

Notas del Curso:
CCO-500 LENGUAJES FORMALES Y AUTÓMATAS
Facultad de Ciencias de la Computación
Benemérita Universidad Autónoma de Puebla

M.I.A. José Juan Palacios Pérez
jpalacio@cs.buap.mx
<http://www.cs.buap.mx/~jpalacio>

Primavera 2002

Estas notas de curso tienen el propósito de servir como referencia básica y material de trabajo. Fundamentalmente se sigue la estructura propuesta en [HU79] enriquecido con [Koz96].

La metodología que se sigue en la exposición de los temas es la siguiente:

1. Antes de presentar formalmente algún concepto nuevo, primero motivaremos su importancia o utilidad mediante un ejemplo (intuitivo) práctico;
2. a continuación, presentaremos su definición rigurosa y propiedades
3. finalmente, una gama de ejercicios deberán realizarse para comprender mejor el(los) conceptos presentados.

Éste es un curso formativo fundamental para el profesionista serio de la computación. En éste curso se sientan las bases (y herramientas formales) que se utilizan en otros cursos; por ejemplo, de las bases matemáticas se discute ampliamente el **principio de inducción matemática**, tal principio tiene como resultado pragmático la definición de procedimientos efectivos recursivos y las propiedades de los mismos.

Se espera que el estudiante posea conocimientos básicos de matemáticas discretas (teoría básica de conjuntos, relaciones y funciones) así como familiaridad de al menos un semestre en la utilización de un lenguaje de programación imperativo (como C o Pascal) y evidentemente familiaridad en la utilización del *shell* de un sistema operativo (Un*x ó DOS).

La meta fundamental de éste curso es **entender los fundamentos de las ciencias de la computación**. Para ello, nos preguntaremos (desde el principio del curso hasta el final):

1. ¿Qué es un *Modelo de Cómputo*? ¿Existe una jerarquía o clasificación de tales modelos?
2. ¿Que significa que una *función* sea *computable*?
3. ¿Existen funciones que no son computables?
4. ¿Cómo depende el poder computacional de las construcciones de programación?

En la búsqueda de respuestas, encontraremos conceptos fundamentales y penetrantes como: *estado*, *transición*, *no determinismo*, *reducibilidad*, *indecidibilidad*, etc. Algunos de los logros más importantes en la teoría de la computación ha sido cristalizar tales conceptos[Koz96].

A lo largo de los años se han propuesto diversos modelos de cómputo, i.e, modelos **abstractos** que permite capturar algún aspecto fundamental de la idea intuitiva de que *algo* pueda ser llevado a cabo (realizado) mecánicamente. Independientemente y al paralelo del desarrollo de tales modelos, el lingüista Noam Chomsky se propuso formalizar la noción de *gramática y lenguaje*¹. Tal esfuerzo tuvo como resultado la definición de la *Jerarquía de Chomsky*, la cual describe la correspondencia entre los modelos de cómputo *clásicos* en orden de poder creciente y la clase de *lenguajes* que tales modelos aceptan:

Modelo de Cómputo Abstracto	Gramáticas y Lenguajes
(<i>memoria finita</i>) Autómata Finito y Expresiones Regulares	Lineal por la Derecha (<i>right-linear</i>)
(<i>memoria finita + pila</i>) Autómata de Pila (<i>pushdown</i>)	Libres de Contexto
(<i>sin restricciones</i>)	Sin restricciones

Aún cuando los modelos mecánicos y las gramáticas parecen ser bastante diferentes a nivel superficial, el proceso básico de *analizar (parsing)* un enunciado en un lenguaje para afirmar si está o no correcto tiene una semejanza muy fuerte con la noción de procedimiento efectivo (algoritmo).

Existe una cuarta clase de lenguajes denominada *sensible al contexto* la cual se encuentra situada entre la tercera y segunda de la tabla, y corresponde a cierta clase de modelos mecánicos conocidos como *Autómatas Linealmente Acotados (linear bounded automata)*.

Debemos citar los modelos de cómputo descritos en la última parte de la jerarquía, ya que un resultado importante es que todos ellos son **equivalentes** en el sentido que permiten capturar la misma clase de funciones computables:

¹Intuitivamente, una gramática es un conjunto de reglas que permiten establecer *cómo* construir expresiones correctas de un lenguaje, siendo un lenguaje fundamentalmente un conjunto de palabras o expresiones tomadas de cierto conjunto finito de símbolos (alfabeto) de acuerdo a las reglas de la gramática.

- Máquinas de Turing (Alan Turing, 1936).
- λ -*calculus* (Alonzo Church, 1933).
- Sistemas de Post (Emil Post, 1943).
- Funciones μ -recursivas (Kurt Gödel, Jacques Herbrand 1936).
- Lógica Combinatoria (Moses Schönfinkel, Haskell B. Curry 1924).

No es coincidencia que una jerarquía naturalmente definida en un campo (modelos mecánicos de cómputo) coincida con una jerarquía definida naturalmente en un campo completamente distinto (lenguajes y gramáticas). Los modelos mecánicos de cómputo fueron identificados en la misma manera que se formulan las teorías en física ó en alguna otra disciplina científica: al estudiar fenómenos de la naturaleza se pueden identificar *patrones recurrentes* que ocurren en varias formas (incluso en la música y arte visual). Aunque las formas podrían ser substancialmente diferentes a nivel superficial tienen parecido significativo entre sí (en términos de propiedades) que permiten sugerir la presencia latente de ciertos principios comunes subyacentes en ambos. Cuando ésto ocurre, resulta significativo intentar, en principio, la construcción de un **modelo general** (abstracto) que capture tales principios subyacentes de la manera *más sencilla posible*, sin preocuparse en los detalles prescindibles de cada manifestación particular del fenómeno. Justamente, éste es el proceso de **abstracción**; el cual, como tal, forma parte de la esencia del progreso científico dado que permite enfocar la atención en los principios importantes los cuales se encuentran ocultos por detalles irrelevantes.

Tal vez el ejemplo más sorprendente de fenómeno que discutiremos en el curso es justamente el concepto de *computabilidad efectiva*. La búsqueda para ofrecer una noción de computabilidad efectiva comenzó alrededor del comienzo del siglo 20 con el desarrollo de la escuela de matemática *formalista*, encabezada por el filósofo Bertrand Russell y el matemático David Hilbert. Ellos deseaban reducir toda la matemática al proceso de la manipulación formal de símbolos. Efectivamente, la manipulación formal de símbolos es una forma de cómputo, aún cuando no existían computadoras disponibles para tales investigadores. Sin embargo, sí se tenía consciencia de las nociones de *algoritmo* y *cómputo*: los matemáticos, filósofos y lógicos reconocen un método constructivo en cuanto lo ven. Existieron varios intentos para atrapar la noción general de computabilidad efectiva, teniendo como resultado varias propuestas con sus propias particularidades; todas ellas, sin embargo, pudieron *simularse* entre sí, de tal suerte que todas resultan ser *computacionalmente equivalentes*. En otras palabras, la manipulación simbólica formal tiene muchos rostros (sistemas deductivos en lógica, sistemas de reescritura, funciones recursivas) pero todas ellas esencialmente describen el mismo fenómeno.

El esfuerzo de Hilbert y Russell fué despedazado por el *Teorema de Incompletitud*, formulado por Kurt Gödel. Básicamente, su resultado establece que no importa que tan poderoso sea un sistema deductivo para describir las propiedades de la teoría de números naturales, siempre será posible construir enunciados simples que sean verdaderos pero indemostrables en ése sistema deductivo. Éste resultado se considera como uno de los mayores logros intelectuales, aún cuando tiene profundas raíces en la *paradoja* de Russell. El Teorema de Incompletitud de Gödel se trata esencialmente de un enunciado acerca de la computabilidad y estaremos en posición de ofrecer una discusión introductoria al final del curso.

Las notas se encuentran disponibles en la URL:

<http://www.cs.buap.mx/~jpalacio>

Comentarios, sugerencias, etc. son bienvenidos!

Se agradece la colaboración para la captura de los primeros tres capítulos a los estudiantes Lourdes Tecuapetla Quécholt, Jeanine Flores Flores y Daniel Ruiz Quintana durante el curso de primavera.

Los errores que queden son totalmente mi responsabilidad. Última revisión: Verano, 2002.

A la dulce memoria de
ESTELA ANDRADE ESCOBAR
1973 - 2002
brillará por siempre tu luz en mi corazón.

Contenido

Preliminares Matemáticos	1
1.1 Problemas de Decisión vs. Funciones	1
1.1.1 Notación de Conjuntos	1
1.1.2 Conjuntos Infinitos	2
1.1.3 Relaciones	3
1.2 Cadenas y Operaciones sobre cadenas	4
1.2.1 Operaciones sobre cadenas	5
1.2.2 Operaciones sobre conjuntos	5
1.3 Grafos	6
1.3.1 Grafos Dirigidos	6
1.3.2 Árboles	6
1.4 Principio de Inducción Matemática	7
 Autómatas Finitos y Conjuntos Regulares	 13
Autómatas Finitos Determinísticos	13
2.2 Composición de AFD's y algunas Propiedades de Cerradura	18
Autómatas Finitos No Determinísticos	21
3.2 Equivalencia entre AFD y AFND	23
3.3 AFND con movimientos ϵ	25
3.3.1 Equivalencia entre AFND con y sin movimientos ϵ	27
Expresiones Regulares	29
4.2 Equivalencia entre AF y ER	30
4.3 Transformación de un AFD a ER	31
4.3.1 Simplificación de Expresiones	33
Propiedades de los Conjuntos Regulares	35
5.1 Limitaciones de los AF	35
5.1.1 Lema de Sondeo (<i>Pumping</i>)	36
5.2 Minimización de Estados de los AFD	37
5.3 Propiedades de Cerradura	41
5.3.1 Substituciones y Homomorfismos	41
5.3.2 Algoritmos de Decisión para Conjuntos Regulares	43
 Autómatas de Pila y Gramáticas Libres de Contexto	 46
Lenguajes Libres de Contexto	46
6.2 Árboles de Análisis Sintáctico	49
6.3 Simplificación de GLC	51

6.3.1	Forma Normal de Chomsky	53
6.3.2	Forma Normal de Greibach	54
6.4	Lema de Sondeo para Lenguajes libres de contexto	55
Autómatas de Pila		58
7.2	Autómatas de Pila y Lenguajes Libres de Contexto	60
Propiedades de los Lenguajes Libres de Contexto		64
8.2	Propiedades de Cerradura	64
8.3	Algoritmos de decisión para LLC	65
8.4	<i>Parsing</i>	66
Máquinas de Turing y Computabilidad Efectiva		70
Máquinas de Turing		70
9.2	El modelo básico de la Máquina de Turing	72
9.3	Funciones (y lenguajes) Computables	75
9.4	Extensiones al modelo básico	77
9.4.1	Multicintas	77
9.4.2	Cintas infinitas en ambas direcciones	78
9.4.3	No determinismo	78
9.5	Máquinas de Turing como enumeradores	79
9.6	La Máquina de Turing Universal	80
9.7	Reducción	83
9.8	El Teorema de Rice	84
A El Teorema de Incompletitud de Gödel		86
A.2	Prueba del Teorema	87

Preliminares Matemáticos

Este capítulo tiene el propósito de recordar y fortalecer las nociones que utilizaremos a lo largo del curso, así como unificar la notación.

1.1 Problemas de Decisión vs. Funciones

Recordemos que básicamente una función f entre dos conjuntos A y B es una correspondencia (denotada por $f : A \rightarrow B$) en la que no se permite que un mismo elemento $x \in A$ esté en correspondencia con más de un elemento $y \in B$.

Primero vamos a establecer una diferencia entre lo que denominaremos un problema de decisión y el concepto de función. Para ello, consideremos el siguiente

Ejemplo 1.1.1 *Queremos saber si es posible responder siempre (afirmativa o negativamente) a la pregunta: “¿existen n dígitos d consecutivos en la expansión decimal de π ?”.*

Una instancia de tal problema es el siguiente: ¿Existen $n = 298051$ dígitos $d = 3$ consecutivos?

3.1415... $\underbrace{333333333..3}_{n}$...

Una forma para poder ofrecer una solución al problema anterior sería la siguiente: existen algoritmos para obtener dígitos de la expansión decimal de π , de tal suerte que podríamos en principio plantear el algoritmo 1 (que se muestra en la parte superior de ésta página).

Sin embargo, hasta donde sabemos, podemos generar cuantos dígitos de π queramos, pues todo parece indicar que la expansión decimal de π es *infinita*; por lo que el anterior no sería propiamente un algoritmo ya que existirían valores de entradas (n, d) tales que el algoritmo no se detendría y regresaría una respuesta ya sea positiva o negativa. Por tanto, la moraleja es que problemas aparentemente inofensivos como el anterior tienen consecuencias profundas desde el punto de vista de la noción (intuitiva por el momento) de computabilidad.

Algoritmo 1 n dígitos d en la expansión decimal de π

Entrada: n, d

Salida: Existen o no n dígitos d en la expansión decimal de π

```
count  $\leftarrow$  0;
ban  $\leftarrow$  FALSO;
while ban=FALSO do
     $nxdig \leftarrow generaDigitodePi()$ ;
    if  $nxdig = d$  then
        count  $\leftarrow$  count + 1;
        if count =  $n$  then
            ban  $\leftarrow$  VERDADERO; “Si”
        end if
    else
        count  $\leftarrow$  0;
    end if
end while
```

Definición 1.1.1 *Un problema de decisión es una función con salida de 1 bit: “sí” o “no”. Para especificar un problema de decisión, se debe especificar:*

- un conjunto A de posibles respuestas,
- el subconjunto $B \subseteq A$ de instancias “sí”.

Ejemplo 1.1.2 *Propiedad: “Ver si un grafo² está conectado”(i.e., si cada uno de los nodos del grafo tiene arcos con todos los demás).*

- A : conjunto de todos los grafos
- B : subconjunto de A en que los grafos tienen todos sus nodos conectados.

1.1.1 Notación de Conjuntos

Recordemos que un conjunto es una *colección de objetos* (miembros) sin repetición. Denotaremos a los conjuntos por las letras mayúsculas romanas: A, B, C . Un conjunto A puede ser especificado de dos maneras:

²La definición de grafo se ofrece en la sec.1.3, pág.6.

1. *Enumerando*, si es posible, todos los elementos que lo componen:

$$A = \{a, b, c, d, \dots, z\}$$

2. Describiendo (sin ambigüedad) una *propiedad* que cumplen los elementos del conjunto:

$$A = \{x \in \text{Letras} \mid x \text{ es minúscula}\}$$

A la primer forma de describir un conjunto le denominamos **forma extensional** ya que menciona a todos y cada uno de sus elementos; mientras que a la segunda forma la denominamos **forma intensional** ya que sólo describimos la propiedad que caracteriza a los elementos del conjunto. Nótese que en el segundo caso indicamos explícitamente la existencia de un conjunto *Letras*, que en éste caso representa un **universo de discurso**³. Utilizaremos varios universos de discurso; en particular, el conjunto de los números naturales que denotaremos por \mathcal{N} . Por ejemplo, a continuación especificamos el conjunto de los números pares:

$$\text{Even} = \{i \in \mathcal{N} \mid i = 2j, \text{ para todo } j \in \mathcal{N}\}$$

Decimos que un conjunto A está **contenido** (o es un subconjunto de) en otro conjunto B si todo elemento de A también está en B y lo denotamos como $A \subseteq B$. Nótese que para cualquier conjunto, $A \subseteq A$. Dos conjuntos A, B son **iguales** si y sólo si tienen exactamente los mismos elementos, y también lo podemos expresar como $A = B$ si y sólo si $A \subseteq B$ y $B \subseteq A$. Además, para cualquier conjunto A , siempre ocurre que $\emptyset \subseteq A$.

Si A está contenido en B pero $B \neq A$ denominamos que A está contenido impropriamente en B y lo denotamos por $A \subset B$.

La **Cardinalidad** (número de elementos) de un conjunto A la denotamos por $|A|$. El conjunto vacío es el único con cardinalidad 0: $|\emptyset| = 0$.

Definición 1.1.2 Se definen las siguientes operaciones sobre conjuntos:

- *Unión:*

$$A \cup B \stackrel{\text{def}}{=} \{x \mid x \in A \text{ ó } x \in B\}$$

- *Intersección:*

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

- *Diferencia:*

$$A - B \stackrel{\text{def}}{=} \{x \mid x \in A \text{ pero } x \notin B\}$$

- *Producto Cartesiano:*

$$A \times B \stackrel{\text{def}}{=} \{(x, y) \mid x \in A \text{ y } y \in B\}$$

donde (x, y) es una pareja ordenada.

Definición 1.1.3 (Conjunto Potencia) Sea A un conjunto. El conjunto potencia de A se denota por 2^A (o también por $\mathcal{P}(A)$) y se define como el conjunto de todos los subconjuntos de A :

$$2^A \stackrel{\text{def}}{=} \{X \mid X \subseteq A\}$$

Ejemplo 1.1.3 Sean los conjuntos $A = \{a, b\}$ y $B = \{b, c\}$, entonces

$$\begin{aligned} A \cup B &= \{a, b, c\}, & A \cap B &= \{b\}, & A - B &= \{a\} \\ A \times B &= \{(a, b), (b, b), (a, c), (b, c)\}, \\ 2^A &= \{\emptyset, \{a\}, \{b\}, \{a, b\}\} \end{aligned}$$

1.1.2 Conjuntos Infinitos

Decimos que dos conjuntos A y B tienen la misma cardinalidad si podemos establecer una correspondencia *uno a uno* (biyectiva) entre los elementos de A con los de B . Si A y B son finitos, entonces si $A \subseteq B$ entonces $|A| \leq |B|$, y si $A \subset B$ entonces $|A| < |B|$. Sin embargo, en el caso de los conjuntos infinitos lo anterior no es cierto. Como bien sabemos, el conjunto de los números naturales \mathcal{N} es infinito, consideremos el conjunto de los números naturales pares *Even*, evidentemente $\text{Even} \subset \mathcal{N}$ pero podemos establecer una correspondencia biunívoca (uno a uno) entre ambos conjuntos: simplemente sea $f(2i) = i$ (de hecho es la definición que utilizamos anteriormente) por tanto los dos conjuntos \mathcal{N} y *Even* tienen la misma cardinalidad.

A los conjuntos que pueden ponerse en correspondencia uno a uno con los naturales los denominamos *infinitos contables* o simplemente contables. Obviamente todo conjunto finito se puede poner en correspondencia uno a uno con un subconjunto finito de los naturales. Sin embargo, existen conjuntos infinitos que no pueden ser puestos en correspondencia con los naturales, por ejemplo, $2^{\mathcal{N}}$ (el conjunto potencia de los naturales). Para ver por qué revisaremos el argumento inventado por Georg Cantor que lleva su nombre.

Diagonalización de Cantor

Supongamos que \mathcal{N} está en correspondencia con $2^{\mathcal{N}}$. Sea la función $\mu : \mathcal{N} \rightarrow 2^{\mathcal{N}}$ la que nos permite establecer tal correspondencia. Formemos un conjunto D de la siguiente manera

$$D = \{x \in \mathcal{N} \mid x \notin \mu(x)\}$$

³No es un conjunto universal!

claramente $D \subseteq \mathcal{N}$ y por tanto $D \in 2^{\mathcal{N}}$. Ello significa que D está en correspondencia con un elemento $d \in \mathcal{N}$: $\mu(d) = D, \dots$ Un momento! Eso significa que si $d \in D$ entonces $d \in \mu(d)$ y por tanto $d \notin D$, lo cual es una contradicción. Similarmente, si partimos del supuesto que $d \notin D$ entonces $d \notin \mu(d)$ por tanto $d \in D$, de nuevo una contradicción.

La presencia de contradicción significa que tal correspondencia μ **no puede existir**.

Resulta muy útil construir la siguiente tabla, en la que colocamos los elementos del conjunto \mathcal{N} en la columna de la izquierda y la evaluación de μ en el renglón superior. En la celda (i, j) colocamos un 1 si $i \in \mu(j)$ y un 0 de lo contrario:

	$\mu(0)$	$\mu(1)$	$\mu(2)$	\dots	$\mu(j)$	\dots
0	0	1	1	\dots	1	\dots
1	1	1	1	\dots	0	\dots
2	0	0	1	\dots	0	\dots
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
i	0	1	0	\dots	1	\dots
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots

(en la tabla, $0 \notin \mu(0)$, $1 \in \mu(0)$ y $i \in \mu(j)$).

El conjunto D se define recorriendo la diagonal de la tabla intercambiando 0's y 1's de tal forma que n se pone en D si y sólo si la entrada n -ésima de la diagonal es 0.⁴

Este principio se cumple en general para cualquier conjunto X (finito ó infinito).

Podemos resumir el método de la diagonalización de la siguiente manera: supóngase que $\chi_0, \chi_1, \chi_2, \dots$ es una enumeración de objetos del mismo tipo (funciones, números naturales, etc.). Podemos construir un objeto χ del mismo tipo que es distinto de cada χ_n utilizando el siguiente lema:

‘Hacer que χ y χ_n difieran en n ’.

Evidentemente, la interpretación del término *difieran* depende del tipo de objeto involucrado: las funciones pueden diferir en el argumento n siempre que estén definidas. En el caso de los conjuntos, la pregunta en n es si el elemento n pertenece o no al conjunto que se está construyendo.

Ejercicios 1.1.1 Utilice el argumento de la diagonalización de Cantor para demostrar que los naturales y los reales no pueden ponerse en correspondencia uno a uno.

1.1.3 Relaciones

Un concepto útil que se sustenta en el producto cartesiano es el de relación.

⁴Es importante que difiera de **todos** los elementos de la diagonal, no es suficiente asegurar que difiera de algunos.

Definición 1.1.4 (Relación Binaria) Una Relación Binaria R definida sobre un conjunto A es un conjunto de parejas tomadas de A : $R = \{(x, y) \mid x, y \in A\}$. Alternativamente, $R \subseteq A \times A$.

En general, una relación es cualquier subconjunto del producto cartesiano de los conjuntos involucrados; por ejemplo, la relación $A^n \subseteq A \times A \times A \times \dots \times A$, mientras que la relación $R_{A \times B} = \{(a, b) \mid a \in A, b \in B\}$ es un subconjunto de $A \times B$. Para éste último caso, A se denomina *dominio* de la relación y B es el *rango* ó *codominio*.

Para cualquier relación binaria R definida sobre un conjunto A a menudo representamos la pareja $(a, b) \in R$ como aRb . Decimos que R en A es:

1. Reflexiva: si se cumple que aRa para todo $a \in A$,
2. Simétrica: si aRb entonces bRa ,
3. Transitiva: si aRb y bRc entonces aRc ;
4. Antisimétrica: si aRb y bRa entonces $a = b$.

Ejemplo 1.1.4 Sea $A = \{a, b, c, d\}$ y la relación $R_1 = \{(a, a), (b, b), (c, c), (d, d), (a, b), (b, c), (a, c), (d, c)\}$. Podemos verificar que R_1 es reflexiva, no es simétrica pues $(b, a) \notin R_1$, si es transitiva pero no es antisimétrica.

Definición 1.1.5 Decimos que una relación binaria es:

1. **Relación de Equivalencia:** si es reflexiva, simétrica y transitiva.
2. **Relación de Orden Parcial:** si es transitiva, antisimétrica y reflexiva.

Ejemplo 1.1.5 Sean las siguientes relaciones definidas sobre $A = \{0, 1, 2\}$:

$$R_2 = \{(0, 0), (1, 1), (2, 2), (0, 1), (0, 2), (1, 2)\}$$

$$R_3 = \{(0, 0), (0, 1), (1, 0), (2, 0), (2, 2), (0, 2), (1, 2), (2, 1), (2, 2), (1, 1)\}$$

Se verifica inmediatamente que R_2 es relación de equivalencia, mientras que R_3 es relación de orden parcial.

Una propiedad importante de las relaciones de equivalencia definidas sobre un conjunto A es que particionan al conjunto A en *clases de equivalencia* no vacías $A = A_i \cup A_2 \cup \dots$ donde para cada i, j ($i \neq j$):

1. $A_i \cap A_j = \emptyset$,
2. para cada $a, b \in A_i$, aRb se cumple,
3. para cada $a \in A_i$ y $b \in A_j$, aRb no se cumple.

Ejemplo 1.1.6 Sea la relación definida sobre el conjunto de los estudiantes del grupo, tales que aRb si y sólo si a y b sacaron la misma calificación en el examen. Claramente R es una relación de equivalencia, las clases de equivalencia corresponden a cada una de las calificaciones: los aprobados, reprobados, etc.

Cerraduras

Si P es un conjunto de Propiedades de relaciones, la *Cerradura* $_P$ de una relación R es la relación más pequeña⁵ que incluye a todas las parejas de R y posee la propiedad P . Por ejemplo, la cerradura transitiva de R se denota por R^+ y se define por:

1. si $(a, b) \in R$ entonces $(a, b) \in R^+$,
2. si $(a, b) \in R^+$ y $(b, c) \in R$ entonces $(a, c) \in R^+$,
3. ninguna otra pareja está en R^+

Más formalmente: para toda $i > 0$

$$R_i^+ \stackrel{\text{def}}{=} R_{i-1} \cup \{(a, c) \mid (a, b) \in R_{i-1}^+, (b, c) \in R\}$$

$$(R_o \stackrel{\text{def}}{=} R)$$

Ejemplo 1.1.7 Sea $R = \{(1, 2), (2, 2), (2, 3)\}$, entonces $R^+ = \{(1, 2), (2, 2), (2, 3), (1, 3)\}$.

La cerradura *reflexiva y transitiva* de una relación binaria R se denota por R^* y se define por

$$R^* \stackrel{\text{def}}{=} R^+ \cup \{(a, a) \mid a \in A\}$$

1.2 Cadenas y Operaciones sobre cadenas

Nuestra primer abstracción es la siguiente:

siempre consideraremos el conjunto de posibles entradas para un problema de decisión como el conjunto de cadenas de longitud finita definidas sobre un alfabeto finito de símbolos.

Lo anterior lo podremos realizar uniforme y sencillamente: los grafos, los números naturales, los árboles, aún programas completos escritos en algún lenguaje podrán ser naturalmente **codificados** como

⁵La cerradura es una definición inductiva.

cadenas de símbolos. Gracias a ésta abstracción únicamente utilizaremos un tipo de datos y pocas operaciones básicas.

Definición 1.2.1 • Un **Alfabeto** Σ es cualquier conjunto finito de símbolos. Utilizaremos la convención de nombrar un alfabeto mediante la letra griega “sigma” mayúscula: por ejemplo $\Sigma = \{a, b, c, d\}$. A los elementos de Σ les denominaremos letras ó símbolos.

• Una **Cadena** (o palabra) definida respecto un alfabeto Σ es cualquier secuencia de longitud finita de símbolos tomados de Σ . Por ejemplo, si $\Sigma = a, b, c, d$ entonces “aabbcdada” es una cadena de Σ de longitud 8. Utilizaremos la convención de nombrar⁶ cadenas mediante u, v, wx, y, z .

• La **longitud** de una cadena w es el número de símbolos que lo componen y lo indicamos así $|w|$. Por ejemplo, si $w = aabbcdada$ entonces $|w| = 8$. La longitud de cualquier cadena siempre es un número natural.

• Existe una única cadena de longitud 0 en Σ y se denomina **cadena nula** ó **vacía**, y se denota por la letra griega⁷ “epsilon” ϵ . Por definición $|\epsilon| = 0$.

• Se denota por Σ^* al conjunto de todas las cadenas que se pueden construir a partir del alfabeto Σ . Por ejemplo,

$$\{a\}^* = \{\epsilon, a, aa, aaa, aaaa, \dots\}$$

$$\{a, b\}^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$$

También podemos denotar al conjunto $\{a\}^*$ por $\{a^n \mid n \geq 0\}$.

La siguiente convención es importante $\emptyset^* \stackrel{\text{def}}{=} \{\epsilon\}$, donde \emptyset denota al conjunto vacío. Si $\Sigma \neq \emptyset$ entonces Σ^* es un conjunto infinito de cadenas cada una de ellas de longitud finita.

Se debe tener cuidado entre los conjuntos y las cadenas:

$$\{a, b\} = \{b, a\} \quad \text{pero} \quad ab \neq ba$$

$$\{a, a, b\} = \{a, b\} \quad \text{pero} \quad aab \neq ab$$

Como precaución adicional, debe tenerse en cuenta que $\emptyset, \{\epsilon\}$ y ϵ son cosas diferentes: el primer conjunto no tiene elementos, el segundo es un conjunto con un sólo elemento y el último es una cadena, no un conjunto.

⁶A menudo, para evitar ambigüedad, encerraremos a un símbolo entre comillas: ‘ a' ’ $\in \Sigma$, y a una cadena entre comillas dobles: “ aba ”, siempre que las comillas **no** formen parte del alfabeto.

⁷No hay que confundirla con el símbolo matemático de “contenido” ó “pertenece a” \in .

1.2.1 Operaciones sobre cadenas

La **concatenación** toma 2 cadenas u, v y construye una nueva cadena uv colocando los símbolos de u y enseguida los correspondientes a v en el mismo orden. Nótese que uv y vu son en general diferentes.

Por ejemplo: sean $u = aab$, $v = abab$, entonces $uv = aababab$ mientras que $vu = ababaab$.

Algunas propiedades muy útiles de la concatenación son:

- Asociatividad: $(uv)z = u(vz)$ (los paréntesis son agrupadores y no forman parte de las cadenas).

Ejemplo: $z = bba$,

$$(uv)z = (aababab)bba = aabababbba$$

$$u(vz) = aab(ababbba) = aabababbba$$

- La cadena nula ϵ es neutro con respecto a la concatenación: $x(\epsilon) = (\epsilon)x = x$.
- $|xy| = |x| + |y|$ la longitud de la concatenación de 2 cadenas es la suma de las longitudes de cada una de ellas. Tenemos el siguiente caso especial de la anterior ecuación: $(a^m)(a^n) = a^{(m+n)}$ para cualquier $m, n \geq 0$.

Establecemos las siguientes convenciones:

- Si ' a' ' $\in \Sigma$ y $x \in \Sigma^*$, escribimos $\#a(x)$ para indicar el número de a 's en x . Por ejemplo, $\#0("001101001000") = 8$ y $\#1("0000") = 0$.
- Escribimos a^n para representar a la cadena formada por n ' a 's consecutivas; por ejemplo, $a^5 = aaaaa$, $a^1 = a$, $a^0 = \epsilon$.

1.2.2 Operaciones sobre conjuntos

A, B, C denotan conjuntos de cadenas (subconjuntos de Σ^*). Definimos las siguientes operaciones sobre conjuntos de Cadenas (además de las definidas anteriormente):

- Complemento en Σ^* :

$$\sim A \stackrel{def}{=} \{x \in \Sigma^* | x \notin A\}$$

- Concatenación:

$$AB \stackrel{def}{=} \{xy | x \in A \text{ y } y \in B\}$$

por ejemplo: $A = \{a, ab\}$, $B = \{b, ba\}$, $AB = \{ab, aba, abb, abba\}$, se deben involucrar *todas* las cadenas. En general $AB \neq BA$: para el ejemplo $\{ab, aba, abb, abba\} \neq \{ba, bab, baa, bab\}$.

- Potencia: la potencia n -ésima A^n de un conjunto A se define *inductivamente*:

$$A^0 \stackrel{def}{=} \{\epsilon\}$$

$$A^{(n+1)} \stackrel{def}{=} AA^n$$

En otras palabras, A^n se forma concatenando n copias de A consigo misma. Nótese que $A^m A^n = A^{m+n}$. Por ejemplo:

$$\begin{aligned} \{ab, aab\}^0 &= \{\epsilon\} \\ \{ab, aab\}^1 &= \{ab, aab\} \\ \{ab, aab\}^2 &= \{abab, abaab, aabab, aabaab\} \\ \{ab, aab\}^3 &= \{ababab, ababaab, abaabab, abaabaab, aababab, aababaab, aabaabaab, aabaabaab\} \end{aligned}$$

De igual forma:

$$\{a, b\}^n = \{x \in \{a, b\}^* \mid |x| = n\}$$

que son todas las cadenas tomadas del alfabeto $\{a, b\}$ de longitud n .

- Cerradura de Kleene: A^* es la unión de **todas** las posibles potencias finitas de A :

$$A^* \stackrel{def}{=} \bigcup_{n \geq 0} A^n$$

$$= A^0 \cup A^1 \cup A^2 \cup A^3 \dots$$

También se define como:

$$A^* = \{a_1 a_2 a_3 \dots a_n \mid n \geq 0, a_i \in A, 1 \leq i \leq n\}$$

Cuando $n = 0$ entonces ϵ está siempre en A^* para cualquier A . Ejemplo:

$$\{aab, baa\}^* = \{\epsilon, aabbaa, aabaab, \dots\}$$

En general, A^* es un conjunto infinito siempre que $A \neq \emptyset$.

- Cerradura Positiva: A^+ para un conjunto A es la unión de todas las potencias de A distintas de cero:

$$A^+ \stackrel{def}{=} AA^* = \bigcup_{n \geq 1} A^n$$

Ejercicios 1.2.1 Sea el alfabeto $\Sigma = \{a, b, c\}$ y los conjuntos $A = \{aab, bba\}$, $B = \{aa, aaaa, ab\}$; obtenga: Σ^* , $A \cup B$, $A \cap B$, $\sim A$, AB , A^2 .

Algunas propiedades importantes de las anteriores operaciones son:

- La Unión, Intersección y Concatenación son asociativas:

$$\begin{aligned}(A \cup B) \cup C &= A \cup (B \cup C) \\ (A \cap B) \cap C &= (A \cap B) \cap C \\ (AB)C &= A(BC)\end{aligned}$$

- Conmutatividad en unión e intersección:

$$\begin{aligned}A \cup B &= B \cup A \\ A \cap B &= B \cap A\end{aligned}$$

- El conjunto \emptyset es identidad para la unión:

$$A \cup \emptyset = A = \emptyset \cup A$$

- El conjunto $\{\epsilon\}$ es identidad para concatenación:

$$\{\epsilon\}A = A\{\epsilon\} = A$$

- El conjunto \emptyset es Aniquilador para concatenación:

$$A\emptyset = \emptyset A = \emptyset$$

- La concatenación se distribuye sobre la unión:

$$\begin{aligned}A(B \cup C) &= AB \cup AC \\ (A \cup B)C &= AC \cup BC \\ A(\bigcup_{i \in I} B_i) &= \bigcup_{i \in I} AB_i\end{aligned}$$

(en el último caso, I se denomina *conjunto de índices* y a todos los B_i se denomina *familia de conjuntos indexados*).

- La concatenación de conjuntos no se distribuye con respecto a la intersección; por ejemplo: sean $A = \{a, ab\}, B = \{b\}, C = \{\epsilon\}$, calcule $A(B \cap C)$ y entonces calcule $AB \cap AC$. Se podrá apreciar que los resultados son diferentes.
- La cerradura Kleene satisface las siguientes propiedades:

$$\begin{aligned}A^*A^* &= A^* \\ A^{**} &= A^* \\ A^* &= \{\epsilon\} \cup AA^* = \{\epsilon\} \cup A^*A \\ \emptyset^* &= \{\epsilon\}\end{aligned}$$

1.3 Grafos

Un grafo G se denota por una tupla $G = (V, E)$ y consiste en

- V es un conjunto finito de vértices (o nodos)
- E es un conjunto finito de aristas, las cuales son parejas de vértices: (v_i, v_j) .

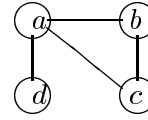


Figura 1.1: Grafo G_1 .

Por ejemplo: sea $G_1 = (V_1, A_1)$ donde $V_1 = \{a, b, c, d\}$, $A_1 = \{(a, b), (b, c), (a, c), (a, d)\}$. En la Figura 1.1 se muestra a G_1 .

Una *ruta* en un grafo es una secuencia de vértices v_1, v_2, \dots, v_k con $k \geq 1$, tal que existe una arista (v_i, v_{i+1}) para cada i ($1 \leq i \leq k$). Si $v_1 = v_k$ entonces la ruta se denomina *ciclo*. La *longitud* de la ruta es $k - 1$. Por ejemplo, d, a, b es una ruta de longitud 2.

1.3.1 Grafos Dirigidos

Un Grafo Dirigido también se denota por $G = (V, E)$, (con V como anteriormente) pero el conjunto E está compuesto por *parejas ordenadas* de vértices denominadas *arcos*. Denotamos un arco por $v_i \rightarrow v_j$ para indicar que v_i puede alcanzar a v_j (pero v_j no necesariamente podrá alcanzar a v_i a menos que esté explícitamente un arco correspondiente). Por ejemplo: $G_2 = (V_1, E_2)$ donde $E_2 = \{(a, c), (b, c), (c, a), (d, b)\}$. En la Figura 1.2 mostramos a G_2 .

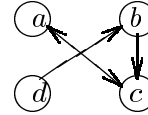


Figura 1.2: Grafo G_2 .

Las rutas se definen de manera similar a los grafos no dirigidos, salvo que seguimos la convención de indicar un arco como $v_i \rightarrow v_{i+1}$. En G_2 , la ruta $d \rightarrow b \rightarrow c \rightarrow a$ tiene tamaño 3. Si $u \rightarrow v$ es un arco, decimos que u es el *predecesor* de v y v es el *sucesor* de u .

Alternativamente, un grafo dirigido se puede definir a partir de una relación binaria definida sobre el conjunto de vértices.

1.3.2 Árboles

Un árbol es un grafo dirigido con las siguientes propiedades:

1. Existe un único vértice denominado raíz del cual existe una ruta a todo vértice; la raíz no tiene predecesor.

2. Cada vértice u distinto de la raíz tiene exactamente un predecesor (nodo padre de u).
3. Los sucesores de cada vértice están ordenados partiendo de la izquierda.

NOTA: El árbol con un solo nodo, tiene la característica de que dicho nodo es hoja y raíz a la vez.

Un árbol binario a lo mas tiene 2 sucesores.

El nivel de un nodo es la longitud de la ruta desde la raíz hacia el nodo. La altura (profundidad) de un árbol se define como la longitud de la ruta más larga partiendo desde la raíz, o como el mayor nivel. Un subárbol de un árbol a partir de un nodo interno u está formado por todos los nodos descendientes de u .

Existen dos formas generales para *visitar* ó recorrer los nodos de un árbol⁸: partiendo de la raíz, y repetidamente

- Primero en Profundidad: se visitan en orden (de izquierda a derecha) los subárboles (hasta alcanzar las hojas). En otras palabras, primero se recorren todos los nodos bajo una ruta hasta alcanzar una hoja.
- Primero en Amplitud: se visitan en orden (de izquierda a derecha) los nodos sucesores, por nivel antes de continuar al siguiente nivel.

Como ejemplo, la fig1.3 muestra el árbol B_1 . En este árbol, la raíz es a , los nodos interiores son a, b, c, d, f, i y las hojas son e, l, g, h, k, j . La longitud de la ruta desde la raíz hacia el nodo g es 2. La profundidad del árbol es 3. Un subárbol es d, h, i, j, k (partiendo de d). El recorrido en profundidad es $a, b, e; f, l; c, g; d, h; i, k; j$ mientras que por amplitud es $a; b, c, d; e, f, g, h, i, j; l, k$.

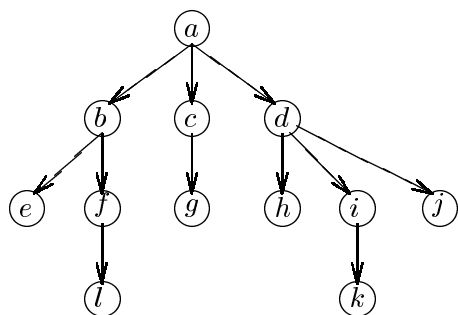


Figura 1.3: Árbol B_1 .

⁸Que son los mismos en general para grafos.

1.4 Principio de Inducción Matemática

Uno de los conceptos fundamentales es el de la inducción. Intuitivamente, podemos considerar a la inducción sobre los números naturales como un **método para escribir una prueba infinita en una forma finita**. Esta idea puede extenderse a otras estructuras, lo importante es hacer notar que para poder aplicar inducción en una estructura es importante que:

- los elementos de la estructura mantengan una relación de *orden parcial*,
- existan ciertos elementos denominados *minimales* los cuales son aquellos que si hay otros más pequeños, solo pueden ser ellos mismos.

Los minimales establecen una *condición de tope*, para evitar que se tengan ciclos infinitos bajo la relación de orden.

Nos enfocaremos sin pérdida de generalidad principalmente a la inducción definida sobre los naturales.

Intuitivamente, el propósito de una demostración es verificar si cierta estructura (conjunto) cumple con una propiedad, es decir, una demostración permite construir una *prueba* la cual ofrece justificación a la afirmación de que la estructura cumple o no con la propiedad. Efectivamente, lo podemos ver como un problema de decisión. Supóngase que tenemos la propiedad P que queremos verificar si se cumple o no para *todo número natural*. Por ejemplo, $P(n)$ podría ser “ n es par ó impar”. Una forma para demostrar $P(n)$ para toda $n \in \mathcal{N}$ podría ser la siguiente: si tenemos una cantidad infinita de tiempo y hojas de papel para escribir, es demostrar que se cumple P en $n = 0$: $P(0)$; a continuación hacer lo mismo para $n = 1$: $P(1)$; etc. Evidentemente, esta forma no es factible, pero si lo fuera el resultado sería la demostración de que $P(n)$ se cumple para todo n . Lo valioso de la inducción es que ofrece una forma finita y simple de demostrar lo anterior.

La forma típica de inducción en los naturales es:

Primera Forma de Inducción en los Naturales

Para demostrar que $P(n)$ es verdadero (se cumple) para todo natural n , es suficiente demostrar $P(0)$ y demostrar que para cualquier $m \in \mathcal{N}$: Si $P(m)$ se cumple implica que P se cumple para *el sucesor inmediato de m* , i.e. $P(m + 1)$ también se cumple.⁹

⁹Alternativamente, Si $P(m - 1)$ se cumple implica $P(m)$ también se cumple.

Esas dos partes reciben un nombre:

1. $P(0)$ se denomina **Base** ó **Caso Base**.
2. $\underbrace{P(m)} \rightarrow P(m+1)$ es el **Paso inductivo**.

La suposición de ‘si $P(m)$ es verdadero’ se denomina *Hipótesis de Inducción (HI)*.

Podemos utilizar la siguiente plantilla:

1. *Meta*: demostrar $P(n)$ para todo $n \in \mathcal{N}$.
2. *Base*: demostrar $P(0)$.
3. *Paso inductivo*: demostrar que para cualquier natural m , Si $P(m)$ implica $P(m+1)$.

Ejemplo 1.4.1 Ofreceremos una prueba por inducción para la propiedad siguiente:

$$\sum_{i=0}^n i = \frac{n(n+1)}{2}$$

siguiendo la plantilla anterior:

1. Caso base: $n = 0$

$$\sum_{i=0}^0 i = 0 = 0(1)/2$$

2. Paso inductivo: nuestra hipótesis de inducción es que la ecuación se cumple para cualquier m , por demostrar que se cumple para $m+1$:

$$\begin{aligned} \sum_{i=0}^{m+1} i &= \sum_{i=0}^m i + (m+1) \\ &= \underbrace{0+1+2+3+\dots+m}_{\frac{m(m+1)}{2}} + (m+1) \\ &= \frac{m(m+1)}{2} + (m+1) \text{ por HI} \\ &= \frac{m(m+1)+2(m+1)}{2} \\ &= \frac{(m+1)(m+2)}{2} \end{aligned}$$

que es justamente a lo que queríamos llegar. Por tanto, la prueba está finalizada.

Nótese que el aspecto importante en éste ejemplo fué aprovechar las propiedades de las sumas, de tal suerte que la escribimos en dos partes una de las cuales podemos aplicar nuestra hipótesis de inducción para concluir con lo que deseamos obtener.

Ejemplo 1.4.2 Demostraremos ahora que

$$\sum_{i=1}^n 2i - 1 = n^2$$

1. Caso base: $n = 1$

$$\sum_{i=1}^1 2i - 1 = 2(1) - 1 = 1 = 1^2$$

2. Paso inductivo: ahora nuestra hipótesis de inducción es suponer que se cumple la ecuación para cualquier m . Por demostrar que se cumple para $m+1$

$$\begin{aligned} \sum_{i=1}^{m+1} 2i - 1 &= \sum_{i=1}^m \underbrace{2i - 1}_{2i - 1} + 2((m+1) - 1) \\ &= m^2 + 2m + 2 - 1 \text{ por HI} \\ &= (m+1)^2 \end{aligned}$$

Finalizaremos ésta sección con un ejemplo muy ilustrativo

Ejemplo 1.4.3 Demostraremos que para cualquier conjunto finito A , la cardinalidad de su conjunto potencia¹⁰ es $2^{|A|}$.

Aplicaremos inducción sobre la cardinalidad de A :

Supóngase que ordenamos a todos los subconjuntos de A de acuerdo a su cardinalidad, desde el más pequeño hasta el mismo A (que es el mayor, ya que la contención es propia):

subconjunto	cardinalidad
\emptyset	0
$\{a\}$	1
$\{a, b\}$	2
\dots	\dots
$\{a_1, \dots, a_n\}$	n

1. Caso Base: $n = 0$, es decir, $A = \emptyset$ pues es el único con $|A| = 0$. Por tanto $2^A = \{\emptyset\}$ y $|2^A| = 1$ que corresponde con $2^0 = 1$.

2. Paso Inductivo: para $n > 0$ nuestra H.I. es la siguiente: asúmase que para cualquier conjunto de n elementos, la cardinalidad de su conjunto potencia es 2^n . Por demostrar que para un conjunto de $n+1$ elementos, la cardinalidad de su conjunto potencia es 2^{n+1} .

Conviene utilizar la siguiente idea para la representación del conjunto potencia: como el conjunto A es finito con n elementos, representaremos el conjunto potencia de A mediante una matriz de bits, cada columna representa cada uno de los elementos de A , y cada renglón indica el correspondiente subconjunto: si el bit en la posición j está encendido significa que en el correspondiente subconjunto X_i el elemento a_j está en ese subconjunto.

¹⁰No confundirse, 2^A es una forma de denotar el conjunto potencia de A mientras que $2^{|A|}$ es la expresión de 2 elevado a la cardinalidad de A , que a fin de cuentas es un número.

Subconjunto	Elementos de A				
	a_1	a_2	a_3	\dots	a_n
\emptyset	0	0	0	\dots	0
$\{a_1\}$	1	0	0	\dots	0
$\{a_1, a_2\}$	1	1	0	\dots	0
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
$\{a_1, a_2, \dots, a_n\}$	1	1	1	\dots	1

Todas las posibles combinaciones indican todos los posibles subconjuntos que se pueden obtener de A . Nótese que los podemos ordenar de acuerdo a su cardinalidad: todos los bit apagados (puestos a 0) en el primer renglón, hasta todos encendidos (en 1) en el último renglón. Es fácil apreciar que el número total de renglones de la matriz es 2^n , lo cual en nuestra prueba es nuestra H.I.

Sea B un conjunto finito de $n + 1$ elementos. Sea b cualquiera de sus elementos. La siguiente afirmación es obvia:

$$2^{|B - \{b\}|} = 2^n$$

ya que $(B - \{b\})$ denota al conjunto en el cual eliminamos a b , ese subconjunto tiene n elementos, y por la H.I. la cardinalidad de su conjunto potencia es 2^n .

Entonces, tomando A como cualquiera de esos subconjuntos $(B - \{b\})$ para cada b , hay 2 subconjuntos de acuerdo a la representación matricial de bits: cuando el bit que representa a b esta encendido y cuando esta apagado. En otras palabras, es como si tuviéramos la misma matriz de n para cada columna asociada a b . Entonces,

$$|2^B| = 2|2^{(B-b)}| = 2(2^n) = 2^{(n+1)}.$$

Segunda Forma de Inducción en los Naturales

Existe una forma equivalente de Inducción que tiene una H.I. con mayor información: para demostrar que una propiedad P se cumple para $n+1$ no sólo a partir de que se cumplió $P(n)$ si no también que se han cumplido $P(0), P(1) \dots P(n-1)$.

Para demostrar que $P(n)$ se cumple para todo natural n , es suficiente demostrar que para cualquier $m \in \mathcal{N}$, si $P(i)$ es verdadera para todo $i < m$, entonces $P(m)$ debe también ser verdadero. Esta segunda forma de inducción se denomina *Completa* ó *Fuerte*¹¹.

¹¹Aunque en realidad no es más fuerte ó completa que la primer forma.

En ésta forma de inducción no hay caso base, únicamente el paso inductivo. Ahora la HI es asumir que a partir de $P(i)$ para todo $i < m$, se sigue que $P(m)$ se cumple. Claro que en particular, cuando $i = 0$ se tiene el caso base de la primer forma de inducción.

La segunda forma de inducción a menudo resulta ser más conveniente para responder a la sig. pregunta ¿cómo extender la técnica de inducción para aplicarla a otros conjuntos (no necesariamente los naturales)? Para ello necesitamos establecer una *correspondencia* entre los elementos del conjunto A que deseamos verificar la propiedad y los naturales.

Ilustremos la idea con un ejemplo:

Ejemplo 1.4.4 *Queremos demostrar la siguiente propiedad: ‘El número de hojas de un árbol binario A es mayor a lo más en 1 que el número de nodos internos’.*

Podemos convertir una propiedad P de los árboles binarios en una propiedad de los naturales utilizando una función $f : A \rightarrow \mathcal{N}$ de la siguiente manera:

$Q(n) \stackrel{def}{=} \text{para todo } a \in A, \text{ si } f(a) = n \text{ entonces } P(a) \text{ se cumple.}$

Para el ejemplo,

$P(t) \stackrel{def}{=} \text{si el árbol } t \text{ tiene } m \text{ nodos internos, su número de hojas es a lo más } m + 1.$

$Q(n) \stackrel{def}{=} \text{para todo } t, \text{ si } altura(t) = n \text{ entonces se cumple } P(t).$

Demostraremos que para todo n se cumple $Q(n)$ utilizando la segunda forma de inducción debido que los subárboles de un nodo interno pueden tener alturas distintas.

Para cualquier $n \in \mathcal{N}$ asumiremos que para cualquier $i < n$ y cualquier árbol binario t , si $altura(t) = i$ entonces se cumple $P(t)$. Demostraremos entonces que para todo árbol binario t si $altura(t) = n$ entonces se cumple $P(t)$.

Tenemos los siguientes casos:

1. Caso $n = 0$, si el árbol binario t tiene $altura(t) = 0$ se cumple trivialmente $P(t)$, ya que significa que t es un árbol binario vacío ó sólo tiene una hoja (que es la misma que la raíz).
2. Caso $n > 0$: cualquier árbol t de altura n debe consistir de un nodo interno con 2 subárboles t_1 y t_2 . Las alturas de tanto t_1 y t_2 son menor a n por tanto podemos asumir que la HI se aplica: $P(t_1)$ y $P(t_2)$ se cumplen

Claramente, el número de hojas de t es la suma

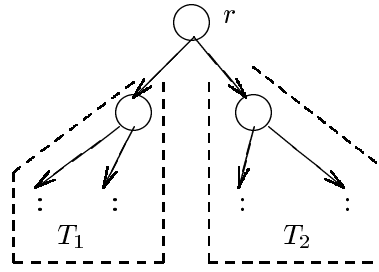


Figura 1.4: Esquema de árbol binario para la prueba del ejemplo 1.4.4.

del número de hojas de t_1 y t_2 (ver Figura 1.4):

$$\begin{aligned} \text{hojas}(t) &= \text{hojas}(t_1) + \text{hojas}(t_2) \\ \text{hojas}(t) &\leq \text{nodos}(t_1) + 1 \\ &\quad + \text{nodos}(t_2) + 1 \\ \text{entonces:} \\ \text{hojas}(t) &\leq \text{nodos}(t) + 1. \end{aligned}$$

Inducción estructural

Finalizaremos éste capítulo con un ejemplo que esencialmente aplica inducción en cadenas de símbolos. Ésta forma de aplicar inducción es muy común en prácticamente toda prueba en los siguientes capítulos.

Ejemplo 1.4.5 Las palabras palíndromas definidas sobre el alfabeto $\Sigma = \{a, b\}$ se pueden describir como:

1. L_1 : Una cadena que se lee igual de izquierda a derecha que viceversa.
2. L_2 : Una cadena que se puede construir a partir de la siguiente definición:
 - (a) La cadena vacía es palíndroma.
 - (b) Si a es un símbolo del alfabeto, la cadena " a " es palíndroma.
 - (c) Si a es un símbolo del alfabeto, y x es palíndroma entonces " axa " es palíndroma.
 - (d) Ninguna otra palabra es palíndroma.

Demostraremos que las definiciones L_1 y L_2 son equivalentes, o en otras palabras, que el conjunto de palabras que describen es el mismo: $L_1 = L_2$.

Recordemos que para demostrar la igualdad entre dos conjuntos, debemos probar contención en ambos sentidos, en éste caso $L_1 \subseteq L_2$ y $L_2 \subseteq L_1$. El segundo caso es inmediato, por lo que mostraremos el primer caso.

Conviene apreciar que para aplicar inducción, podemos ordenar las cadenas que se definen (en L_1 ó en L_2) de acuerdo a su longitud:

cadena(s)	longitud
ϵ	0
a, b	1
aa, bb	2
aba, bab	3
\vdots	\vdots

Demostraremos que $L_1 \subseteq L_2$:

Aplicamos inducción sobre la longitud de cualquier cadena que se lee igual de izquierda a derecha y viceversa. Sea w esa cadena.

- Si $|w| \leq 1$ entonces $w = \epsilon$ ó $w = "a"$, ó $w = "b"$ y claramente w es palíndroma por (a) y (b).
- Si $|w| > 1$ entonces como se lee igual de izquierda a derecha y de derecha a izquierda entonces w comienza y termina con el mismo símbolo, es de la forma $w = ava$ ó $w = bvb$. Aplicando la hipótesis de inducción sobre la cadena v se concluye que w es palíndroma por (c) y (d).

De esa forma, toda cadena descrita por la definición L_1 corresponde a la definición L_2 .

Ejercicios 1.4.1

1. Dado el alfabeto $\Sigma = \{a, b, c\}$ y los conjuntos $A = \{a, aab, aaaab, aaaba, aaabba\}$, $B = \{\epsilon, aab, bbc, cca, bbc, aac\}$ y $C = \{cba, cccb, ccbcc, ccacc, aacbb\}$, obtenga los siguientes:

- (a) $A \cup B, A \cup C, B \cup C$.
- (b) $A \cap B, B \cap C$
- (c) AB, BC
- (d) B^3
- (e) $AB \cap AC$.

2. Demuestre las siguientes propiedades aplicando inducción matemática:

(a)

$$\sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

(b)

$$\sum_{i=0}^n i^3 = \left(\sum_{i=0}^n i \right)^2$$

- (c) El número de hojas de un árbol binario es a lo más 2^n , para n la altura del árbol.
- (d) Un árbol binario balanceado completo es un árbol binario en el cual todos sus nodos internos tienen exactamente dos descendientes. Demuestre que el número de nodos tanto internos como hojas está dado por la sig. expresión:

$$\text{nodos} = \sum_{i=0}^n 2^i$$

donde n es la altura del árbol.

- (e) Demuestre por inducción sobre la longitud de las cadenas que las siguientes definiciones corresponden a la misma clase de cadenas:
- una cadena w sobre el alfabeto $\{ (,) \}$ está balanceada si y sólo si:
 - w tiene igual número de ‘(’ que de ‘)’,
 - cualquier prefijo (subcadena tomada por la izquierda) de w tiene un exceso de ‘(’ que de ‘)’.
 - ϵ está balanceada,
 - Si w está balanceada, ‘(w)’ también está balanceada,
 - Si w y x están balanceadas, también lo está wx ;
 - Ninguna otra cadena del alfabeto está balanceada.

3. Demuestre que las siguientes son relaciones de equivalencia:

- La relación en enteros R_1 definida como iR_1j si y sólo si $i = j$.
- La relación en personas R_2 definida por aR_2b si y sólo si a y b nacieron el mismo día del mismo año.

4. ¿Es infinita la unión de una colección infinita contable de conjuntos infinitos contables? ¿Lo es su producto cartesiano?.

La literatura revisada para éste capítulo es [HU79, BC94, Koz96, Win93, Smu95] y [Mit96].

Autómatas Finitos y Conjuntos Regulares

Autómatas Finitos Determinísticos

¿Que se nos ocurre cuando escuchamos la palabra “autómata”? Esta es una pregunta que siempre he realizado a los alumnos al tocar el tema. Algunas de las ideas que ellos expresan son:

- Algo *autónomo* que se comporta de determinada manera siempre.
- Un dispositivo que siempre hace lo mismo ante la misma situación.
- Tiene “estados” y “cambia” de estado en estado, por ejemplo, un elevador, las puertas automáticas (‘abierto’-‘cerrado’) ó un conmutador.

Lo que sí podemos asegurar es que independientemente de la definición que demos, un autómata es un sistema que lo caracterizamos mediante *estados y transiciones de estados* (ambos conjuntos finitos).

Los Autómatas Finitos (AF) que discutiremos son modelos matemáticos de un sistema que recibe entradas (estímulos) discretas y proporciona salidas (acciones) discretas. El sistema puede estar en cualquiera de un conjunto finito de estados. Intuitivamente, un *estado* es una descripción instantánea del sistema, una especie de fotografía de la realidad congelada en un instante dado. Un estado sintetiza toda la información relevante con respecto a las entradas que haya recibido, tal información es necesaria para determinar cómo evolucionará el sistema en las subsecuentes entradas ó estímulos. Por ejemplo, el mecanismo de control de un elevador básico es un sistema de estado finito: no necesita recordar todas las solicitudes previas, simplemente saber en qué piso se encuentra en un momento dado, cual es la dirección de movimiento (arriba-abajo) y la serie de solicitudes aún no satisfechas.

La evolución del comportamiento del sistema es como consecuencia del cambio de estado. Los cambios de estado se denominan *transiciones*, las cuales suceden de manera espontánea ó en respuesta a estímulos externos. Asumiremos que las transiciones ocurren de manera instantánea, lo cual es evidentemente una abstracción matemática, ya que en realidad, las transiciones toman tiempo para realizarse.

Un sistema que consiste únicamente de un número finito de muchos estados y transiciones entre estados se denomina *sistema de transición de estado finito ó simplemente sistema de estado finito*.

Existen innumerables ejemplos de sistemas de transición de estado en el mundo real: circuitos electrónicos, relojes, elevadores, juegos, etc. En las ciencias de la computación se encuentran muchos ejemplos de sistemas de estado finito (tanto en hardware -circuitos- como en software -editores y analizadores léxicos, éstos últimos parte fundamental de un compilador-) por lo que la teoría de autómatas es una herramienta de diseño y análisis muy útil. Una computadora en sí puede verse como un sistema de estado finito dado que la memoria (tanto RAM como en disco), el estado del procesador, etc. están en algún estado de un conjunto de estados muy grande pero finito. Evidentemente, ésta consideración impone una severa restricción artificial en la capacidad de almacenamiento, trayendo con ello que no sea satisfactorio, matemática o realísticamente: no se captura la esencia de la computación.

El cerebro humano también podría considerarse como un sistema de estado finito, (probablemente el número de neuronas es del orden de 2^{35}) bajo la hipótesis que el estado de cada neurona pueda describirse mediante algunos bits. No obstante, el enfoque de considerar el cerebro como un sistema de estado finito no ofrece utilidad considerable para entender su comportamiento o realizar observaciones importantes.

Quizá la razón más importante para estudiar sistemas de estado finito es la naturalidad del concepto: ocurre en muchos ámbitos muy diversos. Ello indica que el modelo de autómatas permite capturar la noción de una clase de sistemas fundamental.

Primero discutiremos un ejemplo intuitivo para familiarizarnos con el concepto, y entonces analizaremos las definiciones formales.

Ejemplo 2.1.2 *En la orilla de un río se encuentra un hombre, junto con un lobo, una oveja y paja. Hay un bote con la capacidad suficiente para llevar al hombre y a uno de los otros tres. El hombre con la paja, y demas compañeros deben cruzar el río, y el hombre puede llevar a uno solo a la vez. Sin embargo, si el*

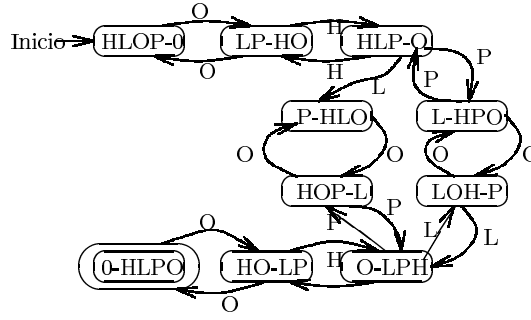


Figura 2.5: Diagrama de transiciones para el problema de cruzar el río.

hombre deja solos al lobo y a la oveja en cualquier lado del río, con toda seguridad que el lobo se comerá a la oveja. Del mismo modo, si la oveja y la paja se quedan juntas, la oveja se comerá a la paja. ¿Es posible que se pueda cruzar el río sin que nadie sea comido?

Podemos modelar el problema apartir de la observación que la información necesaria es en **dónde** se encuentran los ocupantes después de que se efectúa la transición (acción) de cruzar el río: el problema se habrá resuelto cuando se haya elegido una secuencia de transiciones que permita tener a todos los pasajeros en el extremo contrario del río del cual se comenzó. Por tanto, describiremos los estados mediante parejas de conjuntos separadas por un guión, cada conjunto representa a quienes están en esa orilla del río: Hombre (H), Oveja (O), Paja (P), Lobo (L). Así, el **estado inicial** del problema es la tupla

$$\langle H, L, O, P - \emptyset \rangle$$

Nótese que en la parte derecha de la tupla se tiene el conjunto vacío para indicar que ninguno de los pasajeros está en ese extremo del río.

Algunos estados son fatales y deben evitarse, por ejemplo

$$\langle L, O - H, P \rangle$$

ya que el lobo se come a la oveja.

Como mencionamos, las acciones que el hombre toma para atravesar el río son las **transiciones de estados**: puede cruzar solo en la barca (H) ó elegir como pasajero al lobo (L), a la oveja (O) ó a la paja (P). Por ejemplo

$$\langle H, L, O, P - \emptyset \rangle \xrightarrow{L} \langle O, P - H, L \rangle$$

indica que el hombre tomó al lobo como acompañante para cruzar el río. Podemos nombrar a cada

transición indicando el pasajero adicional exclusivamente¹².

La meta, propiamente el **estado final** ó **estado de aceptación** en el cual el problema está resuelto es justamente el *dual* al estado inicial, i.e. todos los pasajeros en el otro extremo del río:

$$\langle \emptyset - H, L, O, P \rangle$$

Para resolver el problema se tiene que elegir una secuencia de movimientos (atravesar el río) que a partir del estado inicial se alcance el estado final. Esa secuencia se puede representar *concatenando* las iniciales de cada pasajero que atravesó el río, i.e. las acciones (transiciones) que se eligen.

Una posible secuencia (solución al problema) es la siguiente :

$$\begin{aligned} &\langle H, L, O, P - \emptyset \rangle \xrightarrow{O} \langle L, P - H, O \rangle \\ &\langle L, P - H, O \rangle \xrightarrow{H} \langle H, L, P - O \rangle \\ &\langle H, L, P - O \rangle \xrightarrow{P} \langle L - O, H, P \rangle \\ &\langle L - O, H, P \rangle \xrightarrow{O} \langle H, L, O - P \rangle \\ &\langle H, L, O - P \rangle \xrightarrow{L} \langle O - H, L, P \rangle \\ &\langle O - H, L, P \rangle \xrightarrow{H} \langle H, O - L, P \rangle \\ &\langle H, O - L, P \rangle \xrightarrow{O} \langle \emptyset - H, L, P \rangle \end{aligned}$$

Lo cual podemos indicar de la siguiente manera:

$$\langle H, L, O, P - \emptyset \rangle \xrightarrow{*} \langle \emptyset - H, L, O, P \rangle$$

Podemos representar el problema mediante un **grafo de transiciones**, como se indica en la Figura 2.5, en la cual se puede observar todas las posibles elecciones que estando en algún estado se puedan tomar. El estado final se encuentra resaltado. Puede apreciarse que existen dos soluciones que son las más cortas (i.e., tienen el mismo tamaño de la ruta

¹²Obviamente, ni el lobo ni la oveja y mucho menos la paja pueden controlar la barca...

correspondiente) pero en general hay un número infinito de soluciones al problema las cuales involucran pasar las veces que se quiera por los mismos estados (i.e *ciclos inútiles*).

Ahora sí estamos listos para analizar las definiciones formales.

Definición 2.1.1 (AFD) *Un Autómata Finito Determinístico M es una estructura representada por la tupla*

$$M = \langle Q, \Sigma, \delta, q_o, F \rangle$$

en donde

- Q es un conjunto finito de estados: $\{q_o, q_1, q_2, \dots, q_n\}$
- Σ es un conjunto finito de símbolos: alfabeto.
- δ es la función de transición que indica el siguiente estado a partir del estado actual y leyendo un símbolo de entrada:

$$\delta : Q \times \Sigma \longrightarrow Q$$

Para cada símbolo de entrada, existe a lo más una transición que sale del estado actual y lleva a otro estado (posiblemente el mismo).

- q_o es el estado inicial en el cual el AF comienza su ejecución.
- $F \subseteq Q$ es el conjunto de estados finales (ó también, estados de aceptación).

El funcionamiento de un AFD M es bastante simple: en cualquier instante M se encuentra en algún estado q de Q y tiene una cabeza lectora para examinar la secuencia de símbolos (tomados de Σ) escritos sobre la cinta de entrada (ver Figura 2.6):

- Inicialmente, M parte del estado inicial q_o , y la cabeza se encuentra sobre el primer símbolo situado en el extremo izquierdo de la cinta.
- Estando el AF en el estado q y la cabeza lectora leyendo el símbolo de entrada actual a , mediante un movimiento instantáneo el AF entra al estado definido por $\delta(q, a)$ moviendo la cabeza lectora un símbolo a la derecha,
 - si $\delta(q, a) \in F$ (i.e., es un estado de aceptación) entonces el AF considera aceptar la cadena de símbolos vista hasta el momento (pero sin incluir necesariamente el símbolo sobre el cual la cabeza lectora se ha movido). Si la cabeza lectora se ha movido al final de la cadena (extremo derecho de la cinta) entonces el AF se detiene y acepta la cadena completa.

- Si en algún estado p no está definida $\delta(p, a)$ entonces el AF se detiene y rechaza la cadena leída hasta el momento.
- Si el AF ha examinado toda la cadena en la cinta pero el estado actual p no está en el conjunto de estados finales F ($p \notin F$) entonces el AF se detiene y rechaza la cadena.

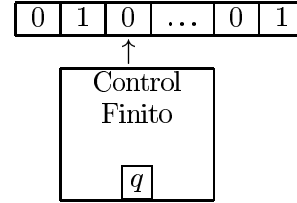


Figura 2.6: El control finito de un AF y la cinta de donde lee la cadena de símbolos de entrada.

Un **cómputo**, por tanto, es la descripción de cambios de estado que sufre un autómata ante una cadena de entrada dada. Un cómputo *siempre termina* puesto que la cadena es finita y siempre es posible saber si está o no el autómata en un estado de aceptación.

Definición 2.1.2 (Grafo de Transición) *Dado un AF, su grafo dirigido de transiciones se construye de la siguiente forma:*

1. Los vértices del digrafo corresponden a los estados del AF.
2. Si existe una transición del estado q al estado p ante la entrada a entonces existe un arco del nodo q al nodo p etiquetado con el símbolo a .
3. En el digrafo, el estado inicial q_o se indica con una flecha (arco) que llega, sin ninguna etiqueta. Los estados finales de aceptación se indican circundándolos.

Ejemplo 2.1.3 Sea $M_1 = \langle Q, \Sigma, \delta, q_o, \{q_o\} \rangle$ un AFD donde $Q = \{q_o, q_1, q_2\}$, $\Sigma = \{0, 1\}$ y δ definida como se muestra en la Figura 2.7.

Sea la cadena de entrada $w = "1100"$, mostramos a continuación la secuencia de transiciones (cómputo) que realiza M_1 :

$$\rightarrow q_o \xrightarrow{1} q_1 \xrightarrow{1} q_o \xrightarrow{0} q_2 \xrightarrow{0} q_o$$

lo cual también podemos describir por la siguiente tabla:

1	1	0	0	
q_o	q_1	q_o	q_2	q_o

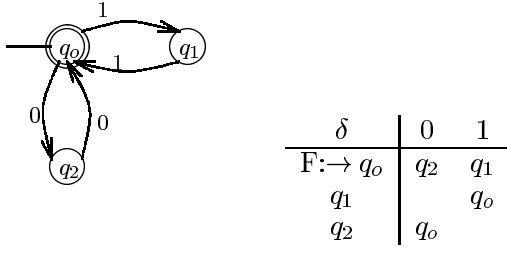


Figura 2.7: AFD M_1

(que se lee: “estando en el estado q_i y leyendo el símbolo a_j en la columna j el AF se mueve al estado indicado por $\delta(q_i, a_j)$ ”)

por lo tanto: M_1 acepta a w (ya que se llegó a un estado de aceptación).

Sea ahora $u = “11100”$, entonces M_1 rechaza a u :

$$\rightarrow q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_0 \xrightarrow{1} q_1 \xrightarrow{0} \text{indefinido}$$

1	1	1	0	0
q_0	q_1	q_0	q_1	<i>indef</i>

En resumen, un AFD acepta una cadena de entrada w si y sólo si la secuencia de transiciones (ruta) que corresponden a cada uno de los símbolos de w (leídos en orden de izquierda a derecha) llevan del estado inicial a algún estado final.

Debemos describir formalmente el funcionamiento de un AF ante una cadena de entrada por lo que debemos extender a la función de transición δ para que acepte cadenas definidas sobre el alfabeto Σ :

Definición 2.1.3 ($\hat{\delta}$) Para un AF M definimos la función extendida $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$ a partir de la función $\delta : Q \times \Sigma \rightarrow Q$, de tal forma que $\hat{\delta}(q, w)$ representa el único estado p tal que existe una ruta en el digrafo de transiciones de M que parte de q y llega a p etiquetada por w . Definimos a $\hat{\delta}$ inductivamente

1. $\hat{\delta}(q, \epsilon) \stackrel{\text{def}}{=} q$, (caso base)
2. $\hat{\delta}(q, wa) \stackrel{\text{def}}{=} \delta(\hat{\delta}(q, w), a)$
para toda cadena $w \in \Sigma^*$, y símbolo $a \in \Sigma$.

El caso base en la definición de $\hat{\delta}$ indica que el AF no cambia de estado a menos que lea un símbolo de entrada, mientras que el paso inductivo indica cómo se debe calcular el siguiente estado ante una secuencia no vacía de símbolos de entrada: primero se encuentra el estado $p = \hat{\delta}(q, w)$ tras leer w y entonces se evalúa $\delta(p, a)$. Nótese que δ y $\hat{\delta}$ conciden en cadenas de longitud 1:

$$\begin{aligned} \hat{\delta}(q, a) &= \hat{\delta}(q, \epsilon a) && \text{dado que “} a \text{”} = \epsilon a \\ &= \delta(\hat{\delta}(q, \epsilon), a) && \text{paso inductivo de la def.} \\ &= \delta(q, a). \end{aligned}$$

Formalmente, una cadena w es **aceptada** por un AFD $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ si y sólo si $\delta(q_0, w) \in F$. El lenguaje aceptado por M lo denotamos por $L(M)$ y es el conjunto de todas las cadenas aceptadas por M

$$L(M) \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

Decimos que un conjunto $A \subseteq \Sigma^*$ es **regular** si $A = L(M)$ para algún AFD M .

Ejemplo 2.1.4 Para el AFD M_1 del ejemplo 2.1.3, mostramos a continuación la secuencia de evaluaciones $\hat{\delta}$ correspondiente a la cadena $u = 11100$:

$$\begin{aligned} \hat{\delta}(q_0, u) &= \delta(\hat{\delta}(q_0, 1110), 0) \\ &= \delta(\delta(\hat{\delta}(q_0, 111), 0), 0) \\ &= \delta(\delta(\delta(\hat{\delta}(q_0, 11), 1), 0), 0) \\ &= \delta(\delta(\delta(\delta(\hat{\delta}(q_0, 1), 1), 0), 0), 0) \\ &= \delta(\delta(\delta(\delta(\delta(\hat{\delta}(q_0, \epsilon), 1), 1), 1), 0), 0), 0) \\ &= \delta(\delta(\delta(\delta(\delta(q_0, 1), 1), 1), 0), 0), 0) \\ &= \delta(\delta(\delta(\delta(q_1, 1), 1), 0), 0), 0) \\ &= \delta(\delta(\delta(q_0, 1), 0), 0), 0) \\ &= \delta(\underbrace{\delta(q_1, 0), 0}, 0) \\ &\quad \text{indefinida} \end{aligned}$$

por tanto u es rechazada por M_1

Antes de mostrar más ejemplos, es de especial importancia ofrecer respuesta a las siguientes preguntas

1. Dado un AFD M , ¿cuál es el lenguaje que reconoce (o acepta) M ?
2. Dada la descripción de un lenguaje **regular** (un conjunto de cadenas descrito ya sea de forma intensional ó extensional) ¿cuál es el AFD que acepta ese lenguaje?

Para responder la primer pregunta, aprovecharemos el AFD M_1 del ejemplo anterior y centraremos nuestra atención en un “subautómata” M'_1 que consiste en $M'_1 = \langle \{q_0, q_1\}, \Sigma, \delta_1, q_0, \{q_0\} \rangle$ donde δ_1 es la misma tabla de δ exceptuando el último renglón

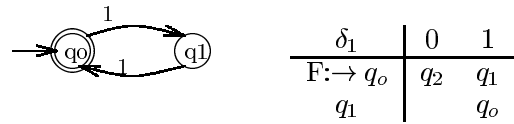


Figura 2.8: AFD M'_1

El mismo análisis se extiende para considerar el estado q_2 por lo que se deja como ejercicio al lector.

Para poder expresar cuál es el lenguaje que acepta M'_1 es útil indicar que **propiedad** cumplen las cadenas que son aceptadas por M'_1 , lo cual significa

analizar la estructura de las cadenas cuando son aceptadas por M'_1 .

En nuestro ejemplo, un vistazo rápido nos permite apreciar que las cadenas son aceptadas si M'_1 alcanza el estado de aceptación q_0 , la cadena más pequeña que acepta M'_1 es ϵ (pues M'_1 parte del estado inicial q_0 , al “leer” ϵ no se mueve sino que se queda en q_0 que también es final); la siguiente cadena (en tamaño) que acepta M_1 es “11” que significa la secuencia de transiciones $q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_0$; etc. Si ordenamos de esa forma las cadenas que son aceptadas por M'_1 tenemos descrito el lenguaje de M'_1 :

$L(M'_1)$	$2n$	n
ϵ	0	0
11	2	1
1111	4	2
\vdots	\vdots	\vdots

En general, si la cadena w está formada por secuencias pares de 1's, entonces $w \in L(M'_1)$ ya que es la única forma en que M'_1 alcanza el estado final q_0 .

El anterior análisis es intuitivo pero nótese que el aspecto importante es analizar estructuralmente las cadenas que alcancen el estado final del AFD (partiendo el AFD obviamente de su estado inicial), o en otras palabras, ¿de que forma se puede llegar al estado de aceptación?. Evidentemente, en general puede haber más de una forma.

Formalmente, demostraremos que M'_1 acepta cadenas formadas por secuencias pares de 1's (equivalentemente, cadenas formadas por parejas de 1's, cadenas de longitud par de 1's). Para ello, aplicaremos Inducción sobre la longitud de las cadenas que acepta M'_1

1. Caso Base:

$$|w| = 0 \text{ (} w = \epsilon \text{)}$$

$$\epsilon \in L(M'_1)$$

$$\hat{\delta}(q_0, \epsilon) = q_0 \in \{q_0\} = F$$

2. Paso Inductivo

H.I.: Suponer para cualquier m , la cadena w cumple $|w| = 2m$ (i.e., es de longitud par) está formada por ‘1’s y pertenece a $L(M'_1)$.

Por demostrar que lo anterior se cumple para cadenas de longitud $\text{sucesor}(m)$ aceptadas por M'_1 :

Habiendo observado w , el autómata está en q_0 , sea v formado como $v = wx$, tal que $|v| = 2(m+1)$ lo cual significa que $x = 11$ pues es la única forma de llegar a un estado de aceptación.

Claramente

$$|v| = |w| + |x| = 2m + 2 = 2(m + 1)$$

Ejercicios 2.1.2

1. Extender éste argumento para M_1 , es decir, demuestre que M_1 estando en q_0 ha visto secuencias de ‘1’s y ‘0’s pares consecutivos.
2. Para cada uno de los siguientes AFD, indique quienes son cada uno de sus componentes (i.e., Q, Σ, F), demuestre cuál es el lenguaje que acepta y muestre si las cadenas w y v dadas son aceptadas o rechazadas por el AFD:

$$(a) \quad w = 01101 \quad \text{y} \quad v = 1110101$$

δ	0	1
$\rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
$F:q_2$	q_0	q_3
q_3	q_1	q_2

$$(b) \quad w = 001100 \quad \text{y} \quad v = 1010101$$

δ	0	1
$F: \rightarrow q_0$	q_2	q_1
q_1	q_0	
q_2	q_0	

Mostraremos ahora un ejemplo en el que se proporciona la descripción de un lenguaje a partir de la cual construiremos un AFD que acepta ése lenguaje:

Ejemplo 2.1.5 Considérese el conjunto

$$L = \{waaav \mid w, v \in \{a, b\}^*\}$$

Una forma alternativa de describir a L es como el conjunto de cadenas tomadas del alfabeto $\{a, b\}$ tales que contienen una subcadena de 3 a 's consecutivas. Por tanto, la cadena $abaaabab$ debe ser aceptada por el AFD correspondiente, pero la cadena $babbabba$ debe ser rechazada.

Para la construcción de un¹³ AFD M cuyo lenguaje corresponde al conjunto L dado, podemos apreciar lo siguiente :

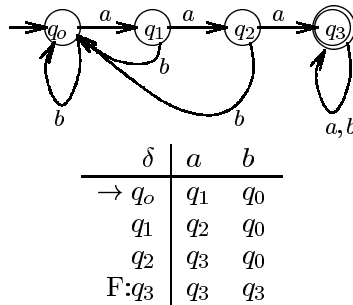
1. La cadena más pequeña de L es aaa .
2. Una cadena de L puede comenzar con cualquier secuencia de b 's, seguida por la subcadena aaa y finalizada por cualquier secuencia de a 's ó b 's
3. Si ocurre alguna b intercalada entre aaa , entonces se espera que más adelante en la cadena esté la subcadena aaa .

¹³Enfatizamos “un” pues en general dada la descripción de un conjunto regular podemos ofrecer más de un AFD que acepta ese conjunto.

Por tanto, podemos construir un AFD M que acepta L de la siguiente manera:

1. el estado inicial registra cualquier prefijo formado por una secuencia de b 's; si se lee una a entonces M pasa al estado q_1 que inicia la verificación de la subcadena aaa ,
2. en q_1 , si lee una a pasa a q_2 , si en q_2 lee una a pasa al estado final q_3 ; de lo contrario, si en alguno de los anteriores estados lee una b es como si volviera a empezar y por tanto se mueve al estado q_0 ;
3. en el estado final q_3 , puede leer cualquier secuencia de a 's ó b 's y continúa en el estado final q_3 pues ya ha visto la subsecuencia aaa y por tanto la cadena pertenece al lenguaje L .

Por tanto M es



El lector puede aplicar inducción en la longitud de las cadenas que acepta M para verificar que $L(M) = L$.

Una forma muy útil para describir lenguajes (y que discutiremos ampliamente a partir del capítulo 6) es la **Notación Backus-Naur (BNF)**, la cual consiste en *reglas* de la forma:

$$\text{Nombre} ::= \text{patrón}$$

donde el *Nombre* permite hacer referencia al objeto que se está definiendo, “ $::=$ ” se lee “tiene la forma” y el *patrón* es la definición del objeto.

Cuando aprendemos un lenguaje de programación (como C, Pascal ó Java) podemos encontrar una descripción como la siguiente :

“Un número natural es una cadena formada por dígitos”.

Tal descripción en forma BNF tiene la siguiente forma:

$$\begin{aligned} \text{dígito} & ::= \{ '0', '1', '2', \dots, '9' \} \\ \text{natural} & ::= \text{dígito}(\text{dígito})^* \end{aligned}$$

El primer patrón indica que un *dígito* es cualquier elemento del conjunto de símbolos que se enuncia en la derecha, mientras que un *natural* es aquella cadena

formada por la concatenación de un *dígito* seguido por una cadena formada por cero ó más *dígitos*. * indica que la expresión encerrada (inmediata izquierda) puede repetirse cualquier número de veces (incluso cero)¹⁴.

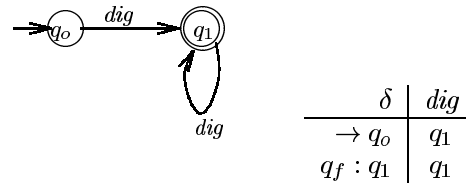
Como ejemplo de cadenas que obedecen al patrón anterior tenemos: “09987”, “440019”, etc.

En el procesador de órdenes (*shell*) de cualquier sistema operativo (como Dos ó Un*x) se cuenta con un comando que permite mostrar los archivos presentes en el directorio actual. Tal comando puede recibir como argumentos un nombre de archivo en particular ó un patrón que describe toda una familia de nombres de archivo que cumple con lo que especifica el patrón, por ejemplo:

`>dir *.c`

mostrará todos los nombres de archivo con extensión “.c”. Como veremos más adelante, los patrones que intuitivamente hemos descrito están inspirados en la definición de *expresiones regulares* que discutiremos más adelante.

Podemos construir un AFD cuyo lenguaje son las cadenas que representan números naturales descritos por el patrón anterior



con $dig = \{ '0', '1', '2', \dots, '9' \}$. Estamos abusando un poco de la notación para indicar que la etiqueta de una transición se indica por un *nombre* con el fin de abreviar y siempre que no cause confusión.

2.2 Composición de AFD's y algunas Propiedades de Cerradura

Un aspecto muy importante en todas las abstracciones que discutiremos es la *reutilización modular*. En particular, para la construcción de un nuevo AFD podemos utilizar (como si se tratara de módulos) otros AFDs. Para ello, debemos asegurar que la **composición** entre esos módulos nos produzca como resultado un AFD (en otras palabras, que a partir de lenguajes regulares podamos obtener otro lenguaje

¹⁴No es coincidencia que se utilice el mismo operador de la cerradura Kleene.

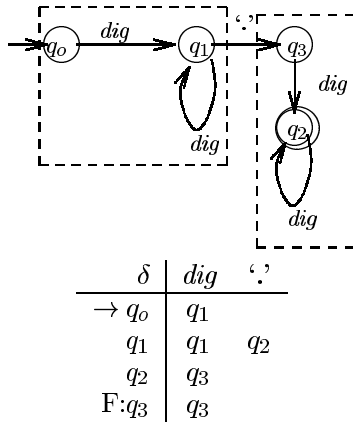
regular). Es ilustrativo considerar el siguiente ejemplo:

Ejemplo 2.2.1 Queremos construir un AFD que acepte el lenguaje de las cadenas que representan números reales de acuerdo a la siguiente descripción:

Un número real tiene una parte entera, a continuación un '.' decimal y una parte decimal.

Como ejemplos de cadenas que siguen la anterior descripción tenemos "097.5321" y "123.4567".

Evidentemente, podemos aprovechar el ejemplo de los números naturales para construir el nuevo AFD M' : tanto la parte entera como la parte decimal se pueden reconocer mediante expresiones formadas por secuencias de dígitos



Pero si queremos que ahora el AFD M' acepte expresiones en las cuales no necesariamente esté presente la parte entera, como ".00", tendríamos

δ	dig	'.'
$\rightarrow q_0$	q_1	q_3
$F: q_1$	q_1	q_2
q_2	q_3	
$F: q_3$	q_3	

En éstos ejemplos podemos reconocer inmediatamente la presencia de la operación de *concatenación* entre AFD: el lenguaje de M se obtiene concatenando los lenguajes que aceptan respectivamente cada uno de los AFDs vistos como submódulos. Mostraremos a continuación que si dos lenguajes A y B son regulares, entonces $A \cup B$, $A \cap B$, $\sim A$ son regulares. Más adelante estableceremos lo propio para A^* y AB .

Construcción del Producto

Sean A, B regulares y sean M_1, M_2 AFDs tales que $L(M_1) = A$ y $L(M_2) = B$, donde

$$M_1 = \langle Q_1, \Sigma, \delta_1, q_{o1}, F_1 \rangle,$$

$$M_2 = \langle Q_2, \Sigma, \delta_2, q_{o2}, F_2 \rangle$$

• Para demostrar que $A \cap B$ es regular construiremos un AFD M_3 tal que $L(M_3) = A \cap B$.

Ante una cadena de entrada w , M_3 simulará el comportamiento de M_1 y M_2 simultáneamente: M_3 acepta a w si y sólo si ambos aceptan w :

$$M_3 = \langle Q_3, \Sigma, \delta_3, q_{o3}, F_3 \rangle$$

donde

$$Q_3 = Q_1 \times Q_2 = \{(p, q) \mid p \in Q_1 \text{ y } q \in Q_2\}$$

$$F_3 = F_1 \times F_2 = \{(p, q) \mid p \in F_1 \text{ y } q \in F_2\}$$

$$q_{o3} = (q_{o1}, q_{o2})$$

La función de transición δ_3 se define como

$$\delta_3((p, q), a) = (\delta_1(p, a), \delta_2(q, a))$$

y su extensión inductiva para cadenas es

$$\hat{\delta}_3((p, q), \epsilon) = (p, q)$$

$$\hat{\delta}_3((p, q), xa) = \delta_3(\hat{\delta}_3((p, q), x), a).$$

Se deja como ejercicio al lector demostrar el siguiente lema:

Lema 2.2.1 Para toda $w \in \Sigma^*$,

$$\hat{\delta}_3((p, q), w) = (\hat{\delta}_1(p, w), \hat{\delta}_2(q, w))$$

Demostramos a continuación que:

$$L(M_3) = L(M_1) \cap L(M_2)$$

Para toda $x \in \Sigma^*$:

$$x \in L(M_3)$$

$$\iff \hat{\delta}_3(q_{o3}, x) \in F$$

$$\iff \hat{\delta}_3((q_{o1}, q_{o2}), x) \in F_1 \times F_2$$

$$\iff (\hat{\delta}_1(q_{o1}, x), \hat{\delta}_2(q_{o2}, x)) \in F_1 \times F_2$$

$$\iff \hat{\delta}_1(q_{o1}, x) \in F_1 \text{ y } \hat{\delta}_2(q_{o2}, x) \in F_2$$

$$\iff x \in L(M_1) \text{ y } x \in L(M_2)$$

$$\iff x \in L(M_1) \cap L(M_2)$$

Construcción del Complemento

• Para demostrar que $\sim A$ es regular si A lo es, sea M un AFD que acepta A y tomemos a M' simplemente intercambiando estados de aceptación por estados de rechazo

$$\hat{\delta}'(q_0, x) \in F' \iff \hat{\delta}(q_0, x) \notin F$$

M' aceptará exactamente lo que M rechaza.

Construcción de la Unión

Podemos demostrar inmediatamente aplicando Leyes de DeMorgan que $A \cup B$ es regular si A y B lo son:

$$A \cup B = \sim(\sim A \cap \sim B)$$

lo cual ofrece un AFD que se parece al AFD del producto pero los estados de aceptación son

$$\begin{aligned} F_3 &= \{(p, q) \mid p \in F_1 \text{ ó } q \in F_2\} \\ &= (F_1 \times Q_2) \cup (Q_1 \times F_2) \end{aligned}$$

Intuitivamente, la unión de dos lenguajes nos permite describir el AFD M_3 correspondiente mediante una **alternativa**: en un estado dado, el AFD elige seguir una transición y finalmente una ruta que lleva a un estado de aceptación de M_1 ó de M_2 . ¿Que pasa si tal elección deba tomarse leyendo en ambos casos el mismo símbolo de entrada actual? Ello significaría que ante el mismo símbolo existe una transición que lleva a estados distintos, lo cual no está definido en un AFD. Esta idea intuitiva nos conduce a la siguiente abstracción: el *no determinismo*.

Ejercicios 2.2.1

- Suponga que δ es la función de transición de un AFD. Demuestre que para cualesquier cadenas de entrada x, y ,

$$\delta(q, xy) = \delta(\delta(q, x), y)$$

(Tip: aplique inducción en $|y|$).

- Obtenga el AFD que acepte los siguientes lenguajes sobre el alfabeto $\Sigma = \{0, 1\}$:
 - El conjunto de todas las cadenas que terminen en '00'.
 - El conjunto de todas las cadenas con tres '0's consecutivos.
 - El conjunto de todas las cadenas tales que todo bloque de cinco símbolos consecutivos contenga al menos dos '0's.
 - El conjunto de todas las cadenas tales que el décimo símbolo (contado de izquierda a derecha) sea un '1'.
 - El conjunto de todas las cadenas con número impar de '0's y número par de '1's.
 - El conjunto de todas las cadenas con número impar de '0's ó número par de '1's.
 - El conjunto de todas las cadenas con número impar de '0's **XOR** número par de '1's.

- Construya la función de transición para el AFD que reconozca todas las cadenas tomadas del alfabeto $\{a, b, c\}$ que comiencen y terminen con una letra distinta.
- Describa en palabras cuál es el lenguaje aceptado por cada uno de los AFD's mostrados en la fig. 2.9:

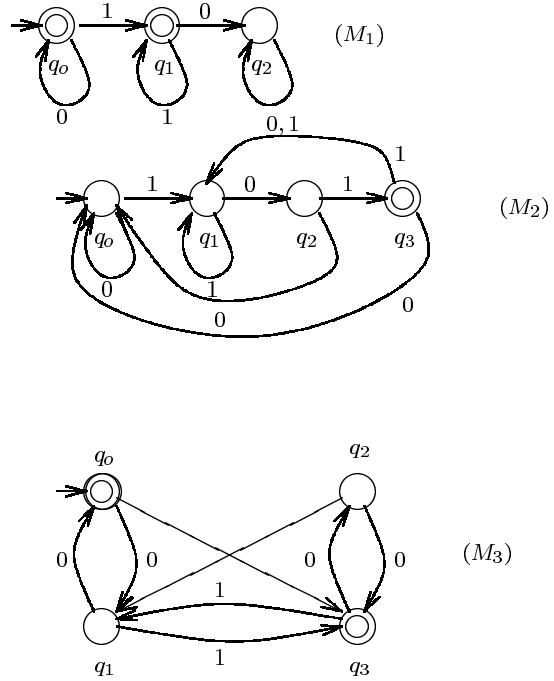


Figura 2.9: Ejercicio 4

- Demuestre que el AFD de la fig. 2.10 acepta todas las cadenas sobre el alfabeto $\{0, 1\}$ con número igual de 0's y 1's, tales que cada prefijo tiene a lo más un 0 más que 1s y a lo más un 1 más que 0s.

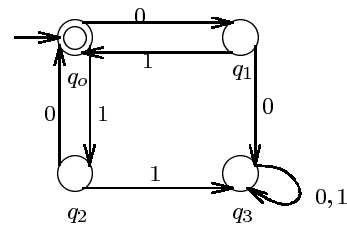


Figura 2.10: Ejercicio 5

- Considere el conjunto de los nombres de todos los alumnos inscritos en éste curso. ¿Es posible construir un AFD que acepte ese lenguaje? Justifique su respuesta.

Autómatas Finitos No Determinísticos

El **nodeterminismo** es una abstracción muy importante en las ciencias de la computación pues se refiere a situaciones en las cuales el siguiente estado de un cómputo no está únicamente determinado a partir del estado actual. El nodeterminismo tiene lugar en problemas de la vida real cuando existe información incompleta acerca del estado ó cuando existen factores externos que pueden afectar el curso del cómputo. Por ejemplo, el comportamiento de un proceso en un sistema distribuido podría depender de los mensajes que otros procesos, tales mensajes llegan en tiempos impredecibles con contenidos impredecibles.

El nodeterminismo también es importante en el diseño de algoritmos eficientes: existen muchas instancias de problemas combinatorios importantes con soluciones nodeterminísticas eficientes, pero no se conocen soluciones determinísticas eficientes. El famoso problema $P = NP$ -si todos los problemas solubles en tiempo polinomial en el *modelo de cómputo universal* nodeterminístico pueden resolverse en tiempo polinomial en el modelo de cómputo universal determinístico- es un problema abierto fundamental en las ciencias de la computación y uno de los problemas abiertos más importantes en toda la matemática.

Para problemas de decisión, el aspecto dominante en lo que nodeterminismo concierne es el paradigma *suponer y verificar*, lo cual significa que ante una entrada dada se supone un cómputo exitoso o una demostración de que la entrada es una instancia “si” del problema de decisión, y entonces verificar que la suposición es efectivamente correcta. En general, en situaciones no determinísticas no podemos saber exactamente cómo evolucionará un cómputo, pero podemos tener idea del rango de posibilidades: esto se modela en la teoría de autómatas permitiendo que el autómata tenga funciones de transición con *múltiples valores de argumentos posibles*, lo cual esencialmente significa que el modelo determinístico se extiende para permitir cero, uno ó más transiciones de un estado leyendo el mismo símbolo de entrada. No obstante, mostraremos que para cualquier lenguaje aceptado por un AF no determinístico (AFND) pode-

mos construir un AFD que acepta el mismo lenguaje, lo cual significa que ambos modelos son equivalentes, aún cuando las máquinas no determinísticas sean exponencialmente más compactas.

Mientras que un AFD solamente permite que para cada estado exista exactamente una transición para cada símbolo de entrada tomado de Σ , en un AFND podemos tener cero, uno ó más transiciones de un estado leyendo el mismo símbolo de entrada que conduzca a estados diferentes (o posiblemente el mismo). Por tanto, podemos considerar que los AFD son casos especiales de AFND.

Considérese el diagrama de transiciones de la Figura 3.11: hay dos aristas etiquetadas con 0 que salen del estado q_0 , una que va al estado q_3 y otra que se queda en q_0 . Informalmente, una cadena de

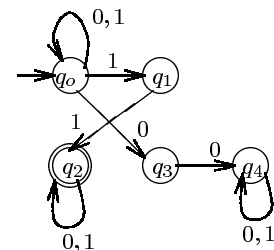


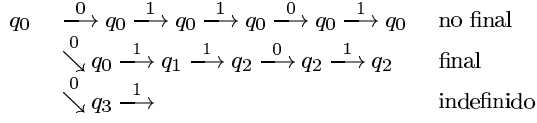
Figura 3.11: Un AFND.

entrada w (formada por n símbolos $w = a_1a_2 \dots a_n$, $n \geq 1$) es aceptada por un AFND si existe al menos una secuencia de transiciones, etiquetadas con cada símbolo a_i de la cadena w , que conducen del estado inicial a alguno de los estados de aceptación. Por ejemplo, la cadena $w = "01101"$ es aceptada por el AFND de la Figura 3.11 pues podemos construir una ruta de q_0 a q_2 :

$$q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_2 \xrightarrow{0} q_2 \xrightarrow{0} q_2$$

En general para un AFND podrían construirse muchas rutas etiquetadas con una cadena dada y por tanto todas deban examinarse para verificar si alguna termina en un estado final. Ello significa que el control finito al leer la cinta (donde está escrita la cadena

de entrada) puede estar en cualquier número de estados simultáneamente, ya que al elegir una posible ruta en un estado es como si se tuvieran múltiples copias del autómata, cada una para cada posible opción, generando así una **proliferación de estados** formando un árbol:



Formalmente,

Definición 3.1.1 Un AFND M es una estructura $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ donde Q, Σ, q_0 y F tienen el mismo significado que para los AFD, pero la función de transición está definida por

$$\delta : Q \times \Sigma \longrightarrow 2^Q$$

(donde 2^Q es el conjunto potencia de Q)

δ regresa **conjuntos de estados** en vez de un sólo estado siguiente: la intención es que $\delta(q, a)$ denote el conjunto de todos los estados p tales que existe una transición de q hacia p etiquetada con a .

Extendemos inductivamente a δ para reconocer cadenas $\hat{\delta} : Q \times \Sigma^* \longrightarrow 2^Q$

1. Caso base:

$$\hat{\delta}(q, \epsilon) = \{q\}$$

no hay cambio de estado si no se lee algún símbolo.

2. Paso Inductivo:

$$\hat{\delta}(q, wa) = \{p \in Q \mid \text{existe } r \in Q, r \in \hat{\delta}(q, w), p \in \delta(r, a)\}$$

Esta condición indica que partiendo del estado q y leyendo la cadena de entrada w seguida del símbolo a el AFND puede estar en el estado p si y sólo si se puede alcanzar algún estado r del cual mediante una transición etiquetada con a alcanza el estado p . Operacionalmente, primero debemos obtener el conjunto de estados que son alcanzables leyendo w (recursivamente) y entonces para ese conjunto de estados, evaluar $\delta(r, a)$.

$$q \xrightarrow{\overbrace{\quad \quad \quad}^w} \dots \xrightarrow{\quad} r \xrightarrow{a} p$$

El lector puede demostrar fácilmente que $\hat{\delta}(q, "a") = \delta(q, a)$.

3. Extendemos la función $\hat{\delta}$ para recibir como argumentos conjuntos de estados: $\hat{\delta} : 2^Q \times \Sigma^*$

$$\hat{\delta}(P, w) = \bigcup_{q \in P} \hat{\delta}(q, w)$$

donde $P \subseteq Q$.

Nótese que en general también podemos plantear lo anterior de la siguiente forma:

$$\begin{aligned}
 \hat{\delta}(P, \epsilon) &= P, \\
 \hat{\delta}(P, xa) &= \bigcup_{q \in R} \delta(q, a)
 \end{aligned}$$

donde $R = \hat{\delta}(P, x)$.

$L(M)$ denota al lenguaje aceptado por un AFND M :

$$L(M) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Por la definición anterior, cualquier AFD $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ es equivalente a un AFND $N = \langle Q, \Sigma, \delta', \{q_0\}, F \rangle$ donde $\delta'(p, q) \stackrel{def}{=} \{\delta(p, a)\}$. En la siguiente sección demostraremos que todo AFND es equivalente a un AFD.

Ejemplo 3.1.2 Mostraremos el funcionamiento del AFND de la Figura 3.11 $M_3 = \langle Q, \Sigma, \delta, q_0, F \rangle$, donde $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $F = \{q_2\}$, $\Sigma = \{0', 1'\}$ y su función de transición está dada por

δ	0	1
$\rightarrow q_0$	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	\emptyset	$\{q_2\}$
$F:q_2$	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	\emptyset
q_4	$\{q_4\}$	$\{q_4\}$

Sea la cadena de entrada $w = "01100"$. La secuencia de transiciones que realiza M_3 para examinar w se muestra a continuación

$$\hat{\delta}(q_0, w) = \{r \in \hat{\delta}(q_0, "0110"), \delta(r, 0)\} \quad (1)$$

$$\hat{\delta}(q_0, "0110") = \{r \in \hat{\delta}(q_0, "011"), \delta(r, 0)\} \quad (2)$$

$$\hat{\delta}(q_0, "011") = \{r \in \hat{\delta}(q_0, "01"), \delta(r, 1)\} \quad (3)$$

$$\hat{\delta}(q_0, "01") = \{r \in \hat{\delta}(q_0, "0"), \delta(r, 1)\} \quad (4)$$

$$\hat{\delta}(q_0, "0") = \{r \in \hat{\delta}(q_0, \epsilon), \delta(r, 0)\} \quad (5)$$

Nótese que se va evaluando recursivamente sobre la longitud de la cadena de entrada hasta que ésta sea mínima solo entonces podemos detener la recursión y consultar la tabla de δ :

$$\begin{aligned}
(5) \quad \delta(\{q_0\}, 0) &= \{q_0, q_3\} = \hat{\delta}(q_0, "0'') \\
(4) \quad \delta(\{q_0, q_3\}, 1) &= \{q_0, q_1\} \cup \{\} \\
&= \{q_0, q_1\} = \hat{\delta}(q_0, "01'') \\
(3) \quad \delta(\{q_0, q_1\}, 1) &= \{q_0, q_1\} \cup \{q_2\} \\
&= \{q_0, q_1, q_2\} = \hat{\delta}(q_0, "011'') \\
(2) \quad \delta(\{q_0, q_1, q_2\}, 0) &= \{q_0, q_3\} \cup \emptyset \cup \{q_2\} \\
&= \{q_0, q_2, q_3\} = \hat{\delta}(q_0, "01100'') \\
(1) \quad \delta(\{q_0, q_2, q_3\}, 0) &= \{q_0, q_3\} \cup \{q_2\} \cup \{q_4\} \\
&= \{q_0, q_2, q_3, q_4\} = \hat{\delta}(q_0, w)
\end{aligned}$$

Por lo tanto si $\hat{\delta}(q_0, w) \cap F \neq \emptyset$ entonces M_3 acepta la cadena w .

Sea ahora la cadena $v = "0101"$. Mostraremos que $v \notin L(M_3)$:

$$\begin{aligned}
(1) \quad \hat{\delta}(q_0, "0101'') &= \{r \in \hat{\delta}(q_0, "010''), p \in \delta(r, 1)\} \\
(2) \quad \hat{\delta}(q_0, "010'') &= \{r \in \hat{\delta}(q_0, "01''), p \in \delta(r, 0)\} \\
(3) \quad \hat{\delta}(q_0, "01'') &= \{r \in \hat{\delta}(q_0, "0''), p \in \delta(r, 1)\} \\
(4) \quad \hat{\delta}(q_0, "0'') &= \{r \in \hat{\delta}(q_0, \epsilon), p \in \delta(r, 0)\}
\end{aligned}$$

termina la recursión y se consulta la tabla de δ :

$$\begin{aligned}
(4) \quad \delta(\{q_0\}, 0) &= \{q_0, q_3\} \\
(3) \quad \delta(\{q_0, q_3\}, 1) &= \delta(q_0, 1) \cup \delta(q_3, 1) \\
&= \{q_0, q_1\} \cup \{\} = \{q_0, q_1\} \\
(2) \quad \delta(\{q_0, q_1\}, 0) &= \delta(q_0, 0) \cup \delta(q_1, 0) \\
&= \{q_0, q_3\} \cup \{\} = \{q_0, q_3\} \\
(1) \quad \delta(\{q_0, q_3\}, 1) &= \delta(q_0, 1) \cup \delta(q_3, 1) \\
&= \{q_0, q_1\} \cup \{\} = \{q_0, q_1\}
\end{aligned}$$

por tanto

$$\hat{\delta}(q_0, "1100'') = \{q_0, q_1\}$$

entonces

$$\{q_0, q_1\} \cap \{q_2\} = \emptyset$$

por lo que M_3 no acepta la cadena v .

3.2 Equivalencia entre AFD y AFND

El siguiente resultado nos indica cómo podemos simular un AFND mediante un AFD: mientras que el AFND al examinar una cadena de entrada construye un árbol de estados, el AFD correspondiente *encapsula* los estados en paquetes donde cada paquete corresponde al conjunto de estados de cada nivel del árbol. La demostración es constructiva en el sentido que se ofrece implícitamente un algoritmo para obtener el AFD equivalente.

Teorema 3.2.1 *Sea L un conjunto (lenguaje) aceptado por un AFND, existe un AFD que acepta L .*

Demostración: Sea $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ un AFND que acepta L . Se define un AFD $M' = \langle Q', \Sigma, \delta', q'_0, F' \rangle$ de la siguiente forma

- $Q' = 2^Q$ es el conjunto potencia¹⁵ de Q .
- F' contiene subconjuntos que a su vez contengan **al menos un estado** en F .

Antes de continuar con la demostración, conviene ilustrar el proceso. Sea

$$M_4 = \langle \{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\} \rangle$$

cuya función δ está dada por:

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

Ordenaremos por tamaño a los elementos de 2^Q :

cardinalidad	conjunto
0	\emptyset
1	$\{q_0\}, \{q_1\}, \{q_2\}$
2	$\{q_0, q_1\}, \{q_1, q_2\}, \{q_0, q_2\}$
3	$\{q_0, q_1, q_2\}$

En la construcción para transformar un AFND a AFD, un elemento de Q' será denotado por un *paquete* $[q_1, \dots, q_n]$, donde cada $q_i \in Q$.

Para q'_0 corresponde el paquete $[q_0]$.

- Se define δ' de la siguiente manera:

$$\begin{aligned}
\delta'([q_1, q_2, \dots, q_i], a) &= [p_1, p_2, \dots, p_i] \\
\text{siempre que } \delta(\{q_1, q_2, \dots, q_i\}, a) &= \{p_1, p_2, \dots, p_i\} \\
\delta'(q'_0, x) &= [q_1, q_2, q_3, \dots, q_j] \\
\text{siempre que } \delta(q_0, x) &= \{q_1, q_2, \dots, q_j\}.
\end{aligned}$$

Para el AFND M_4 que estamos considerando podemos obtener la función δ' a partir de la construcción de paquetes **nuevos** como se muestra a continuación:

$$\begin{aligned}
(1) \quad \delta(\{q_0, q_1\}, 0) &= \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\} \\
(2) \quad \delta(\{q_0, q_1\}, 1) &= \{q_0, q_1, q_2\} \cup \emptyset = \{q_0, q_1, q_2\} \\
(3) \quad \delta(\{q_0, q_1, q_2\}, 0) &= \{q_0, q_1\} \cup \emptyset \cup \{q_1\} \\
&= \{q_0, q_1\} \\
(4) \quad \delta(\{q_0, q_1, q_2\}, 1) &= \{q_0, q_1, q_2\} \cup \emptyset \cup \{q_2\} \\
&= \{q_0, q_1, q_2\}
\end{aligned}$$

No nos interesa obtener exhaustivamente todos los posibles paquetes a partir de todos los posibles conjuntos de estados, solamente aquellos que se obtienen partiendo del paquete inicial $[q_0]$. La tabla para δ' queda como sigue (los números entre paréntesis en la tabla indican a partir de dónde fueron obtenidos los paquetes):

¹⁵En realidad, como veremos en un momento, basta que Q' sea un subconjunto de 2^Q , i.e., $Q' \subseteq 2^Q$, descartando estados no alcanzables desde el estado inicial.

δ'	0	1
$\rightarrow [q_0]$	$[q_0, q_1]$	$[q_0, q_1, q_2]$
$F: [q_0, q_1]$	$[q_0, q_1]$ (1)	$[q_0, q_1, q_2]$ (2)
$F: [q_0, q_1, q_2]$	$[q_0, q_1]$ (3)	$[q_0, q_1, q_2]$ (4)

Nótese que el primer renglón de δ' se obtuvo al convertir en paquetes el primer renglón de δ .

El conjunto de estados finales F' consiste de todos los paquetes que contengan al menos un estado final de F : en el ejemplo $F' = \{[q_0, q_1], [q_0, q_1, q_2]\}$.

La traza de estados ante la cadena "001" para el AFD M'_4 es

$[q_0] \xrightarrow{0} [q_0, q_1] \xrightarrow{0} [q_0, q_1] \xrightarrow{1} [q_0, q_1, q_2]$ la acepta

mientras que M_4 hace lo siguiente:

	0	0	1	
q_0	$\rightarrow q_0$	$\rightarrow q_0$	$\rightarrow q_0$	no acepta
	$\searrow q_1$			no acepta
		$\searrow q_1$		no acepta
			$\searrow q_2$	no acepta
			$\searrow q_1$	acepta

Podemos apreciar inmediatamente que la traza de estados del AFD M'_4 en realidad corresponde a *visitar por nivel* al árbol que muestra las transiciones en el AFND M'_4 : nótese que los conjuntos de estados por nivel en el árbol corresponden a un paquete en la traza que realiza el AFD.

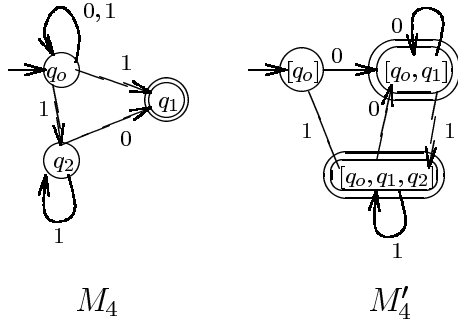


Figura 3.12: Equivalencia entre AFND y AFD.

Ahora podemos continuar con la demostración del teorema: Aplicamos inducción sobre la longitud de la cadena x ($x \in L(M)$ si y sólo si $x \in L(M')$), nuestra H.I. es

$$\delta'(q'_0, x) = [q_1, q_2, \dots, q_r]$$

$$\text{si y sólo si } \delta(q_0, x) = \{q_1, q_2, \dots, q_r\}$$

- Caso base $|x| = 0$: $x = \epsilon$ claramente $\delta'(q_0, \epsilon) = [q_0]$ pues $\delta(q_0, \epsilon) = \{q_0\}$
- Paso Inductivo: suponemos que la H.I. se cumple para w donde $|w| \leq m$. Sea wa una cadena

tal que $|wa| = m + 1$ ($a \in \Sigma$). Entonces se debe cumplir que

$$\delta'(q'_0, wa) = \delta'(\delta'(q'_0, w), a)$$

Por la HI sabemos que

$$\delta'(q'_0, w) = [p_1, p_2, \dots, p_j]$$

$$\text{si y sólo si } \delta(q_0, w) = \{p_1, p_2, \dots, p_j\}$$

Y por la definición de δ' :

$$\delta'([p_1, p_2, \dots, p_j], a) = [r_1, r_2, \dots, r_k]$$

$$\text{si y sólo si } \delta(\{p_1, p_2, \dots, p_j\}, a) = \{r_1, r_2, \dots, r_k\}$$

Por lo tanto

$$\delta'(q'_0, wa) = \delta'(\delta'(q'_0, w), a)$$

$$= \delta'([p_1, p_2, \dots, p_j], a) = [r_1, r_2, \dots, r_k]$$

$$\text{si y sólo si}$$

$$\delta(q_0, wa) = \{r_1, r_2, \dots, r_k\}$$

Finalmente, $(\delta'(q'_0, x) \cap F') \neq \emptyset \iff \delta(q_0, x) \in F$. Por tanto, $L(M) = L(M')$.

Ejemplo 3.2.1 Sea el AFND M_5 dado por la siguiente función de transición:

δ	0	1
$F: \rightarrow q_0$	$\{q_0\}$	$\{q_2\}$
$F: q_1$	$\{q_1, q_2\}$	$\{q_0\}$
$F: q_2$	$\{q_0\}$	$\{q_2\}$

Los pasos en la construcción del AFD correspondiente son:

1. $q'_0 = [q_0]$
2. $\delta(\{q_0\}, '0') = \{q_0\} : [q_0]$
 $\delta(\{q_0\}, '1') = \{q_1\} : [q_1]$
3. $\delta(\{q_1\}, '0') = \{q_1, q_2\} : [q_1, q_2]$
 $\delta(\{q_1\}, '1') = \{q_0\} : [q_0]$
4. $\delta(\{q_1, q_2\}, '0') = \delta(\{q_1\}, '0') \cup \delta(\{q_2\}, '0')$
 $= \{q_1, q_2\} \cup \{q_0\} = \{q_0, q_1, q_2\} : [q_0, q_1, q_2]$
 $\delta(\{q_1, q_2\}, '1') = \delta(\{q_1\}, '1') \cup \delta(\{q_2\}, '1')$
 $= \{q_0\} \cup \{q_2\} = \{q_0, q_2\} : [q_0, q_2]$
5. $\delta(\{q_0, q_1, q_2\}, '0') = \delta(\{q_0\}, '0') \cup \delta(\{q_1\}, '0') \cup \delta(\{q_2\}, '0')$
 $= \{q_0\} \cup \{q_1, q_2\} \cup \{q_0\} = \{q_0, q_1, q_2\} : [q_0, q_1, q_2]$
6. $\delta(\{q_0, q_1, q_2\}, '1') = \delta(\{q_0\}, '1') \cup \delta(\{q_1\}, '1') \cup \delta(\{q_2\}, '1')$
 $= \{q_1\} \cup \{q_0\} \cup \{q_2\} = \{q_0, q_1, q_2\} : [q_0, q_1, q_2]$
7. $\delta(\{q_0, q_2\}, '0') = \delta(\{q_0\}, '0') \cup \delta(\{q_2\}, '0')$
 $= \{q_0\} \cup \{q_0\} = \{q_0\} : [q_0]$. $\delta(\{q_0, q_2\}, '1') = \delta(\{q_0\}, '1') \cup \delta(\{q_2\}, '1')$
 $= \{q_1\} \cup \{q_2\} = \{q_1, q_2\} : [q_1, q_2]$

Por tanto, el AFD M' queda así:

δ'	0	1
F: $\rightarrow [q_0]$	$[q_0]$	$[q_2]$
F: $[q_1]$	$[q_1, q_2]$	$[q_0]$
F: $[q_1, q_2]$	$[q_0, q_1, q_2]$	$[q_0, q_2]$
F: $[q_0, q_2]$	$[q_0]$	$[q_1, q_2]$
F: $[q_0, q_1, q_2]$	$[q_0, q_1, q_2]$	$[q_0, q_1, q_2]$

En general, se siguen los pasos:

1. Obtener todos los paquetes que se puede llegar desde $[q_0]$, esto evita que se calculen estados que no son accesibles y por tanto, inútiles.
2. Para cada paquete nuevo, obtener todos los paquetes que se puede llegar leyendo cada símbolo del alfabeto.
3. Repetir el paso anterior hasta que ya no se encuentre algún paquete nuevo.

Ejercicios 3.2.1

1. Construya un AFND para los siguientes lenguajes. Para al menos 2 cadenas muestre que sean aceptadas ó rechazadas por el autómata para verificar que esté bien construido:
 - (a) El conjunto de todas las cadenas tomadas de 0,1 tales que algunas parejas de '0s' estén separadas por una cadena de longitud $4i$, para alguna $i \geq 0$.
 - (b) El conjunto de todas las cadenas de '0's y '1's en las que el tercer símbolo leído desde la derecha no sea 1 ó que el último símbolo leído desde la izquierda no sea un '0'.
2. Para cada una de los siguientes AFND construya dos cadenas (de longitud al menos 5) y muestre si aceptan ó rechazan cada cadena utilizando tanto la evaluación formal con δ y la evaluación mostrando el arbolito correspondiente. Dibuje los grafos de cada autómata mostrando claramente los estados finales y etiquetando cada transición con el símbolo correspondiente:

- (a) $M_1 = (\{p, q, r, s\}, \{0, 1\}, \delta_1, p, \{s\})$

δ_1	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	\emptyset
s	$\{s\}$	$\{s\}$

- (b) $M_2 = (\{p, q, r, s\}, \{0, 1\}, \delta_2, p, \{q, s\})$

δ_2	0	1
$\rightarrow p$	$\{q, s\}$	$\{q\}$
q	$\{r\}$	$\{q, r\}$
r	$\{s\}$	$\{p\}$
s	\emptyset	$\{p\}$

3. Obtenga el AFD para cada uno de los anteriores AFND. Verifique que el AFD acepte las mismas cadenas que el AFND.

3.3 AFND con movimientos ϵ

Una extensión muy útil (pero que en realidad no otorga más poder expresivo) al modelo de AFND consiste en incorporar transiciones que se activan ante la entrada nula ϵ . Una *transición ϵ* entre los estados p y q se indica de la forma siguiente

$$p \xrightarrow{\epsilon} q$$

el AF puede elegir tal transición *en cualquier instante sin leer algún símbolo del alfabeto*.

Ejemplo 3.3.1 El diagrama de transiciones de la Figura 3.13 es un AFND- ϵ .

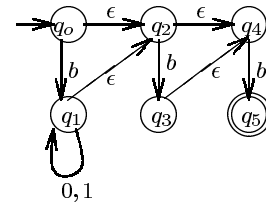


Figura 3.13: M_5

Si la máquina M_5 está en el estado q_0 y lee una b , puede decidir no determinísticamente alguna de las siguientes alternativas:

1. leer b y moverse a q_1 ;
2. moverse a q_2 sin leer algún símbolo y entonces leer b y moverse a q_3 ;
3. moverse a q_2 y luego a q_4 sin leer algún símbolo y entonces leer b y moverse a q_5 .

El conjunto de cadenas aceptado por éste AFND- ϵ es $\{b\}\{0,1\}^* \cup \{b, bb\}$. De la misma forma a como estamos acostumbrados, un AFND- ϵ acepta una cadena de entrada w si y sólo si existe una ruta etiquetada con w desde el estado inicial a algún estado final, tal ruta puede incluir aristas etiquetadas con ϵ (obviamente están implícitas en w).

Recuérdese que ϵ es neutro en la concatenación y además idempotente: $\epsilon w = \epsilon w \epsilon = \epsilon \epsilon w \epsilon = w$.

Por tanto, no importa cuantas veces el AFND- ϵ se mueva mediante varias transiciones ϵ consecutivamente, pues a fin de cuentas es como si fuera una sola transición ϵ .

Formalmente:

Definición 3.3.1 (AFND- ϵ) Un AFND- ϵ es una estructura $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ donde Q, F, Σ, q_0 tienen el mismo significado que en el caso AFND. Pero la función de transición δ se define por

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \longrightarrow 2^Q$$

$\delta(q, a)$ consiste de todos los estados p tales que existe una transición desde q hacia p etiquetada con a , pero ' a ' puede ser un símbolo de Σ ó puede ser ϵ .

Para el AFND- ϵ del ejemplo anterior, su función de transición es la siguiente:

δ	0	1	b	ϵ
$\rightarrow q_0$	\emptyset	\emptyset	$\{q_1\}$	$\{q_2\}$
q_1	$\{q_1\}$	$\{q_1\}$	\emptyset	$\{q_2\}$
q_2	\emptyset	\emptyset	$\{q_3\}$	$\{q_4\}$
q_3	\emptyset	\emptyset	$\{q_3\}$	$\{q_4\}$
q_4	\emptyset	\emptyset	$\{q_5\}$	\emptyset
$F: q_5$	\emptyset	\emptyset	\emptyset	\emptyset

El cambio de estado en un AFND- ϵ puede ocurrir en dos formas:

1. leyendo un símbolo de Σ ;
2. automáticamente, es decir, todos los estados alcanzables "leyendo" ϵ ya sea *implícita* ó *explícitamente*.

Para el segundo caso, recordemos que para los modelos de AF anteriores $\delta(q, \epsilon) = q$, es decir, el AF no se movía si no leía un símbolo. Ahora se consideran dos tipos de transiciones ϵ : las implícitas (de un estado consigo mismo) y las explícitas que acabamos de indicar. Por tanto, debemos extender la función de transición δ a $\hat{\delta}$ la cual toma parejas formadas por un estado y un cadena y regresa un conjunto de estados: $\hat{\delta}(q, w)$ indica todos los estados p alcanzables desde q a lo largo de una ruta etiquetada con w , incluyendo claro transiciones ϵ . De aquí resulta fundamental calcular el conjunto de los estados alcanzables desde un estado dado q *utilizando transiciones ϵ exclusivamente*. A éste conjunto le denominaremos *Cerradura- ϵ (q)*.

Definición 3.3.2 (Cerradura- ϵ) La Cerradura- ϵ de un estado q en un AFND- ϵ es el conjunto de todos los estados alcanzables desde q mediante una (o

varias) transiciones ϵ consecutivas y se define inductivamente:

$$Cerradura-\epsilon(q) = \bigcup_{i=0}^n X_i$$

donde

$$\begin{aligned} X_0 &= \{q\} \\ X_i &= X_{i-1} \cup \{p \mid q \xrightarrow{\epsilon} p, \text{ donde } q \in X_{i-1}, p \in Q\} \end{aligned}$$

para algún $i > 0$.

Mostramos las cerraduras para los estados del ejemplo de la Figura 3.13:

$$\begin{aligned} Cerradura-\epsilon(q_0) &= \{q_0, q_2, q_4\} \\ Cerradura-\epsilon(q_1) &= \{q_1, q_2, q_4\} \\ Cerradura-\epsilon(q_2) &= \{q_2, q_4\} \\ Cerradura-\epsilon(q_3) &= \{q_3, q_4\} \\ Cerradura-\epsilon(q_4) &= \{q_4\} \\ Cerradura-\epsilon(q_5) &= \{q_5\} \end{aligned}$$

La Cerradura- $\epsilon(P)$ para P un conjunto de estados se define por

$$\bigcup_{q \in P} Cerr-\epsilon(q)$$

Utilizaremos indistintamente $Cerr-\epsilon(p)$ ó $Cerradura-\epsilon(p)$.

La función de transición $\hat{\delta} : Q \times \Sigma^* \longrightarrow 2^Q$ se define por

1. $\hat{\delta}(q, \epsilon) = Cerr-\epsilon(q)$
2. Para $w \in \Sigma^*, a \in \Sigma$

$$\hat{\delta}(q, wa) = Cerr-\epsilon(P)$$

$$\text{donde } P = \{p \mid r \in \hat{\delta}(q, w), p \in \delta(r, a)\}.$$

Es conveniente extender $\hat{\delta}$ y δ para que tomen conjuntos de estados como argumentos:

3. $\hat{\delta}(R, a) = \bigcup_{q \in R} \delta(q, a)$,
4. $\hat{\delta}(R, w) = \bigcup_{q \in R} \hat{\delta}(q, w)$

A diferencia que en el caso de AFND, los conjuntos $\hat{\delta}(q, a)$ y $\delta(q, a)$ no siempre son iguales pues $\hat{\delta}(q, a)$ incluye a todos los estados alcanzables desde q etiquetados con a (incluyendo rutas etiquetadas con ϵ) mientras que $\delta(q, a)$ incluye solamente estados alcanzables desde q etiquetados con a . De manera similar, $\hat{\delta}(q, \epsilon) \neq \delta(q, \epsilon)$ (Se invita al lector a que analice el porqué).

El lenguaje aceptado por un AFND- ϵ M es:

$$L(M) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

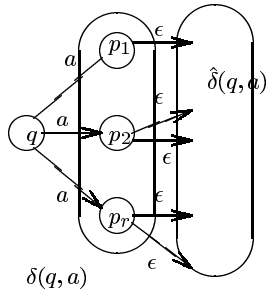


Figura 3.14: Diferencia entre δ y $\hat{\delta}$.

Ejemplo 3.3.2 Sea el AFND- ϵ M_6 :

δ	0	1	2	ϵ
$\rightarrow q_0$	$\{q_0\}$	\emptyset	\emptyset	$\{q_0, q_1\}$
q_1	\emptyset	$\{q_1\}$	\emptyset	$\{q_1, q_2\}$
$F: q_2$	\emptyset	\emptyset	$\{q_2\}$	$\{q_2\}$

El AFND- ϵ acepta la cadena w si y sólo si

$$\hat{\delta}(q_0, w) \cap F \neq \emptyset$$

¿Acepta éste AFND- ϵ la cadena “001”?

Para ello, debemos

1. Calcular las $Cerr-\epsilon$ para cada estado.
2. Calcular $\hat{\delta}(q, a)$ para cada $a \in \Sigma$, y para cada estado $q \in Q$.
3. Evaluar recursivamente $\hat{\delta}(q_0, w)$
4. Si $\hat{\delta}(q_0, w) \cap F \neq \emptyset$ entonces el AFND- ϵ acepta w , de lo contrario, rechaza a w .

Paso a paso:

1.

$$\begin{aligned} Cerr-\epsilon(q_0) &= \{q_0, q_1, q_2\} \\ Cerr-\epsilon(q_1) &= \{q_1, q_2\} \\ Cerr-\epsilon(q_2) &= \{q_2\} \end{aligned}$$

2.

$$\begin{aligned} \hat{\delta}(q_0, '0') &= Cerr-\epsilon(\{p \mid r \in \hat{\delta}(q_0, \epsilon), p \in \delta(r, 0)\}) \\ &= Cerr-\epsilon(\{p \mid r \in \{q_0, q_1, q_2\}, p \in \delta(r, '0')\}) \\ &= Cerr-\epsilon(\delta(q_0, '0') \cup \delta(q_1, 0) \cup \delta(q_2, '0')) \\ &= Cerr-\epsilon(\{q_0\} \cup \emptyset \cup \emptyset) \\ &= Cerr-\epsilon(\{q_0\}) = Cerr-\epsilon(q_0) \\ &= \{q_0, q_1, q_2\}. \end{aligned}$$

$$\begin{aligned} \hat{\delta}(q_0, '1') &= Cerr-\epsilon(\{p \mid r \in \hat{\delta}(q_0, \epsilon), p \in \delta(r, '1')\}) \\ &= Cerr-\epsilon(\delta(q_0, '1') \cup \delta(q_1, '1') \cup \delta(q_2, '1')) \\ &= Cerr-\epsilon(\{q_1\}) = \{q_1, q_2\} \end{aligned}$$

$$\begin{aligned} \hat{\delta}(q_0, '2') &= Cerr-\epsilon(\{p \mid r \in \hat{\delta}(q_0, \epsilon), p \in \delta(r, '2')\}) \\ &= Cerr-\epsilon(\delta(q_0, '2') \cup \delta(q_1, '2') \cup \delta(q_2, '2')) \\ &= Cerr-\epsilon(\{q_2\}) = \{q_2\}. \end{aligned}$$

3.

$$\begin{aligned} \hat{\delta}(q_0, "001") &= Cerr-\epsilon(\{p \mid r \in \hat{\delta}(q_0, "00"), p \in \delta(r, '1')\}) \\ \hat{\delta}(q_0, "00") &= Cerr-\epsilon(\{p \mid r \in \hat{\delta}(q_0, "0"), p \in \delta(r, '0')\}) \\ &= Cerr-\epsilon(\delta(q_0, '0') \cup \delta(q_1, '0') \cup \delta(q_2, '0')) \\ &= Cerr-\epsilon(\{q_0\}) = \{q_0, q_1, q_2\}. \end{aligned}$$

Entonces:

$$\hat{\delta}(q_0, "001") = Cerr-\epsilon(\delta(q_0, '1') \cup \delta(q_1, '1') \cup \delta(q_2, '1')) = \{q_1, q_2\} \cap \{q_1, q_2\} \cap \{q_2\} = \{q_2\}.$$

Por lo tanto el AFND- ϵ acepta la cadena “001”.

3.3.1 Equivalencia entre AFND con y sin movimientos ϵ

Mostraremos cómo simular (obtener/ transformar) un AFND- ϵ mediante un AFND sin movimientos ϵ . La idea yace en el hecho que la *Cerradura- ϵ* empaqueta conjuntos de estados alcanzables a través de ϵ de tal suerte que funciona como una relación de equivalencia.

Teorema 3.3.1 Si L es un lenguaje aceptado por un AFND- ϵ entonces L es aceptado por un AFND sin movimientos ϵ .

Demostración:

Sea $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ un AFND- ϵ .

Se construirá un AFND sin movimientos ϵ

$M' = \langle Q, \Sigma, \delta', q_0, F' \rangle$ donde

$$F' = \begin{cases} F \cup \{q_0\} & \text{si } Cerr-\epsilon(q_0) \cap F \neq \emptyset \\ F & \text{en otro caso.} \end{cases}$$

$$\delta'(q, a) = \hat{\delta}(q, a) \text{ para } q \in Q, a \in \Sigma$$

La garantía que δ' no contiene transiciones ϵ yace en el hecho que $\hat{\delta}$ utiliza a la *Cerradura- ϵ* .

Mostraremos por inducción sobre la longitud de la cadena x que $\delta'(q_0, x) = \hat{\delta}(q_0, x)$

Dado que en general $\delta'(q_0, \epsilon) = \{q_0\} \neq \hat{\delta}(q_0, \epsilon)$ tenemos que comenzar el caso base en 1:

- Base: $|x| = 1$, $x \in \Sigma$, $\delta'(q_0, x) = \hat{\delta}(q_0, x)$.

- Paso inductivo: $|x| > 1$. Sea $x = wa$, $a \in \Sigma$.

$\delta'(q_0, wa) = \delta'(\delta'(q_0, w), a)$ por hipótesis de inducción

$\delta'(q_0, w) = \hat{\delta}(q_0, w)$.

Sea $P = \hat{\delta}(q_0, w)$, entonces hay que mostrar que

$$\delta'(P, a) = \hat{\delta}(q_0, wa)$$

es decir,

$$\begin{aligned} \delta'(P, a) &= \bigcup_{p \in P} \delta'(p, a) = \bigcup_{p \in P} \hat{\delta}(p, a) \\ \bigcup_{q \in P} \delta'(q, a) &= \hat{\delta}(q_0, wa) \end{aligned}$$

Para completar, mostraremos que $\delta'(q_0, x)$ contiene un estado de F' si y sólo si $\hat{\delta}(q_0, x)$ contiene un estado de F :

- Si $x = \epsilon$ lo anterior es inmediato por la definición de F' : $\delta'(q_0, \epsilon) = \{q_0\}$ y a su vez $q_0 \in F'$ siempre que $Cerr-\epsilon(q_0)$ contenga un estado de F .
- Si $x \neq \epsilon$, entonces tiene la forma wa . Si $\delta'(q_0, x)$ contiene un estado de F entonces $\delta'(q_0, x)$ también contiene el mismo estado en F' . Y visceversa, si $\delta'(q_0, x)$ contiene un estado de F' (distinto de q_0) entonces $\hat{\delta}(q_0, x)$ contiene un estado de F . Si $\delta'(q_0, x)$ contiene a q_0 y $q_0 \notin F$ entonces dado que $\hat{\delta}(q_0, x) = Cerr-\epsilon(\delta(\hat{\delta}(q_0, w), a))$ entonces aquel estado que esté en $Cerr-\epsilon(q_0)$ y en F también debe estar en $\hat{\delta}(q_0, x)$.

Ejemplo 3.3.3 Mostramos a continuación el AFND sin movimientos ϵ correspondiente al AFND- ϵ M_6 del ejemplo 3.3.2 anterior:

Como ya tenemos $\hat{\delta}(q_0, a)$ para cada $a \in \Sigma$, sólo resta obtener $\hat{\delta}(q_i, a)$ para $i = 1, 2$:

$$\begin{aligned} \hat{\delta}(q_1, 0) &= Cerr-\epsilon(\delta(\hat{\delta}(q_1, \epsilon), 0)) = \emptyset \\ \hat{\delta}(q_1, 1) &= \{q_1, q_2\} \\ \hat{\delta}(q_1, 2) &= \{q_2\} \\ \hat{\delta}(q_2, 0) &= \emptyset \\ \hat{\delta}(q_2, 1) &= \emptyset \\ \hat{\delta}(q_2, 2) &= \{q_2\} \end{aligned}$$

La tabla de transición para M'_6 queda

δ'	0	1	2
$\rightarrow F: q_0$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1	\emptyset	$\{q_1, q_2\}$	$\{q_2\}$
q_2	\emptyset	\emptyset	$\{q_2\}$

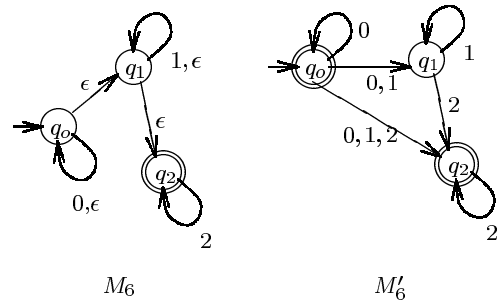


Figura 3.15: AFND sin mov. ϵ M'_6 equivalente al AFND- ϵ M_6 .

Ejercicios 3.3.1

- Para cada uno de los AFND- ϵ que se muestran en la fig. 3.16:
 - Indique quienes son cada uno de los componentes (i.e., Q, Σ, δ, F);
 - Muestre si las cadenas que se indican son aceptadas ó rechazadas:
01010101, 110011
 - Construya el AFND (sin mov. ϵ) correspondiente.
 - Justifique cuál es el lenguaje aceptado.

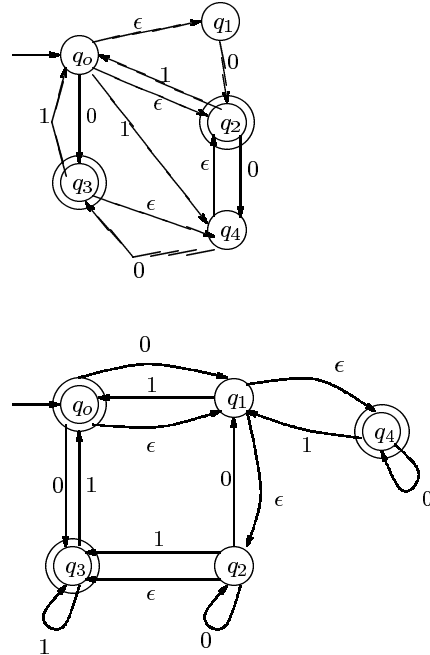


Figura 3.16: Grafos de los AFND- ϵ .

Expresiones Regulares

Los lenguajes que aceptan los AF pueden describirse de manera muy conveniente mediante *Expresiones Regulares*, los cuales podemos considerar como ‘esqueletos’ ó patrones que denotan *conjuntos regulares*.

Definición 4.1.3 Sea Σ un alfabeto. Las *Expresiones Regulares (ER)* definidas sobre Σ y los conjuntos regulares que denotan se definen recursivamente:

ER	Conjunto
\emptyset	\emptyset
a	$\{a\}$
ϵ	$\{\epsilon\}$
$(\mathbf{r} + \mathbf{s})$	$R \cup S$
(\mathbf{rs})	RS
(\mathbf{r}^*)	R^*

donde \mathbf{r}, \mathbf{s} son ERs que denotan a los conjuntos R, S respectivamente. Nótese que los paréntesis únicamente son agrupadores.

Ejemplo 4.1.2 1. Sea la ER $(\mathbf{0} + \mathbf{1})^*$. Analizando detalladamente (obteniendo los correspondientes conjuntos para cada subexpresión):

- $\mathbf{0}$ es la ER que denota al conjunto $\{0\}$,
- $\mathbf{1}$ es la ER que denota al conjunto $\{1\}$,
- $(\mathbf{0} + \mathbf{1}) = \{0\} \cup \{1\} = \{0, 1\}$

Entonces:

$$\{\mathbf{0}, \mathbf{1}\}^* = \{\epsilon, 0, 1, 01, 10, 11, 00, 000, 001, \dots\}$$

es decir, representa el conjunto de todas las posibles cadenas que se pueden formar con ‘0’ y ‘1’.

Podemos prescindir de los paréntesis si establecemos la siguiente *jerarquía de prioridades y asociatividad* de los operadores de Expresiones Regulares: la cerradura $*$ asocia por la izquierda y tiene la mayor prioridad, a continuación la concatenación (asocia por la derecha) y finalmente el operador de alternativa $+$ que también asocia por la derecha.

Por ejemplo la ER $\mathbf{0} + \mathbf{1b}^* + \mathbf{a1}^*$ es igual a

$$(\mathbf{0} + ((\mathbf{1}(\mathbf{b}^*)) + (\mathbf{a}(\mathbf{1}^*))))$$

Analicemos más ejemplos: sea el alfabeto $\Sigma = \{a, b, c\}$,

1. $(\mathbf{a} + \mathbf{b})\mathbf{c}^*$:

$$\begin{aligned} \mathbf{a} &= \{a\} \\ \mathbf{b} &= \{b\} \\ \mathbf{c} &= \{c\} \\ (\mathbf{a} + \mathbf{b}) &= \{a, b\} \\ \mathbf{c}^* &= \{\epsilon, c, cc, ccc, cccc, \dots\} \\ (\mathbf{a} + \mathbf{b})\mathbf{c}^* &= \{a, b, ac, bc, acc, bcc, \dots\} \end{aligned}$$

Por tanto, la ER denota al conjunto de todas las cadenas que comienzan con ‘a’ o ‘b’ seguidas de cualquier número de ‘c’s incluyendo ϵ .

2. $(\mathbf{a}^*\mathbf{c}) + (\mathbf{ab}^*)$.

Tomamos $r = (\mathbf{a}^*\mathbf{c})$ y $s = \mathbf{ab}^*$

$$\begin{aligned} r : \quad \mathbf{a}^* &= \{\epsilon, a, aa, aaa, aaaa, \dots\} \\ \mathbf{a}^*\mathbf{c} &= \{c, ac, aac, aaac, aaaac, \dots\} \end{aligned}$$

$$\begin{aligned} s : \quad \mathbf{b}^* &= \{\epsilon, b, bb, bbb, bbbb, \dots\} \\ \mathbf{ab}^* &= \{a, ab, abb, abbb, abbbb, \dots\} \end{aligned}$$

$$r + s = \{a, c, ac, ab, aac, abb, aaac, abbb, \dots\}$$

3. Sea la ER $(\mathbf{00})^* + (\mathbf{1(11)})^*$ definida sobre el alfabeto $\{0, 1\}$. Tomamos $r = (\mathbf{00})^*$ y $s = \mathbf{1(11)}^*$.

$$\begin{aligned} r : \quad (\mathbf{00}) &= \{00\} \\ (\mathbf{00})^* &= \{\epsilon, 00, 0000, 000000, \dots\} \end{aligned}$$

$$\begin{aligned} s : \quad (\mathbf{11}) &= \{11\} \\ (\mathbf{11})^* &= \{\epsilon, 11, 1111, 111111, \dots\} \\ \mathbf{1(11)}^* &= \{1, 111, 11111, 1111111, \dots\} \end{aligned}$$

$$r + s = \{\epsilon, 00, 0000, \dots, 1, 111, 11111, \dots\}$$

son todas las cadenas pares de 0’s ó cadenas impares de 1’s

4.2 Equivalencia entre AF y ER

Teorema 4.2.1 Sea r una expresión regular que denota al conjunto R . Existe un AFND- ϵ M que acepta al lenguaje que denota r : $L(M) = R$.

Demostración: aplicaremos inducción sobre el número de operadores involucrados en la Expresión Regular r , el AFND- ϵ tendrá un estado final y ninguna transición fuera de ese estado final, tal que $L(M) = L(r)$.

- Caso base: el número de operadores es 0 (no hay alguno), la ER r corresponde a alguna de \emptyset, ϵ, a como se muestra en la Figura 4.17

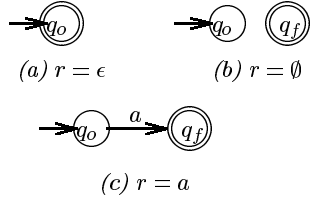


Figura 4.17: AFD para los casos base.

- Paso inductivo: Se asume que la afirmación es verdadera para toda ER con menos de i operadores, para $i \geq 1$. Sea r con i operadores:

1. $r = r_1 + r_2$, ambos r_1 y r_2 tienen menos de i operadores. Aplicando la HI: existen AFND- ϵ

$$\begin{aligned} M_1 &= \langle Q_1, \Sigma_1, \delta_1, q_{o1}, \{f_1\} \rangle \\ M_2 &= \langle Q_2, \Sigma_2, \delta_2, q_{o2}, \{f_2\} \rangle \end{aligned}$$

tales que $L(M_1) = L(r_1)$ y $L(M_2) = L(r_2)$. Se asume que Q_1 y Q_2 son disjuntos.

El AFND- ϵ resultante será

$$M = \langle Q_1 \cup Q_2 \cup \{q_o, f\}, \Sigma_1 \cup \Sigma_2, \delta, q_o, \{f_o\} \rangle$$

donde

$$\begin{aligned} \delta(q_o, \epsilon) &= \{q_{o1}, q_{o2}\} \\ \delta(q, a) &= \begin{cases} \delta_1(q, a) & q \in Q_1 - \{f_1\}, \\ & a \in \Sigma_1 \cup \{\epsilon\} \\ \delta_2(q, a) & q \in Q_2 - \{f_2\}, \\ & a \in \Sigma_2 \cup \{\epsilon\} \end{cases} \\ \delta(f_1, \epsilon) &= \{f_o\} = \delta(f_2, \epsilon) \end{aligned}$$

Por tanto :

$$L(M) = L(M_1) \cup L(M_2)$$

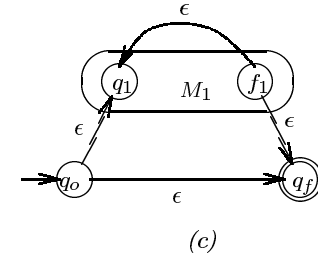
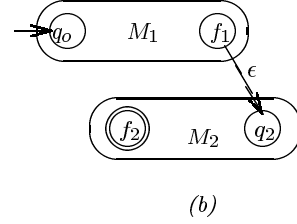
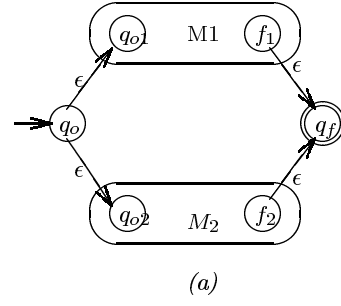


Figura 4.18: Construcciones para cada caso en el paso inductivo: (a) para unión, (b) para concatenación y (c) para la cerradura Kleene.

2. $r = r_1 r_2$. Sean M_1, M_2 como el caso anterior. Entonces

$$M = \langle Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta, q_{o1}, \{f_2\} \rangle$$

donde

$$\delta(q_o, a) = \begin{cases} \delta_1(q, a) & \text{para } q \in Q_1 - \{f_1\}, \\ & a \in \Sigma_1 \cup \{\epsilon\}. \\ \delta_2(q, a) & \text{para } q \in Q_2, \\ & a \in \Sigma_2 \cup \{\epsilon\}. \end{cases}$$

$$\delta(f_1, \epsilon) = \{q_{o2}\}.$$

Por tanto:

$$L(M) = L(M_1)L(M_2)$$

3. $r = r_1^*$. Sea M_1 como los casos anteriores

$$M = \langle Q_1 \cup \{q_o, f_o\}, \Sigma_1, \delta_1, q_o, \{f_o\} \rangle$$

donde

$$\begin{aligned} \hat{\delta}(q_o, \epsilon) &= \hat{\delta}_1(f_1, \epsilon) = \{q_1, f_o\} \\ \hat{\delta}(q, a) &= \delta_1(q, a), \text{ para } q \in Q_1 - \{f_1\}, \\ & \quad a \in \Sigma_1 \cup \{\epsilon\} \end{aligned}$$

Por tanto :

$$L(M) = L(M_1)^*$$

Ejemplo 4.2.1 Construiremos un AFND- ϵ para la ER $ab^* + a$.

Aplicamos las reglas en orden de acuerdo a como está agrupada la expresión: $ab^* + a = (a(b^*)) + a$, por lo que a nivel más externo tiene la forma $r_1 + r_2$ donde $r_1 = a(b^*)$ y $r_2 = a$. A su vez, la ER r_1 tiene la forma $r_3 r_4$ donde $r_3 = a$ y $r_4 = b^*$; ésta última tiene la forma $r_4 = r_5^*$ donde $r_5 = b$. Las AFND- ϵ para las ER r_2, r_3 y r_5 se muestran en la Figura 4.19. Es

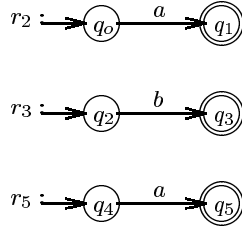


Figura 4.19: AFND- ϵ básicos del ejemplo.

de recalcar que los estados deben estar nombrados de manera distinta entre autómatas que corresponden a diferentes ERs, aún cuando denoten el mismo conjunto: en el ejemplo, es evidente que $r_2 = r_5$ pero los estados de sus autómatas deben ser diferentes (para evitar ambigüedad al agruparlos).

Una vez construídos los *casos básicos* procedemos a construir los AFND- ϵ asociados a las expresiones compuestas r_1, r_4 , etc., en los cuales incorporamos estados como sea necesario: el autómata para r_4 queda como se muestra en la Figura 4.20 (a), mientras que el correspondiente a r_1 es (b) y el AFND- ϵ para la ER inicial es (c).

4.3 Transformación de un AFD a ER

En el Capítulo 2 acostumbramos analizar el conjunto de cadenas que fuesen aceptadas por algún AFD. Tal análisis se basó en aplicar el principio de inducción sobre la longitud de las cadenas en las cuales el AFD alcanza algún estado de aceptación. De esa forma, pudimos caracterizar por casos a las palabras que acepta un AFD dado. Ahora haremos más formal tal análisis y ofreceremos una especie de *forma normal*¹⁶

¹⁶Muy similar a la forma normal conjuntiva en el lenguaje de la lógica matemática.

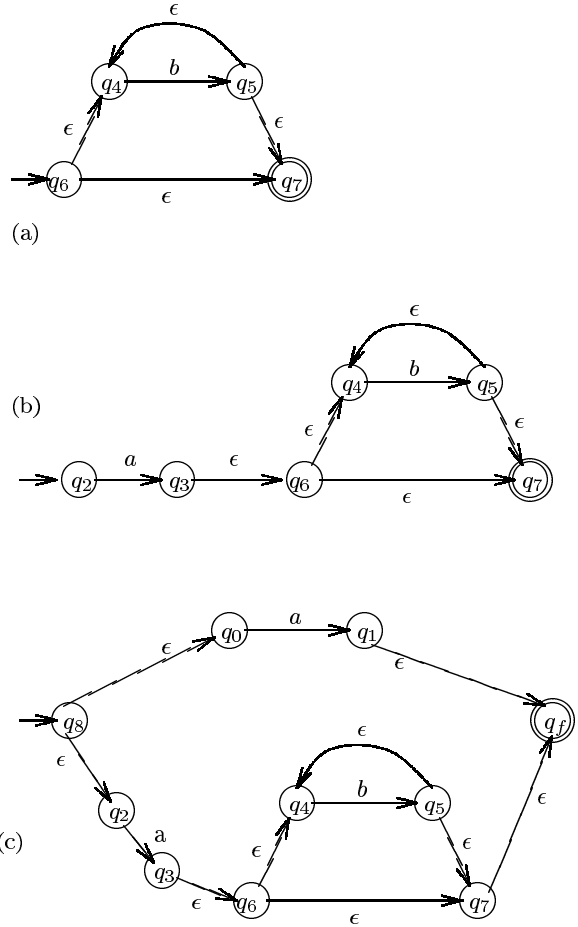


Figura 4.20: AFND- ϵ correspondiente a la ER $ab^* + a$.

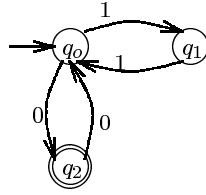
para las expresiones regulares obtenidas a partir de un AFD.

La idea intuitiva consiste en agrupar los conjuntos de cadenas que pueden ser reconocidos desde un estado q_i hasta un estado q_j dentro de un subconjunto de estados X tomando un estado arbitrario como “pivote”; es decir, como si quisiéramos cruzar un río y nos apoyamos en una piedra (el pivote) para alcanzar la otra orilla. El pivote tiene el propósito de separar los casos de las formas que tenemos para cruzar (i.e., las ER asociadas con cada camino).

Ilustraremos la idea con un ejemplo: sea $M_a = \langle Q, \Sigma, \delta, q_0, F \rangle$ donde $Q = \{q_0, q_1, q_2\}$, $F = \{q_2\}$ y

δ	0	1
$\rightarrow q_0$	q_2	q_1
q_1		q_2
$F:q_2$	q_0	

Consideraremos al conjunto de estados $\{q_0, q_2\}$ en el que tomamos como pivote a q_1 .



El conjunto de cadenas aceptadas por este autómata, será representado inductivamente por la expresión regular:

$$\alpha_{q_0, q_2}^Q$$

desde el estado inicial q_0 hasta el estado final q_2 .

En general, se puede elegir cualquier estado como pivote piv y la ER correspondiente es de la forma

$$\alpha_{q_0, q_f}^Q = \alpha_{q_0, q_f}^{Q-piv} + \alpha_{q_0, piv}^{Q-piv} (\alpha_{piv, piv}^{Q-piv})^* \alpha_{piv, q_f}^{Q-piv}$$

Esta expresión significa lo siguiente:

1. Todas las cadenas que puede ver estando el AFD en q_0 y llegando a un estado de aceptación q_f sin tocar el pivote:

$$\alpha_{q_0, q_f}^{Q-piv}$$

ó

2. Todas las cadenas que analiza a partir de q_0 y llegando al pivote, pasando por cualquier estado (excepto el pivote)

$$\alpha_{q_0, piv}^{Q-piv}$$

3. A continuación, las cadenas que se pueden analizar estando repetidamente en el pivote

$$(\alpha_{piv, piv}^{Q-piv})^*$$

4. Finalmente, todas las cadenas que llevan del pivote a un estado de aceptación (salen del pivote).

$$\alpha_{piv, q_f}^{Q-piv}$$

De acuerdo a esto, tenemos las siguientes ER's para M_a (teniendo como pivote a q_1):

$$\alpha_{q_0, q_2}^{q_0 q_1 q_2} = \alpha_{q_0, q_2}^{q_0 q_2} + \alpha_{q_0, q_1}^{q_0 q_2} (\alpha_{q_1, q_1}^{q_0 q_2})^* \alpha_{q_1, q_2}^{q_0 q_2}$$

La primer expresión, $\alpha_{q_0, q_2}^{q_0 q_2}$, indica el conjunto que reconoce el AF partiendo de q_0 y llegando a q_2 utilizando sólo estados $\{q_0, q_2\}$: del digrafo de M_a podemos apreciar que tal ruta acepta cadenas de la forma $0(00)^*$, i.e., cadenas de longitud impar de ceros consecutivos.

Para la expresión $\alpha_{q_0, q_1}^{q_0 q_2}$, indica todas las cadenas que el AF reconoce partiendo de q_0 y llegando a q_1 pasando solamente por los estados $\{q_0, q_2\}$: en otras palabras, de qué manera se puede llegar al estado pivote. A partir del digrafo podemos afirmar que tales cadenas tienen dos partes: un prefijo de número par de ceros (el ciclo de $q_0 \rightarrow q_2 \rightarrow q_0$) y a continuación un 1, i.e., esa parte corresponde a la ER $(00)^*1$.

En seguida, la expresión $(\alpha_{q_1, q_1}^{q_0 q_2})^*$ indica todas las cadenas que reconoce el AF estando en un ciclo sobre el pivote pero utilizando únicamente los estados $\{q_0, q_2\}$: tenemos los siguientes casos:

1. no se mueve: i.e., ϵ ,
2. leer un 1, lo cual hace que se active la transición de q_1 a q_0 , estando en q_0 puede leer cualquier secuencia de 0 par (por el ciclo entre $q_0 \rightarrow q_2$) y entonces regresar a q_1 desde q_0 leyendo un 1; lo cual da la ER $1(00)^*1$.

Ambos casos, de acuerdo a la definición, se encierran en la cerradura Kleene: $(\alpha_{q_1, q_1}^{q_0 q_2})^* = (\epsilon + (1(00)^*1))^*$.

Finalmente, la ER $\alpha_{q_1, q_2}^{q_0 q_2}$ indica las cadenas que se reconocen partiendo desde q_1 y llegando a q_2 (sin pasar por q_1 , i.e., se sale por última vez del pivote): ello significa leer un 1 (para llegar a q_0), y leer al menos un 0 para alcanzar q_2 , estando ahí se puede aceptar cualquier secuencia de 0's par (por el ciclo entre q_0 y q_2). Por tanto la ER en éste caso es $10(00)^*$.

Incorporando todas, la ER que describe el lenguaje de M_a es:

$$\alpha_{q_0, q_2}^{q_0 q_1 q_2} = 0(00)^* + (00)^*1(\epsilon + (1(00)^*1))^*10(00)^*$$

Se invita al lector a que elija ahora otro estado como pivote y pueda verificar que el conjunto regular resultante es equivalente al que denota la ER que se acaba de obtener.

Formalizamos la idea anterior mediante el siguiente resultado.

Teorema 4.3.1 Si L es el lenguaje aceptado por un AFD¹⁷ M , entonces L se denota por una ER.

Demostración: Dado un AFD $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ sea $X \subseteq Q$, para cualquier estado q_i, q_j en Q se puede construir una ER dada por

$$\alpha_{i, j}^X$$

que denote el conjunto de todas las cadenas w tales que existe una ruta de q_i a q_j (con la posible excepción de q_i, q_j) en M etiquetada por w .

¹⁷La misma idea se extiende naturalmente en general a AFND.

Todos los estados a lo largo de tal ruta (con la posible excepción de $q_i = \text{inicio}$ y $q_j = \text{fin}$) están en X : $q_j \in \hat{\delta}(q_i, w)$.

Las expresiones regulares se construyen inductivamente sobre el tamaño del subconjunto X :

1. Caso base: $|X| = 0$ (es decir $X = \emptyset$).

Sean a_1, a_2, \dots, a_k todos los símbolos del alfabeto tales que $q_j \in \delta(q_i, a_l)$, para $1 \leq l \leq k$.

Si $q_i \neq q_j$, sea

$$\alpha_{i,j}^{\emptyset} \stackrel{\text{def}}{=} \begin{cases} a_1 + a_2 + \dots + a_k, & \text{si } k \geq 1 \\ \emptyset & \text{si } k = 0 \end{cases}$$

y si $q_i = q_j$

$$\alpha_{i,j}^{\emptyset} \stackrel{\text{def}}{=} \begin{cases} a_1 + a_2 + \dots + a_k + \epsilon, & \text{si } k \geq 1 \\ \epsilon & \text{si } k = 0 \end{cases}$$

2. Paso inductivo: si $X \neq \emptyset$, podemos elegir cualquier elemento $q \in X$ y tomar

$$\alpha_{i,j}^X \stackrel{\text{def}}{=} \alpha_{i,j}^{X-q} + \alpha_{i,q}^{X-q} (\alpha_{q,q}^{X-q})^* \alpha_{q,j}^{X-q}$$

Nótese que cualquier ruta de q_i a q_j con intermedios en X ya sea que:

- (a) nunca visita a q (el pivote)
- (b) visita a q por primera vez, seguida por un número, (posiblemente 0) de ciclos de q a si mismo, seguido de una ruta de q a q_j (dejando q por ultima vez).

La suma de las expresiones de la forma

$$\alpha_{q_0, q_f}^Q$$

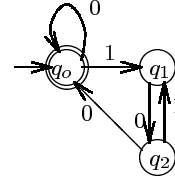
representa el conjunto de cadenas aceptadas por M .

Como regla del pulgar (heurística) se recomienda elegir como estado pivote $q \in X$ aquel que “desconecte” el AFD tanto como sea posible.

Ejemplo 4.3.1 Sea el AFD

$$M = \langle \{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_0\} \rangle$$

δ	0	1
$F \rightarrow q_0$	q_0	q_1
q_1	q_2	
q_2	q_0	q_1



Denotaremos a la ER correspondiente a M por $\alpha_{q_0, q_0}^{\{q_0, q_1, q_2\}}$. Calcularemos paso a paso:

Sea $X = Q - \{q_1\} = \{q_0, q_2\}$

$$\alpha_{q_0, q_0}^X = \alpha_{q_0, q_0}^X + \alpha_{q_0, q_1}^X (\alpha_{q_1, q_1}^X)^* \alpha_{q_1, q_0}^X$$

$$\alpha_{q_0, q_0}^X = 0^*$$

$$\alpha_{q_0, q_1}^X = 0^*1$$

$$\alpha_{q_1, q_1}^X = 01 + 000^*1$$

$$\alpha_{q_1, q_1}^X = 000^*$$

Por tanto la ER correspondiente es:

$$\alpha_{q_0, q_0}^X = 0^* + 0^*1(01 + 000^*1)^*000^*$$

4.3.1 Simplificación de Expresiones

Para ER's α, β si $L(\alpha) = L(\beta)$ decimos que α y β son *equivalentes* y lo denotamos por $\alpha \equiv \beta$. El lector debe demostrar que \equiv es una relación de equivalencia.

Si $\alpha \equiv \beta$, podemos substituir α por β (o viceversa) en cualquier ER, el resultado será equivalente a la ER original. A continuación se muestran un conjunto de equivalencias que se pueden utilizar como una forma de *Álgebra de Expresiones Regulares*:

$$1. \alpha + (\beta + \gamma) \equiv (\alpha + \beta) + \gamma$$

$$2. \alpha + \beta \equiv \beta + \alpha$$

$$3. \alpha + \emptyset \equiv \alpha$$

$$4. \alpha + \alpha \equiv \alpha$$

$$5. \alpha(\beta\gamma) \equiv (\alpha\beta)\gamma$$

$$6. \epsilon\alpha \equiv \alpha\epsilon \equiv \alpha$$

$$7. \alpha(\beta + \gamma) \equiv \alpha\beta + \alpha\gamma$$

$$8. (\alpha + \beta)\gamma \equiv \alpha\gamma + \beta\gamma$$

$$9. \emptyset\alpha \equiv \alpha\emptyset \equiv \emptyset$$

$$10. \emptyset + \alpha\alpha^* \equiv \emptyset + \alpha^*\alpha \equiv \alpha^*$$

$$11. (\alpha\beta)^*\alpha \equiv \alpha(\beta\alpha)^*$$

12. $(\alpha^*\beta)^*\alpha^* \equiv (\alpha + \beta)^*$

13. $(\epsilon + \alpha)^* \equiv \alpha^*$.

Ejemplo 4.3.2 Sea la ER $(1 + 01 + 001)^*(\epsilon + 0 + 00)$.

Utilizando las reglas anteriores podemos simplificarla de la siguiente forma:

$$\begin{aligned} & (1 + 01 + 001)^*(\epsilon + 0 + 00) \\ & \equiv ((\epsilon + 0 + 00)1)^*(\epsilon + 0 + 00) \\ & \equiv ((\epsilon + 0)(\epsilon + 0)1)^*(\epsilon + 0)(\epsilon + 0). \end{aligned}$$

De la ER resultante, se puede apreciar que denota el conjunto de cadenas tomadas del alfabeto $\{0, 1\}$ tales que no hay subcadena con más de dos 0 adyacentes.

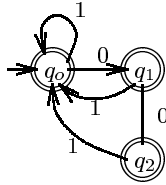


Figura 4.21: AFND-ε para la ER del ejemplo 4.3.2.

Como resultado de las construcciones, podemos apreciar la equivalencia entre las distintas representaciones de los conjuntos regulares. En cada caso, se puede hacer uso de una representación si es que resulta ser más conveniente, pero el poder expresivo es el mismo.

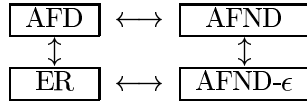


Figura 4.22: Las cuatro construcciones principales que muestran la equivalencia entre los tipos de Autómatas Finitos y Expresiones Regulares: todos denotan a la misma clase de lenguajes.

Ejercicios 4.3.1

1. Escriba expresiones regulares para cada uno de los siguientes lenguajes definidos sobre el alfabeto $\{0, 1\}$:
 - (a) El conjunto de todas las cadenas con a lo más una pareja de '0's consecutivos y a lo más una pareja de '1's consecutivos.
 - (b) El conjunto de todas las cadenas con cada pareja de '0's adyacentes antes de cada pareja de '1's adyacentes.

- (c) El conjunto de todas las cadenas que no contengan a la subcadena '101'.

2. Describa cuáles son los conjuntos de cadenas denotadas por las siguientes expresiones regulares:

- (a) $(11 + 0)^*(00 + 1)^*$
- (b) $(1 + 01 + 001)^*(\epsilon + 0 + 00)^*$
- (c) $(00 + 11 + (01 + 10)(00 + 11)^*)^*$

3. Construya los correspondientes autómatas que aceptan los mismos lenguajes que las siguientes expresiones regulares:

- (a) $10 + (0 + 11)0^*1$
- (b) $01(((10)^* + 111)^* + 0)^*1$
- (c) $((0+1)+(0+1))^* + ((1+0)(1+0)(1+0))^*$

4. Encontrar la ER asociada a cada uno de los AFD que se muestran en la Figura 4.23.

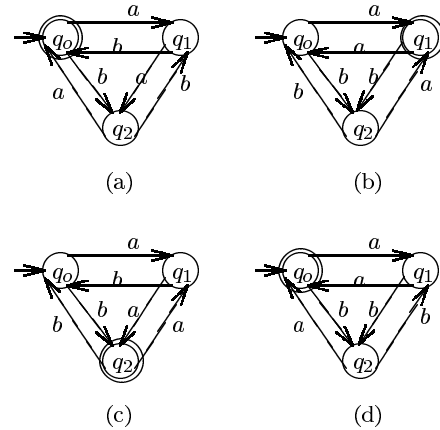


Figura 4.23:

Propiedades de los Conjuntos Regulares

En éste capítulo analizaremos algunas de las propiedades más importantes de los conjuntos regulares a saber: propiedades de cerradura, minimización de estados y equivalencias; así como poder demostrar cuándo un conjunto no es regular, i.e., no puede ser aceptado por ningún AF.

5.1 Limitaciones de los AF

Ya hemos estudiado lo que los AF pueden hacer, veamos que es lo que no pueden hacer. El ejemplo canónico de un conjunto no regular (aquel que no puede ser aceptado por algún AF) es

$$B = \{a^n b^n \mid n \geq 0\} \\ = \{\epsilon, ab, aabb, aaabbb, \dots\}$$

el conjunto de todas las cadenas de la forma a^*b^* con exactamente igual número de a 's que de b 's.

Intuitivamente, para poder aceptar el conjunto B un AF examinando una cadena de la forma a^*b^* tendría que recordar cuando pasó por el punto central entre la parte de a 's y la parte de b 's de tal manera que pueda *recordar* el núm. de a 's que haya visto y verificar que coincide con el número de b 's en cuyo caso aceptaría la cadena

$$aaaa \dots abbb \dots b \\ \uparrow \\ q$$

Más aún, el AF tendría que tener esa capacidad para hacer lo mismo para cualquier cadena de tamaño arbitrario, mayor al número de estados que disponga (recuérdese que Q siempre es finito). Ello constituye una cantidad de información ilimitada dado que no hay manera de que el AF pueda recordar un número infinito de casos utilizando solamente memoria finita. Todo lo que el AF "sabe" es el estado q en el que se encuentra en el momento actual.

Podemos ofrecer una demostración formal por contradicción de que el conjunto B no es regular.

Asumiendo que B fuese regular, significaría que existe un AF $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ tal que $L(M) = B$. Sea $k = |Q|$. Considérese la acción de M ante la entrada $a^n b^n$, para $n \gg k$ (n mucho mayor que k , pero el argumento funciona a partir de $n = k+1$). M comienza en el estado q_0 , dado que la cadena $a^n b^n$ está en B , M debe aceptarla; así que M debe alcanzar algún estado final q_f después de examinar $a^n b^n$:

$$\overbrace{aaaaaaaaaaaaaaaa} \overbrace{bbbbbbbbbbbbbbbb} \\ \uparrow \qquad \qquad \qquad \uparrow \\ q_0 \qquad \qquad \qquad q_f$$

Dado que $n \gg k$, debe existir algún estado p por el cual M necesariamente debe pasar más de una vez mientras está examinando la secuencia inicial de a 's. Ello significa que podemos desmenuzar la cadena $a^n b^n$ en tres subcadenas u, v y w , en donde v es la cadena de a 's examinadas entre dos ocurrencias del estado p :

$$\overbrace{aaaaaaaaaaaaaaaa} \overbrace{abbbbbbbbbbbbb} \\ \uparrow \quad u \quad \uparrow \quad \quad \uparrow \quad \quad \quad \uparrow \quad \quad \quad \uparrow \\ q_0 \quad \quad p \quad \quad p \quad \quad \quad \quad \quad q_f$$

Sea $j = |v| > 0$. para éste ejemplo $j = 7$. Podemos apreciar que

$$\hat{\delta}(q_0, u) = p \\ \hat{\delta}(p, v) = p \\ \hat{\delta}(p, w) = q_f$$

La cadena v puede ser eliminada y la cadena resultante sería aceptada (erroneamente!)

$$\overbrace{aaaaaaaa} \overbrace{abbbbbbbbb} \\ \uparrow \quad \quad \uparrow \quad \quad \quad \uparrow \\ q_0 \quad \quad p \quad \quad \quad q_f$$

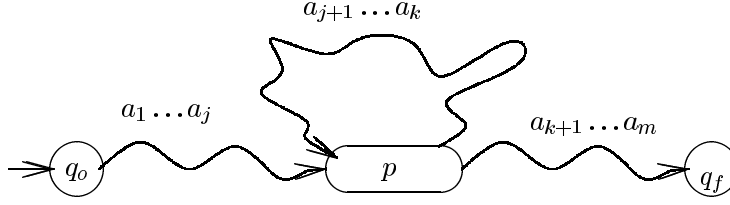


Figura 5.24: Ruta en el diagrama de transiciones del AF M .

Claramente, tal cadena tiene un número estrictamente menor de a 's que de b 's: $uw = a^{n-j}b^n \in L(M)$ pero $uw \notin B$ lo cual es una contradicción de la premisa que $L(M) = B$.

Además, podemos insertar cualquier número de copias que queramos de la subcadena v y la cadena que resulte también sería aceptada por M : $uv^3w = a^{n+2j}b^n \in L(M)$:

$$\begin{aligned} \hat{\delta}(q_0, uvvw) &= \hat{\delta}(\hat{\delta}(\hat{\delta}(\hat{\delta}(\hat{\delta}(q_0, u), v), v), v), w) \\ &= \hat{\delta}(\hat{\delta}(\hat{\delta}(\hat{\delta}(p, v), v), v), w) \\ &= \hat{\delta}(\hat{\delta}(\hat{\delta}(p, v), v), w) \\ &= \hat{\delta}(\hat{\delta}(p, v), w) \\ &= \hat{\delta}(p, w) = q_f \end{aligned}$$

5.1.1 Lema de Sondeo (Pumping)

Podemos expresar de manera muy compacta los argumentos anteriores en un resultado general denominado el *Lema del Sondeo*, el cual es muy útil para demostrar cuando un conjunto no es regular. La idea es que siempre que un AF examine una cadena larga (mayor que el número de estados) y la acepte entonces significa que debe haber un estado repetido (pasa por el más de una vez) y se pueden insertar copias de la subcadena entre las dos ocurrencias de ése estado repetido de tal suerte que la cadena resultante también es aceptada.

Teorema 5.1.1 (Lema del Sondeo) *Sea A un conjunto regular. A cumple con la siguiente propiedad:*

(P) *Existe un $k \geq 0$ tal que para cualesquiera cadenas x, y, z con $xyz \in A$ y $|y| \geq k$, existen cadenas u, v, w tales que $y = uvw$, $v \neq \epsilon$, y para toda $i \geq 0$ la cadena $xuv^i w \in A$.*

Informalmente, si A es regular entonces para cualquier cadena α en A y cualquier subcadena suficientemente larga y (de α) y contiene una subcadena no nula v tal que uno puede obtener tantas copias como se desee y la cadena resultante también

pertenece a A . De aquí el nombre de “sondeo”¹⁸ pues podemos “bombear” tantas veces queramos la cadena v .

Jugando contra el Diablo

Podemos aplicar el Lema del Sondeo para demostrar que un conjunto no es regular de la siguiente forma:

Teorema 5.1.2 (Lema del Sondeo) Forma Contrapositiva. *Sea A un conjunto regular. Supóngase que*

(¬P) *Para toda $k \geq 0$ existen cadenas x, y, z tales que $xyz \in A$ y $|y| \geq k$; y para toda cadena u, v, w con $y = uvw$, $v \neq \epsilon$, existe una $i \geq 0$ tal que la cadena $xuv^i w \notin A$.*

Por tanto, A no es regular.

Para demostrar que un conjunto dado A no es regular necesitamos establecer que (¬P) se cumple para A . La prueba es similar a un juego teniendo como contrincante el diablo: uno desea demostrar que A no es regular y el diablo quiere mostrar que A es regular. El juego procede de la siguiente forma:

1. El diablo elige k (si A de veras es regular, la mejor estrategia del diablo es elegir $k = |Q|$ de algún AF que acepta a A).
2. Uno elige las cadenas x, y, z tales que $xyz \in A$ y $|y| \geq k$.
3. El diablo elige u, v, w tales que $y = uvw$ y $v \neq \epsilon$.
4. Uno elige $i \geq 0$.

Uno gana si $xuv^i w \notin A$, pero el diablo gana si $xuv^i w \in A$. Si uno puede demostrar que se tiene una estrategia vencedora (que siempre se gana el juego sin importar las opciones que tome el diablo) se habrá demostrado esencialmente que la condición (¬P) se cumple para A y por tanto A no es regular.

¹⁸La traducción textual de *pumping* es “bombeo”.

Se debe tener en cuenta la siguiente precaución: existen resultados más fuertes que establecen la regularidad de un conjunto, el lema del sondeo únicamente ofrece una condición necesaria (existen conjuntos que satisfacen **(P)** que no son regulares) por lo que si un conjunto cumple **(P)** no significa que sea regular, para ello se debe construir un AF o una ER correspondiente.

Ejemplo 5.1.1 Aplicaremos el lema del sondeo para demostrar que el conjunto

$$A = \{a^n b^m \mid n \geq m\}$$

no es regular. El diablo apuesta a que A es regular y elige algún núm. k . Una buena respuesta es que uno elija $x = a^k$, $y = b^k$ y $z = \epsilon$. Entonces $xyz = a^k b^k \in A$ con $|y| = k$. El diablo elige u, v, w , donde $|u| = j$, $|v| = m > 0$ y $|w| = n$ tales que $y = uvw$ y $v \neq \epsilon$. No importa que elija el diablo, si uno toma $i = 2$ se gana inmediatamente:

$$\begin{aligned} xuv^2wz &= a^k b^j b^m b^m b^n \\ &= a^k b^{j+2m+n} \\ &= a^k b^{k+m} \notin A \end{aligned}$$

claramente el núm. de b 's es estrictamente mayor al núm. de a 's.

Ejercicios 5.1.1 ¿Cuál de los siguientes lenguajes es un conjunto regular? Justifique su respuesta.

1. $\{0^{2n} \mid n \geq 1\}$.
2. $\{0^m 1^n 0^{m+n} \mid n \geq 1, m \geq 1\}$.
3. $\{0^n \mid n \text{ es primo}\}$.
4. El conjunto de todas las cadenas sobre $\{0,1\}$ tales que no tienen 3 0's consecutivos.
5. El conjunto de todas las cadenas sobre $\{0,1\}$ tales que tengan igual número de 1's que de 0's.
6. $\{x \mid x \in (0+1)^*, y \ x = x^R\}$ donde x^R es x escrita al revés; por ejemplo $001^R = 100$.

5.2 Minimización de Estados de los AFD

En la realización de los ejercicios el lector habrá podido comprobar situaciones en las que se pueden simplificar los AFs, ya sea eliminando estados que no son accesibles desde el estado inicial ó colapsando estados que son equivalentes (fundamentalmente en el sentido que ante una misma cadena de entrada nos conducen al mismo estado). Por ejemplo, supóngase que

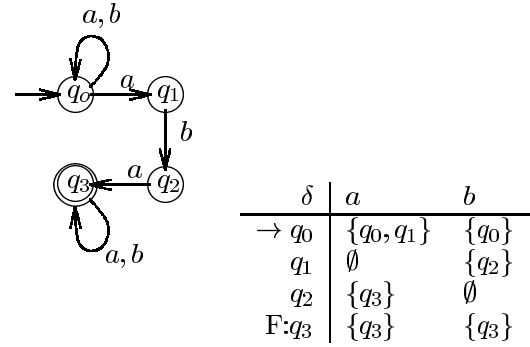
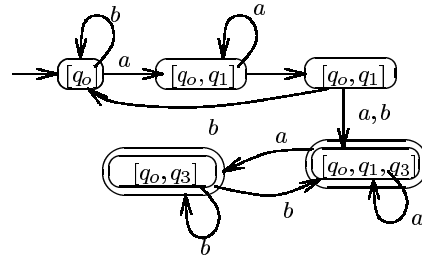


Figura 5.25:

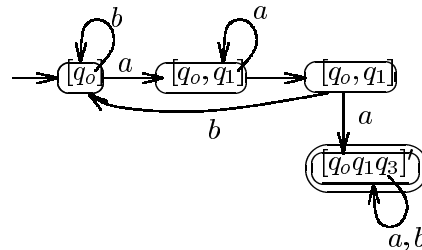
se aplica la transformación al siguiente AFND para obtener el correspondiente AFD (ver Figura 5.25)

Este AFND M acepta todas las cadenas que contengan la subcadena "aba". El AFD M' correspondiente es

δ'	a	b
$\rightarrow [q_0]$	$[q_0, q_1]$	$[q_0]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_2]$
$[q_0, q_2]$	$[q_0, q_1, q_3]$	$[q_0]$
$F: [q_0, q_1, q_3]$	$[q_0, q_1, q_3]$	$[q_0, q_3]$
$F: [q_0, q_3]$	$[q_0, q_1, q_3]$	$[q_0, q_3]$



Nótese que los estados finales se pueden colapsar en uno solo, dado que una vez que M' entre en un estado final ya no puede escaparse a otro estado no final. Por tanto, el anterior AFD es equivalente a



En este ejemplo, la equivalencia entre estados que se muestra es un tanto obvia, sin embargo no siempre es así. En esta sección discutiremos un método mecánico para encontrar los estados equivalentes de

cualquier AFD y colapsarlos, de tal suerte que tendremos como resultado que cualquier conjunto regular S tiene un AFD con *el menor número de estados posible* (salvo renombramiento de estados).

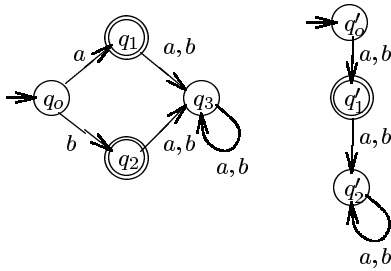
Dado un AFD $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ para un conjunto regular S (i.e., $L(M) = S$), el proceso de *minimización* consiste en dos etapas:

1. Descartar los estados inaccesibles desde el estado inicial, i.e., aquellos $q \in Q$ en los que no existe una cadena $w \in \Sigma^*$ tales que $\hat{\delta}(q_0, w) = q$.
2. Identificar los estados *equivalentes* y colapsarlos.

La eliminación de estados inaccesibles desde q_0 no cambia en absoluto el conjunto regular aceptado, tal eliminación puede hacerse fácilmente a partir de un recorrido primero en profundidad¹⁹ partiendo desde el q_0 en el grafo de transiciones del AF.

Para la segunda etapa, necesitamos precisar la noción de equivalencia entre estados y cómo se realizará el colapso de estados equivalentes. Veamos unos ejemplos antes de dar la definición formal.

Ejemplo 5.2.1 *Estos AFs aceptan el conjunto $\{a, b\}$. En el AF de la izquierda, q_1, q_2 son equivalentes pues alcanzan estados de aceptación dependiendo del símbolo de entrada, pero no es necesario que estén separados y por tanto pueden colapsarse obteniéndose el AF de la derecha.*



Ejemplo 5.2.2 *Ahora, los estados equivalentes en el AFD en la fig.5.26 en la parte izquierda son q_3, q_4 y q_5 y por tanto se pueden colapsar en uno solo, que se muestra en el AFD equivalente a la derecha. El conjunto aceptado por éstos AFD consiste de todas las cadenas sobre $\{a, b\}^*$ de longitud al menos 2.*

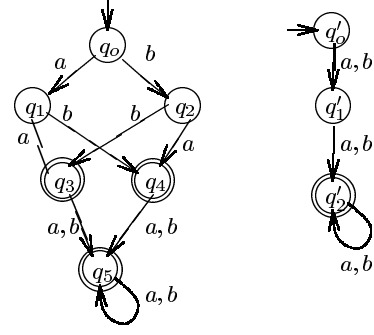


Figura 5.26: AFDs del ejemplo 5.2.2.

sin que cambie el lenguaje que acepta el AFD original), así como la descripción de un algoritmo eficiente para identificar los estados equivalentes y con ello realizar el colapso.

Nunca se debe permitir colapsar un estado de aceptación q_f con un estado no final (de rechazo) p dado que $\hat{\delta}(q_0, w) \in F$ y $p = \hat{\delta}(q_0, u) \notin F$, entonces evidentemente w debe aceptarse y u rechazarse aún después de colapsar estados. A su vez, si queremos colapsar p y q tenemos que también colapsar los estados a los cuales nos conducen, respectivamente $\delta(q, a)$ y $\delta(p, a)$, para alguna $a \in \Sigma$. Ello nos permitirá mantener el determinismo.

Estas dos observaciones permiten indicar que no podemos colapsar un estado p y q si ocurre que $\hat{\delta}(p, w) \in F$ y $\hat{\delta}(q, w) \notin F$ para alguna cadena w . Éste criterio es necesario y suficiente para decidir cuando una pareja de estados deba colapsarse: si existe una $w \in \Sigma^*$ tal que $\hat{\delta}(p, w) \in F$ y $\hat{\delta}(q, w) \notin F$ (o viceversa) entonces p y q no son equivalentes, pero si no existe tal w entonces p y q sí pueden colapsarse.

Definición 5.2.1 *Definimos la relación de equivalencia \approx en Q :*

$$p \approx q \stackrel{\text{def}}{\iff} \forall w \in \Sigma^*, (\hat{\delta}(p, w) \in F \iff \hat{\delta}(q, w) \in F)$$

Esta definición es la justificación formal del criterio de colapsamiento.

El lector puede verificar fácilmente que \approx es una relación de equivalencia (i.e., es simétrica, reflexiva y transitiva). Las clases de equivalencia disjuntas las denotamos por

$$[p] \stackrel{\text{def}}{=} \{q \mid q \approx p\}$$

Todo $p \in Q$ está contenido en exactamente una (y sólo una) clase de equivalencia $[p]$, y

$$p \approx q \iff [p] = [q]$$

Construcción del conjunto Cociente

Describiremos a continuación formalmente cuando dos estados pueden colapsarse de manera segura (i.e.,

¹⁹Este recorrido se discutió en 1.3.1, pág.6.

(las importantes son obviamente para cada estado accesible desde q_0). El número de clases de equivalencia siempre es finito si el conjunto es regular.

Definición 5.2.2 (Autómata Cociente) Dado un AFD $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ el Autómata Cociente asociado a M se denota como M/\approx y sus estados son las clases de equivalencia con respecto \approx

$$M/\approx \stackrel{def}{=} \langle Q', \Sigma, \delta', q'_0, F' \rangle$$

donde

$$\begin{aligned} Q' &\stackrel{def}{=} \{[p] \mid p \in Q\} \\ \delta'([p], a) &\stackrel{def}{=} [\delta(p, a)] \\ q'_0 &\stackrel{def}{=} [q_0] \\ F' &\stackrel{def}{=} \{[p] \mid p \in F\} \end{aligned}$$

Esta construcción ya no puede colapsarse más, si se vuelve a aplicar resulta el mismo AFD M/\approx .

Mediante los siguientes resultados justificaremos la correctitud de la construcción.

Lema 5.2.1 Si $p \approx q$ entonces $\delta(p, a) \approx \delta(q, a)$. Equivalentemente, si $[p] = [q]$ entonces $[\delta(p, a)] = [\delta(q, a)]$.

Demostración: Supongamos que $p \approx q$. Sea $a \in \Sigma$ y $w \in \Sigma^*$.

$$\begin{aligned} \hat{\delta}(\delta(p, a), w) \in F &\iff \hat{\delta}(p, aw) \in F \\ &\iff \hat{\delta}(q, aw) \in F \text{ dado que } p \approx q \\ &\iff \hat{\delta}(\delta(q, a), w) \in F. \end{aligned}$$

Lema 5.2.2 $p \in F \iff [p] \in F'$.

Demostración: La dirección \Rightarrow es inmediata por la definición de F' . Para \Leftarrow necesitamos demostrar que si $p \approx q$ y $p \in F$ entonces $q \in F$, i.e., cada clase de equivalencia es un subconjunto de F ó es disjunta de F . Esto se infiere inmediatamente tomando $w = \epsilon$ en la definición de $p \approx q$.

Lema 5.2.3 Para toda $w \in \Sigma^*$, $\hat{\delta}'([p], w) = [\hat{\delta}(p, w)]$.

Demostración: aplicaremos inducción en $|w|$.

• Caso base $w = \epsilon$:

$$\begin{aligned} \hat{\delta}'([p], \epsilon) &= [p] && \text{por def. de } \hat{\delta}' \\ &= [\hat{\delta}(p, \epsilon)] && \text{por def. de } \hat{\delta}. \end{aligned}$$

• Paso inductivo: asumiremos que $\hat{\delta}'([p], w) = [\hat{\delta}(p, w)]$ y sea $a \in \Sigma$

$$\begin{aligned} \hat{\delta}'([p], wa) &= \delta'(\hat{\delta}'([p], w), a) && \text{por def. de } \hat{\delta}' \\ &= \delta'([\hat{\delta}(p, w)], a) && \text{HI} \\ &= [\delta(\hat{\delta}(p, w), a)] && \text{por def. de } \delta' \\ &= [\hat{\delta}(p, wa)] && \text{por def. de } \hat{\delta}. \end{aligned}$$

Teorema 5.2.1 $(L(M/\approx) = L(M))$

Demostración: para $w \in \Sigma^*$,

$$\begin{aligned} w \in L(M/\approx) &\iff \hat{\delta}'(q'_0, w) \in F' && \text{def. aceptación} \\ &\iff \hat{\delta}'([q_0], w) \in F' && \text{def. } q'_0 \\ &\iff [\hat{\delta}(q_0, w)] \in F' && \text{lema 5.2.3} \\ &\iff \hat{\delta}(q_0, w) \in F && \text{lema 5.2.2} \\ &\iff w \in L(M) && \text{def. aceptación.} \end{aligned}$$

Un Algoritmo de Minimización

Presentamos a continuación un algoritmo para calcular la relación \approx para un AFD M sin estados inaccesibles. El algoritmo marcará parejas (no ordenadas) de estados $\{p, q\}$ en cuanto se descubra que no son equivalentes.

1. Escribir una tabla con las parejas $\{p, q\}$, inicialmente no marcadas.
2. Marcar $\{p, q\}$ si $p \in F$ y $q \notin F$ o viceversa.
3. Repetir lo siguiente hasta que no ocurran más cambios: si existe una pareja no marcada $\{p, q\}$ tal que $\{\delta(p, a), \delta(q, a)\}$ ya está marcada para alguna $a \in \Sigma$, entonces marcar a $\{p, q\}$.
4. Al final, $p \approx q$ si y sólo si $\{p, q\}$ no está marcada.

Algoritmo 2 Minimización de estados de AFD.

Entrada: AFD $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ sin estados inaccesibles.

Salida: \approx de Q .

for $p \in F$ y $q \in Q - F$ **do**

marcar $\{p, q\}$;

end for

for cada $\{p, q\} \in F \times F$ distintos **do**

{*ó $\{p, q\} \in (Q - F) \times (Q - F)$ *}

if $\{\delta(p, a), \delta(q, a)\}$ está marcada **then**

marcar $\{p, q\}$;

marcar recursivamente a todas las parejas en la lista para $\{p, q\}$ y en las listas de las otras parejas así marcadas;

else

{*ninguna $\{\delta(p, a), \delta(q, a)\}$ está marcada*}

for cada $a \in \Sigma$ **do**

poner $\{p, q\}$ en la lista para $\{\delta(p, a), \delta(q, a)\}$
a menos que $\delta(p, a) = \delta(q, a)$

end for

end if

end for

Hay que notar lo siguiente:

- El algoritmo se detendrá tras un pasada a la tabla completa en que no hayan ocurrido más cambios.
- El algoritmo se detiene tras un número finito de pasos, dado que sólo hay $\frac{n!}{k!(n-k)!}$ subconjuntos de tamaño k dentro de un conjunto de tamaño n y que coincide con el número de estados que pueden hacerse, marcando al menos uno en cada pasada.

Mostraremos paso a paso el funcionamiento del algoritmo para el ejemplo 5.2.2: la tabla asociada a la función de transición δ es

δ	a	b
$\rightarrow q_0$	q_1	q_2
q_1	q_3	q_4
q_2	q_4	q_3
F: q_3	q_5	q_5
F: q_4	q_5	q_5
F: q_5	q_5	q_5

1. Inicialmente no hay marcas:

q_1	.				
q_2	.				
q_3	.	.	.		
q_4	
q_5
	q_0	q_1	q_2	q_3	q_4

2. Llenamos la tabla para las parejas de estados tales que $\{p, q\}$ si $p \in F$ y $q \notin F$:

q_1	.				
q_2	.				
q_3	X	X	X		
q_4	X	X	X	.	
q_5	X	X	X	.	.
	q_0	q_1	q_2	q_3	q_4

3. En orden, vemos que la pareja (q_0, q_1) no está marcada y que ante la entrada a , $(\delta(q_0, a) = q_1, \delta(q_1, a) = q_3)$ la cual ya está marcada, por lo que tenemos que marcar a (q_0, q_1) . De manera similar, (q_0, q_2) ante a alcanzan la pareja (q_2, q_4) que también está marcada, por lo tanto marcamos (q_0, q_2)

q_1	⊗				
q_2	⊗				
q_3	X	X	X		
q_4	X	X	X	.	
q_5	X	X	X	.	.
	q_0	q_1	q_2	q_3	q_4

4. Al continuar aplicando el mismo criterio, ya ninguna otra pareja puede ser marcada, por lo

que las parejas no marcadas son estados equivalentes

$$q_1 \approx q_2, q_3 \approx q_4, q_3 \approx q_5, q_4 \approx q_5$$

y por tanto, las clases de equivalencia de estados son:

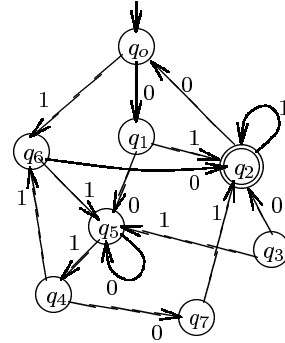
$$q'_0 : [q_0], q'_1 : [q_1, q_2], q'_2 : [q_3, q_4, q_5]$$

Quedando como resultado el AFD de la parte derecha de la fig.5.26, pág. 38.

Ejemplo 5.2.3 Sea el AFD $M = \langle Q, \Sigma, \delta, q_0, F \rangle$

δ	0	1
$\rightarrow q_0$	q_1	q_5
q_1	q_6	q_2
F: q_2	q_0	q_2
q_3	q_3	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

Podemos apreciar inmediatamente, a partir del grafo de transiciones de M que el estado q_3 no es alcanzable desde q_0 , por lo tanto lo debemos eliminar.



Aplicamos el algoritmo anterior, partiendo de la tabla en la cual se han marcado las parejas de estados no finales con el estado final q_2 :

q_1						
q_2	X	X				
q_4	.	.	X			
q_5	.	.	X	.		
q_6	.	.	X	.	.	
q_7	.	.	X	.	.	.
	q_0	q_1	q_2	q_4	q_5	q_6

Podemos descubrir estados que deban marcarse de manera más rápida si para cada pareja de estados marcados identificamos cómo podemos llegar a esos estados leyendo el mismo símbolo (en dirección inversa a como indica el algoritmo); i.e., (q_0, q_2) está

marcado, y podemos llegar a esa pareja a partir de (q_2, q_5) (debido a que $\delta(q_2, 0) = a, \delta(q_5, 0) = c$, por lo que debemos marcar la pareja (q_2, q_5) (que ya lo está, por cierto). Enriqueceremos la notación de la siguiente forma: $(q_2, q_5) \xrightarrow{0} (q_0, q_2)$.

De esa manera:

1. $(q_1, q_2) \xleftarrow{0} (q_0, q_5)$ marcar.
2. $(q_2, q_4) \xleftarrow{1} (q_1, q_6)$ marcar.
3. $(q_5, q_2) \xleftarrow{1} (q_0, q_1)$ marcar.
4. $(q_5, q_2) \xleftarrow{1} (q_7, q_4)$ marcar.
5. $(q_5, q_2) \xleftarrow{1} (q_1, q_4)$ marcar.
6. $(q_6, q_2) \xleftarrow{1} (q_2, q_5)$ marcar.
7. $(q_6, q_2) \xleftarrow{0} (q_7, q_5)$ marcar.
8. $(q_7, q_2) \xleftarrow{0} (q_4, q_5)$ marcar.
9. $(q_4, q_5) \xleftarrow{1} (q_0, q_6)$ marcar.
10. $(q_4, q_5) \xleftarrow{1} (q_4, q_6)$ marcar.
11. $(q_1, q_6) \xleftarrow{0} (q_0, q_7)$ marcar.
12. $(q_2, q_6) \xleftarrow{0} (q_5, q_6)$ marcar.
13. $(q_4, q_5) \xleftarrow{1} (q_6, q_7)$ marcar.

En la tabla reflejamos los cambios indicando cada marca con el anterior subíndice:

q_1	X_3					
q_2	X	X				
q_4	.	X_5	X			
q_5	X_1	X_6	X	X_8		
q_6	X_9	X_2	X	X_{10}	X_{12}	
q_7	X_{11}	.	X	X_4	X_7	X_{13}
	q_0	q_1	q_2	q_4	q_5	q_6

De la tabla final, podemos identificar las clases de equivalencia entre estados

$$[q_0, q_4], [q_1, q_7], [q_5], [q_6], [q_2]$$

De tal manera que el AFD M/\approx correspondiente es:

δ'	0	1
$[q_0, q_4]$	$[q_1, q_7]$	$[q_5]$
$[q_1, q_7]$	$[q_6]$	$[q_2]$
$[q_5]$	$[q_2]$	$[q_6]$
$[q_6]$	$[q_6]$	$[q_0, q_4]$
$F: [q_2]$	$[q_0, q_4]$	$[q_2]$

Ejercicios 5.2.1 Encuentre el AFD cociente para

δ	$\rightarrow q_0$	q_1	q_2	$F: q_3$	q_4	q_5
0	q_1	q_0	q_3	q_3	q_3	q_3
1	q_0	q_2	q_1	q_0	q_5	q_4

5.3 Propiedades de Cerradura

Existen muchas operaciones que al aplicarlas a conjuntos regulares el resultado también es un conjunto regular: como ejemplos tenemos a la unión, concatenación y cerradura Kleene. Decimos entonces que los conjuntos regulares son cerrados bajo concatenación, unión y cerradura Kleene. Decimos que una clase de lenguajes \mathcal{L} tiene cierta *propiedad de cerradura* P_c si dicha clase es *cerrada* bajo la operación P_c . De especial interés son las operaciones de cerradura *efectivas*, i.e., aquellas que dadas descripciones para lenguajes R_1, R_2, \dots, R_k dentro de la clase ($R_i \in \mathcal{L}$), existe un algoritmo para construir una representación del lenguaje R' que resulta al aplicar la operación de cerradura sobre los lenguajes R_i ($1 \leq i \leq k$). Un ejemplo de tales operaciones efectivas es la construcción de unión entre dos lenguajes regulares que se presentó anteriormente (cada lenguaje denotado como ER y el resultado es también una ER). Por lo tanto, la clase de conjuntos regulares es cerrada efectivamente bajo la unión.

También, nótese que las equivalencias que se mostraron entre modelos de autómatas ($AFD \longleftrightarrow AFND \longleftrightarrow AFND-\epsilon$) fueron efectivas ya que se describieron algoritmos que permiten traducir de un modelo (representación) a otro.

Ya se han presentado las propiedades de cerradura dadas a partir de las ER's, como también algunas que involucran operaciones booleanas (complemento, unión, intersección y producto). Discutiremos a continuación algunas propiedades de cerradura especiales.

5.3.1 Substituciones y Homomorfismos

La idea intuitiva de la substitución es la siguiente: para cada $a \in \Sigma$ de algún conjunto regular R asociemos el conjunto regular R_a . Reemplazaremos cada frase $a_1 a_2 \dots a_n \in R$ por el conjunto de palabras de la forma $w_1 w_2 \dots w_n$, donde cada w_i es una palabra arbitraria tomada de R_{a_i} . No debe sorprendernos que el conjunto de palabras resultante así obtenido también es un conjunto regular.

Definición 5.3.1 (Substitución) Una *substitución* $h: \Sigma \rightarrow \Gamma^*$ es una correspondencia entre un alfabeto Σ y subconjuntos de Γ^* , donde Γ es otro alfabeto. En otras palabras, h asocia a un lenguaje a cada símbolo $a \in \Sigma$.

h se extiende a cadenas sobre Σ^* de la siguiente manera:

$$\begin{aligned} h(\epsilon) &= \epsilon \\ h(wa) &= h(w)h(a) \end{aligned}$$

A su vez, puede extenderse a lenguajes mediante

$$h(L) = \bigcup_{w \in L} h(w)$$

Ejemplo 5.3.1 Sea $h(0) = \mathbf{a}$ y $h(1) = \mathbf{b}^*$. $h(0)$ denota al lenguaje $\{a\}$ y $h(1)$ al lenguaje de todas las cadenas que se pueden formar con b .

$h(010)$ corresponde a $\mathbf{ab}^*\mathbf{a}$. Si L es el lenguaje $0^*(0+1)1^*$ entonces $h(L) = \mathbf{a}^*(\mathbf{a} + \mathbf{b}^*)(\mathbf{b}^*)^*$ lo cual se puede simplificar a $\mathbf{a}^*\mathbf{b}^*$.

Teorema 5.3.1 La clase de conjuntos regulares es cerrada bajo substitución.

Demostración: sea R un conjunto regular definido sobre el alfabeto Σ ($R \subseteq \Sigma^*$). Para cada $a \in \Sigma$ sea R_a un conjunto regular ($R_a \subseteq \Gamma^*$). Sea $h : \Sigma \rightarrow \Gamma^*$ la substitución definida por $h(a) = R_a$. Se procede:

1. Seleccionar ER's que describan a R y a cada R_a (para cada $a \in \Sigma$).
2. Remplazar cada ocurrencia de \mathbf{a} en la ER para R por la expresión regular para R_a .
3. Para demostrar que la expresión regular resultante denota a $h(R)$, obsérvese que la substitución de una unión, producto ó cerradura es la unión, producto ó cerradura de las substituciones:

$$\begin{aligned} h(L_1 \cup L_2) &= h(L_1) \cup h(L_2) \\ h(L_1 \times L_2) &= h(L_1) \times h(L_2) \\ h(L_1^*) &= (h(L_1))^* \end{aligned}$$

Aplique inducción (estructural) respecto del número de operadores de la ER R para completar la prueba.

Una substitución de especial interés es el homomorfismo.

Definición 5.3.2 (Homomorfismo) Un homomorfismo h es una substitución tal que $h(a)$ contiene a una sola cadena para cada a . Se considerará que $h(a)$ representa una cadena más que el conjunto que contiene a esa cadena.

En general, cualquier correspondencia $h : \Sigma \rightarrow \Gamma^*$ se puede extender por inducción y de manera única a un homomorfismo definido sobre todo Σ^* : para especificar un homomorfismo sólo necesitamos indicar que valores toma sobre elementos de Σ .

Si $A \subseteq \Sigma^*$, se define la imagen directa por

$$h(A) \stackrel{\text{def}}{=} \{h(x) \mid x \in A\} \subseteq \Gamma^*$$

Dado un $B \subseteq \Gamma^*$, se define la imagen inversa

$$h^{-1}(B) \stackrel{\text{def}}{=} \{x \mid h(x) \in B\} \subseteq \Sigma^*$$

Como ejemplo tenemos lo siguiente:

Ejemplo 5.3.2 Sea $h(0) = aa$ y $h(1) = aba$. Entonces $h(010) = aaabaaa$. Si $L_1 = (01)^*$ entonces $h(L_1) = (aaaba)^*$.

Sea $L_2 = (\mathbf{ab} + \mathbf{ba})^*\mathbf{a}$. ¿Quién es $h^{-1}(L_2)$? Veamos que ninguna cadena de L_2 que comience con b puede tener preimagen puesto que $h(0)$ y $h(1)$ siempre comienzan con a . Entonces, si $h^{-1}(w) \neq \emptyset$ y $w \in L_2$ entonces w debe ser de la forma au , para alguna u . Si $u = \epsilon$, entonces $h^{-1}(w)$ sería vacío; por tanto w debe ser de la forma abu' , para alguna $u' \in L_2$. Concluimos que toda palabra en $h^{-1}(w)$ debe comenzar con 1 puesto que $h(1) = aba$, entonces $u' = a$, $w = aba$ y $h^{-1}(w) = \{1\}$, la cual es la única; por lo tanto $h^{-1}(L_2) = \{1\}$.

Nótese que $h(h^{-1}(L_2)) = \{aba\} \neq L_2$.

Ejercicios 5.3.1 Demostrar que $h(h^{-1}(L)) \subseteq L$ y $L \subseteq h^{-1}(h(L))$ para cualquier lenguaje L .

La clase de los conjuntos regulares es cerrada bajo imágenes directa e inversa de homomorfismos.

Teorema 5.3.2 Sea $h : \Sigma^* \rightarrow \Gamma^*$ un homomorfismo. Si $B \subseteq \Gamma^*$ es regular, entonces su imagen inversa $h^{-1}(B)$ también es regular.

Demostración: La cerradura bajo homomorfismo se sigue inmediatamente de la cerradura bajo substitución, ya que todo homomorfismo es una substitución en la cual $h(a)$ tiene un miembro.

Para demostrar cerradura bajo imagen inversa, sea $M = \langle Q, \Gamma, \delta, q_0, F \rangle$ un AFD tal que $L(M) = B$. Se describe la construcción de un nuevo AFD $M' = \langle Q, \Sigma, \delta', q_0, F \rangle$ el cual acepta $h^{-1}(B)$ al leer $a \in \Gamma$ y simular a M en $h(a)$; i.e., se define δ' por

$$\delta'(q, a) \stackrel{\text{def}}{=} \hat{\delta}(q, h(a))$$

(Utilizamos $\hat{\delta}$ pues $h(a)$ podría ser toda una cadena, incluso ϵ , en vez de sólo un símbolo).

Aplicamos inducción en $|x|$ (para toda $x \in \Sigma^*$) para mostrar que

$$\hat{\delta}'(q, x) = \hat{\delta}(q, h(x))$$

- Para el caso base, $x = \epsilon$, entonces

$$\hat{\delta}'(q, \epsilon) = q = \hat{\delta}(q, \epsilon) = \hat{\delta}'(q, h(\epsilon))$$

- Para el paso inductivo, asumimos que $\hat{\delta}'(q, x) = \hat{\delta}(q, h(x))$. Así

$$\begin{aligned} \hat{\delta}'(q, xa) &= \delta'(\hat{\delta}'(q, x)) && \text{def. } \hat{\delta} \\ &= \delta'(\hat{\delta}(q, h(x)), a) && \text{HI} \\ &= \hat{\delta}(\hat{\delta}(q, h(x)), h(a)) && \text{def. } \delta' \\ &= \hat{\delta}(\hat{\delta}(q, h(x)), h(a)) && \text{propiedad} \\ &= \hat{\delta}(q, h(xa)) && \text{propiedad} \end{aligned}$$

Finalmente,

$$\begin{aligned} x \in L(M') &\iff \hat{\delta}(q_0, x) \in F \\ &\iff \hat{\delta}(q_0, h(x)) \in F \\ &\iff h(x) \in L(M) \\ &\iff x \in h^{-1}(L(M)) \end{aligned}$$

La utilización de la *Cerradura- ϵ* en el capítulo anterior fué un ejemplo de homomorfismo. Los homomorfismos también son útiles para simplificar pruebas, estableciendo que si un conjunto A no es regular y podemos construir un homomorfismo h de A hacia B , podemos demostrar que B tampoco es regular.

Presentaremos a continuación un ejemplo de propiedad de cerradura que no es efectiva: no hay un algoritmo en general que permita construirla.

Definición 5.3.3 Se define el cociente de los lenguajes L_1 y L_2 como

$$L_1/L_2 = \{x \mid \text{existe } y \in L_2 \text{ tal que } xy \in L_1\}$$

Ejemplo 5.3.3 Sean $L_1 = 0^*10^*$ y $L_2 = 10^*1$. Dado que cada cadena $y \in L_2$ tiene dos 1's y cada cadena $xy \in L_1$ solo tiene un 1, no hay x tal que $xy \in L_1$ con $y \in L_2$; por lo tanto $L_1/L_2 = \emptyset$.

Sea $L_3 = 0^*1$, claramente $L_1/L_3 = 0^*$ puesto que las cadenas de la forma $xy \in L_1$ donde $x = 0^*$ y $y = 1$: las palabras tanto en L_1 como en L_3 solo tienen un 1, por lo que no es posible que otras cadenas, salvo 0^* estén en el cociente L_1/L_3 .

Teorema 5.3.3 La clase de los conjuntos regulares es cerrada bajo la operación cociente sobre conjuntos arbitrarios: si R es regular y A es cualquier conjunto (regular o no), entonces R/A es regular.

Demostración: sea $M = \langle Q, \Gamma, \delta, q_0, F \rangle$ un AF que acepta R (i.e., $L(M) = R$). Podemos construir un AF $M' = \langle Q, \Gamma, \delta, q_0, F' \rangle$ que acepte el conjunto cociente R/A , para lo cual, se comporta como M excepto que los estados de aceptación de M' son todos los estados q de M tales que existe una $y \in A$ para los cuales $\delta(q, y) \in F$. Entonces,

$$\delta(q_0, w) \in F' \iff \exists y \in A, \delta(q_0, wy) \in F$$

Debe apreciarse inmediatamente lo siguiente: dado que el conjunto A es arbitrario, podría no existir algún algoritmo para determinar si existe alguna $y \in A$ para la cual $\delta(q, y) \in F$. Por tanto, en general no podemos decir qué subconjunto de Q se deba elegir para formar F' .

Si A también es regular, por los resultados de la minimización de autómatas siempre es posible construir el cociente.

5.3.2 Algoritmos de Decisión para Conjuntos Regulares

Ante un AF (o una ER) siempre nos hemos preguntado primero cuál es el lenguaje que acepta (denota). Como consecuencia, cabe preguntarse si tal lenguaje es finito, infinito ó no tiene elementos (vacío). A su vez, también es útil saber cuándo dos conjuntos regulares son equivalentes (tienen los mismos elementos). De vital importancia es saber si podemos contestar a tales preguntas de *manera efectiva*: plantear algoritmos que resuelvan tales problemas de decisión.

Infinitud, finitud y vacuidad

Teorema 5.3.4 El conjunto de palabras aceptadas por un AF M con n estados es:

- No vacío si y sólo si M acepta una palabra w tal que $|w| < n$.
- Infinito si y sólo si M acepta una palabra w tal que $n \leq |w| < 2n$.

En consecuencia, existe un algoritmo para decidir si un AF M acepta cero, un número finito ó un número infinito de palabras.

Demostración: se plantean a continuación, en cada caso, los algoritmos correspondientes. Para el primer caso, se puede verificar fácilmente cuándo un AF acepta \emptyset eliminando de su digrafo todos los estados inaccesibles ante cualquier entrada partiendo del estado inicial. Si queda al menos un estado final, el lenguaje correspondiente es no vacío. A su vez, sin cambiar el lenguaje aceptado, se pueden eliminar todos los estados que no son finales y aquellos desde los cuales ningún estado de aceptación se pueda alcanzar. Por otra parte, el AF acepta un lenguaje infinito si y sólo si en el digrafo resultante existe un ciclo.

1. La parte “si” es inmediata. Supóngase que M acepta un conjunto $A \neq \emptyset$. Sea w una cadena tan corta como cualquier otra cadena aceptada. Por el Lema de Sondeo, $|w| < n$ si w fuese la más corta, y $|w| \geq n$ para cuando no es la más corta, en cuyo caso se puede descomponer en uvx donde ux es la cadena más corta del lenguaje. Evidentemente, el lenguaje es no vacío.
2. En la dirección *sólo si*, si $w \in L(M)$ y $n \leq |w| < 2n$, entonces por el Lema de Sondeo $L(M)$ es infinito.

De manera complementaria, si $L(M)$ es infinito entonces existe una cadena $w \in L(M)$ con $|w| \geq n$. Si $|w| < 2n$ la prueba se termina. En otro

caso, si no hay palabra alguna cuya longitud está entre n y $2n - 1$ entonces sea w tal que $|w| \geq 2n$ pero tan corta como cualquier otra palabra de $L(M)$. De nuevo, por el Lema del Sondeo, se puede descomponer a w en xyz con $1 \leq |y| \leq n$ y $xz \in L(M)$. Por tanto, no es válida la suposición de que w fuese la cadena más corta de longitud mayor ó igual a $2n$, o que $n \leq |xz| \leq 2n - 1$. Por tanto, el AF acepta alguna palabra de longitud l para $n \leq l \leq 2n$.

En el primer inciso, el algoritmo para decidir vacuidad de $L(M)$ es “verificar si una palabra de longitud a lo más n está en $L(M)$ ”, el cual evidentemente es un procedimiento que termina. Por su parte, en el inciso 2 corresponde el siguiente algoritmo: “verificar si alguna palabra de longitud entre n y $2n - 1$ está en $L(M)$ ” el cual también es un procedimiento efectivo.

Equivalencia

A continuación se muestra que existe un algoritmo para decidir si dos AFs son equivalentes.

Teorema 5.3.5 *Existe un algoritmo para determinar si dos AF aceptan el mismo conjunto.*

Demostración: Sean M_1, M_2 AF's que aceptan L_1, L_2 respectivamente. Por las propiedades de cerradura (booleanas, unión, intersección, complemento, etc.) el conjunto $(L_1 \cap \sim L_2) \cup (\sim L_1 \cap L_2)$ es aceptado por algún AF M_3 : se puede verificar que M_3 acepta una palabra si y sólo si $L_1 \neq L_2$; por el teorema anterior, existe un algoritmo para determinar si $L_1 = L_2$.

Ejercicios 5.3.2

1. ¿Es la clase de los conjuntos regulares cerrada bajo unión infinita? Justifique.
2. ¿Cuál es la relación entre la clase de los conjuntos regulares y la clase más pequeña de lenguajes cerrada bajo unión, intersección y complemento que contiene a todos los conjuntos finitos?
3. Sea L cualquier subconjunto de 0^* . Demuestre que L^* es regular.
4. Proponga una construcción entre autómatas finitos (en el mismo sentido a las que se discutieron para concatenación, unión, cerradura Kleene, etc.) para demostrar que la clase de los conjuntos regulares es cerrada bajo substitución.

5. ¿Es la clase de los conjuntos regulares cerrada bajo substitución inversa?
6. Sea h el homomorfismo donde $h(a) = 01$ y $h(b) = 0$. Encontrar
 - (a) $h^{-1}(L_1)$ para $L_1 = (10 + 1)^*$.
 - (b) $h(L_2)$ para $L_2 = (a + b)^*$.
 - (c) L_1/L_2 y L_2/L_1 .

Autómatas de Pila y Lenguajes Libres de Contexto

Lenguajes Libres de Contexto

Ya hemos podido apreciar que una de las limitaciones de los AFs es que no pueden reconocer el lenguaje $\{0^n 1^n\}$ debido a que no se puede registrar para todo n con un número finito de estados. Un lenguaje *isomorfo* al anterior es el lenguaje de las expresiones con paréntesis balanceados, el cual se puede extender para describir expresiones anidadas de uso muy amplio en los lenguajes de programación; por ejemplo, en los bloques **begin** - **end** de Pascal, las expresiones S en lisp, etc. Por lo tanto, aún cuando los lenguajes regulares permiten describir identificadores, palabras claves y patrones de uso común en computación necesitamos un modelo más poderoso para describir las estructuras sintácticas de los lenguajes de programación. Las *Gramáticas Libres de Contexto (GLC)* y los lenguajes que generan (*Lenguajes Libres de Contexto (LLC)*) nos permiten definir la sintaxis de los lenguajes de programación como también formalizar las nociones fundamentales de la teoría de compilación como son el *parsing*, la traducción conducida por sintaxis, etc.

La forma típica de las GLC es la notación *Backus-Naur* (mencionada brevemente en el capítulo 2) la cual consiste de un conjunto finito de reglas ó cláusulas descrita en dos partes, separadas por ‘::=’. En la parte izquierda aparece una *variable* (*símbolo no terminal*) que representa un lenguaje; las variables se definen *recursivamente* en términos de otras variables y de *átomos* ó símbolos primitivos denominados *terminales*. La parte derecha es justamente la definición de un lenguaje, el cual contiene cadenas que se forman concatenando las cadenas del lenguaje de otras variables posiblemente junto con algunos terminales.

Por ejemplo, la siguiente es un fragmento de una definición de la sintaxis²⁰ de un lenguaje tipo Pascal:

```

<stmt>      ::= <ifstmt> | <whilestmt>
              | <beginstmt>
<ifstmt>     ::= if <exprbool> then <stmt>
              else <stmt>
<whilestmt>  ::= while <exprbool> do <stmt>
<beginstmt>  ::= begin <stmtlist> end
<stmtlist>   ::= <stmt> | <stmt> ; <stmtlist>

```

²⁰En notación Backus-Naur.

La primer regla indica que un enunciado puede ser un enunciado condicional (*if*) o un ciclo (*while*) ó un bloque. A su vez, un enunciado condicional comienza con la palabra reservada **if** a continuación debe estar una expresión booleana seguida por la palabra reservada **do** y en seguida un enunciado. Efectivamente, las palabras reservadas son expresiones básicas (átomos o terminales). Nótese cómo las reglas son recursivas, por ejemplo, la regla para una lista de enunciados indica que es ya sea un solo enunciado ó un enunciado seguido de ‘;’ y a continuación una lista de enunciados.

Los modelos mecánicos que corresponden a las GLC son los Autómatas de Pila (*Push Down Automata*) que son como los AF excepto que tienen adicionalmente una pila para almacenamiento. Recuérdese que una pila permite solamente dos operaciones: *push* (empilar un elemento en el tope) y *pop* (desempilar el elemento situado en el tope), la cual le permite registrar información en forma LIFO (primero en entrar, último en salir). Intuitivamente, para que pueda reconocer el lenguaje $\{0^n 1^n\}$ un AP puede empilar cada 0 que lea y cuando comience a leer 1’s desempilar los 0 que ya tiene en la pila. Si al final de la cadena de entrada se vació la pila entonces acepta la cadena, de lo contrario la rechaza (ver Figura 6.27).

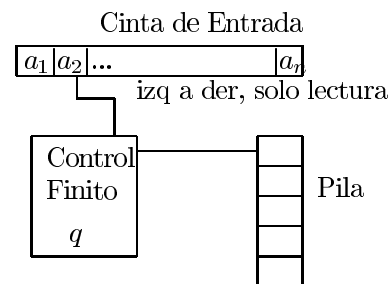


Figura 6.27: Esquema de un Autómata de Pila (AP).

Definiremos formalmente a tales máquinas y demostraremos que la clase de lenguajes aceptados coinciden con la clase de lenguajes libres de contexto. Aún cuando la clase de los lenguajes libres de contexto contiene propiamente a la clase de lenguajes re-

se puede sintetizar mediante la expresión

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$$

donde el operador $|$ significa *alternativa*.

Definición 6.1.4 (GLC) Una Gramática Libre de Contexto (GLC) es una descripción estructural precisa de un lenguaje. Formalmente es una tupla $G = \langle Vn, Vt, P, S \rangle$, donde:

- V_n es el conjunto finito de **símbolos no terminales**
- V_t es el conjunto finito de **símbolos terminales** ($V_n \cap V_t = \emptyset$, i.e. son disjuntos)
- P es el conjunto finito de **producciones** que se pueden ver como relaciones definidas en $V_n \times (V_n \cup V_t^*)$
- S es el **símbolo inicial** de la gramática.

Las producciones tienen la forma $A \rightarrow \alpha$ donde $A \in Vn$ y α es una expresión ya sea compuesta tanto por símbolos no terminales como terminales ($\alpha \in (Vt \cup Vn)^*$) ó también puede ser la expresión nula ϵ .

Una producción puede verse como una **regla de reescritura** que indica cómo reemplazar símbolos no terminales por la expresión correspondiente en la parte derecha de la producción. Así, partiendo de algún $A \in V_n$, se pueden aplicar las reglas de P hasta alcanzar eventualmente a una expresión compuesta únicamente por símbolos terminales. A ésta expresión se le denomina *frase ó lexema*, mientras que a las expresiones formadas por símbolos terminales y no terminales se les denomina *formas de frase*.

En éste capítulo y el siguiente seguiremos las convenciones siguientes:

1. Letras romanas mayúsculas (A, B, C, D, E, S) representan símbolos no terminales (*variables*).
2. Letras romanas minúsculas (a, b, c, d, e), dígitos o expresiones encerradas entre comillas ($'$, $'$) representan símbolos terminales.
3. X, Y, Z, W representan terminales o no terminales (metavARIABLES).
4. u, v, w, x, y, z representan cadenas de terminales (frases).
5. Letras griegas minúsculas α, β, γ representan formas de frases.

Gracias a estas convenciones, es común describir una gramática enlistando únicamente sus producciones: si se tiene la gramática

$$A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k$$

Definición 6.1.5 (Derivaciones) Para cualquier GLC G , si $A \rightarrow \beta$ en P y $\alpha, \gamma \in (Vn \cup Vt)^*$, entonces $\alpha A \gamma \Rightarrow \alpha \beta \gamma$, lo cual se lee “la forma de frase $\alpha A \gamma$ **deriva directamente** a $\alpha \beta \gamma$ ”. Si se tiene que para $\alpha_1, \alpha_2, \dots, \alpha_k \in (Vn \cup Vt)^*, (k \geq 1)$, $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_k$ se dice que “ α_1 **deriva a** α_k ”, lo cual se escribe $\alpha_1 \xRightarrow{*} \alpha_k$.

La relación \Rightarrow entre formas de frase tiene las siguientes propiedades:

1. $\alpha \overset{*}{\Rightarrow} \alpha$ para cualquier α ,
2. Si $\alpha \overset{*}{\Rightarrow} \beta$ y $\beta \overset{*}{\Rightarrow} \gamma$ entonces $\alpha \overset{*}{\Rightarrow} \gamma$.

En otras palabras, $\overset{*}{\Rightarrow}$ es la cerradura reflexiva y transitiva de \Rightarrow .

Definición 6.1.6 (Lenguaje Libre de Contexto)
Dada una GLC G , se define al lenguaje generado por G como el conjunto de frases derivadas por G partiendo del símbolo inicial

$$L(G) = \{w \mid w \in Vt^* \text{ y } S \xRightarrow{*} w\}$$

Un lenguaje libre de contexto es aquel generado por alguna GLC.

Ejemplo 6.1.2 Sea $G_0 = \langle Vn_0, Vt_0, P_0, S \rangle$ definido por:

$$\begin{aligned} Vn_0 &= \{exp, dig\} \\ Vt_0 &= \{'0', '1', '2', \dots, '9', '+', '-'\} \\ S &= exp \\ P_0 &= \begin{cases} exp \rightarrow & \begin{array}{l} exp + dig \\ exp - dig \end{array} \\ dig \rightarrow & \begin{array}{l} dig \\ 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array} \end{cases} \end{aligned} \quad \begin{array}{l} (1) \\ (2) \\ (3) \\ (4) \end{array}$$

A continuación se muestra la derivación de la frase “ $4-3+1$ ”,

$$\begin{aligned} exp &\Rightarrow_1 exp + dig \\ &\Rightarrow_2 exp - dig + dig \\ &\Rightarrow_3 dig - dig + dig \\ &\Rightarrow_4 4 - dig + dig \\ &\Rightarrow_4 4 - 3 + dig \\ &\Rightarrow_2 4 - 3 + 1 \end{aligned}$$

(los subíndices de cada ‘ \Rightarrow ’ indican el número de la producción utilizada en ese paso).

Por tanto “4-3+1” pertenece al lenguaje generado por G_0 .

En cada paso de una derivación se pueden aplicar las reglas (producciones) en cualquier orden, por lo cual se debe elegir:

- Se puede elegir siempre el no terminal *más a la izquierda* en cada paso, la derivación así obtenida recibe el nombre de *derivación más a la izquierda*: $wA\gamma \Rightarrow w\beta\gamma$, para una $A \rightarrow \beta \in P$. Si α deriva a β mediante una derivación por la izquierda se escribe $\alpha \Rightarrow_{mi} \beta$.

Por ejemplo, la derivación mostrada anteriormente es una derivación más por la izquierda de “ $4 - 3 + 1$ ”, a continuación se muestra una derivación más por la derecha para la misma expresión

Cuando, para la misma expresión es posible encontrar más de una derivación más por la izquierda (o por la derecha), se dice que la gramática correspondiente es **ambigua**. Este aspecto tendrá repercusiones en términos de su implantación como podremos apreciar más adelante.

$$Vt_1 = \{0, 1, 2, \dots, 9, *, -\}$$

$$P_1 = \begin{cases} exp \rightarrow & \begin{array}{l} exp * exp \\ exp - exp \\ dig \end{array} \\ dig \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{cases}$$
$$\begin{array}{lll} exp \Rightarrow_2 & exp - exp & \Rightarrow_3 dig - exp \\ & \Rightarrow_4 5 - exp & \Rightarrow_1 5 - exp * exp \\ & \Rightarrow_3 5 - dig * exp & \Rightarrow_4 5 - 8 * exp \\ & \Rightarrow_3 5 - 8 * dig & \Rightarrow_4 5 - 8 * 2 \end{array}$$

La ambigüedad, a final de cuentas, significa que una expresión del lenguaje pueda tener *más de una interpretación*, lo cual para nuestros propósitos no será permitido. En esta última gramática, tal ambigüedad está asociada con los operadores “*” y “-”, por lo que debe establecerse su *asociatividad* y *precedencia* para evitar tal ambigüedad. Detallaremos este aspecto para eliminar la ambigüedad más adelante.

- Describa cuáles son los símbolos terminales, no terminales, etc.
- Describa en lenguaje natural cual es el lenguaje que generan.
- Construya, para alguna expresión del lenguaje, una derivación más por la izq. y otra por la derecha.
- Concluya si la gramática es o no ambigua.

- 48

De igual forma que en los AFs, ante una GLC G , la primer pregunta es ¿cual es el lenguaje generado por G ? Se puede formular una descripción en lenguaje natural al respecto, por ejemplo, para $L(G_0)$ podemos afirmar:

El lenguaje de expresiones aritméticas en notación infija sencillas $L(G_0)$ es el conjunto de cadenas compuestas por dígitos separados por operadores '+' y '-' que sigue las siguientes reglas:

1. *Un dígito es una expresión correcta.*
2. *Si E_1 y E_2 son expresiones correctas, también lo son $E_1 + E_2$ y $E_1 - E_2$.*
3. *Ninguna otra expresión construída del alfabeto es una expresión correcta de este lenguaje.*

Sin embargo, esta descripción *informal* no resulta suficiente, sino que debe demostrarse formalmente que efectivamente se está *construyendo un conjunto de expresiones*: para ello, se puede construir una *prueba por inducción*. Normalmente deben demostrarse por inducción dos *contenciones*²¹:

1. Mostrar que toda frase derivable a partir de S es una expresión del lenguaje.
2. Mostrar que toda cadena del lenguaje puede ser derivable de S .

Mostraremos a continuación para el caso de nuestra gramática G_0 , para ello refinaremos convenientemente la descripción informal: $L(G_0)$ describe el conjunto de expresiones de longitud impar, formadas por parejas de dígitos separados por un *operador* '+' ó '-', es decir, si la cadena tiene n dígitos entonces debe tener $n - 1$ operadores (o equivalentemente, si tiene n operadores entonces tiene $n + 1$ dígitos).

1. Para el primer caso, aplicamos inducción sobre el número de pasos n de una derivación:
 - (a) Caso base: $n = 2$, $exp \Rightarrow dig \Rightarrow k$, donde k es cualquier dígito $(0, \dots, 9)$.
 - (b) Suponiendo que todas las cadenas derivadas con menos de i pasos ($i \geq 2$) producen expresiones correctas, por demostrar que una expresión construída a partir de una derivación con exactamente i pasos también es una expresión correcta. Consideremos una derivación por la izquierda con exactamente i pasos:

$$exp \Rightarrow exp \ op \ dig \overset{*}{\Rightarrow} w \ op \ dig$$

²¹A final de cuentas se trata de mostrar la igualdad entre dos conjuntos: el de las expresiones derivables de S y el conjunto de expresiones que se supone consiste el lenguaje.

donde $op \in \{+, -\}$ y w es una frase que cumple con la hipótesis de inducción (i.e. tiene n dígitos y $n - 1$ operadores). Claramente, la expresión $w \ op \ dig$ genera una frase que tiene $n + 1$ dígitos y n operadores, terminando así esta parte de la prueba.

2. Ahora, para el segundo aspecto, aplicamos inducción con respecto a la longitud de las expresiones (impar):

- (a) Caso base: $n = 1$. Claramente podemos derivar todas las expresiones de longitud 1 de este lenguaje, pues son propiamente dígitos mediante derivaciones de 2 pasos: $exp \Rightarrow dig \Rightarrow '0', '1', \text{etc.}$
- (b) Ahora nuestra hipótesis de inducción es que toda expresión correcta de longitud menor a $2n + 1$ (para $n \geq 1$) es derivable a partir de exp . Sea w una cadena de longitud exactamente $2n + 1$ ($|w| = 2n + 1$), probaremos que w es derivable a partir de exp :

$$exp \Rightarrow exp \ op \ dig \overset{*}{\Rightarrow} v \ op \ dig \Rightarrow w$$

donde claramente v es de longitud $2n - 1$:
 $|v| = 2(n - 1) + 1 = 2n - 1$.

Ejercicios 6.1.4 *Demuestre (aplicando el principio de inducción) cuál es el lenguaje que generan las gramáticas del ejercicio anterior.*

6.2 Árboles de Análisis Sintáctico

Un Árbol de Análisis Sintáctico (AAS) es una representación gráfica de una derivación; no muestra la elección relativa al orden de substitución, por lo tanto ofrece una descripción estructural de las expresiones. Dada una GLC G :

1. La raíz del AAS está etiquetada con S .
2. Cada hoja está etiquetada con un componente léxico (terminal) o con ϵ (cadena nula).
3. Cada nodo interior está etiquetada con un símbolo no-terminal.
4. Si A es un no-terminal que etiqueta a un nodo interior y $X_1 X_2 \dots X_n$ son las etiquetas de los hijos de ese nodo, (de izquierda a derecha) entonces $A \rightarrow X_1 X_2 \dots X_n$ es una producción en P (ver Figura 6.2).



A partir de esta definición se puede inferir un algoritmo para construir un AAS para una expresión dada y una GLC G , partiendo del nodo raíz (etiquetado por S) y llegando hasta los nodos hojas las cuales quedan etiquetadas por los terminales que concatenados en orden de izquierda a derecha forman la frase original. Un aspecto sumamente importante es que el recorrido de un AAS permite realizar la **evaluación** de la expresión correspondiente: para este ejemplo, la evaluación de los nodos hojas resulta el valor asociado con los dígitos, y la evaluación de un nodo interior resulta la aplicación de la *función semántica* (es decir, en este caso la operación aritmética correspondiente, suma y resta) a los valores recién obtenidos.

Para la gramática G_0 y derivación (5.3.2), el AAS que genera la expresión correspondiente se muestra en la Figura 6.29.

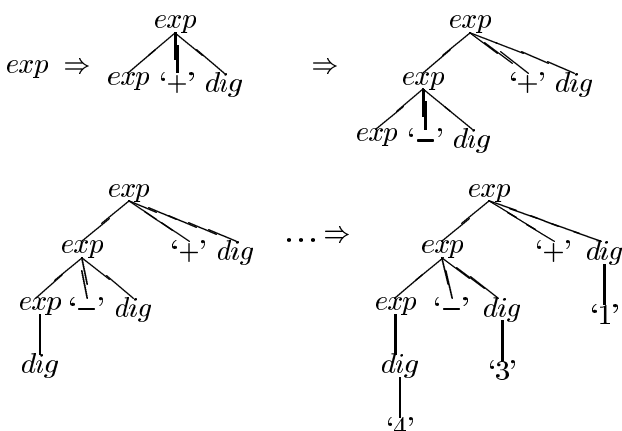


Figura 6.29: Construcción del AAS a partir de la derivación 5.3.2.

Alternativamente, el lenguaje $L(G)$ se define como el conjunto de cadenas generadas por el AAS. Correspondientemente, una GLC G es ambigua cuando una expresión puede ser generada por más de un AAS. La Figura 6.30 muestra 2 AAS's para la expresión " $5 - 8 * 2$ " en G_1 .

En (a) el árbol crece por la izquierda mientras que en (b) crece por la derecha, lo cual significa que en algún punto para la derivación a partir de un nodo se tiene *más de una producción a elegir* (más de un camino) que permite reconocer la *misma subexpresión* (en este caso, la frase completa); en (a) se

eligió la producción 1 al inicio para reducir el primer *exp* (raíz), mientras que en (b) se eligió la producción 2 para reducir el mismo nodo.

Ejercicios 6.2.1 Para cada una de las gramáticas del ejercicio anterior, construya al menos un AAS para alguna expresión dada, muestre en cada caso si la gramática es ambigua.

Asociatividad y Precedencia de operadores

Los 2 AAS anteriores nos establecen 2 formas de agrupamiento ó asociatividad distintas para con los operadores, intuitivamente, si un operador relaciona dentro de un subárbol en mayor profundidad entonces se considera que asocia más fuerte:

1. El AAS (a) indica que el operador ‘ $-$ ’ asocia primero (más fuerte) que ‘ $*$ ’, mientras que el AAS (b) indica lo contrario. Como bien sabemos, la expresión “ $5 - 8 * 2$ ” se evalúa como $5 - (8 * 2) = 5 - 16 = -9$. Sin reglas para especificar la precedencia relativa de los operadores, los *agrupadores* (paréntesis) serían indispensables para indicar los operandos de un operador. Habitualmente, al operador de multiplicación se le establece *mayor prioridad o precedencia* (considera a sus operandos antes) que al operador de resta. A su vez, los operadores $*$, $/$ tienen la misma prioridad, y en seguida los operadores $+$, $-$.
2. Sin embargo, además de prioridad se debe establecer las *asociatividad* ó convenciones de agrupamiento de los operadores, i.e. si asocian por la izquierda o por la derecha: cuando se tienen expresiones que involucran operadores de misma precedencia, por ejemplo $9 + 3 - 1$, ¿cual de las siguientes convenciones es la correcta: $9 + (3 - 1)$ ó $(9 + 3) - 1$? Como es usual, en este caso se considera la segunda, dando como resultado 11.

Un operador es *asociativo por la izquierda* si las subexpresiones que contienen apariciones múltiples del operador (o algún otro de la misma precedencia) se agrupan de izquierda a derecha: los operadores $+$, $-$, $*$, $/$ son asociativos por la izquierda.

Correspondientemente, un operador es *asociativo por la derecha* si las subexpresiones involucrando operadores de igual o menor precedencia se agrupan de derecha a izquierda: el operador de potencia ó exponenciación $^{\wedge}$ es asociativo por la derecha $2^{3^4} = 2^{(3^4)} = 2^{81}$.

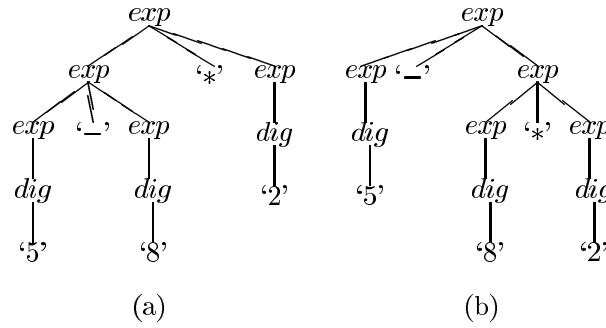


Figura 6.30: G_1 es ambigua: se pueden construir dos AAS diferentes para la misma frase.

Ejercicios 6.2.2 • Defina una GLC (no ambigua) que reconozca expresiones aritméticas involucrando los 5 operadores aritméticos principales (+, -, *, /, ^), agrupadores ('(', ')') y que siga las convenciones de asociatividad y prioridad mencionadas. Para 5 expresiones, construya su AAS correspondiente.

Esta definición indica dos condiciones que todo símbolo X no terminal debe cumplir para considerársele útil:

1. Una frase w debe ser derivable de X : $X \xRightarrow{*} w$
2. X debe ocurrir en alguna forma de frase derivable desde el símbolo inicial S : $S \xRightarrow{*} \alpha X \beta$.

Tales condiciones no son suficientes, pues debe asegurarse que X no ocurra en alguna forma de frase en la cual contenga alguna variable Y a partir de la cual ninguna frase pueda derivarse.

Eliminaremos los no terminales inútiles de una GLC como resultado de los lemas que se describen a continuación. El primero de ellos elimina los no terminales inútiles, mientras que el segundo refina el conjunto de símbolos terminales.

Lema 6.3.1 Dada una GLC $G = \langle Vn, Vt, P, S \rangle$ tal que $L(G) \neq \emptyset$, se puede encontrar una GLC $G' = \langle Vn', Vt, P', S \rangle$ equivalente a G la cual, para cada $X \in Vn'$ existe alguna frase $w \in Vt^*$ que es derivada por X : $X \xRightarrow{*} w$.

Demostración: se ofrece el algoritmo 3 para calcular Vn' . P' es el conjunto de todas las producciones $X \rightarrow \alpha$ tales que $X \in Vn'$ y $\alpha \in (Vn' \cup Vt)^*$.

Algoritmo 3 Cálculo de Vn' .

Entrada: Vn, Vt, P ,

Salida: Vn'

$OldV \leftarrow \emptyset$;

$NewV \leftarrow \{X \mid X \rightarrow w \text{ para alguna frase } w \in Vt^*\}$;

while $OldV \neq NewV$ **do**

$OldV \leftarrow NewV$;

$NewV \leftarrow OldV \cup \{A \mid A \rightarrow \alpha, \text{ para alguna forma de frase } \alpha \in (Vt \cup OldV)^*\}$;

end while

$Vn' \leftarrow NewV$.

6.3 Simplificación de GLC

Existen diversas formas en las cuales se puede restringir el formato de las producciones sin reducir el poder expresivo de las GLC. Si L es un lenguaje libre de contexto no vacío, entonces L puede generarse a partir de una GLC G con las siguientes propiedades:

- Cada no terminal y cada terminal de G aparecen en la derivación de alguna frase en L .
- No hay producciones de la forma $A \rightarrow B$ (para $A, B \in Vn$).

Si $\epsilon \notin L$ no se necesitan producciones $A \rightarrow \epsilon$ en G , lo cual conduce que las producciones de G sean de la forma $A \rightarrow BC$ y $A \rightarrow a$, para $A, B, C \in Vn$ y $a \in Vt$ arbitrarios. De manera alternativa, las producciones de G pueden ser de la forma $A \rightarrow a\alpha$ para α posiblemente vacía. Las anteriores dos formas se denominan la forma normal de Chomsky y la forma normal de Greibach, respectivamente, las cuales analizaremos a partir de las siguientes consideraciones.

Eliminación de Símbolos Inútiles

Primero identificaremos los símbolos inútiles en una GLC $G = \langle Vn, Vt, P, S \rangle$ y mostraremos cómo eliminarlos (i.e., obtener una GLC G' equivalente a G la cual no contiene símbolos inútiles).

Definición 6.3.1 Un símbolo no terminal x es útil si existe una derivación $S \xRightarrow{*} \alpha X \beta \Rightarrow w$ (para $w \in Vt^*$). De lo contrario se dice que X es inútil.

Aplicando primero el lema 6.3.1 y entonces el lema siguiente podemos convertir una GLC G en una equivalente sin símbolos inútiles²².

Lema 6.3.2 *Dada una GLC $G = \langle Vn, Vt, P, S \rangle$ podemos construir una GLC equivalente $G' = \langle Vn', Vt', P', S \rangle$ la cual, para cada símbolo $X \in (Vn \cup Vt)$ existen formas de frase α, β tales que $S \xRightarrow{*} \alpha X \beta$.*

Demostración: el conjunto de símbolos $(Vn \cup Vt)$ que aparecerán en las formas de frase de G' (y por tanto, P' solo tiene a las producciones que tengan símbolos en dicho conjunto) se obtiene por el algoritmo 4.

Algoritmo 4 Cálculo de P' .

```

 $Vn' \leftarrow Vn' \cup \{S\};$ 
if  $A \in Vn'$  y  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$  then
   $Vn' \leftarrow Vn' \cup \{ \text{variables que ocurren en } \alpha_k \}$  para
     $k = 1, \dots, n;$ 
   $Vt' \leftarrow Vt' \cup \{ \text{terminales que ocurren en } \alpha_k \}$ 
    para  $k = 1, \dots, n;$ 
end if

```

Ejemplo 6.3.1 Sea la GLC $G_4 = \langle \{S, A, B\}, \{a, c\}, P, S \rangle$, donde P

$$\begin{array}{lcl} S & \rightarrow & AB \mid c \mid AS \mid C \\ A & \rightarrow & SA \mid BA \mid a \\ C & \rightarrow & c \end{array}$$

Aplicando el lema 6.3.1 a G_4 encontramos que ninguna frase se puede derivar de B , por tanto eliminamos a B de Vn' y a las producciones $S \rightarrow AB, A \rightarrow BA$. Ello da como resultado

$$G'_4 = \langle Vn', Vt, P', S \rangle$$

$$P' = \{S \rightarrow AS \mid c \mid C, A \rightarrow SA \mid a, C \rightarrow c\}$$

Aplicando entonces el lema 6.3.2 a G'_4 vemos que solo S, A, a, c aparecen en formas de frase por lo que eliminamos a $C \rightarrow c$. La GLC G''_4 equivalente a G_4 sin símbolos inútiles queda como

$$G''_4 = \langle Vn', Vt', P'', S \rangle$$

$$P'' = \{S \rightarrow AS \mid c, A \rightarrow SA \mid a\}$$

Eliminación de Producciones ϵ y unitarias

Una producción ϵ es de la forma $A \rightarrow \epsilon$. Si $\epsilon \in L(G)$ no podemos eliminar todas las producciones ϵ , de lo

²²Si se aplican en orden inverso no se garantiza que se eliminen todos los símbolos inútiles.

contrario si podemos. Para ello debemos determinar para cada $A \in Vn$ si $A \xRightarrow{*} \epsilon$. De ser el caso, decimos que A es *nulificable*. Reemplazaremos cada producción $B \rightarrow X_1 X_2 \dots X_n$ por las producciones en las cuales eliminamos subconjuntos de X_i que son nulificables, si ocurre que todas las X_i son nulificables (lo cual significa que $B \rightarrow \epsilon$), debemos descartar la producción $B \rightarrow \epsilon$.

Por su parte, una *producción unitaria* tienen la forma $A \rightarrow B$. Tanto las producciones unitarias como las producciones ϵ son estorbosas pues dificultan determinar si al aplicar una producción se logra algún progreso para derivar una frase. Las producciones unitarias pueden causar ciclos en la derivación, y las producciones ϵ se pueden generar formas de frase de no terminales muy grandes y al final eliminarlas todas. Sin producciones unitarias y ϵ cada paso de la derivación logra progreso en el sentido que la forma de frase crece estrictamente ó se incorpora un nuevo terminal en cada paso de la derivación. No podemos simplemente desprendernos de dichas producciones ya que podrían ser necesarias para generar algunas cadenas de $L(G)$.

Lema 6.3.3 *Sea $G = \langle Vn, Vt, P, S \rangle$ una GLC cualquiera, existe una GLC G' sin producciones ϵ ó producciones unitarias tal que $L(G') = L(G) - \{\epsilon\}$.*

Demostración: sea \hat{P} el conjunto más pequeño de producciones que contiene a P y es cerrado bajo las siguientes cláusulas:

1. si $A \rightarrow \alpha B \beta$ y $B \rightarrow \epsilon$ están en \hat{P} , entonces $A \rightarrow \alpha \beta$ está en \hat{P} ;
2. Si $A \rightarrow B$ y $B \rightarrow \gamma$ está en \hat{P} , entonces $A \rightarrow \gamma$ está en \hat{P} .

Podemos construir a \hat{P} inductivamente a partir de P incorporando producciones como sea requerido para satisfacer las dos cláusulas anteriores. Nótese que sólo se incorporan un número finito de producciones dado que cada nueva parte derecha es una subcadena de una parte derecha de una producción vieja; en otras palabras, estamos resumiendo lo que hace una pareja vieja de producciones en una sola nueva producción. Por tanto \hat{P} es finito.

Sea \hat{G} la GLC

$$\hat{G} = \langle Vn, Vt, \hat{P}, S \rangle$$

Dado que $P \subseteq \hat{P}$ cada derivación de G también es derivación de \hat{G} por lo que $L(G) \subseteq L(\hat{G})$. Pero gracias a que cada nueva producción que se incorporó por las cláusulas anteriores se puede simular en dos pasos por las producciones que la causaron, se tiene que $L(\hat{G}) \subseteq L(G)$, por lo tanto $L(G) = L(\hat{G})$.

Mostraremos a continuación que para cualquier frase no nula $w \in Vt^*$, toda derivación $S \xRightarrow{*} w$ de longitud mínima no usa alguna producción ϵ ó una producción unitaria; por tanto tales producciones son superfluas y pueden omitirse de \hat{G} con toda seguridad.

Sea $w \neq \epsilon$ y considérese una derivación de longitud mínima $S \xRightarrow{*} w$. Supóngase para propósitos de contradicción que una producción ϵ , $B \rightarrow \epsilon$ ha sido utilizada en algún punto de la derivación, i.e.

$$S \xRightarrow{*} \gamma B \lambda \Rightarrow \gamma \lambda \xRightarrow{*} w$$

donde alguna γ, λ son no nulas, de otra forma w sería nula contradiciendo la premisa de que w no es nula. La ocurrencia de la primer B en una forma de frase ocurre en la derivación cuando se aplicó la producción $A \rightarrow \alpha B \beta$:

$$S \xRightarrow{*}_m \eta A \kappa \Rightarrow \eta \underbrace{\alpha B \beta}_n \kappa \xRightarrow{*}_n \gamma B \lambda \Rightarrow \gamma \lambda \xRightarrow{*}_k w$$

donde los subíndices indican el número de pasos de derivación, para $m, n, k \geq 0$. Por la segunda cláusula, $A \rightarrow \alpha \beta$ también está en \hat{P} y por tanto ésta producción pudo haberse aplicado en lugar de $A \rightarrow \alpha B \beta$ de forma que se obtiene una derivación para w estrictamente más corta

$$S \xRightarrow{*}_m \eta A \kappa \Rightarrow \eta \underbrace{\alpha \beta}_k \kappa \Rightarrow \gamma \lambda \xRightarrow{*}_k w$$

lo cual contradice la afirmación de que se parte de una derivación de longitud mínima.

De forma similar, puede demostrarse que las producciones unitarias no aparecen en las derivaciones de longitud mínima en \hat{G} lo cual se deja como ejercicio al lector.

Por tanto, las producciones ϵ y unitarias no son necesarias para derivar cadenas no nulas. Si las descartamos de \hat{G} obtenemos una GLC G' que deriva el lenguaje $L(G) - \{\epsilon\}$.

6.3.1 Forma Normal de Chomsky

Una vez que hemos prescindido de las producciones unitarias y ϵ , es tarea simple poner la GLC resultante en la forma normal de Chomsky (FNC):

1. Para cada terminal $a \in Vt$, se introduce un nuevo no terminal A_a y una producción $A_a \rightarrow a$, y se reemplaza toda ocurrencia de a en la parte derecha de una producción que originalmente ya estaba en P (excepto en producciones de la forma $B \rightarrow a$) por $A_a \rightarrow a$.

2. Todas las producciones quedan de la forma

$$A \rightarrow a \text{ ó } A \rightarrow B_1 B_2 \dots B_k, \text{ con } k \geq 2$$

donde B_i son no terminales.

El conjunto de cadenas derivadas con ésta nueva gramática es exactamente el mismo, solo necesitan un paso más antes de generar un símbolo terminal.

3. Para cada producción

$$A \rightarrow B_1 B_2 \dots B_k \text{ con } k \geq 3$$

se introduce un nuevo noterminal C y se reemplaza tal producción con las dos producciones

$$A \rightarrow B_1 C \text{ y } C \rightarrow B_2 B_3 \dots B_k$$

Se hace este proceso hasta que todas las partes derechas de las producciones sean de longitud a lo más dos.

Ejemplo 6.3.2 *Construiremos la forma normal de Chomsky para*

$$\{a^n b^n \mid n \geq 0\} - \{\epsilon\} = \{a^n b^n \mid n \geq 1\}$$

Partimos de las producciones siguientes

$$S \rightarrow aSb \mid \epsilon$$

la cual genera el lenguaje $\{a^n b^n \mid n \geq 0\}$. A continuación eliminamos las producciones ϵ para obtener

$$S \rightarrow aSb \mid ab$$

la cual genera $\{a^n b^n \mid n \geq 1\}$. Entonces, se siguen los pasos descritos: se incorporan los no terminales A, B y se reemplazan las anteriores producciones por

$$S \rightarrow ASB \mid AB, \quad A \rightarrow a, \quad B \rightarrow b$$

Finalmente, incorporamos un no terminal C para reemplazar la parte derecha que tiene más de tres símbolos, en éste caso $S \rightarrow ASB$ por

$$S \rightarrow AC, \quad C \rightarrow SB$$

La GLC en forma normal de Chomsky es

$$\begin{aligned} S &\rightarrow AB \mid AC, & C &\rightarrow SB, \\ A &\rightarrow a, & B &\rightarrow b \end{aligned}$$

Ejercicios 6.3.1 • Eliminar los símbolos inútiles de $S \rightarrow AB \mid CA, B \rightarrow BC \mid AB, A \rightarrow a, C \rightarrow aB \mid b$.

• Construya las formas normales de Chomsky para las siguientes gramáticas:

1. $S \rightarrow [S] \mid SS \mid \epsilon$
2. $S \rightarrow \neg A \mid (S \supset S) \mid p \mid q$.
3. $S \rightarrow aB \mid bA, A \rightarrow a \mid aS \mid bAA, B \rightarrow b \mid bS \mid aBB$.

6.3.2 Forma Normal de Greibach

Para convertir una GLC $G = \langle Vn, Vt, P, S \rangle$ arbitraria (excepto posiblemente por ϵ) en forma normal de Greibach (FNG) partimos (por simplicidad) de considerar que G está en forma normal de Chomsky. La construcción que se describe a continuación produce una GLC en forma normal de Greibach con a lo más dos terminales en la parte derecha y consta de dos fases en las que se construyen dos GLC G_1 y G_2 auxiliares:

Entrada:		Salida:
G	$\rightsquigarrow G_1 \rightsquigarrow G_2$	$\rightsquigarrow G'$
(en FNC)		(en FNG)

Antes de mostrar cada fase, conviene mostrar un resultado que será utilizado: hemos podido apreciar que los lenguajes libres de contexto pueden describir conjuntos que los lenguajes regulares no pueden, sin embargo, no hemos enfatizado que las GLC pueden describir naturalmente a los conjuntos regulares. A continuación, en la tabla se muestran cuatro tipos especiales de GLC que generan exactamente a los conjuntos regulares:

Tipo de GLC	
Lineal por la derecha:	
$A \rightarrow xB$	$A \rightarrow x$
Fuertemente lineal por la derecha:	
$A \rightarrow aB$	$A \rightarrow \epsilon$
Lineal por la izquierda:	
$A \rightarrow Bx$	$A \rightarrow x$
Fuertemente lineal por la izquierda:	
$A \rightarrow Ba$	$A \rightarrow \epsilon$

donde A, B son no terminales, a indica un terminal y x es una frase. Nótese la forma que exhiben: no hay restricciones en el sentido que A y B en cada producción necesariamente deban ser diferentes. Se deja al lector demostrar que los cuatro tipos de GLC anteriores generan a los conjuntos regulares (y por lo tanto son equivalentes entre sí); la recomendación es demostrar por inducción sobre el número de símbolos en la definición de un no terminal (parte derecha de las producciones) bajo la apreciación ilustrada en la Figura 6.31.

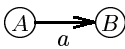

Producción	Autómata
$A \rightarrow aB$	
$A \rightarrow aA$ $A \rightarrow \epsilon$	

Figura 6.31: Idea para la prueba.

Ahora podemos comenzar a describir la construcción de G' en forma normal de Greibach a partir de G en forma normal de Chomsky: recuérdese que para $\alpha, \beta \in (Vn \cup Vt)^*$ formas de frase, escribimos

$$\alpha \xRightarrow{*}_{mi} \beta$$

para indicar que β se derivó de α utilizando una derivación más por la izquierda. Para $A \in Vn, a \in Vt$ se define

$$R_{A,a} = \{\beta \in Vn^* \mid A \xRightarrow{*}_{mi} a\beta\}$$

Por ejemplo, para la GLC

$$S \rightarrow AB \mid AC \mid SS, C \rightarrow SB, A \rightarrow (' , B \rightarrow ')$$

se tiene

$$\begin{aligned} C &\xRightarrow{*}_{mi} SB \xRightarrow{*}_{mi} SSB \xRightarrow{*}_{mi} SSSB \\ &\xRightarrow{*}_{mi} ACSSB \xRightarrow{*}_{mi} (CSSB \end{aligned}$$

por tanto $CSSB \in R_{C,('}$.

El conjunto $R_{A,a}$ es regular sobre el alfabeto Vn y por tanto se puede construir una GLC fuertemente lineal por la izquierda con no terminales $\{A' \mid A \in Vn\}$, como terminales el conjunto Vn , como símbolo inicial S' y producciones dadas por

$$\{A' \rightarrow B'C \mid A \rightarrow BC \in P\} \cup \{A' \rightarrow \epsilon \mid A \rightarrow a \in P\}$$

Nótese que los terminales de ésta gramática son los no terminales Vn de G , pero por el momento consideremos que ello tiene sentido.

Dado que $R_{A,a}$ es regular, se puede a su vez definir una gramática fuertemente lineal por la derecha, llamémosla $G_{A,a}$, en la cual sus producciones son todas de la forma $X \rightarrow BY$ ó $X \rightarrow \epsilon$ para X, Y no terminales de $G_{A,a}$ y $B \in Vn$. Indiquemos por $T_{A,a}$ al símbolo inicial de $G_{A,a}$. Asíumase que los conjuntos de no terminales de G y $G_{A,a}$ son disjuntos (si no lo fueran, simplemente se renombran). Construiremos la gramática G_1 incorporando todos los no terminales y producciones de $G_{A,a}$ a G . El símbolo inicial de G_1 será S (i.e., el símbolo inicial de G). Las producciones de G_1 son de las siguientes formas

$$X \rightarrow b \quad X \rightarrow \epsilon \quad X \rightarrow BY$$

Nótese que G_1 es equivalente a G dado que ninguno de los nuevos no terminales se pueden derivar de S .

Ahora, sea G_2 la GLC obtenida de G_1 eliminando toda producción de la forma

$$X \rightarrow BY$$

y reemplazándola por

$$X \rightarrow bT_{B,b}Y$$

para toda $b \in Vt$. Las producciones de G_2 tienen la forma

$$X \rightarrow b \quad X \rightarrow \epsilon \quad X \rightarrow bT_{B,b}Y$$

Finalmente, se eliminan las producciones ϵ de G_2 . La gramática resultante está en forma normal de Greibach con a lo más dos no terminales en la parte derecha de cualquier producción.

Ejemplo 6.3.3 *Mostraremos la construcción de la forma normal de Greibach para la gramática cuyo lenguaje son las palabras formadas por 0 y 1 balanceados.*

Partimos de la GLC en forma normal de Chomsky:

$$\begin{array}{l} S \rightarrow AB \mid AC \mid SS, \quad C \rightarrow SB \\ A \rightarrow 0 \quad \quad \quad B \rightarrow 1 \end{array}$$

Primero calculamos los conjuntos regulares $R_{D,d}$:

$$\begin{array}{l} R_{S,0} = (B + C)S^* \\ R_{C,0} = (B + C)S^*B \\ R_{A,0} = R_{B,1} = \{\epsilon\} \end{array}$$

todos los demás resultan \emptyset . A continuación mostramos gramáticas fuertemente lineales por la derecha para cada uno de los anteriores conjuntos:

$$\begin{array}{l} T_{S,0} \rightarrow BX \mid CX, \quad X \rightarrow SX \mid \epsilon \\ T_{C,0} \rightarrow BY \mid CY, \quad Y \rightarrow SY \mid BZ \quad Z \rightarrow \epsilon \\ T_{A,0} \rightarrow \epsilon \quad \quad \quad T_{B,1} \rightarrow \epsilon \end{array}$$

Incorporando las nuevas producciones obtenidas con las de G original y realizando las substituciones indicadas, se obtiene G_2 :

$$\begin{array}{l} S \rightarrow 0T_{A,0}B \mid 0T_{A,0}C \mid 0T_{S,0}S \quad C \rightarrow 0T_{S,0}B \\ T_{S,0} \rightarrow 1T_{B,1}X \mid 0T_{C,0}X \quad X \rightarrow 0T_{S,0}X \mid \epsilon \\ T_{C,0} \rightarrow 1T_{B,1}Y \mid 0T_{C,0}Y \quad Y \rightarrow 0T_{S,0}Y \mid 1T_{B,1}Z \\ T_{A,0} \rightarrow \epsilon \quad \quad \quad B \rightarrow 1 \\ A \rightarrow 0 \quad \quad \quad T_{B,1} \rightarrow \epsilon \\ Z \rightarrow \epsilon \end{array}$$

Eliminando las producciones ϵ se tiene la gramática final en forma normal de Greibach:

$$\begin{array}{l} S \rightarrow 0B \mid 0C \mid 0T_{S,0}S \quad C \rightarrow 0T_{S,0}B \\ T_{S,0} \rightarrow 1X \mid 0T_{C,0}X \mid 0T_{C,0} \quad X \rightarrow 0T_{S,0}X \mid 0T_{S,0} \\ T_{C,0} \rightarrow 1Y \mid 0T_{C,0}Y \quad Y \rightarrow 0T_{S,0}Y \mid 1 \\ A \rightarrow 0 \quad \quad \quad B \rightarrow 1 \end{array}$$

Ejercicios 6.3.2 *Para las GLC del ejercicio anterior obtenga las equivalentes en forma normal de Greibach.*

6.4 Lema de Sondeo para Lenguajes libres de contexto

La versión del lema de sondeo para Lenguajes libres de contexto es similar a la anterior para conjuntos regulares. Se puede utilizar para demostrar que un lenguaje no es libre de contexto. Presentaremos primero la versión positiva para entonces plantear la versión contrapositiva que de la misma forma como en el caso de conjuntos regulares la podemos describir como un juego entre dos contendientes.

Lema 6.4.1 (Lema de Sondeo para LLC) *Para todo LCC L , existe $k \geq 0$ tal que para toda cadena $z \in L$ de longitud al menos k , z puede descomponerse en cinco subcadenas $z = uvwxy$ en la cual $vx \neq \epsilon$, $|vwx| \leq k$, y para toda $i \geq 0$, $uv^iwx^iy \in L$.*

Informalmente, para todo LLC L , toda cadena suficientemente larga en L se puede descomponer en cinco subcadenas tales que los tres segmentos de en medio no son demasiado grandes (i.e., tienen a lo más longitud k fijo), el segundo y cuarto no son ambos nulos y no importa cuantas copias extra de las subcadenas segunda y cuarta uno bombee simultáneamente, la cadena resultante también está en L . Se puede notar inmediatamente que éste lema difiere de su primo para conjuntos regulares en que se pueden bombear dos cadenas v y x separadas por una cadena w .

La idea fundamental que da como resultado éste lema es que para una gramática en forma normal de Chomsky²³ cualquier árbol de análisis sintáctico para una cadena muy larga debe tener una ruta muy larga, y toda ruta considerablemente larga deben existir al menos dos ocurrencias de algún no terminal. Por ejemplo, considérese la siguiente GLC:

$$S \rightarrow AC \mid AB, \quad A \rightarrow a, \quad B \rightarrow b, \quad C \rightarrow SB$$

la cual permite describir el lenguaje $\{a^n b^n \mid n \geq 1\}$. En la Figura 6.32 se muestra un árbol de análisis sintáctico para la cadena $a^4 b^4$:

Los AAS para cadenas suficientemente largas deben tener rutas proporcionalmente largas debido que el número de símbolos puede a lo más duplicarse²⁴ cuando se desciende un nivel: se tienen a lo más 2^0 símbolos en la raíz, 2^1 símbolos en el nivel 1, etc. hasta 2^n símbolos en el nivel n , para $n + 1$ la profundidad del árbol.

²³No es una restricción, en principio la gramática puede no estar en forma normal de Chomsky pero estando en esa forma simplifica mucho la exposición.

²⁴Recuérdese que en forma normal de Chomsky, todo AAS tiene factor de ramificación a lo más 2.

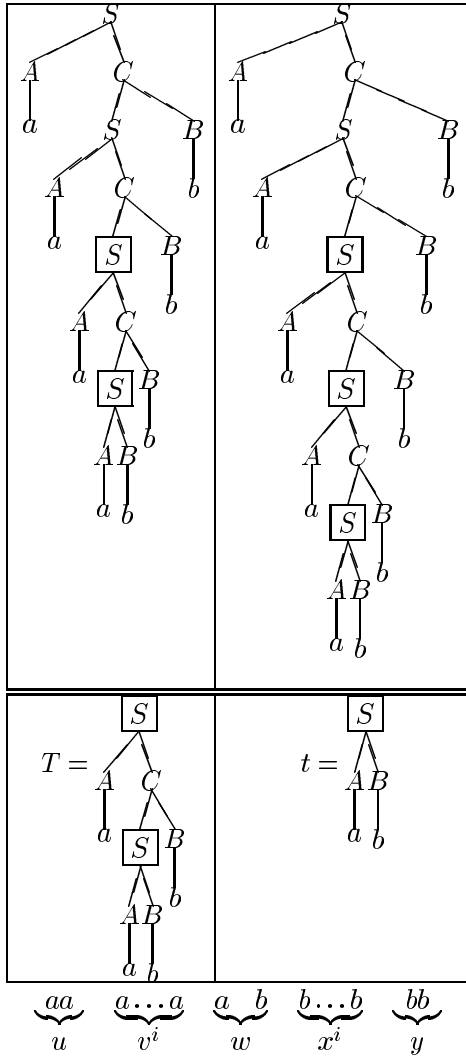


Figura 6.32: Ilustración del lema de sondeo para LLC: el AAS de la izquierda corresponde a la frase $aaaabbbb$, mientras que el de la derecha se obtuvo reemplazando t por T , permitiendo derivar la frase $aaaaabbbb$.

A continuación ofrecemos una prueba del lema del sondeo para LLC:

Demostración: sea G una GLC en FNC para L . Tómese $k = 2^{n+1}$, para n el número de no terminales de G ($n = |V_n|$). Sea z una cadena de L donde $|z| \geq k$. Por el argumento que se ilustró con el AAS de la Figura 6.32, cualquier AAS en G para z debe tener profundidad al menos $n+1$. Considérese la ruta más larga del AAS (en la Figura, la ruta desde S en la raíz hasta el terminal b situado más a la izquierda en la frase). Tal ruta es de longitud al menos $n+1$, por lo tanto debe contener al menos $n+1$ ocurrencias de no terminales. Ello significa que algunos no terminales

ocurren más de una vez a lo largo de la ruta más larga. Tómese la primera pareja de ocurrencias del mismo no terminal en la ruta, leídas de abajo hacia arriba. En el ejemplo anterior, las indicamos encerrándolas en un cuadrado (ver Figura 6.32). Sea X el no terminal con las dos ocurrencias. Descomponemos a z en las subcadenas $uvwxy$ tales que w es la frase derivada por la ocurrencia inferior de X y vwx es la frase derivada por la ocurrencia superior de X . En el ejemplo, $w = ab$ y $vwx = aabb$.

Sea T la raíz del subárbol correspondiente a la ocurrencia superior de X y sea t el subárbol enraizado en la ocurrencia inferior de X (ver parte inferior de la Figura 6.32). Reemplazando t por una copia de T en el AAS original, se obtiene un AAS válido para la cadena correspondiente a uv^2wx^2y (AAS de la derecha en la Figura 6.32). Podemos realizar este reemplazo para obtener AAS válidos para uv^iwx^iy , $i \geq 0$. Nótese que $vx \neq \epsilon$.

Nótese también que $|vwx| \leq k$ dado que se elige la primera ocurrencia repetida de un no terminal leído desde la parte inferior del AAS (i.e., correspondiente al subárbol t), y debemos ver tal repetición cuando subimos a la altura n (correspondiente al subárbol T). Dado que tomamos la ruta más larga en el árbol, la profundidad de tal subárbol bajo la ocurrencia superior (T) del no terminal repetido X es a lo más $n+1$, por lo que tenemos no más de $2^{n+1} = k$ terminales.

Jugando contra el Diablo

Como su primo para lenguajes regulares, el lema de sondeo para LLC es más útil en su forma contrapositiva, en la cual se establece que para demostrar que A no es libre de contexto es suficiente demostrar la siguiente propiedad:

Lema 6.4.2 (Forma Contrapositiva) *Para todo $k \geq 0$ existe una cadena $z \in A$ de longitud al menos k ($|z| \geq k$) tal que para cualquier forma de descomponer z en subcadenas $z = uvwxy$ con $vx \neq \epsilon$, $|vwx| \leq k$, existe una $i \geq 0$ tal que*

$$uv^iwx^iy \notin A$$

Mostrar la propiedad anterior significa que uno tiene una estrategia ganadora en el siguiente juego contra el diablo:

1. El diablo elige $k \geq 0$.
2. Uno elige $z \in A$ con $|z| \geq k$.
3. El diablo elige la descomposición de z en subcadenas u, v, w, x, y tal que $z = uvwxy$, $|vx| > 0$ y $|vwx| \leq k$.

4. Uno elige una $i \geq 0$. Si la cadena $uv^iwx^iy \notin A$ uno gana.

Ejemplo 6.4.1 Aplicaremos el lema de sondeo para LLC para demostrar que el conjunto

$$A = \{a^n b^n a^n \mid n \geq 0\}$$

no es libre de contexto.

Para ellos, mostraremos que siempre podemos ganar al diablo, sin importar que k elija al principio:

El diablo elige una k en el paso 1. Una buena elección en el paso 2 es $z = a^k b^k a^k$, claramente $z \in A$ y $|z| = 3k \geq k$. En el paso 3, el diablo elige la descomposición de z en u, v, w, x, y con $|vx| > 0$ y $|vwx| \leq k$. Al elegir simplemente $i = 2$ uno gana para cualquiera de las descomposiciones que el diablo elija:

1. si ya sea v ó x contienen al menos una a y al menos una b , entonces uv^2wx^2y no es de la forma $a^n b^n a^n$ y por tanto no está en A .
2. Si v y x contienen sólo a 's entonces uv^2wx^2y tiene dos veces más a 's que b 's y tampoco está en A . (Similarmente si v y x contienen solamente b 's).
3. Si v ó x contiene solamente a 's y el otro solamente b 's entonces uv^2wx^2y tampoco es de la forma $a^n b^n a^n$.

Esto muestra que $uv^iwx^iy \notin A$ y por tanto A no es libre de contexto.

Ejercicios 6.4.1

1. Demuestre que los siguientes lenguajes no son libres de contexto:
 - (a) $\{a^i b^j c \mid i < j < k\}$
 - (b) $\{a^i b^j \mid j = i^2\}$
 - (c) $\{a^i \mid i \text{ es primo}\}$
 - (d) $L = \{a + b + c\}^*$ donde $\#a(w) = \#b(w) = \#c(w)$ para toda $w \in L$.
2. Indique cuales de los siguientes son LLC:
 - (a) $\{a^i b^j \mid i \neq j \text{ y } i \neq 2j\}$
 - (b) $(\mathbf{a} + \mathbf{b})^* - \{(a^n b^n)^n \mid n \geq 1\}$
 - (c) $\{ww^R w \mid w \in (\mathbf{a} + \mathbf{b})^*\}$.

Autómatas de Pila

Como mencionamos en el capítulo anterior, el modelo mecánico cuya clase de lenguajes coincide exactamente con la clase de lenguajes libres de contexto es el Autómata de Pila (AP, *Push down automaton*, o también, Autómata de Empilamiento). Básicamente es una extensión al modelo de autómatas finito no determinístico que tiene una pila para almacenar información en forma *primero en entrar, último en salir* (LIFO). De igual forma que su antecesor, el AP lee símbolos de izquierda a derecha de su cinta de entrada. En cada paso la máquina desempila el símbolo en el tope de la pila, lee el símbolo en la cinta bajo la cabeza lectora y en base al estado actual puede empilar una secuencia de símbolos en la pila, mover la cabeza lectora a la derecha y entrar en un nuevo estado de acuerdo a la regla de transición. También permite *transiciones-ε* en las que puede desempilar y empilar símbolos sin leer el siguiente símbolo de entrada de su cinta (o lo que es lo mismo, sin mover la cabeza lectora).

Aunque puede almacenar una cantidad ilimitada de información en su pila, no puede leer símbolos de la misma en orden arbitrario, primero debe desempilar los símbolos situados en la parte superior (tope) de la misma, en cuyo caso se pierden. Por tanto, el acceso a la información de la pila es limitada.

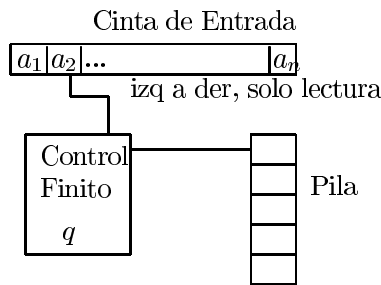


Figura 7.33: Esquema de un Autómata de Pila (AP).

Definición 7.1.1 (AP) *Un Autómata de Pila no determinístico (AP) es una tupla:*

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, \perp, F \rangle$$

donde:

- Q es un conjunto finito de estados,
- Σ es el alfabeto de entrada (cinta),
- Γ es el alfabeto de la pila,
- $\delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma) \times (Q \times \Gamma^*)$ es la relación de transición,
- $q_0 \in Q$ es el estado inicial,
- $\perp \in \Gamma$ es el símbolo inicial de la pila,
- $F \subseteq Q$ es el conjunto de estados de aceptación.

Transiciones

La transición

$$((p, a, A), (q, B_1 B_2 \dots B_k)) \in \delta$$

significa que estando la máquina en el estado p leyendo el símbolo de entrada a en la cinta y estando A en el tope de la pila, puede desempilar A y empilar a los símbolos $B_1 B_2 \dots B_k$ (B_1 en el tope y B_k al fondo), mover su cabeza lectora una celda a la derecha de a y entrar en el estado q . Nótese que la notación anterior indica claramente que δ es en general una relación.

La forma habitual de describir una *transición* es

$$\delta(p, a, A) = \{(q_1, \gamma_1), (q_2, \gamma_2), \dots, (q_m, \gamma_m)\}$$

donde $q_i \in Q$ ($1 \leq i \leq m$), $\gamma_i \in \Gamma^*$ (i.e., es una serie de símbolos del alfabeto de la pila). La transición indica que estando A en el tope de la pila y leyendo el símbolo a en la cinta, puede entrar en cualquier estado q_i , desempilar A y empilar en su lugar la secuencia γ_i y avanzar una celda a la derecha la cabeza lectora. Obviamente no se permite elegir un estado p_i y una secuencia γ_j para $i \neq j$ en un movimiento.

Por otra parte, la descripción de transición

$$((p, \epsilon, A), (q, B_1 B_2 \dots B_k)) \in \delta$$

significa que estando la máquina en el estado p con A en el tope de la pila, a continuación desempila A y empila $B_1 B_2 \dots B_k$ (B_1 primero y B_k al final), deja

su cabeza lectora donde está (no la mueve) y entra al estado q . Una manera equivalente de denotar la transición anterior es

$$(p, \epsilon, A) \xrightarrow{\delta} (q, B_1 B_2 \dots B_k)$$

En general

$$\delta(p, \epsilon, A) = \{(q_1, \gamma_1), (q_2, \gamma_2), \dots, (q_m, \gamma_m)\}$$

para todo $1 \leq i \leq m$.

Configuraciones

Una configuración ó descripción instantánea de un AP registra el estado actual (un elemento de $Q \times \Sigma^* \times \Gamma^*$), así como la porción de la cinta de entrada aún no leída (todo lo situado a la derecha de la cabeza lectora) y el contenido actual de la pila. Una configuración, por lo tanto, ofrece información completa del estado global de la máquina en un instante durante un cómputo. Por ejemplo, la configuración

$$(q, baaab, ABAC\perp)$$

corresponde a la que se muestra en la Figura 7.34

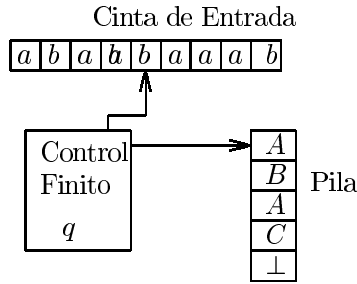


Figura 7.34: Descripción instantánea de un AP.

Nótese que la porción que ya se vió de la cinta de entrada (situada a la izquierda de la cabeza lectora) no necesita mostrarse en la configuración pues no afecta el cómputo que va a realizarse. En general, el conjunto de configuraciones es infinito.

La *configuración inicial* ante la cadena de entrada w es (q_0, w, \perp) , i.e., la máquina siempre comienza en el estado inicial q_0 con la cabeza lectora apuntando al símbolo de w situado más a la izquierda y la pila vacía (lo cual se indica por \perp).

El cambio de configuraciones se define mediante la relación *Siguiente Configuración*:

Definición 7.1.2 (\rightarrow_M) *La relación Siguiente Configuración indicada por \rightarrow_M describe cómo el AP M cambia de una configuración a otra en un solo paso, solamente hay dos casos:*

1. Si se tiene registrada la transición

$$((p, a, A), (q_i, \gamma_i)) \in \delta$$

entonces para cualquier $w \in \Sigma^*$ y $\beta \in \Gamma^*$,

$$(p, aw, A\beta) \rightarrow_M (q_i, w, \gamma_i\beta)$$

2. y si es el caso que

$$((p, \epsilon, A), (q_i, \gamma_i)) \in \delta$$

entonces para cualquier $w \in \Sigma^*$ y $\beta \in \Gamma^*$,

$$(p, w, A\beta) \rightarrow_M (q_i, w, \gamma_i\beta)$$

En el primer caso, M se come al símbolo de entrada a ; en la pila, de $A\beta$ se cambió a $\gamma_i\beta$ indicando que A fué desempilada y γ_i se empiló, y el estado p se transformó a algún q_i . En el segundo caso todo es similar excepto que la cadena de entrada w no cambió lo cual indica que no se comió algún símbolo de entrada (implícitamente se comió a ϵ).

Definición 7.1.3 ($\xrightarrow{*}_M$) *Se define la Cerradura Reflexiva y Transitiva de \rightarrow_M , denotada por $\xrightarrow{*}_M$ como sigue: para C, D, E configuraciones*

$$\begin{aligned} C &\xrightarrow{0}_M D \stackrel{def}{\iff} C = D, \\ C &\xrightarrow{n+1}_M D \stackrel{def}{\iff} \exists E, C \xrightarrow{n}_M E \text{ y } E \xrightarrow{1}_M D, \\ C &\xrightarrow{*}_M D \stackrel{def}{\iff} \exists n \geq 0, C \xrightarrow{n}_M D. \end{aligned}$$

Aceptación

Aunque hay dos definiciones de aceptación, por estado final ó por vaciado de la pila, a fin de cuentas son equivalentes:

Definición 7.1.4 (Por Estado Final) *Sea un AP $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \perp, F \rangle$, se define al lenguaje $L(M)$ aceptado por M por estado final como*

$$L(M) \stackrel{def}{=} \{w \mid (q_0, w, \perp) \xrightarrow{*}_M (p, \epsilon, \gamma)\}$$

es decir, M acepta la entrada w si entra en un estado de aceptación $p \in F$ después de haber examinado completamente a w ($\gamma \in \Gamma^*$).

Definición 7.1.5 (Por Vaciado de Pila) *Se define $N(M)$ el lenguaje aceptado por M por vaciado de pila (ó pila nula) como*

$$N(M) \stackrel{def}{=} \{w \mid (q_0, w, \perp) \xrightarrow{*}_M (p, \epsilon, \epsilon)\}$$

es decir, M vacía la pila después de examinar la cadena de entrada completamente, sin importar en qué estado quede (i.e., $p \in Q$).

Dado que se permiten transiciones ϵ , el AP puede estar en un ciclo potencialmente infinito sin leer algún símbolo de la cinta de entrada.

Ejemplo 7.1.2 *Mostramos un AP para reconocer el lenguaje de paréntesis balanceados ‘(,)’ por vaciado de pila:*

$$M = \langle \{q_o\}, \{(' , ')'\}, \{A, \perp\}, \delta, q_o, \perp, \{\}, \rangle$$

- i) $\delta(q_o, (' , \perp) = \{(q_o, A\perp)\}$
- ii) $\delta(q_o, (' , A) = \{(q_o, AA)\}$
- iii) $\delta(q_o, (' , \epsilon) = \{(q_o, \epsilon)\}$
- iv) $\delta(q_o, \epsilon, \perp) = \{(q_o, \epsilon)\}$

Las primeras dos transiciones indican que si se lee un ‘(’ se empila una A (“abre”), la tercer transición indica que si se lee un ‘)’ y siempre que haya una A en el tope de la pila se desempila la A y no se empila algún símbolo. La cuarta transición se activa cuando se alcanza el final de la cadena de entrada.

El siguiente es un cómputo que muestra la aceptación de la cadena de entrada “(())()”:

	Configuración	transición aplicada
	$(q_o, (())(), \perp)$	inicio
\rightarrow_M	$(q_o, (())(), A\perp)$	i)
\rightarrow_M	$(q_o, (())(), AA\perp)$	ii)
\rightarrow_M	$(q_o, (())(), A\perp)$	iii)
\rightarrow_M	$(q_o, (())(), \perp)$	iii)
\rightarrow_M	$(q_o, (())(), A\perp)$	i)
\rightarrow_M	$(q_o, (())(), \epsilon, \perp)$	iii)
\rightarrow_M	$(q_o, (())(), \epsilon, \epsilon)$	iv)

Nótese que M pudo aplicar la transición iv) prematuramente en el primer paso, conduciendo a la siguiente configuración:

$$(q, (())(), \epsilon)$$

Dado que la máquina es no determinística lo anterior no es problema pues es suficiente con que alguna secuencia de transiciones conduzca a la configuración de aceptación. Como ejercicio, el lector debe construir el árbol que muestre las posibles configuraciones al aplicar distintas transiciones.

Para demostrar que la máquina está correcta, se debe verificar que para toda cadena x balanceada de Σ^* existe una secuencia de transiciones que conducen a una configuración de aceptación partiendo de la configuración inicial; y que para cualquier cadena con paréntesis no balanceados no existe alguna secuencia de transiciones que terminan en una configuración de aceptación.

Terminamos la sección con algunos comentarios útiles:

1. En un *autómata de pila determinístico* se necesita adicionalmente un marcador que indique en donde se termina la cadena de entrada. Tal marcador no es necesario en un AP no determinístico pues simplemente la máquina continuamente supone dónde podría finalizar la cadena (i.e., elegir aplicar alguna transición ϵ), si ocurre que vacía su pila antes de finalizar de examinar toda la cadena de entrada simplemente fué acaso una suposición equivocada.
2. El símbolo \perp simplemente es para indicar la configuración inicial, pero se puede tratar como a cualquier otro símbolo de la pila (empilarse o desempilarse en cualquier momento).
3. Una transición $\delta(p, a, A) = \{(q, \beta)\}$ ó $\delta(p, \epsilon, A) = \{(q, \beta)\}$ no se aplica a menos que se encuentre A en el tope de la pila; en particular, ninguna transición se aplica si la pila está vacía, en cuyo caso se dice que la máquina está *atorada*.
4. En la aceptación por vaciado de pila, la pila debe estar completamente vacía en una configuración, i.e., *después* de aplicar una transición.

7.2 Autómatas de Pila y Lenguajes Libres de Contexto

Demostraremos a continuación que la clase de lenguajes que aceptan los AP son LCC. Para ello, primero demostraremos que cualquier AP que termina por vaciado de pila es equivalente a un AP que termina por estado de aceptación (y viceversa). Tal equivalencia es efectiva.

La construcción que se presentará aprovecha los aspectos comunes en la simulación de un AP que termina por estado de aceptación por un AP que termina por vaciado de pila, de tal suerte que pondremos ambas construcciones juntas y sólo señalaremos los aspectos en que difieren.

Teorema 7.2.1 $(L(M) \longleftrightarrow N(M))$ Sea L el lenguaje de un AP M_1 por estado de aceptación, entonces existe un AP M_2 por vaciado de pila tal que $N(M_2) = L(M_1)$.

A su vez, si L es el lenguaje que acepta M_2 por vaciado de pila, entonces existe un AP M_1 que acepta L por estado de aceptación: $L(M_1) = L$.

Demostración: como es habitual, primero indicaremos cómo se construye M_2 a partir de M_1 y viceversa. Esto lo haremos en una sola definición. Entonces mostraremos en cada caso que el lenguaje es exactamente el mismo.

Sea

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, \perp, F \rangle$$

un AP que acepta por estado de aceptación ó por vaciado de pila. Sean q'_o y q_e dos estados nuevos (i.e., no están presentes en Q) y sea \top un nuevo símbolo para pila no presente en Γ . Defínase

$$G \stackrel{def}{=} \begin{cases} Q & \text{si } M \text{ acepta por vaciado de pila,} \\ F & \text{si } M \text{ acepta por estado final.} \end{cases}$$

$$\Theta \stackrel{def}{=} \begin{cases} \{\top\} & \text{si } M \text{ acepta por vaciado de pila,} \\ \Gamma \cup \{\top\} & \text{si } M \text{ acepta por estado final.} \end{cases}$$

Considérese el AP:

$$M' = \langle Q \cup \{q_o, q_e\}, \Sigma, \Gamma \cup \{\top\}, \delta', q'_o, \top, \{q_e\} \rangle$$

donde δ' contiene todas las transiciones de δ además de las siguientes transiciones:

$$\begin{aligned} i) \quad & \delta'(q'_o, \epsilon, \top) = \{(q_o, \perp \top)\} \\ ii) \quad & \delta'(q, \epsilon, A) = \{(q_e, A)\}, \quad q \in G, A \in \Theta \\ iii) \quad & \delta'(q_e, \epsilon, A) = \{(q_e, \epsilon)\} \quad A \in \Gamma \cup \{\top\} \end{aligned}$$

M' tiene un nuevo estado inicial q'_o , un nuevo símbolo de pila \top y un nuevo estado final q_e . En el primer paso, por la transición i) empila \perp sobre \top y entonces entra en el estado inicial de M . Entonces hace exactamente lo que M hace ante cualquier entrada $a \in \Sigma$ y las correspondientes configuraciones de la pila. En algún punto entrará al estado q_e de acuerdo a ii); una vez que entra en éste estado puede vaciar toda la pila, coincidiendo así el funcionamiento de aceptación por estado final y por vaciado de pila.

Ahora demostraremos que los lenguajes coinciden: $L(M') = N(M)$. Supondremos primero que M acepta por vaciado de pila. Si $x \in N(M)$ entonces

$$(q_0, x, \perp) \xrightarrow{n}_M (q, \epsilon, \epsilon)$$

para algún n . Podemos descomponer la secuencia de configuraciones que realiza M' al simular a M en

$$\begin{aligned} (q'_o, x, \top) & \longrightarrow_{M'} (q_0, x, \perp \top) \xrightarrow{n}_{M'} (q, \epsilon, \top) \\ & \longrightarrow_{M'} (q_e, \epsilon, \top) \longrightarrow_{M'} (q_e, \epsilon, \epsilon) \end{aligned}$$

lo cual demuestra que $x \in L(M')$.

Ahora, supondremos que M acepta por estado final, $x \in L(M)$:

$$(q_0, x, \perp) \xrightarrow{n}_M (q, \epsilon, \gamma), \text{ donde } q \in F$$

De igual forma, M' al examinar x realiza los siguientes cambios de configuración

$$\begin{aligned} (q'_o, x, \top) & \longrightarrow_{M'} (q_0, x, \perp \top) \xrightarrow{n}_{M'} (q, \epsilon, \gamma \top) \\ & \longrightarrow_{M'} (q_e, \epsilon, \gamma \top) \xrightarrow{*}_{M'} (q_e, \epsilon, \epsilon) \end{aligned}$$

De esa manera, en cualquier caso, M' acepta x . Dado que x es cualquier cadena arbitraria, claramente $L(M) \subseteq L(M')$.

Ahora la otra parte del *sólo si*: supongamos que M' acepta x , lo cual significa que

$$\begin{aligned} (q'_o, x, \top) & \longrightarrow_{M'} (q_0, x, \perp \top) \xrightarrow{n}_{M'} (q, w, \gamma \top) \\ & \longrightarrow_{M'} (q_e, w, \gamma \top) \xrightarrow{*}_{M'} (q_e, \epsilon, \epsilon) \end{aligned}$$

para algún estado $q \in G$. Entonces, $w = \epsilon$ dado que M' no puede leer algún símbolo una vez que entra al estado q_e , por lo tanto tenemos la siguiente transición

$$iv) \quad (q_0, x, \perp) \xrightarrow{n}_{M'} (q_e, \epsilon, \gamma)$$

Considerando las definiciones de G y Θ así como las transiciones ii) que gobiernan el primer movimiento hacia el estado q_e , nos preguntamos cómo es que ocurre la siguiente transición

$$(q, \epsilon, \gamma \top) \longrightarrow_{M'} (q_e, \epsilon, \gamma \top)$$

Si M acepta por vaciado de pila, entonces debe ocurrir que $\gamma = \epsilon$. Por otra parte, si M acepta por estado final, entonces debemos tener que $q \in F$. En cualquier caso, la transición iv) indica que M acepta x . Por tanto, $L(M') \subseteq L(M)$, concluyendo claramente que ambos lenguajes son iguales.

Ahora mostraremos cómo construir un AP a partir de una GLC.

Teorema 7.2.2 (LLC \Rightarrow AP) *Si L es un lenguaje libre de contexto entonces existe un autómata de pila M que acepta L por vaciado de pila: $N(M) = L$.*

Demostración: Sea $G = \langle Vn, Vt, P, S \rangle$ una GLC en forma normal de Greibach tal que $L = L(G)$. Sea $M = \langle \{q\}, Vt, Vn, \delta, q, S, \emptyset \rangle$ un AP, donde δ se define como sigue: para cada producción de G de la forma $A \rightarrow a\gamma$, $\delta(q, a, A)$ contiene a (q, γ) .

Antes de proceder a demostrar que $N(M) = L(G)$, veremos un ejemplo para apreciar cómo el AP M simulará las derivaciones más por la izquierda que realiza G .

Consideremos la siguiente GLC en forma normal de Greibach que genera el lenguaje de expresiones formadas con paréntesis balanceados, a la derecha de cada producción se muestra la transición correspondiente en el AP como se especificó:

$$\begin{aligned} i) \quad & S \rightarrow '(BS & \delta(q, '(S) = \{(q, BS)\} \\ ii) \quad & S \rightarrow '(B & \delta(q, '(S) = \{(q, B)\} \\ iii) \quad & S \rightarrow '(SB & \delta(q, '(S) = \{(q, SB)\} \\ iv) \quad & S \rightarrow '(SBS & \delta(q, '(S) = \{(q, SBS)\} \\ v) \quad & B \rightarrow ') & \delta(q, ') = \{(q, \epsilon)\} \end{aligned}$$

Para cualquier secuencia de formas de frase en la derivación más por la izquierda para una cadena de

entrada x corresponde una secuencia de configuraciones en el AP M ; por ejemplo, sea $x = “(((())())”$

Regla	formas de frase	configuración de M
	S	$(q, “(((())())” S)$
iii)	$(SB$	$(q, “(())()” SB)$
iv)	$((SBSB$	$(q, “()()()” SBSB)$
ii)	$((BBSB$	$(q, “”)()” BBSB)$
v)	$((()BSB$	$(q, “()()” BSB)$
v)	$((())SB$	$(q, “()” SB)$
ii)	$((())(BB$	$(q, “”)” BB)$
v)	$((())()B$	$(q, “”) B)$
v)	$((())())$	$(q, \epsilon \epsilon)$

En la secuencia de derivaciones (segunda columna) la cadena x se genera de izquierda a derecha, un terminal a cada paso de derivación; de la misma forma que la cadena x se examina por el AP: la forma de frase en cada paso de la derivación coincide con el contenido de la pila en la configuración correspondiente.

Vamos a formalizar la observación anterior en un lema general que relaciona las formas de frase en cada paso de una derivación más por la izquierda de x bajo la GLC G y las configuraciones de M al aceptar la entrada x .

Lema 7.2.1 *Para cualesquiera cadenas $z, w \in Vt^*$, $\gamma \in Vn^*$ y $A \in Vn$, A deriva más a la izquierda a $z\gamma$ en n pasos si y sólo si $(q, zw, A) \xrightarrow{n}_M (q, w, \gamma)$.*

(Del ejemplo anterior, en el cuarto renglón, podemos identificar que $z = “(((“$, $w = “)()”$, $\gamma = BBSB$, $A = S$ y $n = 3$).

Demostración: aplicaremos inducción sobre n :

- Caso base: $n = 0$ tenemos

$$\begin{aligned}
 A \Rightarrow_0 z\gamma &\iff A = z\gamma \\
 &\iff z = \epsilon \text{ y } \gamma = A \\
 &\iff (q, zw, A) = (q, w, \gamma) \\
 &\iff (q, zw, A) \xrightarrow{0}_M (q, w, \gamma).
 \end{aligned}$$

- Para el paso inductivo conviene separar el *si* y *sólo si* en su dos implicaciones respectivas:

\Rightarrow) consideremos primero una derivación más por la izquierda $A \xrightarrow{*}_{n+1} z\gamma$. Sea $B \rightarrow c\beta$ la última producción aplicada, donde $c \in Vt \cup \{\epsilon\}$ y $\beta \in Vn^*$, entonces

$$A \xrightarrow{*}_n uB\alpha \Rightarrow uc\beta\alpha = z\gamma$$

donde $z = uc$ y $\gamma = \beta\alpha$. Por la HI

$$(q, ucw, A) \xrightarrow{n}_M (q, cw, B\alpha).$$

Por la definición de M , sabemos que

$$\delta(q, c, B) = \{(q, \beta)\}$$

de tal suerte que

$$(q, cw, B\alpha) \xrightarrow{1}_M (q, w, \beta\alpha).$$

Combinando las anteriores expresiones tenemos que

$$(q, zw, A) = (q, ucw, A) \xrightarrow{n+1}_M (q, w, \beta\alpha) = (q, w, \gamma).$$

\Leftarrow) ahora sea la siguiente secuencia de $n + 1$ transiciones en M :

$$(q, zw, A) \xrightarrow{n+1}_M (q, w, \gamma)$$

y sea la última transición aplicada:

$$\delta(q, c, B) = \{(q, \beta)\}$$

Entonces $z = uc$ para algún $u \in Vt^*$, $\gamma = \beta\alpha$ para alguna $\alpha \in \Gamma^*$ y

$$(q, ucw, A) \xrightarrow{n}_M (q, cw, B\alpha) \longrightarrow_M (q, w, \beta\alpha).$$

Por la HI, se tiene la siguiente derivación más por la izquierda

$$A \Rightarrow_n uB\alpha$$

y por la construcción de M , se tiene la siguiente producción en G

$$B \rightarrow c\beta$$

Aplicando tal producción a la forma de frase $uB\alpha$ se obtiene

$$A \Rightarrow_n uB\alpha \Rightarrow uc\beta\alpha = zw$$

mediante una derivación más por la izquierda.

Ya sólo resta mostrar, auxiliándonos del lema anterior, que $N(M) = L(G)$:

$$\begin{aligned}
 x \in L(G) &\iff S \xrightarrow{*}_{mi} x && \text{def. de } L(G) \\
 &\iff (q, x, S) \xrightarrow{*}_M (q, \epsilon, \epsilon) && \text{lema anterior} \\
 &\iff x \in N(M) && \text{def. de } N(M).
 \end{aligned}$$

Ejercicios 7.2.1 *Obtenga los correspondientes AP para las gramáticas del ejercicio 6.1.3, pág 48.*

Teorema 7.2.3 (AP \Rightarrow LL) *Si L es un lenguaje aceptado por un autómata de pila M por vaciado de pila ($N(M) = L$) entonces L es libre de contexto.*

Demostración: realizaremos ésta prueba en dos pasos:

1. todo AP puede simularse por un AP con un sólo estado,
2. todo AP con un sólo estado tiene una GLC correspondiente.

El segundo paso está prácticamente completado por el teorema anterior ya que la construcción que se describió es *invertible*.

Sea $M = \langle \{q\}, \Sigma, \Gamma, \delta, q, \perp, \emptyset \rangle$ un AP que acepta cadenas por vaciado de pila. Definase la GLC $G = \langle \Gamma, \Sigma, P, \perp \rangle$ donde P contiene una producción de la forma $A \rightarrow c\gamma$ para cada transición $\delta(q, c, A) = \{(q, \gamma)\}$ donde $c \in \Sigma \cup \{\epsilon\}$. Aplicando el teorema anterior tal como está, se tiene que $L(G) = N(M)$.

Sólo queda simular un AP arbitrario mediante un AP con un sólo estado. Esencialmente, mantendremos encapsulada toda la información de estado del AP en la pila. Por la construcción del teorema 7.2.1 podemos asumir que partimos de un AP M de la forma $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \perp, \{q_f\} \rangle$ y vacía su pila una vez que entra al estado q_f .

Sea $\Gamma' \stackrel{\text{def}}{=} Q \times \Gamma \times Q$, los elementos de Γ' son tuplas $[p, A, q]$ que encapsulan la información de estados. Construimos un nuevo AP M' :

$$M' = \langle \{q_*\}, \Sigma, \Gamma', \delta', q_*, [q_0, \perp, q_f], \emptyset \rangle$$

que acepta por vaciado de pila. M' puede examinar una cadena x comenzando con $[p, A, q]$ en su pila y finalizar con la pila vacía *si y sólo si* M puede examinar a x comenzando en el estado p con A en la pila y terminar en el estado q con la pila nula.

La relación de transición δ' de M' se define como sigue: para cada transición definida en δ :

$$\delta(p, c, A) = \{(q_0, B_1 B_2 \dots B_k)\}$$

(donde $c \in \Sigma \cup \{\epsilon\}$) incluir en δ' las transiciones

$$\delta'(q_*, c, [p, A, q_k]) = \{(q_*, [q_0, B_1, q_1] \dots [q_{k-1}, B_k, q_k])\}$$

para todas las posibles opciones de q_1, q_2, \dots, q_k . Para $k = 0$, lo anterior se reduce a:

$$\begin{aligned} \text{Si } \delta(p, c, A) &= \{(q_0, \epsilon)\} \text{ entonces} \\ \delta'(q_*, c, [p, A, q_0]) &= \{(q_*, \epsilon)\} \end{aligned}$$

Intuitivamente, M' simula a M suponiendo no determinísticamente en qué estados estará M en ciertos puntos futuros salvando tales suposiciones en la pila y entonces verificando después si esas suposiciones fueron correctas. El siguiente lema formaliza tal relación entre los cálculos de M y M' :

Lema 7.2.2 *Sea M' un AP construido a partir de M como se indicó. Entonces*

$$(p, x, \gamma) \xrightarrow{n}_M (q, \epsilon, \epsilon) \text{ (donde } \gamma = B_1 B_2 \dots B_k)$$

si y sólo si existen estados q_0, q_1, \dots, q_k tales que $p = q_0, q = q_k$ y

$$(q_*, [q_0 B_1 q_1] \dots [q_{k-1} B_k q_k]) \xrightarrow{n}_{M'} (q_*, \epsilon, \epsilon)$$

En particular,

$$(p, x, B) \xrightarrow{n}_M (q, \epsilon, \epsilon)$$

$$\Downarrow$$

$$(q_*, [q_0 B_1 q_1] \dots [q_{k-1} B_k q_k]) \xrightarrow{n}_{M'} (q_*, \epsilon, \epsilon)$$

Demostración: aplicaremos inducción sobre n

- Caso base: $n = 0$, ambos lados son equivalentes a la afirmación de que $p = q, x = \epsilon$ y $k = 0$.

- Paso inductivo: supóngase ahora que se tiene la siguiente secuencia de $n + q$ transiciones

$$(p, x, \gamma) \xrightarrow{n+1}_M (q, \epsilon, \epsilon) \text{ (donde } \gamma = B_1 B_2 \dots B_k)$$

Sea $\delta(p, c, B_1) = \{(r, C_1 C_2 \dots C_m)\}$ la primer transición aplicada, donde $c \in \Sigma \cup \{\epsilon\}$ y $m \geq 0$. Entonces $x = cw$ y se tienen las siguientes transiciones:

$$\begin{aligned} (p, x, B_1 B_2 \dots B_k) &\xrightarrow{\quad}_M (r, w, C_1 C_2 \dots B_2 \dots B_k) \\ &\xrightarrow{n}_M (q, \epsilon, \epsilon) \end{aligned}$$

Por la HI, existen $r_0, r_1, \dots, r_{m-1}, q_1, q_2, \dots, q_k$ tales que $r = r_0, q = q_k$ y

$$\begin{aligned} (q_*, w, [r_0 C_1 r_1] [r_1 C_2 r_2] \dots [r_{m-1} C_m q_1] [q_1 B_2 q_2] \\ \dots [q_{k-1} B_k q_k]) \\ \xrightarrow{n}_M (q_*, \epsilon, \epsilon) \end{aligned}$$

y también por la construcción de M' se tiene

$$\delta'(q_*, c, [p B_1 q_1]) = \{(q_*, [r_0 C_1 r_1] [r_1 C_2 r_2] \dots [r_{m-1} C_m q_1])\}$$

Combinándolas, obtenemos

$$\begin{aligned} (q_*, x, B_1 B_2 \dots B_k) \\ \xrightarrow{\quad}_M (r_0, w, C_1 C_2 \dots C_m B_2 \dots B_k) \\ \xrightarrow{n}_M (q_*, \epsilon, \epsilon) \end{aligned}$$

Finalmente,

Teorema 7.2.4 ($L(M) = L(M')$)

Demostración: para toda $x \in \Sigma^*$:

$$\begin{aligned} x \in L(M') &\iff (q_*, x, (q_0 \perp q_f)) \xrightarrow{*}_M (q_*, \epsilon, \epsilon) \\ &\iff (q_0, x, \perp) \xrightarrow{*}_M (q_f, \epsilon, \epsilon) \\ &\iff x \in L(M). \end{aligned}$$

Propiedades de los Lenguajes Libres de Contexto

Ya hemos examinado el Lema de sondeo para los lenguajes libres de contexto, por lo que en éste capítulo revisaremos algunas de las propiedades de cerradura más importantes para los lenguajes libres de contexto, así como algoritmos de decisión. Finalizaremos el capítulo revisando el aspecto más sobresaliente de los Lenguajes Libres de Contexto: su aplicación en la fase de análisis sintáctico (*parsing*) en el diseño y construcción de compiladores.

8.2 Propiedades de Cerradura

Teorema 8.2.1 *Los LLC son cerrados bajo unión, concatenación y cerradura Kleene.*

Demostración: Para los tres casos consideremos que A y B son LLC generados por las GLC $G_1 = \langle Vn_1, Vt_1, P_1, S_1 \rangle$ y $G_2 = \langle Vn_2, Vt_2, P_2, S_2 \rangle$ respectivamente.

Podemos formar la GLC que genere $A \cup B$ combinando todas las producciones de G_1, G_2 junto con un nuevo símbolo inicial S y las nuevas producciones $S \rightarrow S_1 \mid S_2$. Hay que asegurarse que G_1 y G_2 tengan sus conjuntos de no terminales disjuntos entre sí, de lo contrario simplemente hay que renombrarlos. La nueva GLC $A \cup B$ tiene la forma: $G_{A \cup B} = \langle Vn_1 \cup Vn_2 \cup \{S\}, Vt_1 \cup Vt_2, P, S \rangle$ donde $P = P_1 \cup P_2 \cup \{S \rightarrow S_1 \mid S_2\}$.

Para la concatenación, podemos formar la GLC que genere a AB combinando G_1 y G_2 mediante la producción $S \rightarrow G_1 G_2$.

Y para la cerradura Kleene, para generar A^* extendemos a G_1 con una nueva producción $S \rightarrow S_1 S \mid \epsilon$.

Teorema 8.2.2 *Los LLC son cerrados bajo intersección con conjuntos regulares.*

Demostración: Sea A libre de contexto y R un lenguaje regular, se puede demostrar que $A \cap R$ es libre de contexto mediante la construcción de producto involucrando a un AP M_A para A y a un AFD M_R para R (similar a cuando se demostró

que la intersección de dos lenguajes regulares es regular) de tal forma que la máquina M' asociada a $A \cap R$ simula *en paralelo* el comportamiento de M_A y M_R : si $M_A = \langle Q_A, \Sigma, \Gamma, \delta_A, q_0, \perp, F_A \rangle$ y $M_R = \langle Q_R, \Sigma, \delta_R, p_0, F_R \rangle$ construimos a $M' = \langle Q_A \times Q_R, \Sigma, \Gamma, \delta, [p_0, q_0], \perp, F_A \times F_R \rangle$. M' simula los movimientos de M_A ante la entrada ϵ sin cambiar el estado de M_R , cuando M' realiza un movimiento ante el símbolo de entrada a , significa que tanto M_A como M_R realizan sus movimientos correspondientes. M' acepta una cadena w si y sólo si tanto M_R como M_A aceptan w . Para ello, M' tiene como estados paquetes que contienen la pareja de estados para M_R y M_A . La función de transición para M' $\delta([p, q], a, X)$ contiene a $([p', q'], \gamma)$ si y sólo si $\delta_R(p, a) = p'$ y $\delta_A(q, a, X)$ contiene a (q', γ) . Para completar la prueba, el lector debe demostrar por inducción en i (el número de pasos) que

$$\begin{aligned} ([p_0, q_0], w, \perp) &\xrightarrow{i}_{M'} ([p, q], \epsilon, \gamma) \\ &\text{si y sólo si} \\ (q_0, w, \perp) &\xrightarrow{i}_{M_A} (q, \epsilon, \gamma) \text{ y } \delta_R(p_0, w) = p \end{aligned}$$

Teorema 8.2.3 *Los LLC son cerrados bajo sustitución y homomorfismo.*

Demostración: Sea L un LLC definido sobre Σ y generado por la GLC $G = \langle Vn, Vt, P, S \rangle$ ($L(G) = L$). Para cada $a \in \Sigma$, sea L_a un LLC generado a su vez por una GLC G_a . Asúmase que los símbolos de G y de cada G_a son distintos. Construiremos una GLC G' que generará el lenguaje sustitución de L_a en L : los no terminales de G' son los no terminales de G junto con los no terminales de cada G_a ; mientras que los terminales de G' son los terminales de las G_a 's. El símbolo inicial de G' es S , las producciones de G' son todas las producciones de cada G_a junto con aquellas producciones formadas tomando una producción $A \rightarrow \alpha$ de P y substituyéndola por S_a (el símbolo inicial de G_a , para cada instancia de $A \in \Sigma$ que ocurra en α).

Ejemplo 8.2.1 *Sea L es conjunto de cadenas con*

igual número de a 's que de b 's, y sean $L_a = \{0^n 1^n \mid n \geq 1\}$ y $L_b = \{ww^R \mid w \in (\mathbf{0} + \mathbf{2})^*\}$.

Para G podemos elegir las siguientes producciones

$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

Para G_a : $S_a \rightarrow 0S_a1 \mid 01$

Para G_b : $S_b \rightarrow 0S_b0 \mid 2S_b2 \mid \epsilon$

Si h es la substitución $h(a) = L_a$ y $h(b) = L_b$ entonces $h(L)$ se genera por la siguiente GLC:

$$S \rightarrow S_a S S_b S \mid S_b S S_a S \mid \epsilon \quad \begin{array}{l} S_a \rightarrow 0S_a1 \mid 01 \\ S_b \rightarrow 0S_b0 \mid 2S_b2 \mid \epsilon \end{array}$$

Dado que un homomorfismo es un tipo especial de substitución, se sigue inmediatamente que los LLC son cerrados bajo homomorfismo. Como para el caso de los lenguajes regulares, los LLC también son cerrados bajo homomorfismo inverso: la prueba²⁵ consiste en construir una máquina (AP) que reconozca el lenguaje de h^{-1} .

Sin embargo, existen varias propiedades de cerradura de los lenguajes regulares que los lenguajes libres de contexto no tienen, entre ellos intersección y complemento.

Teorema 8.2.4 *Los LLC no son cerrados bajo intersección.*

Demostración: la construcción del producto no funciona para LLC pues no hay manera de simular dos pilas independientes con una sola pila. Para ésta demostración basta con dar un contraejemplo: el lenguaje $\{a^n b^n c^n \mid n \geq 0\}$ que no es libre de contexto se puede obtener a partir de dos LLC:

$$\begin{aligned} \{a^n b^n c^n \mid n \geq 0\} = \\ \{a^m b^m c^m\} \cap \{a^m b^n c^n\} \\ (\text{en ambos casos, } m, n \geq 0) \end{aligned}$$

Corolario 8.2.1 *Los LLC no son cerrados bajo complemento.*

Demostración: dado que sabemos que los LLC son cerrados bajo unión, si fueran cerrados bajo complemento entonces por leyes de De Morgan podríamos expresar para L_1, L_2 LLC: $L_1 \cap L_2 = \sim(\sim L_1 \cup \sim L_2)$. Pero eso significa que los LLC serían cerrados bajo intersección, lo cual contradice el resultado del teorema anterior. Por lo tanto, no son cerrados bajo complemento.

²⁵Para detalles, consultar [HU79]

8.3 Algoritmos de decisión para LLC

Existen preguntas acerca de los lenguajes libres de contexto que no pueden resolverse en general²⁶, entre ellas están cuándo dos GLC's son equivalentes, cuándo el complemento de un LLC es también LLC, y cuándo una GLC arbitraria es ambigua. A continuación discutimos las preguntas para las que sí se pueden plantear algoritmos.

Infinitud, finitud y vacuidad

Teorema 8.3.1 *Existen algoritmos de decisión para determinar si un lenguaje libre de contexto es vacío, finito ó infinito.*

Demostración: como en el caso de los lenguajes regulares, ésta prueba se puede hacer utilizando del lema de sondeo, en cuyo caso el algoritmo resultante es tremendamente ineficiente. De hecho, en la parte de simplificación de GLC se ofreció un algoritmo eficiente para verificar cuándo un LLC es vacío: $L(G)$ es no vacío *si y sólo si* partiendo del símbolo inicial se genera alguna frase.

Para verificar si $L(G)$ es finito, aplicamos el algoritmo para obtener la forma normal de Chomsky G' de G sin símbolos inútiles que genera $L(G) - \{\epsilon\}$. El lenguaje de G' es finito *si y sólo si* $L(G)$ también lo es: la manera más sencilla de verificar finitud de G' es dibujar un grafo dirigido teniendo por nodo cada no terminal y arista entre A y B si existe alguna producción de la forma $A \rightarrow BC$ ó $A \rightarrow CB$ para cualquier C (i.e., dibujar un *grafo de alcanzabilidad*). $L(G')$ es finito *si y sólo si* éste grafo no tiene ciclos:

\Rightarrow si el grafo tiene un ciclo $A_0, A_1, \dots, A_n, A_0$ entonces se puede derivar

$$A_0 \Rightarrow \alpha_1 A_1 \beta_1 \Rightarrow \dots \Rightarrow \alpha_n A_n \beta_n \Rightarrow \alpha_{n+1} A_0 \beta_{n+1}$$

para α_i, β_i formas de frase conformadas únicamente por no terminales y $|\alpha_i \beta_i| = i$.

Dado que no hay símbolos inútiles, entonces $\alpha_{n+1} \xrightarrow{*} w$ y $\beta_{n+1} \xrightarrow{*} x$ para algunas frases w, x con longitud de al menos $n+1$. w y x no pueden ambas ser ϵ pues $n \geq 0$. De nuevo, gracias a que no hay símbolos inútiles podemos encontrar palabras u y z tales que $S \xrightarrow{*} u A_0 z$ y una palabra v tal que $A_0 \xrightarrow{*} v$. Entonces, para toda i

$$\begin{aligned} S \xrightarrow{*} u A_0 z &\xrightarrow{*} u w A_0 x z \xrightarrow{*} u w^2 A_0 x^2 z \xrightarrow{*} \dots \\ &\xrightarrow{*} u w^i v x^i z \end{aligned}$$

²⁶Son problemas *indecidibles* en el sentido que no existe un algoritmo que las resuelva para cualquier entrada posible.

Como $|wx| > 0$ entonces $uw^i vx^j z$ no puede ser igual a $uw^j vx^i z$ si $i \neq j$; por lo tanto G' deriva un número infinito de palabras.

⇐) Ahora supongamos que el grafo de G' no tiene ciclos. Definamos el *rango* de un noterminal A como la longitud de la ruta más larga en el grafo que comienza con A . Dado que no hay ciclos, entonces el rango de A es finito. Si se tiene $A \rightarrow BC$ entonces el rango de B y C son menores estrictamente al rango de A . El lector debe aplicar inducción sobre el rango r de A para demostrar que ninguna frase derivada de A tiene longitud mayor que 2^r , mostrando así que $L(G')$ es finito.

Ejemplo 8.3.1 *Considérese a la GLC G en FNC cuyas producciones se muestran*

$$\begin{array}{l|l} S \rightarrow AB & A \rightarrow BC \\ B \rightarrow CC & C \rightarrow a \end{array}$$

El grafo de alcanzabilidad se muestra en la Figura 8.35(izquierda). Ese grafo no tiene ciclos, los rangos de S, A, B y C son 3, 2, 1 y 0 respectivamente. Dado que la ruta más larga es $S \rightarrow A \rightarrow B \rightarrow C$, G no deriva cadenas de longitud mayor a $2^3 = 8$. De hecho la cadena más larga generada por S es

$$S \Rightarrow_{mi} AB \Rightarrow BCB \Rightarrow CCCB \Rightarrow CCCCC \xRightarrow{*}_{mi} aaaad$$

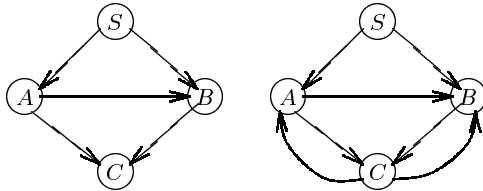


Figura 8.35: Grafos de alcanzabilidad correspondientes a G en forma normal de Chomsky, y su extensión G'

Si incorporamos la producción $C \rightarrow AB$ entonces obtenemos el grafo de la derecha, el cual tiene varios ciclos, como por ejemplo $A \rightarrow B \rightarrow C \rightarrow A$. Por tanto, podemos hallar una derivación $A \xRightarrow{*} \alpha_3 A \beta_3$, en particular $A \Rightarrow BC \Rightarrow CCC \Rightarrow CABC$ donde $\alpha_3 = C$ y $\beta_3 = BC$. Dado que $C \xRightarrow{*} a$ y $BC \xRightarrow{*} ba$ tenemos que $A \xRightarrow{*} aAba$. Por lo tanto $S \xRightarrow{*} Ab$ y $A \xRightarrow{*} a$ tenemos que $S \xRightarrow{*} a^i a(ba)^i b$ para toda $i \geq 0$, lo cual significa que el lenguaje correspondiente es infinito.

8.4 Parsing

Dado un lenguaje libre de contexto A y una cadena sobre un alfabeto $x \in \Sigma^*$, ¿cómo podemos decir si $x \in A$? Éste es el primer problema al que se enfrenta un compilador: indicar si el archivo de entrada que se recibe (código fuente) está sintácticamente bien escrito. A ésta fase se le denomina *Análisis Sintáctico*, la cual es fundamental para las demás fases de la compilación. Si tenemos una GLC G para A podemos convertir a G en forma normal de Chomsky y entonces intentar todas las posibles derivaciones de longitud $2|x| - 1$ para ver si alguna de ellas produce a x . Evidentemente, tal estrategia es con mucho ineficiente, pues en general se tiene un número exponencial de las derivaciones. Si producimos un AP correspondiente de nuevo tenemos un número exponencial en las secuencias de cómputo (debido al no determinismo).

A continuación discutiremos el algoritmo conocido como CYK propuesto por Cocke, Kasami y Younger. El desempeño de CYK es de *tiempo cúbico* y es un ejemplo de la técnica de *programación dinámica*. La idea básica es la siguiente: determina para cada subcadena w de x el conjunto de todos los no terminales que genera a w . Este proceso se realiza inductivamente sobre la longitud de w .

Por sencillez, asumamos que la GLC G está en forma normal de Chomsky. Ilustraremos el funcionamiento de CYK con el siguiente ejemplo: sea la GLC que genera el lenguaje de las cadenas no nulas con igual número de a 's que de b 's:

$$\begin{array}{l|l} S \rightarrow AB \mid BA \mid SS \mid AC \mid BD, & B \rightarrow b \\ A \rightarrow a & D \rightarrow SA \\ C \rightarrow SB & \end{array}$$

Ejecutaremos CYK sobre la cadena de entrada $x = aabbab$. Sea $n = |x|$ (en éste caso $n = 6$). Dibujaremos $n + 1$ rayas que separan cada una de las letras de x y las numeramos desde 0 a n :

$$\begin{array}{c} | a | a | b | b | a | b | \\ 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \end{array}$$

Para $0 \leq i < j \leq n$ denotaremos por $x_{i,j}$ la subcadena de x entre las rayas i y j . En éste ejemplo $x_{1,4} = abb$ y $x_{2,6} = bbab$. La cadena completa x es $x_{0,6}$. Construyamos una tabla T con $\binom{n}{2}$ entradas, una para cada pareja i, j :

0						
-	1					
-	-	2				
-	-	-	3			
-	-	-	-	4		
-	-	-	-	-	5	
-	-	-	-	-	-	6

Cada entrada i, j de T (denotada por $T_{i,j}$) se refiere a la subcadena $x_{i,j}$. El propósito del algoritmo es llenar cada entrada $T_{i,j}$ con el conjunto de no terminales de G que generen a la subcadena $x_{i,j}$. Para ello procedemos inductivamente (las cadenas más chicas primero). Comenzamos con subcadenas de longitud 1, las cuales son de la forma $x_{i,i+1}$ (para $0 \leq i \leq n-1$) y corresponden a las entradas que están sobre la diagonal superior. Si existe una producción $X \rightarrow x_{i,i+1}$ entonces ponemos al noterminal X en la entrada $T_{i,i+1}$:

0						
A	1					
-	A	2				
-	-	B	3			
-	-	-	B	4		
-	-	-	-	A	5	
-	-	-	-	-	B	6

En el ejemplo, B se pone en $T_{3,4}$ debido a que $x_{3,4} = b$ y se tiene la producción $B \rightarrow b$. En general, $T_{i,j}$ puede contener varios no terminales pues pueden haber varias producciones que deriven a $x_{i,j}$.

Ahora procedemos con subcadenas de longitud 2, las cuales corresponden a la diagonal inmediatamente bajo la que acabamos de llenar: para cada subcadena $x_{i,i+2}$ la descomponemos a su vez en dos subcadenas no nulas $x_{i,i+1}$ y $x_{i+1,i+2}$ de longitud 1 cada una y verificamos las entradas correspondientes a tales subcadenas $T_{i,i+1}$ y $T_{i+1,i+2}$ (las cuales ocurren inmediatamente por arriba y a la derecha de $T_{i,i+2}$). Seleccionamos un no terminal de cada una de tales entradas (e.g, X de $T_{i,i+1}$ y Y de $T_{i+1,i+2}$) y buscamos si existen producciones de la forma $Z \rightarrow XY$ en G . Para cada producción que encontremos colocamos a Z en la entrada $T_{i,i+2}$, y hacemos esto para todas las posibles opciones de $X \in T_{i,i+1}$ y $Y \in T_{i+1,i+2}$.

En el ejemplo, para $x_{0,2} = aa$ encontramos que $A \in T_{0,1}$ y $A \in T_{1,2}$ por lo que buscamos alguna producción que tenga a AA en su parte derecha. Como no hay, ponemos \emptyset en $T_{0,2}$. Para $T_{1,3}$ encontramos a A inmediatamente encima y a B a la derecha, por lo que buscamos una producción que tenga a AB en la parte derecha: si hay y es $S \rightarrow AB$ por lo que ponemos a S en $T_{1,3}$. Continuamos de ésta manera hasta llenar la diagonal.

0						
A	1					
\emptyset	A	2				
-	S	B	3			
-	-	\emptyset	B	4		
-	-	-	S	A	5	
-	-	-	-	S	B	6

Continuamos de esa manera con las subcadenas de longitud 3, 4, etc. El resultado final es:

0						
A	1					
\emptyset	A	2				
\emptyset	S	B	3			
S	C	\emptyset	B	4		
D	S	\emptyset	S	A	5	
S	C	\emptyset	C	S	B	6

Vemos que la entrada $T_{0,6}$ tiene a S lo cual significa que

$$S \xRightarrow{*} x_{0,6} = x$$

por lo que concluimos que x es generada por G .

A continuación se muestra el algoritmo CYK.

Algoritmo 5 CYK

Entrada: $GLC\ G = \langle Vn, Vt, P, S \rangle, x$

Salida: x es generada por G o no.

```

1: for  $i = 0$  to  $n - 1$  do
2:    $T_{i,i+1} := \emptyset$ ;
3:   for  $A \rightarrow a$  do
4:     if  $a = x_{i,i+1}$  then
5:        $T_{i,i+1} := T_{i,i+1} \cup \{A\}$ 
6:     end if
7:   end for
8: end for
9: for  $m = 2$  to  $n$  do
10:  for  $i = 0$  to  $n - m$  do
11:     $T_{i,i+m} := \emptyset$ ;
12:    for  $j = i + 1$  to  $i + m - 1$  do
13:      for  $A \rightarrow BC$  do
14:        if  $B \in T_{i,j}$  y  $C \in T_{j,i+m}$  then
15:           $T_{i,i+m} := T_{i,i+m} \cup \{A\}$ 
16:        end if
17:      end for
18:    end for
19:  end for
20: end for
```

El desempeño de CYK puede describirse de la siguiente manera: como G es fija, el paso 2 toma un tiempo constante para su ejecución, por tanto, los pasos 1 y 2 toman tiempo $O(n)$ (lineal). Los ciclos anidados de los pasos 9 y 10 causan que los pasos dentro de ellos (11 a 15) sean ejecutados a lo más n^2

veces. El paso 11 toma tiempo constante, mientras que los pasos 12 y 13 hacen que el paso 14 sea ejecutado n o menos veces; por lo tanto al ser ejecutados $O(n^2)$ veces, el tiempo total en el paso 14 es $O(n^3)$, que es el tiempo total del algoritmo.

Ejercicios 8.4.1 Aplique el algoritmo CYK para determinar si las cadenas $w = aaaaa$ y $v = aaaaaa$ son generadas por la GLC del ejemplo 8.3.1.

Fundamentalmente, el trabajo del analizador sintáctico consiste en construir (implícita ó explícitamente) el correspondiente AAS para la expresión de entrada. Los analizadores sintácticos se clasifican de acuerdo a las clases de gramáticas que pueden reconocer: LL (*left most derivation, left to right scanning*) el cual, a partir de un examen de la entrada de izquierda a derecha intenta construir una derivación más por la izquierda. La gramática debe no ser ambigua y de tal forma que viendo el símbolo terminal actual de la cadena de entrada pueda elegir inmediatamente una producción correspondiente.

Muchas GLC no son LL, por lo que existe otro orden: LR (*left to right scanning, right most derivation*) en la cual el propósito es construir una derivación más por la derecha en orden inverso. De ésta clase de gramáticas (y por ende de los analizadores sintácticos correspondientes) se desprende toda una jerarquía de acuerdo al número de *símbolos por anticipado* que se examinen antes de elegir alguna producción. A toda ésta familia de analizadores sintácticos se les denomina genéricamente *por desplazamiento y reducción*, en el sentido que trabajan con una tabla y una pila (de manera muy similar a un AP). La tabla consta de dos partes, una función *acción* que indica que deba hacerse (desplazar ó reducir) y la función *ir_a* que indica las transiciones entre estados. El programa que maneja el analizador sintáctico se comporta de la siguiente manera: determina s_m el estado en el tope de la pila y el símbolo a_i de entrada actual. Consulta la tabla en posición $acción[s_m, a_i]$ la cual puede tener uno de los cuatro valores:

1. *desplazar* un estado s , (implícitamente, se avanza leyendo más símbolos de entrada)
2. *reducir* el tope de la pila por una producción $A \rightarrow \beta$ (significa que se ha visto una cadena que se puede derivar por A)
3. aceptar (éxito)
4. indicación de error sintáctico.

La diferencia entre tipos de analizadores LR radica en la construcción de la tabla, generando tablas de pocas entradas (SLR), LR canónico y LALR que es

el más utilizado y produce tablas compactas. Gracias a que éste proceso de construcción de analizadores sintácticos puede automatizarse (dada una GLC, obtener el programa analizador) se disponen de herramientas denominadas *compilador de compiladores* como Yacc en Un*x (bison para GNU) que ofrecen código en C, las cuales pueden tratar inclusive gramáticas ambiguas.

Lenguajes inherentemente ambiguos

Para finalizar éste capítulo, mencionamos un resultado que establece la existencia de LLC inherentemente ambiguos, i.e., aquellos para los cuales toda GLC es ambigua. Es sencillo mostrar GLC ambiguas para cualquier lenguaje, recordemos que la ambigüedad indica que en un paso de la derivación de una frase hay más de una posibilidad (producción) que podemos aplicar. La demostración de existencia de LLC inherentemente ambiguos es bastante tediosa, por lo que se remite al lector a [HU79] ya que no se hará uso de tal resultado.

Máquinas de Turing y Computabilidad Efectiva

Máquinas de Turing

En la búsqueda por la definición de proceso efectivo ó algoritmo, comenzamos revisando el modelo mecánico más sencillo (los autómatas finitos), a continuación mostramos sus limitaciones y revisamos el modelo de los autómatas de pila y los lenguajes libres de contexto. No fué sorpresa alguna que tal modelo extienda naturalmente a los autómatas finitos. Terminaremos nuestra jornada en la búsqueda de una definición de la noción *computable* presentando el modelo más poderoso en la jerarquía: la Máquina de Turing (MT). Tampoco será sorpresa que esencialmente la Máquina de Turing es un autómata sin restricción alguna: con una cinta con capacidad de almacenamiento infinito, con la capacidad de leer y escribir sobre la cinta y de mover la cabeza lectora/escritora en cualquier dirección sobre la cinta y no solamente de izquierda a derecha. Lo que será sorprendente son los resultados que discutiremos para precisar la noción de computabilidad *universal*.

La noción intuitiva de procedimiento efectivo ó algoritmo ha tenido lugar muchas veces en las discusiones de los capítulos previos: hemos presentado algoritmos que permiten mostrar la equivalencia entre AFND y AFD, si el lenguaje aceptado por un AFD es finito, vacío, infinito; equivalencia entre ER y AF, minimización de estados de un AF, etc. Muchos de tales procedimientos son algoritmos pues siempre terminan ante cualquier entrada dada, muchos de tales algoritmos son *recursivos* en el sentido que se sustentan en una definición constructiva: aplicamos el principio de inducción, y el algoritmo tiene claramente una parte que corresponde al caso base y una parte que corresponde al paso inductivo sobre el cual se realizan invocaciones recursivas. Por tanto, hemos establecido a lo largo del curso una correspondencia entre el concepto de recursividad y la inducción matemática. Continuaremos enfatizando éste aspecto ya que las demostraciones por inducción, además de elegantes, nos ofrecen la construcción de un algoritmo recursivo correspondiente.

Hoy en día se considera que el modelo de Máquina de Turing es la formalización del concepto de procedimiento efectivo y por tanto de la noción de computabilidad. Otros formalismos que se han propuesto a lo largo de la historia se ha demostrado que

pueden expresarse (*codificarse*) en la MT, por tanto a pesar de sus aparentes diferencias todos esos formalismos son computacionalmente equivalentes. Ésta afirmación conduce a que la noción de computabilidad captura justamente al modelo de computadora física que se dispone hoy en día: todo problema que pueda plantearse como un algoritmo es computable. Tal es en esencia la *Tesis de Church-Turing*, la cual, a pesar de no poderse demostrar formalmente (i.e., no es un teorema) declara que justamente todos los formalismos (Máquinas de Turing, λ -calculus, funciones μ -recursivas, etc.) son equivalentes. Si preferimos enfocarnos al modelo de la MT es debido a que fué el primer modelo que puede considerarse completamente programable y es una extensión natural a los modelos mecánicos que hemos discutido. Aún cuando se puede argumentar que existen aspectos de la computación que no son contempladas por el modelo de la MT (e.g., cómputo aleatorio, interactivo, etc.), lo que no se puede objetar es que la noción de procedimiento efectivo es capturado por la MT de manera robusta.

Una vez que la noción de procedimiento efectivo (algoritmo) fué formalizado, se demostró que no existen algoritmos para computar muchas funciones; la existencia de tales funciones puede mostrarse a partir de un argumento que involucra *enumeración* (conteo) más adelante demostraremos que la clase de todos los algoritmos es enumerable. Por otra parte, en el capítulo de los preliminares se demostró que el conjunto potencia de los naturales no es numerable; siguiendo un argumento similar se puede demostrar que la clase de funciones definidas en $\mathcal{N} \rightarrow \{0, 1\}$ tampoco es numerable. Por tanto, existen funciones que no pueden ser computables por ningún algoritmo, lo cual en sí no es sorprendente; pero lo que sin duda debe sorprendernos es que muchas funciones y problemas tanto en computación como en matemática y otras disciplinas no son computables.

Universalidad y Auto referencia

Uno de los aspectos más sobresalientes en los modelos de computabilidad (Máquinas de Turing, λ -calculus, funciones μ -recursivas, etc.) es la idea de *programas*



Figura 9.36: Alonzo Church, Alan Turing y Kurt Gödel (junto con Albert Einstein).

como datos: cada uno de tales sistemas es suficientemente poderoso para codificar los programas como si fueran datos y manipularlos²⁷. Esta idea conduce a la noción de *simulación universal* en la que un programa (ó máquina) universal U se construye para recibir como entrada la codificación de un programa ó máquina M y una cadena x como argumento para M , y realizar una simulación (i.e., interpretación) de M ante la entrada x .

Una consecuencia de la universalidad es la noción de *auto referencia*, ésta capacidad condujo al descubrimiento de problemas no computables: por ejemplo, ante una máquina universal U , ¿que pasa si recibe su propio código como entrada? En general, ¿se puede saber si cualquier programa M al ser interpretado por U se detendrá *siempre* ante cualquier entrada x ? Tales son problemas muy importantes que a los constructores de compiladores les gustaría resolver en general, sin embargo se puede demostrar que esos problemas son insolubles.

Tal vez el ejemplo más sobresaliente del poder de la auto referencia es el Teorema de Incompletitud de Kurt Gödel. Al comienzo del siglo XX, Russell y Whitehead publicaron su obra *Principia Mathematica* cuyo propósito era reducir la matemática a la manipulación simbólica, de manera independiente de la semántica. Esto fué en parte para comprender y evitar las paradojas en teoría de conjuntos que fueron

descubiertas por Russell y otros. El movimiento se conoció como *programa formalista* y tuvo como figura importante al matemático David Hilbert. Atrajo a muchos seguidores y alimentó el desarrollo de la lógica matemática clásica tal como se conoce hoy en día. La creencia de los seguidores del programa formalista era que todas las verdades matemáticas podrían derivarse en un sistema formal fijo, a partir de un conjunto de axiomas y aplicando reglas de inferencia de manera completamente mecánica. El sistema deductivo formal más popular en esa época era la *aritmética de Peano (PA)*, considerado el adecuado para expresar y derivar todos los enunciados verdaderos en la teoría de los números naturales. El teorema de incompletitud²⁸ demostró que existen enunciados simples sobre la teoría de números que son verdaderos pero inde demostrables en PA. El resultado afecta no solamente a PA, sino a cualquier extensión *razonable* (i.e., suficientemente poderosa) de PA. Tal revelación fué impactante en el programa formalista.

Gödel demostró el teorema de incompletitud aplicando auto referencia. La observación básica es que el lenguaje de la teoría de números es suficientemente expresivo para describirse a sí mismo y a las pruebas en PA. Por ejemplo, se puede escribir un enunciado de la teoría de números que dice que otro enunciado de la teoría de números tiene una prueba en

²⁷En LISP, dado que todo es una *expresión -S* se puede interpretar fácilmente programas de LISP dentro de LISP mismo.

²⁸Recuérdese que un sistema deductivo formal es **correcto** si todo teorema -enunciado demostrable- es verdadero; y es **completo** si todo enunciado verdadero es demostrable.

PA, y se puede razonar sobre tales enunciados en PA mismo. Gödel construyó un enunciado verdadero que se puede parafrasear como “yo no soy demostrable”.

La noción de universalidad claramente condujo al desarrollo de computadoras como las conocemos actualmente: la versatilidad de las computadoras yace considerablemente en la noción de programa almacenado: software que puede leerse y ejecutarse por hardware. El germen de tal idea estuvo presente en el trabajo teórico de Alan Mathison Turing en 1936, por tanto se le considera con justa medida como el padre de la computación moderna²⁹.

9.2 El modelo básico de la Máquina de Turing

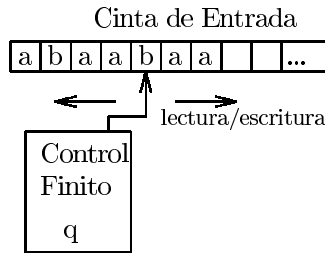


Figura 9.37: El modelo básico de la Máquina de Turing.

Como comentamos al inicio de éste capítulo, informalmente una Máquina de Turing es un AF extendido: su cinta de entrada es infinita por la derecha y su cabeza se puede mover en cualquier dirección y puede tanto leer como escribir símbolos. Inicialmente se recibe una cadena de entrada de longitud finita sobre el inicio de la cinta y el resto de las celdas contienen un espacio en blanco \square . La máquina inicia en el estado q_0 y su cabeza se encuentra situada sobre el primer símbolo de la cadena de entrada (leído de izquierda a derecha). En cada paso, lee un símbolo sobre la cinta bajo su cabeza y dependiendo del estado actual:

1. cambia su estado,
2. escribe un símbolo en la celda bajo la cabeza,
3. mueve su cabeza una celda a la izquierda ó derecha.

La acción que tome en cada situación está dada por su función de transición δ . La máquina acepta la cadena

de entrada si una vez examinada la cadena se detiene en algún estado de aceptación, de lo contrario, si se detiene en un estado que no es de aceptación significa que rechaza la cadena. Sin embargo, a diferencia de los AF, en algunas entradas la MT podría caer en un ciclo y continuar ejecutando transiciones sin detenerse, y por lo tanto sin aceptar ó rechazar la cadena.

Definición 9.2.1 (MT) Una Máquina de Turing determinística con una cinta se denota por la tupla

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, \square, F \rangle$$

donde

- Q es el conjunto finito de estados,
- Σ es el alfabeto de entrada,
- Γ es el alfabeto de la cinta ($\Sigma \subseteq \Gamma$),
- $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ es la función de transición,
- $q_0 \in Q$ es el estado inicial
- $F \subseteq Q$ es el conjunto de estados de aceptación (finales)
- $\square \in \Gamma$ es el símbolo de espacio en blanco ($\square \notin \Sigma$).

(Algunos autores también incluyen como componentes a \vdash un símbolo que indica el comienzo de la cinta, t un único estado final de aceptación y r un único estado de rechazo).

Intuitivamente, $\delta(p, a) = (q, b, d)$ significa que “cuando la MT está en el estado p y examinando el símbolo a , a continuación escribe b en esa celda, mueve la cabeza en la dirección d y entra al estado q ”. L y R indican izquierda y derecha respectivamente. Si la cabeza de la máquina está sobre el inicio de la cinta, un movimiento a la izquierda no tiene efecto. Resulta también conveniente indicar con S que la cabeza no se desplaza de la celda sobre la que se encuentra. A menudo se sigue la convención que si la máquina entra en un estado de aceptación ya no lo deja (y similarmente para el caso de entrar en un estado de rechazo). Como ocurre con los AF, una MT que su función de transición no está definida para alguna pareja $a \in \Sigma, q \in Q$ se detiene inmediatamente y rechaza la cadena.

Nos referiremos al conjunto de estados y a la función de transición colectivamente como *control finito*.

²⁹Eso sin mencionar que también sentó los retos e ideas fundamentales de lo que ahora se conoce como *Inteligencia Artificial*.

Definición 9.2.2 (ID) Se denota a una **descripción instantánea** ó **configuración** por $\alpha_1 q \alpha_2$, donde q denota el estado actual de M y la cadena $\alpha_1 \alpha_2 \in \Gamma^*$ denota el contenido de la cinta hasta el símbolo más a la derecha distinto de \square ó el símbolo situado a la izquierda de la cabeza lectora, el que esté más a la izquierda (nótese que nada impide que \square esté dentro de $\alpha_1 \alpha_2$). Se considera que la cabeza lectora está examinando el símbolo más a la izquierda de α_2 ; si ésta cadena es ϵ entonces la cabeza está sobre \square .

Una forma alternativa más explícita para describir una configuración es la siguiente: en cualquier instante, la cinta de la máquina M contiene una cadena semi infinita de la forma $x \square^\omega$, donde $x \in \Gamma^*$ de longitud finita y \square^ω representa a la cadena semi infinita $\square \square \square \dots$ (donde ω denota al ordinal infinito más pequeño). Una configuración por tanto es un elemento de $Q \times \{x \square^\omega \mid x \in \Gamma^*\} \times \mathcal{N}$: (p, z, n) indica que en el estado actual p la cinta contiene a z y la posición actual de la cabeza lectora es sobre la celda n .

La configuración inicial sobre la entrada $x \in \Sigma^*$ es

$$(q_0, x \square^\omega, 0)$$

Definición 9.2.3 (Movimiento \rightarrow) Se define la relación siguiente **configuración** de la misma forma como con los AP: para una cadena $z \in \Gamma^\omega$, sea z_n el n -ésimo símbolo de z (el símbolo más a la izquierda es z_0) y denotaremos por $s_b^n(z)$ la cadena que resulta de substituir b por z_n en la posición n de z , e.g.

$$s_b^4(\text{baacabca} \dots) = \text{baababca} \dots$$

La relación \rightarrow se define por

$$(p, z, n) \rightarrow \begin{cases} (q, s_b^n(z), n-1) & \text{si } \delta(p, z_n) = (q, b, L), \\ (q, s_b^n(z), n+1) & \text{si } \delta(p, z_n) = (q, b, R). \end{cases}$$

Intuitivamente, si la cinta contiene z y M está en el estado p examinando la n -ésima celda y δ indica escribir b , moverse a la izquierda y entrar al estado q , entonces después de aplicar ese paso la cinta contendrá $s_b^n(z)$ la cabeza se habrá movido a la celda $n-1$ y el nuevo estado será q . De manera correspondiente ocurre ante un movimiento por la derecha.

Indicamos por $\xrightarrow{*}$ a la cerradura reflexiva y transitiva de \rightarrow .

Definición 9.2.4 ($L(M)$) El lenguaje aceptado por M es el conjunto de las cadenas en Σ^* que provocan a M entrar a un estado de aceptación partiendo del estado inicial q_0 y una vez examinada toda la cadena:

$$L(M) = \{x \mid x \in \Sigma^*, (q_0, x \square^\omega, 0) \xrightarrow{*} (q_f, \gamma, n)\}$$

para alguna $\gamma \in \Gamma^\omega$, $n \in \mathcal{N}$ y $q_f \in F$.

Evidentemente la MT rechaza x si $(q_0, x \square^\omega, 0) \not\xrightarrow{*} (q, \gamma, n)$ para cualquier $q \notin F$.

Se dice que la máquina se **detiene** ante la entrada x si ya sea que la acepta ó la rechaza. Es posible que ni la acepte o rechace, lo cual significa que se continúa ejecutándose indefinidamente (se *cicla*) ante x .

Definición 9.2.5 Una Máquina de Turing M se denomina **total** si se detiene ante toda cadena de entrada, i.e., ya sea que la acepta ó la rechaza.

Un conjunto de cadenas es:

- **Recursivamente Enumerable** (r.e.) si es el $L(M)$ para alguna Máquina de Turing M ,
- *co-r.e* si su complemento es r.e.,
- **Recursivo** si es el $L(M)$ para alguna Máquina de Turing M total.

Dada una **propiedad** (o también, *predicado*) P para cadenas definidas sobre el alfabeto Σ (por ejemplo, P puede ser la propiedad de si la cadena tiene un número impar de cierto símbolo de Σ) decimos que P es:

Decidible si el conjunto de todas las cadenas que cumplen con P es un conjunto recursivo, i.e., existe una MT total M que acepta a las cadenas $x \in \Sigma^*$ que cumplen P y rechaza aquellas que no:

$$L(M) = \{x \in \Sigma^* \mid P(x)\} \text{ es recursivo}$$

Semidecidible (parcialmente decidible) si el conjunto de las cadenas que cumplen P es un conjunto recursivamente enumerable, lo cual significa que existe una MT M que ante x dada como entrada en su cinta la acepta si cumple con la propiedad P y la rechaza o continúa ejecutándose indefinidamente (se *cicla*) si x no cumple P :

$$L(M) = \{x \in \Sigma^* \mid P(x)\} \text{ es r.e.}$$

Indecidible si el conjunto de las cadenas que cumplen P no es r.e.: ninguna MT se detiene ante la entrada x lo cual significa que no se puede saber si x tiene o no la propiedad P .

Los adjetivos *recursivo* y *r.e.* se usan para referirse a conjuntos mientras que *decidible*, *indecidible* y *parcialmente decidible* se aplican a propiedades;

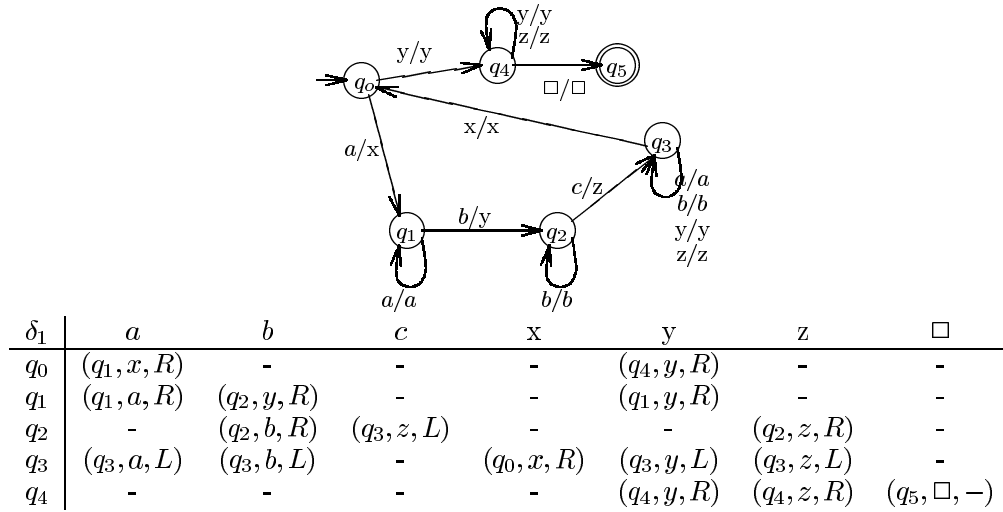


Figura 9.38: Diagrama y función de transición para la MT del ejemplo 9.2.1.

nótese que ambas nociones son equivalentes:

P es decidable $\iff \{x \mid P(x)\}$ es recursivo.
 A es recursivo $\iff "x \in A"$ es decidable,
 P es semidecible $\iff \{x \mid P(x)\}$ es r.e.
 A es r.e. $\iff "x \in A"$ es semidecible,
 P es indecible $\iff \{x \mid P(x)\}$ no es r.e.
 A no es r.e. $\iff "x \in A"$ es indecible.

A continuación se muestra que la cadena $aabbcc$ es aceptada:

$q_0aabbcc \rightarrow xq_1abbcc \rightarrow xaq_1bbcc$
 $\rightarrow xayq_2bcc \rightarrow xaybq_2cc \rightarrow xayq_3bcc$
 $\rightarrow xaq_3ybcc \rightarrow xq_3aybcc \rightarrow q_3xaybcc$
 $\rightarrow xq_0aybcc \rightarrow xxq_1ybcc \rightarrow xxyq_1bcc$
 $\rightarrow xxyq_2zcc \rightarrow xxyyzq_2c \rightarrow xxyyq_3zz$
 $\rightarrow xxyq_3yzz \rightarrow xxq_3yyzz \rightarrow xq_3xyyzz$
 $\rightarrow xxq_0yyzz \rightarrow xxyq_4yzz \rightarrow xxyyq_4zz$
 $\rightarrow xxyyzq_4z \rightarrow xxyyzzq_4 \rightarrow xxyyzzq_5$

Ejemplo 9.2.1 La Figura 9.38 muestra a la función de transiciones de la Máquina de Turing M_1 que acepta el lenguaje $\{a^n b^n c^n \mid n \geq 1\}$ (el cual no es libre de contexto), donde $M_1 = \langle Q_1, \Sigma_1, \Gamma_1, \delta_1, q_0, \square, \{q_5\} \rangle$, $Q_1 = \{q_0, q_1, q_2, q_3, q_4, q_5\}$, $\Sigma_1 = \{a, b, c\}$, $\Gamma_1 = \{a, b, c, x, y, z\}$.

Su funcionamiento es el siguiente: partiendo del estado inicial q_0 marca con una 'x' la primer a que encuentre (de izquierda a derecha) cambiando a q_1 ; a continuación, examina hacia la derecha hasta encontrar la primer b que marca con una 'y' (cambiando a q_2) y continúa por la derecha hasta encontrar la primer c que marca con una 'z' (cambiando a q_3); entonces desplaza la cabeza hacia la izquierda hasta encontrar la primer 'x', se mueve una celda a la derecha y cambia a q_0 dando comienzo de nuevo el ciclo. Cuando en q_0 se encuentra con una 'y' inmediatamente después de una 'x' entonces cambia a q_4 y se desplaza continuamente a la derecha, verificando que sólo haya marcas 'y' y 'z' en la cinta. Si es así, alcanza el final de la cadena (\square) y termina en el estado de aceptación q_5 .

Mientras que la cadena $aaabbccc$ es rechazada:

$q_0aaabbccc \rightarrow xq_1aabbccc \rightarrow xaq_1abbccc$
 $\rightarrow xaaq_1bbccc \rightarrow xaaq_2bbccc \rightarrow xaaq_2bccc$
 $\rightarrow xaaq_3bccc \rightarrow xaaq_3ybccc \rightarrow xaaq_3ybccc$
 $\rightarrow xq_3aaybccc \rightarrow q_3xaybccc \rightarrow q_3xaybccc$
 $\rightarrow xxq_1aybccc \rightarrow xxaq_1ybccc \rightarrow xxaq_1ybccc$
 $\rightarrow xxaayq_2zcc \rightarrow xxaayyzq_2c \rightarrow xxaayyq_3zcc$
 $\rightarrow xxaayq_3yzzc \rightarrow xxaq_3yyzzc \rightarrow xxaq_3yyzzc$
 $\rightarrow xq_3xayyzzc \rightarrow xxq_0ayyzzc \rightarrow xxq_0ayyzzc$
 $\rightarrow xxxq_1yzzc \rightarrow xxxyyq_1zzc$

Estudiaremos tres usos de las Máquinas de Turing:

1. Como aceptores de cadenas, cuyo uso ya hemos ilustrado.
2. Como evaluadoras de funciones.
3. Como enumeradoras de cadenas de lenguajes.

Discutiremos a continuación el uso de las Máquinas de Turing para evaluar funciones y en la sección 9.5 discutiremos su uso para enumerar (en listar recursivamente) los elementos de un conjunto r.e.

9.3 Funciones (y lenguajes) Computables

Como hemos mencionado, tal vez el aspecto más sobresaliente del modelo de las Máquinas de Turing es que describen formalmente la noción de algoritmo y por lo tanto de *función computable*. A su vez, el modelo de MT se extiende naturalmente a la *MT Universal*: máquinas programables cuyos programas pueden almacenarse, cargarse en memoria y ejecutarse.

Sin pérdida de generalidad, consideraremos funciones de aridad uno³⁰ definidas sobre los naturales $\mathcal{N} \rightarrow \mathcal{N}$. El primer paso consiste en elegir una representación general para codificar a los naturales sobre símbolos del alfabeto de la MT. En adelante, a menos que indiquemos lo contrario, siempre consideraremos el alfabeto binario $\Sigma = \{0, 1\}$. Cualquier natural i puede representarse sobre la cinta de la MT como la cadena 0^{i+1} (esto nos permite diferenciar a la representación del 0 y del 1). Para funciones de k argumentos: i_1, i_2, \dots, i_k se colocan en la cinta separados por un '1': $0^{i_1+1}10^{i_2+1} \dots 0^{i_k+1}$. Si $m \in \mathcal{N}$ denotaremos la *codificación* de m por $\lceil m \rceil \in \Sigma^*$.

Definimos a continuación las nociones de funciones (recursivas) parciales y totales:

Definición 9.3.1 (Función (Recursiva) Parcial)
La función parcial $f(x) : \mathcal{N} \rightarrow \mathcal{N}$ **computada** por la Máquina de Turing M se define como sigue: considérese una configuración inicial para M dada por $(q_0, \lceil x \rceil \square^\omega, 0)$, entonces

$$f(x) = \begin{cases} \#0(w) + 1 & \text{si } (q_0, \lceil x \rceil \square^\omega, 0) \xrightarrow{*} (q_f, w \square^\gamma, n), \\ \text{indefinido, de lo contrario.} \end{cases}$$

para algún $w, \gamma \in \Gamma^*$, $q_f \in F$, $n \in \mathcal{N}$. (Recuérdese que la notación $\#0(w)$ indica el número de 0's en la cadena w).

Definición 9.3.2 (Función (Recursiva) Total)
Una función parcial $f(x)$ se denomina **total** si está definida para todo argumento x , es decir, existe una MT total M que computa a $f(x)$. Alternativamente, se dice que f es una **función recursiva total**.

En consecuencia, una función parcial es *Turing-computable* si existe una MT que la computa. Denotaremos por \mathcal{TC} a la clase de todas las funciones Turing computables.

Evidentemente, el lector habrá notado la relación entre las funciones recursivas parciales y totales con respecto de los conjuntos r.e. y recursivos.

³⁰Existe un resultado, conocido como *Teorema s-m-n* que ruginosamente hablando establece una correspondencia entre funciones *computables* de cualquier aridad con funciones *computables* de aridad uno.

Ejemplo 9.3.1 A continuación se muestra una MT total que computa $x + y$.

$M = \langle \{q_0, q_1, q_2, q_3, q_4\}, \Sigma, \Gamma, \delta, q_0, \square, \emptyset \rangle$			
δ	0	1	\square
$\rightarrow q_0$	$(q_0, 0, R)$	$(q_1, 0, R)$	-
q_1	$(q_1, 0, R)$	-	(q_2, \square, L)
q_2	(q_3, \square, L)	-	-
q_3	(q_4, \square, S)	-	-

El funcionamiento de la MT es el siguiente: recorre la cadena hasta encontrar el primer '1' (separador entre x y y) y lo reemplaza por un '0', continúa hasta el final de la cadena y cambia el último 0 por \square , se detiene en el estado de aceptación q_f . La cadena resultante claramente tiene número de ceros correspondiente a $x + y + 1$.

Mostramos el cómputo para $2 + 1$:

$$\begin{aligned} q_0 000100 \square &\longrightarrow 0 q_0 00100 \square \longrightarrow 00 q_0 0100 \square \\ &\longrightarrow 000 q_0 100 \square \longrightarrow 0000 q_1 00 \square \longrightarrow 00000 q_1 0 \square \\ &\longrightarrow 000000 q_1 \square \longrightarrow 000000 q_2 0 \square \longrightarrow 000000 \square q_3 \square \end{aligned}$$

Caracterización Inductiva de \mathcal{TC}

Nos preguntamos a continuación si existe alguna relación entre el término *recursivo* con respecto de la naturaleza de las funciones Turing-computables. La respuesta es positiva y consiste en describir a la clase de las funciones Turing computables como un *conjunto inductivo* cerrado bajo operaciones que preservan computabilidad. Tales operaciones también son efectivamente computables³¹.

Lema 9.3.1 Las siguientes funciones básicas son computables:

1. la función **cero**: $0(x) = 0, x \in \mathcal{N}$,
2. la función **sucesor**: $x + 1, x \in \mathcal{N}$,
3. la función **proyección**: U_i^n dada por $U_i^n(x_1, x_2, \dots, x_n) = x_i$, para cada $n \geq 1$ y $1 \leq i \leq n$.

Demostración: constrúyase MT totales para computar cada una de esas funciones.

Un aspecto fundamental para las operaciones que se describen a continuación es la *normalización y composición* de Máquinas de Turing, es decir, el uso de *subrutinas*. Consideraremos por el momento que hemos establecido normas para invocar una subrutina dentro de la función δ de una MT; mostraremos un ejemplo a continuación:

³¹Las demostraciones se omiten, pero el lector interesado las puede encontrar en las notas del curso *Funciones Recursivas y Máquinas de Turing*

Ejemplo 9.3.2 La siguiente Máquina de Turing computa a la función recursiva total $x \times y$. Recibe como entrada $0^m 10^n$ y termina con $0^{m \times n}$ rodeada por \square .

Primero coloca un '1' al final de $0^m 10^n$ y entonces copia el bloque de n 0's al extremo de la derecha m veces, cada vez eliminando uno de los m 0's; el resultado parcial queda como $10^n 10^{m \times n}$. Para finalizar, el prefijo $10^n 1$ se elimina dejando el resultado $0^{m \times n}$. De vital importancia es la rutina *copiar* la cual toma configuraciones de la forma $0^m q_1 0^n 10^i$ y alcanza eventualmente $0^m q_5 0^n 10^{i+n}$. Mostramos la rutina *copiar* a continuación:

δ	0	1	X	\square
q_1	(q_2, X, R)	$(q_4, 1, L)$	-	-
q_2	$(q_2, 0, R)$	$(q_2, 1, R)$	-	$(q_3, 0, L)$
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_1, X, R)	-
q_4	-	$(q_5, 1, R)$	$(q_4, 0, L)$	-

Necesitamos convertir la configuración inicial $q_0 0^m 10^n$ (equivalentemente, $(q_0, 0^m 10^n \square^\omega, 0)$) a $\square 0^{m-1} 1 q_1 0^n 1$ (i.e., $(q_1, \square 0^{m-1} 1 q_1 0^n 1 \square^\omega, m+1)$), lo cual se logra incorporando los siguientes renglones a la tabla δ anterior:

q_0	-	-	-	(q_6, \square, R)
q_6	$(q_6, 0, R)$	$(q_1, 1, R)$	-	-

(Nótese la invocación a *copiar* desde $(q_6, 1)$).

La parte final que se encarga de convertir la configuración $\square^i 0^{m-i} 1 q_5 0^n 10^{n \times i}$ para que quede como $\square^{i+1} 0^{m-i-1} 1 q_1 0^n 10^{n \times i}$ en la que se vuelve a llamar a *copiar* y verificar si $i = m$, en cuyo caso se elimina el prefijo $10^n 1$ y se detiene el cómputo en el estado $q_{12} \in F$:

δ	0	1	X	\square
q_5	$(q_7, 0, L)$	-	-	-
q_7	-	$(q_8, 1, L)$	-	-
q_8	$(q_9, 0, L)$	-	-	(q_{10}, \square, R)
q_9	$(q_9, 0, L)$	-	-	(q_0, \square, R)
q_{10}	-	(q_{11}, \square, R)	-	-
q_{11}	(q_{11}, \square, R)	(q_{12}, \square, R)	-	-

Mostramos a continuación las siguientes operaciones efectivas que preservan computabilidad:

- **Substitución:** Supóngase que $f(y_1, \dots, y_k)$ y $g_1(\vec{x}), \dots, g_k(\vec{x})$ son funciones computables (donde $\vec{x} = (x_1, \dots, x_n)$). La función $h(\vec{x})$ definida por

$$h(\vec{x}) \simeq f(g_1(\vec{x}), \dots, g_k(\vec{x}))$$

es computable.

- **Recursión Primitiva:** Sea $\vec{x} = (x_1, \dots, x_n)$, y sean funciones computables $f(\vec{x})$ y $g(\vec{x}, y, z)$, entonces existe una única función $h(\vec{x}, y)$ que satisface las siguientes ecuaciones de recursión

$$\begin{aligned} h(\vec{x}, 0) &\stackrel{def}{=} f(\vec{x}), \\ h(\vec{x}, y+1) &\stackrel{def}{=} g(\vec{x}, y, h(\vec{x}, y)). \end{aligned}$$

Cuando $n = 0$, las ecuaciones anteriores toman la forma:

$$\begin{aligned} h(0) &\stackrel{def}{=} a, \\ h(y+1) &\stackrel{def}{=} g(y, h(y)). \end{aligned}$$

para $a \in \mathcal{N}$.

A la clase de funciones cerrada bajo las anteriores operaciones se les da el nombre de *funciones primitivas recursivas*.

El siguiente operador permite describir *problemas de búsqueda*:

- **Minimización acotada:** se define al operador de minimización μ : para cualquier función $f(\vec{x}, y)$

$$\mu y (f(\vec{x}, y) = 0) = \begin{cases} \text{el menor } y \text{ tal que} \\ (i) f(\vec{x}, z) \text{ está definida} \\ \text{para toda } z \leq y, \\ (ii) f(\vec{x}, z) = 0, \\ \text{si tal } y \text{ existe.} \\ \text{indefinida, si no existe tal } y. \end{cases}$$

$\mu y (\dots)x$ se lee 'el menor y tal que \dots ', el cual simplemente se le denomina operador μ .

Sea $f(\vec{x}, y)$ computable, entonces la función

$$g(\vec{x}) = \mu y (f(\vec{x}, y) = 0)$$

también es computable.

Ejemplo 9.3.3 Mostramos algunos ejemplos de funciones computables:

1. la suma $x + y$ es primitiva recursiva: por recursión utilizando a las funciones (cero) $0(x)$ y sucesor

$$\begin{aligned} x + 0 &= y \\ x + (y + 1) &= (x + y) + 1. \end{aligned}$$

2. $x \times y$ es primitiva recursiva: por recursión utilizando a las funciones (cero) $0(x)$ y (suma) $z + x$ en

$$\begin{aligned} x \times 0 &= 0 \\ x \times (y + 1) &= (x \times y) + x. \end{aligned}$$

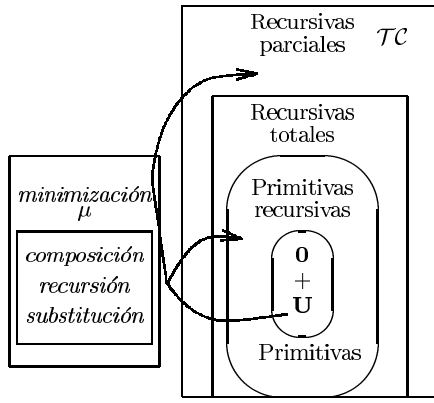


Figura 9.39: La clase de funciones computables \mathcal{TC} como un *conjunto inductivo*.

3. la función signo $sg(x) = \begin{cases} 0 & \text{si } x = 0, \\ 1 & \text{si } x \neq 0 \end{cases}$ es
primitiva recursiva: $sg(0) = 0$
 $sg(x+1) = 1$.

Aún cuando todas las funciones primitivas recursivas son totales, existen funciones totales que no son primitivas recursivas, el ejemplo clásico es la función de Ackermann:

$$\begin{aligned} A(0, y) &\stackrel{def}{=} y + 1, \\ A(x + 1, 0) &\stackrel{def}{=} A(x, 1), \\ A(x + 1, y + 1) &\stackrel{def}{=} A(x, A(x + 1, y)). \end{aligned}$$

Continuaremos analizando la noción de computabilidad en la sección 9.6.

9.4 Extensiones al modelo básico

Antes de describir la tercera utilidad de las Máquinas de Turing, conviene presentar primero algunas propiedades de los conjuntos recursivos y r.e. Primero, los conjuntos recursivos son cerrados bajo complemento: si A es recursivo, también lo es $\sim A$; para mostrarlo, existe una MT total M tal que $L(M) = A$ dado que A es recursivo. Simplemente se cambian los estados de aceptación y rechazo para obtener una MT M' total que acepta $\sim A$: $L(M') = \sim A$.

Si M no es total, la construcción anterior no da el complemento, dado que los estados de rechazo y aceptación no funcionan para MT no totales; M' aceptaría cadenas que M rechaza y rechazaría aquellas que M acepta, pero M' también se ciclaría cuando

M se cicla, por tanto ante esas cadenas no hay forma de saber si son o no aceptadas por M ni M' .

La clase de los conjuntos recursivos está contenida en la clase de los conjuntos r.e., y por tanto existen lenguajes que son r.e. pero no son recursivos. Si tanto A como $\sim A$ son r.e. entonces A es recursivo, para mostrarlo, sean M y M' Máquinas de Turing tales que $L(M) = A$ y $L(M') = \sim A$. Sea N una nueva MT tal que ante la entrada X ejecuta simultáneamente tanto a M como a M' en dos *pistas* diferentes de su cinta; i.e., el alfabeto de N contiene los símbolos

$\frac{a}{a}$	$\frac{\hat{a}}{c}$	$\frac{a}{\hat{c}}$	$\frac{\hat{a}}{\hat{c}}$
---------------	---------------------	---------------------	---------------------------

donde a es un símbolo de la cinta de M , c es un símbolo de la cinta de M' y la marca $\hat{}$ indica dónde se encuentra situada la cabeza para M y M' que se está simulando. N ejecuta alternadamente un paso de M y un paso de M' , los estados de M y M' se almacenan en el control finito de N . Si M acepta x entonces N acepta x , si M' acepta a x entonces N rechaza x inmediatamente. Exactamente uno de estos eventos puede ocurrir, por tanto N se detiene ante cualquier entrada x y $L(N) = A$.

Se pueden resumir las propiedades más importantes de los conjuntos r.e. y recursivos de la siguiente forma: Sean A y $\sim A$ dos lenguajes complementarios; entonces se cumple sólo una de las siguientes afirmaciones

1. ambos A y $\sim A$ son recursivos,
2. ninguno de A y $\sim A$ es r.e.,
3. uno entre A y $\sim A$ es r.e. pero no recursivo, el otro no es r.e.

A continuación presentaremos *sabores* distintos de las Máquinas de Turing, evidenciando con esto que el concepto de computabilidad es considerablemente robusto. Tales extensiones parecerían más poderosos ó expresivos que el modelo básico, sin embargo son computacionalmente equivalentes.

9.4.1 Multicintas

Ya hemos mostrado cómo simular con una sola cinta el disponer de varias pistas. Ahora mostraremos cómo simular MT con varias cintas utilizando una MT con una sola cinta³². Una MT M con tres cintas tiene tres cabezas independientes para cada cinta; la entrada se coloca inicialmente en la primer cinta, y las demás cintas están en blanco. En cada paso M lee los tres símbolos bajo sus cabezas, imprime un

³²Lo cual es gracias a que \mathcal{N}^n es numerable.

símbolo en cada cinta y mueve sus cabezas de manera independiente (i.e., en direcciones distintas inclusive), entrando en un nuevo estado. Su función de transición es de la forma

$$\delta : Q \times \Gamma^3 \longrightarrow Q \times \Gamma^3 \times \{L, R\}^3$$

(donde obviamente Γ es el alfabeto de M).

Para evitar ambigüedad, consideraremos que el símbolo \vdash indica el comienzo de la cinta.

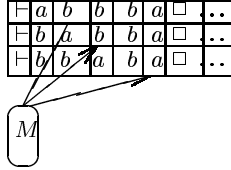


Figura 9.40: MT M multicintas.

Construiremos una MT N con una sola cinta equivalente a M . N tendrá un alfabeto de cinta extendido de tal suerte que permita considerar que su cinta está dividida en tres pistas, cada pista contendrá el contenido de una cinta de M . Se marcará exactamente un símbolo en cada pista para indicar que tal símbolo está siendo examinado por la cabeza correspondiente de M en ese instante. A continuación se muestra la configuración de N correspondiente a la configuración de M de la Figura 9.40.

\vdash	\vdash	a	\hat{b}	b	b	a	$\square \dots$
	\vdash	b	a	\hat{b}	b	a	$\square \dots$
	\vdash	b	b	a	b	\hat{a}	$\square \dots$

El alfabeto para N es $\Sigma \cup \{\vdash\} \cup (\Gamma \cup \Gamma')^3$ donde $\Gamma' \stackrel{\text{def}}{=} \{\hat{c} \mid c \in \Gamma\}$.

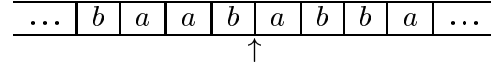
Ante la entrada $x = a_1 a_2 \dots a_n$, N comienza con su cinta así:

\vdash	\vdash	a_1	a_2	\dots	a_n	$\square \dots$
	\vdash	\square	\square	\dots	\square	$\square \dots$
	\vdash	\square	\square	\dots	\square	$\square \dots$

Cada paso de M será simulado por varios pasos de N : en un paso de M , N empieza en la izquierda de la cinta, examina las pistas hasta encontrar las tres marcas y los registra en su control finito, entonces determina la acción a realizar de acuerdo a la función δ de M , la cual tiene codificada en su propia función δ' . En base a esa información, escribe sobre los símbolos marcados y mueve la marca de acuerdo a la dirección correspondiente. Regresa al extremo izquierdo de la cinta y continúa simulando el siguiente paso de M .

9.4.2 Cintas infinitas en ambas direcciones

Tener cintas infinitas en ambas direcciones no otorga más poder: simplemente se elige algún sitio para “doblarla”



y se simula mediante una cinta infinita en una sola dirección con dos pistas:

\vdash	a	b	b	a	\dots
	b	a	a	b	\dots

Evidentemente, la pista superior se utiliza para simular a la MT original cuando su cabeza se encuentra sobre la parte derecha del punto elegido, mientras que la pista inferior corresponde a la parte izquierda.

9.4.3 No determinismo

Una Máquina de Turing no determinística (MTND) tiene una cinta semi-infinita por la derecha; para cada estado y símbolo examinado la máquina tiene un conjunto finito de posibles elecciones para la siguiente configuración, cada opción consiste en un nuevo estado, un símbolo a escribir en la cinta y una dirección de la cabeza:

$$\delta(q, a) = \{(p_1, a_1, D_1), (p_2, a_2, D_2), \dots, (p_k, a_k, D_k)\}$$

La MTND acepta su entrada si existe alguna secuencia de elecciones en las transiciones que conduzcan a un estado de aceptación (partiendo obviamente de la configuración inicial).

Al igual que con los AF, el no determinismo no agrega más poder expresivo, aún si combinamos el no determinismo con las versiones de multicintas o cintas infinitas en ambas direcciones. A continuación mostraremos cómo simular una MTND mediante una MT determinística y se deja como ejercicio al lector probar que el no determinismo agregado a alguna otra extensión de las MT no ofrece más poder expresivo.

Dado que para cualquier estado y símbolo de una MTND M existe un número finito de opciones, sea r el mayor k (número de opciones) para todo $a \in \Sigma, q \in Q$. Ello indica que cualquier secuencia finita de elecciones se puede representar por una secuencia de dígitos desde 1 hasta r (obviamente no todas son utilizadas, ya que pueden existir situaciones en las que se tenga menos de r opciones).

La MT determinística M' que simulará a M tendrá tres cintas: la primera contendrá la entrada, la segunda servirá para *generar sistemáticamente* secuencias de dígitos desde 1 hasta r , en orden de menor

a mayor. Para cada secuencia, M' copiará la entrada en la tercera cinta y sobre ella simulará a M utilizando la secuencia actual en la segunda cinta para dictar la elección correspondiente. Si M entra en un estado de aceptación M' acepta a su vez la entrada; si existe alguna secuencia de opciones que conduzcan a una configuración de aceptación M' generará eventualmente tal secuencia sobre la segunda cinta. De lo contrario M' no aceptará (pues M no aceptaría la entrada si no hay alguna secuencia de opciones exitosa).

Nótese la relación entre ésta estrategia y el recorrido de un árbol r -ario en forma primero en amplitud: si M necesita n pasos para aceptar/rechazar una cadena, M' necesitará del orden de n^r pasos (exponencial!) en el peor de los casos.

9.5 Máquinas de Turing como enumeradores

Ahora discutiremos el tercer uso de la Máquina de Turing: su capacidad para enlistar los elementos de un conjunto. De hecho, las Máquinas de Turing fueron presentadas originalmente por Alan Turing en forma de máquinas enumeradoras para enumerar las expansiones decimales de números reales computables y valores de funciones en los reales. Recordemos que un conjunto r.e. (recursivamente enumerable) es aquél que es aceptado por alguna MT. El término *recursivamente enumerable* viene de la idea que los elementos de un conjunto r.e. pueden enumerarse uno a la vez de manera mecánica.

Definición 9.5.1 (Máquina Enumeradora) Una máquina enumeradora M tiene un control finito y dos cintas, una de lectura/escritura para trabajo (sobre el alfabeto Γ) y una de sólo escritura para resultados (sobre el alfabeto Σ). La cabeza de la cinta de trabajo se puede mover en cualquier dirección, mientras que la cabeza de la cinta de resultados se mueve únicamente de izquierda a derecha una celda a la vez escribiendo alguna palabra tomada de Σ^* separadas por '#'. No tiene estados de aceptación o rechazo, no recibe entrada. La máquina comienza en su estado inicial con ambas cintas en blanco, cambia de estados de acuerdo a su función de transición, en algún momento puede entrar en un estado de enumeración y se dice que la cadena que se encuentre escrita sobre la cinta de resultados ha sido enumerada. La máquina se ejecuta por siempre. El conjunto $G(E)$ denota a todas las cadenas en Σ^* que son enumeradas ó generadas por la máquina enumeradora E . Si la máquina nunca entra en su estado de enumeración, $L(E) = \emptyset$, o puede enumerar

inifitamente muchas cadenas, incluso se permite que una misma cadena sea enumerada más de una vez.

Las máquinas enumeradoras y las Máquinas de Turing son equivalentes en poder expresivo: si L es $G(M)$ para alguna máquina enumeradora M entonces L es r.e. (i.e., existe una MT M' tal que $L = L(M')$) y viceversa. En particular, los conjuntos recursivos tienen la propiedad que sus palabras pueden generarse en orden de tamaño creciente.

Teorema 9.5.1 ($G(E) = L(M)$) La clase de los conjuntos enumerados por las máquinas enumeradoras coincide con la clase de los lenguajes r.e.: $L = G(E)$ para alguna máquina enumeradora E si y sólo si $L(M)$ para alguna MT M .

Demostración: primero hay que demostrar que dada una máquina enumeradora E se puede construir una MT M tal que $G(E) = L(M)$: sea M con tres pistas, ante la entrada x M la copia a alguna de sus pistas y entonces simula a E utilizando las otras dos pistas. Para cada cadena enumerada por E , M compara esa cadena con x y la acepta si coinciden. Por tanto, $x \in L(M)$ si y sólo si $x \in G(E)$.

En el otro sentido, dada una MT M podemos construir una máquina enumeradora E tal que $G(E) = L(M)$: queremos que E simule a M en toda cadena posible de Σ^* y enumere solamente a aquellas que son aceptadas por M .

Un enfoque erróneo es que E escriba las cadenas de Σ^* una a una en la pista inferior de su cinta de trabajo siguiendo *algún orden*. Entonces E simula a M ante cada x generada; si M acepta x , E copia a x a su cinta de salida, y continúa con la siguiente cadena. El problema es que M podría no detenerse ante alguna x y como E no tiene forma de saberlo anticipadamente no podría revisarse la siguiente cadena en la lista. Una solución pausable es realizar la simulación en *tiempo compartido*, lo cual significa que E realiza varias simulaciones a la vez, dividiendo el espacio de trabajo en segmentos para cada simulación y ejecutando algunos pasos en cada segmento (como si se tratase de un sistema multiprocesos): de esa forma E puede simular a M sobre toda entrada aún si alguna de las simulaciones no se detiene nunca.

El ordenamiento en la presentación de las cadenas se fija en una forma bastante conocida por nosotros:

Definición 9.5.2 (Orden canónico) Consiste en presentar a los miembros de Σ^* por tamaño, con palabras de igual tamaño en "orden numérico". Dado $\Sigma = \{a_0, a_1, a_2, \dots, a_{k-1}\}$ establecemos que a_i es el "dígito" i en base k ; las palabras de longitud n son los números de 0 a $k^n - 1$ en base k .

Ejemplo 9.5.1 Si $\Sigma = \{0, 1\}$, el orden canónico es $\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots$

Éste ordenamiento puede extenderse naturalmente a ordenar parejas (i, j) (tal como se utilizó en la enumeración diagonal de los elementos de $\mathcal{N} \times \mathcal{N}$). Una máquina enumeradora puede generar (i, j) tal que estén ordenadas por $i + j$, y entre parejas de igual suma, en orden creciente sobre i , i.e., se genera la secuencia

$$(0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0), \dots$$

Así, cada (i, j) ocupa el lugar $i + \frac{(i+j-1)(i+j-2)}{2}$ en la secuencia.

Ejercicios 9.5.1 1. Construya MTs que reconozcan los siguientes lenguajes:

- (a) $\{ww^R \mid w \in (0+1)^*\}$
- (b) El conjunto de las cadenas con igual número de 0's y 1's.

2. Construya MTs que computen las siguientes funciones:

$$(a) f(x) = x^2, h(x) = x!$$

3. Construya una MT que genere el orden canónico sobre las parejas (i, j) . Demuestre que los conjuntos recursivos son aquellos cuyas palabras se generan en orden canónico.

4. Construya una MT que enumere el lenguaje $\{0^n 1^n \mid n \geq 1\}$.

5. Demuestre que si L es aceptado por una MTND con k cintas, entonces L es aceptado por una MT determinística con una cinta semi infinita por la derecha.

9.6 La Máquina de Turing Universal

Ahora podremos apreciar el enorme poder de las Máquinas de Turing: existen MTs que pueden *simular* cualquier otra MT cuya descripción haya sido *codificada* apropiadamente como cadena de entrada. Para ello, primero es necesario establecer una convención de codificación para las Máquinas de Turing, sobre el alfabeto³³ $\Sigma = \{0, 1\}$. Esta codificación es tan sencilla que permitirá que toda la información de una MT M pueda determinarse fácilmente. Nos podremos plantear la pregunta: “¿La MT M acepta la cadena w ?”.

³³No hay ninguna pérdida de generalidad: puede demostrarse que cualquier otro alfabeto puede a su vez codificarse en $\{0, 1\}$

Codificación

Dada una MT $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \square, \{q_f\} \rangle$, la codificación de M es una cadena de $\{0, 1\}^*$ formada por dos partes:

1. El encabezado consiste de la información de M y tiene la forma

$$0^n 10^m 10^k 10^s 10^t 10^r 10^u 10^v$$

lo cual significa que la máquina tiene n estados ($|Q| = n$) representados por los números 0 a $n - 1$; tiene m símbolos en su alfabeto de entrada representados por los números 0 a $m - 1$, de los cuales los primeros k son símbolos de entrada (i.e., pertenecen a Σ); los estados de inicio, aceptación y rechazo son respectivamente s, t y r ; mientras que los símbolos para marcar el fin de cinta (opcional \vdash) y espacio en blanco (\square) son u y v .

2. El resto de la cadena constituye la descripción del *programa* de la máquina, dada en terminos desde luego de su función de transición. Dada la definición genérica de un cambio de configuración

$$\delta(p, a) = (q, b, D)$$

se codifica por la cadena

$$0^p 10^a 10^q 10^b 10^m$$

donde $m = 1$ si $D = L$ y $m = 2$ si $D = R$.

Denotaremos la codificación de $\delta(p, a)$ por $[\delta_{p,a}]$ y la codificación de M por $[M]$.

Sólo resta indicar el inicio y fin de la codificación de la MT:

$$111 \underbrace{0^n 1 \dots 0^v}_{[q_0]} 11 \underbrace{[\delta_{1,1}]}_{[s]} 11 \underbrace{[\delta_{1,2}]}_{[t]} 11 \dots 111$$

Nótese que en el cuerpo del programa, cada codificación de δ se separa por dos 1's.

Una vez que disponemos de una codificación adecuada, podemos construir una *Máquina de Turing Universal* U tal que

$$L(U) \stackrel{\text{def}}{=} \{[M] \# [x] \mid x \in L(M)\}$$

Los alfabetos de U son $\Sigma = \{0, 1\}$ y $\Gamma = \Sigma \cup \{\#, \square\}$ ($\#$ funciona como delimitador).

U primero verifica que la cadena de entrada, es decir, las codificaciones de M y x sean válidas, de no serlo inmediatamente rechaza la expresión. Si las codificaciones de M y x son correctas, U simula paso a paso a M ante x , lo cual esencialmente procede de la siguiente forma: la cinta de U se divide en tres pistas,

$[M]$ se coloca en la primer pista, mientras que $[x]$ en la segunda pista; la cual, se utilizará también para almacenar el contenido de la cinta de M . La pista final se utilizará para recordar el estado y la posición actual de la cabeza de M . Entonces, U simula a M paso a paso de acuerdo a la descripción δ de M sobre x : si M se detiene aceptando o rechazando a x U hace lo mismo.

En general, cada paso de M necesita varios pasos de U .

Ejemplo 9.6.1 *Considérese a la siguiente MT $M = \langle \{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, \square\}, \delta, q_0, \square, \{q_1\} \rangle$ cuya función de transición δ es*

δ	0	1	\square
q_0	-	$(q_2, 0, R)$	-
q_2	$(q_0, 1, R)$	$(q_1, 0, R)$	$(q_2, 1, L)$

Entonces, la expresión $[M]\#1011$ es:

11100010001001010010001110100100010100
11000101010010011010010010100
110001000100010010111#1011

(Las cajitas son sólo para legibilidad).

Diagonalización

Cabe preguntarnos inmediatamente: ¿es $L(U)$ recursivo? Sabemos que $L(U)$ es r.e., pero ¿cómo podemos asegurar que no sea recursivo? Primero demostraremos que un lenguaje, denominado *el lenguaje digonal* no es r.e. y mostraremos instancias que nos permitirán responder a la anterior y a otras muchas preguntas cercanamente relacionadas.

Recuérdese que en el capítulo de Preliminares demostramos aplicando el argumento de la Diagonalización de Cantor que no es posible establecer una correspondencia entre los \mathcal{N} y $2^{\mathcal{N}}$ (pág. 2). Esencialmente, supusimos (para obtener una contradicción) que existía una función sobre $f : \mathcal{N} \rightarrow 2^{\mathcal{N}}$ y construimos el conjunto $B = \{x \in \mathcal{N} \mid x \notin f(x)\}$, la cual es la manera formal de *completar la diagonal*³⁴. Dado que $B \subseteq \mathcal{N}$, debe existir $y \in \mathcal{N}$ tal que $f(y) = B$; al preguntarnos si $y \in f(y)$ nos conduce a la contradicción:

$$\begin{aligned} y \in f(y) &\text{ si y sólo si } y \in B && \text{ dado que } B = f(y) \\ y \in f(y) &\text{ si y sólo si } y \notin B && \text{ por definición de } B. \end{aligned}$$

Lo cual indica que no puede existir tal f .

³⁴Dado que consideramos que esa f nos permitía colocar en una tabla todos los valores de \mathcal{N} en la columna de la extrema izquierda y todos los correspondientes subconjuntos. Completar la diagonal significa construir B tal que difiere en cada elemento de la diagonal de la tabla.

Similarmente, podemos utilizar el mismo argumento para demostrar que el lenguaje *diagonal*

$$L_d = \{w_i \mid w_i \notin L(M_i)\}$$

(que podemos describir informalmente como: “el conjunto de todas las cadenas del alfabeto Σ bajo el orden canónico en posición i tales que no están en el lenguaje de la i -ésima Máquina de Turing”) no puede ser aceptado por ninguna Máquina de Turing.

Para ello, primero vamos a mostrar que la clase de todas las Máquinas de Turing es enumerable (lo cual es gracias a la codificación que hemos presentado) y entonces mostraremos que L_d no es r.e.

Teorema 9.6.1 (\mathcal{TC} es enumerable)

Demostración: genérese en orden canónico los códigos j para cada MT M_j (en otras palabras, la MT M_j está codificada por el valor natural j escrito en la forma binaria de codificación ya presentada).

Teorema 9.6.2 (L_d no es r.e.)

Demostración: se debe garantizar que ninguna MT acepta a L_d : aplíquese la técnica de diagonalización de Cantor sobre la tabla en la cual se tienen como columnas a las MT M_j y como renglones a las palabras $w_i \in \Sigma^*$ dispuestas en orden canónico.

Ahora podemos responder la pregunta si $L(U)$ es recursivo.

Teorema 9.6.3 ($L(U)$ no es recursivo)

Demostración: hemos demostrado que L_d no es r.e. y por tanto tampoco es recursivo, se deduce entonces que $\sim L_d$ no es recursivo, donde $\sim L_d = \{w_i \mid M_i \text{ acepta } w_i\}$. Demostraremos que $L(U)$ no es recursivo *reduciendo* $\sim L_d$ a $L(U)$: supongamos que existe un algoritmo (i.e., una MT total) A que reconoce $L(U)$, entonces podemos reconocer a $\sim L_d$ como sigue: dada una cadena $w \in \Sigma^*$, determínese el valor i del orden canónico correspondiente tal que $w_i = w$. Tal valor i a su vez indica el valor en binario para la i -ésima MT M_i . Al alimentar al algoritmo A con la tupla $[M_i]\#w_i$ el cual aceptará a w si y sólo si M_i acepta a w_i . Ello indica que se dispone a su vez de un algoritmo (i.e., una MT total) para aceptar $\sim L_d$, lo cual es una contradicción. Dado que no puede existir tal algoritmo, la suposición de la existencia de A también es falsa.

El lenguaje universal $L(U)$ se encuentra estrechamente relacionado con los siguientes problemas:

1. El problema de *Detención (Halt Problem)*:

$$HP \stackrel{def}{=} \{[M]\#[x] \mid M \text{ se detiene en } x\}$$

lo cual significa preguntarse si cualquier MT arbitraria M se detiene ante una entrada x arbitraria.

2. El problema de *membresía*:

$$MP \stackrel{def}{=} \{[M]\#[x] \mid x \in L(M)\}$$

a su vez significa si dada una MT arbitraria M y una cadena arbitraria x , x pertenece al lenguaje de M .

Demostraremos que ambos conjuntos no son recursivos.

Recuérdese que la MT Universal U permite *interpretar* el comportamiento de una MT M que recibe codificada como $[M]$, junto con una cadena de entrada codificada $[x]$. En principio, U no realiza ningún análisis a priori de M para saber si se detiene o no ante x (i.e., U realiza una *simulación a ciegas* de M). Obviamente, se puede preguntar si existe alguna manera más eficiente para que U determine, antes de realizar la simulación de M , si es que M se detenga o no eventualmente ante x ; si se puede responder a tal pregunta U evitará realizar trabajo en vano. Construiremos una MT U' que recibe $[M]\#[x]$ y

- Se detiene y acepta si M se detiene y acepta x ,
- Se detiene y rechaza si M se detiene y rechaza x ,
- Se detiene y rechaza si M se cicla (no se detiene) ante x .

Lo anterior significa que $L(U') = L(U) = MP$ es un conjunto recursivo.

Cabe preguntarse si U' puede construirse en general (i.e., para toda MT M y toda x). Lamentablemente, la respuesta es negativa. No obstante existen MTs para las cuales es posible determinar por anticipado si se detienen o no ante una entrada x mediante alguna *heurística*, no hay un método general que funcione para toda MT M . Demostraremos ésta afirmación aplicando el método de la Diagonalización de Cantor.

Teorema 9.6.4 (HP es indecidible)

Demostración: Como en el caso de L_d , dado el alfabeto $\Sigma = \{0, 1\}$ disponemos a la clase \mathcal{TC} enumerada por $x \in \Sigma^*$, i.e. M_x indica a la MT con alfabeto de entrada Σ cuya codificación es x (si x no representara

una descripción legal de alguna MT, tomaremos que M_x es alguna MT arbitraria fija, i.e., una MT trivial con un único estado inmediatamente se detiene). Efectivamente, la serie M_x presenta la enumeración bajo el orden canónico asociado a Σ^* de \mathcal{TC} :

\mathcal{N}	0	1	2	..	n	...
x	ϵ	0 1	00 01	..	$(0+1)^n$...
M_x	M_ϵ	M_0 M_1	M_{00} M_{01}	..	$M_{(0+1)^n}$...

Ésta enumeración la colocamos en el renglón superior de la tabla infinita, y en la columna del extremo izquierdo colocamos a las cadenas $x \in \Sigma^*$ bajo el orden canónico:

	M_ϵ	M_0	M_1	M_{00}	M_{01}	...
ϵ	L	H	H	L	H	...
0	H	H	H	H	L	...
1	L	L	H	H	L	...
01	L	H	H	L	L	...
10	L	L	L	H	L	...
:	:	:	:	:	:	:
$(0+1)^n$	L	H	L	H	H	...
:	:	:	:	:	:	..

Cada columna y indica si M_y se detiene (H) o no (L) ante cada $x \in \Sigma^*$.

Para propósitos de contradicción, supóngase que existe una MT total K que se encarga de aceptar el lenguaje HP , lo cual significa que puede determinar *en tiempo finito* el contenido de la entrada (x, y) de la tabla anterior para cualquier x, y . Esto es, ante $[M]\#[x]$,

- K se detiene y acepta si M se detiene al en x ,
- K se detiene y rechaza si M se cicla (no se detiene) ante x .

Considérese entonces la MT N que ante cualquier $x \in \Sigma^*$:

1. construye a la MT M_x a partir de x y escribe $M_x\#x$ en su cinta,
2. ejecuta K ante $M_x\#x$, aceptando x si K rechaza $M_x\#x$ y entrando a un ciclo trivial si K acepta $M_x\#x$.

(Evidentemente, N complementa la diagonal de la tabla anterior).

Entonces, para todo $x \in \Sigma^*$:

$$\begin{array}{ll} N \text{ se detiene ante } x & \\ \Updownarrow & \\ K \text{ rechaza } M_x\#x & \text{por definición de } N \\ \Updownarrow & \\ M_x \text{ se cicla ante } x & \text{por suposición de } K \end{array}$$

Por lo que N exhibe un comportamiento distinto de cada M_x en al menos la cadena x ; por lo que N no está en la enumeración anterior de \mathcal{TC} , lo cual es una contradicción.

La suposición que condujo a la contradicción es que fuese posible determinar las entradas de la tabla de manera efectiva, i.e., la existencia de K : en general, no hay forma de detener la simulación de una MT arbitraria después de un tiempo finito y asegurar con ello que dicha MT nunca se va a detener.

Es un tanto evidente que MP no es recursivo ya que mostramos que $L(U)$ no es recursivo, además podemos mostrar que si existe una forma para decidir membresía en general entonces se puede utilizar tal resultado como subrutina para decidir detención en general. A ésta técnica se le denomina *reducción* y la discutiremos a continuación.

Ejercicios 9.6.1 Dada una MT arbitraria M , ¿es decidable si

1. M tiene al menos 342 estados?
2. toma más de 243 pasos ante la entrada ϵ ?
3. toma más de 423 pasos ante alguna entrada?
4. toma más de 432 pasos ante toda entrada?
5. mueve su cabeza más de 148 celdas mas allá del inicio de la cinta ante la entrada ϵ ?
6. acepta ϵ ?
7. acepta alguna cadena?
8. acepta toda cadena?
9. acepta un conjunto finito?
10. acepta un conjunto regular?
11. acepta un lenguaje libre de contexto?
12. acepta un conjunto recursivo?

Justifica tu respuesta en cada caso.

9.7 Reducción

La última técnica que discutiremos nos permite mostrar si un nuevo problema B es tan difícil ó pertenece a la misma clase de un problema conocido A a partir del hecho que existe una correspondencia computable que permite plantear una solución de B como una instancia de solución de A . Si tal correspondencia existe, entonces B es tan difícil (o pertenece a la misma clase de problemas) que A . Ilustraremos la idea demostrando el siguiente teorema:

Teorema 9.7.1 (MP es indecidible)

Demostración: el propósito es describir cómo utilizar una MT total que decida membresía como una subrutina para decidir detención: dada una MT M ante una entrada x , supóngase que se quiere saber si M se detiene ante x . Constrúyase una nueva MT N que es exactamente como M pero N acepta su entrada x siempre que M se detenga (acepte o rechace) ante x . Por tanto, se ha reducido el problema de decidir si $x \in L(N)$ (membresía) al problema de decidir si M se detiene ante x (detención). Si el problema de membresía fuese decidable entonces también lo sería el problema de detención; como ya demostramos que detención HP es indecidible entonces MP también es indecidible.

Definición 9.7.1 (Reducción muchos-a-uno)

Dados conjuntos $A \subseteq \Sigma^*$ y $B \subseteq \Delta^*$, una reducción muchos a uno de A hacia B es una función computable

$$\sigma : \Sigma^* \longrightarrow \Delta^*$$

tal que para toda $x \in \Sigma^*$,

$$x \in A \iff \sigma(x) \in B$$

en otras palabras, las cadenas en A van a las cadenas de B bajo σ y las cadenas que no están en A deben ir a cadenas que no están en B . σ no necesita ser sobre o inyectiva, pero si debe ser total y computable.

Para denotar que A es reducible a B via σ se escribe $A \leq_m B$, el subíndice m significa “muchos-a-uno” y se utiliza para distinguirla de otros tipos de relaciones de reducibilidad.

La relación \leq_m entre lenguajes es transitiva: si $A \leq_m B$ y $B \leq_m C$ entonces $A \leq_m C$: lo cual es debido que si $A \xrightarrow{\sigma} B$ y $B \xrightarrow{\tau} C$, entonces $\tau \circ \sigma$ es computable y reduce A a C .

Ejemplo 9.7.1 Demostraremos que es indecidible el problema si una MT dada acepta ϵ : construimos una MT M' a partir de M y una cadena x que acepta ϵ si y sólo si M se detiene ante x . En éste ejemplo

$$\begin{aligned} A &= \{M\#x \mid M \text{ se detiene ante } x\} = HP, \\ B &= \{M \mid \epsilon \in L(M)\} \end{aligned}$$

y σ va de $M\#x$ a M' .

Teorema 9.7.2 (\leq_m) 1. Si $A \leq_m B$ y B es r.e., entonces también lo es A . Equivalentemente, si $A \leq_m B$ y A no es r.e., tampoco lo es B .

2. Si $A \leq_m B$ y B es recursivo, también lo es A . Equivalentemente, si $A \leq_m B$ y A no es r.e. tampoco lo es B .

Demostración:

1. Supóngase que $A \leq_m B$ vía σ y B es r.e. Sea M una MT tal que $L(M) = B$. Constrúyase una MT N para A de la siguiente manera: ante la entrada x , primero se computa $\sigma(x)$ y entonces se ejecuta M ante $\sigma(x)$, aceptando si M acepta. Así:

$$\begin{array}{ccc}
 N \text{ acepta } x & & \\
 \Downarrow & & \\
 M \text{ acepta } x & \text{def. de } N & \\
 \Downarrow & & \\
 \sigma(x) \in B & \text{def. de } M & \\
 \Downarrow & & \\
 x \in A & \text{def. reducción} &
 \end{array}$$

2. Recuérdesse que un conjunto es recursivo si y sólo si tanto él como su complemento son r.e. Supóngase que $A \leq_m B$ vía σ y B es recursivo. Nótese que $\sim A \leq_m \sim B$ es posible por la misma σ . Como se supone que B es recursivo entonces tanto B como $\sim B$ son r.e. Por el inciso anterior, tanto A como $\sim A$ son r.e., lo cual conduce que A es recursivo.

Ejemplo 9.7.2 *Mostraremos que el conjunto FIN y su complemento no son r.e., donde*

$$\begin{aligned}
 \text{FIN} &= \{M \mid L(M) \text{ es finito} \} \\
 \sim \text{FIN} &= \{M \# x \mid M \text{ no se detiene ante } x\}
 \end{aligned}$$

Reduciremos $\sim \text{HP}$ a FIN y su complemento, i.e.

1. $\sim \text{HP} \leq_m \text{FIN}$

Debemos construir una función computable total σ tal que

$$M \# x \in \sim \text{HP} \iff \sigma(M \# x) \in \text{FIN}$$

lo cual significa construir una MT $M' = \sigma(M \# x)$ la cual

$$M \text{ no se detiene ante } x \iff L(M') \text{ es finito}$$

Dado $M \# x$, M' hace lo siguiente ante cualquier entrada y :

- (a) borra su entrada y ,
- (b) escribe a x (la cual puede tener almacenada en su control finito) sobre su cinta,
- (c) simula a M ante x ,
- (d) acepta a y si M se detiene ante x .

Nótese que si M se detiene ante x , M' acepta cualquier $y \in \Sigma^*$, pero si M no se detiene ante x entonces M' no acepta ninguna y . A partir de tal descripción, se sigue inmediatamente que

$$\begin{aligned}
 M \text{ se detiene ante } x &\Rightarrow L(M') = \Sigma^* \\
 &\Rightarrow L(M') \text{ es infinito,} \\
 M \text{ no se detiene ante } x &\Rightarrow L(M') = \emptyset \\
 &\Rightarrow L(M') \text{ es finito.}
 \end{aligned}$$

2. $\sim \text{HP} \leq_m \text{FIN}$

Por la definición de reducción, una función total y computable τ que hace posible $\sim \text{HP} \leq_m \sim \text{FIN}$, también permite que $\text{HP} \leq_m \text{FIN}$, por lo que se mostrará cómo construir τ tal que:

$$M \# x \in \text{HP} \iff \tau(M \# x) \in \text{FIN}$$

Ello significa construir una MT $M'' = \tau(M \# x)$ tal que

$$M'' \text{ se detiene ante } x \iff L(M'') \text{ es finito}$$

Dado $M \# x$, M'' ante la entrada y :

- (a) salva a y en alguna pista,
- (b) escribe a x en otra pista de la cinta,
- (c) simula a M en x para $|y|$ pasos (borra un símbolo de y para cada paso de M en x que simule),
- (d) acepta a y si M no se detuvo dentro del tiempo indicado, en otro caso rechaza y .

Si M no se detiene ante x entonces M'' se detiene y acepta y tras $|y|$ pasos de la simulación, para todo $y \in \Sigma^*$. Por otra parte, si M se detiene ante x lo hace tras completar n pasos, entonces M'' acepta y si $|y| < n$ y rechaza a y si $|y| \geq n$. En éste caso M'' acepta toda cadena de longitud menor a n y rechaza toda cadena de longitud mayor ó igual a n , i.e. $L(M'')$ es finito y

$$\begin{aligned}
 M \text{ se detiene ante } x &\Rightarrow L(M'') = \{y \mid |y| < \text{el tiempo de ejecución de } M \text{ ante } x\} \\
 &\Rightarrow L(M'') \text{ es finito,} \\
 M \text{ no se detiene ante } x &\Rightarrow L(M'') = \Sigma^* \\
 &\Rightarrow L(M'') \text{ es infinito.}
 \end{aligned}$$

9.8 El Teorema de Rice

Esencialmente, el teorema de Rice dice que la indecidibilidad es la regla, no la excepción.

Teorema 9.8.1 (Teorema de Rice) *Toda propiedad no trivial de los conjuntos r.e. es indecidible.*

Antes de la demostración, conviene establecer lo siguiente: dado un alfabeto fijo Σ , indiquemos por \mathcal{RE} a la clase de los conjuntos r.e. de Σ^* , i.e., $\mathcal{RE} \subseteq 2^{\Sigma^*}$. Una *propiedad*³⁵ de los conjuntos r.e. es una función

$$P : \mathcal{RE} \longrightarrow \{\perp, \top\}$$

donde \top y \perp representan verdadero y falso, respectivamente. Por ejemplo, la propiedad de vacuidad se representa por la función

$$P(A) = \begin{cases} \top & \text{si } A = \emptyset, \\ \perp & \text{si } A \neq \emptyset. \end{cases}$$

El preguntarnos si una propiedad P es *decidible* significa representar el conjunto correspondiente de manera finita, adecuada para ser la entrada de una MT. Podemos asumir que los conjuntos r.e. se representan por MTs que los aceptan; pero hay que tener presente que la propiedad de un conjunto no es la propiedad de una MT, las propiedades de los conjuntos son verdaderas o falsas *independientemente* de la MT en particular que se haya elegido para representar a los conjuntos.

Ejemplo 9.8.1 La siguiente tabla muestra algunas propiedades de los conjuntos r.e. contrastándolas con propiedades de las MTs que **no** son propiedades de los conjuntos r.e.:

Conjuntos r.e.	Props. en MT
$L(M)$ es finito	M tiene al menos 481 estados
$L(M)$ es regular	M se detiene ante toda entrada
M acepta 101	M rechaza 0010
$L(M)$ es LLC	existe una MT M' mínima equivalente a M

En cada caso, en la columna de la izquierda se tienen propiedades que puede cumplir el lenguaje aceptado de la MT, mientras que en la columna de la derecha no se tienen propiedades de conjuntos, pues en cada caso se pueden contruir dos MTs que aceptan el mismo conjunto pero uno satisface la propiedad y el otro no.

Una propiedad no trivial es aquella que no es universalmente verdadera ni universalmente falsa: al menos debe existir un conjunto r.e. que satisface la propiedad y al menos uno que no la satisface. El lector puede inferir que sólo existen dos propiedades triviales y ambas son trivialmente decidibles.

Demostración: sea P una propiedad no trivial de \mathcal{RE} . Asúmase (sin pérdida de generalidad) que

³⁵Recuérdese la definición de problema de decisión presentada en los preliminares.

$P(\emptyset) = \perp$ (el argumento es similar si $P(\emptyset) = \top$). Dado que P es no trivial, debe existir un $A \in \mathcal{RE}$ tal que $P(A) = \top$. Sea K una MT que acepta a A .

Reduciremos el problema HP al conjunto $C = \{M \mid P(L(M)) = \top\}$, lo cual demostrará que C es indecible. Dado $M \# x$ constrúyase una MT $M' = \sigma(M \# x)$ que ante la entrada y

1. salva a y en una pista separada,
2. escribe a x es otra pista, (como en el caso de FIN)
3. ejecuta a M ante x ,
4. si M se detiene ante x , M' ejecuta a K sobre y y acepta si K acepta.

Si M no se detiene ante x , entonces la simulación no se detiene y la entrada y de M' no será aceptada, esto se cumple para toda y , lo cual significa que $L(M') = \emptyset$. Pero si M se detiene ante x entonces M' siempre alcanza el último paso y la entrada y será aceptada únicamente cuando la acepta a su vez K :

$$\begin{aligned} M \text{ se detiene ante } x &\Rightarrow L(M') = A \\ &\Rightarrow P(L(M')) = P(A) = \top, \\ M \text{ no se detiene ante } x &\Rightarrow L(M') = \emptyset \\ &\Rightarrow P(\emptyset) = \perp. \end{aligned}$$

Lo cual es la reducción de HP a C ; por lo tanto C no es recursivo y es indecible si $L(M)$ satisface P .

Ejercicios 9.8.1 Demuestre que las siguientes propiedades de conjuntos r.e no son r.e.:

1. $L = \emptyset$
2. $L = \Sigma^*$
3. L es recursivo
4. L no es recursivo
5. L tiene sólo un elemento
6. L es regular
7. $L - L(U) = \emptyset$.

Apéndice A

El Teorema de Incompletitud de Gödel

En 1931 Kurt Gödel demostró que ningún sistema de prueba para la teoría de números naturales puede demostrar todas las fórmulas (enunciados) verdaderas. En consecuencia, ningún sistema de prueba puede demostrar su propia consistencia. Con nuestra comprensión de las reducciones sobre conjuntos r.e., estamos listos para comprender éste teorema y ofrecer una descripción de la demostración. En la discusión, asumimos familiaridad con el lenguaje del cálculo de predicados de primer orden (lógica clásica) con igualdad y nos referiremos al mismo indistintamente como *lenguaje formal de la teoría de \mathcal{N}* .

Muchos conceptos útiles de la teoría de los números naturales pueden formalizarse en el lenguaje, e.g:

1. $\text{INTDIV}(x, y, q, r) \stackrel{\text{def}}{=} [x = q \times y + r \wedge r < y]$ “ q es el cociente y r el residuo de x entre y ”.
2. $\text{DIV}(y, x) \stackrel{\text{def}}{=} [\exists q \text{INTDIV}(x, y, q, 0)]$ “ y divide a x ”.
3. $\text{PRIMO}(x) \stackrel{\text{def}}{=} [x \geq 2 \wedge \forall y (\text{DIV}(y, x) \supset (y = 1 \wedge y = x))]$ “ x es primo”.
4. $\text{POTEN}_2(x) \stackrel{\text{def}}{=} [\forall y (\text{DIV}(y, x) \wedge \text{PRIMO}(y)) \supset y = 2]$ “ x es potencia de dos”.
5. $\text{BIT}(y, x) \stackrel{\text{def}}{=} [\text{POTEN}_2(y) \wedge \forall q, r (\text{INTDIV}(x, y, q, r) \supset \text{IMPAR}(q))]$ “ $y = 2^k$ y el k -ésimo bit en la representación binaria de x es 1”. Ésta representación es útil para tratar a los números como cadenas de bits y usarlos a su vez como índices para obtener otros números.

Como es habitual, cada enunciado tiene un valor de verdad bien definido¹ bajo su interpretación natural en \mathcal{N} :

¹Se asume que el lenguaje formal es libremente generado a partir del conjunto de variables proposicionales y cerrado bajo

- $P : \forall x \exists y [y = x + 1]$ “todo número tiene sucesor”.
- $Q : \forall x \exists y [x = y + 1]$ “todo número tiene predecesor”.

El primer enunciado es verdadero, ($\hat{v}(P) = \top$) mientras que el segundo es falso ($\hat{v}(Q) = \perp$), entendiéndo que $\hat{v}(B)$ denota la interpretación de B en el universo \mathcal{N} .

Aritmética de Peano

El sistema de prueba más popular para expresar la teoría de los números naturales se denomina la *aritmética de Peano* (PA); el cual consiste de un conjunto finito de axiomas:

1. $\forall x [\neg(0 = x + 1)]$ 0 no tiene sucesor
2. $\forall x [\forall y (x + 1 = y + 1 \supset x = y)]$ la primitiva *sucesor* es uno a uno
3. $\forall x [x + 0 = x]$ 0 es identidad para +
4. $\forall x \forall y [x + (y + 1) = (x + y) + 1]$ + es asociativo
5. $\forall x [x \times 0 = 0]$ 0 es aniquilador de \times
6. $\forall x \forall y [x \times (y + 1) = (x \times y) + x]$ \times se distribuye sobre +
7. $(\phi(0) \wedge \forall x [\phi(x) \supset \phi(x + 1)]) \supset \forall x \phi(x)$ axioma de inducción.

y una pareja de reglas de inferencia²:

$$\begin{array}{cc} \text{Modus ponens} & \text{Generalización} \\ \frac{\phi \quad \phi \supset \psi}{\psi} & \frac{\phi}{\forall x \phi} \end{array}$$

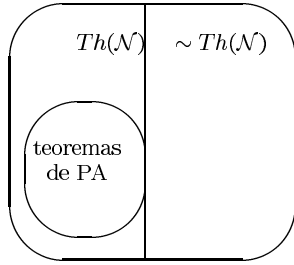
los constructores asociados a los conectivos, dicho de manera equivalente, la GLC correspondiente es no ambigua.

²En el sistema clásico de Hilbert.

Entre los axiomas de PA existen axiomas que se aplican a la lógica de primer orden en general, por ejemplo para manipular fórmulas proposicionales como $(\phi \wedge \psi) \supset \psi$, cuantificadores $(\forall x \phi(x)) \supset \phi(19)$ e igualdad, como $\forall x, y, z [x = y \wedge y = z \supset x = z]$.

Como es habitual, una *prueba* de ϕ_n es una secuencia $\phi_0, \phi_1, \dots, \phi_n$ de fórmulas tales que cada ϕ_i es ya sea un axioma o se sigue de fórmulas que ocurrieron anteriormente en la secuencia aplicando alguna regla de inferencia. Un enunciado del lenguaje se denomina *teorema* del sistema si tiene una prueba.

Un sistema de prueba se denomina *correcto* si todos los teoremas son verdaderos (i.e., no es posible demostrar un enunciado falso), lo cual es un requisito básico de todo sistema de inferencia *razonable*: si los teoremas fuesen falsos se podría demostrar cualquier cosa. El sistema PA es correcto, se puede demostrar por inducción sobre la longitud de las pruebas: todo axioma por definición es verdadero, y cualquier consecuencia derivada de las reglas de inferencia a partir de premisas verdaderas es verdadera. La correctitud de un sistema significa que la contención ilustrada en la Figura A.1 se cumple.



todas las formulas bien formadas

Figura A.1: Correctitud del sistema PA.

Un sistema de prueba es *completo* si todo enunciado verdadero es también un teorema: el conjunto de teoremas coincide con $Th(\mathcal{N})$ el cual denota el conjunto de enunciados verdaderos (teoría). Denotamos por $\models \phi$ para indicar que ϕ es verdadero, mientras que $\vdash \phi$ significa que ϕ es demostrable en el sistema.

A.2 Prueba del Teorema

Gödel demostró que para cualquier sistema de prueba razonable, se puede construir un enunciado de la teoría de los naturales ϕ , que afirma su propia no demostrabilidad en el sistema:

$$\models \phi \iff \nvdash \phi \quad (\text{A.1})$$

En cualquier sistema de prueba razonable (incluyendo por supuesto a PA) se cumple que para toda

ψ

$$\vdash \psi \Rightarrow \models \psi \quad (\text{A.2})$$

Por tanto, ϕ debe ser verdadero necesariamente dado que de otra forma

$$\begin{aligned} \nmodels \phi &\Rightarrow \vdash \phi && \text{por A.1} \\ &\Rightarrow \models \phi && \text{por A.2} \end{aligned}$$

lo cual es una contradicción. Dado que ϕ es verdadera, por A.1 no es demostrable.

La construcción de ϕ es muy interesante en sí misma dado que captura la noción de auto-referencia. Tal poder está presente en las Máquinas de Turing como también en varios lenguajes de programación modernos; por ejemplo, el siguiente programa en C se imprime a sí mismo:

```
char *s='char *s=%c%s%c;%cmain(){
printf(s,34,s,34,10,10);}';
main(){printf(s,34,s,34,10,10);}
```

(donde 34 y 10 corresponden a los códigos ascii para comillas dobles y retorno de carro, respectivamente). Si se entiende lo que hace el programa anterior se ha entendido la idea principal bajo la construcción de Gödel.

El argumento central es que en PA ó en cualquier otro sistema de prueba para la teoría de los naturales:

1. el conjunto de teoremas es r.e.,
2. el conjunto $Th(\mathcal{N})$ de enunciados verdaderos no es r.e.

Por tanto, los dos conjuntos no pueden ser iguales y el sistema de prueba no puede ser completo. Éste enfoque se debe a Turing.

El conjunto de teoremas de PA puede enumerarse mecánicamente enlistando primero todos los axiomas; y al aplicar sistemáticamente las reglas de inferencia en toda forma posible, eventualmente se genera todo enunciado demostrable.

Por tanto, la prueba se reduce a demostrar que

Lema A.2.1 ($Th(\mathcal{N})$ no es r.e.)

Demostración: se mostrará la existencia de la reducción $\sim HP \leq_m Th(\mathcal{N})$. Dado $M \# x$, se puede construir un enunciado γ en el lenguaje de la teoría de \mathcal{N} tal que

$$M \# x \in \sim HP \iff \gamma \in Th(\mathcal{N})$$

lo cual significa

$$M \text{ no se detiene ante } x \text{ si y sólo si } \gamma \text{ es verdadera} \\ (\hat{v}(\gamma) = \top)$$

Utilizando la fórmula $\text{BIT}(x, y)$ se puede construir una serie de fórmulas que culminen en la fórmula $\text{COMPVAL}_{M,x}(y)$ la cual dice que y representa una historia de cómputo (secuencia de configuraciones $\alpha_0, \alpha_1, \dots, \alpha_n$) válido de M ante la entrada x ; tal secuencia de configuraciones está codificada en un alfabeto Σ y cumple lo siguiente

1. α_0 es la configuración inicial de M ante x ,
2. α_{i+1} se sigue de α_i tras aplicar δ ,
3. α_n es la configuración en la que M se detiene.

El decir que M no se detiene ante x es equivalente a afirmar que no existe una historia de cómputo válido:

$$\gamma \stackrel{\text{def}}{=} \neg \exists y [\text{COMPVAL}_{M,x}(y)]$$

lo cual constituye la reducción de $\sim \text{HP}$ a $\text{Th}(\mathcal{N})$.

Los detalles de la construcción de γ son en sí muy interesantes, y una forma es como la presenta [Koz96]: se asume que el alfabeto sobre el cual se realiza la codificación tiene tamaño p para p un número primo, de tal forma que todo natural tiene una representación en p única. Se define al conjunto C formado por tuplas $(x_1, x_2, x_3, x_4, x_5, x_6)$ de valores codificados tales que si $\delta(q, a) = (p, b, R)$ entonces $(a, a, b, a, b, b) \in C$, i.e., para algunas configuraciones $\alpha_i = \omega a q a b \omega'$ se obtiene tras aplicar δ la siguiente configuración $\alpha_{i+1} = \omega a b p b \omega'$.

Se procede a construir enunciados verdaderos sobre la ejecución de M que culminen en el enunciado que M no se detiene ante x :

1. $\text{POT}_p(x) \stackrel{\text{def}}{=} [\forall z (\text{DIV}(z, x) \wedge \text{PRIMO}(z) \supset z = p) \supset y = 2]$ “El número x es potencia de p ” (donde p es primo y fijo que depende de M).
2. $\text{LONG}(v, d) \stackrel{\text{def}}{=} [\text{POT}_p(d) \wedge v < d]$ “El número d es potencia de p y especifica la longitud de la cadena $v \in \Sigma^*$ ”.
3. $\text{DIGITO}(v, y, b) \stackrel{\text{def}}{=} [\exists u, a (v = a + b \times y + u \times p \times y \wedge a < y \wedge b < p)]$ “El dígito en código primo base p de v en posición y es b ”.
4. $\text{3DIGS}(v, y, b, c, d) \stackrel{\text{def}}{=} [\exists u, a (v = a + by + cpy + dp^2y + up^3y \wedge a < y \wedge b < p \wedge c < p \wedge d < p)]$ “Los tres dígitos en código primo base p de v en posición y son b, c, d ”.

En los últimos dos enunciados se asume que y es potencia de p .

Ahora se presenta la utilidad de los anteriores enunciados: codificar movimientos de la MT M :

5. $\text{EMPATE}(v, y, z) \stackrel{\text{def}}{=} [\bigvee_c (\text{3DIGS}(v, y, a, b, c) \wedge \text{3DIGS}(v, z, d, e, f))]$ (donde $c = (a, b, c, d, e, f) \in C$, i.e., representa toda tupla codificada de las configuraciones) “Los tres dígitos en código primo base p de v en posición y coinciden con los 3 dígitos (codificados también en base p) de v en posición z de acuerdo a δ ”.
6. $\text{MOV}(v, c, d) \stackrel{\text{def}}{=} [\forall y (\text{POT}_p(y) \wedge yp^2c < d) \supset \text{EMPATE}(v, y, yc)]$ “La cadena v representa una secuencia de configuraciones sucesivas de M , cuya longitud va de c a d ”.
7. $\text{INICIO}(v, c) \stackrel{\text{def}}{=} [\bigwedge_{i=0}^n \text{DIGITO}(v, p^i, k_i) \wedge p^n < c \wedge \forall y (\text{POT}_p(y) \wedge p^n < y < c \supset \text{DIGITO}(v, t, k))]$ “La cadena v describe la configuración inicial de M ante x de longitud c ” (asumiendo que c es potencia de p ; n y p^i son constantes fijas que dependen de M).
8. $\text{FIN}(v, d) \stackrel{\text{def}}{=} [\exists y (\text{POT}_p(y) \wedge y < d \wedge \bigvee_{a \in H} \text{DIGITO}(v, y, a))]$ (donde H es el conjunto de todos los dígitos codificados base p que corresponden a los símbolos de Σ que contienen estados de detención) “La cadena v contiene un estado de detención”.
9. $\text{COMPVAL}_{M,x}(v) \stackrel{\text{def}}{=} [\exists c, d (\text{POT}_p(c) \wedge c < d \wedge \text{LONG}(v, d) \wedge \text{INICIO}(v, c) \wedge \text{MOV}(v, c, d) \wedge \text{FIN}(v, d))]$ “La cadena v representa una historia de cómputo válido de M ante x ”.

Y finalmente

10. $\neg \exists v \text{COMPVAL}_{M,x}(v)$ “no existe una secuencia de configuraciones de M ante x que termine” (i.e., M no se detiene ante x).

Lo cual finaliza la demostración.

Bibliografía

- [BC94] Daniel Pierre Bovet and Pierluigi Crescenzi. *Introduction to the Theory of Complexity*. Prentice Hall, 1994.
- [HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley Publishing Co., 1979.
- [Koz96] Dexter C. Kozen. Automata and computability. <http://www.cs.cornell.edu/Info/courses/Fall-96/syllabus.html>, Aug 1996. Dept. of Computer Science, Cornell University.
- [Mit96] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
- [Smu95] Raymond Smullyan. *Santán, Cantor y el infinito*. Gedisa, 1995.
- [Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: an Introduction*. MIT Press, 1993.