GIC regular que genere $0^*10^*10^*$:

$$\begin{cases} S \rightarrow 0S \mid 1A \\ A \rightarrow 0A \mid 1B \\ B \rightarrow 0B \mid \lambda \end{cases}$$

4.7.3 Teorema. *Dada una GIC regular $G = (V, \Sigma, S, P)$, existe un AFN $M = (Q, \Sigma, q_0, F, \Delta)$ tal que $L(M) = L(G)$.*

Demostración. Se construye $M = (Q, \Sigma, q_0, F, \Delta)$ haciendo $Q = V$, $q_0 = S$ y

$$\begin{cases} B \in \Delta(A, a) & \text{para cada producción } A \rightarrow aB. \\ A \in F & \text{si } A \rightarrow \lambda. \end{cases}$$

Usando razonamientos similares a los del Teorema 4.7.2, se puede demostrar que

$$A \xrightarrow{*G} wB \quad \text{si y sólo si} \quad B \in \Delta(A, w), \text{ para todo } w \in \Sigma^*, w \neq \lambda,$$

de donde $L(M) = L(G)$. Los detalles se dejan como ejercicio. \square

4.7.4 Corolario. 1. *Un lenguaje es regular si y solamente si es generado por una gramática regular.*

2. *Todo lenguaje regular es un LIC (pero no viceversa).*

Demostración.

1. Se sigue del Teorema 4.7.2, el Teorema 4.7.3 y del Teorema de Kleene.
2. Se sigue de la parte 1. Por otro lado, $\{a^i b^i : i \geq 0\}$ es LIC pero no es regular. \square

4.7.5 Definición. Una GIC se llama **regular por la derecha** si sus producciones son de la forma

$$\begin{cases} A \rightarrow wB, & w \in \Sigma^*, B \in V, \\ A \rightarrow \lambda & \end{cases}$$

4.7.6 Teorema. *Las gramáticas regulares y las gramáticas regulares por la derecha generan los mismos lenguajes, es decir, los lenguajes regulares. Dicho de otra manera, la definición de gramática regular es equivalente a la definición de gramática regular por la derecha.*

Demostración. Una gramática regular es obviamente regular por la derecha. Recíprocamente, en una gramática regular por la derecha $G = (V, \Sigma, S, P)$, una producción de la forma

$$A \rightarrow a_1 a_2 \cdots a_n B$$

donde los $a_i \in \Sigma$, $n \geq 2$, $B \in V$, se puede simular con producciones de la forma $A \rightarrow aB$ y $A \rightarrow \lambda$. En efecto, se introducen $n - 1$ variables *nuevas* A_1, \dots, A_{n-1} cuyas *índicas* producciones son:

$$\begin{array}{ll} A & \rightarrow a_1 A_1 \\ A_1 & \rightarrow a_2 A_2 \\ & \vdots \\ A_{n-1} & \rightarrow a_n B \end{array}$$

De esta manera se puede construir una gramática regular equivalente a G . □

Ejercicios de la sección 4.7

① Encontrar GIC regulares que generen los siguientes lenguajes:

- (i) ab^*a .
- (ii) $(ab \cup ba)^*$.
- (iii) $a^+b \cup b^+a^*b$.
- (iv) $0^*(10^* \cup 01^*)$.

② Una GIC se llama **regular por la izquierda** si sus producciones son de la forma:

$$\begin{cases} A \rightarrow Bw, & w \in \Sigma^*, B \in V \\ A \rightarrow \lambda \end{cases}$$

Demostrar que las gramáticas regulares y las gramáticas regulares por la izquierda generan los mismos lenguajes.

!③ Completar los detalles de la demostración del Teorema 4.7.3.

4.8. Eliminación de las variables inútiles

En una GIC puede haber dos tipos de variables inútiles: aquéllas que nunca aparecen en el curso de una derivación y aquéllas que no se pueden convertir en cadenas de terminales. A continuación se precisan estos conceptos.

4.8.1. Definiciones.

- (i) Una variable A es **alcanzable** o **accesible** si existen $u, v \in (V \cup \Sigma)^*$ tales que $S \xrightarrow{*} uAv$. La variable inicial S es alcanzable por definición.
- (ii) Una variable A es **terminable** si existe $w \in \Sigma^*$ tal que $A \xrightarrow{*} w$. En particular, si $A \rightarrow \lambda$ es una producción entonces A es terminable.
- (iii) A es una variable **inútil** si no es alcanzable o no es terminable.

Existen algoritmos para detectar todas las variables inútiles de una GIC que permiten construir una gramática G' equivalente a una gramática G dada, de tal manera que G' no tenga variables inútiles.

4.8.2. Algoritmo para encontrar las variables terminables.

El siguiente algoritmo sirve para encontrar todas las variables terminables de una GIC.

$$\mathbf{TERM}_1 := \{A \in V : \text{Existe una producción de la forma } A \rightarrow w, w \in \Sigma^*\}.$$

$$\mathbf{TERM}_{i+1} := \mathbf{TERM}_i \cup \{A \in V : \exists \text{ producción } A \rightarrow w, w \in (\Sigma \cup \mathbf{TERM}_i)^*\}.$$

Obtenemos una sucesión creciente de conjuntos de variables:

$$\mathbf{TERM}_1 \subseteq \mathbf{TERM}_2 \subseteq \mathbf{TERM}_3 \subseteq \dots$$

Como el conjunto de variables es finito, existe k tal que

$$\mathbf{TERM}_k = \mathbf{TERM}_{k+1} = \mathbf{TERM}_{k+2} = \dots$$

El conjunto **TERM** de variables terminables es entonces

$$\mathbf{TERM} := \bigcup_{i \geq 1} \mathbf{TERM}_i$$

El anterior algoritmo se puede presentar de la siguiente forma:

INICIALIZAR:

TERM := $\{A \in V : \exists \text{ producción } A \rightarrow w, w \in \Sigma^*\}$

REPETIR:

TERM := **TERM** $\cup \{A \in V : \exists \text{ producción } A \rightarrow w, w \in (\Sigma \cup \text{TERM})^*\}$

HASTA:

No se añaden nuevas variables a **TERM**

Ejemplo Encontrar las variables terminables de la siguiente gramática.

$$G : \begin{cases} S \rightarrow ACD \mid bBd \mid ab \\ A \rightarrow aB \mid aA \mid C \\ B \rightarrow aDS \mid aB \\ C \rightarrow aCS \mid CB \mid CC \\ D \rightarrow bD \mid ba \\ E \rightarrow AB \mid aDb \end{cases}$$

Solución.

$$\text{TERM}_1 = \{S, D\}.$$

$$\text{TERM}_2 = \{S, D\} \cup \{B, E\} = \{S, D, B, E\}.$$

$$\text{TERM}_3 = \{S, D, B, E\} \cup \{A\} = \{S, D, B, E, A\}.$$

$$\text{TERM}_4 = \{S, D, B, E, A\} \cup \{\} = \{S, D, B, E, A\}.$$

$$\text{TERM} = \{S, A, B, D, E\}.$$

Conjunto de variables no terminables: $\{C\}$.

4.8.3. Algoritmo para encontrar las variables alcanzables.

El siguiente algoritmo sirve para encontrar todas las variables alcanzables de una GIC.

$$\text{ALC}_1 := \{S\}.$$

$$\text{ALC}_{i+1} := \text{ALC}_i \cup \{A \in V : \exists \text{ produc. } B \rightarrow uAv, B \in \text{ALC}_i \text{ } u, v \in (V \cup \Sigma)^*\}.$$

Obtenemos una sucesión creciente de conjuntos de variables:

$$\text{ALC}_1 \subseteq \text{ALC}_2 \subseteq \text{ALC}_3 \subseteq \dots$$

Como el conjunto de variables es finito, existe k tal que

$$\text{ALC}_k = \text{ALC}_{k+1} = \text{ALC}_{k+2} = \dots$$

El conjunto **ALC** de variables alcanzables es entonces

$$\mathbf{ALC} = \bigcup_{i \geq 1} \mathbf{ALC}_i$$

El anterior algoritmo se puede presentar de la siguiente forma:

INICIALIZAR:

$$\mathbf{ALC} := \{S\}$$

REPETIR:

$$\mathbf{ALC} := \mathbf{ALC} \cup \{A \in V : \exists \text{ producción } B \rightarrow uAv, B \in \mathbf{ALC} \text{ y} \\ u, v \in (V \cup \Sigma)^*\}$$

HASTA:

No se añaden nuevas variables a **ALC**

Ejemplo Encontrar las variables alcanzables de la siguiente gramática.

$$G : \begin{cases} S \rightarrow aS \mid AaB \mid ACS \\ A \rightarrow aS \mid AaB \mid AC \\ B \rightarrow bB \mid DB \mid BB \\ C \rightarrow aDa \mid ABD \mid ab \\ D \rightarrow aD \mid DD \mid ab \\ E \rightarrow FF \mid aa \\ F \rightarrow aE \mid EF \end{cases}$$

Solución.

$$\mathbf{ALC}_1 = \{S\}.$$

$$\mathbf{ALC}_2 = \{S\} \cup \{A, B, C\} = \{S, A, B, C\}.$$

$$\mathbf{ALC}_3 = \{S, A, B, C\} \cup \{D\} = \{S, A, B, C, D\}.$$

$$\mathbf{ALC}_4 = \{S, A, B, C, D\} \cup \{ \} = \{S, A, B, C, D\}$$

$$\mathbf{ALC} = \{S, A, B, C, D\}.$$

Conjunto de variables no alcanzables: $\{E, F\}$.

Dada una GIC G , los dos algoritmos anteriores (algoritmo 4.8.2 y algoritmo 4.8.3) se pueden llevar a cabo en dos órdenes diferentes:

$$(I) \quad G \xrightarrow{\text{Algoritmo}} \text{Eliminar variables} \quad \text{no-terminables} \quad G_1 \xrightarrow{\text{Algoritmo}} \text{Eliminar variables} \quad \text{no-alcanzables} \quad G_2$$

$$(II) \quad G \xrightarrow{\text{Algoritmo}} \text{ Eliminar variables no-alcanzables } G_3 \xrightarrow{\text{Algoritmo}} \text{ Eliminar variables no-terminables } G_4$$

Esto da lugar a las siguientes preguntas:

- (1) ¿Es $G_2 = G_4$?
- (2) ¿ G_2 tiene variables inútiles?
- (3) ¿ G_4 tiene variables inútiles?

El siguiente ejemplo muestra que la respuesta a la pregunta (1) es *no* y que al realizar los algoritmos en el orden (II) pueden permanecer en G_4 algunas variables inútiles.

Ejemplo Considérese la siguiente gramática G .

$$G : \begin{cases} S \rightarrow a \mid AB \\ A \rightarrow aA \mid \lambda \end{cases}$$

Aplicación de los algoritmos en el orden (I):

Variables terminables de G : **TERM**= $\{S, A\}$.

$$G_1 : \begin{cases} S \rightarrow a \\ A \rightarrow aA \mid \lambda \end{cases}$$

Variables alcanzables de G_1 : **ALC**= $\{S\}$.

$$G_2 : \{S \rightarrow a\}$$

Se tiene que $L(G_2) = \{a\}$.

Aplicación de los algoritmos en el orden (II):

Variables alcanzables de G : **ALC**= $\{S, A, B\}$.

$$G_3 : \begin{cases} S \rightarrow a \mid AB \\ A \rightarrow aA \mid \lambda \end{cases}$$

Variables terminables de G_3 : **TERM**= $\{S, A\}$.

$$G_4 : \begin{cases} S \rightarrow a \\ A \rightarrow aA \mid \lambda \end{cases}$$

Se observa que en G_4 , A no es alcanzable. Además, $G_2 \neq G_4$.

No obstante, si los algoritmos se llevan a cabo en el orden (I) la gramática G_2 ya no tiene variables inútiles. Nos podemos convencer de eso observando que las variables alcanzables obtenidas al finalizar el procedimiento siguen siendo terminables. Más precisamente, si una variable A permanece al finalizar el procedimiento completo, será es alcanzable, y si la derivación $A \xrightarrow{*} w$, con $w \in \Sigma^*$, se podía hacer antes de eliminar las variables no alcanzables, también se podrá realizar en la gramática final ya que todas las variables que aparezcan en esa derivación serán alcanzables.

Ejemplo Eliminar las variables inútiles de la siguiente gramática G .

$$G : \begin{cases} S \rightarrow SBS \mid BC \mid Bb \\ A \rightarrow AA \mid aA \\ B \rightarrow aBCa \mid b \\ C \rightarrow aC \mid ACC \mid abb \\ D \rightarrow aAB \mid ab \\ E \rightarrow aS \mid bAA \\ F \rightarrow aDb \mid aF \end{cases}$$

Solución. Ejecutamos los algoritmos en el orden (I):

$$\mathbf{TERM}_1 = \{B, C, D\}.$$

$$\mathbf{TERM}_2 = \{B, C, D\} \cup \{S, F\}.$$

$$\mathbf{TERM}_3 = \{B, C, D, S, F\} \cup \{E\} = \{B, C, D, S, F, E\}.$$

$$\mathbf{TERM}_4 = \{B, C, D, S, F, E\} \cup \{ \}.$$

La única variable no-terminable de G es A . Por lo tanto, G es equivalente a la siguiente gramática G_1 :

$$G_1 : \begin{cases} S \rightarrow SBS \mid BC \mid Bb \\ B \rightarrow aBCa \mid b \\ C \rightarrow aC \mid abb \\ D \rightarrow ab \\ E \rightarrow aS \\ F \rightarrow aDb \mid aF \end{cases}$$

Variables alcanzables de G_1 :

$$\mathbf{ALC}_1 = \{S\}.$$

$$\mathbf{ALC}_2 = \{S\} \cup \{B, C\}.$$

$$\mathbf{ALC}_3 = \{S, B, C\} \cup \{ \}.$$

Las variables D, E, F son no alcanzables. Por lo tanto, G es equivalente a la siguiente gramática G_2 , que no tiene variables inútiles.

$$G_2 : \begin{cases} S \rightarrow SBS \mid BC \mid Bb \\ B \rightarrow aBCa \mid b \\ C \rightarrow aC \mid abb \end{cases}$$

Ejercicios de la sección 4.8

① Eliminar las variables inútiles de la siguiente gramática:

$$G : \begin{cases} S \rightarrow SS \mid SBB \mid CCE \\ A \rightarrow aE \mid bE \\ B \rightarrow bB \mid Db \\ C \rightarrow aC \mid bB \\ D \rightarrow aDb \mid ab \mid \lambda \\ E \rightarrow aA \mid bB \end{cases}$$

② Eliminar las variables inútiles de la siguiente gramática:

$$G : \begin{cases} S \rightarrow EA \mid SaBb \mid aEb \\ A \rightarrow DaD \mid bD \\ B \rightarrow bB \mid Ab \mid \lambda \\ C \rightarrow aC \mid bBC \\ D \rightarrow aEb \mid ab \\ E \rightarrow aA \mid bB \mid \lambda \\ F \rightarrow Fb \mid Fa \mid a \end{cases}$$

4.9. Eliminación de las producciones λ

4.9.1. Definiciones.

- (i) Una producción de la forma $A \rightarrow \lambda$ se llama **producción λ** .
- (ii) Una variable A se llama **anulable** si $A \xrightarrow{*} \lambda$.

4.9.2. Algoritmo para encontrar las variables anulables.

ANUL₁ := { $A \in V : A \rightarrow \lambda$ es una producción}.

ANUL _{$i+1$} := **ANUL** _{i} \cup { $A \in V : \exists$ producción $A \rightarrow w, w \in (\text{ANUL}_i)^*$ }.

Obtenemos una sucesión creciente de conjuntos de variables:

$$\text{ANUL}_1 \subseteq \text{ANUL}_2 \subseteq \text{ANUL}_3 \subseteq \dots$$

Como el conjunto de variables es finito, existe $k \in \mathbb{N}$ tal que

$$\text{ANUL}_k = \text{ANUL}_{k+1} = \text{ANUL}_{k+2} = \dots$$

El conjunto **ANUL** de variables anulables es entonces

$$\text{ANUL} := \bigcup_{i \geq 1} \text{ANUL}_i$$

El anterior algoritmo se puede presentar de la siguiente forma:

INICIALIZAR:

$$\text{ANUL} := \{A \in V : A \rightarrow \lambda \text{ es una producción}\}$$

REPETIR:

$$\text{ANUL} := \text{ANUL} \cup \{A \in V : \exists \text{ prod. } A \rightarrow w, w \in (\text{ANUL})^*\}$$

HASTA:

No se añaden nuevas variables a **ANUL**

4.9.3 Teorema. Dada una GIC G , se puede construir una GIC G' equivalente a G sin producciones λ , excepto (posiblemente) $S \rightarrow \lambda$.

Demostración. Una vez que haya sido determinado el conjunto **ANUL** de variables anulables, por medio del algoritmo 4.9.2, las producciones de λ se pueden eliminar (excepto $S \rightarrow \lambda$) añadiendo nuevas producciones que simulen el efecto de las producciones λ eliminadas. Más concretamente, por cada producción $A \rightarrow u$ de G se añaden las producciones de la forma $A \rightarrow v$ obtenidas suprimiendo de la cadena u una, dos o más variables anulables presentes, de todas las formas posibles. La gramática G' así obtenida es equivalente a la gramática original G . \square

Ejemplo Eliminar las producciones λ de la siguiente gramática G .

$$G : \quad \left\{ \begin{array}{l} S \rightarrow AB \mid ACA \mid ab \\ A \rightarrow aAa \mid B \mid CD \\ B \rightarrow bB \mid bA \\ C \rightarrow cC \mid \lambda \\ D \rightarrow aDc \mid CC \mid ABb \end{array} \right.$$

Solución. Primero encontramos las variables anulables de G por medio del algoritmo 4.9.2:

$$\mathbf{ANUL}_1 = \{C\}.$$

$$\mathbf{ANUL}_2 = \{C\} \cup \{D\} = \{C, D\}.$$

$$\mathbf{ANUL}_3 = \{C, D\} \cup \{A\} = \{C, D, A\}.$$

$$\mathbf{ANUL}_4 = \{C, D, A\} \cup \{S\} = \{C, D, A, S\}.$$

$$\mathbf{ANUL}_5 = \{C, D, A, S\} \cup \{\} = \{C, D, A, S\}.$$

Al eliminar de G la producciones λ (la única es $C \rightarrow \lambda$) se obtiene la siguiente gramática equivalente a G :

$$G' : \begin{cases} S \rightarrow AB \mid ACA \mid ab \mid B \mid CA \mid AA \mid AC \mid A \mid C \mid \lambda \\ A \rightarrow aAa \mid B \mid CD \mid aa \mid C \mid D \\ B \rightarrow bB \mid bA \mid b \\ C \rightarrow cC \mid c \\ D \rightarrow aDc \mid CC \mid ABb \mid ac \mid C \mid Bb \end{cases}$$

Ejercicios de la sección 4.9

① Eliminar las producciones λ de la siguiente gramática:

$$G : \begin{cases} S \longrightarrow BCB \\ A \longrightarrow aA \mid ab \\ B \longrightarrow bBa \mid A \mid DC \\ C \longrightarrow aCb \mid D \mid b \\ D \longrightarrow aB \mid \lambda \end{cases}$$

② Eliminar las producciones λ de la siguiente gramática:

$$G : \begin{cases} S \rightarrow EA \mid SaBb \mid aEb \\ A \rightarrow DaD \mid bD \mid BEB \\ B \rightarrow bB \mid Ab \mid \lambda \\ D \rightarrow aEb \mid ab \\ E \rightarrow aA \mid bB \mid \lambda \end{cases}$$

4.10. Eliminación de las producciones unitarias

4.10.1. Definiciones.

- (i) Una producción de la forma $A \rightarrow B$ donde A y B son variables, se llama **producción unitaria**.
- (ii) El **conjunto unitario** de una variable A (también llamado **conjunto cadena** de A) se define de la siguiente manera:

$$\mathbf{UNIT}(A) := \{X \in V : \exists \text{ una derivación } A \xrightarrow{*} X \text{ que usa únicamente producciones unitarias}\}.$$

Por definición, $A \in \mathbf{UNIT}(A)$.

4.10.2. Algoritmo para encontrar las producciones unitarias.

El siguiente algoritmo sirve para encontrar el conjunto unitario $\mathbf{UNIT}(A)$ de una variable A .

$$\mathbf{UNIT}_1(A) := \{A\}.$$

$$\mathbf{UNIT}_{i+1}(A) := \mathbf{UNIT}_i(A) \cup \{X \in V : \exists \text{ producción } Y \rightarrow X, Y \in \mathbf{UNIT}_i(A)\}.$$

Para el conjunto de producciones unitarias se tiene que:

$$\mathbf{UNIT}_1(A) \subseteq \mathbf{UNIT}_2(A) \subseteq \mathbf{UNIT}_3(A) \subseteq \dots$$

Puesto que el conjunto de variables es finito, la anterior es una sucesión finita y se tiene

$$\mathbf{UNIT}(A) = \bigcup_{i \geq 1} \mathbf{UNIT}_i(A)$$

El anterior algoritmo se puede representar de la siguiente forma:

INICIALIZAR:

$$\mathbf{UNIT}(A) := \{A\}$$

REPETIR:

$$\mathbf{UNIT}(A) := \mathbf{UNIT}(A) \cup \{X \in V : \exists \text{ una producción } Y \rightarrow X \text{ con } Y \in \mathbf{UNIT}(A)\}$$

HASTA:

No se añaden nuevas variables $\mathbf{UNIT}(A)$

4.10.3 Teorema. Dada una GIC G , se puede construir una GIC G' equivalente a G sin producciones unitarias.

Demostración. Las producciones unitarias de G se pueden eliminar añadiendo para cada variable A de G las producciones (no unitarias) de las variables contenidas en el conjunto unitario $\text{UNIT}(A)$. La gramática G' así obtenida es equivalente a la gramática original G . \square

Ejemplo Eliminar las producciones unitarias de la siguiente gramática.

$$G : \begin{cases} S \rightarrow AS \mid AA \mid BA \mid \lambda \\ A \rightarrow aA \mid a \\ B \rightarrow bB \mid bC \mid C \\ C \rightarrow aA \mid bA \mid B \mid ab \end{cases}$$

Solución. Aplicando el algoritmo para cada una de las variables de G , se tiene que:

$$\text{UNIT}_1(S) = \{S\}.$$

$$\text{UNIT}_2(S) = \{S\} \cup \{\ } = \{S\}.$$

$$\text{UNIT}_1(A) = \{A\}.$$

$$\text{UNIT}_2(A) = \{A\} \cup \{ \ } = \{A\}.$$

$$\text{UNIT}_1(B) = \{B\}.$$

$$\text{UNIT}_2(B) = \{B\} \cup \{C\} = \{B, C\}.$$

$$\text{UNIT}_3(B) = \{B, C\} \cup \{B\} = \{B, C\}.$$

$$\text{UNIT}_1(C) = \{C\}.$$

$$\text{UNIT}_2(C) = \{C\} \cup \{B\} = \{C, B\}.$$

$$\text{UNIT}_3(C) = \{C, B\} \cup \{C\} = \{C, B\}.$$

Eliminando las producciones unitarias se obtiene una gramática G' equivalente:

$$G' : \begin{cases} S \rightarrow AS \mid AA \mid BA \mid \lambda \\ A \rightarrow aA \mid a \\ B \rightarrow bB \mid bC \mid aA \mid bA \mid ab \\ C \rightarrow aA \mid bA \mid ab \mid bB \mid bC \end{cases}$$

Ejemplo Eliminar las producciones unitarias de la siguiente gramática.

$$G : \begin{cases} S \rightarrow ACA \mid CA \mid AA \mid A \mid C \mid \lambda \\ A \rightarrow aAa \mid aa \mid B \mid C \\ B \rightarrow cC \mid D \mid C \\ C \rightarrow bC \\ D \rightarrow aA \mid \lambda \end{cases}$$

Solución. Realizando el algoritmo para cada una de las variables de G se obtiene:

$$\text{UNIT}(S) = \{S, A, C, B, D\}.$$

$$\text{UNIT}(A) = \{A, B, C, D\}.$$

$$\text{UNIT}(B) = \{B, C, D\}.$$

$$\text{UNIT}(C) = \{C\}.$$

$$\text{UNIT}(D) = \{D\}.$$

Eliminando las producciones unitarias se obtiene una gramática G' equivalente:

$$G' : \begin{cases} S \rightarrow ACA \mid CA \mid AA \mid \lambda \mid aAa \mid aa \mid bC \mid cC \mid aA \\ A \rightarrow aAa \mid aa \mid cC \mid bC \mid aA \mid \lambda \\ B \rightarrow cC \mid bC \mid aA \mid \lambda \\ C \rightarrow bC \\ D \rightarrow aA \mid \lambda \end{cases}$$

Ejercicios de la sección 4.10

- ① Eliminar las producciones unitarias de la siguiente gramática:

$$G : \begin{cases} S \rightarrow Ba \mid A \mid \lambda \\ A \rightarrow Aa \mid a \\ B \rightarrow bB \mid S \end{cases}$$

- ② Eliminar las producciones unitarias de la siguiente gramática:

$$G : \begin{cases} S \rightarrow BBa \mid A \mid B \mid ab \mid \lambda \\ A \rightarrow Aa \mid B \mid D \mid aC \\ B \rightarrow bB \mid aA \mid b \\ C \rightarrow ABb \mid A \mid aB \\ D \rightarrow cC \mid c \end{cases}$$

- ③ Eliminar las producciones unitarias de la siguiente gramática:

$$G : \begin{cases} S \rightarrow ACA \mid ab \mid B \mid CA \mid A \mid C \mid \lambda \\ A \rightarrow aAa \mid B \mid CD \mid aa \mid D \\ B \rightarrow bB \mid bA \mid b \\ C \rightarrow cC \mid c \\ D \rightarrow ABb \mid ac \mid C \mid Bb \end{cases}$$

4.11. Forma Normal de Chomsky (FNC)

Una GIC G está en **Forma Normal de Chomsky (FNC)** si satisface:

1. G no tiene variables inútiles.
2. G no tiene producciones λ (excepto posiblemente $S \rightarrow \lambda$).
3. Todas las producciones son de la forma: $A \rightarrow a$ (producciones simples) ó $A \rightarrow BC$ (producciones binarias).

En particular, una gramática en FNC no tiene producciones unitarias.

4.11.1 Teorema (Procedimiento de conversión a FNC). *Toda GIC G es equivalente a una gramática en Forma Normal de Chomsky.*

Demostración. Podemos transformar G en una gramática en FNC, equivalente a G , ejecutando los algoritmos de las secciones anteriores en el siguiente orden:

1. Eliminar las variables no terminales.
2. Eliminar las variables no alcanzables.
3. Eliminar las producciones λ (excepto, posiblemente, $S \rightarrow \lambda$).
4. Eliminar las producciones unitarias.
5. Las producciones resultantes (diferentes de $S \rightarrow \lambda$) son de la forma: $A \rightarrow a$ ó $A \rightarrow w$, donde $|w| \geq 2$. Estas últimas se pueden simular con producciones de la forma $A \rightarrow BC$ o $A \rightarrow a$. Se introduce primero, para cada $a \in \Sigma$, una variable nueva T_a cuya única producción es $T_a \rightarrow a$. A continuación, se introducen nuevas variables, con producciones binarias, para simular las producciones deseadas. \square

La parte 5 del procedimiento anterior se ilustra en los dos siguientes ejemplos.

Ejemplo Simular la producción $A \rightarrow abBaC$ con producciones simples y binarias.

Solución. Introducimos las variables T_a y T_b , y las producciones $T_a \rightarrow a$ y $T_b \rightarrow b$. Entonces $A \rightarrow abBaC$ se simula con:

$$\begin{cases} A \rightarrow T_a T_b B T_a C \\ T_a \rightarrow a \\ T_b \rightarrow b \end{cases}$$

Ahora introducimos nuevas variables T_1, T_2, T_3 y las producciones binarias necesarias. Las únicas producciones de estas nuevas variables son las mostradas:

$$\left\{ \begin{array}{l} A \rightarrow T_a T_1 \\ T_1 \rightarrow T_b T_2 \\ T_2 \rightarrow B T_3 \\ T_3 \rightarrow T_a C \\ T_a \rightarrow a \\ T_b \rightarrow b \end{array} \right.$$

Ejemplo Simular la producción $A \rightarrow BAaCbb$ con producciones simples y binarias.

Solución. Introducimos las variables T_a y T_b , y las producciones $T_a \rightarrow a$ y $T_a \rightarrow b$. Entonces $A \rightarrow BAaCbb$ se simula con:

$$\left\{ \begin{array}{l} A \rightarrow B A T_a C T_b T_b \\ T_a \rightarrow a \\ T_b \rightarrow b \end{array} \right.$$

Ahora introducimos nuevas variables T_1, T_2, T_3, T_4 y las producciones binarias necesarias. Las únicas producciones de estas nuevas variables son las mostradas:

$$\left\{ \begin{array}{l} A \rightarrow B T_1 \\ T_1 \rightarrow A T_2 \\ T_2 \rightarrow T_a T_3 \\ T_3 \rightarrow C T_4 \\ T_4 \rightarrow T_b T_b \\ T_a \rightarrow a \\ T_b \rightarrow b \end{array} \right.$$

En los siguientes ejemplos se ilustra el procedimiento completo para convertir una gramática dada a la Forma Normal de Chomsky (FNC).

Ejemplo Encontrar una GIC en FNC equivalente a la siguiente a la gramática:

$$G : \left\{ \begin{array}{l} S \rightarrow AB \mid aBC \mid SBS \\ A \rightarrow aA \mid C \\ B \rightarrow bbB \mid b \\ C \rightarrow cC \mid \lambda \end{array} \right.$$

Solución. El conjunto de variables terminables es

$$\mathbf{TERM} = \{B, C, S, A\},$$

y el conjunto de variables alcanzables es

$$\mathbf{ALC} = \{S, A, B, C\}.$$

Es decir, la gramática no tiene variables inútiles. El conjunto de variables anulables es

$$\mathbf{ANUL} = \{C, A\}.$$

Al eliminar las producciones λ de G (la única es $C \rightarrow \lambda$) se obtiene la gramática equivalente G_1 :

$$G_1 : \begin{cases} S \rightarrow AB \mid aBC \mid SBS \mid B \mid aB \\ A \rightarrow aA \mid C \mid a \\ B \rightarrow bbB \mid b \\ C \rightarrow cC \mid c \end{cases}$$

A continuación encontramos los conjuntos unitarios de todas las variables:

$$\mathbf{UNIT}(S) = \{S, B\}.$$

$$\mathbf{UNIT}(A) = \{A, C\}.$$

$$\mathbf{UNIT}(B) = \{B\}.$$

$$\mathbf{UNIT}(C) = \{C\}.$$

Al eliminar las producciones unitarias obtenemos la gramática equivalente G_2 :

$$G_2 : \begin{cases} S \rightarrow AB \mid aBC \mid SBS \mid aB \mid bbB \mid b \\ A \rightarrow aA \mid a \mid cC \mid c \\ B \rightarrow bbB \mid b \\ C \rightarrow cC \mid c \end{cases}$$

Luego introducimos las variables nuevas T_a , T_b y T_c , y las producciones $T_a \rightarrow a$, $T_b \rightarrow b$ y $T_c \rightarrow c$ con el propósito de que todas las producciones sean unitarias o de la forma $A \rightarrow w$, donde $|w| \geq 2$.

$$G_3 : \begin{cases} S \rightarrow AB \mid T_aBC \mid SBS \mid T_aB \mid T_bT_bB \mid b \\ A \rightarrow T_aA \mid a \mid T_cC \mid c \\ B \rightarrow T_bT_bB \mid b \\ C \rightarrow T_cC \mid c \\ T_a \rightarrow a \\ T_b \rightarrow b \\ T_c \rightarrow c \end{cases}$$

Finalmente, se introducen nuevas variables, con producciones binarias, para simular las producciones de la forma $A \rightarrow w$, donde $|w| \geq 2$:

$$G_4 : \begin{cases} S \rightarrow AB \mid T_a T_1 \mid ST_2 \mid T_a B \mid T_b T_3 \mid b \\ A \rightarrow T_a A \mid T_C C \mid a \mid c \\ B \rightarrow T_b T_3 \mid b \\ C \rightarrow T_c C \mid c \\ T_1 \rightarrow BC \\ T_2 \rightarrow BS \\ T_3 \rightarrow T_b B \\ T_a \rightarrow a \\ T_b \rightarrow b \\ T_c \rightarrow c \end{cases}$$

Ejemplo Encontrar una GIC en FNC equivalente a la siguiente a la gramática:

$$G : \begin{cases} S \rightarrow aS \mid aA \mid D \\ A \rightarrow aAa \mid aAD \mid \lambda \\ B \rightarrow aB \mid BC \\ C \rightarrow aBb \mid CC \mid \lambda \\ D \rightarrow aB \mid bA \mid aa \mid A \end{cases}$$

Solución. $\text{TERM} = \{A, C, D, S\}$. Eliminando la variable no-terminable B obtenemos:

$$G_1 : \begin{cases} S \rightarrow aS \mid aA \mid D \\ A \rightarrow aAa \mid aAD \mid \lambda \\ C \rightarrow CC \mid \lambda \\ D \rightarrow bA \mid aa \mid A \end{cases}$$

El conjunto de las variables alcanzables de G_1 es $\text{ALC} = \{S, A, D\}$. Eliminando la variable no-alcanzable C obtenemos:

$$G_2 : \begin{cases} S \rightarrow aS \mid aA \mid D \\ A \rightarrow aAa \mid aAD \mid \lambda \\ D \rightarrow bA \mid aa \mid A \end{cases}$$

El conjunto de variables anulables de G_2 es **ANUL** = { A, D, S }. Eliminando las producciones λ obtenemos:

$$G_3 : \begin{cases} S \rightarrow aS \mid aA \mid D \mid a \mid \lambda \\ A \rightarrow aAa \mid aAD \mid aa \mid aA \mid aD \mid a \\ D \rightarrow bA \mid aa \mid A \mid b \end{cases}$$

A continuación encontramos los conjuntos unitarios de todas las variables:

$$\begin{aligned}\mathbf{UNIT}(S) &= \{S, D, A\}. \\ \mathbf{UNIT}(A) &= \{A\}. \\ \mathbf{UNIT}(D) &= \{D, A\}.\end{aligned}$$

Al eliminar las producciones unitarias obtenemos la gramática equivalente G_4 :

$$G_4 : \begin{cases} S \rightarrow aS \mid aA \mid a \mid \lambda \mid aAa \mid aAD \mid aa \mid aD \mid bA \mid b \\ A \rightarrow aAa \mid aAD \mid aa \mid aA \mid aD \mid a \\ D \rightarrow bA \mid aa \mid b \mid aAa \mid aAD \mid aa \mid aA \mid aD \mid a \end{cases}$$

Finalmente, simulamos las producciones de G_4 con producciones unitarias y binarias:

$$G_5 : \begin{cases} S \rightarrow T_a S \mid T_a A \mid T_a T_1 \mid T_a T_2 \mid T_a T_a \mid T_a D \mid T_b A \mid a \mid b \mid \lambda \\ A \rightarrow T_a T_1 \mid T_a T_2 \mid T_a T_a \mid T_a A \mid T_a D \mid a \\ D \rightarrow T_b A \mid T_a T_a \mid b \mid T_a T_1 \mid T_a T_2 \mid T_a T_a \mid T_a A \mid T_a D \mid a \\ T_1 \rightarrow AT_a \\ T_2 \rightarrow AD \\ T_a \rightarrow a \\ T_b \rightarrow b \end{cases}$$

En algunas aplicaciones de la FNC es necesario exigir que la variable inicial S no aparezca en el cuerpo de ninguna producción. Si S aparece en el lado derecho de alguna producción se dice que S es **recursiva** ya que esto da lugar a derivaciones de la forma $S \xrightarrow{+} uSv$, con $u, v \in (V \cup \Sigma)^*$. El siguiente teorema es un resultado muy sencillo; establece que cualquier GIC se puede transformar en una GIC equivalente en la cual la variable inicial no es recursiva.

4.11.2 Teorema. *Dada una GIC $G = (V, \Sigma, S, P)$ se puede construir una GIC $G' = (V', \Sigma, S', P')$ equivalente a G de tal manera que el símbolo inicial S' de G' no aparezca en lado derecho de las producciones de G' .*

Demostración. La nueva gramática G' tiene una variable más que G , la variable S' , que actúa como la nueva variable inicial. Es decir, $V' = V \cup \{S'\}$. El conjunto de producciones P' está dado por $P' = P \cup \{S' \rightarrow S\}$. Es claro que $L(G) = L(G')$ y el símbolo inicial S' no aparece en el cuerpo de las producciones. \square

Según este resultado, el papel de la variable inicial de la nueva gramática G' es únicamente iniciar las derivaciones.

Ejemplo Encontrar una GIC G' equivalente a la siguiente gramática G de tal manera que la variable inicial de G' no sea recursiva.

$$G : \begin{cases} S \rightarrow ASB \mid BB \\ A \rightarrow aA \mid a \\ B \rightarrow bBS \mid \lambda \end{cases}$$

Solución. Según se indicó en la demostración del Teorema 4.11.2, la gramática pedida G' es

$$G' : \begin{cases} S' \rightarrow S \\ S \rightarrow ASB \mid BB \\ A \rightarrow aA \mid a \\ B \rightarrow bBS \mid \lambda \end{cases}$$

Nótese que S sigue siendo recursiva pero ya no es la variable inicial de la gramática.

Ejemplo Encontrar una GIC en FNC equivalente a la gramática G del ejemplo anterior, de tal manera que su variable inicial no sea recursiva.

Solución. Comenzamos transformando G en G' , como se hizo en el ejemplo anterior. En G' todas las variables son útiles y **ANUL** = $\{B, S, S'\}$. Eliminando las producciones λ obtenemos:

$$G_1 : \begin{cases} S' \rightarrow S \mid \lambda \\ S \rightarrow ASB \mid BB \mid AB \mid AS \mid A \\ A \rightarrow aA \mid a \\ B \rightarrow bBS \mid bS \mid bB \mid b \end{cases}$$

Los conjuntos unitarios son: **UNIT**(S') = $\{S', S\}$, **UNIT**(S) = $\{S, A\}$, **UNIT**(A) = $\{A\}$, **UNIT**(B) = $\{B\}$. Eliminando las producciones unita-

rias se obtiene la gramática:

$$G_2 : \begin{cases} S' \rightarrow ASB | BB | AB | AS | aA | a | \lambda \\ S \rightarrow ASB | BB | AB | AS | aA | a \\ A \rightarrow aA | a \\ B \rightarrow bBS | bS | bB | b \end{cases}$$

Simulando las producciones de G_2 con producciones unitarias y binarias se obtiene:

$$G_3 : \begin{cases} S' \rightarrow AT_1 | BB | AB | AS | T_aA | a | \lambda \\ S \rightarrow AT_1 | BB | AB | AS | T_aA | a \\ A \rightarrow T_aA | a \\ B \rightarrow T_bT_2 | T_bS | T_bB | b \\ T_a \rightarrow a \\ T_b \rightarrow b \\ T_1 \rightarrow SB \\ T_2 \rightarrow BS \end{cases}$$

Ejercicios de la sección 4.11

① Encontrar una gramática en FNC equivalente a la siguiente GIC:

$$G : \begin{cases} S \rightarrow ABC | BaC | aB \\ A \rightarrow Aa | a \\ B \rightarrow BAB | bab \\ C \rightarrow cC | c \end{cases}$$

② Encontrar una gramática en FNC equivalente a la siguiente GIC:

$$G : \begin{cases} S \rightarrow aASb | BAb \\ A \rightarrow Aa | a | \lambda \\ B \rightarrow BAB | bAb \\ C \rightarrow cCS | \lambda \end{cases}$$

③ Para la gramática del ejercicio ② encontrar una GIC equivalente en FNC, de tal manera que su variable inicial no sea recursiva.

4.12. Forma Normal de Greibach (FNG)

Una GIC está en **Forma Normal de Greibach (FNG)** si

1. La variable inicial no es recursiva.
2. G no tiene variables inútiles.
3. G no tiene producciones λ (excepto posiblemente $S \rightarrow \lambda$).
4. Todas las producciones son de la forma: $A \rightarrow a$ (producciones simples) ó $A \rightarrow aB_1B_2\dots B_k$, donde las B_i son variables.

Las derivaciones en una gramática que esté en FNG tienen dos características notables: en cada paso aparece un único terminal y, en segundo lugar, la derivación de una cadena de longitud n ($n \geq 1$) tiene exactamente n pasos.

Existe un procedimiento algorítmico para transformar una GIC dada en una gramática equivalente en FNG. Para presentar el procedimiento necesitamos algunos resultados preliminares.

4.12.1 Definición. Una variable se llama **recursiva a la izquierda** si tiene una producción de la forma:

$$A \rightarrow Aw, \quad w \in (V \cup \Sigma)^*.$$

La recursividad a izquierda se puede eliminar, como se muestra en el siguiente teorema.

4.12.2 Teorema (Eliminación de la recursividad a la izquierda). *Las producciones de una variable A cualquiera se pueden dividir en dos clases:*

$$\left\{ \begin{array}{l} A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \\ A \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_m \end{array} \right.$$

donde $\alpha_i, \beta_i \in (V \cup \Sigma)^*$ y el primer símbolo de β_i es diferente de A . Sin alterar el lenguaje generado, las anteriores producciones se pueden simular, reemplazándolas por las siguientes:

$$\left\{ \begin{array}{l} A \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_m \mid \beta_1Z \mid \beta_2Z \mid \dots \mid \beta_mZ \\ Z \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n \mid \alpha_1Z \mid \alpha_2Z \mid \dots \mid \alpha_nZ \end{array} \right.$$

donde Z es una variable completamente nueva.

Demostración. Se puede observar que, tanto con las producciones originales como las nuevas, A genera el lenguaje

$$\{\beta_1, \beta_2, \dots, \beta_m\} \cdot \{\alpha_1, \alpha_2, \dots, \alpha_n\}^*$$

□

4.12.3 Lema. En una GIC cualquiera, una producción $A \rightarrow uBv$ se puede reemplazar (similar) por

$$A \rightarrow uw_1v \mid uw_2v \mid \dots \mid uw_nv$$

siendo $B \rightarrow w_1 \mid w_2 \mid \dots \mid w_n$ todas las producciones de B .

Demostración. Inmediato. □

4.12.4 Teorema (Procedimiento de conversión a FNG). Toda GIC G es equivalente a una gramática en Forma Normal de Greibach.

Demostración. Suponemos que la gramática dada está en FNC. Esto simplifica el procedimiento, aunque éste es válido (con modificaciones menores) para una gramática arbitraria si se eliminan primero las variables inútiles, las producciones λ y las producciones unitarias. La conversión a FNG se realiza ejecutando los siguientes pasos:

1. Enumerar las variables en un orden arbitrario pero fijo durante el procedimiento. S debe ser la variable con orden 1.
2. Para cada variable A de la gramática original, siguiendo el orden elegido, modificar sus producciones de tal manera que el primer símbolo del cuerpo de cada producción (primer símbolo a la derecha de la flecha) sea un terminal o una variable con orden mayor que el de A . Para lograrlo se usa el teorema de eliminación de la recursividad a la izquierda, Teorema 4.12.2, y el Lema 4.12.3, todas las veces que sea necesario.
3. Utilizar el Lema 4.12.3 para modificar las producciones de las variables *originales* de tal manera que el primer símbolo del cuerpo de cada producción sea un terminal. Esto se hace siguiendo el orden inverso de enumeración de las variables: última, penúltima, etc.
4. Utilizar de nuevo el Lema 4.12.3, para modificar las producciones de las variables *nuevas* de tal manera que el primer símbolo del cuerpo de cada producción sea un terminal. □

Ejemplo Encontrar una gramática en FNG equivalente a la siguiente gramática (que está en FNC):

$$G : \begin{cases} S \rightarrow AA \mid a \\ A \rightarrow AA \mid b \end{cases}$$

Paso 1: Aquí solamente hay un orden posible para variables: S, A .

Paso 2: En este paso sólo hay que eliminar la recursividad a izquierda de la variable A . Al hacerlo se obtiene la gramática:

$$\begin{cases} S \rightarrow AA \mid a \\ A \rightarrow b \mid bZ \\ Z \rightarrow A \mid AZ \end{cases}$$

Paso 3:

$$\begin{cases} S \rightarrow bA \mid bZA \mid a \\ A \rightarrow b \mid bZ \\ Z \rightarrow A \mid AZ \end{cases}$$

Paso 4:

$$\begin{cases} S \rightarrow bA \mid bZA \mid a \\ A \rightarrow b \mid bZ \\ Z \rightarrow b \mid bZ \mid BZZ \end{cases}$$

Ejemplo Encontrar una gramática en FNG equivalente a la siguiente gramática (que está en FNC):

$$\begin{cases} S \rightarrow AB \mid BC \\ A \rightarrow AB \mid a \\ B \rightarrow AA \mid CB \mid a \\ C \rightarrow a \mid b \end{cases}$$

Paso 1: Orden de las variables: S, B, A, C . Este orden es muy adecuado porque el cuerpo de las producciones de B comienza con A o C , que son variables de orden mayor.

$$\begin{cases} S \rightarrow AB \mid BC \\ B \rightarrow AA \mid CB \mid a \\ A \rightarrow AB \mid a \\ C \rightarrow a \mid b \end{cases}$$

Paso 2:

$$\left\{ \begin{array}{l} S \rightarrow AB \mid BC \\ B \rightarrow AA \mid CB \mid a \\ A \rightarrow a \mid aZ \\ C \rightarrow a \mid b \\ Z \rightarrow B \mid BZ \end{array} \right.$$

Paso 3:

$$\left\{ \begin{array}{l} S \rightarrow aB \mid aZB \mid aAC \mid aZAC \mid aBC \mid bBC \mid aC \\ B \rightarrow aA \mid aZA \mid aB \mid bB \mid a \\ A \rightarrow a \mid aZ \\ C \rightarrow a \mid b \\ Z \rightarrow B \mid BZ \end{array} \right.$$

Paso 4:

$$\left\{ \begin{array}{l} S \rightarrow aB \mid aZB \mid aAC \mid aZAC \mid aBC \mid bBC \mid aC \\ B \rightarrow aA \mid aZA \mid aB \mid bB \mid a \\ A \rightarrow a \mid aZ \\ C \rightarrow a \mid b \\ Z \rightarrow aA \mid aZA \mid aB \mid bB \mid a \mid aAZ \mid aZAZ \mid aBZ \mid bBZ \mid aZ \end{array} \right.$$

El siguiente ejemplo ilustra que el procedimiento de conversión a la forma FNG puede dar lugar a docenas de producciones, incluso a partir de una gramática relativamente sencilla.

Ejemplo

Encontrar una gramática en FNG equivalente a la siguiente gramática:

$$\left\{ \begin{array}{l} S \rightarrow AB \\ A \rightarrow AB \mid CB \mid a \\ B \rightarrow AB \mid b \\ C \rightarrow AC \mid c \end{array} \right.$$

Paso 1: Orden de las variables: S, A, B, C .

Paso 2:

$$\left\{ \begin{array}{l} S \rightarrow AB \\ A \rightarrow CB \mid a \mid CBZ_1 \mid aZ_1 \\ B \rightarrow CBB \mid aB \mid CBZ_1B \mid aZ_1B \mid b \\ C \rightarrow CBC \mid aC \mid CBZ_1C \mid aZ_1C \mid c \\ Z_1 \rightarrow B \mid BZ_1 \end{array} \right.$$

Prosiguiendo con el paso 2, se elimina la recursividad a izquierda de la variable C:

$$\left\{ \begin{array}{l} S \rightarrow AB \\ A \rightarrow CB \mid a \mid CBZ_1 \mid aZ_1 \\ B \rightarrow CBB \mid aB \mid CBZ_1B \mid aZ_1B \mid b \\ C \rightarrow aC \mid aZ_1C \mid c \mid aCZ_2 \mid aZ_1CZ_2 \mid cZ_2 \\ Z_1 \rightarrow B \mid BZ_1 \\ Z_2 \rightarrow BC \mid BZ_1C \mid BCZ_2 \mid BZ_1CZ_2 \end{array} \right.$$

Paso 3:

$$\left\{ \begin{array}{l} S \rightarrow 14 \text{ producciones} \\ A \rightarrow a \mid aZ_1 \mid 6 \text{ producciones} \mid 6 \text{ producciones} \\ B \rightarrow aB \mid aZ_1B \mid b \mid 6 \text{ producciones} \mid 6 \text{ producciones} \\ C \rightarrow aC \mid aZ_1C \mid c \mid aCZ_2 \mid aZ_1CZ_2 \mid cZ_2 \\ Z_1 \rightarrow B \mid BZ_1 \\ Z_2 \rightarrow BC \mid BZ_1C \mid BCZ_2 \mid BZ_1CZ_2 \end{array} \right.$$

Paso 4: El número de producciones de la nueva gramática se incrementa drásticamente:

$$\left\{ \begin{array}{l} S \rightarrow 14 \text{ producciones} \\ A \rightarrow a \mid aZ_1 \mid 6 \text{ producciones} \mid 6 \text{ producciones} \\ B \rightarrow aB \mid aZ_1B \mid b \mid 6 \text{ producciones} \mid 6 \text{ producciones} \\ C \rightarrow aC \mid aZ_1C \mid c \mid aCZ_2 \mid aZ_1CZ_2 \mid cZ_2 \\ Z_1 \rightarrow 15 \text{ producciones} \mid 15 \text{ producciones} \\ Z_2 \rightarrow 15 \text{ producciones} \mid 15 \text{ producciones} \mid 15 \text{ producciones} \mid 15 \text{ producciones} \end{array} \right.$$

La gramática original tenía 8 producciones; la nueva gramática en FNG tiene un total de 139 producciones.

Ejercicios de la sección 4.12

① Encontrar una gramática en FNG equivalente a la siguiente GIC:

$$\left\{ \begin{array}{l} S \rightarrow CA \mid AC \mid a \\ A \rightarrow BA \mid AB \mid b \\ B \rightarrow AA \mid a \mid b \\ C \rightarrow AC \mid CC \mid a \end{array} \right.$$

② Encontrar una gramática en FNG equivalente a la siguiente GIC:

$$\left\{ \begin{array}{l} S \rightarrow BB \mid BC \mid b \\ A \rightarrow AC \mid CA \mid a \\ B \rightarrow BB \mid a \\ C \rightarrow BC \mid CA \mid a \end{array} \right.$$

③ Encontrar una gramática en FNG equivalente a la siguiente GIC:

$$\left\{ \begin{array}{l} S \rightarrow SC \mid AA \mid a \\ A \rightarrow CA \mid AB \mid a \\ B \rightarrow AC \mid b \\ C \rightarrow CA \mid AS \mid b \end{array} \right.$$

Nota: hay que eliminar primero la recursividad de la variable S .

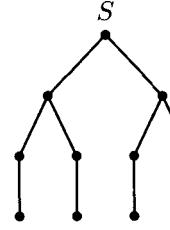
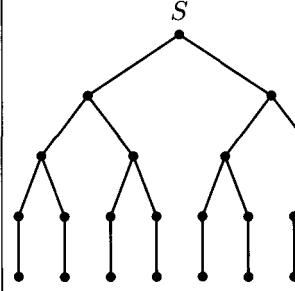
4.13. Lema de bombeo para LIC

Una de las consecuencias más importantes de la Forma Normal de Chomsky es el lema de bombeo para lenguajes independientes del contexto, el cual es útil, entre muchas aplicaciones, para demostrar que ciertos lenguajes no son LIC.

Nos referiremos a gramáticas en FNC con variable inicial no recursiva. Puesto que las producciones son unitarias ($A \rightarrow a$) o binarias ($A \rightarrow BC$), en cada nodo el árbol de una derivación se ramifica en dos nodos, a lo sumo. Tales árboles se denominan **binarios**. Si la producción $S \rightarrow \lambda$ está presente, su único propósito es generar la cadena λ .

4.13.1 Teorema. *Sea $G = (V, \Sigma, S, P)$ una gramática en FNC y $w \in \Sigma^*$. Si la longitud de la trayectoria más larga en un árbol de derivación de $S \xrightarrow{*} w$ tiene k (o menos) nodos, entonces $|w| \leq 2^{k-2}$. Aquí $k \geq 2$.*

Demostración. La siguiente tabla muestra las relaciones obtenidas entre $k = \text{número de nodos de la trayectoria más larga de } S \xrightarrow{*} w$ y la longitud de w , en los casos $k = 2$, $k = 3$, $k = 4$ y $k = 5$. En la tabla se muestran los casos extremos, es decir, los árboles con el mayor número posible de nodos. Se observa que $|w| \leq 2^{k-2}$. Una demostración rigurosa del caso general se hace por inducción sobre k . \square

$k = \text{número de nodos de la trayectoria más larga}$	Árbol de derivación	Longitud de w
$k = 2$		$ w = 1 = 2^0 = 2^{k-2}$
$k = 3$		$ w \leq 2 = 2^1 = 2^{k-2}$
$k = 4$		$ w \leq 4 = 2^2 = 2^{k-2}$
$k = 5$		$ w \leq 8 = 2^3 = 2^{k-2}$

4.13.2 Corolario. Sea $G = (V, \Sigma, S, P)$ una gramática en FNC y $w \in \Sigma^*$.

- (1) Si la longitud de la trayectoria más larga en un árbol de derivación de $S \xrightarrow{*} w$ tiene $k + 2$ (o menos) nodos, entonces $|w| \leq 2^k$. Aquí $k \geq 0$.

(2) Si $|w| > 2^k$ (con $k \geq 0$) entonces la longitud de la trayectoria más larga en un árbol de derivación de $S \xrightarrow{*} w$ tiene más de $k + 2$ nodos.

Demostración.

(1) Se sigue inmediatamente del Teorema 4.13.1.

(2) Es la afirmación contra-recíproca de la parte (1). \square

4.13.3. Lema de bombeo para LIC. *Dado un LIC L , existe una constante n (llamada constante de bombeo de L) tal que toda $z \in L$ con $|z| > n$ se puede descomponer en la forma $z = uvwxy$ donde:*

(1) $|vwx| \leq n$.

(2) $uv^iwx^i y \in L$ para todo $i \geq 0$.

(3) $v \neq \lambda$ ó $x \neq \lambda$.

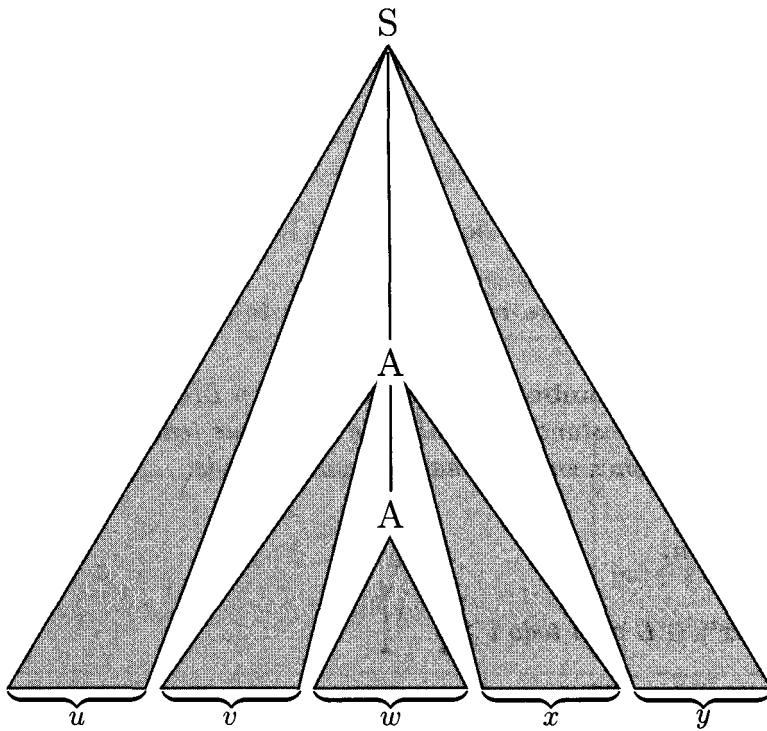
Demostración. Sea $G = (V, \Sigma, S, P)$ una gramática en FNC, con variable inicial no recursiva, tal que $L(G) = L$. Tal gramática existe por los Teoremas 4.11.1 y 4.11.2. Sea $k = |V|$ = número de variables de G y $n = 2^k$. Sea $z \in L$ con $|z| > n = 2^k$. Por la parte (2) del Corolario 4.13.2, la trayectoria más larga en el árbol de una derivación $S \xrightarrow{*} z$ tiene más de $k + 2$ nodos. Consideremos los últimos $k + 2$ nodos de tal trayectoria (siguiendo el orden que va desde la raíz S hasta las hojas del árbol). El último nodo de esa trayectoria es un terminal $a \in \Sigma$ y los restantes $k + 1$ nodos son variables. Como hay sólo k variables en la gramática, entonces hay por lo menos una variable $A \neq S$ repetida en la trayectoria. Por lo tanto, existen cadenas $u, v, w, x, y \in \Sigma^*$ tales que

$$S \xrightarrow{*} uAy, \quad A \xrightarrow{*} vAx, \quad A \xrightarrow{*} w.$$

Así que

$$S \xrightarrow{*} uAy \xrightarrow{*} uvAxy \xrightarrow{*} uvwxy = z.$$

La siguiente gráfica ilustra la situación:



Puesto que la trayectoria más larga en el árbol de derivación de $A \xrightarrow{*} vAx \xrightarrow{*} vwx$ tiene $k+2$ nodos o menos, por la parte (1) del Corolario 4.13.2, podemos concluir que $|vwx| \leq 2^k = n$. Además:

$$S \xrightarrow{*} uAy \xrightarrow{*} uvAxy \xrightarrow{*} uv^iAx^iy \xrightarrow{*} uv^iwx^iy, \quad \text{para todo } i \geq 0.$$

Obsérvese que el caso $i = 0$ corresponde a la derivación $S \xrightarrow{*} uAy \xrightarrow{*} uw y$.

Finalmente, la derivación $A \xrightarrow{*} vAx$ se puede escribir como

$$A \xrightarrow{} BC \xrightarrow{*} vAx$$

utilizando una producción de la forma $A \rightarrow BC$ como primer paso. Se deduce que u y x no pueden ser ambas λ porque se tendría $BC \xrightarrow{*} A$, lo cual es imposible en una gramática en FNC (recuérdese que la única producción λ en la gramática es, posiblemente, $S \rightarrow \lambda$; pero S no aparece en el cuerpo de ninguna producción de G ya que S no es recursiva). Se deduce entonces que $v \neq \lambda$ ó $x \neq \lambda$. Esto demuestra las propiedades (1), (2) y (3) del lema de bombeo. \square

Ejemplo Demostrar que el lenguaje $L = \{a^i b^i c^i : i \geq 0\}$ sobre $\Sigma = \{a, b, c\}$ no es un LIC.

Solución. Argumento por contradicción. Si L fuera LIC, por el lema de bombeo, existiría una constante de bombeo n . Sea $z = a^n b^n c^n$; se tiene que $z \in L$ y $|z| > n$. Por lo tanto, z se puede descomponer como $z = a^n b^n c^n = uvwxy$ con las propiedades (1), (2) y (3) del lema de bombeo. Puesto que $|vwx| \leq n$, en la cadena vwx no pueden aparecer los tres terminales a , b y c simultáneamente (para que aparezcan los tres terminales simultáneamente, una subcadena de $a^n b^n c^n$ debe tener longitud $\geq n+2$). Como $v \neq \lambda$ ó $x \neq \lambda$, se distinguen dos casos:

- Caso 1. Alguna de las cadenas v ó x contiene dos tipos de terminales. Entonces en uv^2wx^2y aparecen algunas b es seguidas de a es o algunas c es seguidas de b es. En cualquier caso, $uv^2wx^2y \notin L$.
- Caso 2. Las cadenas v y x contienen un sólo tipo de terminal cada una (o sólo a es o sólo b es o sólo c es). Como en vwx no aparecen los tres terminales a , b y c simultáneamente, en la cadena bombeada uv^2wx^2y se altera el número de dos de los terminales a , b , c , a lo sumo, pero no de los tres. Por lo tanto, $uv^2wx^2y \notin L$.

Pero el lema de bombeo afirma que $uv^2wx^2y \in L$. Esta contradicción muestra que L no es un LIC.

Ejemplo Demostrar que el lenguaje $L = \{a^i : i \text{ es primo}\}$ sobre $\Sigma = \{a\}$ no es un LIC.

Solución. Argumento por contradicción. Si L fuera LIC, por el lema de bombeo, existiría una constante de bombeo n . Sea $z = a^m$ con m primo $m > n$ y $m > 2$ (m existe porque el conjunto de los números primos es infinito). Entonces $z \in L$ y $|z| > n$. Por lo tanto, z se puede descomponer como $z = a^m = uvwxy$ con las propiedades (1), (2) y (3) del lema de bombeo.

Sea $|u| + |w| + |y| = k$; entonces $|v| + |x| = m - k \geq 1$. Por el lema de bombeo, $uv^iwx^i y \in L$ para todo $i \geq 0$; es decir, $|uv^iwx^i y|$ es primo para todo $i \geq 0$. Pero

$$|uv^iwx^i y| = k + |v^i| + |x^i| = k + i|v| + i|x| = k + i(|v| + |x|) = k + i(m - k).$$

- (i) Si $k = 0$, tomando $i = m$ se obtiene que $k + i(m - k) = 0 + i(m - 0) = im = mm$ que no es primo, pues $m > 2$.
- (ii) Si $k = 1$, tomando $i = 0$ se obtiene que $k + i(m - k) = 1 + i(m - 1) = 1 + 0(m - 1) = 1$ que no es un número primo.

(iii) Si $k > 1$, tomando $i = k$ se obtiene que

$$|uv^kwx^ky| = k + k(m - k) = k(1 + m - k),$$

el cual no es un número primo pues $k > 1$ y como $m - k \geq 1$, entonces $1 + m - k \geq 2$.

Por (i), (ii) y (iii) se puede escoger i de tal manera que $|uv^iwx^iy|$ no sea un número primo, lo cual contradice que $uv^iwx^iy \in L$ para todo $i \geq 0$. Esta contradicción muestra que L no es un LIC.

Ejercicios de la sección 4.13

Utilizar el lema de bombeo para demostrar que los siguientes lenguajes no son LIC:

- ① $L = \{a^i b^i c^j : j \geq i\}$, sobre $\Sigma = \{a, b, c\}$.
- ② $L = \{a^i b^j c^k : 1 \leq i \leq j \leq k\}$, sobre $\Sigma = \{a, b, c\}$.
- ③ $L = \{0^i 1^{2i} 0^i : i \geq 1\}$, sobre $\Sigma = \{0, 1\}$.
- ④ $L = \{a^i b^i c^i d^i : i \geq 0\}$, sobre $\Sigma = \{a, b, c, d\}$.
- ⑤ $L = \{a^i b^j c^i d^j : i, j \geq 0\}$, sobre $\Sigma = \{a, b, c, d\}$.
- ⑥ $L = \{ww : w \in \{0, 1\}^*\}$.
- !⑦ $L = \{a^i : i \text{ es un cuadrado perfecto}\}$.

4.14. Propiedades de clausura de los LIC

En la sección 3.2 se vio que los lenguajes regulares son cerrados bajo la unión, la concatenación, la estrella de Kleene y todas las operaciones booleanas. Los LIC poseen propiedades de clausura mucho más restringidas: son cerrados para las operaciones regulares (Teorema 4.14.1) pero, en general, no son cerrados para intersección, complementos ni diferencias (Teorema 4.14.2).

4.14.1 Teorema. *La colección de los lenguajes independientes del contexto es cerrada para las operaciones regulares (unión, concatenación y estrella de Kleene). Es decir, dadas GIC $G_1 = (V_1, \Sigma, S_1, P_1)$ y $G_2 = (V_2, \Sigma, S_2, P_2)$ tales que $L(G_1) = L_1$ y $L(G_2) = L_2$, se pueden construir GIC que generen los lenguajes $L_1 \cup L_2$, $L_1 L_2$ y L_1^* , respectivamente.*

Demostración. Si pérdida de generalidad, podemos suponer que G_1 y G_2 no tienen variables en común (en caso contrario, simplemente cambiamos los nombres de las variables). Para construir una GIC G que genere $L_1 \cup L_2$ introducimos una variable *nueva* S , la variable inicial de G , junto con las producciones $S \rightarrow S_1$ y $S \rightarrow S_2$. Las producciones de G_1 y G_2 se mantienen. Concretamente,

$$G = (V_1 \cup V_2 \cup \{S\}, \Sigma, S, P_1 \cup P_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}).$$

Esquemáticamente, G tiene el siguiente aspecto:

$$\begin{array}{l} S \rightarrow S_1 \mid S_2 \\ S_1 \rightarrow \cdots \\ \vdots \qquad \vdots \\ S_2 \rightarrow \cdots \\ \vdots \qquad \vdots \end{array} \left. \begin{array}{l} \text{producciones de } G_1 \\ \text{producciones de } G_2 \end{array} \right\}$$

Claramente, $L(G) = L_1 \cup L_2$.

Una GIC G que genere $L_1 L_2$ se construye similarmente, añadiendo la producción $S \rightarrow S_1 S_2$. Es decir,

$$G = (V_1 \cup V_2 \cup \{S\}, \Sigma, S, P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}).$$

Esquemáticamente, G es la gramática:

$$\begin{array}{l} S \rightarrow S_1 S_2 \\ S_1 \rightarrow \cdots \\ \vdots \qquad \vdots \\ S_2 \rightarrow \cdots \\ \vdots \qquad \vdots \end{array} \left. \begin{array}{l} \text{producciones de } G_1 \\ \text{producciones de } G_2 \end{array} \right\}$$

Claramente, $L(G) = L_1 L_2$.

Para generar L_1^* se define G como

$$G = (V_1 \cup \{S\}, \Sigma, S, P_1 \cup \{S \rightarrow S_1 S, S \rightarrow \lambda\}).$$

Esquemáticamente, G es la gramática:

$$\begin{array}{l} S \rightarrow S_1 S \mid \lambda \\ S_1 \rightarrow \cdots \\ \vdots \qquad \vdots \end{array} \left. \begin{array}{l} \text{producciones de } G_1 \end{array} \right\}$$

Ejemplo Utilizar las construcciones del Teorema 4.14.1 para encontrar una GIC que genere el lenguaje $L_1L_2^*$ donde $L_1 = ba^+$ y $L_2 = \{a^i b^j a^i : i \geq 0, j \geq 1\}$.

Solución. El lenguaje L_1 se puede generar con la gramática

$$\begin{cases} S_1 \rightarrow bA \\ A \rightarrow aA \mid a \end{cases}$$

y L_2 con

$$\begin{cases} S_2 \rightarrow aS_2a \mid bB \\ B \rightarrow bB \mid \lambda \end{cases}$$

La siguiente gramática genera L_2^* :

$$\begin{cases} S_3 \rightarrow S_2S_3 \mid \lambda \\ S_2 \rightarrow aS_2a \mid bB \\ B \rightarrow bB \mid \lambda \end{cases}$$

Finalmente, el lenguaje $L_1L_2^*$ se puede generar con

$$\begin{cases} S \rightarrow S_1S_3 \\ S_1 \rightarrow bA \\ A \rightarrow aA \mid a \\ S_3 \rightarrow S_2S_3 \mid \lambda \\ S_2 \rightarrow aS_2a \mid bB \\ B \rightarrow bB \mid \lambda. \end{cases}$$

4.14.2 Teorema. *La colección de los lenguajes independientes del contexto no es cerrada (en general) para las siguientes operaciones:*

(1) *Intersección.*

(2) *Complemento.*

(3) *Diferencia.*

Demostración.

(1) La intersección de dos LIC puede ser un lenguaje que no es LIC. Considerérense, como ejemplo, los lenguajes

$$\begin{aligned} L_1 &= \{a^i b^i c^j : i, j \geq 0\}, \\ L_2 &= \{a^i b^j c^j : i, j \geq 0\}. \end{aligned}$$

Tanto L_1 como L_2 son LIC porque son generados por las gramáticas G_1 y G_2 , respectivamente:

$$G_1 : \begin{cases} S \rightarrow AB \\ A \rightarrow aAb \mid \lambda \\ B \rightarrow cB \mid \lambda \end{cases} \quad G_2 : \begin{cases} S \rightarrow AB \\ A \rightarrow aA \mid \\ B \rightarrow bBc \mid \lambda \end{cases}$$

Pero $L_1 \cap L_2 = \{a^i b^i c^i : i \geq 0\}$ no es un LIC, según se mostró, usando el lema de bombeo, en la sección 4.13.

- (2) Razonamos por contradicción: si el complemento de todo LIC fuera un LIC se podría concluir que la intersección de dos LIC L_1 y L_2 sería un LIC ya que $L_1 \cap L_2 = \overline{L_1} \cup \overline{L_2}$. Esto estaría en contradicción con la parte (1) del presente teorema.
- (3) Razonamos por contradicción: si la diferencia de dos LIC cualesquiera fuera un LIC se podría concluir que el complemento de un LIC L sería también un LIC ya que $\overline{L} = \Sigma^* - L$. Esto estaría en contradicción con la parte (2) del presente teorema. \square

El siguiente teorema afirma que los LIC también son cerrados bajo homomorfismos.

4.14.3 Teorema. *Sea $h : \Sigma^* \rightarrow \Gamma^*$ un homomorfismo. Si L es un LIC sobre Σ , entonces $h(L)$ es un LIC sobre Γ .*

Demostración. La demostración consiste en transformar una gramática G que genere el lenguaje L en una gramática G' que genere $h(L)$. Para ello basta mantener las mismas variables de G y definir las producciones de G' , a partir de las de G , cambiando cada terminal a por $h(a)$. Es fácil ver que una derivación $S \xrightarrow{+} w$ en G , con $w \in \Sigma^*$, se puede transformar en una derivación $S \xrightarrow{+} h(w)$ en G' ; esto muestra $h(L) \subseteq L(G')$.

Para establecer la otra contenencia, es decir, $L(G') \subseteq h(L)$, hay que demostrar que si $S \xrightarrow{+}_{G'} z$, con $z \in \Gamma^*$, entonces z es de la forma $z = h(w)$ para algún $w \in L$. Esto puede hacerse considerando el árbol de la derivación $S \xrightarrow{+}_{G'} z$. Dicho árbol se puede transformar en un árbol de una derivación en G , modificando adecuadamente las hojas: las hojas del nuevo árbol forman una cadena $w \in \Sigma^*$ y las hojas del árbol inicial forman la cadena $h(w)$. \square

Ejemplo Utilizar homomorfismos para concluir que el lenguaje $L = \{0^i 1^i 2^i 3^i : i \geq 0\}$, sobre el alfabeto $\{0, 1, 2, 3\}$ no es LIC.

Solución. La idea es “convertir” L en el lenguaje $\{a^i b^i c^i : i \geq 0\}$, que no es LIC, según se mostró en la sección 4.13. Razonamos de la siguiente manera: si L fuera un LIC, lo sería también $h(L)$, donde h es el homomorfismo $h : \{0, 1, 2, 3\}^* \rightarrow \{a, b, c\}^*$ definido por $h(0) = a$, $h(1) = b$, $h(2) = c$ y $h(3) = \lambda$. Pero

$$h(L) = \{h(0)^i h(1)^i h(2)^i h(3)^i : i \geq 0\} = \{a^i b^i c^i : i \geq 0\}.$$

Por consiguiente, L no es un LIC.

Ejercicios de la sección 4.14

- ① Utilizar las construcciones del Teorema 4.14.1 para encontrar GIC que generen los siguientes lenguajes:
 - (i) $a^+ (a \cup bab)^* (b^* \cup a^* b)$.
 - (ii) $(L_1 \cup L_2)L_3^*$, donde $L_1 = ab^*a$, $L_2 = a \cup b^+$ y $L_3 = \{a^i ba^i : i \geq 0\}$.
 - (iii) $L_1 \cup L_2^* L_3$, donde $L_1 = ab^*a$, $L_2 = \{a^i b^j c^j d^i : i, j \geq 1\}$ y $L_3 = b^+$.
- ② Sea $G = (V, \Sigma, S, P)$ una gramática que genera al lenguaje L . ¿La gramática $G = (V, \Sigma, S, P \cup \{S \rightarrow SS, S \rightarrow \lambda\})$ genera a L^* ?
- ③ (i) Mostrar que los dos lenguajes siguientes, sobre $\Sigma = \{a, b, c, d\}$, son LIC:

$$L_1 = \{a^i b^i c^j d^j : i, j \geq 1\},$$

$$L_2 = \{a^i b^j c^j d^k : i, j, k \geq 1\}.$$
 (ii) Demostrar que $L_1 \cap L_2$ no es un LIC.
- ④ Utilizar homomorfismos para concluir que los siguientes lenguajes sobre el alfabeto $\{0, 1\}$ no son LIC:
 - (i) $L = \{u : |u| \text{ es un número primo}\}$.
 - (ii) $L = \{u : |u| \text{ es un cuadrado perfecto}\}$.
- ⑤ Demostrar que los LIC son cerrados para la operación de reflexión. Concretamente, demostrar que si L es un LIC, también lo es el lenguaje $L^R = \{w^R : w \in L\}$.

4.15. Algoritmos de decisión para GIC

En esta sección consideraremos problemas de decisión para GIC, similares a los problemas para autómatas presentados en la sección 3.6. Dada una propiedad \mathcal{P} , referente a gramáticas independientes del contexto, un problema de decisión para \mathcal{P} consiste en buscar un algoritmo, aplicable a una GIC arbitraria G , que responda SI o NO a la pregunta: ¿satisface G la propiedad \mathcal{P} ? Los algoritmos vistos en el presente capítulo (para encontrar las variables terminables, las alcanzables, las anulables, etc) son frecuentemente útiles en el diseño de algoritmos de decisión más complejos.

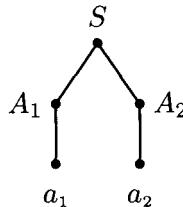
Problema 1 (Problema de la vacuidad). *Dada una gramática $G = (V, \Sigma, S, P)$, ¿es $L(G) \neq \emptyset$?*

Algoritmo de decisión: ejecutar el algoritmo para determinar el conjunto **TERM** de variables terminables. $L(G) \neq \emptyset$ si y sólo si $S \in \text{TERM}$.

Problema 2 (Problema de la pertenencia). *Dada una GIC $G = (V, \Sigma, S, P)$ y una cadena $w \in \Sigma^*$, ¿se tiene $w \in L(G)$?*

Para resolver este problema primero convertimos G a la forma FNC, con variable inicial no recursiva, siguiendo el procedimiento de la sección 4.11.

A partir de una GIC G en FNC podemos diseñar un algoritmo bastante ineficiente para decidir si $w \in L(G)$: se encuentran *todas* las posibles derivaciones a izquierda (o los árboles de derivación) que generen cadenas de longitud $n = |w|$. Más específicamente, las cadenas de longitud 1 se pueden derivar únicamente con producciones de la forma $S \rightarrow a$. Las cadenas de longitud 2 sólo tienen árboles de derivación de la forma:



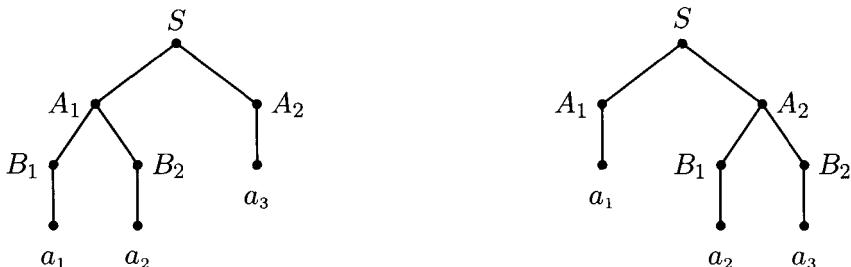
en los que aparecen exactamente 3 variables. Para derivar cadenas de longitud 3 sólo se puede proceder de dos formas:

$$S \Rightarrow A_1 A_2 \Rightarrow B_1 B_1 A_2 \xrightarrow{3} a_1 a_2 a_3,$$

6

$$S \Rightarrow A_1 A_2 \Rightarrow a_1 A_2 \Rightarrow a_1 B_1 B_2 \xrightarrow{2} a_1 a_2 a_3.$$

Los árboles de estas derivaciones son:



Cada uno de estos árboles tiene exactamente 5 nodos etiquetados con variables. La situación general es la siguiente: un árbol de derivación de una cadena de longitud n tiene exactamente $2n - 1$ nodos etiquetados con variables. Puesto que la raíz del árbol es S y S no es recursiva, en un árbol de derivación de una cadena de longitud n hay exactamente $2n - 2$ nodos interiores etiquetados con variables. La demostración general puede hacerse por inducción sobre n y la dejamos como ejercicio para el estudiante.

Por consiguiente, para determinar si una cadena dada de longitud n es o no generada por G , consideraremos todos los posibles árboles de derivación con $2n - 2$ variables interiores. Este algoritmo es ineficiente porque si G tiene k variables, hay que chequear no menos de k^{2n-2} árboles (esto sin contar las posibilidades para las hojas). Por consiguiente, el procedimiento tiene complejidad exponencial con respecto al tamaño de la entrada.

Para resolver el problema de la pertenencia hay un algoritmo muy eficiente (su complejidad es polinomial) en el que se usa la llamada *programación dinámica* o *tabulación dinámica*, técnica para llenar tablas progresivamente, re-utilizando información previamente obtenida. El algoritmo que presentaremos se denomina *algoritmo CYK* (nombre que corresponde a las iniciales de los investigadores Cocke, Younger y Kasami). El algoritmo (exhibido en la página siguiente) tiene como entrada una GIC G en FNC y una cadena de n terminales $w = a_1 a_2 \cdots a_n$; se aplica llenando una tabla de n filas (una por cada terminal de la entrada w) y n columnas. X_{ij} es el conjunto de variables de las que se puede derivar la subcadena de w cuyo primer símbolo está en la posición i y cuya longitud es j . O sea,

$$X_{ij} = \text{conjunto de variables } A \text{ tales que } A \xrightarrow{+} a_i a_{i+1} \cdots a_{i+j-1}.$$

Al determinar los conjuntos X_{ij} se obtienen las posibles maneras de derivar subcadenas de w que permitan construir una derivación de la cadena completa w . La tabla se llena por columnas, de arriba hacia abajo; la primera columna ($j = 1$) corresponde a las subcadenas de longitud 1, la segunda columna ($j = 2$) corresponde a las subcadenas de longitud 2, y así sucesivamente. La última columna ($j = n$) corresponde a la única subcadena de

longitud n que tiene w , que es la propia cadena w . Se tendrá que $w \in L(G)$ si y sólo si $S \in X_{1n}$.

Algoritmo CYK

ENTRADA:

Gramática G en FNC y cadena de n terminales $w = a_1a_2 \cdots a_n$.

INICIALIZAR:

$j = 1$. Para cada i , $1 \leq i \leq n$,

$X_{ij} = X_{i1} :=$ conjunto de variables A tales que $A \rightarrow a_i$ es una producción de G .

REPETIR:

$j := j + 1$. Para cada i , $1 \leq i \leq n - j + 1$,

$X_{ij} :=$ conjunto de variables A tales que $A \rightarrow BC$ es una producción de G , con $B \in X_{ik}$ y $C \in X_{i+k,j-k}$, considerando todos los k tales que $1 \leq k < j - 1$.

HASTA: $j = n$.

SALIDA: $w \in L(G)$ si y sólo si $S \in X_{1n}$.

Ejemplo

Vamos a aplicar el algoritmo CYK a la gramática:

$$G : \begin{cases} S \rightarrow BA \mid AC \\ A \rightarrow CC \mid b \\ B \rightarrow AB \mid a \\ C \rightarrow BA \mid a \end{cases}$$

y a la cadena $w = bbab$. Se trata de determinar si $w \in L(G)$ o no. La tabla obtenida al hallar los X_{ij} , $1 \leq i \leq 4$, es la siguiente:

		$j = 1$	$j = 2$	$j = 3$	$j = 4$
b	$i = 1$	$\{A\}$	—	$\{B\}$	$\{S, C\}$
b	$i = 2$	$\{A\}$	$\{B, S\}$	$\{S, C\}$	
a	$i = 3$	$\{B, C\}$	$\{S, C\}$		
b	$i = 1$	$\{A\}$			

A continuación se indica de manera detallada cómo se obtuvo la tabla anterior, columna por columna:

- $j = 1$. Se obtiene directamente de las producciones de G .
- $j = 2$. Para X_{12} se buscan cuerpos de producciones en $X_{11}X_{21} = \{A\}\{A\} = \{A\}$. Así que $X_{12} = \{ \}$.
 Para X_{22} se buscan cuerpos de producciones en $X_{21}X_{31} = \{A\}\{B, C\} = \{AB, AC\}$. Así que $X_{22} = \{B, S\}$.
 Para X_{23} se buscan cuerpos de producciones en $X_{31}X_{41} = \{B, C\}\{A\} = \{BA, CA\}$. Así que $X_{23} = \{S, C\}$.
- $j = 3$. Para X_{13} se buscan cuerpos de producciones en $X_{11}X_{22} \cup X_{12}X_{31} = \{A\}\{B, S\} \cup \{ \} = \{AB, AS\}$. Así que $X_{13} = \{B\}$.
 Para X_{23} se buscan cuerpos de producciones en $X_{21}X_{32} \cup X_{22}X_{41} = \{A\}\{S, C\} \cup \{B, S\}\{A\} = \{AS, AC\} \cup \{BA, SA\}$. Así que $X_{23} = \{S, C\}$.
- $j = 4$. Para X_{14} se buscan cuerpos de producciones en $X_{11}X_{23} \cup X_{12}X_{32} \cup X_{13}X_{41} = \{A\}\{S, C\} \cup \{ \} \cup \{B\}\{A\} = \{AS, AC\} \cup \{BA\}$. Así que $X_{14} = \{S, C\}$.

Puesto que la variable S pertenece al conjunto X_{14} , se concluye que la cadena $w = bbab$ es generada por G .

Consideremos ahora la entrada $w = baaba$, de longitud 5. Al hallar los X_{ij} , $1 \leq i \leq 5$, se obtiene la tabla siguiente. Como $S \in X_{15}$, se concluye que $w \in L(G)$.

		$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$
b	$i = 1$	$\{A\}$	$\{B, S\}$	—	—	$\{S, B, C\}$
a	$i = 2$	$\{B, C\}$	$\{A\}$	$\{A\}$	$\{S, B, C\}$	
a	$i = 3$	$\{B, C\}$	$\{S, C\}$	$\{A\}$		
b	$i = 4$	$\{A\}$	$\{B, S\}$			
a	$i = 5$	$\{B, C\}$				

Al procesar la entrada $w = aaba$ se obtiene la tabla siguiente. Como S no pertenece al conjunto X_{14} , se deduce que w no es generada por G .

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
$i = 1$	$\{B, C\}$	$\{A\}$	$\{B\}$	—
$i = 2$	$\{B, C\}$	—	—	
$i = 3$	$\{B\}$	—		
$i = 4$	$\{B, C\}$			

Problema 3 (Problema de la infinitud). Dada una gramática $G = (V, \Sigma, S, P)$, ¿es $L(G)$ infinito?

El lema de bombeo sirve para establecer un criterio que permite resolver este problema (de manera análoga a lo que sucede en el caso de los lenguajes regulares). El criterio aparece en el siguiente teorema.

4.15.1 Teorema. Sea $G = (V, \Sigma, S, P)$ una gramática en FNC, con variable inicial no recursiva, tal que $L(G) = L$, y sea $k = |V| =$ número de variables de G . El lenguaje L es infinito si y sólo si contiene una cadena z tal que $2^k < |z| \leq 2^{k+1}$.

Demostración. Si $z \in L$ y $2^k < |z| \leq 2^{k+1}$, entonces por la demostración del lema de bombeo, z se puede descomponer como $z = uvwxy$, donde $|vwx| \leq 2^k$, $v \neq \lambda$. L posee infinitas cadenas: $uv^iwx^i y$ para todo $i \geq 0$.

Recíprocamente, si L es infinito, existe $z \in L$ con $|z| \geq 2^k$. Por la demostración del lema de bombeo, $w = uvwxy$ donde $|vwx| \leq 2^k$; además, $v \neq \lambda$ ó $x \neq \lambda$. Si $|z| \leq 2^{k+1}$, la demostración termina. Si $|z| > 2^{k+1} = 2^k + 2^k$, puesto que $|z| = |uy| + |vwx|$, se tendrá

$$|uwy| \geq |uy| = |z| - |vwx| \geq |z| - 2^k > 2^k.$$

De nuevo, si $|uwy| < 2^{k+1}$, la demostración termina; en caso contrario, se prosigue de esta forma hasta encontrar una cadena en L cuya longitud ℓ satisfaga $2^k < \ell \leq 2^{k+1}$. \square

Utilizando el Teorema 4.15.1 podemos ahora presentar un algoritmo de decisión para el problema de la infinitud:

1. Convertir la gramática G dada a una gramática equivalente G' en Forma Normal de Chomsky.
2. Aplicar el algoritmo CYK a G' , con cada una de las cadenas $|z|$ cuya longitud ℓ satisfaga $2^k < \ell \leq 2^{k+1}$, siendo k el número de variables de G' . L es infinito si y sólo si alguna de las cadenas examinadas está en $L(G')$.

Obsérvese que este algoritmo tiene de complejidad exponencial ya que el número de cadenas z tales que $2^k < |z| \leq 2^{k+1}$ es m^{2^k} , donde m es el número de terminales en la gramática dada.

-  Hay muchos problemas referentes a gramáticas que son *indecidibles*, es decir, problemas para los cuales no existen algoritmos de decisión. Aunque no tenemos las herramientas para demostrarlo, los siguientes problemas son indecidibles:

1. Dada una gramática G , ¿es G ambigua?
2. Dada una gramática G , ¿genera G todas las cadenas de terminales?, es decir, $\{L(G) = \Sigma^*\}$?
3. Dadas dos gramáticas G_1 y G_2 , ¿generan G_1 y G_2 el mismo lenguaje?, es decir, $\{L(G_1) = L(G_2)\}$?

Ejercicios de la sección 4.15

- ① Sea $G = (V, \Sigma, S, P)$ una gramática dada. Encontrar algoritmos para los siguientes problemas de decisión:
 - (i) ¿Genera G la cadena vacía λ ?
 - (ii) ¿Hay dos variables diferentes A y B tales que $A \xrightarrow{*} B$?
 - (iii) ¿Hay en $L(G)$ alguna cadena de longitud 1250? Ayuda: hay un número finito de cadenas con longitud 1250.
 - (iv) ¿Hay en $L(G)$ alguna cadena de longitud mayor que 1250? Ayuda: habría un número infinito de cadenas por examinar; usar el Teorema 4.15.1.
 - (v) ¿Hay en $L(G)$ por lo menos 1250 cadenas? Ayuda: usar la misma idea del problema (iv).
- ② Encontrar un algoritmo para el siguiente problema de decisión: dada una gramática $G = (V, \Sigma, S, P)$ y una variable $A \in V$, ¿es A recursiva?, es decir, ¿existe una derivación de la forma $A \xrightarrow{+} uAv$, con $u, v \in (V \cup \Sigma)^*$?
- ③ Encontrar un algoritmo para el siguiente problema de decisión: dado un lenguaje finito L y una GIC G , ¿se tiene $L \subseteq L(G)$?

④ Sea G la gramática

$$G : \begin{cases} S \rightarrow BA \mid AB \\ A \rightarrow CA \mid a \\ B \rightarrow BB \mid b \\ C \rightarrow BA \mid c \end{cases}$$

Ejecutar el algoritmo CYK para determinar si las siguientes cadenas w son o no generadas por G :

- | | |
|------------------|-------------------|
| (i) $w = bca.$ | (iii) $w = cabb.$ |
| (ii) $w = acbc.$ | (iv) $w = bbbaa.$ |

Capítulo 5

Autómatas con pila

En el presente capítulo presentamos el modelo de autómata requerido para aceptar los lenguajes independientes del contexto: el autómata con pila no-determinista. Existe también la versión determinista pero, a diferencia de lo que sucede con los modelos AFD y AFN, los autómatas con pila deterministas y no-deterministas no resultan ser computacionalmente equivalentes.

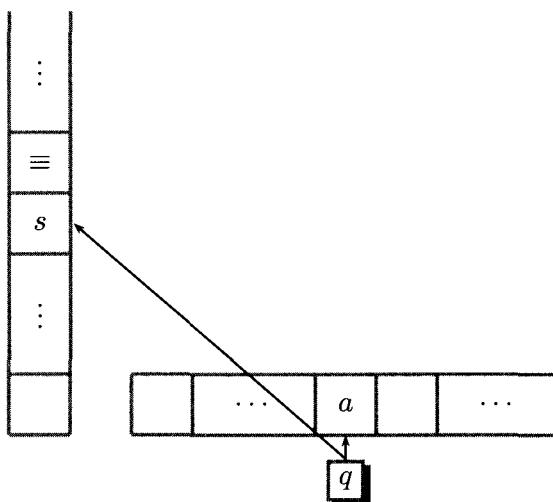
5.1. Autómatas con Pila Deterministas (AFPD)

Un **Autómata Finito con Pila Determinista** (AFPD) es una 7-upla, $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$, con los siguientes componentes:

1. Q es el conjunto (finito) de estados internos de la unidad de control.
2. $q_0 \in Q$ es el estado inicial.
3. F es el conjunto de estados finales o de aceptación, $\emptyset \neq F \subseteq Q$.
4. Σ es el alfabeto de entrada, también llamado alfabeto de cinta.
5. Γ es el alfabeto de pila.
6. $z_0 \in \Gamma$ es el marcador de fondo, también llamado símbolo inicial de pila (z_0 no pertenece al alfabeto de entrada Σ).
7. Δ es la función de transición del autómata:

$$\Delta : Q \times (\Sigma \cup \lambda) \times \Gamma \rightarrow (Q \times \Gamma^*).$$

Como en los modelos ya considerados (AFD, AFN y AFN- λ), un AFPD procesa cadenas sobre una cinta de entrada semi-infinita, pero hay una cinta adicional, llamada pila, que es utilizada por el autómata como lugar de almacenamiento. En un momento determinado, la unidad de control del autómata escanea un símbolo a sobre la cinta de entrada y el símbolo s en el tope o cima de la pila, como lo muestra la siguiente gráfica:



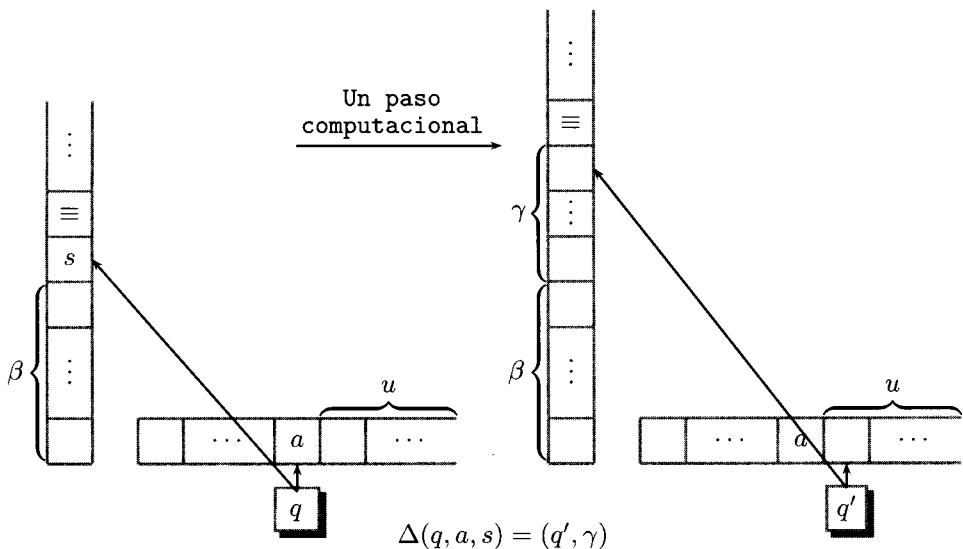
La transición

$$\Delta(q, a, s) = (q', \gamma)$$

representa un **paso computacional**: la unidad de control pasa al estado q' y se mueve a la derecha; además, borra el símbolo s que está en el tope de la pila, escribe la cadena γ (cadena que pertenece a Γ^*) y pasa a escanear el nuevo tope de la pila. La gráfica que aparece en la parte superior de la página siguiente ilustra un paso computacional. Recalcamos que en cada momento, el autómata sólo tiene acceso al símbolo que está en el tope de la pila; además, el contenido de la pila siempre se lee desde arriba (el tope) hacia abajo. Por estas dos razones la pila se dibuja verticalmente.

Casos especiales de transiciones:

1. $\Delta(q, a, s) = (q', s)$. En este caso, el contenido de la pila no se altera.
2. $\Delta(q, a, s) = (q', \lambda)$. El símbolo s en el tope de la pila se borra y la unidad de control pasa a escanear el nuevo tope de la pila, que es el símbolo colocado inmediatamente debajo de s .



3. $\Delta(q, \lambda, s) = (q', \gamma)$. Ésta es una transición λ o transición espontánea: el símbolo sobre la cinta de entrada no se procesa y la unidad de control no se mueve a la derecha, pero el tope s de la pila es reemplazado por la cadena γ . Para garantizar el determinismo, $\Delta(q, a, s)$ y $\Delta(q, \lambda, s)$, con $a \in \Sigma$, no pueden estar simultáneamente definidos (de lo contrario el autómata tendría una opción no-determinista). Las transiciones espontáneas en un AFPD permiten que el autómata cambie el contenido de la pila sin procesar (o consumir) símbolos sobre la cinta de entrada.

Configuración o descripción instantánea. Es una tripla $(q, au, s\beta)$ que representa lo siguiente: el autómata está en el estado q , au es la parte no procesada de la cadena de entrada y la unidad de control está escaneando el símbolo a . La cadena $s\beta$ es el contenido *total* de la pila; siendo s el símbolo colocado en el tope.

La notación $(q, au, s\beta)$ para configuraciones instantáneas es muy cómoda: para representar el paso computacional de la figura que aparece arriba escribimos simplemente

$$(q, au, s\beta) \vdash (q', u, \gamma\beta).$$

Aquí el autómata utilizó la transición $\Delta(q, a, s) = (q', \gamma)$.

La notación

$$(q, u, \beta) \stackrel{*}{\vdash} (p, v, \gamma)$$

significa que el autómata pasa de la configuración instantánea (q, u, β) a la configuración instantánea (p, v, γ) en cero, uno o más pasos computacionales.

Configuración inicial. Para una cadena de entrada $w \in \Sigma^*$, la configuración inicial es (q_0, w, z_0) . Al comenzar el procesamiento de toda cadena de entrada, el contenido de la pila es z_0 , que sirve como marcador de fondo.

Configuración de aceptación. La configuración (p, λ, β) , siendo p un estado final o de aceptación, se llama configuración de aceptación. Esto significa que, para ser aceptada, una cadena de entrada debe ser procesada completamente y la unidad de control debe terminar en un estado de aceptación. La cadena β que queda en la pila puede ser cualquier cadena perteneciente a Γ^* .

Lenguaje aceptado por un AFPD. El lenguaje aceptado por un AFPD M se define como

$$L(M) := \{w \in \Sigma^* : (q_0, w, z_0) \xrightarrow{*} (p, \lambda, \beta), p \in F\}.$$

O sea, una cadena es aceptada si se puede ir desde la configuración inicial hasta una configuración de aceptación, en cero, uno o más pasos.

- ☞ En el modelo AFPD se permite que la transición $\Delta(q, a, s)$ no esté definida, para algunos valores $q \in Q$, $a \in \Sigma$, $s \in \Gamma$. Esto implica que el cómputo de algunas cadenas de entrada puede abortarse sin que se procesen completamente.
- ☞ No se debe confundir la tripla que aparece en la función de transición $\Delta(q, a, s)$ con la tripla (q, u, β) que representa una configuración instantánea.
- ☞ La definición de la función de transición Δ requiere que haya por lo menos un símbolo en la pila. No hay cómputos con pila vacía.
- ☞ Para los autómatas con pila se pueden hacer diagramas de transiciones, similares a los ya conocidos, pero resultan de poca utilidad práctica ya que el procesamiento completo de una cadena de entrada depende del contenido de la pila, el cual puede cambiar en cada paso computacional.
- ☞ Los analizadores sintácticos en compiladores se comportan generalmente como autómatas con pila deterministas.

Un AFPD puede simular un AFD simplemente ignorando la pila; de esto se deduce que los lenguajes regulares son aceptados por autómatas AFPD. El siguiente teorema establece formalmente este resultado.

5.1.1 Teorema. *Todo lenguaje regular L es aceptado por algún AFPD.*

Demostración. Sea $M = (Q, q_0, F, \Sigma, \delta)$ un AFD que acepta a L . El AFPD $M' = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$ definido haciendo $\Gamma = \{z_0\}$ y

$$\Delta(q, a, z_0) = (\delta(q, a), z_0), \text{ para todo } a \in \Sigma, q \in Q,$$

satisface claramente $L(M') = L(M) = L$. \square

Sin usar la pila un AFPD no puede hacer nada más que un AFD, pero utilizando la pila como lugar de almacenamiento, un AFPD puede aceptar lenguajes no regulares, como se muestra en el siguiente ejemplo.

Ejemplo Diseñar un AFPD que acepte el lenguaje $L = \{a^i b^i : i \geq 1\}$, sobre el alfabeto $\Sigma = \{a, b\}$. Recordemos que L no es regular y no puede ser aceptado por ningún autómata normal (sin pila).

Solución. La idea es copiar las a es en la pila y borrar luego una a por cada b que sea leída sobre la cinta. Una cadena será aceptada si es procesada completamente y en la pila sólo queda el marcador de fondo z_0 . Concretamente, $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$, donde

$$\begin{aligned}\Sigma &= \{a, b\}, \\ \Gamma &= \{z_0, A, B\}, \\ Q &= \{q_0, q_1, q_2\}, \\ F &= \{q_2\},\end{aligned}$$

y la función de transición está dada por:

$$\begin{aligned}\Delta(q_0, a, z_0) &= (q_0, Az_0), \\ \Delta(q_0, a, A) &= (q_0, AA), \\ \Delta(q_0, b, A) &= (q_1, \lambda), \\ \Delta(q_1, b, A) &= (q_1, \lambda), \\ \Delta(q_1, \lambda, z_0) &= (q_2, z_0).\end{aligned}$$

Podemos ilustrar el procesamiento de varias cadenas de entrada. Sea, inicialmente, $u = aaabbb$.

$$\begin{aligned}(q_0, aaabbb, z_0) &\vdash (q_0, aabb, Az_0) \vdash (q_0, abbb, AAz_0) \vdash (q_0, bbb, AAAz_0) \\ &\vdash (q_1, bb, AAz_0) \vdash (q_1, b, Az_0) \vdash (q_1, \lambda, z_0) \vdash (q_2, \lambda, z_0).\end{aligned}$$

La última es una configuración de aceptación; por lo tanto la cadena $u = aaabbb$ es aceptada.

Para la cadena de entrada $v = aabbb$, se obtiene el siguiente procesamiento:

$$\begin{aligned} (q_0, aabbb, z_0) &\vdash (q_0, abbb, Az_0) \vdash (q_0, bbb, AAz_0) \vdash (q_1, bb, Az_0) \\ &\vdash (q_1, b, z_0) \vdash (q_2, b, z_0). \quad [\text{cómputo abortado}] \end{aligned}$$

Obsérvese que el autómata ha ingresado al estado de aceptación q_2 pero la cadena de entrada no es aceptada debido a que no se ha procesado completamente; (q_2, b, z_0) no es una configuración de aceptación.

Para la cadena de entrada $w = aaabb$, se tiene:

$$\begin{aligned} (q_0, aaabb, z_0) &\vdash (q_0, aabb, Az_0) \vdash (q_0, abb, AAz_0) \vdash (q_0, bb, AAAz_0) \\ &\vdash (q_1, b, AAz_0) \vdash (q_1, \lambda, Az_0). \end{aligned}$$

A pesar de que se ha procesado completamente la cadena de entrada w , la configuración (q_1, λ, Az_0) no es de aceptación. Por lo tanto, $w = aaabb$ no es aceptada.

Ejemplo Diseñar un AFPD que acepte el lenguaje

$$L = \{wcw^R : w \in \{a, b\}^*\}.$$

sobre el alfabeto $\Sigma = \{a, b, c\}$. Nótese que las cadenas w y w^R sólo poseen aes y/o bes.

Solución. La idea es acumular los símbolos en la pila hasta que aparezca la c . Luego se comparan los símbolos leídos con los almacenados en la pila, borrando en cada paso el tope de la pila. La cadena de entrada será aceptada si es procesada completamente y en la pila sólo queda el marcador de fondo z_0 . En detalle, $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$, donde

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

$$\Gamma = \{z_0, A, B\},$$

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

$$F = \{q_2\},$$

y la función de transición está dada por:

$$\begin{aligned}\Delta(q_0, a, z_0) &= (q_0, Az_0), \\ \Delta(q_0, b, z_0) &= (q_0, Bz_0), \\ \Delta(q_0, c, z_0) &= (q_2, z_0) \quad (\text{para aceptar la cadena } c), \\ \Delta(q_0, a, A) &= (q_0, AA), \\ \Delta(q_0, a, B) &= (q_0, AB), \\ \Delta(q_0, b, A) &= (q_0, BA), \\ \Delta(q_0, b, B) &= (q_0, BB), \\ \Delta(q_0, c, A) &= (q_1, A), \\ \Delta(q_0, c, B) &= (q_1, B), \\ \Delta(q_1, a, A) &= (q_1, \lambda), \\ \Delta(q_1, b, B) &= (q_1, \lambda), \\ \Delta(q_1, \lambda, z_0) &= (q_2, z_0).\end{aligned}$$

Ejercicios de la sección 5.1

- ① Diseñar AFPD que acepten los siguientes lenguajes sobre $\Sigma = \{a, b\}$:
 - (i) $L = \{a^i b^{2i} : i \geq 1\}$.
 - (ii) $L = \{a^{2i} b^i : i \geq 1\}$.
- ② Diseñar AFPD que acepten los siguientes lenguajes sobre $\Sigma = \{0, 1\}$:
 - (i) $L = \{0^i 1^j 0^i : i, j \geq 1\}$.
 - (ii) $L = \{1^i 0^j 1^{i+j} : i, j \geq 1\}$.

5.2. Autómatas con pila no-deterministas (AFPN)

Un **Autómata Finito con Pila No-Determinista** (AFPN) consta de los mismos siete parámetros de un AFPD, $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$, pero la función de transición Δ está definida como:

$$\Delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow \wp_f(Q \times \Gamma^*),$$

donde $\wp_f(Q \times \Gamma^*)$ es el conjunto de subconjuntos finitos de $Q \times \Gamma^*$. Para $q \in Q$, $a \in \Sigma \cup \{\lambda\}$ y $s \in \Gamma$, $\Delta(q, a, s)$ es de la forma

$$\Delta(q, a, s) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_k, \gamma_k)\}.$$

El significado de esta transición es: al leer el símbolo a sobre la cinta de entrada, la unidad de control puede pasar (aleatoriamente) a uno de los estados p_i ($1 \leq i \leq k$) y se mueve a la derecha. Sobre la pila hace lo siguiente: borra el símbolo s que está en el tope y escribe la cadena γ_i (cadena que pertenece a Γ^*).

A diferencia de lo que sucede con los AFPD, en el modelo AFPN las transiciones $\lambda, \Delta(q, \lambda, s)$, no tienen restricción alguna.

El lenguaje aceptado por un AFPN M se define como:

$$L(M) := \{w \in \Sigma^* : \text{existe un cómputo } (q_0, w, z_0) \stackrel{*}{\vdash} (p, \lambda, \beta), p \in F\}.$$

O sea, una cadena w es aceptada si existe por lo menos un procesamiento de w desde la configuración inicial hasta una configuración de aceptación. La cadena β que queda en la pila puede ser cualquier cadena de Γ^* .

Ejemplo Diseñar un AFPN que acepte el lenguaje $\{a^i b^i : i \geq 0\}$, sobre el alfabeto $\Sigma = \{a, b\}$.

Solución. Para aceptar la cadena vacía se necesita una transición λ desde el estado inicial. El autómata presentado a continuación coincide con el autómata del primer ejemplo de la sección 5.1, excepto por dicha transición espontánea. $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$ donde

$$\begin{aligned}\Sigma &= \{a, b\}, \\ \Gamma &= \{z_0, A, B\}, \\ Q &= \{q_0, q_1, q_2\}, \\ F &= \{q_2\},\end{aligned}$$

y la función de transición es:

$$\begin{aligned}\Delta(q_0, \lambda, z_0) &= \{(q_2, z_0)\} \quad (\text{para aceptar } \lambda), \\ \Delta(q_0, a, z_0) &= \{(q_0, Az_0)\}, \\ \Delta(q_0, a, A) &= \{(q_0, AA)\}, \\ \Delta(q_0, b, A) &= \{(q_1, \lambda)\}, \\ \Delta(q_1, b, A) &= \{(q_1, \lambda)\}, \\ \Delta(q_1, \lambda, z_0) &= \{(q_2, z_0)\}.\end{aligned}$$

En este autómata el no-determinismo surge únicamente por la presencia simultánea de $\Delta(q_0, \lambda, z_0)$ y $\Delta(q_0, a, z_0)$.

Ejemplo Diseñar un AFPN que acepte el lenguaje de las cadenas sobre el alfabeto $\Sigma = \{a, b\}$ con igual número de *aes* que de *b*s.

Solución. La idea es acumular las *aes* o *b*s consecutivas en la pila. Si en el tope de la pila hay una *A* y el autómata lee una *b*, se borra la *A*; similarmente, si en el tope de la pila hay una *B* y el autómata lee una *a*, se borra la *B*. La cadena de entrada será aceptada si es procesada completamente y en la pila sólo queda el marcador de fondo z_0 . Sólo se requieren dos estados. Concretamente, $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$, donde

$$\begin{aligned}\Sigma &= \{a, b\}, \\ \Gamma &= \{z_0, A, B\}, \\ Q &= \{q_0, q_1\}, \\ F &= \{q_1\},\end{aligned}$$

y la función de transición está dada por:

$$\begin{aligned}\Delta(q_0, a, z_0) &= \{(q_0, Az_0)\}, \\ \Delta(q_0, b, z_0) &= \{(q_0, Bz_0)\}, \\ \Delta(q_0, a, A) &= \{(q_0, AA)\}, \\ \Delta(q_0, b, B) &= \{(q_0, BB)\}, \\ \Delta(q_0, a, B) &= \{(q_0, \lambda)\}, \\ \Delta(q_0, b, A) &= \{(q_0, \lambda)\}, \\ \Delta(q_0, \lambda, z_0) &= \{(q_1, z_0)\}.\end{aligned}$$

El no-determinismo se presenta únicamente por la presencia simultánea de $\Delta(q_0, a, z_0)$, $\Delta(q_0, b, z_0)$ y $\Delta(q_0, \lambda, z_0)$.

En contraste con lo que sucede con los modelos AFD y AFN, los modelos de autómata con pila determinista (AFPD) y no-determinista (AFPN) no resultan ser computacionalmente equivalentes: existen lenguajes aceptados por autómatas AFPN que no pueden ser aceptados por ningún AFD. Un ejemplo concreto es el lenguaje $L = \{ww^R : w \in \Sigma^*\}$. Como se mostrará a continuación, se puede construir un autómata con pila no-determinista para aceptar a L , pero no es posible diseñar ningún AFD que lo haga. La demostración de esta imposibilidad es bastante complicada y no la podemos presentar en el presente curso.

Ejemplo Diseñar un AFPN que acepte el lenguaje $L = \{ww^R : w \in \Sigma^*\}$, donde $\Sigma = \{a, b\}$. No es difícil ver que L es el lenguaje de los palíndromos de longitud par.

Solución. En el último ejemplo de la sección 5.1 se construyó un AFPD que acepta el lenguaje $\{wcw^R : w \in \{a, b\}^*\}$. El lenguaje L del presente ejemplo es similar, excepto que ya no aparece el separador c entre w y w^R . El no-determinismo se puede usar para permitirle al autómata la opción de “adivinar” cuál es la mitad de la cadena de entrada. Si acierta, procederá a comparar el resto de la cadena de entrada con los símbolos acumulados en la pila. Si no acierta, el autómata continuará acumulando símbolos en la pila y no llegará a un estado de aceptación. Si la cadena de entrada tiene la forma deseada, entre todos los cálculos posibles estará aquél en el que el autómata adivina correctamente cuándo ha llegado a la mitad de la cadena.

M se define como $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$ donde

$$\begin{aligned}\Sigma &= \{a, b\}, \\ \Gamma &= \{z_0, A, B\}, \\ Q &= \{q_0, q_1, q_2\}, \\ F &= \{q_2\},\end{aligned}$$

y la función de transición está dada por:

$$\begin{aligned}\Delta(q_0, a, z_0) &= \{(q_0, Az_0)\}, \\ \Delta(q_0, b, z_0) &= \{(q_0, Bz_0)\}, \\ \Delta(q_0, \lambda, z_0) &= \{(q_2, z_0)\} \quad (\text{para aceptar } \lambda), \\ \Delta(q_0, a, A) &= \{(q_0, AA), (q_1, \lambda)\}, \\ \Delta(q_0, a, B) &= \{(q_0, AB)\}, \\ \Delta(q_0, b, A) &= \{(q_0, BA)\}, \\ \Delta(q_0, b, B) &= \{(q_0, BB), (q_1, \lambda)\}, \\ \Delta(q_1, a, A) &= \{(q_1, \lambda)\}, \\ \Delta(q_1, b, B) &= \{(q_1, \lambda)\}, \\ \Delta(q_1, \lambda, z_0) &= \{(q_2, z_0)\}.\end{aligned}$$

Entre estas transiciones se destacan

$$\begin{aligned}\Delta(q_0, a, A) &= \{(q_0, AA), (q_1, \lambda)\}, \\ \Delta(q_0, b, B) &= \{(q_0, BB), (q_1, \lambda)\}\end{aligned}$$

las cuales le permiten al autómata una opción no-determinista: o seguir acumulando símbolos en la pila, en el estado q_0 , o suponer que se ha llegado a la mitad de la cadena de entrada. En este último caso, la unidad de control

pasa al estado q_1 y comienza a borrar los símbolos ya almacenados en la pila.

Ejercicios de la sección 5.2

① Diseñar APFN que acepten los siguientes lenguajes:

- (i) $L = \{0^i 1^j : i, j \geq 0, i \neq j\}$, sobre $\Sigma = \{0, 1\}$.
- (ii) $L = \{a^i b^j : i \geq j \geq 0\}$.
- !(iii) $L = \{a^{2i} b^{3i} : i \geq 0\}$, sobre $\Sigma = \{a, b\}$.
- !(iv) $L = \{0^i 1^j : 0 \leq i \leq j \leq 2i\}$, sobre $\Sigma = \{0, 1\}$.

② Como se demostró en la sección 4.13, el lenguaje $L = \{a^i b^i c^i : i \geq 0\}$ no es LIC y, por consiguiente, no puede ser aceptado por ningún autómata con pila. No obstante, podríamos concebir el siguiente plan para aceptar a L : acumular en la pila dos Aes por cada a leída en la cinta, borrar luego una A por cada b leída y, finalmente, borrar una A por cada c . Si la cadena de entrada es de la forma $a^i b^i c^i$, se llegará al marcador de fondo z_0 en el preciso momento en el que se consume completamente la entrada. Concretamente, M está definido como $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$ donde

$$\begin{aligned}\Sigma &= \{a, b, c\}, \\ \Gamma &= \{z_0, A\}, \\ Q &= \{q_0, q_1, q_2, q_3, q_4\}, \\ F &= \{q_4\},\end{aligned}$$

y la función de transición está dada por:

$$\begin{aligned}\Delta(q_0, \lambda, z_0) &= \{(q_4, z_0)\} \quad (\text{para aceptar } \lambda), \\ \Delta(q_0, a, z_0) &= \{(q_1, AAz_0)\}, \\ \Delta(q_1, a, A) &= \{(q_1, AAA)\}, \\ \Delta(q_1, b, A) &= \{(q_2, \lambda)\}, \\ \Delta(q_2, b, A) &= \{(q_2, \lambda)\}, \\ \Delta(q_2, c, A) &= \{(q_3, \lambda)\}, \\ \Delta(q_3, c, A) &= \{(q_3, \lambda)\}, \\ \Delta(q_3, \lambda, z_0) &= \{(q_4, z_0)\}.\end{aligned}$$

¿Acepta este autómata el lenguaje $\{a^i b^i c^i : i \geq 0\}$? En caso contrario, ¿qué lenguaje acepta?

5.3. Aceptación por pila vacía

En todos los modelos de autómatas que hemos considerado en este curso, la aceptación de cadenas está determinada por los estados finales o de aceptación. Para los autómatas con pila existe otra noción de aceptación: la aceptación por pila vacía, definida a continuación. Cuando se usa esta noción, los autómatas no requieren un conjunto F de estados finales, solamente los seis restantes componentes: Q , q_0 , Σ , Γ , z_0 y Δ .

5.3.1 Definición. Dado un autómata con pila $M = (Q, q_0, \Sigma, \Gamma, z_0, \Delta)$, ya sea AFPD o AFPN, el **lenguaje aceptado por M por pila vacía** se define como

$$N(M) := \{w \in \Sigma^* : (q_0, w, z_0) \stackrel{*}{\vdash} (p, \lambda, \lambda)\}.$$

O sea, una cadena es aceptada por pila vacía si se puede ir, en cero, uno o más pasos, desde la configuración inicial hasta una configuración en la que la pila esté completamente desocupada¹. Nótese que, para ser aceptada, la cadena de entrada w debe ser procesada completamente.

Para autómatas AFPN las nociones de aceptación por pila vacía y por estados finales resultan ser equivalentes, como se establece en los dos siguientes teoremas. Es importante anotar que para autómatas deterministas AFPD los dos tipos de aceptación *no* resultan ser equivalentes.

5.3.2 Teorema. Si $L = L(M)$ para algún autómata con pila AFPN M , entonces $L = N(M')$ para algún AFPN M' . Es decir, M' acepta por pila vacía lo que M acepta por estado final.

Demostración. Sea $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$. M' se diseña modificando M de tal manera que vacíe su pila cuando M haya aceptado una cadena de entrada. Concretamente, se define M' como

$$M' = (Q \cup \{p_0, p\}, p_0, \Sigma, \Gamma \cup \{r_0\}, r_0, \Delta')$$

donde p_0 (estado inicial) y p son estados nuevos, y r_0 es el nuevo marcador de fondo. La función de transición Δ' se define así:

1. $\Delta'(p_0, \lambda, r_0) = \{(q_0, z_0 r_0)\}$. Transición λ mediante la cual el nuevo símbolo inicial de pila se coloca en el fondo. Esto impedirá que una cadena sea accidentalmente aceptada si el autómata original M vacía la pila.

¹La N en la notación $N(M)$ proviene de la expresión ‘pila nula’, sinónimo de ‘pila vacía’.

2. $\Delta(q, a, s) \subseteq \Delta'(q, a, s)$ para todo $q \in Q$, $a \in \Sigma$ ó $a = \lambda$ y $s \in \Gamma$. Esto quiere decir que M' simula a M : todas las transiciones del autómata original también se pueden realizar en el nuevo autómata.
3. $(p, s) \in \Delta'(q, \lambda, s)$ para todo $q \in F$, $s \in \Gamma \cup \{r_0\}$. Mediante esta transición λ , M' pasa al nuevo estado p siempre que q sea un estado de aceptación de M .
4. $\Delta'(p, \lambda, s) = \{(p, \lambda)\}$. Mediante esta transición λ , M' borra todo el contenido de la pila.

Obsérvese que las transiciones λ de los numerales 3 y 4 no consumen ningún símbolo en la cadena de entrada. Además, la única manera de que M' vacíe completamente la pila es ingresando al estado p , lo cual puede hacer únicamente desde un estado de aceptación de M .

Si w es aceptada por M , o sea si $w \in L(M)$, M realiza un cómputo de la forma

$$(q_0, w, z_0) \xrightarrow{*} (q, \lambda, \beta)$$

donde $q \in F$ y $\beta \in \Gamma^*$. Entonces en M' se puede efectuar el siguiente cómputo:

$$(p_0, w, r_0) \vdash (q_0, w, z_0 r_0) \xrightarrow{*} (q, \lambda, \beta r_0) \vdash (p, \lambda, \beta r_0) \xrightarrow{*} (p, \lambda, \lambda).$$

Por lo tanto, $w \in N(M')$.

Un razonamiento similar muestra que $w \in N(M')$ implica $w \in L(M)$ (ejercicio para el estudiante). En conclusión, $L(M) = N(M')$. \square

5.3.3 Teorema. *Si $L = N(M)$ para algún autómata con pila AFPN M , entonces $L = L(M')$ para algún AFPN M' . Es decir, M' acepta por estado final lo que M acepta por pila vacía.*

Demostración. Sea $M = (Q, q_0, \Sigma, \Gamma, z_0, \Delta)$ un AFPN que acepta por pila vacía. M' se diseña añadiendo un nuevo estado q_f a M de tal manera que M' ingrese a tal estado (será el único estado de aceptación) solamente cuando M haya vaciado su pila. Concretamente, se define M' como

$$M' = (Q \cup \{p_0, p_f\}, p_0, \{p_f\}, \Sigma, \Gamma \cup \{r_0\}, r_0, \Delta')$$

donde p_0 (estado inicial) y p_f (único estado de aceptación) son estados nuevos, y r_0 es el nuevo símbolo inicial de pila. La función de transición Δ' se define así:

1. $\Delta'(p_0, \lambda, r_0) = \{(q_0, z_0 r_0)\}$. Transición λ mediante la cual el nuevo símbolo inicial de pila se coloca en el fondo. Cuando M' encuentre el marcador de fondo r_0 , sabrá que M ha vaciado su pila.
2. $\Delta(q, a, s) \subseteq \Delta'(q, a, s)$ para todo $q \in Q$, $a \in \Sigma$ ó $a = \lambda$ y $s \in \Gamma$. Esto quiere decir que M' simula a M : todas las transiciones del autómata original también se pueden realizar en el nuevo autómata.
3. $(p_f, s) \in \Delta'(q, \lambda, r_0)$ para todo $q \in Q$. Mediante esta transición λ , M' pasa al estado de aceptación p_f cuando detecte el marcador de fondo r_0 . O sea, M' acepta cuando M vacíe su pila.

Si w es aceptada por M , o sea si $w \in N(M)$, M realiza un cómputo de la forma

$$(q_0, w, z_0) \xrightarrow{*} (q, \lambda, \lambda)$$

donde $q \in Q$. Entonces en M' se puede efectuar el siguiente cómputo:

$$(p_0, w, r_0) \vdash (q_0, w, z_0 r_0) \xrightarrow{*} (q, \lambda, r_0) \vdash (p_f, \lambda, r_0).$$

Por lo tanto, $w \in L(M')$.

Un razonamiento similar muestra que $w \in L(M')$ implica $w \in N(M)$ (ejercicio para el estudiante). En conclusión, $N(M) = L(M')$. \square

Ejercicios de la sección 5.3

- ① Modificar los autómatas de los tres ejemplos de la sección 5.2 para que acepten por pila vacía y no por estado final.
- ② Diseñar AFPN que acepten por pila vacía los siguientes lenguajes:
 - (i) $L = \{a^i b^{2i} : i \geq 1\}$, sobre $\Sigma = \{a, b\}$.
 - (ii) $L = \{a^{2i} b^i : i \geq 1\}$, sobre $\Sigma = \{a, b\}$.
 - (iii) $L = \{0^i 1^j : i, j \geq 0, i \neq j\}$, sobre $\Sigma = \{0, 1\}$.
 - !(iv) $L = \{0^i 1^j : 0 \leq i \leq j \leq 2i\}$, sobre $\Sigma = \{0, 1\}$.

- !③ Completar los detalles faltantes en las demostraciones de los Teoremas 5.3.2 y 5.3.3. ¿Por qué estas demostraciones no son válidas para autómatas deterministas?

5.4. Autómatas con pila y LIC. Parte I.

Los lenguajes aceptados por los AFPN son exactamente los lenguajes independientes del contexto. Éste es un resultado análogo al Teorema de Kleene para lenguajes regulares, aunque en el caso de los autómatas con pila, los modelos deterministas no son computacionalmente equivalentes a los no-deterministas. En la presente sección consideraremos la primera parte de la correspondencia entre AFPN y LIC.

5.4.1 Teorema. *Dada una GIC G , existe un AFPN M tal que $L(G) = L(M)$.*

Bosquejo de la demostración. Para una gramática $G = (\Sigma, V, S, P)$ dada, se construye un AFPN que utiliza la pila para simular la derivación de cadenas realizada por G . M requiere solamente tres estados, independientemente del número de variables y producciones de G . Específicamente, el autómata M se define como $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$, donde $Q = \{q_0, q_1, q_2\}$, $F = \{q_2\}$ y $\Gamma = \Sigma \cup V \cup \{z_0\}$. La función de transición Δ se define de la siguiente manera:

1. $\Delta(q_0, \lambda, z_0) = \{(q_1, Sz_0)\}$. Transición λ mediante la cual M coloca el símbolo inicial de la gramática, S , en el tope de la pila al iniciar el procesamiento de una cadena de entrada.
2. Para cada variable $A \in V$,

$$\Delta(q_1, \lambda, A) = \{(q_1, u) : A \rightarrow u \text{ es una producción de la gramática } G\}.$$

Mediante estas transiciones, M utiliza la pila para simular las derivaciones: si el tope de la pila es A y en la derivación se usa la producción $A \rightarrow u$, el tope de la pila A es substituido por u .

3. Para cada símbolo terminal $a \in \Sigma$, $\Delta(q_1, a, a) = \{(q_1, \lambda)\}$. Mediante estas transiciones, M borra los terminales del tope de la pila al consumirlos sobre la cinta de entrada.
4. $\Delta(q_1, \lambda, z_0) = \{(q_2, z_0)\}$. M ingresa al estado de aceptación q_2 cuando detecta el marcador de fondo z_0 .

El autómata M está diseñado de tal forma que si $S \xrightarrow{*} w$ es una derivación a izquierda en la gramática G , entonces existe un procesamiento

$$(q_0, w, s_0) \vdash (q_0, w, Ss_0) \stackrel{*}{\vdash} (q_2, \lambda, z_0)$$

que simula la derivación.

Recíprocamente, puede demostrarse que si $(q_0, w, z_0) \stackrel{*}{\vdash} (q_2, \lambda, z_0)$ entonces $S \xrightarrow{*} w$ en la gramática G . \square

Ejemplo Sea G la gramática:

$$G : \begin{cases} S \rightarrow aAbS \mid bBa \mid \lambda \\ A \rightarrow aA \mid a \\ B \rightarrow bB \mid b \end{cases}$$

Según la construcción del Teorema 5.4.1, el autómata M está dado por $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$ donde

$$\begin{aligned} Q &= \{q_0, q_1, q_2\}, \\ F &= \{q_2\}, \\ \Gamma &= \{a, b, S, A, B, z_0\}, \end{aligned}$$

y la función de transición Δ es:

$$\begin{aligned} \Delta(q_0, \lambda, z_0) &= \{(q_1, Sz_0)\}, \\ \Delta(q_1, \lambda, S) &= \{(q_1, aAbS), (q_1, bBa), (q_1, \lambda)\}, \\ \Delta(q_1, \lambda, A) &= \{(q_1, aA), (q_1, a)\}, \\ \Delta(q_1, \lambda, B) &= \{(q_1, bB), (q_1, b)\}, \\ \Delta(q_1, a, a) &= \{(q_1, \lambda)\}, \\ \Delta(q_1, b, b) &= \{(q_1, \lambda)\}, \\ \Delta(q_1, \lambda, z_0) &= \{(q_2, z_0)\}. \end{aligned}$$

Podemos ilustrar la correspondencia entre derivaciones en G y procesamientos en M con la cadena $aabbba$, la cual tiene la siguiente derivación a izquierda:

$$S \xrightarrow{} aAbS \xrightarrow{} aabS \xrightarrow{} aabbBa \xrightarrow{} aabbba.$$

El autómata M simula esta derivación de la cadena $aabbba$ así:

$$\begin{aligned} (q_0, aabbba, z_0) &\vdash (q_1, aabbba, Sz_0) \vdash (q_1, aabbba, aAbSz_0) \\ &\vdash (q_1, abba, AbSz_0) \vdash (q_1, abba, abSz_0) \\ &\vdash (q_1, bba, bSz_0) \vdash (q_1, bba, Sz_0) \vdash (q_1, bba, bBaz_0) \\ &\vdash (q_1, ba, Baz_0) \vdash (q_1, ba, baz_0) \vdash (q_1, a, az_0) \\ &\vdash (q_1, \lambda, z_0) \vdash (q_2, \lambda, z_0). \end{aligned}$$

Ejemplo La siguiente gramática genera los palíndromos de longitud par, sobre $\Sigma = \{a, b\}$, es decir, el lenguaje $L = \{ww^R : w \in \Sigma^*\}$:

$$S \rightarrow aSa \mid bSb \mid \lambda.$$

Siguiendo el procedimiento del Teorema 5.4.1 podemos construir un AFPN que acepta a L . $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$ donde

$$\begin{aligned} Q &= \{q_0, q_1, q_2\}, \\ F &= \{q_2\}, \\ \Gamma &= \{a, b, S, z_0\}. \end{aligned}$$

La función de transición Δ está dada por:

$$\begin{aligned} \Delta(q_0, \lambda, z_0) &= \{(q_1, Sz_0)\}, \\ \Delta(q_1, \lambda, S) &= \{(q_1, aSa), (q_1, bSb), (q_1, \lambda)\}, \\ \Delta(q_1, a, a) &= \{(q_1, \lambda)\}, \\ \Delta(q_1, b, b) &= \{(q_1, \lambda)\}, \\ \Delta(q_1, \lambda, z_0) &= \{(q_2, z_0)\}. \end{aligned}$$

Puede observarse que este autómata es diferente del exhibido en el tercer ejemplo de la sección 5.2.

Ejercicios de la sección 5.4

- ① Construir un AFPN M que acepte el lenguaje generado por la siguiente gramática:

$$G : \quad \begin{cases} S \rightarrow Aba \mid AB \mid \lambda \\ A \rightarrow aAS \mid a \\ B \rightarrow bBA \mid \lambda. \end{cases}$$

Encontrar una derivación a izquierda en G de la cadena $w = aaababa$ y procesar luego la cadena w con el autómata M , simulando la derivación.

- ② Diseñar una gramática, con una sola variable, que genere el lenguaje $L = \{a^{2i}b^{3i} : i \geq 0\}$ y utilizar luego el procedimiento del Teorema 5.4.1 para construir un AFPN que acepte a L . Comparar este autómata con el construido en el ejercicio ③ de la sección 5.2.

5.5. Autómatas con pila y LIC. Parte II. \mathbb{X}

En la presente sección consideraremos la segunda parte de la correspondencia entre AFPN y LIC. Demostraremos que para todo AFPN que acepta por pila vacía existe una GIC que genera el lenguaje aceptado por el autómata. Las gramáticas obtenidas son, en general, bastante complejas, con un gran número de variables y de producciones. Hay que advertir también que el procedimiento puede dar lugar a muchas variables inútiles (no-terminables o no alcanzables).

5.5.1 Teorema. *Dado un AFPN $M = (Q, q_0, \Sigma, \Gamma, z_0, \Delta)$ que acepta por pila vacía, existe una GIC $G = (\Sigma, V, S, P)$ tal que $L(G) = N(M)$.*

Demostración. En la gramática G las variables (aparte de la variable inicial S) serán tripletas de la forma $[qXp]$ donde $q, p \in Q$ y $X \in \Gamma$. Las producciones de G se definen de la siguiente manera:

1. Si $(p, \lambda) \in \Delta(q, a, X)$, se añade la producción $[qXp] \rightarrow a$.
2. Si $(p, \lambda) \in \Delta(q, \lambda, X)$, se añade la producción $[qXp] \rightarrow \lambda$.
3. Si $(r, Y_1Y_2 \cdots Y_k) \in \Delta(q, a, X)$, donde a puede ser un símbolo del alfabeto Σ ó $a = \lambda$ y $k \geq 1$, se añaden todas las producciones de la forma

$$[qXr_k] \rightarrow a[rY_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k]$$

para todas las secuencias posibles r_1, r_2, \dots, r_{k-1} de estados de Q .

4. Para todo $p \in Q$ se añade la producción $S \rightarrow [q_0z_0p]$.

La gramática G así definida pretende simular con derivaciones a izquierda los cálculos de M ; el significado intuitivo de la variable $[qXp]$ es: “al extraer X del tope de la pila, se pasa del estado q al estado p ”. La producción

$$[qXr_k] \rightarrow a[rY_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k]$$

del numeral 3 indica las posibles maneras en las que M puede extraer la cadena $Y_1Y_2 \cdots Y_k$ de la pila, una vez se haya sustituido el tope de la pila X por dicha cadena, pasando del estado q al estado r y consumiendo el símbolo a .

Demostraremos primero la inclusión $N(M) \subseteq L(G)$. Para todo $q, p \in Q$, $X \in \Gamma$ y $w \in \Sigma^*$, se demostrará la implicación

$$(5.1) \quad \text{si } (q, w, X) \stackrel{+}{\vdash} (p, \lambda, \lambda) \text{ entonces } [qXp] \stackrel{+}{\Rightarrow} w,$$

por inducción sobre el número de pasos del cálculo $(q, w, X) \stackrel{+}{\vdash} (p, \lambda, \lambda)$. Cuando hay un sólo paso, el cálculo es de la forma $(q, a, X) \vdash (p, \lambda, \lambda)$ o de la forma $(q, \lambda, X) \vdash (p, \lambda, \lambda)$. Si $(q, a, X) \vdash (p, \lambda, \lambda)$, entonces $(p, \lambda) \in \Delta(q, a, X)$; así que $[qXp] \rightarrow a$ es una producción de G , y se obtendrá la derivación $[qXp] \Rightarrow a$. Si $(q, \lambda, X) \vdash (p, \lambda, \lambda)$, entonces $(p, \lambda) \in \Delta(q, \lambda, X)$; así que $[qXp] \rightarrow \lambda$ es una producción de G y se obtendrá $[qXp] \Rightarrow \lambda$.

Para el razonamiento inductivo, supóngase que $(q, w, X) \stackrel{n}{\vdash} (p, \lambda, \lambda)$ donde $n > 1$. Considerando el primer paso de este cálculo de n pasos, podemos escribir:

$$(5.2) \quad (q, ax, X) \vdash (r_0, x, Y_1 Y_2 \cdots Y_k) \stackrel{*}{\vdash} (p, \lambda, \lambda),$$

donde $a \in \Sigma$ ó $a = \lambda$. Cuando $a \in \Sigma$, $w = ax$ para alguna cadena $x \in \Sigma^*$; cuando $a = \lambda$, $x = w$. En el primer paso de 5.2 se ha aplicado la transición $(r_0, Y_1 Y_2 \cdots Y_k) \in \Delta(q, a, X)$ de M . Por la definición de la gramática G ,

$$[qXr_k] \rightarrow a[r_0Y_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k]$$

es una producción, para todas las secuencias posibles r_1, r_2, \dots, r_{k-1} de estados de Q .

Según 5.2, desde la configuración instantánea $(r_0, x, Y_1 Y_2 \cdots Y_k)$ el autómata llega hasta la configuración (p, λ, λ) , consumiendo completamente la cadena x y vaciando la pila. La cadena x se puede escribir entonces como $x = w_1 w_2 \cdots w_k$, siendo w_i la cadena consumida por el autómata para extraer el símbolo Y_i del tope de la pila. En consecuencia, existe una secuencia de estados $r_1, r_2, \dots, r_{k-1}, r_k = p$ tales que

$$\begin{aligned} (r_0, x, Y_1 Y_2 \cdots Y_k) &= (r_0, w_1 w_2 \cdots w_k, Y_1 Y_2 \cdots Y_k) \stackrel{+}{\vdash} (r_1, w_2 \cdots w_k, Y_2 \cdots Y_k) \\ &\stackrel{+}{\vdash} (r_2, w_3 \cdots w_k, Y_3 \cdots Y_k) \stackrel{+}{\vdash} (r_{k-1}, w_k, Y_k) \stackrel{+}{\vdash} (r_k, \lambda, \lambda). \end{aligned}$$

Para $i = 1, 2, \dots, k$ se tiene así una secuencia de cálculos parciales

$$(r_{i-1}, w_i, Y_i) \stackrel{+}{\vdash} (r_i, \lambda, \lambda),$$

donde $r_k = p$. Por la hipótesis de inducción, $[r_{i-1}Y_ir_i] \stackrel{+}{\Rightarrow} w_i$ para $i = 1, 2, \dots, k$. Por consiguiente,

$$[qXp] = [qXr_k] \Rightarrow a[r_0Y_1r_1][r_1Y_2r_2] \cdots [r_{k-1}Y_kr_k] \stackrel{+}{\Rightarrow} aw_1 w_2 \cdots w_k = w.$$

Esto demuestra la implicación 5.1. Por lo tanto, si w es aceptada por M , siendo $w \neq \lambda$, se tendrá $(q_0, w, z_0) \stackrel{+}{\vdash} (p, \lambda, \lambda)$, y usando 5.1 se concluirá que $S \implies [q_0 z_0 p] \stackrel{+}{\implies} w$. Si $w = \lambda$ es aceptada por M , necesariamente $(q_0, \lambda, z_0) \vdash (p, \lambda, \lambda)$ para algún estado p . Esto significa que $(q_0, \lambda, z_0) \vdash (p, \lambda, \lambda)$, y se tendrá $S \implies [q_0 z_0 p] \implies \lambda$. Esto demuestra que $N(M) \subseteq L(G)$.

Para establecer $L(G) \subseteq N(M)$ se demuestra la implicación

$$(5.3) \quad \text{si } [q X p] \stackrel{+}{\implies} w, \text{ entonces } (q, w, X) \stackrel{+}{\vdash} (p, \lambda, \lambda),$$

por inducción sobre el número de pasos en la derivación $[q X p] \stackrel{+}{\implies} w$. Este razonamiento inductivo es similar al usado para probar la implicación recíproca 5.1, y se deja como ejercicio para el estudiante interesado. Si en G se puede derivar la cadena w , es decir, si $S \stackrel{*}{\implies} w$, la primera producción aplicada será de la forma $S \rightarrow [q_0 z_0 p]$. De donde, $S \implies [q_0 z_0 p] \stackrel{+}{\implies} w$. Usando 5.3 se concluye $(q_0, w, z_0) \stackrel{+}{\vdash} (p, \lambda, \lambda)$, o sea $w \in N(M)$. \square

Ejemplo Vamos a aplicar la construcción del Teorema 5.5.1 para encontrar una gramática G que genere el lenguaje de las cadenas sobre el alfabeto $\Sigma = \{a, b\}$ que tienen igual número de aes que de bes, a partir del AFPN $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$ con los siguientes componentes. $\Sigma = \{a, b\}$, $\Gamma = \{z_0, A, B\}$, $Q = \{q_0, q_1\}$, $F = \{q_1\}$ y la función de transición Δ está dada por:

$$\begin{aligned}\Delta(q_0, a, z_0) &= \{(q_0, Az_0)\}, \\ \Delta(q_0, b, z_0) &= \{(q_0, Bz_0)\}, \\ \Delta(q_0, a, A) &= \{(q_0, AA)\}, \\ \Delta(q_0, b, B) &= \{(q_0, BB)\}, \\ \Delta(q_0, a, B) &= \{(q_0, \lambda)\}, \\ \Delta(q_0, b, A) &= \{(q_0, \lambda)\}, \\ \Delta(q_0, \lambda, z_0) &= \{(q_1, \lambda)\}.\end{aligned}$$

M es una modificación del autómata presentado en el segundo ejemplo de la sección 5.2 (en ese ejemplo M aceptaba por estado final; aquí M acepta por pila vacía).

Las variables de G son S y todas las tripletas de la forma $[q X p]$ donde $q, p \in Q$ y $X \in \Gamma$. Hay, por lo tanto, 13 variables, a saber:

$$\begin{aligned}S, [q_0 z_0 q_0], [q_0 A q_0], [q_0 B q_0], [q_0 z_0 q_1], [q_0 A q_1], [q_0 B q_1], \\ [q_1 z_0 q_0], [q_1 A q_0], [q_1 B q_0], [q_1 z_0 q_1], [q_1 A q_1], [q_1 B q_1].\end{aligned}$$

A continuación se presentan las producciones de G .

Producción obtenida de $\Delta(q_0, \lambda, z_0) = \{(q_1, \lambda)\}$:

$$[q_0 z_0 q_1] \rightarrow \lambda.$$

Producción obtenida de $\Delta(q_0, a, B) = \{(q_0, \lambda)\}$:

$$[q_0 B q_0] \rightarrow a.$$

Producción obtenida de $\Delta(q_0, b, A) = \{(q_0, \lambda)\}$:

$$[q_0 A q_0] \rightarrow b.$$

Producciones obtenidas de $\Delta(q_0, a, z_0) = \{(q_0, Az_0)\}$:

$$\begin{aligned} [q_0 z_0 q_0] &\rightarrow a[q_0 A q_0][q_0 z_0 q_0] \mid a[q_0 A q_1][q_1 z_0 q_0] \\ [q_0 z_0 q_1] &\rightarrow a[q_0 A q_0][q_0 z_0 q_1] \mid a[q_0 A q_1][q_1 z_0 q_1]. \end{aligned}$$

Producciones obtenidas de $\Delta(q_0, b, z_0) = \{(q_0, Bz_0)\}$:

$$\begin{aligned} [q_0 z_0 q_0] &\rightarrow b[q_0 B q_0][q_0 z_0 q_0] \mid b[q_0 B q_1][q_1 z_0 q_0] \\ [q_0 z_0 q_1] &\rightarrow b[q_0 B q_0][q_0 z_0 q_1] \mid b[q_0 B q_1][q_1 z_0 q_1]. \end{aligned}$$

Producciones obtenidas de $\Delta(q_0, a, A) = \{(q_0, AA)\}$:

$$\begin{aligned} [q_0 A q_0] &\rightarrow a[q_0 A q_0][q_0 A q_0] \mid a[q_0 A q_1][q_1 A q_0] \\ [q_0 A q_1] &\rightarrow a[q_0 A q_0][q_0 A q_1] \mid a[q_0 A q_1][q_1 A q_1]. \end{aligned}$$

Producciones obtenidas de $\Delta(q_0, b, B) = \{(q_0, BB)\}$:

$$\begin{aligned} [q_0 B q_0] &\rightarrow b[q_0 B q_0][q_0 B q_0] \mid b[q_0 B q_1][q_1 B q_0] \\ [q_0 B q_1] &\rightarrow b[q_0 B q_0][q_0 B q_1] \mid b[q_0 B q_1][q_1 B q_1]. \end{aligned}$$

Finalmente, las producciones de la variable inicial S son:

$$S \rightarrow [q_0 z_0 q_0] \mid [q_0 z_0 q_1].$$

Al examinar las producciones se puede observar que todas las variables de la forma $[q_1 X q]$, con $X \in \Gamma$ y $q \in Q$, son inútiles ya que no tienen producciones. Con la terminología ya conocida, dichas variables son no-terminables y, por consiguiente, las producciones en las que aparecen se pueden eliminar. Otra variable no terminable es $[q_0 z_0 q_0]$. Además, las variables $[q_0 A q_1]$ y $[q_0 B q_1]$ son inalcanzables, así que se pueden eliminar, junto con todas

sus producciones. Realizando estas simplificaciones se obtiene la siguiente gramática:

$$\begin{aligned} S &\rightarrow [q_0 z_0 q_1] \\ [q_0 z_0 q_1] &\rightarrow a[q_0 A q_0][q_0 z_0 q_1] \mid b[q_0 B q_0][q_0 z_0 q_1] \mid \lambda \\ [q_0 A q_0] &\rightarrow a[q_0 A q_0][q_0 A q_0] \mid b \\ [q_0 B q_0] &\rightarrow b[q_0 B q_0][q_0 B q_0] \mid a. \end{aligned}$$

Como se indicó en la demostración, en las gramáticas construidas según el método del Teorema 5.5.1, las derivaciones a izquierda corresponden a los cómputos en el autómata dado. Podemos ilustrar este punto, en el presente ejemplo, con la cadena de entrada $w = bbabaa$. El siguiente es un cálculo de aceptación de w en M :

$$\begin{aligned} (q_0, bbabaa, z_0) &\vdash (q_0, babaa, Bz_0) \vdash (q_0, abaa, BBz_0) \vdash (q_0, baa, Bz_0) \\ &\vdash (q_0, aa, BBz_0) \vdash (q_0, a, Bz_0) \vdash (q_0, \lambda, z_0) \vdash (q_1, \lambda, \lambda). \end{aligned}$$

La derivación a izquierda en G que corresponde a este cálculo es:

$$\begin{aligned} S &\Rightarrow [q_0 z_0 q_1] \Rightarrow b[q_0 B q_0][q_0 z_0 q_1] \Rightarrow bb[q_0 B q_0][q_0 B q_0][q_0 z_0 q_1] \\ &\Rightarrow bba[q_0 B q_0][q_0 z_0 q_1] \Rightarrow bbab[q_0 B q_0][q_0 B q_0][q_0 z_0 q_1] \\ &\Rightarrow bbaba[q_0 B q_0][q_0 z_0 q_1] \Rightarrow bbabaa[q_0 z_0 q_1] \Rightarrow bbabaa. \end{aligned}$$

Obsérvese que, en cada paso de la derivación, el contenido actual de la pila se puede leer examinando las segundas componentes de las tripletas.

Para hacer más legibles las producciones de la gramática G obtenida en este ejemplo, cambiamos los nombres de las variables así: $C = [q_0 z_0 q_1]$, $D = [q_0 A q_0]$ y $E = [q_0 B q_0]$. Con esta nomenclatura, la gramática se puede escribir como:

$$G : \left\{ \begin{array}{l} S \rightarrow C \\ C \rightarrow aDC \mid bEC \mid \lambda \\ D \rightarrow aDD \mid b \\ E \rightarrow bEE \mid a \end{array} \right.$$

Puesto que la única producción de S es $S \rightarrow C$, las variables S y C se pueden identificar, dando lugar a la siguiente gramática simplificada equivalente:

$$\left\{ \begin{array}{l} S \rightarrow aDS \mid bES \mid \lambda \\ D \rightarrow aDD \mid b \\ E \rightarrow bEE \mid a \end{array} \right.$$

Ejercicios de la sección 5.5

- ① Con respecto al ejemplo de esta sección, procesar con M la cadena de entrada $w = baabbbbaa$ y luego derivar w en la gramática G , simulando dicho procesamiento.
- !② Diseñar un AFPN que acepte el lenguaje L de las cadenas sobre el alfabeto $\Sigma = \{a, b\}$ que tienen el doble número de a s que de b s. Aplicar luego la construcción del Teorema 5.5.1 para encontrar una gramática que genere a L .

Máquinas de Turing

En este capítulo se presenta la Máquina de Turing (MT) que es el modelo de autómata con máxima capacidad computacional: la unidad de control puede desplazarse a izquierda o a derecha y sobre-escribir símbolos en la cinta de entrada. Argumentaremos que la máquina de Turing tiene la misma potencia o capacidad de los computadores reales.

6.1. Máquinas de Turing como aceptadoras de lenguajes

Una **máquina de Turing (MT)**, $M = (Q, q_0, F, \Sigma, \Gamma, \mathbf{b}, \delta)$, consta de siete componentes:

1. Q es el conjunto (finito) de estados internos de la unidad de control.
2. $q_0 \in Q$ es el estado inicial.
3. F es el conjunto de estados finales o de aceptación, $\emptyset \neq F \subseteq Q$.
4. Σ es el alfabeto de entrada.
5. Γ es el alfabeto de cinta, que incluye a Σ , es decir, $\Sigma \subseteq \Gamma$.
6. $\mathbf{b} \in \Gamma$ es el símbolo “blanco” (el símbolo \mathbf{b} no puede hacer parte del alfabeto de entrada Σ).
7. δ es la función de transición de la máquina:

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\leftarrow, \rightarrow\}$$

δ es una *función parcial*, es decir, puede no estar definida en algunos elementos del dominio. La flecha \leftarrow denota desplazamiento a izquierda mientras que \rightarrow denota desplazamiento a la derecha. La transición

$$\delta(q, a) = (p, b, D)$$

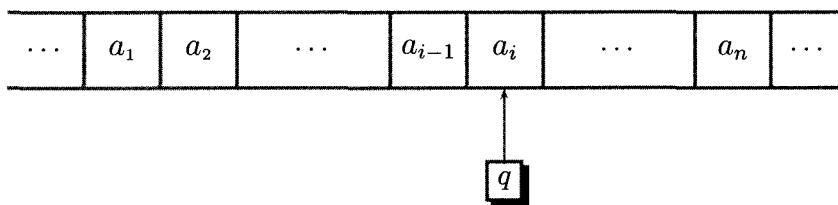
significa: estando en el estado q , escaneando el símbolo a , la unidad de control borra a , escribe b y se mueve en el estado p , ya sea a la izquierda (si el desplazamiento D es \leftarrow) o a la derecha (si D es \rightarrow).

Una máquina de Turing M procesa cadenas de entrada $w \in \Sigma^*$ colocadas sobre una cinta infinita en ambas direcciones. Para procesar una cadena de entrada w , la unidad de control de M está en el estado inicial q_0 escaneando el primer símbolo de w . Las demás celdas o casillas de la cinta contienen el símbolo blanco b .

Descripción o configuración instantánea. Es una expresión de la forma

$$a_1 a_2 \cdots a_{i-1} q a_i \cdots a_n$$

donde los símbolos a_1, \dots, a_n pertenecen al alfabeto de cinta Γ y $q \in Q$. Esta expresión representa el estatus actual del cómputo:



Es decir, la descripción instantánea $a_1 a_2 \cdots a_{i-1} q a_i \cdots a_n$ indica que la unidad de control de M está en el estado q escaneando el símbolo a_i . Se supone que todas las casillas, a la izquierda de a_1 y a la derecha de a_n , contienen el símbolo blanco, b .

Ejemplos concretos de descripciones instantáneas son:

$$\begin{aligned} & aabq_2baaa \\ & q_5ababba \\ & ab\text{b}\text{b}aabq_0bba \end{aligned}$$

La configuración o descripción instantánea inicial, o simplemente *configuración inicial*, es q_0w , donde w es la cadena de entrada. Ésta se coloca en cualquier parte de la cinta de entrada.

Paso computacional. El paso de una descripción instantánea a otra, por medio de una transición definida por δ , se denomina un paso computacional y se denota por

$$u_1qu_2 \vdash v_1pv_2.$$

Aquí $u_1, u_2, v_1, v_2 \in \Gamma^*$ y $p, q \in Q$. Un ejemplo concreto es

$$abbaq_2ba \vdash abbq_1aca$$

en la cual la máquina utilizó la transición $\delta(q_2, b) = (q_1, c, \leftarrow)$.

La notación

$$u_1qu_2 \stackrel{*}{\vdash} v_1pv_2$$

significa que M puede pasar de la descripción instantánea u_1qu_2 a la descripción instantánea v_1pv_2 en cero, uno o más pasos computacionales.

Cómputos especiales. Durante el cómputo o procesamiento de una cadena de entrada hay dos situaciones especiales que se pueden presentar:

1. El cómputo termina porque en determinado momento no hay transición definida.
2. El cómputo no termina; esto es lo que se denomina un “bucle infinito” o un “ciclo infinito”. Esta situación se representa con la notación

$$u_1qu_2 \stackrel{*}{\vdash} \infty$$

la cual indica que el cómputo que se inicia en la descripción instantánea u_1qu_2 no se detiene nunca.

Un detalle para tener en cuenta es que las transiciones con símbolo \mathbf{b} , $\delta(q, \mathbf{b}) = (p, s, D)$, no son las mismas transiciones λ usadas para autómatas en capítulos anteriores. Una transición λ en un autómata tiene lugar independientemente del símbolo leído y la unidad de control permanece estacionaria, mientras que una transición $\delta(q, \mathbf{b}) = (p, s, D)$ en una MT requiere que el símbolo blanco \mathbf{b} esté escrito en la casilla escaneada; además, la unidad de control sobre-escribe el blanco y realiza un desplazamiento D .

Lenguaje aceptado por una MT. Una cadena de entrada w es aceptada por una MT M si el cómputo que se inicia en la configuración inicial q_0w

termina en una configuración instantánea w_1pw_2 , p estado de aceptación, en la cual M se *detiene completamente*. El lenguaje $L(M)$ aceptado por una MT M se define entonces como

$$L(M) := \{w \in \Sigma^* : q_0 w \xrightarrow{*} w_1 pw_2, p \in F, w_1, w_2 \in \Gamma^*, M \text{ se detiene en la configuración } w_1 pw_2\}.$$

La noción de aceptación para máquinas de Turing es más flexible que para autómatas: una cadena de entrada no tiene que ser leída en su totalidad para que sea aceptada; sólo se requiere que la máquina se detenga completamente, en un momento determinado, en un estado de aceptación.

Para simplificar los argumentos sobre máquinas de Turing, *en el modelo estándar se supone que la unidad de control siempre se detiene al ingresar a en un estado de aceptación. Es decir, no se permiten transiciones $\delta(q, a)$ cuando $q \in F$.*

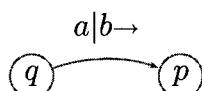
Las máquinas de Turing originan las siguientes clases de lenguajes:

1. L es un lenguaje *recursivamente enumerable* (RE) si existe una MT M tal que $L(M) = L$.
2. L es un lenguaje *recursivo* si existe una MT M tal que $L(M) = L$ y M se detiene con todas las cadenas de entrada.

Las denominaciones “lenguaje recursivamente enumerable” y “lenguaje recursivo” pueden parecer un tanto extrañas; esta terminología es anterior a la aparición del modelo de máquina de Turing y se ha mantenido vigente hasta el presente. En el Teorema 6.8.3 se justificará la denominación “recursivamente enumerable” (RE) al establecerse que para todo lenguaje RE se puede construir una MT que enumere secuencialmente sus cadenas.

Obviamente, todo lenguaje recursivo es recursivamente enumerable, pero la afirmación recíproca no es (en general) válida, como se demostrará en la sección 7.5. Esto permitirá concluir que el fenómeno de máquinas que nunca se detienen no se puede eliminar de la teoría de la computación.

Diagrama de transiciones o diagrama de flujo de una MT. La función de transición δ de una MT se puede presentar como un digrafo etiquetado. Así, la transición $\delta(q, a) = (p, b, \rightarrow)$ se representa por



Unidad de control estacionaria. Para simplificar la descripción del modelo estándar, se ha exigido que la unidad de control se desplace hacia la izquierda o hacia la derecha en cada transición. No obstante, se puede permitir que la unidad de control no se mueva en un determinado paso computacional, es decir, que no realice ningún desplazamiento. Este tipo de transición lo escribimos en la forma:

$$\delta(q, a) = (p, b, -)$$

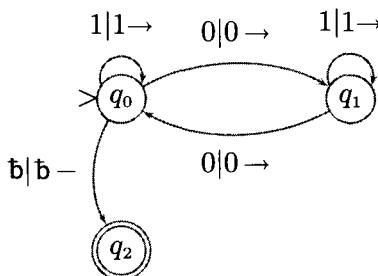
donde $a, b \in \Gamma$; $p, q \in Q$ y $-$ significa que la unidad de control no se mueve. Tal transición se puede simular por medio de un movimiento a la derecha seguido de un retorno a la izquierda. Así, para simular la transición $\delta(q, a) = (p, b, -)$ utilizamos un estado auxiliar nuevo q' y las transiciones:

$$\delta(q, a) = (q', b, \rightarrow),$$

$$\delta(q', s) = (p, s, \leftarrow), \quad \text{para todo símbolo } s \in \Gamma.$$

La directriz de no-desplazamiento también se puede simular con un movimiento a la izquierda seguido de un retorno a la derecha. En cualquier caso, concluimos que las MT en las que hay transiciones estacionarias, $\delta(q, a) = (p, b, -)$, aparte de las transiciones normales, aceptan los mismos lenguajes que las MT estándares. Nótese que las transiciones estacionarias se asemejan a las transiciones λ en autómatas, excepto por el hecho de que las máquinas de Turing tienen la capacidad de sobre-escribir los símbolos escaneados.

Ejemplo La siguiente MT acepta el lenguaje de las cadenas con un número par de ceros, sobre el alfabeto $\{0, 1\}$.



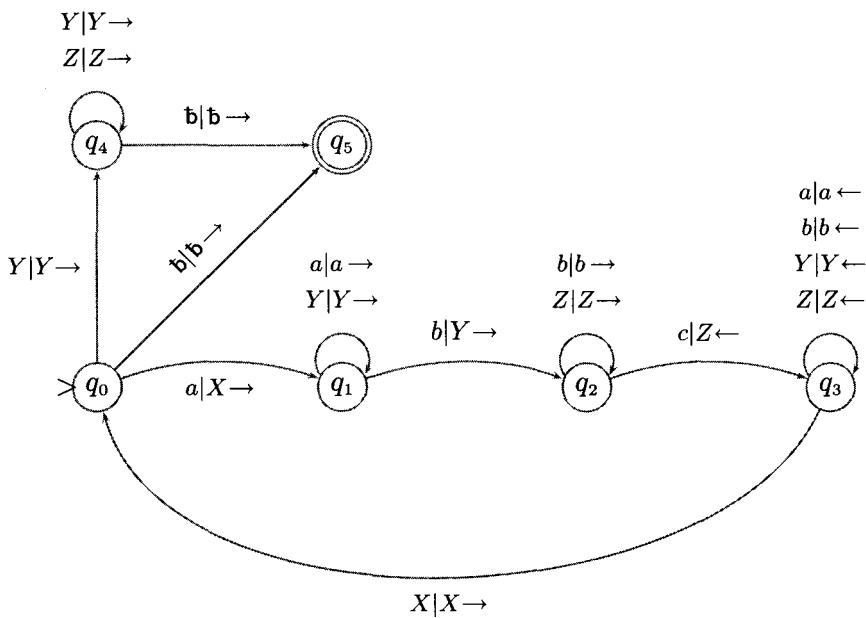
Ejemplo En este ejemplo se construye una MT M que acepta el lenguaje $L = \{a^i b^i c^i : i \geq 0\}$ y que se detiene al procesar todas las entradas. Por consiguiente, L es un lenguaje recursivo aunque no es LIC,

como se demostró en la sección 4.13; es decir, L no puede ser aceptado por ningún autómata con pila.

Sea M la MT con parámetros

$$\begin{aligned}\Sigma &= \{a, b, c\}, \\ \Gamma &= \{a, b, c, X, Y, Z, \text{b}\}, \\ Q &= \{q_0, q_1, q_2, q_3, q_4, q_5\}, \\ F &= \{q_5\},\end{aligned}$$

y cuya función de transición está representada por el siguiente diagrama:



La idea utilizada para el diseño de esta MT se puede describir así: la unidad de control cambia la primera a por X y se mueve a la derecha hasta encontrar la primera b , la cual se sobre-escribe por una Y . Luego se mueve hacia la derecha hasta encontrar la primera c , la cual se cambia por Z . El control retrocede entonces hacia la izquierda en busca de la primera X que encuentre en su camino; este retorno se hace en el estado q_3 . La máquina avanza luego hacia la derecha, hasta la primera a que quede en la cinta, y todo el proceso anterior se repite. Si la cadena de entrada tiene la forma requerida, todas las a s serán reemplazadas por X s, las b s por Y s y las c s por Z s. Una vez terminada la transformación, el control se mueve hacia la derecha, en el estado q_4 , hasta encontrar la primera celda marcada con el símbolo blanco b . La MT está diseñada de tal forma que si la cadena

de entrada no tiene la forma requerida, el procesamiento terminará en un estado diferente del estado de aceptación q_5 .

A continuación procesamos la cadena de entrada $w = aabbcc \in L$.

$$\begin{aligned}
 q_0 aabbcc &\vdash X q_1 abbcc \vdash X aq_1 bbcc \vdash X aY q_2 bcc \vdash X aY bq_2 cc \vdash X aY bq_3 bZc \\
 &\stackrel{*}{\vdash} q_3 X aY bZc \vdash X q_0 aY bZc \vdash X X q_1 Y bZc \vdash X X Y q_1 bZc \\
 &\vdash X X Y Y q_2 Zc \vdash X X Y Z q_2 c \vdash X X Y q_3 ZZ \stackrel{*}{\vdash} X q_3 XY ZZ \\
 &\vdash X X q_0 YY ZZ \vdash X X Y q_4 Y ZZ \stackrel{*}{\vdash} X X Y Y ZZ q_4 b \\
 &\vdash X X Y Y ZZ b q_5 b.
 \end{aligned}$$

La cadena de entrada $w = aaabbcc$, que no está en L , se procesa así:

$$\begin{aligned}
 q_0 aaabbcc &\vdash X q_1 aabbcc \vdash X aaq_1 bbcc \vdash X aaY q_2 bcc \vdash X aaY bq_2 cc \\
 &\vdash X aaY bq_3 bZc \stackrel{*}{\vdash} q_3 X aaY bZc \vdash X q_0 aaY bZc \vdash X X aq_1 Y bZc \\
 &\vdash X X aY q_1 bZc \vdash X X aYY q_2 Zc \vdash X X aYZ q_2 c \vdash X X aY q_3 ZZ \\
 &\stackrel{*}{\vdash} X q_3 X aY ZZ \vdash X X q_0 aYY ZZ \vdash X X X q_1 YY ZZ \\
 &\stackrel{*}{\vdash} X X X YY q_1 ZZ \text{ (cómputo abortado).}
 \end{aligned}$$

Por otro lado, la cadena $abbbcc$, que tampoco está en L , se procesaría así:

$$\begin{aligned}
 q_0 abbcc &\vdash X q_1 bbcc \vdash X Y q_2 bcc \vdash X Y bq_2 cc \vdash X Y q_3 bZc \stackrel{*}{\vdash} q_3 X Y bZc \\
 &\vdash X q_0 Y bZc \vdash X Y q_4 bZc \text{ (cómputo abortado).}
 \end{aligned}$$

Ejercicios de la sección 6.1

Según los ejercicios de la sección 4.13, los siguientes lenguajes no son LIC. Diseñar MT que los acepten, escribiendo explícitamente la idea utilizada en el diseño. Presentar cada MT por medio de un diagrama de transiciones.

- ① $L = \{a^i b^i c^j : j \geq i\}$.
- ② $L = \{0^i 1^{2i} 0^i : i \geq 0\}$.
- ③ $L = \{a^i b^j c^k : 1 \leq i \leq j \leq k\}$.
- ④ $L = \{a^i b^i c^i d^j : i, j \geq 0\}$.
- ⑤ $L = \{a^i b^i c^i d^i : i \geq 0\}$.

6.2. Subrutinas o macros

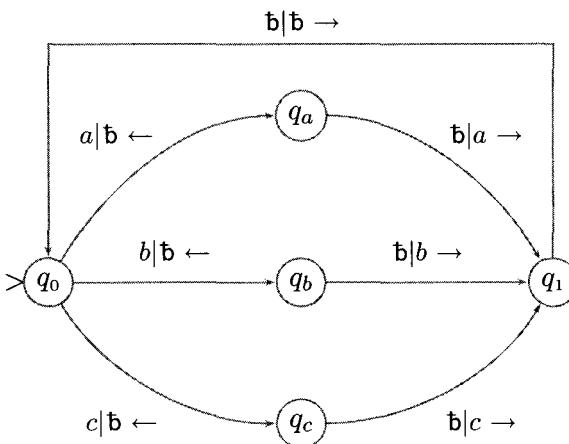
Hay procedimientos intermedios, como copiar o trasladar cadenas, que se repiten frecuentemente en el diseño de máquinas de Turing. Es útil identificar algunos de estos procedimientos, similares a las subrutinas o macros de los programas computacionales, para simplificar el diseño de máquinas más complejas. Las subrutinas se pueden considerar como máquinas de Turing con estado inicial pero sin estados de aceptación.

Ejemplo Subrutina TI, translación a la izquierda. Esta subrutina toma una cadena de entrada y la traslada una casilla hacia la izquierda. Si la entrada es la cadena $a_1a_2 \dots a_k$, la acción de TI se puede representar en la forma

$$\begin{array}{ll} \text{Entrada: } & \underline{\mathbf{b}} \ a_1 \cdots a_{k-1} a_k \ \mathbf{b} \\ & \downarrow \uparrow \quad \downarrow \quad \downarrow \uparrow \uparrow \\ \text{Salida: } & a_1 a_2 \cdots a_k \ \underline{\mathbf{b}} \ \mathbf{b} \end{array}$$

La flecha \uparrow indica que las casillas señaladas sobre la cinta coinciden. Los símbolos subrayados en las dos cintas representan los símbolos leídos por la unidad de control al iniciar y al terminar la subrutina, respectivamente.

La MT que aparece a continuación sirve para implementar la subrutina TI cuando el alfabeto de entrada es $\{a, b, c\}$:



A continuación definimos algunas otras subrutinas útiles; se solicita al estudiante diseñar MT que los implementen.

Subrutina TD, translación a la derecha. Traslada la cadena de entrada una casilla a la derecha:

$$\begin{array}{ll} \text{Entrada: } & \underline{\mathbf{b}} a_1 a_2 \cdots a_k \mathbf{b} \\ & \uparrow \downarrow \uparrow \quad \uparrow \quad \uparrow \\ \text{Salida: } & \mathbf{b} \underline{\mathbf{b}} a_1 \cdots a_{k-1} a_k \end{array}$$

Subrutina BI, primer blanco a la izquierda. Busca el primer símbolo \mathbf{b} a la izquierda de una cadena w que no posee blancos:

$$\begin{array}{ll} \text{Entrada: } & \mathbf{b} w \underline{\mathbf{b}} \\ & \uparrow \quad \uparrow \\ \text{Salida: } & \underline{\mathbf{b}} w \mathbf{b} \end{array}$$

Subrutina BD, primer blanco a la derecha. Busca el primer símbolo \mathbf{b} a la derecha de una cadena w que no posee blancos:

$$\begin{array}{ll} \text{Entrada: } & \underline{\mathbf{b}} w \mathbf{b} \\ & \uparrow \quad \uparrow \\ \text{Salida: } & \mathbf{b} w \underline{\mathbf{b}} \end{array}$$

Subrutina NBI, primer símbolo no-blanco a la izquierda. Busca el primer símbolo diferente de \mathbf{b} a la izquierda de una cadena de blancos:

$$\begin{array}{ll} \text{Entrada: } & a \mathbf{b} \mathbf{b} \cdots \mathbf{b} \mathbf{b} \\ & \uparrow \uparrow \downarrow \quad \uparrow \uparrow \\ \text{Salida: } & \underline{a} \mathbf{b} \mathbf{b} \cdots \mathbf{b} \mathbf{b} \end{array}$$

Subrutina NBD, primer símbolo no-blanco a la derecha. Busca el primer símbolo diferente de \mathbf{b} a la derecha de una cadena de blancos:

$$\begin{array}{ll} \text{Entrada: } & \underline{\mathbf{b}} \mathbf{b} \cdots \mathbf{b} \mathbf{b} a \\ & \uparrow \downarrow \uparrow \quad \uparrow \uparrow \\ \text{Salida: } & \mathbf{b} \mathbf{b} \cdots \mathbf{b} \mathbf{b} \underline{a} \end{array}$$

Subrutina COPD, copia a la derecha. Copia la cadena w a la derecha de si misma, separando las dos w por un blanco \mathbf{b} . La cadena w no posee blancos:

$$\begin{array}{ll} \text{Entrada: } & \underline{\mathbf{b}} w \mathbf{b} \cdots \mathbf{b} \\ & \uparrow \quad \uparrow \quad \uparrow \\ \text{Salida: } & \mathbf{b} w \mathbf{b} w \underline{\mathbf{b}} \end{array}$$

Subrutina COPI, copia a la izquierda. Copia la cadena w a la izquierda de si misma, separando las dos w por un blanco \mathbf{b} . La cadena w que no posee

blancos:

$$\begin{array}{ll} \text{Entrada:} & \underline{\mathbf{b}} \cdots \underline{\mathbf{b}} w \underline{\mathbf{b}} \\ & \downarrow \quad \downarrow \quad \downarrow \\ \text{Salida:} & \underline{\mathbf{b}} w \underline{\mathbf{b}} w \underline{\mathbf{b}} \end{array}$$

Subrutina INT, intercambio. Intercambia las cadenas u y v (estas cadenas no poseen blancos y no necesariamente son de la misma longitud):

$$\begin{array}{ll} \text{Entrada:} & \underline{\mathbf{b}} u \underline{\mathbf{b}} v \underline{\mathbf{b}} \\ & \downarrow \quad \downarrow \\ \text{Salida:} & \underline{\mathbf{b}} v \underline{\mathbf{b}} u \underline{\mathbf{b}} \end{array}$$

Ejercicios de la sección 6.2

- ① Diseñar MT que implementen las subrutinas TD, BI, BD, NBI, NBD, COPD, COPI e INT cuando el alfabeto de entrada es $\{a, b\}$.
- ② Diseñar MT que implementen las subrutinas TD, BI, BD, NBI, NBD, COPD, COPI e INT cuando el alfabeto de entrada es $\{0, 1, 2\}$.
- ③ Sea $\Sigma = \{a, b\}$. Diseñar MT para las siguientes subrutinas:
 - (i) TI_2 . Traslada la cadena de entrada 2 casillas a la izquierda.
 - (ii) TD_2 . Traslada la cadena de entrada 2 casillas a la derecha.
 - (iii) TI_k . Traslada la cadena de entrada k casillas a la izquierda (k es un entero fijo ≥ 1).
 - (iv) TD_k . Traslada la cadena de entrada k casillas a la derecha (k es un entero fijo ≥ 1).

6.3. Máquinas de Turing como calculadoras de funciones

Como las máquinas de Turing tienen la capacidad de transformar las cadenas de entrada, borrando o sobre-escribiendo símbolos, se pueden utilizar como mecanismos para calcular funciones. Formalmente, una MT $M = (Q, q_0, q_f, \Sigma, \Gamma, \mathbf{b}, \delta)$ **calcula o computa una función** $h : \Sigma^* \rightarrow \Gamma^*$ si para toda entrada $w \in \Sigma^*$ se tiene

$$q_0 w \xrightarrow{*} q_f v, \quad \text{siempre que } v = h(w).$$

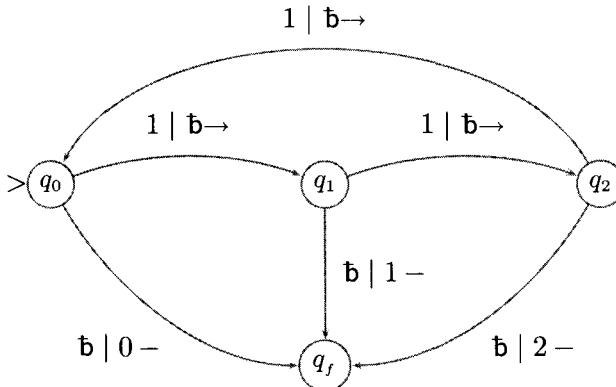
Si existe una MT que calcule la función h , se dice que h es **Turing-calculable** o **Turing-computable**. El modelo de MT aquí utilizado coincide con el estándar, pero no hay estados de aceptación. El estado q_f , llamado

estado final, se usa para terminar el procesamiento de la entrada y producir la salida. No se permiten transiciones desde el estado final q_f . Nótese que M debe terminar el procesamiento en la configuración $q_f v$, o sea, la unidad de control debe estar escaneando el primer símbolo de la salida v .

Si M no se detiene con una entrada particular w , la función h no está definida en w . Es decir, el dominio de h está formado únicamente por aquellas cadenas para las cuales M se detiene. Es corriente escribir $h(w) \downarrow$ para indicar que h está definida en w (h converge en w) y $h(w) \uparrow$ para indicar que h no está definida en w (h diverge en w). Si la función h está definida para toda entrada w (es decir, si M se detiene para toda w) se dice que h es una **función total**. Una función indefinida para algunas entradas se dice que es una **función parcial**.

Ejemplo Diseñar una MT M que calcule el residuo de división de n por 3, para cualquier número natural $n \geq 1$ escrito en el sistema de numeración unitario (n se escribe como una secuencia de n unos).

Solución. Los posibles residuos de división por 3 son 0, 1 y 2, por lo cual bastan 3 estados, aparte de q_f , para calcular esta función. M recorre de izquierda a derecha la secuencia de entrada borrando los unos y pasando alternadamente por los estados q_0 (que representa el residuo 0), q_1 (residuo 1) y q_2 (residuo 2). El siguiente es el diagrama de transiciones de M :



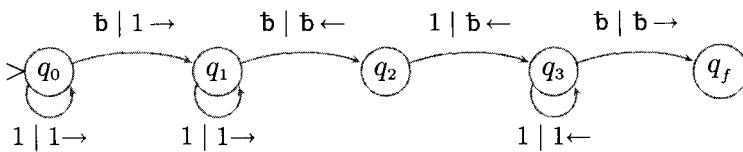
La noción de función Turing-computable se puede extender fácilmente a funciones de varios argumentos. Concretamente, una función h de k argumentos es Turing-computable si para toda k -upla (w_1, w_2, \dots, w_k) se tiene

$$q_0 w_1 b w_2 b \cdots b w_k b \stackrel{*}{\vdash} q_f v, \quad \text{siempre que } v = h(w_1, w_2, \dots, w_k).$$

Nótese que para escribir la entrada en la cinta, los k argumentos w_1, w_2, \dots, w_k se separan entre sí con el símbolo blanco b . Igual que antes, no se permiten transiciones desde el estado final q_f . Esta definición abarca tanto los casos de las funciones totales (definidas para toda entrada) como los de las funciones parciales (indefinidas para algunas entradas).

Ejemplo Diseñar una MT M que calcule la función suma, en el sistema de numeración unitario. Ésta es una función de dos argumentos, $h(n, m) = n + m$, donde $n, m \geq 1$. Con entrada $1^n \text{b} 1^m$, M debe producir como salida 1^{n+m} . Las secuencias de unos, 1^n y 1^m , representan los números naturales n y m , respectivamente.

Solución. M va a transformar $1^n \text{b} 1^m$ en 1^{n+m} trasladando la cadena 1^m una casilla hacia la izquierda. El separador b se sobre-escribe por 1 y el último 1 de 1^m se sobre-escribe por el símbolo b . El diagrama de transiciones de la MT que implementa esta idea es:



Ejercicios de la sección 6.3

Diseñar MT que calculen las siguientes funciones; hacer diagramas de transiciones para las máquinas obtenidas. Las entradas numéricas se dan en el sistema de numeración unitario, como en los ejemplos de esta sección.

- ① La función de paridad de los números naturales ($n \geq 1$):

$$h(n) = \begin{cases} 1, & \text{si } n \text{ es par,} \\ 0, & \text{si } n \text{ es impar.} \end{cases}$$

- ② $h(n) = 2n$, para todo número natural $n \geq 1$.

- ③ Para $n, m \geq 1$,

$$f(n, m) = \begin{cases} 1, & \text{si } n \geq m, \\ 0, & \text{si } n < m. \end{cases}$$

- ④ Para $n, m \geq 1$, $h(n, m) = \max(n, m)$.

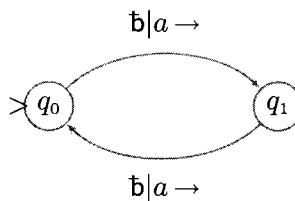
- ⑤ Para $i, j, k \geq 1$, $h(i, j, k) = j$. Esta función se llama “segunda proyección”. Observación: en general, la i -ésima proyección sobre k variables, $h(n_1, \dots, n_i, \dots, n_k) = n_i$, es Turing-computable.

6.4. Máquinas de Turing como generadoras de lenguajes

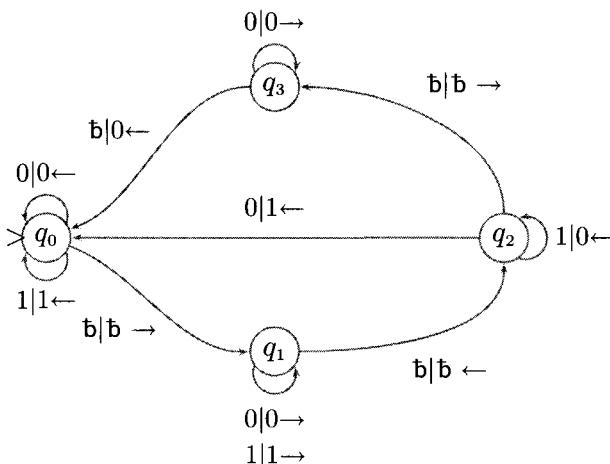
Otra faceta importante de las MT es su capacidad para generar lenguajes, tarea para la cual no son necesarios los estados de aceptación. Concretamente, una MT $M = (Q, q_0, \Sigma, \Gamma, \mathbf{b}, \delta)$ **genera el lenguaje** $L \subseteq \Sigma^*$ si

1. M comienza a operar con la cinta en blanco en el estado inicial q_0 .
2. Cada vez que M retorna al estado inicial q_0 , hay una cadena u perteneciente al lenguaje L escrita sobre la cinta.
3. Todas las cadenas de L son, eventualmente, generadas por M .

Ejemplo La siguiente MT genera cadenas con un número par de *aes* sobre el alfabeto $\Sigma = \{a\}$, o sea, el lenguaje $L = \{a^{2i} : i \geq 1\}$.



Ejemplo La siguiente MT genera todas cadenas de ceros y unos en el orden lexicográfico (las cadenas se ordenan por longitud y las cadenas de la misma longitud se ordenan ortográficamente de izquierda a derecha, considerando $0 < 1$): $0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, 0001, \dots$



Ejercicios de la sección 6.4

- ① Diseñar MT que generen los siguientes lenguajes:
- $L = \{a^i b^i : i \geq 1\}$.
 - $L = \{a^i c b^i : i \geq 1\}$.
 - $L = \{a^i : i \text{ es divisible por } 3, i \geq 1\}$.
- ② Modificar la MT del último ejemplo de esta sección para diseñar una MT que genere todas las cadenas de $\{a, b, c\}^*$ en orden lexicográfico. Ayuda: no se necesitan más estados.
- ③ Sea Σ un alfabeto de n símbolos, $\Sigma = \{a_1, a_2, \dots, a_n\}$. Modificar la MT del último ejemplo de esta sección para diseñar una MT que genere Σ^* en orden lexicográfico. Ayuda: no se necesitan más estados.

6.5. Variaciones del modelo estándar de MT

Hemos visto cómo las máquinas de Turing se pueden utilizar para aceptar lenguajes (en lo cual se asemejan a los autómatas de capítulos anteriores), para generar lenguajes y para calcular funciones de uno o de varios argumentos. En esta sección veremos que, a pesar de su simplicidad, el modelo estándar de máquina de Turing es suficientemente flexible como para simular acciones computacionales más complejas, entre las que se encuentran el uso de múltiples cintas y el no-determinismo.

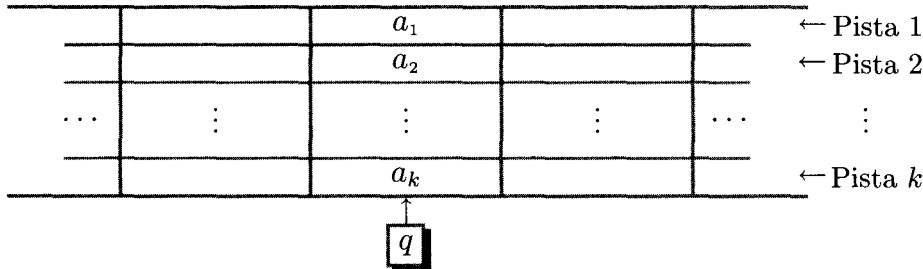
6.5.1. Estado de aceptación único

Las máquinas de Turing diseñadas como aceptadoras de lenguajes se pueden convertir, sin alterar el lenguaje aceptado, en máquinas con un único estado de aceptación. Esto se puede hacer porque en la definición del modelo estándar se ha exigido que una MT siempre se detenga cuando la unidad de control ingresa en un estado de aceptación. Por consiguiente, al diseñar una MT sólo es necesario un estado de aceptación. De manera concreta, si una MT tiene varios estados de aceptación, podemos modificarla cambiando cada transición de la forma $\delta(q, a) = (p, b, D)$, donde $a, b \in \Gamma$ y p es un estado de aceptación original, por la transición $\delta(q, a) = (q_f, b, D)$, en la que q_f es el único estado de aceptación de la nueva máquina.

De esta manera, se puede considerar que toda MT tiene un único estado inicial y un único estado de aceptación. Este es un hecho importante para la codificación binaria de máquinas de Turing (sección 7.1).

6.5.2. Máquina de Turing con cinta dividida en pistas

En el modelo multi-pista, la cinta está dividida en un número finito k de pistas, como se muestra en la siguiente figura:



La función de transición adquiere la siguiente forma:

$$\delta(q, (a_1, a_2, a_3, \dots, a_k)) = (p, (b_1, b_2, b_3, \dots, b_k), D)$$

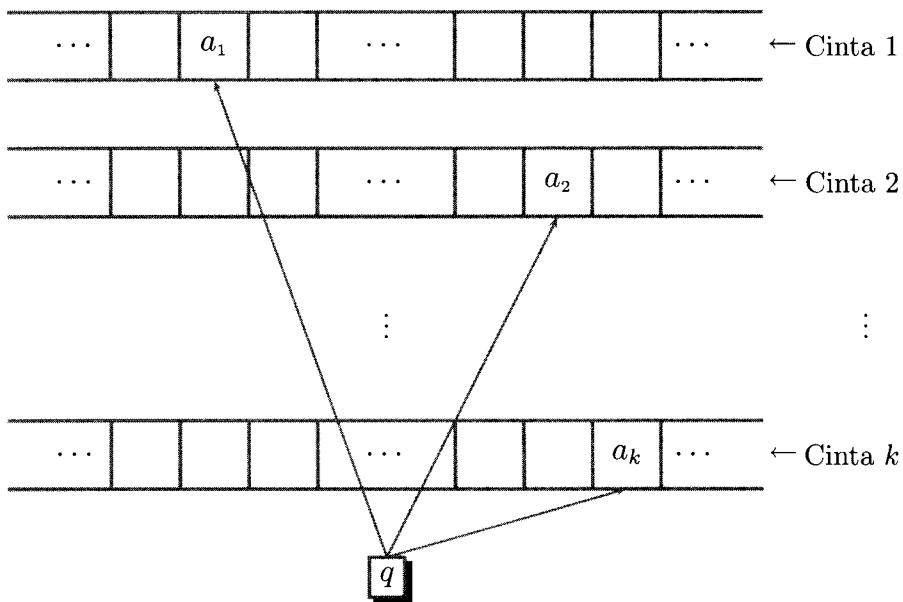
donde los a_i y los b_i son símbolos del alfabeto de cinta Γ y D es \leftarrow , \rightarrow ó $-$. En un paso computacional, la unidad de control cambia simultáneamente el contenido de las k pistas de la celda escaneada y realiza luego uno de los desplazamientos \rightarrow , \leftarrow ó $-$.

Simulación. Las máquinas de Turing que actúan sobre una cinta dividida en k pistas aceptan los mismos lenguajes que las MT estándares. Para concluir tal afirmación, basta considerar el modelo multi-pistas como una MT normal en la que el alfabeto de cinta está formado por el conjunto de k -uplas (s_1, s_2, \dots, s_k) , donde los $s_i \in \Gamma$. Es decir, el nuevo alfabeto de cinta es el producto cartesiano $\Gamma^k = \Gamma \times \Gamma \times \dots \times \Gamma$ (k veces).

Ejemplo Las pistas se usan por lo general para señalar con “marcas” o “marcadores” ciertas posiciones en la cinta. La MT diseñada en el ejemplo de la sección 6.1 para aceptar el lenguaje $L = \{a^i b^i c^i : i \geq 1\}$ utiliza implícitamente la idea de marcadores: reemplazar las *as* por *Xs*, las *bs* por *Ys* y las *cs* por *Zs* no es otra cosa que colocar las marcas *X*, *Y* y *Z* en las posiciones deseadas. Estas marcas se pueden colocar en una pista diferente, sin necesidad de sobre-escribir los símbolos de la cadena de entrada.

6.5.3. Máquina de Turing con múltiples cintas

En el modelo multi-cintas hay k cintas diferentes, cada una dividida en celdas o casillas, como se muestra en la figura que aparece en la parte superior de la página siguiente.



Inicialmente, la cadena de entrada se coloca en la primera cinta y las demás cintas están llenas de blancos. En un paso computacional, la unidad de control cambia el contenido de la casilla escaneada en cada cinta y realiza luego uno de los desplazamientos \rightarrow , \leftarrow o $-$. Esto se hace de manera independiente en cada cinta; la unidad de control tiene entonces k “visores” que actúan independientemente en cada cinta. La función de transición tiene la siguiente forma:

$$\delta(q, (a_1, a_2, a_3, \dots, a_k)) = (p, (b_1, D_1), (b_2, D_2), (b_3, D_3), \dots, (b_k, D_k)),$$

donde los a_i y los b_i son símbolos del alfabeto de cinta Γ , y cada D_i es un desplazamiento \rightarrow , \leftarrow ó $-$.

Simulación. A pesar de que el modelo multi-cinta parece, a primera vista, más poderoso que el modelo estándar, resulta que una MT con múltiples cintas se puede simular con una MT multi-pista. A continuación bosquejaremos el procedimiento de simulación.

Una MT con k cintas se simula con una MT que actúa sobre una única cinta dividida en $2k + 1$ pistas. Cada cinta de la máquina multi-cinta da lugar a dos pistas en la máquina simuladora: la primera simula la cinta propiamente dicha y la segunda tiene todas sus celdas en blanco, excepto una, marcada con un símbolo especial X , que indica la posición actual del visor de la máquina original en dicha cinta. La pista adicional de la máquina

simuladora se utiliza para marcar, con un símbolo especial Y , las posiciones más a la izquierda y más a la derecha de la unidad de control en la máquina original. Para simular un solo paso computacional, la nueva máquina requiere hacer múltiples recorridos a izquierda y a derecha, actualizando el contenido de las pistas y la posición de los marcadores X y Y .

Ejemplo Diseñar una MT con dos cintas que acepte el lenguaje $L = \{a^i b^i c^i : i \geq 0\}$.

Solución. Una MT con dos cintas es más fácil de diseñar que la MT estándar presentada en el segundo ejemplo de la sección 6.1. La idea es copiar en la segunda cinta una X por cada a leída y, al aparecer las b s, avanzar hacia la derecha en la primera cinta y hacia la izquierda en la segunda. Al aparecer las c s se avanza hacia la derecha en ambas cintas. Si la cadena de entrada tiene la forma $a^i b^i c^i$, se detectará simultáneamente el símbolo en blanco b en ambas cintas.

La función de transición requerida para implementar esta idea es:

$$\begin{aligned}\delta(q_0, (a, \text{b})) &= (q_1, (a, \rightarrow), (X, \rightarrow)), \\ \delta(q_1, (a, \text{b})) &= (q_1, (a, \rightarrow), (X, \rightarrow)), \\ \delta(q_1, (b, \text{b})) &= (q_2, (b, -), (\text{b}, \leftarrow)), \\ \delta(q_2, (b, X)) &= (q_2, (b, \rightarrow), (X, \leftarrow)), \\ \delta(q_2, (c, \text{b})) &= (q_3, (c, -), (\text{b}, \rightarrow)), \\ \delta(q_3, (c, X)) &= (q_3, (c, \rightarrow), (X, \rightarrow)), \\ \delta(q_3, (\text{b}, \text{b})) &= (q_4, (\text{b}, -), (\text{b}, -)), \\ \delta(q_0, (\text{b}, \text{b})) &= (q_4, (\text{b}, -), (\text{b}, -)).\end{aligned}$$

Se han utilizado cuatro estados; q_4 es el único estado de aceptación.

6.5.4. Máquinas de Turing no-deterministas (MTN)

En el modelo no-determinista se permite que, en un paso computacional, la unidad de control escoja aleatoriamente entre varias transiciones posibles. La función δ tiene la siguiente forma:

$$\delta(q, a) = \{(p_1, b_1, D_1), (p_2, b_2, D_2), \dots, (p_k, b_k, D_k)\}$$

donde los a_i y los b_i son símbolos del alfabeto de cinta Γ , los p_i son estados y cada D_i es un desplazamiento \rightarrow , \leftarrow ó $-$.

La noción de aceptación en una MTN es similar a la de los modelos no-deterministas de autómatas considerados antes: una cadena de entrada

w es aceptada si existe *por lo menos un cálculo*, a partir de la configuración inicial q_0w , que termine en la configuración u_1pu_2 , con $p \in F$. Como antes, los cálculos en los cuales la máquina se detiene en un estado de no aceptación, o en los cuales no se detiene, son irrelevantes para la aceptación.

Simulación. Una MTN no tiene mayor poder computacional que una MT estándar, es decir, una MTN se puede simular con una MT, como lo explicaremos a continuación.

Sea M una MTN; se bosquejará la construcción de una MT M' tal que $L(M) = L(M')$. Sea n el número máximo de transiciones en los conjuntos $\delta(q, a)$, considerando todo símbolo $a \in \Gamma$ y todo estado $q \in Q$. Para cada $a \in \Gamma$ y $q \in Q$, las transiciones contenidas en $\delta(q, a)$ se pueden enumerar entre 1 y n . Si hay menos de n transiciones en un $\delta(q, a)$ particular, se repite arbitrariamente una de ellas hasta completar n . De esta manera, cada una de las transiciones $\delta(q, a)$ se puede considerar como un conjunto de n opciones indexadas:

$$\delta(q, a) = \left\{ \underbrace{(p_1, b_1, D_1)}_1, \underbrace{(p_2, b_2, D_2)}_2, \dots, \underbrace{(p_k, b_k, D_k)}_n \right\} \quad \leftarrow \text{índice}$$

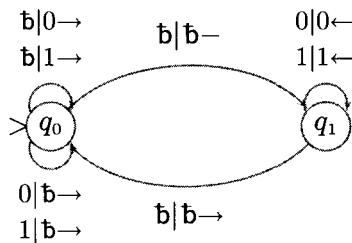
En algunos $\delta(q, a)$ habrá opciones repetidas, pero esta repetición no altera el lenguaje aceptado.

La MT M' que simulará a M tendrá tres cintas. La primera cinta almacena la entrada, la segunda genera de forma sistemática todas las secuencias finitas de números entre 1 y n : primero las secuencias de longitud 1, luego las secuencias de longitud 2, luego las de longitud 3 y así sucesivamente (para esto se puede usar una subrutina como la construida en el ejercicio ③ de la sección 6.4).

Para cada secuencia $i_1i_2 \cdots i_k$ generada en la cinta 2, M' copia la cadena de entrada sobre la cinta 3 y simula la computación de k pasos que hace sobre ella la máquina original M , utilizando en el paso j la opción con índice j en el $\delta(q, a)$ correspondiente. Si M' se detiene en un estado de aceptación, entonces tanto M como M' aceptan la entrada. Si el cálculo simulado no termina en la aceptación, M' borra el contenido de las cintas 2 y 3, genera otra secuencia en la cinta 2, copia de nuevo la entrada en la cinta 3 e inicia una nueva simulación.

Si la entrada es aceptada por la máquina original M , existirá un cálculo que se detiene en un estado de aceptación y, eventualmente, M' encontrará y simulará tal cálculo.

Ejemplo La siguiente MT M no-determinista genera todas las cadenas binarias (cadenas de ceros y unos). M “escoge” primero una cadena binaria w por medio de las opciones no-deterministas $b \mid 0 \rightarrow$ y $b \mid 1 \rightarrow$ y retrocede hacia el primer símbolo de w , retornando al estado q_0 . Luego borra todos los símbolos de la cadena generada w y procede a generar una nueva.



Siempre que M retorna al estado q_0 hay una cadena binaria escrita en la cinta. Todas las cadenas binarias serán eventualmente generadas, aleatoriamente. En contraste, la MT determinista del segundo ejemplo de la sección 6.4 genera estas cadenas sistemáticamente (en orden lexicográfico).

Para generar de manera no-determinista todas las cadenas de Σ^* , donde Σ es un alfabeto cualquiera, bastan también dos estados (ejercicio ③).

Ejercicios de la sección 6.5

- ① Diseñar máquinas de Turing multi-pistas que acepten los siguientes lenguajes. Escribir explícitamente la idea utilizada en el diseño.
 - (i) $L = \{a^i b^i c^i : i \geq 0\}$.
 - (ii) $L = \{a^i b^{2i} a^i : i \geq 0\}$.
- ② Diseñar máquinas de Turing multi-cintas que acepten los siguientes lenguajes (que no son LIC, según los ejercicios ⑥ y ⑦ de la sección 4.13). Escribir explícitamente la idea utilizada en el diseño.
 - !(i) $L = \{ww : w \in \{0, 1\}^*\}$.
 - !(ii) $L = \{a^i : i \text{ es un cuadrado perfecto}\}$.
- ③ Sea $\Sigma = \{s_1, \dots, s_m\}$ un alfabeto cualquiera. Diseñar una máquina de Turing no-determinista que genere todas las cadenas de Σ^* .

6.6. Simulación de autómatas por medio de máquinas de Turing

Intuitivamente, parece claro que un autómata determinista AFD se puede simular con una MT estándar y una autómata con pila se puede simular con una MT que actúa sobre dos cintas. En esta sección presentaremos los detalles concretos de estas simulaciones. Concluiremos que tanto los lenguajes regulares como los independientes del contexto son recursivos.

6.6.1. Simulación de autómatas

Un autómata (modelo AFD, AFN o AFN- λ) se puede simular con una máquina de Turing: primero se convierte en un AFD $M = (Q, q_0, F, \Sigma, \delta)$ equivalente, usando las técnicas del Capítulo 2. Luego se construye una MT M' tal que $L(M) = L(M')$ añadiendo un nuevo estado q_f , que será el único estado de aceptación de M' , y transiciones desde los estados de F hasta q_f , en presencia del símbolo blanco b . Es decir, $M' = (Q', q_0, F', \Sigma, \Gamma, \text{b}, \delta')$ donde

$$\begin{aligned} Q' &= Q \cup \{q_f\}, \quad q_f \text{ es un estado nuevo,} \\ \Gamma &= \Sigma \cup \{\text{b}\}, \\ F' &= \{q_f\}, \\ \delta'(q, s) &= (\delta(q, s), s, \rightarrow), \quad \text{para } q \in Q, s \in \Sigma, \\ \delta'(q, \text{b}) &= (q_f, \text{b}, \leftarrow), \quad \text{para todo } q \in F. \end{aligned}$$

Nótese que la MT M' así construida se detiene con cualquier entrada w . Puesto que los lenguajes regulares son precisamente los aceptados por los AFD, hemos establecido el siguiente teorema.

6.6.1 Teorema. *Todo lenguaje regular es recursivo.*

6.6.2. Simulación de autómatas con pila

Sea $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \Delta)$ un autómata con pila (modelo AFPD o AFPN). Para simular a M se puede construir una MT M' que actúe sobre dos cintas: la primera cinta simula la cinta de entrada y la segunda simula la pila del autómata. M' será una MT determinista o no-determinista dependiendo de si el autómata dado es un AFPD o un AFPN. Se define $M' = (Q', q_0, F', \Sigma, \Gamma', \text{b}, \delta)$ por medio de

$$Q' = Q \cup \{q_f\}, \quad \text{donde } q_f \text{ es un estado nuevo,}$$

$$\begin{aligned}\Gamma' &= \Sigma \cup \Gamma \cup \{\mathbf{b}\}, \\ F' &= \{q_f\}.\end{aligned}$$

La MT M' coloca inicialmente el símbolo inicial de pila sobre la cinta 2:

$$\delta(q_0, (s, \mathbf{b})) = (q_0, (s, z_0), (-, -)), \quad \text{para cualquier } s \in \Sigma.$$

Una transición de la forma $\Delta(q, a, s) = (p, \gamma)$ se simula con varias transiciones: se trata de lograr que M sobre-escriba el símbolo s en la cinta 2, que representa el tope de la pila, por la cadena γ . Por ejemplo, la transición $\Delta(q, a, s) = (p, a_1 a_2 a_3)$ se simula añadiendo un estado auxiliar q_e junto con las transiciones

$$\begin{aligned}\delta(q, (a, s)) &= (q_e, (a, -), (a_3, \rightarrow)), \\ \delta(q_e, (a, \mathbf{b})) &= (q_e, (a, -), (a_2, \rightarrow)), \\ \delta(q_e, (a, \mathbf{b})) &= (p, (a, \rightarrow), (a_1, -)).\end{aligned}$$

Para no alterar el lenguaje aceptado, cada transición de este tipo requiere la adición de un estado especial q_e diferente. Una transición λ del autómata M , $\Delta(q, \lambda, s) = (p, \gamma)$, se simula de manera similar: la posición de la unidad de control y el contenido de la primera cinta no cambian durante el procedimiento.

Finalmente, se debe añadir la transición

$$\delta(q, (\mathbf{b}, s)) = (q_f, (\mathbf{b}, s), (-, -)), \quad \text{para todo } s \in \Gamma \text{ y todo } q \in F.$$

Lo anterior muestra que el comportamiento del autómata M se simula completamente con la MT M' y se tiene que $L(M) = L(M')$.

Nótese que la MT así construida no necesariamente se detiene con todas las cadenas de entrada ya que el autómata original M puede ingresar a un bucle infinito con transiciones λ que inserten y extraigan indefinidamente símbolos en la pila. Por consiguiente, para concluir que todo LIC es recursivo es necesario restringir primero las transiciones del autómata. En la demostración del siguiente teorema se indica cómo hacerlo.

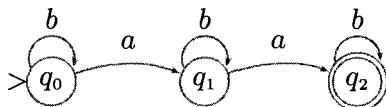
6.6.2 Teorema. *Todo LIC es un lenguaje recursivo.*

Demostración. Sabemos que un LIC L dado se puede generar con una GIC G en FNC en la que la variable inicial no sea recursiva. En G no hay transiciones λ , excepto posiblemente $S \rightarrow \lambda$. A partir de G podemos construir un AFPN M tal que $L(G) = L(M) = L$ usando el procedimiento

del Teorema 5.4.1, y luego una MT M' que simule a M , en la forma indicada en la presente sección. Se puede observar que M y M' siempre se detienen al procesar una cadena de entrada cualquiera: no ingresan nunca en bucles infinitos debido al tipo restringido de las transiciones que surgen de la gramática G . \square

Ejercicios de la sección 6.6

- ① Simular el siguiente AFD por medio de una MT M y hacer el diagrama de transiciones de la máquina M .

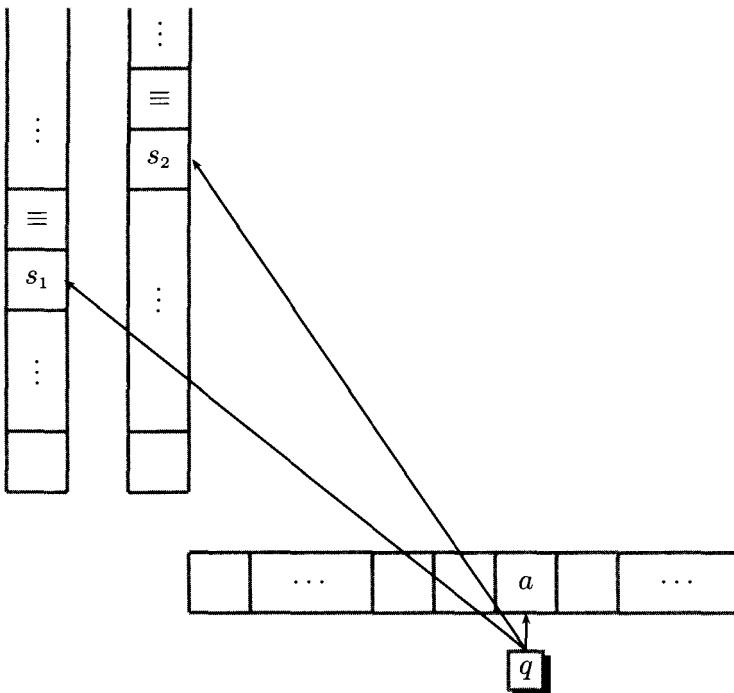


- ② Simular por medio de una MT M el autómata con pila del primer ejemplo de la sección 5.1, el cual acepta el lenguaje $\{a^i b^i : i \geq 1\}$ sobre el alfabeto $\Sigma = \{a, b\}$, y cuya función de transición es:

$$\begin{aligned}\Delta(q_0, a, z_0) &= (q_0, Az_0), \\ \Delta(q_0, a, A) &= (q_0, AA), \\ \Delta(q_0, b, A) &= (q_1, \lambda), \\ \Delta(q_1, b, A) &= (q_1, \lambda), \\ \Delta(q_1, \lambda, z_0) &= (q_2, z_0).\end{aligned}$$

6.7. Autómatas con dos pilas (AF2P) \ddagger

En esta sección mostraremos que el modelo de autómata con dos pilas (AF2P) es también equivalente a la máquina de Turing. Un autómata con dos pilas es esencialmente un AFPD, tal como se definió en el capítulo 5, con la adición de una pila más. Las pilas tienen la misma restricción que antes: el autómata sólo tiene acceso al símbolo que está en el tope de cada pila. Un paso computacional depende del estado actual de la unidad de control, del símbolo escaneado en la cinta y de los dos topes de pila, como se muestra en la siguiente gráfica:



Podríamos también definir autómatas con k pilas, $k \geq 1$, pero tal modelo no aumenta la capacidad computacional que se consigue con dos pilas. Para limitarnos al modelo determinista debemos dotar al autómata de una faceta más: un símbolo auxiliar, $\$$, llamado **marcador final de entrada**, que se escribe al final de cada entrada y le sirve al autómata para saber cuándo ha consumido completamente la cadena de entrada. El símbolo $\$$ no forma parte del alfabeto de entrada Σ .

Una transición en un AF2P tiene la forma

$$\Delta(q, a, s_1, s_2) = (q', \gamma_1, \gamma_2),$$

la cual tiene el siguiente significado: en presencia del símbolo a en la cinta de entrada, la unidad de control pasa del estado q al estado q' y se mueve a la derecha. Además, borra el símbolo s_1 que está en el tope de la primera pila y lo sobre-escribe por la cadena γ_1 , y borra el símbolo s_2 que está en el tope de la segunda pila y lo sobre-escribe por γ_2 . La unidad de control pasa a escanear los nuevos topes de cada pila. Las cadenas γ_1 y γ_2 pertenecen a Γ^* , siendo Γ el alfabeto de pila.

También se permiten transiciones λ o transiciones espontáneas,

$$\Delta(q, \lambda, s_1, s_2) = (q', \gamma_1, \gamma_2).$$

Con esta transición, el símbolo sobre la cinta de entrada no se procesa y la unidad de control no se mueve a la derecha, pero los topes de pila s_1 y s_2 son reemplazados por las cadenas γ_1 y γ_2 , respectivamente. Para garantizar el determinismo, $\Delta(q, a, s_1, s_2)$ y $\Delta(q, \lambda, s_1, s_2)$, con $a \in \Sigma$, no pueden estar simultáneamente definidos. Las transiciones λ en un AF2P permiten que el autómata cambie el contenido de las pilas sin procesar (o consumir) símbolos sobre la cinta de entrada.

Inicialmente cada pila contiene únicamente el marcador z_0 en el fondo, y la cadena de entrada se coloca en la cinta de entrada, en la forma usual. Nótese que sólo es necesario exigir que la cinta de entrada sea infinita en una dirección ya que la unidad de control no puede nunca retornar a la izquierda.

Ejemplo Diseñar un AF2P que acepte el lenguaje $L = \{0^i 1^{2i} 0^i : i \geq 0\}$. Según el ejercicio ③ de la sección 4.13, este lenguaje no es LIC y, por lo tanto, no puede ser aceptado por un autómata con una sola pila.

Solución. La idea es copiar los ceros iniciales de la cadena de entrada en la primera pila y el doble de ellos en la segunda. Al aparecer los unos en la cadena de entrada, se van borrando los ceros de la segunda pila, sin alterar el contenido de la primera pila. Al aparecer la última cadena de ceros en la entrada, se borran los ceros acumulados en la primera pila. Si la entrada tiene la forma $0^i 1^{2i} 0^i$, se detectará el marcador inicial de entrada \$ simultáneamente con los marcadores de fondo z_0 en ambas pilas. Para implementar esta idea utilizamos cuatro estados; q_0 es el estado inicial y q_3 el único estado de aceptación.

En detalle, si la entrada tiene i ceros iniciales, se acumulan i ceros en la primera pila y $2i$ ceros en la segunda, por medio de las dos instrucciones

$$\begin{aligned}\Delta(q_0, 0, z_0, z_0) &= (q_0, 0z_0, 00z_0), \\ \Delta(q_0, 0, 0, 0) &= (q_0, 00, 000).\end{aligned}$$

Se borra luego un 0 en la segunda pila por cada 1 leído en la cinta, por medio de las transiciones

$$\begin{aligned}\Delta(q_0, 1, 0, 0) &= (q_1, 0, \lambda), \\ \Delta(q_1, 1, 0, 0) &= (q_1, 0, \lambda).\end{aligned}$$

Al vaciar así la segunda pila, el contenido de la primera no se altera. Finalmente, se borra un 0 en la primera pila por cada 0 leído en la cinta, con las transiciones

$$\Delta(q_1, 0, 0, z_0) = (q_2, \lambda, z_0),$$

$$\Delta(q_2, 0, 0, z_0) = (q_2, \lambda, z_0).$$

Al encontrar $\$$ en la cinta de entrada y z_0 en el fondo de cada pila, se ingresa al estado de aceptación q_3 :

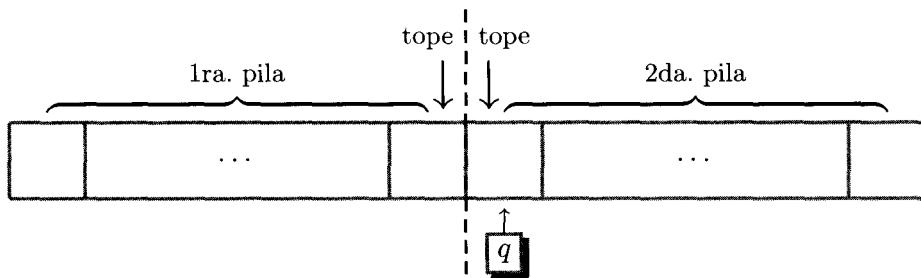
$$\Delta(q_2, \$, z_0, z_0) = (q_3, z_0, z_0).$$

Hay que añadir también la transición $\Delta(q_0, \$, z_0, z_0) = (q_3, z_0, z_0)$ para aceptar la cadena vacía.

El siguiente teorema establece que una MT estándar se puede simular con un AF2P.

6.7.1 Teorema. *Dada una máquina de Turing M , se puede construir un autómata con dos pilas M' que acepte el mismo lenguaje que M .*

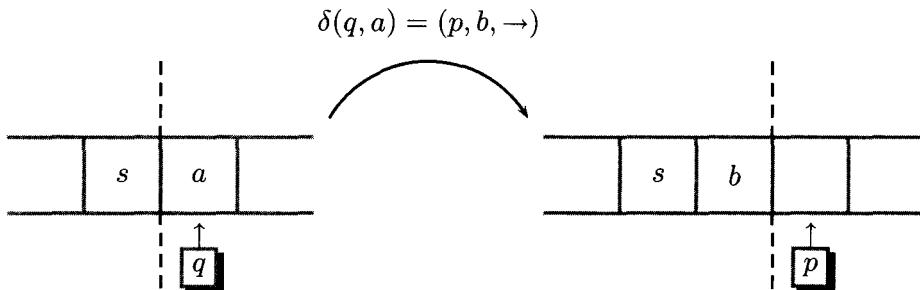
Demostración. Sea M una MT estándar dada, con función de transición δ . La idea básica de la simulación es hacer que, en cada momento, la primera pila de M' contenga la parte ubicada a la izquierda de la unidad de control de M , y la segunda pila contenga la parte ubicada encima y a la derecha. La siguiente gráfica muestra la cinta de M y las dos pilas de la simulación. En cada momento las pilas contienen solo un número finito de casillas no vacías.



Inicialmente, M' copia la cadena de entrada w en la primera pila y luego la traslada de ésta a la segunda pila (el marcador final de entrada $\$$ le permite al autómata saber cuándo ha terminado de copiar toda la entrada en la primera pila). Al terminar esta operación, la primera pila queda vacía, excepto por el marcador de fondo z_0 , y la segunda pila contiene toda la cadena de entrada. La unidad de control de M' está escaneando el tope de la segunda pila que es exactamente el primer símbolo de w . Nótese que para hacer este doble traslado se necesita solamente un número finito de estados, número que depende del alfabeto de entrada pero no de la longitud de w .

El resto de la simulación lo realiza M' por medio de transiciones λ mientras escanea el marcador final de entrada $\$$ sobre la cinta. A continuación indicamos cómo simula M' las transiciones de M .

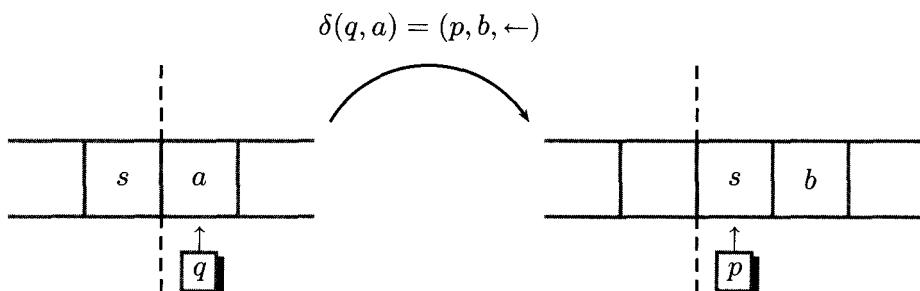
Desplazamiento a la derecha: la transición $\delta(q, a) = (p, b, \rightarrow)$ de M es simulada por M' con transiciones de la forma $\Delta(q, \lambda, s, a) = (p, sb, \lambda)$, por cada símbolo s (s es el símbolo ubicado a la izquierda de a). La siguiente gráfica ilustra la situación:



Hay dos excepciones por considerar:

1. Si M realiza la transición $\delta(q, a) = (p, b, \rightarrow)$ y la primera pila sólo contiene el marcador de fondo z_0 , M' ejecuta la acción $\Delta(q, \lambda, z_0, a) = (p, z_0, \lambda)$.
2. Si M' está escaneando el marcador de fondo z_0 en la segunda pila, lo que significa que M está leyendo una casilla en blanco y usando la transición $\delta(q, b) = (p, b, \rightarrow)$, M' debe ejecutar la acción $\Delta(q, \lambda, s, z_0) = (p, sb, z_0)$.

Desplazamiento a la izquierda: la transición $\delta(q, a) = (p, b, \leftarrow)$ de M es simulada por M' con transiciones de la forma $\Delta(q, \lambda, s, a) = (p, \lambda, sb)$, por cada símbolo s (s es el símbolo ubicado a la izquierda de a). La siguiente gráfica ilustra la situación:



Aquí hay que tener en cuenta una excepción: si s es el marcador de fondo z_0 , M' debe ejecutar la acción $\Delta(q, \lambda, z_0, a) = (p, z_0, \text{bb})$. Esto significa que la primera pila sigue vacía y M' queda escaneando el símbolo blanco b en el tope de la segunda pila. \square

Ejercicios de la sección 6.7

- ① Mostrar cómo se puede simular un AF2P con una MT multi-cintas.
- ② Diseñar autómatas con dos pilas que acepten los siguientes lenguajes:
 - (i) $L = \{a^i b^j c^i d^j : i, j \geq 1\}$.
 - (ii) $L = \{a^{2i} b^{3i} : i \geq 1\}$.
 - (iii) El lenguaje de las cadenas sobre el alfabeto $\{0, 1\}$ que tienen el doble número de ceros que de unos.

6.8. Propiedades de clausura de los lenguajes RE y de los lenguajes recursivos

En esta sección presentaremos algunas propiedades de clausura de los lenguajes recursivos y de los RE; en los ejercicios del final de la sección aparecen otras propiedades similares.

6.8.1 Teorema. 1. *El complemento de un lenguaje recursivo también es recursivo.*

2. *La unión de dos lenguajes recursivos es recursivo.*
3. *La unión de dos lenguajes RE es RE.*

Demostración.

1. Sea L un lenguaje recursivo aceptado por la MT M . La máquina M se detiene con cualquier entrada, ya sea en un estado de aceptación o en uno de rechazo. Se puede construir una MT M' que acepte a \bar{L} , haciendo que los estados de aceptación de M dejen de serlo en M' . De esta forma, las cadenas aceptadas por M serán rechazadas por M' . Adicionalmente, desde los estados de no-aceptación en los cuales M se detiene, definimos transiciones a un estado (nuevo) de aceptación de M' . De esta forma, las cadenas no aceptadas por M sí serán aceptadas por M' . Por consiguiente, $L(M') = \bar{L}$.

2. Sean L_1 y L_2 dos lenguajes aceptados por las MT M_1 y M_2 , respectivamente. Para demostrar las partes 2 y 3 del presente teorema podríamos proceder como se hizo en el Teorema de Kleene al demostrar que la unión de dos lenguajes regulares es regular; esto es, basta unir las máquinas M_1 y M_2 en paralelo por medio de transiciones espontáneas (transiciones λ) desde un nuevo estado inicial. Recuérdese que las transiciones espontáneas en una máquina de Turing adquieren la forma $\delta(q, s) = (q', s, -)$. No obstante, las máquinas así construidas son no-deterministas. Resulta más conveniente, pensando en aplicaciones posteriores, diseñar MT multi-cintas deterministas para aceptar la unión de dos lenguajes.

Supóngase entonces que L_1 y L_2 son recursivos, es decir, M_1 y M_2 se detienen con toda entrada. Construimos una MT M con dos cintas que simule a M_1 en la primera cinta y a M_2 en la segunda. M procesa inicialmente la entrada en la primera cinta; si M_1 acepta, M también aceptará. Si M_1 se detiene en un estado de no-aceptación, M procederá a procesar la entrada en la segunda cinta, simulando a M_2 . Si M_2 acepta, M también aceptará pero si M_2 se detiene en un estado de rechazo, M también se detendrá y no aceptará. Puesto que tanto M_1 como M_2 siempre se detienen, $L(M) = L_1 \cup L_2$.

3. El procedimiento del numeral anterior ya no sirve en este caso porque es posible que M_1 o M_2 nunca se detengan al procesar una entrada particular. En lugar de una simulación consecutiva de las máquinas dadas, se necesita que M simule *simultáneamente* a M_1 y a M_2 . Para lograrlo utilizamos también dos cintas, la primera para simular a M_1 y la segunda para simular a M_2 . En cada paso computacional, M simula un paso de M_1 y uno de M_2 .

Resulta muy instructivo presentar explícitamente una máquina M que implemente la anterior idea intuitiva. Aparte de los estados especiales q_e y q_f , definidos más adelante, los estados de M son parejas de la forma (p, q) , donde p es un estado de M_1 y q un estado de M_2 . Cada par de transiciones

$$\begin{aligned}\delta_1(q_1, s_1) &= (p_1, r_1, D_1) && (\text{transición de } M_1), \\ \delta_2(q_2, s_2) &= (p_2, r_2, D_2) && (\text{transición de } M_2),\end{aligned}$$

da lugar a la siguiente transición en la máquina M :

$$\delta((q_1, q_2), (s_1, s_2)) = ((p_1, p_2), (r_1, D_1), (r_2, D_2)).$$

Si alguno de los estados q_1 ó q_2 es un estado de aceptación de M_1 ó M_2 , respectivamente, hacemos que M ingrese y se detenga en un estado q_f (estado nuevo), que será el único estado de aceptación de M . Esto se consigue con la transición

$$\delta((q_1, q_2), (s_1, s_2)) = (q_f, (s_1, -), (s_2, -)).$$

También hay que permitir que M siga operando en una de las cintas incluso si se ha detenido en la otra en un estado de no aceptación. Por ejemplo, si $\delta_1(q_1, s_1)$ no está definida en M_1 , pero $\delta_2(q_2, s_2) = (p_2, r_2, D_2)$ es una transición de M_2 , definimos

$$\delta((q_1, q_2), (s_1, s_2)) = ((q_e, p_2), (s_1, -), (s_2, D_2)),$$

donde q_e es un estado especial (nuevo). Por medio de esta transición, cuando M ingrese en el estado q_e en alguna de las cintas, se detendrá en dicha cinta pero continúa la simulación en la otra. Si M ingresa al estado (q_e, q_e) , se detendrá sin aceptar. \square

El siguiente teorema establece una importante conexión entre las nociones de lenguaje recursivo y lenguaje RE, la cual se puede demostrar con la técnica de simulación simultánea de dos MT usada en el Teorema 6.8.1. Encontraremos aplicaciones útiles de este resultado en el Capítulo 7.

6.8.2 Teorema. *Un lenguaje L es recursivo si y sólo si L y su complemento \bar{L} son RE.*

Demostración. En la dirección izquierda a derecha, la conclusión es directa: si L es recursivo, obviamente es RE. \bar{L} es también recursivo (parte 1 del Teorema 6.8.1) y por lo tanto RE.

La parte esencial de este teorema es la otra dirección: si L y \bar{L} son RE, entonces L debe ser recursivo. Para verlo, partimos de dos máquinas M_1 y M_2 que acepten a L y \bar{L} , respectivamente. Construimos luego una MT M que simule simultáneamente a M_1 y M_2 , tal como se hizo en la parte 3 del Teorema 6.8.1. Puesto que para una entrada w sólo hay dos posibilidades, $w \in L$ o $w \in \bar{L}$, entonces la máquina M eventualmente se detendrá en la cinta 1 o en la cinta 2. En el primer caso, M acepta la entrada y en el segundo caso la rechaza. Como M se detiene con toda entrada, L es recursivo. \square

El siguiente teorema justifica la denominación ‘recursivamente enumerable’ para los lenguajes aceptados por máquinas de Turing.

6.8.3 Teorema. *Para todo lenguaje L aceptado por una MT M , se puede construir una MT M' que enumere secuencialmente las cadenas de L .*

Bosquejo de la demostración. La MT M' tendrá tres cintas. Las cadenas de L serán enumeradas secuencialmente en la cinta 1 usando el símbolo $\&$ como separador: $w_1\&w_2\&w_3\&\dots$. La cinta 2 se usa para generar todas las cadenas de Σ^* en el orden lexicográfico, utilizando la subrutina construida en el ejercicio ③ de la sección 6.4, modificada de tal manera que las cadenas queden separadas entre sí por el separador $\&$. En la cinta 2 también se escribe, a la izquierda de la lista de cadenas, un contador que registra el número de cadenas generadas. La cinta 3 se usa para simular el procesamiento de M sobre las cadenas que se generan en la cinta 2.

Para precisar, M' procede según las siguientes acciones:

Acción 1. M' genera sobre la cinta 2 la primera cadena de Σ^* (o sea, λ) y simula en la cinta 3 un movimiento de la acción de M sobre λ .

Acción 2. M' genera sobre la cinta 2 la segunda cadena de Σ^* y simula, en la cinta 3, dos movimientos de la acción de M sobre dicha cadena, así como un movimiento más del procesamiento de la cadena λ .

Acción i , ($i \geq 1$). M' genera sobre la cinta 2 la i -ésima cadena de Σ^* y simula, en la cinta 3, i movimientos de la acción de M sobre dicha cadena, así como un movimiento más del procesamiento de las $i - 1$ cadenas previamente generadas en la cinta 2. Se incrementa luego, en la cinta 2, el contador i de cadenas generadas (éste es también el contador del número de movimientos simulados para *cada* cadena generada).

Al concluir la acción i se han generado en la segunda cinta las i primeras cadenas de Σ^* y se han simulado, en la cinta 3, los i primeros movimientos que M realiza sobre esas cadenas. Si durante la simulación, alguna cadena es aceptada por M , M' la copia en la cinta 1. Las casillas ocupadas en la cinta 3 por las simulaciones de cadenas ya aceptadas se “tachan” (es decir, se marcan con un símbolo especial), de tal manera que M' no tenga que procesarlas en los pasos subsiguientes.

Para acomodar las simulaciones, cada vez más extensas, puede ser necesario usar en la cinta 3 subrutinas de desplazamiento de cadenas, a izquierda o a derecha. En todo caso, al terminar la acción i , M ha utilizado una porción finita en cada una de las tres cintas. \square

Ejercicios de la sección 6.8

- ① Utilizando razonamientos similares a los del Teorema 6.8.1 demostrar lo siguiente:
 - (i) Si L_1 y L_2 son recursivos, $L_1 \cap L_2$ es recursivo.
 - (ii) Si L_1 y L_2 son RE, $L_1 \cap L_2$ es RE.
 - (iii) Si L_1 y L_2 son recursivos, $L_1 - L_2$ es recursivo.
- ② Sabiendo que existen lenguajes RE no recursivos (lo cual será demostrado en la sección 7.5), demostrar que el complemento de un lenguaje RE no necesariamente es RE.
- !③ Demostrar que tanto los lenguajes recursivos como los lenguajes RE son cerrados para la concatenación. Es decir, si L_1 y L_2 son recursivos, L_1L_2 también es recursivo, y si L_1 y L_2 son RE, L_1L_2 es RE.

Advertencia: no basta unir las máquinas de Turing en serie por medio de transiciones espontáneas, como se hizo en la demostración del Teorema de Kleene; hay que tener en cuenta que una MT puede aceptar cadenas de entrada sin leerlas (consumirlas) completamente.

- !④ Demostrar que tanto los lenguajes recursivos como los lenguajes RE son cerrados para la estrella de Kleene. Es decir, si L es recursivo, L^* también es recursivo, y si L RE, L^* es RE. Ayuda: tener en cuenta la misma advertencia del ejercicio ③.
- !⑤ Sea L un lenguaje recursivo pero no RE, demostrar que para toda MT M que acepte a L hay infinitas cadenas de entrada con las cuales M no se detiene nunca. Ayuda: razonar por contradicción.

6.9. Máquinas de Turing, computadores, algoritmos y la tesis de Church-Turing

Si bien la máquina de Turing antecedió en varias décadas a la implementación física de los computadores actuales, ha resultado ser un modelo muy conveniente para representar “lo computable” (lo que es capaz de hacer *cualquier* dispositivo físico de computación secuencial).

6.9.1. Máquinas de Turing y algoritmos

Según nuestra experiencia en las secciones anteriores, diseñar una MT es muy similar a escribir un programa computacional ya que la función de transición de una MT no es otra cosa que un conjunto de instrucciones. Se establece así una conexión intuitiva directa entre máquinas de Turing y algoritmos. La declaración conocida como “tesis de Church-Turing” afirma que dicha conexión es en realidad una equivalencia.

6.9.1. *Tesis de Church-Turing.* *Todo algoritmo puede ser descrito por medio de una máquina de Turing.*

En su formulación más amplia, la tesis de Church-Turing abarca tanto los algoritmos que producen una salida para cada entrada como aquéllos que no terminan (ingresan en bucles infinitos) para algunas entradas.

Para apreciar su significado y su alcance, hay que enfatizar que la Tesis de Church-Turing no es un enunciado matemático susceptible de demostración, ya que involucra la noción intuitiva de *algoritmo*. En otras palabras, la tesis no se puede demostrar. Se podría refutar, no obstante, exhibiendo un procedimiento efectivo, que todo el mundo acepte que es un verdadero algoritmo y que no pueda ser descrito por una máquina de Turing. Pero tal refutación no se ha producido hasta la fecha; de hecho, la experiencia acumulada durante décadas de investigación ha corroborado una y otra vez la tesis de Church-Turing.

Hay dos hechos más que contribuyen a apoyar la tesis:

1. La adición de recursos computacionales a las máquinas de Turing (múltiples pistas o cintas, no determinismo, etc) no incrementa el poder computacional del modelo básico. Esto es un indicativo de que la máquina de Turing, no sólo es extremadamente flexible, sino que representa el límite de lo que un dispositivo de computación secuencial puede hacer.
2. Todos los modelos o mecanismos computacionales propuestos para describir formalmente la noción de algoritmo han resultado ser equivalentes a la máquina de Turing, en el sentido de que lo que se puede hacer con ellos también se puede hacer con una MT adecuada, y viceversa. Entre los modelos de computación secuencial equivalentes a la máquina de Turing podemos citar:
 - Las funciones parciales recursivas (modelo de Gödel y Kleene, 1936).

- El cálculo- λ (modelo de Church, 1936).
- Sistemas de deducción canónica (modelo de Post, 1943).
- Algoritmos de Markov (modelo de Markov, 1951).
- Las gramáticas no-restringidas¹ (modelo de Chomsky, 1956).
- Las máquinas de registro (modelo de Shepherdson-Sturgis, 1963).

En el próximo capítulo tendremos la oportunidad de usar la tesis de Church-Turing en situaciones concretas.

6.9.2. Máquinas de Turing y computadores

Hagamos un bosquejo —bastante superficial e incompleto— de cómo los computadores comunes y las máquinas de Turing se pueden simular entre sí.

En primer lugar, una MT está determinada por un conjunto finito de instrucciones (su función de transición) y está definida con referencia a alfabetos finitos. Por lo tanto, se puede programar un computador para que simule una MT dada, siempre y cuando se disponga de una capacidad de almacenamiento (memoria, discos, etc) *potencialmente* infinita, que corresponda a la cinta infinita de una MT. Teóricamente, por lo menos, esta simulación es concebible y plausible.

Recíprocamente, el modelo multi-cintas de máquina de Turing se puede usar para simular el funcionamiento de un computador típico. Se utilizaría una cinta para simular la memoria principal del computador, otra para las direcciones de memoria y un número adicional (pero finito) de cintas para simular los discos de almacenamiento presentes en un computador real. La máquina de Turing así construida es básicamente la llamada *Máquina de Turing universal* que se presentará detalladamente en la sección 7.2.

¹Se puede demostrar que los lenguajes RE son *exactamente* los lenguajes generados por las gramáticas de tipo 0, o no-restringidas, mencionadas en la sección 4.1.

Problemas indecidibles

La importancia de un modelo simple pero capaz de emular cualquier dispositivo de computación —como lo es la máquina de Turing— radica no sólo en que permite estudiar *lo que puede hacer* sino también *lo que no puede hacer* una máquina de cómputo, aun si dispone de recursos inagotables, como tiempo y espacio de almacenamiento ilimitados (es decir, cinta infinita y tiempo de ejecución indefinido). En este capítulo estudiaremos problemas que *ninguna* máquina de Turing puede resolver.

7.1. Codificación y enumeración de máquinas de Turing

Toda MT se puede codificar como una secuencia binaria finita, es decir una secuencia finita de ceros y unos. En esta sección presentaremos una codificación válida para todas las MT que actúen sobre un alfabeto de entrada Σ pre-establecido. Para simplificar la codificación, suponemos que toda MT tiene un único estado inicial, denotado q_1 , y un único estado final, denotado q_2 (en la sección 6.5.1 se mostró que esta modificación siempre se puede hacer, sin alterar los lenguajes aceptados).

El alfabeto de cinta de una MT M es de la forma

$$\Gamma = \{s_1, s_2, \dots, s_m, \dots, s_p\}$$

donde s_1 representa el símbolo blanco \texttt{b} , $\Sigma = \{s_2, \dots, s_m\}$ es el alfabeto de entrada y s_{m+1}, \dots, s_p son los símbolos auxiliares utilizados por M (cada MT utiliza su propia colección finita de símbolos auxiliares). Todos estos símbolos se codifican como secuencias de unos:

<u>Símbolo</u>	<u>Codificación</u>
s_1 (símbolo \bar{b})	1
s_2	11
s_3	111
\vdots	\vdots
s_m	$\underbrace{11 \cdots 1}_{m \text{ veces}}$
\vdots	\vdots
s_p	$\underbrace{11 \cdots 1}_{p \text{ veces}}$

Los estados de una MT, $q_1, q_2, q_3, \dots, q_n$, se codifican también con secuencias de unos:

<u>Estado</u>	<u>Codificación</u>
q_1 (inicial)	1
q_2 (final)	11
\vdots	\vdots
q_n	$\underbrace{11 \cdots 1}_{n \text{ veces}}$

Las directrices de desplazamiento \rightarrow, \leftarrow y $-$ se codifican con 1, 11 y 111, respectivamente. Una transición $\delta(q, a) = (p, b, D)$ se codifica usando ceros como separadores entre los estados, los símbolos del alfabeto de cinta y la directriz de desplazamiento D . Así, la transición $\delta(q_3, s_2) = (q_5, s_3, \rightarrow)$ se codifica como

01110110111110111010

En general, la codificación de una transición cualquiera $\delta(q_i, s_k) = (q_j, s_\ell, D)$ es

$01^i 01^k 01^j 01^\ell 01^t 0$

donde $t = 1$ ó 2 ó 3 , según D sea \rightarrow, \leftarrow ó $-$. Obsérvese que aparecen *exactamente* seis ceros separados por secuencias de unos.

Una MT se codifica escribiendo consecutivamente las secuencias de las codificaciones de todas sus transiciones. Más precisamente, la codificación de una MT M es de la forma

$C_1 C_2 \cdots C_r$

donde las C_i son las codificaciones de las transiciones de M . Puesto que el orden en que se presentan las transiciones de una MT no es relevante,

una misma MT tiene varias codificaciones diferentes. Esto no representa ninguna desventaja práctica o conceptual ya que no se pretende que las codificaciones sean únicas.

Ejemplo Considérese la siguiente MT M que acepta el lenguaje a^+b :

$$\begin{aligned}\delta(q_1, a) &= (q_3, a, \rightarrow) \\ \delta(q_3, a) &= (q_3, a, \rightarrow) \\ \delta(q_3, b) &= (q_4, b, \rightarrow) \\ \delta(q_4, b) &= (q_2, b, -)\end{aligned}$$

Si los símbolos del alfabeto de cinta b , a y b se codifican con 1, 11 y 111, respectivamente, la MT M se puede codificar con la siguiente secuencia binaria:

0101101110110100111011011101101001110111011101101001111010110101110

la cual se puede escribir también como

0101²01³01²01001³01²01³01²01001³01³01⁴01³01001⁴0101²0101³0.

Cambiando el orden de las cuatro transiciones de M obtendríamos en total $4! = 24$ codificaciones diferentes para M .

Es claro que no todas las secuencias binarias representan una MT: la codificación de una MT no puede comenzar con 1 ni pueden aparecer tres ceros consecutivos. Así, las secuencias 0101000110, 010001110 y 1011010110111010 no codifican ninguna MT. No es difícil concebir un algoritmo que determine si una secuencia binaria finita dada es o no una MT y que la decodifique en caso afirmativo. Una cadena binaria que codifique una MT se denomina un **código válido de una MT**. Si una cadena binaria no es un código válido de una MT, supondremos que codifica la MT con un solo estado y sin transiciones; tal MT no acepta ninguna cadena, es decir, acepta el lenguaje \emptyset . De esta forma, *todas las cadenas binarias representan máquinas de Turing*.

Por otro lado, las cadenas de ceros y unos se pueden ordenar lexicográficamente; el orden se establece por longitud y las cadenas de la misma longitud se ordenan ortográficamente de izquierda a derecha (considerando $0 < 1$). Este orden comienza así:

0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, 0001, 0010, ...

Tenemos entonces una enumeración de todas las cadenas binarias. Dado un alfabeto Σ , cada cadena binaria en la anterior lista representa, según la codificación definida arriba, una máquina de Turing que actúa sobre Σ . Podemos entonces hablar de la i -ésima máquina de Turing, la cual denominaremos por M_i . Nótese que en la enumeración M_1, M_2, M_3, \dots , cada MT aparece varias veces (porque al cambiar el orden de las transiciones se obtiene una codificación diferente). Las MT que aceptan el lenguaje \emptyset aparecen infinitas veces en la enumeración.

También será necesario codificar el conjunto de todas las cadenas sobre el alfabeto de entrada $\Sigma = \{s_2, \dots, s_m\}$, de tal manera que toda cadena tenga un código binario y toda secuencia binaria represente una cadena de Σ^* . Una vez codificados los símbolos de Σ , las cadenas de Σ^* se pueden codificar usando 0 como separador. Por ejemplo, la cadena $aab\bar{b}ab$ se codifica como

01101101110101101110

si los símbolos a y b hacen parte de Σ y se han codificado como 11 y 111. Nótese que en la codificación de una cadena $w \in \Sigma^*$ no aparecen nunca dos ceros consecutivos. En general, la codificación de una cadena $s_{i_1}s_{i_2}\cdots s_{i_k} \in \Sigma^*$ es

$$01^{i_1}01^{i_2}0\cdots 01^{i_k}0.$$

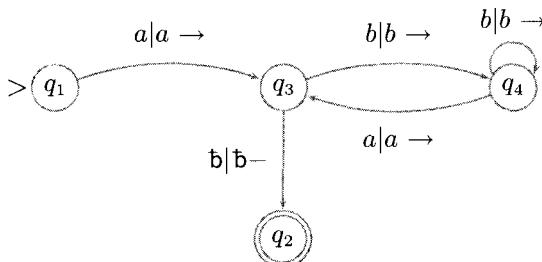
Las cadenas de Σ^* se pueden ordenar lexicográficamente estableciendo primero un orden arbitrario (pero fijo) para los símbolos de Σ , por ejemplo, $s_2 < s_3 < \cdots < s_m$. El orden lexicográfico de las cadenas de Σ^* induce un orden en el conjunto de las codificaciones; obtenemos así una enumeración w_1, w_2, w_3, \dots de los *códigos* de cadenas de Σ^* . Esto nos permite hablar de la i -ésima cadena de entrada, w_i .

Aunque tenemos dos enumeraciones diferentes, w_1, w_2, w_3, \dots (códigos de cadenas sobre Σ), y M_1, M_2, M_3, \dots (códigos de máquinas de Turing sobre Σ), no habrá peligro de confusión pues el contexto permitirá saber a cuál de las dos enumeraciones nos estamos refiriendo.

- » Es corriente identificar las cadenas de entrada w y las MT M con sus respectivas codificaciones binarias, y haremos eso en lo sucesivo.
- » El mecanismo de enumeración presentado en esta sección permite también concluir que el conjunto de todas las máquinas de Turing (sobre un alfabeto dado Σ) es infinito enumerable.

Ejercicios de la sección 7.1

- ① Sea M la MT definida por el siguiente diagrama de estados:



Determinar el lenguaje aceptado por M y codificar la máquina M siguiendo el esquema presentado en esta sección (codificar los símbolos del alfabeto de cinta b , a y b con 1, 11 y 111, respectivamente).

- ② Las siguientes secuencias binarias codifican MT que actúan sobre el alfabeto de entrada $\Sigma = \{a, b\}$, siguiendo el esquema de codificación presentado en esta sección. Decodificar las máquinas y determinar en cada caso el lenguaje aceptado (los símbolos b , a y b están codificados como 1, 11 y 111, respectivamente).

- (i) $0101101110110100111011101110100111101110 \xrightarrow{\text{sigue}} 111101110100111101101111101101001111010110101110$
- (ii) $0101^30101^30100101^201^301^20100101^201^501^20100 \xrightarrow{\text{sigue}} 1^301^201^401^201001^401^30101^301001^501^30101^30100 \xrightarrow{\text{sigue}} 1^50101^20101^30010101^20101^3$

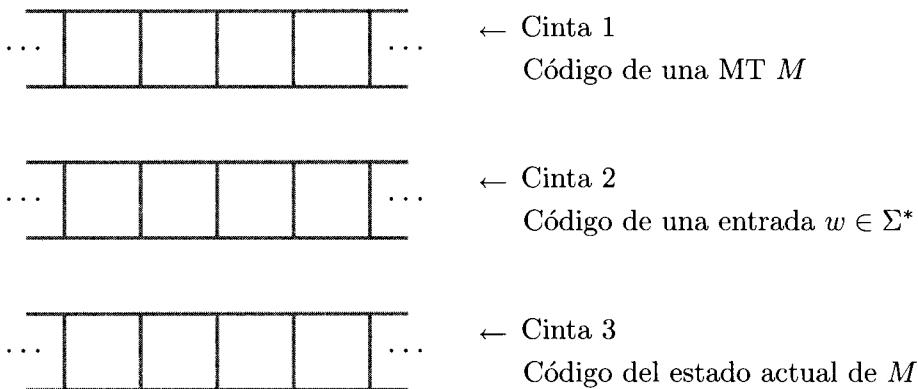
- ③ Supóngase que el alfabeto de entrada es $\Sigma = \{a, b, c, d\}$ y que los símbolos a , b , c y d se codifican como 11, 111, 1111 y 11111, respectivamente. Supóngase también que se establece el orden $a < b < c < d$ para los símbolos de Σ .

- (i) Codificar las cadenas $dacb$, $cddab$ y $badbcc$.
 - (ii) Escribir explícitamente las cadenas binarias w_6 , w_{15} , w_{22} y w_{25} .
 - (iii) ¿Qué puesto ocupa el código de la cadena cbd en el orden w_1 , w_2 , w_3 , ...? En otros términos, encontrar el índice i tal que w_i es el código de cbd .
- ④ Escribir las codificaciones de las siguientes MT: M_7 , M_{14} , M_{65} , M_{150} y M_{255} . ¿Cuál es el lenguaje aceptado por tales máquinas?

7.2. Máquina de Turing universal

La máquina de Turing universal M_u simula el comportamiento de todas las MT (sobre un alfabeto de entrada Σ dado). M_u procesa pares (M, w) , siendo M la codificación de una MT determinada y w la codificación de una cadena de entrada perteneciente a Σ^* (estas codificaciones se hacen en la forma indicada en la sección 7.1). La pareja (M, w) se puede presentar también como una cadena binaria, en la forma $M0w$. Es decir, los códigos de M y w se separan con un cero. Puesto que el código de M termina en 0 y el de w comienza con 0, en la cadena $M0w$ aparecen tres ceros consecutivos únicamente en el sitio que separa los códigos de M y w .

M_u es una MT con tres cintas cuyo alfabeto de cinta es $\{0, 1, \ddot{\text{b}}\}$ (el símbolo blanco $\ddot{\text{b}}$ de M_u difiere del símbolo blanco $\ddot{\text{b}}$ utilizado por las demás máquinas de Turing). La primera cinta contiene el código de una MT M determinada; la segunda cinta contiene inicialmente el código de una entrada w y se usa para simular el procesamiento que hace M de w . La tercera cinta se usa para almacenar el estado actual de M , también codificado: q_1 se codifica como 1, q_2 se codifica como 11, q_3 como 111, etc. En la siguiente gráfica se muestran las tres cintas de M_u :



La entrada $M0w$, que representa el par (M, w) , se escribe en la primera cinta; las otras dos cintas están inicialmente marcadas con el símbolo $\ddot{\text{b}}$ de M_u . La máquina M_u procede de la siguiente manera:

1. Deja el código de M en la primera cinta y pasa el código de w a la segunda cinta. Como se indicó arriba, para separar los códigos de M y w se busca el único sitio de la cadena que tiene tres ceros consecutivos.

2. La cadena 1, que representa el estado inicial q_1 , se coloca en la tercera cinta. La unidad de control pasa a escanear el primer símbolo de cada cadena binaria, en cada una de las tres cintas.
3. Examina el código de M para determinar si representa una MT. En caso negativo, M_u se detiene sin aceptar (recuérdese que los códigos no válidos representan una MT que no acepta ninguna cadena).
4. M_u utiliza la información de las cintas 2 y 3 para buscar en la cinta 1 la transición que sea aplicable. Si encuentra una transición aplicable, M_u simula en la cinta 2 lo que haría M y cambia el estado señalado en la cinta 3. Esto requiere re-escribir la cadena de la cinta 2 desplazando adecuadamente los símbolos a izquierda o a derecha; para lo cual se utilizan las subrutinas mencionadas en la sección 6.2. La simulación continúa de esta forma, si hay transiciones aplicables. Después de realizar una transición, la unidad de control regresa, en la primera y tercera cintas, al primer símbolo de la cadena.

Si al procesar una entrada w , M_u se detiene en el único estado de aceptación de M (que es q_1 y está codificado como 1), entonces la cadena w será aceptada. Por consiguiente, M_u tiene también un único estado de aceptación, q_1 , que es el mismo estado de aceptación de cualquier otra MT.

5. Si M_u no encuentra una transición aplicable o se detiene en un estado que no es de aceptación, M_u se detiene sin aceptar, como lo haría M .

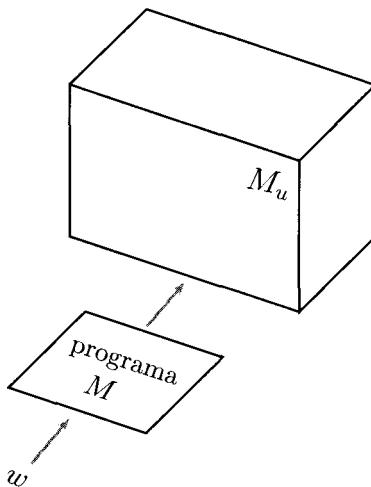
Se tiene entonces que M_u acepta la entrada $M0w$ si y solamente si M acepta a w . De modo que el lenguaje aceptado por la máquina de Turing universal M_u se puede describir explícitamente; este lenguaje se denomina corrientemente el **lenguaje universal** y se denota con L_u :

$$L_u = \{M0w : \text{la MT } M \text{ acepta la cadena } w \in \Sigma^*\}.$$

El lenguaje universal L_u es, por consiguiente, un lenguaje RE.

Utilizando codificaciones binarias, en lugar de unitarias, es posible construir una máquina de Turing universal más eficiente (véanse los detalles en el recuadro gris al final de esta sección).

Para comprender el significado del concepto de máquina de Turing universal podemos recurrir a una útil analogía, ilustrada en la siguiente figura:



Como la función de transición de una MT no es más que una lista de instrucciones, podemos concebir una MT como un *programa computacional* y la máquina de Turing universal resulta ser un *mecanismo en el que podemos ejecutar todos los programas*. En otros términos, la máquina de Turing universal es *una máquina de Turing programable*.



Como hemos utilizado una codificación unitaria (cadenas de unos) de longitud variable para los símbolos del alfabeto de cinta y para los estados, el funcionamiento de la máquina de Turing universal es muy ineficiente: las acciones descritas en el numeral 4 exigen continuos desplazamientos de las cadenas previamente escritas en la cinta 2. Se podría utilizar una codificación binaria para asegurar que cada símbolo codificado ocupe el mismo número de casillas. Concretamente, si una MT M utiliza el lenguaje de cinta $\Gamma = \{s_1, s_2, \dots, s_m, \dots, s_p\}$, donde $s_1 = \text{b}$, un símbolo de Γ se puede codificar por medio de una codificación binaria de exactamente k bits (codificación de longitud k), siendo k el menor entero tal que $p \leq 2^k$. Las cadenas de Γ^* se pueden codificar luego usando un símbolo adicional, por ejemplo $\#$, como separador.

La codificación binaria también se puede utilizar para los estados y para las transiciones de una MT cualquiera. De esta forma, se puede construir una máquina de Turing universal más eficiente, que funcione como se describió en los numerales 1 a 5, pero que utilice codificaciones binarias de longitud fija para re-escribir las transiciones de cada MT M y para simularla.

7.3. Algoritmos de aceptación para lenguajes RE

Recurriendo a la tesis de Church-Turing, se puede concluir que un lenguaje L es RE si se exhibe un **algoritmo de aceptación**¹ para L . Para una entrada u , el algoritmo debe finalizar con aceptación si y sólo si $u \in L$. Si $u \notin L$, el algoritmo puede detenerse, sin aceptar, o puede no detenerse nunca.

Ejemplo El argumento que se usó en la sección anterior para construir la máquina de Turing universal M_u y concluir que L_u es RE se puede presentar como un algoritmo de aceptación:

1. Entrada: $M0w$.
2. Ejecutar la MT M con la entrada w .
3. Aceptar si M se detiene en un estado de aceptación.

Este algoritmo finaliza con aceptación si M acepta a w . Si M no acepta a w , el algoritmo puede terminar cuando M se detenga en un estado de rechazo, o puede no terminar nunca. En conclusión: el algoritmo finaliza con aceptación de la entrada $M0w$ si y sólo si M acepta a w .

Ejemplo Demostrar que el lenguaje

$$L_a = \{M : L(M) \neq \emptyset\} = \{M : M \text{ acepta alguna cadena}\}.$$

es RE.

Solución. Usando el Teorema 6.8.3 se puede presentar el siguiente algoritmo para aceptar a L_a :

1. Entrada: una MT M arbitraria.
2. Construir la MT M' del Teorema 6.8.3; M' enumera secuencialmente las cadenas de $L(M)$.
3. Detenerse y aceptar cuando M' genere la primera cadena.

¹Aquí estamos usando la noción de *algoritmo* en una acepción muy amplia. Por lo general, se exige que un algoritmo siempre debe finalizar con una salida, es decir, no debe tener bucles infinitos.

Si M acepta alguna cadena, ésta será generada eventualmente por M' . Si M no acepta ninguna cadena, el anterior algoritmo nunca termina. En conclusión: el algoritmo finaliza con aceptación de la entrada M si y sólo si M acepta alguna cadena.

También podemos presentar un algoritmo no-determinista para aceptar a L_a . En un algoritmo no-determinista hay una etapa de “conjetura” que corresponde a una búsqueda exhaustiva en un algoritmo determinista. He aquí un algoritmo no-determinista para aceptar a L_a :

1. Entrada: una MT M arbitraria.
2. Escoger de manera no-determinista una cadena w sobre el alfabeto de cinta. Esto se hace con una subrutina generadora como la del ejemplo de la sección 6.5.4. Una escogencia no-determinista generalmente se llama una *conjetura*.
3. Ejecutar la MT M con la entrada w .
4. Aceptar si M se detiene en un estado de aceptación.

Nótese que si M acepta aunque sea una cadena, ésta será encontrada, eventualmente, en el paso 2. Si M no acepta ninguna cadena, el anterior algoritmo nunca termina. En conclusión: el algoritmo finaliza con aceptación de la entrada M si y solo si M acepta alguna cadena.

Los dos algoritmos de este ejemplo son esencialmente el mismo (lo cual corresponde al hecho de que las MT deterministas y las no-deterministas tienen el mismo poder computacional): en el primero se hace una búsqueda exhaustiva de una cadena aceptada por M ; en el segundo se hace una conjectura de una cadena aceptada. Ambos algoritmos finalizan únicamente si la MT M acepta alguna cadena.

Ejercicios de la sección 7.3

Presentar algoritmos de aceptación, similares a los de los ejemplos de esta sección, para concluir que los siguientes lenguajes son RE:

- ① $L_p = \{M0w : M \text{ se detiene con entrada } w\}$.
- ② $L_b = \{M : M \text{ se detiene al operar con la cinta en blanco}\}$.
- ③ $L = \{M : M \text{ se detiene con al menos una entrada}\}$.
- ④ $L = \{M : M \text{ acepta por lo menos dos cadenas}\}$.
- ⑤ $L = \{(M_1, M_2) : L(M_1) \cap L(M_2) \neq \emptyset\}$. Nota: Las parejas de MT (M_1, M_2) se pueden codificar en la forma M_10M_2 .

7.4. Lenguajes que no son RE

En esta sección exhibiremos un lenguaje que no es RE, o sea, un lenguaje que no puede ser reconocido por ninguna MT. Recordemos que para concluir que el lenguaje $L = \{a^i b^i : i \geq 0\}$ no es regular y que el lenguaje $L = \{a^i b^i c^i : i \geq 0\}$ no es LIC utilizamos razonamientos por contradicción (basados en lemas de bombeo). Aquí también razonaremos por contradicción, aunque el lenguaje definido es completamente artificial y la contradicción se obtiene por medio de un “argumento diagonal” de interacción entre las enumeraciones w_1, w_2, w_3, \dots (entradas) y M_1, M_2, M_3, \dots (máquinas de Turing).

7.4.1 Teorema. *El lenguaje*

$$L = \{w_i : w_i \text{ no es aceptada por } M_i\}$$

no es RE, es decir, no es aceptado por ninguna MT.

Demostración. Razonamos suponiendo que L sí es RE para llegar a una contradicción. Si L fuera RE sería aceptado por una MT M_k , con respecto a la enumeración de máquinas de Turing ya descrita. Es decir, $L = L(M_k)$ para algún k . Se tendría entonces

$$w_k \in L \implies w_k \text{ no es aceptada por } M_k \implies w_k \notin L(M_k) = L.$$

$$w_k \notin L \implies w_k \notin L(M_k) \implies w_k \text{ es aceptada por } M_k \implies w_k \in L.$$

Por lo tanto, $w_k \in L \iff w_k \notin L$, lo cual es una contradicción. \square

El lenguaje L del Teorema 7.4.1 se denomina **lenguaje de diagonalización** y se denota con L_d :

$$L_d = \{w_i : w_i \text{ no es aceptada por } M_i\}.$$

A lo largo de este capítulo encontraremos otros lenguajes que no son RE.



El “argumento diagonal” del Teorema 7.4.1 recuerda el argumento utilizado por Cantor para demostrar que el conjunto de los números reales no es enumerable. Tal argumento consiste en suponer, razonando por contradicción, que el conjunto de los números reales entre 0 y 1 es enumerable: r_1, r_2, r_3, \dots . Si se escriben las expansiones decimales de los números (evitando las secuencias de nueves finales, para eliminar representaciones múltiples), se obtendría una “matriz infinita” de la forma:

$$r_1 = 0.a_{11}a_{12}a_{13} \dots$$

$$r_2 = 0.a_{21}a_{22}a_{23} \dots$$

$$r_3 = 0.a_{31}a_{32}a_{33} \dots$$

$$\vdots \quad \vdots \quad \vdots$$

Un número real $r = 0.b_1b_2b_3 \dots b_k \dots$, en el que $b_i \neq a_{ii}$ y $b_i \neq 9$ para todo i , sería diferente de todos y cada uno de los r_k . Es decir, dada cualquier enumeración de los reales del intervalo $[0, 1]$, se puede siempre construir un real que no esté en la lista, y esto se puede lograr modificando los dígitos de la diagonal principal.

Este argumento es completamente similar al de la demostración del Teorema 7.4.1. En la siguiente matriz infinita hay un 1 en la posición (i, j) si M_i acepta a w_j , y un 0 si M_i no acepta a w_j :

	w_1	w_2	w_3	w_4	\dots
M_1	0	0	1	0	\dots
M_2	1	0	0	0	\dots
M_3	0	1	1	0	\dots
\vdots	\vdots	\vdots	\vdots	\vdots	

Las filas representan *todos* los lenguajes RE. Por *diagonalización* construimos un lenguaje que no puede estar en esa lista y, por lo tanto, no puede ser RE. El lenguaje L_d se construye “complementando” la diagonal (cambiando ceros por unos y unos por ceros).

7.5. Lenguajes RE no recursivos

A continuación mostraremos que existen lenguajes RE que no son recursivos, lo cual quiere decir que la contenencia

$$\text{Lenguajes recursivos} \subsetneq \text{Lenguajes RE}$$

es estricta o propia (no hay igualdad). Esto implica que existen lenguajes que pueden ser aceptados por MT específicas pero en cualquier MT que los acepte habrá cómputos que nunca terminan (obviamente, los cómputos de las cadenas aceptadas siempre terminan). De este hecho extraemos la siguiente importante conclusión: *los cómputos interminables, o bucles infinitos, no se pueden eliminar de la teoría de la computación*.

El primer ejemplo de un lenguaje RE no-recursivo es el lenguaje universal L_u presentado en la sección 7.2.

7.5.1 Teorema. *El lenguaje universal,*

$$L_u = \{M0w : M \text{ acepta a } w\},$$

es RE pero no es recursivo.

Demostración. En las secciones 7.2 y 7.3 se vió que L_u es RE. Para mostrar que L_u no es recursivo razonamos por contradicción: suponemos que existe una MT \mathcal{M} que procesa todas las entradas $M0w$ y se detiene siempre en un estado de ‘aceptación’ (si M acepta a w) o en uno de ‘rechazo’ (si M acepta a w). Esta suposición permitirá construir una MT \mathcal{M}' que acepte el lenguaje L_d del Teorema 7.4.1, de lo cual se deduciría que L_d es RE, contradiciendo así la conclusión de dicho teorema.

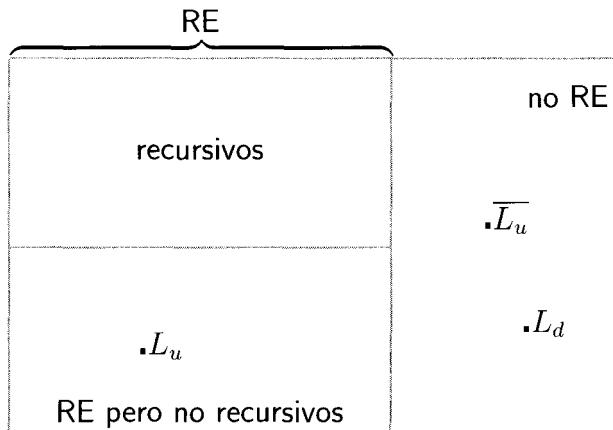
Con una entrada $w \in \Sigma^*$, la máquina \mathcal{M}' procede así: enumera sistemáticamente las cadenas w_1, w_2, w_3, \dots hasta que encuentra un k tal que $w = w_k$. Luego simula (o invoca) a \mathcal{M} con entrada M_k0w_k , decidiendo si M_k acepta o no a w_k . Por lo tanto, \mathcal{M}' acepta el lenguaje L_d , o sea, $L(\mathcal{M}') = L_d$. Esto significa, en particular, que L_d es RE lo cual contradice el Teorema 7.4.1. \square

El siguiente teorema sirve como fuente de ejemplos de lenguajes no RE.

7.5.2 Teorema. *Si L es un lenguaje RE no recursivo, \overline{L} no es RE.*

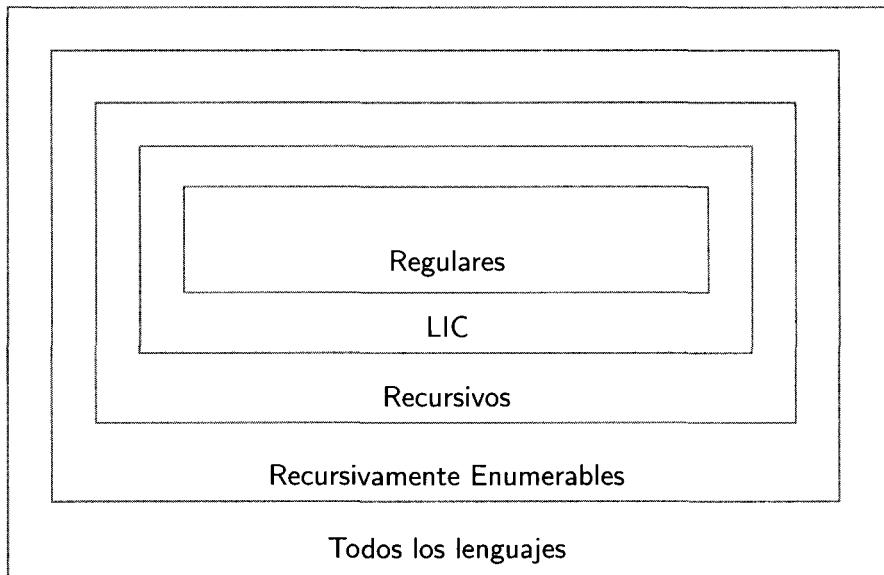
Demostración. Razonamiento por contradicción: por el Teorema 6.8.2, si \overline{L} fuera RE, L sería recursivo. \square

Como aplicación de este resultado podemos concluir que el complemento del lenguaje universal, $\overline{L_u}$, no es RE. La siguiente gráfica muestra la relación entre los lenguajes recursivos, los lenguajes RE y los no RE:



Los resultados de las dos últimas secciones permiten establecer las relaciones de contenencia entre las colecciones de lenguajes estudiadas en este curso (sobre un alfabeto Σ dado):

$$\text{Regulares} \subsetneq \text{LIC} \subsetneq \text{Recursivos} \subsetneq \text{RE} \subsetneq \wp(\Sigma^*) .$$



En capítulos anteriores mostramos que hay lenguajes recursivos que no son LIC y lenguajes LIC que no son regulares. En el presente capítulo hemos demostrado la existencia de lenguajes RE que no son recursivos y de lenguajes que no son RE. Por tal razón, todas las contenencias anteriores son estrictas.

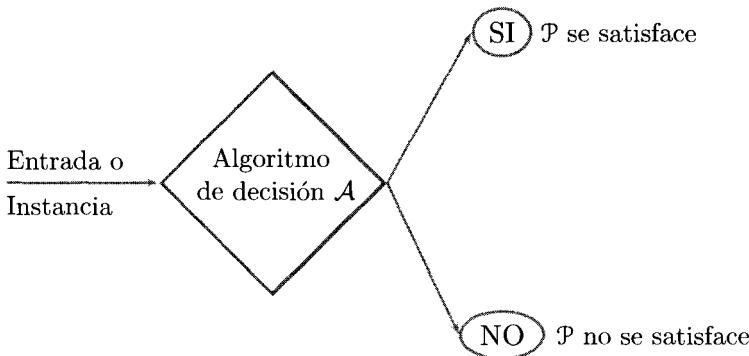
En la siguiente tabla se muestra, a manera de resumen, la relación entre los lenguajes estudiados y las máquinas que los aceptan.

Lenguajes	Máquinas aceptadoras
Regulares	Autómatas finitos ($AFD \equiv AFN \equiv AFN-\lambda$)
LIC	Autómatas con pila no-deterministas (AFPN)
RE	Máquinas de Turing (MT)
Recursivos	Máquinas de Turing que se detienen con toda entrada

7.6. Problemas indecidibles o irresolubles

Dada una propiedad \mathcal{P} referente a máquinas de Turing, un problema de decisión para \mathcal{P} consiste en buscar un algoritmo \mathcal{A} , aplicable a toda MT M (es decir, a toda codificación binaria), que responda SI o NO a la pregunta: ¿satisface M la propiedad \mathcal{P} ? Si existe un algoritmo de decisión, se dice que el problema \mathcal{P} es **decidable** o **resoluble**; en caso contrario, el problema \mathcal{P} es **indecidible** o **irresoluble**.

Un algoritmo de decisión debe ser aplicable uniformemente a todas las entradas (¡hay infinitas entradas!) y terminar con una de las conclusiones ‘SI’ o ‘NO’:



Según la tesis de Church-Turing (sección 6.9), identificamos algoritmos con máquinas de Turing y, por lo tanto, decir que el problema \mathcal{P} es **indecidible** equivale a afirmar que el lenguaje

$$L = \{M : M \text{ es el código de una MT que satisface } \mathcal{P}\}$$

no es recursivo.

Ejemplo El hecho de que el lenguaje universal L_u no es recursivo, establecido en el Teorema 7.5.1, equivale a afirmar que el siguiente problema de decisión (el “problema universal”) es indecidible:

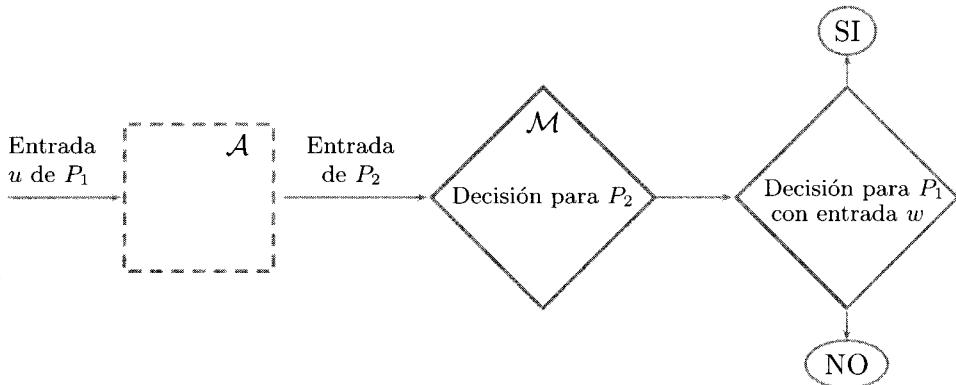
Dada una MT M cualquiera, sobre un alfabeto de cinta Σ pre-determinado, y una cadena $w \in \Sigma^*$, ¿acepta M a w ?

Nótese que las entradas o instancias para este problema son de la forma $M0w$ donde M es el código de una MT y w es el código de una entrada (sobre el alfabeto Σ).

Técnica de reducción de problemas

Conociendo que ciertos problemas son indecidibles, se puede concluir que otros problemas de decisión también lo son si se razona por contradicción. Para ser más precisos, supóngase que ya se sabe que un cierto problema P_1 es indecidible (como el problema universal, por ejemplo). Podríamos concluir que un problema dado P_2 es indecidible razonando así: si P_2 fuera decidible también lo sería P_1 . Esta contradicción mostrará que el problema P_2 no puede ser decidible. Cuando se emplea este razonamiento, se dice que *el problema P_1 se reduce al problema P_2* .

Para utilizar esta técnica de reducción, es necesario diseñar un algoritmo \mathcal{A} (o una máquina de Turing) que sea capaz de convertir una entrada cualquiera u del problema P_1 en entradas para el problema P_2 de tal manera que, al aplicar la supuesta MT \mathcal{M} que resuelve el problema P_2 , se llegue a una decisión, SI o NO, del problema P_1 para la entrada u . La siguiente gráfica ilustra este procedimiento; el algoritmo \mathcal{A} , que aparece representado por el rectángulo a trozos, es la parte esencial del procedimiento de reducción.

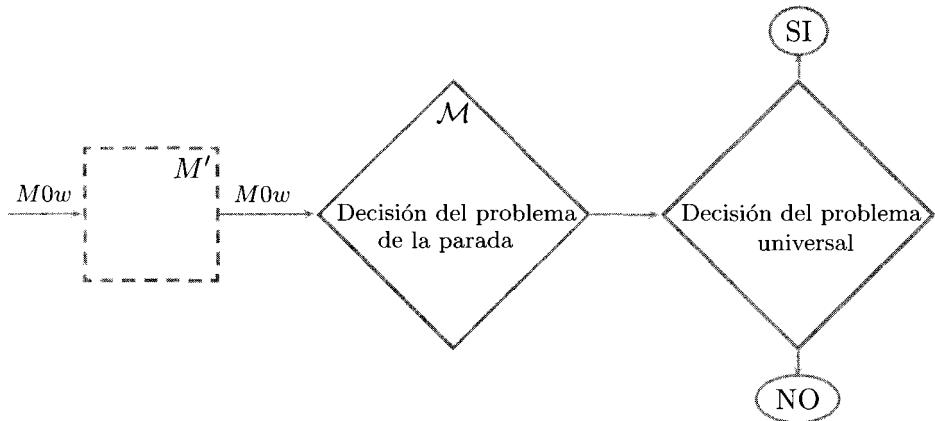


Ejemplo Problema de la parada o problema de la detención. Este famoso problema (*halting problem*, en inglés), estudiado por el propio Turing, consiste en preguntar si existe un algoritmo para el siguiente problema de decisión:

Dada una MT M cualquiera, sobre el alfabeto de cinta Σ , y una cadena $w \in \Sigma^*$, ¿se detiene M al procesar la entrada w ?

El problema universal se puede reducir al problema de la parada. En otros términos, asumiendo la existencia de una MT \mathcal{M} que resuelva el problema

de la parada se puede resolver el problema universal. La gráfica siguiente esboza el razonamiento.



Sea $M0w$ una entrada arbitraria (M y w codifican MT y cadenas de Σ^* , respectivamente). La máquina M' solamente devuelve la entrada $M0w$, ya que las entradas para el problema universal y para el problema de la parada coinciden. Puesto que \mathcal{M} es capaz de decidir si M se detiene o no con entrada w , tendríamos un algoritmo para resolver el problema universal:

1. Si M no se detiene con entrada w , entonces M no acepta a w .
2. Si M se detiene con entrada w , procesar w con M y determinar si es aceptada o no.

Conclusión: si el problema de la parada fuera decidable, también lo sería el problema universal.

El anterior argumento también permite concluir que el lenguaje

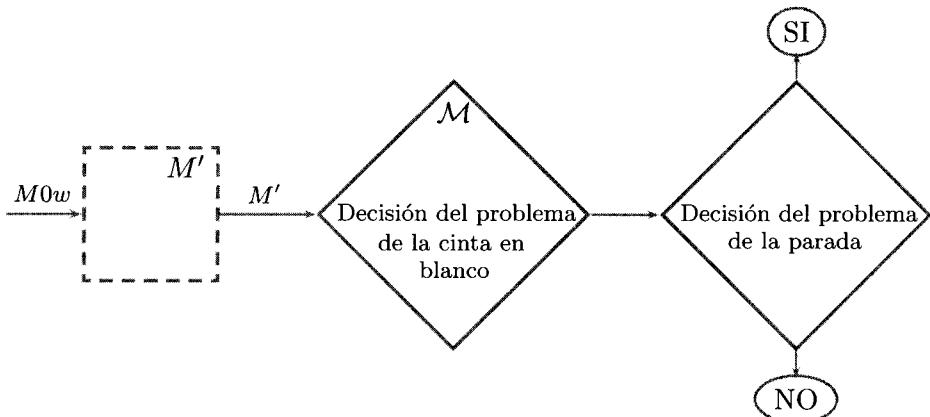
$$L_p = \{M0w : M \text{ se detiene con entrada } w\}$$

no es recursivo, aunque, según el ejercicio ① de la sección 7.3, L_p es RE.

Ejemplo Problema de la cinta en blanco.

Dada una MT M cualquiera, sobre el alfabeto de cinta Σ , ¿se detiene M al iniciar su funcionamiento con la cinta en blanco (todas las celdas marcadas con b)?

El problema de la parada se puede reducir al problema de la cinta en blanco, es decir, asumiendo la existencia de una MT \mathcal{M} que resuelva el problema de la cinta en blanco se puede resolver el problema de la parada. La siguiente gráfica esboza el razonamiento:



Sea $M0w$ una entrada arbitraria. Construimos una MT M' que empiece a operar con la cinta en blanco y, como primera instrucción, escriba la cadena $M0w$ en una porción reservada (finita) de la cinta. M' simula luego el procesamiento que hace M con entrada w . Como M' inicia su procesamiento con la cinta en blanco, podemos ejecutar la máquina (algoritmo) \mathcal{M} , con entrada M' (codificada). \mathcal{M} decide si M' se detiene o no y, por lo tanto, se obtiene una decisión sobre si M se detiene o no con entrada w . Conclusión: si el problema de la cinta en blanco fuera decidable, también lo sería el problema de la parada.

Lo anterior también permite concluir que el lenguaje

$$L_b = \{M : M \text{ se detiene al operar con la cinta en blanco}\}$$

no es recursivo.

Ejercicios de la sección 7.6

- ① Mediante la técnica de reducción de problemas mostrar que los siguientes problemas de decisión son indecidibles:
 - (i) Problema de la impresión: dada $M = (Q, q_0, F, \Sigma, \Gamma, \mathbf{b}, \delta)$ una MT cualquiera y un símbolo $s \in \Gamma$, ¿imprimirá M alguna vez el símbolo s sobre la cinta si M inicia su funcionamiento con

la cinta en blanco? Ayuda: reducir el problema de la cinta en blanco a este problema.

- (ii) Problema del ingreso a un estado: dada $M = (Q, q_0, F, \Sigma, \Gamma, \mathbf{b}, \delta)$ una MT cualquiera, una cadena de entrada $w \in \Sigma^*$ y estado $q \in Q$, ¿ingresará M al estado q al procesar la cadena w ? Ayuda: reducir el problema universal a este problema.
 - !(iii) Dada una MT M , $\mathcal{L}(M) = \Sigma^*$? Ayuda: el problema de la cinta en blanco se puede reducir a este problema. Para ello, diseñar un algoritmo que genere el código de una MT M' de tal manera que se cumpla: “ M' acepta cualquier cadena si y sólo si M se detiene con la cinta en blanco”.
 - !(iv) Dada una MT M , $\mathcal{L}(M) \neq \emptyset$? Ayuda: utilizar una idea similar a la del problema (iii).
 - (v) Dadas dos MT M_1 y M_2 cualesquiera, $\mathcal{L}(M_1) = \mathcal{L}(M_2)$? Ayuda: si este problema fuera decidible, también lo sería el problema (iii), tomando como M_2 una MT adecuada.
 - (vi) Dadas dos MT M_1 y M_2 cualesquiera, $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$? Ayuda: si este problema fuera decidible, también lo sería el problema (iii), tomando como M_1 una MT adecuada.
 - (vii) Dadas dos MT M_1 y M_2 cualesquiera, $\mathcal{L}(M_1) \cap \mathcal{L}(M_2) \neq \emptyset$? Ayuda: si este problema fuera decidible, también lo sería el problema (iv), tomando como M_2 una MT adecuada.
- ② Demostrar que si el problema de la parada fuera resoluble, todo lenguaje RE sería recursivo.
- ③ Demostrar que los lenguajes $\overline{\mathcal{L}_a}$, $\overline{\mathcal{L}_b}$ y $\overline{\mathcal{L}_p}$ no son RE. Explícitamente, estos lenguajes están definidos como
- $$\overline{\mathcal{L}_a} = \{M : \mathcal{L}(M) = \emptyset\} = \{M : M \text{ no acepta ninguna cadena}\},$$
- $$\overline{\mathcal{L}_b} = \{M : M \text{ no se detiene al operar con la cinta en blanco}\},$$
- $$\overline{\mathcal{L}_p} = \{M0w : M \text{ no se detiene con entrada } w\}.$$
- Ayuda: usar el Teorema 7.5.2.
- ④ Demostrar que el lenguaje $\{(M_1, M_2) : \mathcal{L}(M_1) \cap \mathcal{L}(M_2) = \emptyset\}$ no es RE.

Bibliografía

- [ASU] A. AHO, R. SETHI & J. ULLMAN, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.
- [B] F. BECKMAN, *Mathematical Foundations of Programming*, Addison-Wesley, 1980.
- [C] N. CUTLAND, *Computability*, Cambridge University Press, 1980.
- [GPR] P. GARCÍA, T. PÉREZ, J. RUIZ, et al., *Teoría de Autómatas y Lenguajes Formales*, Alfaomega, 2001.
- [HU1] J. HOPCROFT & J. ULLMAN, *Formal Languages and their Relation to Automata*, Addison-Wesley, 1969.
- [HU2] J. HOPCROFT & J. ULLMAN, *Introduction to Automata Theory, Language and Computation*, Addison-Wesley, 1979.
- [HMU] J. HOPCROFT , R. MOTWANI & J. ULLMAN, *Introducción a la Teoría de Autómatas, Lenguajes y Computación*, Segunda Edición, Pearson-Educación, 2002.
- [K] D. KELLEY, *Teoría de Autómatas y Lenguajes Formales*, Prentice Hall, 1995.
- [LP] H. LEWIS & C. PAPADIMITRIOU, *Elements of the Theory of Computation*, Prentice Hall, 1981.
- [R] G. RÉVÉSZ, *Introduction to Formal Language Theory*, Dover Publications, 1991.
- [Sa] J. SAVAGE, *Models of Computation*, Addison-Wesley, 2000.

- [Su] T. SUDKAMP, *Languages and Machines*, Addison-Wesley, 1988.
- [SW] R. SOMMERHALDER & S. VAN WESTRHENEN, *The Theory of Computability*, Addison-Wesley, 1988.

La colección Notas de Clase de la Facultad de Ciencias, es un espacio abierto a los profesores para publicar su experiencia docente, recopilada en el tiempo a través de notas escritas. Se espera que las interrelaciones autor-colegas y profesores-estudiantes sean la mejor manera para que estas notas se depuren, a fin de que en un futuro mediato adquieran carátula de texto.

**Comité Editorial
Facultad de Ciencias**