Lenguajes Formales y Autómatas

José de Jesús Lavalle Martínez
FCC, BUAP
Traducción de partes del libro
An Introduction to Formal Languages and Autómata
Linz P. 6th Edition (2017)
jlavalle@cs.buap.mx

Índice general

Capítulo 1

Conceptos preliminares

Tres ideas fundamentales son los temas principales de este curso: lenguajes, gramáticas y autómatas. En el transcurso de nuestro estudio, exploraremos muchos resultados sobre estos conceptos y sobre su relación entre sí. Pero primero, debemos comprender el significado de los términos.

1.1. Lenguajes

Todos estamos familiarizados con la noción de lenguajes naturales, como el español, el inglés y el francés. Aún así, para la mayoría de nosotros probablemente resultará difícil decir exactamente lo que significa la palabra "lenguaje".

Los diccionarios definen informalmente el término como un sistema adecuado para la expresión de ciertas ideas, hechos o conceptos, incluyendo un conjunto de símbolos y reglas para su manipulación. Si bien esto nos da una idea intuitiva de lo que es un lenguaje, no es suficiente como definición para el estudio de los lenguajes formales. Necesitamos una definición precisa del término.

Comenzamos con un conjunto Σ, finito y no vacío, de símbolos, llamado **alfabeto**. A partir de los símbolos individuales, construimos **cadenas**, que son secuencias finitas de símbolos del alfabeto.

Por ejemplo, si el alfabeto $\Sigma = \{a, b\}$, entonces $abab$ y $aaabbba$ son cadenas en Σ. Con pocas excepciones, usaremos letras minúsculas $a, b, c, \ldots$ para elementos de Σ y $u, v, w, \ldots$ para nombres de cadenas. Escribiremos,

por ejemplo,

$$w = abaaa$$

para indicar que la cadena denominada w tiene el valor específico $abaaa$.

La **concatenación** de dos cadenas w y v es la cadena que se obtiene añadiendo los símbolos de v al extremo derecho de w, es decir, si

$$w = a_1 a_2 \cdots a_n$$

y

$$v = b_1 b_2 \cdots b_m,$$

entonces la concatenación de w y v, denotada por wv, es

$$wv = a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m.$$

La **reversa** de una cadena se obtiene escribiendo los símbolos en orden inverso; si w es una cadena como se muestra arriba, entonces su reversa w^R es

$$w^R = a_n \cdots a_2 a_1.$$

La **longitud** de una cadena w, denotada por $|w|$, es el número de símbolos en la cadena. Con frecuencia necesitaremos hacer referencia a la cadena vacía, que es una cadena sin ningún símbolo. Será denotada por λ. Las siguientes relaciones simples

$$|\lambda| = 0,$$
$$\lambda w = w\lambda = w$$

son válidas para toda w.

Cualquier cadena de símbolos consecutivos en w es una **subcadena** de w. Si

$$w = vu,$$

entonces se dice que las subcadenas v y u son un **prefijo** y un **sufijo** de w, respectivamente. Por ejemplo, si $w = abbab$, entonces $\{\lambda, a, ab, abb, abba, abbab\}$ es el conjunto de todos los prefijos de w, mientras que bab, ab, b son algunos de sus sufijos.

Las propiedades simples de las cadenas, como su longitud, son muy intuitivas y probablemente necesiten poca elaboración. Por ejemplo, si u y v son

cadenas, entonces la longitud de su concatenación es la suma de las longitudes individuales, es decir,

$$|uv| = |u| + |v|. \tag{1.1}$$

Pero aunque esta relación es obvia, es útil poder precisarla y probarla. Las técnicas para hacerlo son importantes en situaciones más complicadas.

Ejemplo 1.1 Demuestre que (1.1) se cumple para cualquier u y v. Para probar esto, primero necesitamos una definición de la longitud de una cadena. Hacemos tal definición de manera recursiva mediante

$$\begin{aligned} |a| &= 1, \\ |wa| &= |w| + 1, \end{aligned}$$

para todo $a \in \Sigma$ y w cualquier cadena en Σ. Esta definición es una declaración formal de nuestra comprensión intuitiva de la longitud de una cadena: la longitud de un solo símbolo es uno, y la longitud de cualquier cadena aumenta en uno si le agregamos otro símbolo. Con esta definición formal, estamos listos para probar (1.1) mediante inducción.

Por definición, (1.1) se cumple para todo u de cualquier longitud y todo v de longitud 1, por lo que tenemos que la base de la inducción se cumple. Como suposición inductiva, asumimos que (1.1) se cumple para todo u de cualquier longitud y todo v de longitud $1, 2, \ldots, n$. Ahora tome cualquier v de longitud $n + 1$ y escríbala como $v = wa$. Luego,

$$\begin{aligned} |v| &= |w| + 1, \\ |uv| &= |uwa| = |uw| + 1. \end{aligned}$$

Por la hipótesis inductiva (que es aplicable ya que w es de longitud n),

$$|uw| = |u| + |w|$$

así que

$$|uv| = |u| + |w| + 1 = |u| + |v|.$$

Por lo tanto, (1.1) se cumple para todo u y todo v de longitud hasta $n + 1$, completando el paso inductivo y la demostración. □

Si w es una cadena, entonces w^n representa la cadena obtenida al concatenar w n veces. Como caso especial, definimos

$$w^0 = \lambda,$$

para toda w.

Si Σ es un alfabeto, usamos Σ^* para denotar el conjunto de cadenas obtenido al concatenar cero o más símbolos de Σ. El conjunto Σ^* siempre contiene λ. Para excluir la cadena vacía, definimos

$$\Sigma^+ = \Sigma^* - \{\lambda\}.$$

Mientras que Σ es finito por suposición, Σ^* y Σ^+ son siempre infinitos ya que no hay límite en la longitud de las cadenas en estos conjuntos. Un lenguaje se define muy generalmente como un subconjunto de Σ^*. Una cadena en un lenguaje L se llamará **oración** de L.

Esta definición es bastante amplia; cualquier conjunto de cadenas de un alfabeto Σ puede considerarse un lenguaje. Más adelante estudiaremos métodos mediante los cuales se pueden definir y describir lenguajes específicos; esto nos permitirá estructurar este concepto bastante amplio. Por el momento, sin embargo, solo veremos algunos ejemplos específicos.

Ejemplo 1.2 Sea $\Sigma = \{a, b\}$. Entonces

$$\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}.$$

El lenguaje

$$\{a, aa, aab\}$$

es un lenguaje sobre Σ. Debido a que tiene un número finito de oraciones, lo llamamos lenguaje finito. El conjunto

$$L = \{a^n b^n : n \geq 0\}$$

también es un lenguaje sobre Σ. Las cadenas *aabb* y *aaaabbbb* están en el lenguaje L, pero la cadena *abb* no está en L. Este lenguaje es infinito. Los lenguajes más interesantes son infinitos. □

Dado que los lenguajes son conjuntos, la unión, intersección y diferencia de dos lenguajes se definen inmediatamente. El complemento de un lenguaje se define con respecto a Σ^*; es decir, el complemento de L es

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

La reversa de un lenguaje L es el conjunto de todas las reversas de las cadenas de L, es decir,

$$L^R = \{w^R : w \in L\}.$$

La concatenación de dos lenguajes L_1 y L_2 es el conjunto de todas las cadenas obtenidas al concatenar cualquier elemento de L_1 con cualquier elemento de L_2; específicamente,

$$L_1L_2 = \{xy : x \in L_1, y \in L_2\}.$$

Definimos L^n como L concatenado consigo mismo n veces, con los casos especiales

$$L^0 = \{\lambda\} \text{ y } L^1 = L,$$

para todo lenguaje L.

Finalmente, definimos la **cerradura estrella** de un lenguaje como

$$L^* = \bigcup_{i \geq 0} L^i$$

y la **cerradura positiva** como

$$L^+ = \bigcup_{i \geq 1} L^i$$

Ejemplo 1.3 Si

$$L = \{a^n b^n : n \geq 0\},$$

entonces

$$L^2 = \{a^n b^n a^m b^m : n \geq 0, m \geq 0\}.$$

Tenga en cuenta que n y m no están relacionados; la cadena aabbaaabbb está en L_2.

La reversa de L se describe fácilmente en notación de conjuntos como

$$L^R = \{b^n a^n : n \geq 0\},$$

pero es considerablemente más difícil describir $\overline{L}$ o L^* de esta manera. Unos pocos intentos lo convencerán rápidamente de la limitación de la notación de conjuntos para la especificación de lenguajes complicados. □

1.2. Gramáticas

Para estudiar lenguajes matemáticamente, necesitamos un mecanismo para describirlos. El lenguaje cotidiano es impreciso y ambiguo, por lo que las

descripciones informales en lenguaje natural a menudo son inadecuadas. La notación de conjuntos utilizada en los Ejemplos 1.2 y 1.3 es más adecuada, pero limitada.

A medida que avancemos, aprenderemos sobre varios mecanismos de definición de lenguajes que son útiles en diferentes circunstancias. Aquí presentamos uno común y poderoso, la noción de **gramática**.

Una gramática para el idioma inglés nos dice si una oración en particular está bien formada o no. Una regla típica de la gramática inglesa es "una oración puede constar de una frase nominal seguida de un predicado". Más concisamente escribimos esto como

$$< sentence > \rightarrow < noun\_phrase >< predicate >,$$

con la interpretación obvia. Por supuesto, esto no es suficiente para tratar con oraciones reales. Ahora debemos proporcionar definiciones para los constructos recién introducidos $< noun\_phrase >$ y $< predicate >$. Si lo hacemos mediante

$$< noun\_phrase > \rightarrow < article >< noun >,$$
$$< predicate > \rightarrow < verb >$$

y si asociamos las palabras "a" y "the" con $< article >$, "boy" y "dog" con $< noun >$, y "runs" y "walks" con $< verb >$, entonces la gramática nos dice que las oraciones "a boy runs" y "the dog walks" están correctamente formadas. Si tuviéramos que dar una gramática completa, entonces, en teoría, cada oración bien formada podría explicarse de esta manera.

Este ejemplo ilustra la definición de un concepto general en términos de conceptos simples. Comenzamos con el concepto de nivel superior, aquí $< sentence >$, y lo reducimos sucesivamente a los bloques de construcción irreductibles del lenguaje. La generalización de estas ideas nos lleva a gramáticas formales.

Definición 1.1 Una gramática G se define como un cuadruple

$$G = (V, T, S, P),$$

donde

- V es un conjunto finito de objetos llamados **variables**,

- T es un conjunto finito de objetos llamados **símbolos terminales**,
- $S \in V$ es un símbolo especial llamado variable **inicial**,
- P es un conjunto finito de **producciones**.

Se supondrá, sin más mención, que los conjuntos V y T no están vacíos y que son disjuntos.

Las reglas de producción son el corazón de una gramática; especifican cómo la gramática transforma una cadena en otra, y a través de esto definen un lenguaje asociado a la gramática. En nuestra discusión asumiremos que todas las reglas de producción son de la forma

$$x \to y,$$

donde x es un elemento de $(V \cup T)^+$ y y está en $(V \cup T)^*$. Las producciones se aplican de la siguiente manera: Dada una cadena w de la forma

$$w = uxv,$$

decimos que la producción $x \to y$ es aplicable a esta cadena, y podemos usarla para reemplazar x con y, obteniendo así una nueva cadena

$$z = uyv.$$

Esto se escribe como

$$w \Rightarrow z.$$

Decimos que w **deriva** z o que z se deriva de w. Las cadenas sucesivas se derivan aplicando las producciones de la gramática en orden arbitrario. Una producción se puede utilizar siempre que sea aplicable y se puede aplicar tantas veces como se desee. Si

$$w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n,$$

decimos que w_1 deriva w_n y lo denotamos mediante

$$w_1 \overset{*}{\Rightarrow} w_n.$$

El $*$ indica que se puede tomar un número no especificado de pasos (incluido cero) para derivar w_n partiendo de w_1.

Al aplicar las reglas de producción en un orden diferente, una gramática determinada normalmente puede generar muchas cadenas. El conjunto de todas esas cadenas terminales es el lenguaje definido o generado por la gramática.

Definición 1.2 Sea $G = (V, T, S, P)$ una gramática. Entonces el conjunto

$$L(G) = \left\{ w \in T^* : S \overset{*}{\Rightarrow} w \right\}$$

es el lenguaje generado por G.

Si $w \in L(G)$, entonces la secuencia

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n \Rightarrow w$$

es una derivación de la oración w. Las cadenas $S, w_1, w_2, \ldots, w_n$, que contienen tanto variables como terminales, se denominan formas oracionales de la derivación.

Ejemplo 1.4 Considere la gramática

$$G = (\{S\}, \{a, b\}, S, P)$$

con P dado por

$$S \to aSb,$$
$$S \to \lambda.$$

Entonces

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb,$$

así que podemos escribir

$$S \overset{*}{\Rightarrow} aabb.$$

La cadena $aabb$ es una oración en el lenguaje generado por G, mientras que $aaSbb$ es una forma oracional.

Una gramática G define completamente a $L(G)$, pero puede que no sea fácil obtener una descripción muy explícita del lenguaje a partir de la gramática. Aquí, sin embargo, la respuesta es bastante clara. No es difícil conjeturar que

$$L(G) = \{a^n b^n; n \geq 0\},$$

y es fácil probarlo. Si notamos que la regla $S \to aSb$ es recursiva, se sugiere inmediatamente una demostración por inducción. Primero mostramos que todas las formas oracionales deben tener la forma

$$w_i = a^i S b^i. \tag{1.2}$$

Suponga que (1.2) se cumple para todas las formas oracionales w_i de longitud $2i + 1$ o menos. Para obtener otra forma oracional (que no es una oración), solo podemos aplicar la producción $S \to aSb$. Esto nos da

$$a^i S b^i \Rightarrow a^{i+1} S b^{i+1},$$

de modo que toda forma oracional de longitud $2i + 3$ también tiene la forma (1.2). Dado que (1.2) es obviamente cierto para $i = 1$, se cumple por inducción para todo i. Finalmente, para obtener una oración, debemos aplicar la producción $S \to \lambda$, y vemos que

$$S \overset{*}{\Rightarrow} a^n S b^n \Rightarrow a^n b^n$$

representa todas las posibles derivaciones. Por tanto, G sólo puede derivar cadenas de la forma $a^n b^n$.

También tenemos que demostrar que se pueden derivar todas las cadenas de esta forma. Esto es fácil; simplemente aplicamos $S \to aSb$ tantas veces como sea necesario, seguido de $S \to \lambda$. □

Ejemplo 1.5 Encuentra una gramática que genere

$$L = \{a^n b^{n+1} : n \geq 0\}.$$

La idea detrás del ejemplo anterior se puede extender a este caso. Todo lo que tenemos que hacer es generar una b extra. Esto se puede hacer con una producción $S \to Ab$ y con otras producciones elegidas para que A pueda derivar el lenguaje del ejemplo anterior. Razonando de esta manera, obtenemos la gramática $G = (\{S, A\}, \{a, b\}, S, P)$, con producciones

$$\begin{aligned} S &\to Ab, \\ A &\to aAb, \\ A &\to \lambda. \end{aligned}$$

Derive algunas oraciones específicas para convencerse de que esto funciona. □

Los ejemplos anteriores son bastante fáciles, por lo que argumentos rigurosos pueden parecer superfluos. Pero a menudo no es tan fácil encontrar una gramática para un lenguaje descrito de manera informal o dar una caracterización intuitiva del lenguaje definido por una gramática.

Para mostrar que un lenguaje dado es generado por una determinada gramática G, debemos ser capaces de demostrar (a) que cada $w \in L$ puede derivarse de S usando G y (b) que cada cadena así derivada está en L.

Ejemplo 1.6 Tome $\Sigma = \{a, b\}$, y sean $n_a(w)$ y $n_b(w)$ el número de símbolos a y b en la cadena w, respectivamente. Luego la gramática G con producciones

$$\begin{aligned} S &\to SS, \\ S &\to \lambda, \\ S &\to aSb, \\ S &\to bSa \end{aligned}$$

genera el lenguage

$$L = \{w : n_a(w) = n_b(w)\}.$$

Esta afirmación no es tan obvia y debemos proporcionar argumentos convincentes.

En primer lugar, está claro que cada forma oracional de G tiene el mismo número de símbolos a y b, ya que las únicas producciones que generan una a, a saber, $S \to aSb$ y $S \to bSa$, generan simultáneamente una b. Por lo tanto, cada elemento de $L(G)$ está en L. Es un poco más difícil ver que cada cadena en L se puede derivar con G.

Comencemos por analizar el problema en resumen, considerando las diversas formas que puede tener $w \in L$. Suponga que w comienza con a y termina con b. Entonces tiene la forma

$$w = aw_1b,$$

donde w_1 también está en L. Podemos pensar en este caso como que es derivado a partir de

$$S \Rightarrow aSb$$

si S de hecho deriva cualquier cadena en L. Se puede hacer un argumento similar si w comienza con b y termina con a. Pero esto no se ocupa de todos los casos, ya que una cadena en L puede comenzar y terminar con el mismo

símbolo. Si escribimos una cadena de este tipo, digamos $aabbba$, vemos que se puede considerar como la concatenación de dos cadenas más cortas $aabb$ y ba, ambas en L.

¿Es esto cierto en general? Para mostrar que esto es así, podemos usar el siguiente argumento: Supongamos que, comenzando en el extremo izquierdo de la cadena, contamos +1 para una a y -1 para una b.

Si una cadena w comienza y termina con a, entonces la cuenta será +1 después del símbolo más a la izquierda y -1 inmediatamente antes del más a la derecha. Por lo tanto, la cuenta debe pasar por cero en algún lugar en el medio de la cadena, lo que indica que dicha cadena debe tener la forma

$$w = w_1 w_2$$

donde w_1 y w_2 están en L. Este caso se puede resolver con la producción $S \to SS$.

Una vez que vemos el argumento de manera intuitiva, estamos listos para proceder con más rigor. Nuevamente usamos inducción. Suponga que todo $w \in L$ con $|w| \leq 2n$ se puede derivar con G. Tome cualquier $w \in L$ de longitud $2n + 2$. Si $w = aw_1b$, entonces w_1 está en L y $|w_1| = 2n$. Por lo tanto, por suposición,

$$S \stackrel{*}{\Rightarrow} w_1$$

Entonces

$$S \Rightarrow aSb \stackrel{*}{\Rightarrow} aw_1b = w$$

es posible, y w se puede derivar con G. Obviamente, se pueden hacer argumentos similares si $w = bw_1a$.

Si w no tiene esta forma, es decir, si comienza y termina con el mismo símbolo, entonces el argumento de conteo nos dice que debe tener la forma $w = w_1w_2$, con w_1 y w_2 ambos en L y de longitud menor que o igual a $2n$. Por lo tanto, nuevamente vemos que

$$S \Rightarrow SS \stackrel{*}{\Rightarrow} w_1S \stackrel{*}{\Rightarrow} w_1w_2 = w$$

es posible.

Ya que la hipótesis inductiva se satisface claramente para $n = 1$, se cumple el caso base y la afirmación es verdadera para todo n, completando la demostración. □

Normalmente, un lenguaje dado tiene muchas gramáticas que lo generan. Aunque estas gramáticas son diferentes, son equivalentes en cierto sentido.

Decimos que dos gramáticas G_1 y G_2 son equivalentes si generan el mismo lenguaje, es decir, si

$$L(G_1) = L(G_2).$$

Como veremos más adelante, no siempre es fácil ver si dos gramáticas son equivalentes.

Ejemplo 1.7 Considere la gramática $G_1 = (\{A, S\}, \{a, b\}, S, P_1)$, con P_1 que consta de las producciones

$$S \to aAb|\lambda,$$
$$A \to aAb|\lambda.$$

Aquí presentamos una notación abreviada conveniente en la que varias reglas de producción con los mismos lados izquierdos se escriben en la misma línea, con los lados derechos alternativos separados por $|$. En esta notación $S \to aAb|\lambda$ representa las dos producciones $S \to aAb$ y $S \to \lambda$.

Esta gramática es equivalente a la gramática G del ejemplo 1.4. La equivalencia es fácil de probar mostrando que

$$L(G_1) = \{a^n b^n : n \geq 0\}.$$

□

1.3. Autómatas

Un autómata es un modelo abstracto de una computadora digital. Como tal, cada autómata incluye algunas características esenciales. Tiene un mecanismo de lectura de entrada. Se asumirá que la entrada es una cadena sobre un alfabeto dado, escrita en un archivo de entrada, que el autómata puede leer pero no cambiar.

El archivo de entrada se divide en celdas, cada una de las cuales puede contener un símbolo. El mecanismo de entrada puede leer el archivo de entrada de izquierda a derecha, un símbolo a la vez. El mecanismo de entrada también puede detectar el final de la cadena de entrada (detectando una condición de final de archivo).

El autómata puede producir una salida de alguna forma. Puede tener un dispositivo de **almacenamiento** temporal, que consta de un número ilimitado de celdas, cada una de las cuales puede contener un solo símbolo

de un alfabeto (no necesariamente el mismo que el alfabeto de entrada). El autómata puede leer y cambiar el contenido de las celdas de almacenamiento.

Finalmente, el autómata tiene una **unidad de control**, que puede estar en cualquiera de un número finito de **estados internos**, y que puede cambiar de estado de alguna manera definida. La Figura 1.1 muestra una representación esquemática de un autómata general.

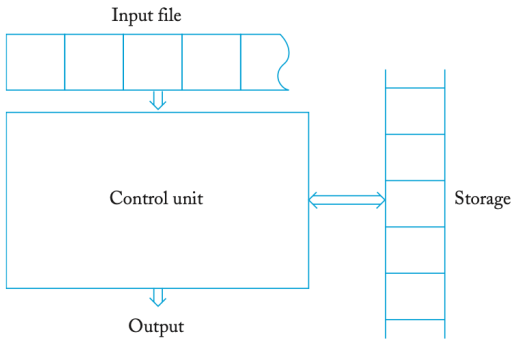


Figura 1.1: Representación esquemática de un autómata general.

Se supone que un autómata opera en un marco de tiempo discreto. En cualquier momento dado, la unidad de control se encuentra en algún estado interno y el mecanismo de entrada está escaneando un símbolo particular en el archivo de entrada.

El estado interno de la unidad de control en el siguiente paso de tiempo lo determina la **función de transición** o **estado siguiente**. Esta función de transición proporciona el siguiente estado en términos del estado actual, el símbolo de entrada actual y la información que se encuentra actualmente en el almacenamiento temporal.

Durante la transición de un intervalo de tiempo al siguiente, se puede producir una salida o cambiar la información del almacenamiento temporal. El término **configuración** se utilizará para referirse a un estado particular de la unidad de control, archivo de entrada y almacenamiento temporal. La transición del autómata de una configuración a la siguiente se denominará **movimiento**.

Este modelo general cubre todos los autómatas que discutiremos en este curso. Un control de estado finito será común a todos los casos específicos, pero las diferencias surgirán de la forma en que se puede producir la salida y la naturaleza del almacenamiento temporal. Como veremos, la naturaleza del

almacenamiento temporal gobierna el poder de diferentes tipos de autómatas.

Para discusiones posteriores, será necesario distinguir entre autómatas deterministas y autómatas no deterministas.

Un autómata determinista es aquel en el que cada movimiento está determinado de forma única por la configuración actual. Si conocemos el estado interno, la entrada y el contenido del almacenamiento temporal, podemos predecir exactamente el comportamiento futuro del autómata.

En un autómata no determinista, esto no es así. En cada punto, un autómata no determinista puede tener varios movimientos posibles, por lo que solo podemos predecir un conjunto de acciones posibles. La relación entre los autómatas deterministas y no deterministas de varios tipos jugará un papel importante en nuestro estudio.

Un autómata cuya respuesta de salida se limita a un simple "sí" o "no" se denomina **aceptador**. Presentado con una cadena de entrada, un aceptador acepta la cadena o la rechaza. Un autómata más general, capaz de producir cadenas de símbolos como salida, se llama **transductor**.

1.4. Ejercicios

1. ¿Cuántas subcadenas aab hay en ww^Rw, donde $w = aabbab$?

2. Use inducción sobre n para demostrar que $|u^n| = n|u|$ para todas las cadenas u y para todo n.

3. El reverso de una cadena, introducido informalmente anteriormente, se puede definir con mayor precisión mediante las reglas recursivas

$$a^R = a,$$
$$(wa)^R = aw^R,$$

   para todo $a \in \Sigma$, $w \in \Sigma^*$. Use esto para demostrar que

$$(uv)^R = v^R u^R,$$

   para todo $u, v \in \Sigma^+$.

4. Sea $L = \{ab, aa, baa\}$. ¿Cuáles de las siguientes cadenas están en L^* : $abaabaaabaa, aaaabaaaa, baaaaabaaaab, baaaaabaa$? ¿Qué cadenas están en L^4?

5. Sea $\Sigma = \{a, b\}$ y $L = \{aa, bb\}$. Utilice la notación de conjuntos para describir $\overline{L}$.

6. Demuestre que
$$(L_1 L_2)^R = L_2^R L_1^R$$
para todos los lenguajes L_1 y L_2.

7. Encuentre una gramática para el lenguaje $L = \{a^n, \text{ donde n es par}\}$.

8. Dé una descripción sencilla del lenguaje generado por la gramática con producciones
$$S \to aaA,$$
$$A \to bS,$$
$$S \to \lambda.$$

9. Encuentre tres cadenas en el lenguaje generado por
$$S \to aSb|bSa|a.$$

10. Complete los argumentos en el Ejemplo 1.7, mostrando que $L(G_1)$ en efecto genera el lenguaje dado en el Ejemplo 1.4.

Capítulo 2

Autómatas finitos

Nuestra introducción en el primer capítulo a los conceptos básicos de computación, particularmente la discusión de los autómatas, es breve e informal. En este punto, solo tenemos una comprensión general de qué es un autómata y cómo se puede representar mediante un grafo.

Para progresar, debemos ser más precisos, brindar definiciones formales y comenzar a desarrollar resultados rigurosos. Comenzamos con los aceptadores finitos, que son un caso especial simple del esquema general presentado en el capítulo anterior. Este tipo de autómatas se caracteriza por no tener almacenamiento temporal.

Dado que un archivo de entrada no se puede reescribir, un autómata finito está severamente limitado en su capacidad para "recordar" cosas durante el cálculo. Se puede retener una cantidad finita de información en la unidad de control colocando la unidad en un estado específico.

Pero dado que el número de tales estados es finito, un autómata finito solo puede lidiar con situaciones en las que la información que se almacenará en cualquier momento está estrictamente limitada.

2.1. Aceptadores finitos deterministas

Definición 2.1 Un **aceptador finito determinista** o **dfa** se define por el quintuple $M = (Q, \Sigma, \delta, q_0, F)$, donde

- Q es un conjunto finito de estados internos,
- Σ es un conjunto finito de símbolos llamado alfabeto de entrada,

- $\delta : Q \times \Sigma \to Q$ es una función total llamada función de transición,
- $q_0 \in Q$ es el estado inicial,
- $F \subseteq Q$ es un conjunto de estados finales.

Un aceptador finito determinista opera de la siguiente manera. En el momento inicial, se supone que está en el estado inicial q_0, con su mecanismo de entrada en el símbolo más a la izquierda de la cadena de entrada. Durante cada movimiento del autómata, el mecanismo de entrada avanza una posición hacia la derecha, por lo que cada movimiento consume un símbolo de entrada.

Cuando se alcanza el final de la cadena, la cadena se acepta si el autómata se encuentra en uno de sus estados finales. De lo contrario, la cadena se rechaza. El mecanismo de entrada solo puede moverse de izquierda a derecha y lee exactamente un símbolo en cada paso. Las transiciones de un estado interno a otro se rigen por la función de transición δ. Por ejemplo, si

$$\delta(q_0, a) = q_1,$$

entonces, si el dfa está en el estado q_0 y el símbolo de entrada actual es a, el dfa pasará al estado q_1.

Al hablar de los autómatas, es esencial tener una imagen clara e intuitiva con la que trabajar. Para visualizar y representar autómatas finitos, utilizamos grafos de transición, en los que los vértices representan estados y las aristas representan transiciones. Las etiquetas en los vértices son los nombres de los estados, mientras que las etiquetas en las aristas son los valores actuales del símbolo de entrada.

Por ejemplo, si q_0 y q_1 son estados internos de algún dfa M, entonces el grafo asociado con M tendrá un vértice etiquetado como q_0 y otro etiquetado como q_1. Una arista etiquetada (q_0, q_1) representa la transición $\delta(q_0, a) = q_1$. El estado inicial será identificado por una flecha entrante sin etiqueta que no se origina en ningún vértice. Los estados finales se dibujan con un círculo doble.

Más formalmente, si $M = (Q, \Sigma, \delta, q_0, F)$ es un aceptador finito determinista, entonces su grafo de transición asociado G_M tiene exactamente $|Q|$ vértices, cada uno etiquetado con un $q_i \in Q$ diferente.

Para cada regla de transición $\delta(q_i, a) = q_j$, el grafo tiene una arista (q_i, q_j) etiquetada a. El vértice asociado con q_0 se llama vértice inicial, mientras que

los etiquetados con $q_f \in F$ son los vértices finales. Es una cuestión trivial convertir de la definición $(Q, \Sigma, \delta, q_0, F)$ de un dfa a su representación como grafo de transición y viceversa.

Ejemplo 2.1 El grafo de la Figura 2.1 representa al dfa

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q1\}),$$

donde δ está definida mediante

$$\begin{aligned} \delta(q_0, 0) &= q_0, & \delta(q_0, 1) &= q_1, \\ \delta(q_1, 0) &= q_0, & \delta(q_1, 1) &= q_2, \\ \delta(q_2, 0) &= q_2, & \delta(q_2, 1) &= q_1. \end{aligned}$$

Este dfa acepta la cadena 01. Comenzando en el estado q_0, se lee primero el símbolo 0. Mirando las aristas del grafo, vemos que el autómata permanece en el estado q_0. A continuación, se lee el 1 y el autómata pasa al estado q_1. Ahora estamos al final de la cadena y, al mismo tiempo, en un estado final q_1. Por tanto, se acepta la cadena 01.

El dfa no acepta la cadena 00, ya que después de leer dos ceros consecutivos, estará en el estado q_0. Con un razonamiento similar, vemos que el autómata aceptará las cadenas 101, 0111 y 11001, pero no 100 ni 1100.

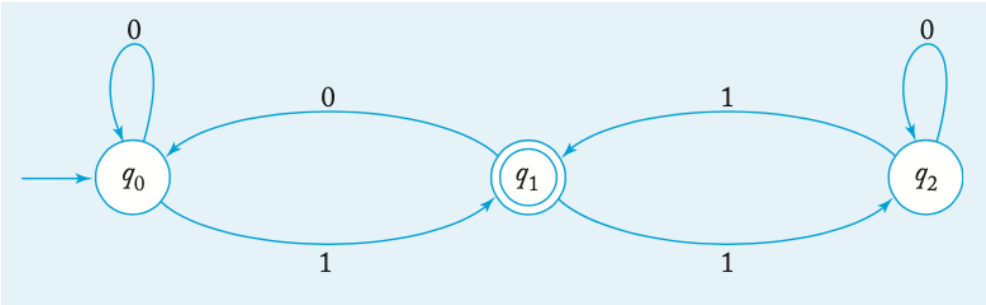


Figura 2.1: Grafo que representa al autómata del Ejemplo 2.1.

□

Es conveniente introducir la función de transición extendida $\delta^* : Q \times \Sigma^* \to Q$. El segundo argumento de δ^* es una cadena, en lugar de un solo símbolo,

y su valor da el estado en el que se encontrará el autómata después de leer esa cadena . Por ejemplo, si

$$\delta(q_0, a) = q_1 \tag{2.1}$$

y

$$\delta(q_1, b) = q_2, \tag{2.2}$$

entonces

$$\delta^*(q_0, ab) = q_2.$$

Formalmente, podemos definir recursivamente δ^* mediante

$$\delta^*(q, \lambda) = q, \tag{2.3}$$
$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2.4}$$

para todo $q \in Q$, $w \in \Sigma^*$ y $a \in \Sigma$. Para ver por qué esto es apropiado, apliquemos estas definiciones al caso simple anterior.

$$\begin{aligned}
\delta^*(q_0, ab) &\overset{2,4}{=} \delta(\delta^*(q_0, a), b) \\
&\overset{2,4}{=} \delta(\delta(\delta^*(q_0, \lambda), a), b) \\
&\overset{2,3}{=} \delta(\delta(q_0, a), b) \\
&\overset{2,1}{=} \delta(q_1, b) \\
&\overset{2,2}{=} q_2,
\end{aligned}$$

como se esperaba.

2.1.1. Lenguajes y DFAs

Una vez dada una definición precisa de un aceptador, ahora estamos listos para definir formalmente lo que entendemos por lenguaje asociado. La asociación es obvia: el lenguaje es el conjunto de todas las cadenas aceptadas por el autómata.

Definición 2.2 El lenguaje aceptado por un dfa $M = (Q, \Sigma, \delta, q_0, F)$ es el conjunto de todas las cadenas en Σ aceptadas por M. En notación formal,

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

Tenga en cuenta que requerimos que δ, y en consecuencia δ^*, sean funciones totales. En cada paso, se define un movimiento único, por lo que se justifica llamar determinista a dicho autómata. Un dfa procesará cada cadena en Σ^* y la aceptará o no la aceptará. No aceptación significa que el dfa se detiene en un estado no final, de modo que

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

Ejemplo 2.2 Considere el dfa en la Figura 2.2.

Al dibujar la Figura 2.2 permitimos el uso de dos etiquetas en una sola arista. Estas aristas con etiquetas múltiples son una forma abreviada de dos o más transiciones distintas: La transición se toma siempre que el símbolo de entrada coincide con cualquiera de las etiquetas de la arista.

El autómata de la Figura 2.2 permanece en su estado inicial q_0 hasta que se encuentra la primera b. Si éste es también el último símbolo de la entrada, entonces se acepta la cadena. De lo contrario, el dfa entra en el estado q_2, del que nunca podrá escapar.

El estado q_2 es un estado trampa. Vemos claramente en el grafo que el autómata acepta todas las cadenas que constan de un número arbitrario de as, seguidas de una sola b. Todas las demás cadenas de entrada se rechazan. En notación de conjuntos, el lenguaje aceptado por el autómata es

$$L = \{a^n b : n \geq 0\}.$$

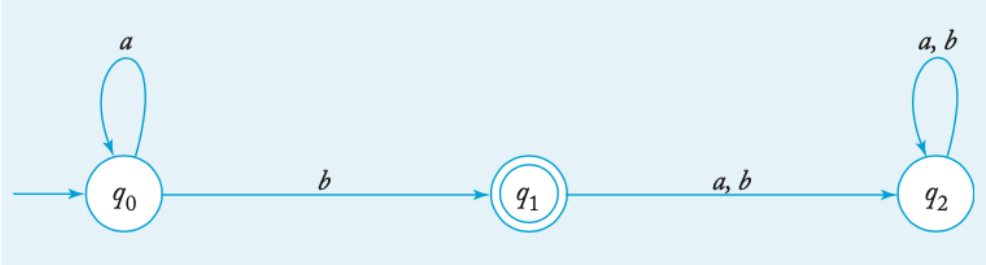


Figura 2.2: Grafo que representa al autómata del Ejemplo 2.2.

□

Estos ejemplos muestran lo convenientes que son los grafos de transición para trabajar con autómatas finitos. Si bien es posible basar todos los argumentos estrictamente en las propiedades de la función de transición y su

extensión a través de (2.3) y (2.4), los resultados son difíciles de seguir. En nuestra discusión, usamos grafos, que son más intuitivos, en la medida de lo posible.

Para hacerlo, debemos, por supuesto, tener cierta seguridad de que la representación no nos engañará y de que los argumentos basados en grafos son tan válidos como los que utilizan las propiedades formales de δ. El siguiente resultado preliminar nos da esta seguridad.

Teorema 2.1 Sea $M = (Q, \Sigma, \delta, q_0, F)$ un aceptador finito determinista, y sea G_M su grafo de transición asociado. Entonces, para cada $q_i, q_j \in Q$ y $w \in \Sigma^+, \delta^*(q_i, w) = q_j$ si y sólo si en G_M hay una camino con etiqueta w de q_i a q_j.

Prueba: Esta afirmación es bastante obvia al examinar casos tan simples como el Ejemplo 2.1. Puede demostrarse rigurosamente utilizando inducción sobre la longitud de w. Suponga que la afirmación es verdadera para todas las cadenas v con $|v| \leq n$. Considere entonces cualquier w de longitud $n+1$ y escríbala como

$$w = va.$$

Supongamos ahora que $\delta^*(q_i, v) = q_k$. Dado que $|v| = n$, debe haber un camino en G_M etiquetado con v de q_i a q_k. Pero si $\delta^*(q_i, w) = q_j$, entonces M debe tener una transición $\delta(q_k, a) = q_j$, de modo que por construcción G_M tiene una arista (q_k, q_j) con etiqueta a. Por lo tanto, hay un camino en G_M etiquetado con $va = w$ entre q_i y q_j. Dado que el resultado es obviamente cierto para $n = 1$, podemos afirmar por inducción que, para todo $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{2.5}$$

implica que hay un camino en G_M de q_i a q_j etiquetado con w. El argumento se puede invertir de una manera sencilla para mostrar que la existencia de tal camino implica (2.5), completando así la prueba. ■

Nuevamente, el resultado del teorema es tan intuitivamente obvio que una demostración formal parece innecesaria. Revisamos los detalles por dos razones.

La primera es que es un ejemplo simple pero típico de una prueba inductiva en relación con los autómatas.

La segunda es que el resultado se usará una y otra vez, por lo que afirmarlo y demostrarlo como un teorema nos permite argumentar con bastante

confianza usando grafos. Esto hace que nuestros ejemplos y pruebas sean más transparentes de lo que serían si usáramos las propiedades de δ^*.

Si bien los grafos son convenientes para visualizar autómatas, también son útiles otras representaciones. Por ejemplo, podemos representar la función δ como una tabla. La tabla 2.1 es equivalente al grafo de la Figura 2.2. Aquí, la etiqueta de la fila es el estado actual, mientras que la etiqueta de la columna representa el símbolo de entrada actual. La entrada en la tabla define el siguiente estado.

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

Tabla 2.1: Tabla equivalente al grafo de la Figura 2.2.

Es evidente a partir de este ejemplo que un dfa puede implementarse fácilmente como un programa de computadora; por ejemplo, como una simple búsqueda en una tabla o como una secuencia de instrucciones if.

La mejor implementación o representación depende de la aplicación específica. Los grafos de transición son muy convenientes para los tipos de argumentos que queremos hacer aquí, por lo que los usamos en la mayoría de nuestras discusiones.

Ejemplo 2.3 Encuentre un aceptador finito determinista que reconozca el conjunto de todas las cadenas en $\Sigma = \{a, b\}$ comenzando con el prefijo ab.

El único problema aquí son los dos primeros símbolos de la cadena; una vez leídas, no se necesitan más decisiones. Aún así, el autómata tiene que procesar toda la cadena antes de tomar su decisión. Por lo tanto, podemos

resolver el problema con un autómata que tiene cuatro estados: un estado inicial, dos estados para reconocer ab que termina en un estado de trampa final y un estado de trampa no final. Si el primer símbolo es una a y el segundo es una b, el autómata pasa al estado de trampa final, donde permanecerá ya que el resto de la entrada no importa. Por otro lado, si el primer símbolo no es una a o el segundo no es una b, el autómata entra en el estado de trampa no final. La solución simple se muestra en la Figura 2.3.

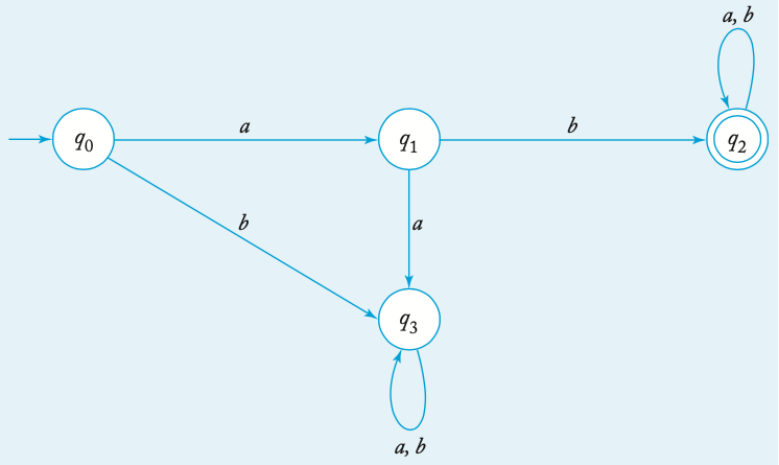


Figura 2.3: Grafo que representa al autómata del Ejemplo 2.3.

□

Ejemplo 2.4 Busque un dfa que acepte todas las cadenas en 0,1, excepto las que contengan la subcadena 001.

Al decidir si se ha producido la subcadena 001, necesitamos saber no sólo el símbolo de entrada actual, sino también recordar si ha sido o no precedido por uno o dos ceros. Podemos realizar un seguimiento de esto poniendo el autómata en estados específicos y etiquetándolos en consecuencia.

Al igual que los nombres de variables en un lenguaje de programación, los nombres de estado son arbitrarios y se pueden elegir por razones mnemotécnicas. Por ejemplo, el estado en el que dos 0 eran los símbolos inmediatamente anteriores se puede etiquetar simplemente como 00.

Si la cadena comienza con 001, debe rechazarse. Esto implica que debe haber una ruta etiquetada como 001 desde el estado inicial a un estado no final. Por conveniencia, este estado no final está etiquetado como 001. Este

estado debe ser un estado de trampa, porque los símbolos posteriores no importan. Todos los demás estados son estados de aceptación.

Esto nos da la estructura básica de la solución, pero aún debemos agregar provisiones para la subcadena 001 que ocurre en el medio de la entrada. Debemos definir Q y δ de modo que el autómata recuerde todo lo que necesitemos para tomar la decisión correcta.

En este caso, cuando se lee un símbolo, necesitamos saber alguna parte de la cadena a la izquierda, por ejemplo, si los dos símbolos anteriores eran 00 o no. Si etiquetamos los estados con los símbolos relevantes, es muy fácil para ver cuáles deben ser las transiciones. Por ejemplo,

$$\delta(00, 0) = 00$$

porque esta situación surge sólo si hay tres ceros consecutivos. Sólo nos interesan los dos últimos, un hecho que recordamos al mantener el dfa en el estado 00. En la Figura 2.4 se muestra una solución completa.

Vemos en este ejemplo lo útiles que son las etiquetas mnemotécnicas en los estados para realizar un seguimiento de las cosas. Rastree algunas cadenas, como 100100 y 1010100, para ver que la solución sea correcta.

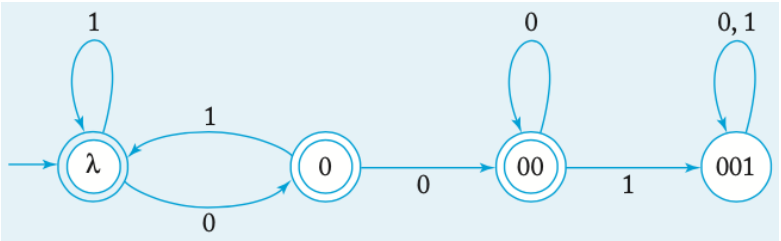


Figura 2.4: Grafo que representa al autómata del Ejemplo 2.4.

□

2.1.2. Lenguajes regulares

Todo autómata finito acepta algún lenguaje. Si consideramos todos los posibles autómatas finitos, obtenemos un conjunto de lenguajes asociados con ellos. Llamaremos familia a ese conjunto de lenguajes. La familia de lenguajes que aceptan los aceptadores finitos deterministas es bastante limitada.

La estructura y propiedades de los lenguajes de esta familia se aclararán a medida que avance nuestro estudio; por el momento simplemente asignaremos un nombre a esta familia.

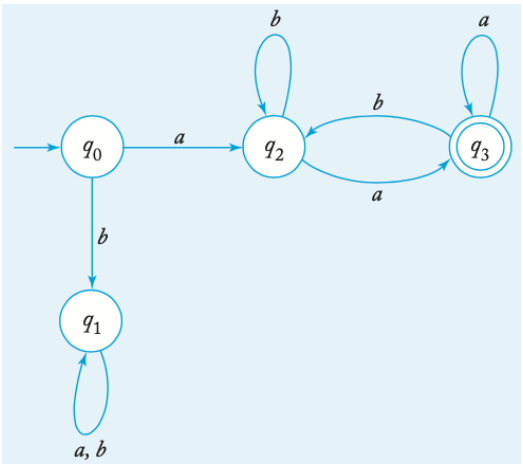


Figura 2.5: Grafo que representa al autómata del Ejemplo 2.5.

Definición 2.3 Un lenguaje L se llama *regular* si y sólo si existe algún aceptador finito determinista M tal que

$$L = L(M).$$

Ejemplo 2.5 Muestre que el lenguaje

$$L = \{awa; w \in \{a, b\}^*\}$$

es regular.

Para demostrar que este o cualquier otro lenguaje es regular, todo lo que tenemos que hacer es encontrar un dfa para ello. La construcción de un dfa para este lenguaje es similar al Ejemplo 2.3, pero un poco más complicada. Lo que debe hacer este dfa es verificar si una cadena comienza y termina con una a; lo que está en medio es inmaterial.

La solución se complica por el hecho de que no hay una forma explícita de probar el final de la cadena. Esta dificultad se supera simplemente poniendo el dfa en un estado final siempre que se encuentre la segunda a. Si este no es el final de la cadena y se encuentra otra b, el dfa saldrá del estado final. El escaneo continúa de esta manera, cada uno llevando al autómata a su estado final. La solución completa se muestra en la Figura 2.5.

Nuevamente, rastree algunos ejemplos para ver por qué esto funciona. Después de una o dos pruebas, será obvio que la dfa acepta una cadena si y sólo si comienza y termina con una a. Dado que hemos construido un dfa para el lenguaje, podemos afirmar que, por definición, el lenguaje es regular. □

Ejemplo 2.6 Sea L el lenguaje del Ejemplo 2.5, muestre que L^2 es regular.

Nuevamente mostramos que el lenguaje es regular construyendo un dfa para él. Podemos escribir una expresión explícita para L_2, a saber,

$$L^2 = \{aw_1aaw_2a : w_1, w_2 \in \{a, b\}^*\}.$$

Por lo tanto, necesitamos un dfa que reconozca dos cadenas consecutivas de esencialmente la misma forma (pero no necesariamente idénticas en valor). El diagrama de la Figura 2.5 se puede utilizar como punto de partida, pero el vértice q_3 debe modificarse. Este estado ya no puede ser final ya que, en este punto, debemos empezar a buscar una segunda subcadena de la forma awa.

Para reconocer la segunda subcadena, replicamos los estados de la primera parte (con nuevos nombres), con q_3 como comienzo de la segunda parte. Dado que la cadena completa se puede dividir en sus partes constituyentes dondequiera que ocurra aa, permitimos que la primera aparición de dos a consecutivas sea el disparador que lleva al autómata a su segunda parte. Podemos hacer esto haciendo $\delta(q_3, a) = q_4$. La solución completa está en la Figura 2.6. Este dfa acepta L^2, que por lo tanto es regular.

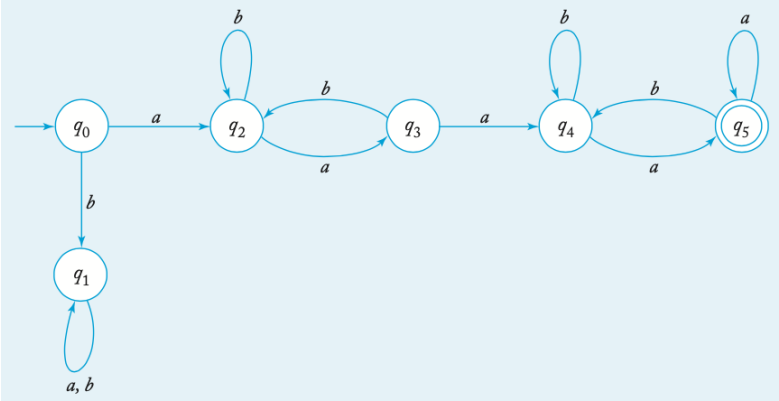


Figura 2.6: Grafo que representa al autómata del Ejemplo 2.6.

□

El último ejemplo sugiere la conjetura de que si un lenguaje L es regular, también lo son $L^2, L^3, \ldots$. Veremos más adelante que esto es correcto.

2.1.3. Ejercicios

1. ¿Cuáles de las cadenas 0001, 01101, 00001101 son aceptadas por el dfa en la Figura 2.1?

2. Traduzca la gráfica de la Figura 2.4 a notación δ.

3. Para $\Sigma = \{a, b\}$, construya dfa's que acepten los conjuntos que consisten de

   a) todas las cadenas con un número par de símbolos a.

   b) todas las cadenas con un número par de símbolos a y un número impar de símbolos b.

   c) todas las cadenas con al menos un símbolo b y exactamente dos símbolos a.

   d) todas las cadenas con exactamente dos símbolos a y más de tres símbolos b.

4. Construya dfas para los siguientes lenguajes sobre $\Sigma = \{a, b\}$.

   a) $L = \{ab^4wb^2 : w \in \{a, b\}^*\}$.

   b) $L = \{ab^n a^m : n \geq 3, m \geq 2\}$.

   c) $L = \{w : |w| \mod 3 \neq 0\}$.

   d) $L = \{w : |w| \mod 5 = 0\}$.

5. Demuestre que si cambiamos la Figura 2.5, haciendo q_3 un estado no final y haciendo q_0, q_1, q_2 estados finales, el dfa resultante acepta $\overline{L}$.

6. Generalice la observación en el ejercicio previo. Específicamente, demuestre que si $M = (Q, \Sigma, \delta, q_0, F)$ y $\widehat{M} = (Q, \Sigma, \delta, q_0, Q - F)$ son dos dfas, entonces $\overline{L(M)} = L(\widehat{M})$.

7. Use las ecuaciones (2.3) y (2.4) para demostrar que

   $$\delta^*(q, wv) = \delta^*(\delta^*(q, w), v)$$

   para toda $w, v \in \Sigma^*$.

2.2. Aceptadores finitos no deterministas

Si examina los autómatas que hemos visto hasta ahora, notará una característica común: se define una transición única para cada estado y cada símbolo de entrada. En la definición formal, esto se expresa diciendo que δ es una función total. Esta es la razón por la que llamamos deterministas a estos autómatas.

Ahora complicamos las cosas dando algunas opciones de autómatas en algunas situaciones en las que es posible más de una transición. A estos autómatas los llamaremos no deterministas.

El no determinismo es, a primera vista, una idea inusual. Las computadoras son máquinas deterministas y el elemento de elección parece fuera de lugar. No obstante, el no determinismo es un concepto útil, como veremos.

2.2.1. Definición de un aceptador no determinista

El no determinismo significa una elección de movimientos para un autómata. En lugar de prescribir un movimiento único en cada situación, permitimos un conjunto de movimientos posibles. Formalmente, logramos esto definiendo la función de transición para que su rango sea un conjunto de estados posibles.

Definición 2.4 Un **aceptador finito no determinista** o **nfa** se define por el quintuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

donde Q, Σ, q_0, F se definen como para los aceptadores finitos deterministas, pero

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Tenga en cuenta que existen tres diferencias principales entre esta definición y la definición de un dfa. En un aceptador no determinista, el rango de δ es el conjunto potencia 2^Q, por lo que su valor no es un solo elemento de Q, sino un subconjunto de él.

Este subconjunto define el conjunto de todos los estados posibles que puede alcanzar la transición. Si, por ejemplo, el estado actual es q_1, se lee el símbolo a, y

$$\delta(q_1, a) = \{q_0, q_2\},$$

entonces q_0 o q_2 podrían ser el siguiente estado del nfa. Además, permitimos λ como segundo argumento de δ. Esto significa que el nfa puede hacer una transición sin consumir un símbolo de entrada.

Aunque todavía asumimos que el mecanismo de entrada solo puede viajar hacia la derecha, es posible que esté estacionario en algunos movimientos. Finalmente, en un nfa, el conjunto $\delta(q_i, a)$ puede estar vacío, lo que significa que no hay una transición definida para esta situación específica.

Al igual que los dfa, los aceptadores no deterministas se pueden representar mediante grafos de transición. Los vértices están determinados por Q, mientras que una arista (q_i, q_j) con la etiqueta a está en el grafo si y sólo si $q_j \in \delta(q_i, a)$. Tenga en cuenta que dado que a puede ser la cadena vacía, puede haber algunas aristas etiquetadas con λ.

Una cadena es aceptada por un nfa si hay alguna secuencia de posibles movimientos que pondrán la máquina en un estado final cuando se haya leído toda la cadena. Una cadena se rechaza (es decir, no se acepta) sólo si no hay una secuencia posible de movimientos mediante la cual se pueda alcanzar un estado final.

Por lo tanto, se puede considerar que el no determinismo implica una percepción "intuitiva" mediante la cual se puede elegir el mejor movimiento en cada estado (asumiendo que el nfa quiere aceptar cada cadena).

Ejemplo 2.7 Considere el grafo de transición de la Figura 2.7. Describe un aceptador no determinista ya que hay dos transiciones etiquetadas como a que salen de q_0. □

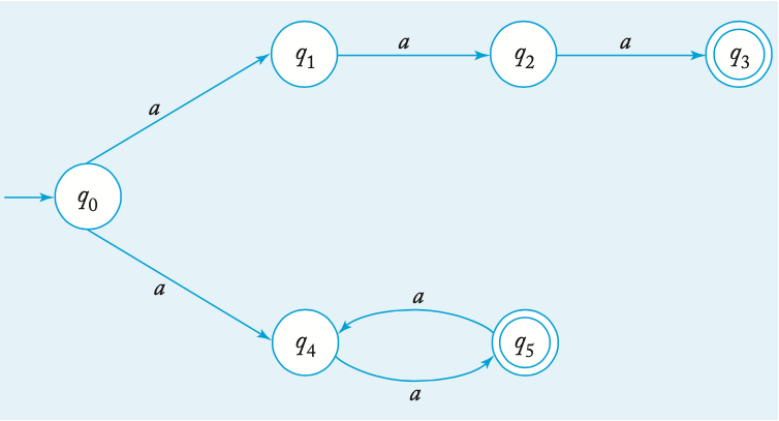


Figura 2.7: Grafo que representa al autómata del Ejemplo 2.7.

Ejemplo 2.8 En la Figura 2.8 se muestra un autómata no determinista. No es determinista no sólo porque varias aristas con la misma etiqueta se originan en un vértice, sino también porque tiene una transición λ. Algunas transiciones, como $\delta(q_2, 0)$, no están especificadas en el grafo. Esto debe interpretarse como una transición al conjunto vacío, es decir, $\delta(q_2, 0) = \emptyset$.

El autómata acepta las cadenas $\lambda, 1010$ y 101010, pero no 110 ni 10100. Tenga en cuenta que para 10 hay dos caminos alternativos, uno que conduce a q_0 y el otro a q_2. Aunque q_2 no es un estado final, la cadena se acepta porque hay un camino que conduce a un estado final. □

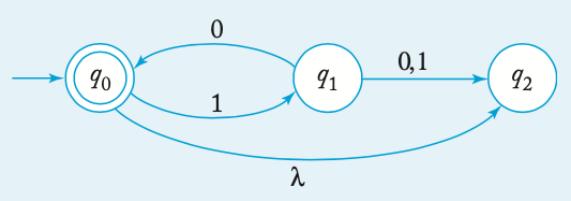


Figura 2.8: Grafo que representa al autómata del Ejemplo 2.8.

Nuevamente, la función de transición se puede extender para que su segundo argumento sea una cadena. Requerimos de la función de transición extendida δ^* que si

$$\delta^*(q_i, w) = Q_j,$$

entonces Q_j es el conjunto de todos los estados posibles en los que puede estar el autómata, habiendo comenzado en el estado q_i y habiendo leído w. Una definición recursiva de δ^*, análoga a (2.3) y (2.4), es posible, pero no particularmente esclarecedora. Se puede hacer una definición más fácil de apreciar a través de grafos de transición.

Definición 2.5 Para un nfa, la función de transición extendida se define de modo que $\delta^*(q_i, w)$ contenga q_j si y sólo si hay una camino en el grafo de transición de q_i a q_j etiquetado w. Esto es válido para todo $q_i, q_j \in Q$ y $w \in \Sigma^*$.

Ejemplo 2.9 La Figura 2.9 representa un nfa. Tiene varias transiciones λ y algunas transiciones indefinidas como $\delta(q_1, a)$ y $\delta(q_2, a)$. Suponga que queremos encontrar $\delta^*(q_1, a)$ y $\delta^*(q_2, \lambda)$. Hay un camino etiquetado a que involucra

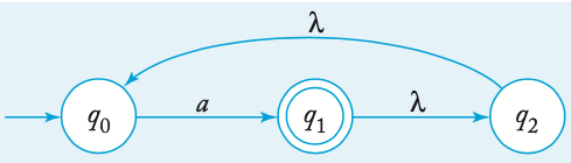


Figura 2.9: Grafo que representa al autómata del Ejemplo 2.9.

dos transiciones λ de q_1 a sí mismo. Al usar algunos de las aristas λ dos veces, vemos que también hay caminos que involucran transiciones λ a q_0 y q_2. Así,

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Dado que hay un arista λ entre q_2 y q_0, tenemos inmediatamente que $\delta^*(q_2, \lambda)$ contiene a q_0. Además, dado que se puede llegar a cualquier estado desde sí mismo sin hacer ningún movimiento y, en consecuencia, sin utilizar ningún símbolo de entrada, $\delta^*(q_2, \lambda)$ también contiene q_2. Por lo tanto,

$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Usando tantas transiciones λ como sea necesario, también puede verificar que

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

□

Como vemos la Definición 2.5 dice lo que δ^* debe hacer, pero no dice cómo realizarlo. Es decir, tenemos que desarrollar una definición formal para δ^*, para esto debemos notar que cuando un nfa tiene transiciones λ debemos tomar en cuenta que la cadena $w = a_1 a_2 \cdots a_n$ la tenemos que procesar como $w = \lambda a_1 \lambda a_2 \lambda \cdots \lambda a_n \lambda$.

De la misma manera no podemos conformarnos con declaraciones como las del Ejemplo 2.9 donde se menciona "Usando tantas transiciones λ como sea necesario ...". Por ello tenemos que definir una forma precisa para saber cómo calcular las transiciones λ que un nfa puede hacer, es decir, qué estados son alcanzables mediante transiciones λ a partir de un conjunto de estados, esto es porque δ en nfas tiene como rango un conjunto de estados. Para lograr esto definimos la cerradura reflexiva y transitiva de transiciones λ, denotada por λ^* y llamada **cerradura** λ.

Definición 2.6 La *cerradura* λ de un conjunto de estados $\lambda^* : 2^Q \to 2^Q$ se define de la siguiente manera.

$$\lambda^*(\emptyset) = \emptyset, \tag{2.6}$$
$$\lambda^*(\{q\}) = \{q\} \cup \lambda^*(\delta(q, \lambda)), \tag{2.7}$$
$$\lambda^*(\{q_1, q_2, \ldots, q_n\}) = \lambda^*(\{q_1\}) \cup \lambda^*(\{q_2, \ldots, q_n\}), n \geq 2. \tag{2.8}$$

La Ecuación 2.6 nos dice que un conjunto vació de estados no puede alcanzar estado alguno, es decir, como resultado es $\emptyset$.

La ecuación 2.7 nos dice que un conjunto de estados con un solo elemento q puede llegar mediante transiciones λ a él mismo o a la cerradura λ del conjunto de estados que son alcanzables desde q por una transición λ.

La ecuación 2.8 expresa que un conjunto de estados de cardinalidad mayor que uno puede alcanzar los estados dados por la cerradura λ de cada uno de sus estados. Nótese que se calculan todos los estados alcanzables sin importar los estados por los que haya que pasar mediante transiciones λ, esto es gracias a la definición recursiva de λ^*.

Observe que si un nfa no contiene transiciones λ por (2.7) $\lambda^*(\{q\}) = \{q\}$, para todo $q \in Q$.

Ejemplo 2.10 Calcule λ^* para los estados q_0, q_1 y q_2 de la Figura 2.9.

Solución:

Para q_0 tenemos:

$$\begin{aligned} \lambda^*(\{q_0\}) &\overset{2,7}{=} \{q_0\} \cup \lambda^*(\delta(q_0, \lambda)) \\ &= \{q_0\} \cup \lambda^*(\emptyset) \\ &\overset{2,6}{=} \{q_0\} \cup \emptyset \\ &= \{q_0\}. \end{aligned} \tag{2.9}$$

Para q_1 tenemos:

$$\begin{aligned}\lambda^*(\{q_1\}) &\overset{2,7}{=} \{q_1\} \cup \lambda^*(\delta(q_1, \lambda)) \\ &= \{q_1\} \cup \lambda^*(\{q_2\}) \\ &\overset{2,7}{=} \{q_1\} \cup \{q_2\} \cup \lambda^*(\delta(q_2, \lambda)) \\ &= \{q_1\} \cup \{q_2\} \cup \lambda^*(\{q_0\}) \\ &\overset{2,9}{=} \{q_1\} \cup \{q_2\} \cup \{q_0\} \\ &= \{q_0, q_1, q_2\}.\end{aligned} \tag{2.10}$$

Para q_2 tenemos:

$$\begin{aligned}\lambda^*(\{q_2\}) &\overset{2,7}{=} \{q_2\} \cup \lambda^*(\delta(q_2, \lambda)) \\ &= \{q_2\} \cup \lambda^*(\{q_0\}) \\ &\overset{2,9}{=} \{q_2\} \cup \{q_0\} \\ &= \{q_0, q_2\}.\end{aligned} \tag{2.11}$$

□

Una vez definida λ^* podemos dar la definición de δ^*.

Definición 2.7 Sea $\delta^* : Q \times \Sigma^* \to 2^Q$ definida de la siguiente manera.

$$\delta^*(q, \lambda) = \lambda^*(\{q\}), \tag{2.12}$$

$$\delta^*(q, wa) = \lambda^*(\delta'(\delta^*(q, w), a)), \tag{2.13}$$

donde $\delta' : 2^Q \times \Sigma \to 2^Q$ es la extensión de δ para conjuntos de estados definida de la siguiente manera.

$$\delta'(R, a) = \bigcup_{q \in R} \delta(q, a). \tag{2.14}$$

Ejemplo 2.11 Calcule $\delta^*(q_2, aa)$ para el nfa de la Figura 2.9.

Solución:

$$
\begin{aligned}
\delta^*(q_2, aa) &\overset{2,13}{=} \lambda^*(\delta'(\delta^*(q_2, a), a)) \\
&\overset{2,13}{=} \lambda^*(\delta'(\lambda^*(\delta'(\delta^*(q_2, \lambda), a)), a)) \\
&\overset{2,12}{=} \lambda^*(\delta'(\lambda^*(\delta'(\lambda^*(\{q_2\}), a)), a)) \\
&\overset{2,11}{=} \lambda^*(\delta'(\lambda^*(\delta'(\{q_0, q_2\}, a)), a)) \\
&\overset{2,14}{=} \lambda^*(\delta'(\lambda^*(\delta(q_0, a) \cup \delta(q_2, a)), a)) \\
&= \lambda^*(\delta'(\lambda^*(\{q_1\} \cup \delta(q_2, a)), a)) \\
&= \lambda^*(\delta'(\lambda^*(\{q_1\} \cup \emptyset), a)) \\
&= \lambda^*(\delta'(\lambda^*(\{q_1\}), a)) \\
&\overset{2,10}{=} \lambda^*(\delta'(\{q_0, q_1, q_2\}, a)) \\
&\overset{2,14}{=} \lambda^*(\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a)) \\
&= \lambda^*(\{q_1\} \cup \delta(q_1, a) \cup \delta(q_2, a)) \\
&= \lambda^*(\{q_1\} \cup \emptyset \cup \delta(q_2, a)) \\
&= \lambda^*(\{q_1\} \cup \emptyset \cup \emptyset) \\
&= \lambda^*(\{q_1\}) \\
&\overset{2,10}{=} \{q_0, q_1, q_2\}.
\end{aligned}
$$

Ejemplo 2.12 Calcule $\delta^*(q_0, aa)$ para el nfa de la Figura 2.9.

Solución:

$$\begin{aligned}
\delta^*(q_0, aa) &\overset{2,13}{=} \lambda^*(\delta'(\delta^*(q_0, a), a)) \\
&\overset{2,13}{=} \lambda^*(\delta'(\lambda^*(\delta'(\delta^*(q_0, \lambda), a)), a)) \\
&\overset{2,12}{=} \lambda^*(\delta'(\lambda^*(\delta'(\lambda^*(\{q_0\}), a)), a)) \\
&\overset{2,11}{=} \lambda^*(\delta'(\lambda^*(\delta'(\{q_0\}, a)), a)) \\
&\overset{2,14}{=} \lambda^*(\delta'(\lambda^*(\delta(q_0, a)), a)) \\
&= \lambda^*(\delta'(\lambda^*(\{q_1\}), a)) \\
&\overset{2,10}{=} \lambda^*(\delta'(\{q_0, q_1, q_2\}, a)) \\
&\overset{2,14}{=} \lambda^*(\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a)) \\
&= \lambda^*(\{q_1\} \cup \delta(q_1, a) \cup \delta(q_2, a)) \\
&= \lambda^*(\{q_1\} \cup \emptyset \cup \delta(q_2, a)) \\
&= \lambda^*(\{q_1\} \cup \emptyset \cup \emptyset) \\
&= \lambda^*(\{q_1\}) \\
&\overset{2,10}{=} \{q_0, q_1, q_2\}.
\end{aligned}$$

□

Definición 2.8 El lenguaje L aceptado por un nfa $M = (Q, \Sigma, \delta, q_0, F)$ se define formalmente de la siguiente manera

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

En palabras, el lenguaje consta de todas las cadenas w para las que hay un camino etiquetado w desde el vértice inicial del grafo asociado con el nfa hasta algún vértice final.

Ejemplo 2.13 ¿Cuál es el lenguaje aceptado por el autómata en la Figura 2.8? Es fácil ver en el grafo que la única forma en que el nfa puede detenerse en un estado final es si la entrada es una repetición de la cadena 10 o la cadena vacía. Por tanto, el autómata acepta el lenguaje $L = \{(10)^n : n \geq 0\}$.

¿Qué sucede cuando a este autómata se le presenta la cadena $w = 110$? Después de leer el prefijo 11, el autómata se encuentra en el estado q_2, con la

transición $\delta(q_2, 0)$ indefinida. Llamamos a esta situación una **configuración muerta**, y podemos visualizarla como el autómata simplemente deteniéndose sin más acción. Pero siempre debemos tener en cuenta que tales visualizaciones son imprecisas y conllevan algún peligro de mala interpretación. Lo que podemos decir precisamente es que

$$\delta^*(q_0, 110) = \emptyset.$$

Por lo tanto, no se puede alcanzar un estado final procesando $w = 110$, y por lo tanto la cadena no se acepta. □

2.2.2. ¿Por qué el no determinismo?

El no determinismo a veces es útil para resolver problemas fácilmente. Mire el nfa en la Figura 2.7. Está claro que hay que tomar una decisión. La primera alternativa conduce a la aceptación de la cadena a^3, mientras que la segunda acepta todas las cadenas con un número par de as.

El lenguaje aceptado por el nfa es $\{a^3\} \cup \{a^{2n} : n \geq 1\}$. Si bien es posible encontrar un dfa para este lenguaje, el no determinismo es bastante natural. El lenguaje es la unión de dos conjuntos bastante diferentes, y el no determinismo nos permite decidir desde el principio qué caso queremos.

La solución determinista no está tan obviamente relacionada con la definición, por lo que es un poco más difícil de encontrar. A medida que avancemos, veremos otros ejemplos más convincentes de la utilidad del no determinismo.

En la misma línea, el no determinismo es un mecanismo eficaz para describir algunos lenguajes complicados de forma concisa. Note que la definición de una gramática involucra un elemento no determinista. En

$$S \to aSb|\lambda$$

en cualquier momento podemos elegir entre la primera o la segunda producción. Esto nos permite especificar muchas cadenas diferentes usando sólo dos reglas.

Finalmente, existe una razón técnica para introducir el no determinismo. Como veremos, ciertos resultados teóricos se establecen más fácilmente para nfas que para dfas.

Nuestro siguiente resultado importante indica que no existe una diferencia esencial entre estos dos tipos de autómatas. En consecuencia, permitir el no determinismo a menudo simplifica los argumentos formales sin afectar la generalidad de la conclusión.

2.2.3. Ejercicios

1. Para el nfa de la Figura 2.8 calcule $\delta^*(q_0, 1011)$ y $\delta^*(q_1, 01)$.

2. Para el nfa de la Figura 2.8 calcule $\delta^*(q_0, 1010)$ y $\delta^*(q_1, 00)$.

3. Para el nfa de la Figura 2.9 calcule $\delta^*(q_0, a)$ y $\delta^*(q_1, \lambda)$.

4. Diseñe un nfa con no más de cinco estados para el conjunto $\{abab^n : n \geq 0\} \cup \{aba^n : n \geq 0\}$.

5. Construya un nfa con tres estados que acepte el lenguaje $\{ab, abc\}^*$.

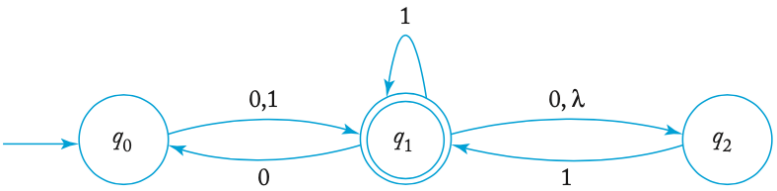
6. Construya un nfa con tres estados que acepte el lenguaje

$$L = \{a^n; n \geq 1\} \cup \{b^m a^k : m \geq 0, k \geq 0\}.$$

7. Construya un nfa con cuatro estados para

$$L = \{a^n : n \geq 0\} \cup \{b^n a : n \geq 1\}.$$

8. ¿Cuáles de las cadenas 00, 01001, 10010, 000, 0000 son aceptadas por el siguiente nfa?



9. Sea L el lenguaje aceptado por el nfa en la Figura 2.7. Construya un nfa que acepte $L \cup \{a^5\}$.

10. Construya un nfa que acepte $\{a\}^*$ y sea tal que si en su grafo de transición se elimina una sola arista (sin ningún otro cambio), el autómata resultante acepta $\{a\}$.

2.3. Equivalencia entre dfas y nfas

Llegamos ahora a una cuestión fundamental. ¿En qué sentido son diferentes los dfas y nfas? Evidentemente, existe una diferencia en su definición, pero esto no implica que exista una distinción esencial entre ellos. Para explorar esta pregunta, presentamos el concepto de equivalencia entre autómatas.

Definición 2.9 Se dice que dos aceptadores finitos, M_1 y M_2, son equivalentes si

$$L(M_1) = L(M_2),$$

es decir, si ambos aceptan el mismo lenguaje

Como se mencionó, generalmente hay muchos aceptadores para un lenguaje dado, por lo que cualquier dfa o nfa tiene muchos aceptadores equivalentes.

Ejemplo 2.14 El dfa que se muestra en la Figura 2.10 es equivalente al nfa en la Figura 2.8 ya que ambos aceptan el lenguaje $\{(10)^n : n \geq 0\}$. □

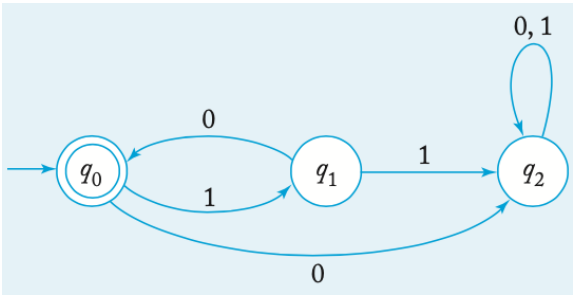


Figura 2.10: Autómata equivalente al autómata de la Figura 2.8.

Cuando comparamos diferentes clases de autómatas, invariablemente surge la pregunta de si una clase es más poderosa que la otra. Por "más poderoso" queremos decir que un autómata de un tipo puede lograr algo que ningún autómata del otro tipo puede hacer.

Examinemos esta pregunta para los aceptadores finitos. Dado que un dfa es en esencia un tipo restringido de nfa, está claro que cualquier lenguaje que sea aceptado por un dfa también lo es por algunos nfa. Pero lo contrario no es tan obvio.

Hemos añadido el no determinismo, por lo que es al menos concebible que haya un lenguaje aceptado por algún nfa para el que, en principio, no podemos encontrar un dfa. Pero resulta que no es así. Las clases de dfa y nfa son igualmente poderosas: para cada lenguaje aceptado por algún nfa, hay un dfa que acepta el mismo lenguaje.

Este resultado no es obvio y ciertamente debe demostrarse. El argumento, como la mayoría de los argumentos de este libro, será constructivo. Esto significa que en realidad podemos ofrecer una forma de convertir cualquier nfa en un dfa equivalente.

La construcción no es difícil de entender; una vez que la idea es clara, se convierte en el punto de partida para un argumento riguroso. El fundamento de la construcción es el siguiente. Después de que un nfa ha leído una cadena w, es posible que no sepamos exactamente en qué estado estará, pero podemos decir que debe estar en un estado de un conjunto de estados posibles, digamos $\{qi, qj, ..., qk\}$.

Un dfa equivalente después de leer la misma cadena debe estar en algún estado definido. ¿Cómo podemos hacer que estas dos situaciones se correspondan? La respuesta es un buen truco: etiquete los estados del dfa con un conjunto de estados de tal manera que, después de leer w, el dfa equivalente estará en un solo estado etiquetado $\{qi, qj, ..., qk\}$.

Dado que para un conjunto de $|Q|$ estados hay exactamente $2^{|Q|}$ subconjuntos, el correspondiente dfa tendrá un número finito de estados.

La mayor parte del trabajo en esta construcción sugerida radica en el análisis del nfa para obtener la correspondencia entre los posibles estados y las entradas. Antes de llegar a la descripción formal de esto, ilustremos con un ejemplo sencillo.

Ejemplo 2.15 Convierta el nfa de la Figura 2.11 en un dfa equivalente.

El nfa comienza en el estado q_0, por lo que el estado inicial del dfa se etiquetará como $\{q_0\}$. Después de leer una a, el nfa puede estar en el estado q_1 o, al hacer una transición λ, en el estado q_2. Por lo tanto, el dfa correspondiente debe tener un estado etiquetado $\{q_1, q_2\}$ y una transición

$$\delta(\{q_0\}, a) = \{q_1, q_2\}.$$

Ahora hemos introducido en el dfa el estado $\{q_1, q_2\}$, por lo que necesitamos encontrar las transiciones fuera de este estado. Recuerde que este estado

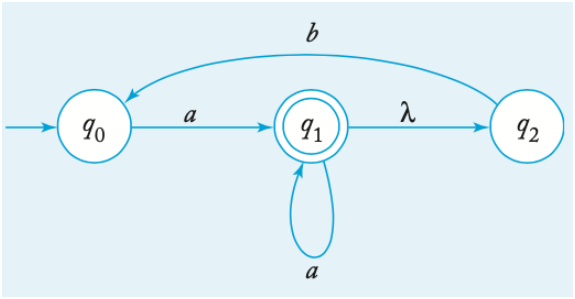


Figura 2.11: Autómata para el Ejemplo 2.15.

del dfa corresponde a dos posibles estados del nfa, por lo que debemos volver a referirnos al nfa.

Si el nfa está en el estado q_1 y lee una a, puede ir a q_1. Además, desde q_1 el nfa puede hacer una transición λ a q_2. Si, para la misma entrada, el nfa está en el estado q_2, entonces no hay una transición especificada. Por lo tanto,

$$\delta(\{q_1, q2\}, a) = \{q_1, q_2\}.$$

Similarmente,

$$\delta(\{q_1, q2\}, b) = \{q_0\}.$$

En este punto, cada estado tiene todas las transiciones definidas. El resultado, que se muestra en la Figura 2.12, es un dfa, equivalente al nfa con el que comenzamos. El nfa de la Figura 2.11 acepta cualquier cadena para la que $\delta^*(q_0, w)$ contenga q_1. Para que el dfa correspondiente acepte cada w, cualquier estado cuya etiqueta incluya q_1 debe convertirse en un estado final. □

Teorema 2.2 Sea L el lenguaje aceptado por un aceptador finito no determinista $M_N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Entonces existe un aceptador finito determinista $M_D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ tal que

$$L = L(M_D).$$

Demostración: Dado M_N, usamos el procedimiento nfa-to-dfa, mostrado a continuación, para construir el grafo de transición G_D para M_D. Para comprender la construcción, recuerde que G_D debe tener ciertas propiedades. Cada vértice debe tener exactamente $|\Sigma|$ aristas salientes, cada una etiquetada con un elemento diferente de Σ. Durante la construcción, es posible que

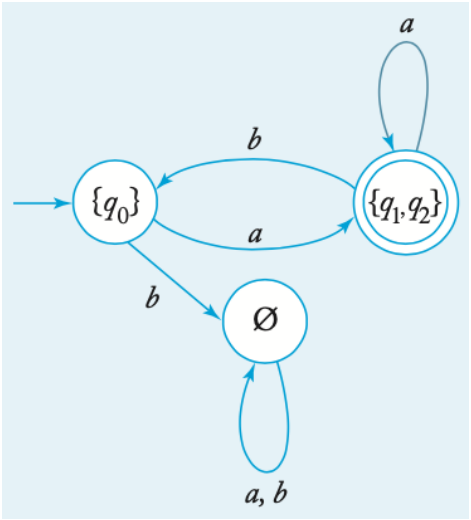


Figura 2.12: dfa equivalente al nfa de la Figura 2.11.

falten algunas de las aristas, pero el procedimiento continúa hasta que estén todas allí.

procedimiento nfa-a-dfa

1. Cree un grafo G_D con vértice $\{q_0\}$. Identifique este vértice como el vértice inicial.

2. Repita los siguientes pasos hasta que no falten más aristas.

   a) Tome cualquier vértice $\{q_i, q_j, \ldots, q_k\}$ de G_D que no tenga aristas salientes para $a \in \Sigma$. Calcule $\delta_N^*(q_i, a), \delta_N^*(q_j, a), \ldots, \delta_N^*(q_k, a)$. Si

   $$\delta_N^*(q_i, a) \cup \delta_N^*(q_j, a) \cup \cdots \cup \delta_N^*(q_k, a) = \{q_l, q_m, \ldots, q_n\},$$

   cree un vértice para G_D etiquetado $\{q_l, q_m, \ldots, q_n\}$ si aún no existe.

   b) Agregue a G_D una arista de $\{q_i, q_j, \ldots, q_k\}$ a $\{q_l, q_m, \ldots, q_n\}$ y etiquételo con a.

3. Cada estado de G_D cuya etiqueta contiene cualquier $q_f \in F_N$ se identifica como un vértice final.

4. Si M_N acepta λ, el vértice $\{q_0\}$ en G_D también se convierte en un vértice final.

Está claro que este procedimiento siempre termina. Cada paso a través del ciclo en el paso 2 agrega una arista a G_D. Pero G_D tiene como máximo $2^{|Q_N|}|\Sigma|$ aristas, de modo que el bucle finalmente se detiene. Para mostrar que la construcción también da la respuesta correcta, argumentamos por inducción sobre la longitud de la cadena de entrada.

Suponga que para cada v de longitud menor o igual a n, la presencia en G_N de un camino etiquetado v de q_0 a q_i implica que en G_D hay un camino etiquetado v de $\{q_0\}$ a un estado $Q_i = \{\ldots, q_i, \ldots\}$.

Considere ahora cualquier $w = va$ y observe un camino en G_N etiquetado como w de q_0 a q_l. Entonces debe haber un camino etiquetado v de q_0 a q_i y una arista (o una secuencia de aristas) etiquetada a de q_i a q_l.

Según el supuesto inductivo, en G_D habrá un camino etiquetado v desde $\{q_0\}$ a Q_i. Pero por construcción, habrá una arista de Q_i a algún estado cuya etiqueta contenga q_l. Por lo tanto, la suposición inductiva se cumple para todas las cadenas de longitud $n + 1$. Como es obviamente cierto para $n = 1$, lo es para todos n.

El resultado entonces es que siempre que $\delta_N^*(q_0, w)$ contiene un estado final q_f, también lo hace la etiqueta de $\delta_D^*(q_0, w)$. Para completar la demostración, invertimos el argumento para mostrar que si la etiqueta de $\delta_D^*(q_0, w)$ contiene q_f, entonces $\delta_N^*(q_0, w)$ también la debe contener. ■

Los argumentos en esta prueba, aunque correctos, son ciertamente algo concisos, y muestran sólo los pasos principales. Seguiremos esta práctica en el resto del documento, enfatizando las ideas básicas en una prueba y omitiendo detalles menores, que tal vez desee completar usted mismo.

La construcción de la prueba anterior es tediosa pero importante. Hagamos otro ejemplo para asegurarnos de que entendemos todos los pasos.

Ejemplo 2.16 Convierta el nfa de la Figura 2.13 en una máquina determinista equivalente.

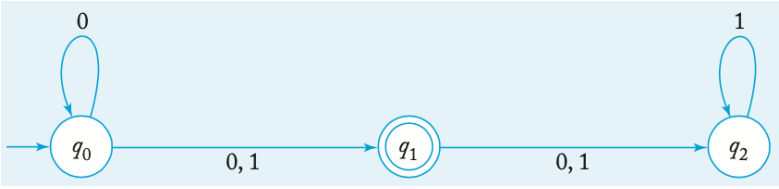


Figura 2.13: nfa para el Ejemplo 2.16.

Como $\delta_N(q_0, 0) = \{q_0, q_1\}$, introducimos el estado $\{q_0, q_1\}$ en G_D y agregamos una arista etiquetada como 0 entre $\{q_0\}$ y $\{q_0, q_1\}$. De la misma manera, considerando $\delta_N(q_0, 1) = \{q_1\}$ nos da el nuevo estado $\{q_1\}$ y una arista etiquetada 1 entre él y $\{q_0\}$.

Ahora hay varias aristas faltantes, por lo que continuamos usando la construcción del Teorema 2.2. Al observar el estado $\{q_0, q_1\}$, vemos que no hay una arista saliente etiquetada como 0, por lo que calculamos

$$\delta_N^*(q_0, 0) \cup \delta_N^*(q_1, 0) = \{q_0, q_1, q_2\}.$$

Esto nos da el nuevo estado $\{q_0, q_1, q_2\}$ y la transición $\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_1, q_2\}$. Entonces, usando $a = 1, i = 0, j = 1, k = 2$,

$$\delta_N^*(q_0, 1) \cup \delta_N^*(q_1, 1) \cup \delta_N^*(q_2, 1) = \{q_1, q_2\}$$

es necesario introducir otro estado más $\{q_1, q_2\}$. En este punto, tenemos el autómata parcialmente construido que se muestra en la Figura 2.14. Dado que todavía faltan algunas aristas, continuamos hasta obtener la solución completa en la Figura 2.15.

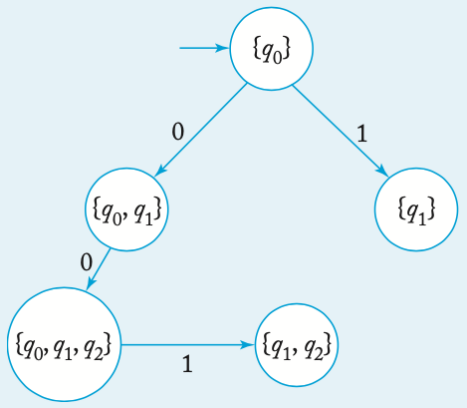


Figura 2.14: dfa parcial equivalente al nfa de la Figura 2.13.

□

Una conclusión importante que podemos sacar del Teorema 2.2 es que todos los lenguajes aceptados por un nfa son regulares.

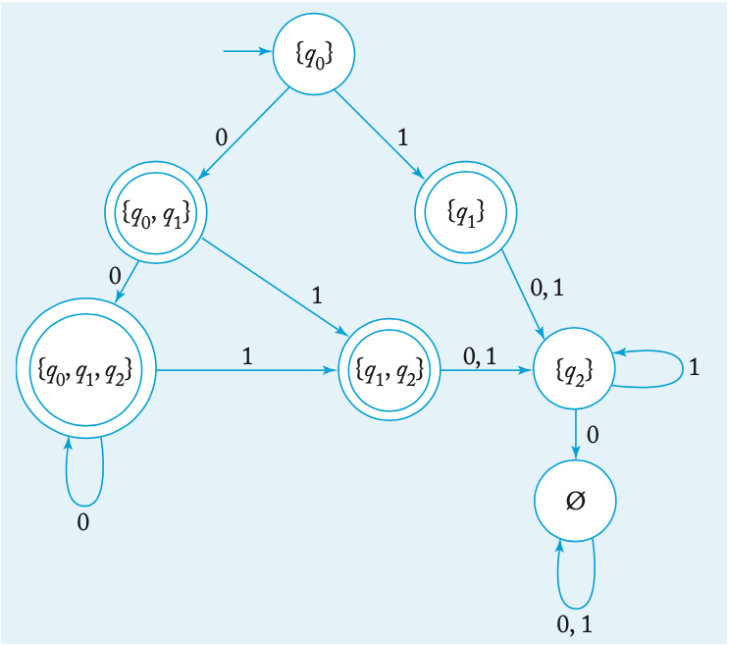


Figura 2.15: dfa completo equivalente al nfa de la Figura 2.13.

2.3.1. Ejercicios

1. Convierta los siguientes nfas a sus equivalentes dfas, donde en cada caso q_0 es el estado inicial y q_2 el estado final.

   a)

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

   b)

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

c)

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

2. ¿Es cierto que para cada nfa $M = (Q, \Sigma, \delta, q_0, F)$, el complemento de $L(M)$ es igual al conjunto $\{w \in \Sigma^* : \delta^*(q_0, w) \cap (Q - F) \neq \emptyset\}$? Si es así, pruébelo; en caso contrario, dé un contraejemplo.

3. Demuestre que todos los lenguajes finitos son regulares.

4. Demuestre que si L es regular, también lo es L^R.

Capítulo 3

Lenguajes y gramáticas regulares

Según nuestra definición, un lenguaje es regular si existe un aceptador finito que lo reconoce. Por lo tanto, cada lenguaje regular puede ser descrito por algún dfa o algún nfa. Esta descripción puede resultar muy útil, por ejemplo, si queremos mostrar la lógica mediante la cual decidimos si una cadena determinada está en un lenguaje determinado.

Pero en muchos casos, necesitamos formas más concisas de describir los lenguajes regulares. En este capítulo, analizamos otras formas de representar lenguajes regulares. Estas representaciones tienen importantes aplicaciones prácticas, tema que se aborda en algunos de los ejemplos y ejercicios.

3.1. Expresiones regulares

Una forma de describir los lenguajes regulares es mediante la notación de expresiones regulares. Esta notación implica una combinación de cadenas de símbolos de algún alfabeto Σ, paréntesis y los operadores $+, \cdot$ y $*$.

El caso más simple es el lenguaje $\{a\}$, que será denotado por la expresión regular a. Un poco más complicado es el lenguaje $\{a, b, c\}$, para el cual, usando $+$ para denotar unión, tenemos la expresión regular $a+b+c$. Usamos $\cdot$ para la concatenación y $*$ para la cerradura estrella. La expresión $(a + (bc))^*$ representa la cerradura estrella de $\{a\} \cup \{bc\}$, es decir, el lenguaje $\{\lambda, a, bc, aa, abc, bca, bcbc, aaa, aabc, \ldots\}$.

3.1.1. Definición formal de expresiones regulares

Construimos expresiones regulares a partir de constituyentes primitivos aplicando repetidamente ciertas reglas recursivas. Esto es similar a la forma en que construimos expresiones aritméticas familiares.

Definición 3.1 Sea Σ un alfabeto. Entonces

1. $\emptyset, \lambda$ y a son expresiones regulares. Se les llama **expresiones regulares primitivas**.

2. Si r_1 y r_2 son expresiones regulares, también lo son $r_1 + r_2, r_1 \cdot r_2, r_1^*$ y (r_1).

3. Una cadena es una expresión regular si y sólo si puede derivarse de las expresiones regulares primitivas mediante un número finito de aplicaciones de las reglas en 2.

Ejemplo 3.1 Para $\Sigma = \{a, b, c\}$, la cadena

$$(a + b \cdot c)^* \cdot (c + \emptyset)$$

es una expresión regular, ya que se construye mediante la aplicación de las reglas anteriores.

Por ejemplo, si tomamos $r_1 = c$ y $r_2 = \emptyset$, encontramos que $c + \emptyset$ y $(c + \emptyset)$ también son expresiones regulares. Repitiendo esto, eventualmente generamos la cadena completa.

Por otro lado, $(a + b+)$ no es una expresión regular, ya que no hay forma de que se pueda construir a partir de las expresiones regulares primitivas.

□

3.1.2. Lenguajes asociados con expresiones regulares

Se pueden usar expresiones regulares para describir algunos lenguajes simples. Si r es una expresión regular, entonces $L(r)$ denota el lenguaje asociado con r.

Definición 3.2 El lenguaje $L(r)$ denotado por cualquier expresión regular r está definido por las siguientes reglas, donde r_1 y r_2 son expresiones regulares.

1. $\emptyset$ es una expresión regular que denota el conjunto vacío,
2. λ es una expresión regular que denota $\{\lambda\}$,
3. Para cada $a \in \Sigma$, a es una expresión regular que denota $\{a\}$.
4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

Las últimas cuatro reglas de esta definición se utilizan para reducir recursivamente $L(r)$ a componentes más simples; las tres primeras son las condiciones de terminación de esta recursividad. Para ver qué lenguaje denota una expresión dada, aplicamos estas reglas repetidamente.

Ejemplo 3.2 Exhiba el lenguaje $L(a^* \cdot (a + b))$ en notación de conjuntos.

$$\begin{aligned} L(a^* \cdot (a+b)) =& L(a^*)L(a+b) \\ =& (L(a))^*(L(a) \cup L(b)) \\ =& \{\lambda, a, aa, aaa, \ldots\}\{a, b\} \\ =& \{a, aa, aaa, \ldots, b, ab, aab, aaab, \ldots\}. \end{aligned}$$

□

Hay un problema con las reglas 4 a 7 en la Definición 3.2. Definen un lenguaje precisamente si se dan r_1 y r_2, pero puede haber alguna ambigüedad al dividir una expresión complicada en partes. Considere, por ejemplo, la expresión regular $a \cdot b + c$. Podemos considerar que está formado por $r_1 = a \cdot b$ y $r_2 = c$. En este caso, encontramos $L(a \cdot b + c) = \{ab, c\}$.

Pero no hay nada en la Definición 3.2 que nos impida tomar $r_1 = a$ y $r_2 = b + c$. Ahora obtenemos un resultado diferente, $L(a \cdot b + c) = \{ab, ac\}$. Para superar esto, podríamos requerir que todas las expresiones estén entre paréntesis, pero esto da resultados engorrosos.

En su lugar, usamos una convención familiar de las matemáticas y los lenguajes de programación. Establecemos un conjunto de reglas de precedencia para la evaluación en las que la cerradura estrella precede a la concatenación y la concatenación precede a la unión.

Además, el símbolo para la concatenación puede omitirse, por lo que podemos escribir $r_1 r_2$ en lugar de $r_1 \cdot r_2$. Con un poco de práctica, podemos ver rápidamente qué lenguaje denota una expresión regular en particular.

Ejemplo 3.3 Para $\Sigma = \{a, b\}$, la expresión

$$r = (a + b)^*(a + bb)$$

es regular. Denota el lenguaje

$$L(r) = \{a, bb, aa, abb, ba, bbb, \ldots\}.$$

Podemos ver esto considerando las diversas partes de r. La primera parte, $(a + b)^*$, representa cualquier cadena de as y bs. La segunda parte, $(a + bb)$ representa una a o una doble b. En consecuencia, $L(r)$ es el conjunto de todas las cadenas en $\{a, b\}$, terminadas por una a o una bb.

□

Ejemplo 3.4 La expresión

$$r = (aa)^*(bb)^*b$$

denota el conjunto de todas las cadenas con un número par de as seguida por un número impar de bs; esto es,

$$L(r) = \{a^{2n}b^{2m+1} : n \geq 0, m \geq 0\}.$$

□

Pasar de una descripción informal o notación de conjuntos a una expresión regular tiende a ser un poco más difícil.

Ejemplo 3.5 Para $\Sigma = \{0, 1\}$, dé una expresión regular r tal que

$$L(r) = \{w \in \Sigma^* : w \text{ tiene al menos un par de ceros consecutivos}\}.$$

Se puede llegar a una respuesta razonando algo como esto: cada cadena en $L(r)$ debe contener 00 en algún lugar, pero lo que viene antes y lo que sigue es

completamente arbitrario. Una cadena arbitraria en $\{0,1\}$ se puede denotar por $(0+1)^*$. Juntando estas observaciones, llegamos a la solución

$$r = (0+1)^*00(0+1)^*.$$

□

Ejemplo 3.6 Encuentre una expresión regular para el lenguaje

$$L = \{w \in \{0,1\}^* : w \text{ no tiene un par de ceros consecutivos}\}.$$

Aunque parece similar al ejemplo 3.5, la respuesta es más difícil de construir. Una observación útil es que siempre que aparece un 0, debe ir seguido inmediatamente por un 1. Dicha subcadena puede ir precedida y seguida de un número arbitrario de unos.

Esto sugiere que la respuesta implica la repetición de cadenas de la forma $1\cdots101\cdots1$, es decir, el lenguaje denotado por la expresión regular $(1^*011^*)^*$. Sin embargo, la respuesta sigue siendo incompleta, ya que no se tienen en cuenta las cadenas que terminan en 0 o que constan de 1s. Después de atender estos casos especiales llegamos a la respuesta

$$r = (1^*011^*)^*(0+\lambda) + 1^*(0+\lambda).$$

Si razonamos de manera ligeramente diferente, podríamos encontrar otra respuesta. Si vemos L como la repetición de las cadenas 1 y 01, la expresión más corta

$$r = (1+01)^*(0+\lambda)$$

podría ser propuesta. Aunque las dos expresiones se ven diferentes, ambas respuestas son correctas, ya que denotan el mismo lenguaje. Generalmente, hay un número ilimitado de expresiones regulares para cualquier lenguaje.

Tenga en cuenta que este lenguaje es el complemento del lenguaje del Ejemplo 3.5. Sin embargo, las expresiones regulares no son muy similares y no sugieren claramente la estrecha relación entre los dos lenguajes.

□

El último ejemplo introduce la noción de equivalencia de expresiones regulares. Decimos que dos expresiones regulares son equivalentes si denotan el mismo lenguaje.

3.1.3. Ejercicios

1. Encuentre una expresión regular para el conjunto

$$\{a^n b^m : n \geq 3, m \text{ es impar}\}.$$

2. Encuentre una expresión regular para el conjunto

$$\{a^n b^m : (n+m) \text{ es impar}\}.$$

3. Dé expresiones regulares para los siguientes lenguajes.

   a) $L_1 = \{a^n b^m : n \geq 3, m \leq 4\}$.

   b) $L_2 = \{a^n b^m : n < 4, m \leq 4\}$.

   c) El complemento de L_1.

   d) El complemento de L_2.

4. Encuentre una expresión regular para

$$L = \{ab^n w : n \geq 4, w \in \{a, b\}^+\}.$$

5. Encuentre una expresión regular para

$$L = \{vwv : v, w \in \{a, b\}^*, |v| = 2\}.$$

3.2. Conexión entre expresiones regulares y lenguajes regulares

Como sugiere la terminología, la conexión entre los lenguajes regulares y las expresiones regulares es estrecha. Los dos conceptos son esencialmente los mismos; para cada lenguaje regular hay una expresión regular, y para cada expresión regular hay un lenguaje regular. Mostraremos esto en dos partes.

3.2.1. Las expresiones regulares denotan lenguajes regulares

Primero mostramos que si r es una expresión regular, entonces $L(r)$ es un lenguaje regular. Nuestra definición dice que un lenguaje es regular si es aceptado por algún dfa. Debido a la equivalencia de nfas y dfas, un lenguaje también es regular si es aceptado por algún nfa.

Ahora demostramos que si tenemos una expresión regular r, podemos construir un nfa que acepte $L(r)$. La construcción de esto se basa en la definición recursiva de $L(r)$. Primero construimos autómatas simples para las partes 1, 2 y 3 de la Definición 3.2, luego mostramos como se pueden combinar para implementar las partes más complicadas 4, 5 y 7.

Teorema 3.1 Sea r una expresión regular. Entonces existe algún aceptador finito no determinista que acepta $L(r)$. En consecuencia, $L(r)$ es un lenguaje regular.

Demostración: Comenzamos con autómatas que aceptan los lenguajes de las expresiones regulares simples $\emptyset, \lambda$ y $a \in \Sigma$. Estos se muestran en la Figura 3.1 (a), (b) y (c), respectivamente.

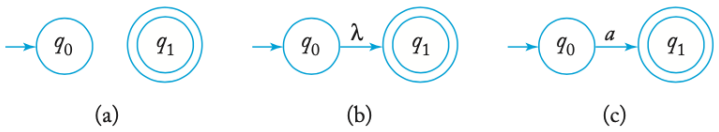


Figura 3.1: (a) nfa que acepta $\emptyset$, (b) nfa que acepta $\{\lambda\}$, (c) nfa que acepta $\{a\}$.

Suponga ahora que tenemos autómatas $M(r_1)$ y $M(r_2)$ que aceptan lenguajes denotados por expresiones regulares r_1 y r_2, respectivamente. No necesitamos construir explícitamente estos autómatas, pero podemos representarlos esquemáticamente, como en la Figura 3.2.

En este esquema, el vértice del grafo a la izquierda representa el estado inicial, el de la derecha el estado final. Afirmamos que para cada nfa hay uno equivalente con un solo estado final, por lo que no perdemos nada al suponer que sólo hay un estado final.

Con $M(r_1)$ y $M(r_2)$ representados de esta manera, luego construimos autómatas para las expresiones regulares r_1+r_2, r_1r_2 y r_1^*. Las construcciones se muestran en las Figuras 3.3 a 3.5.

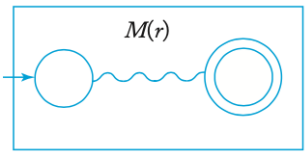


Figura 3.2: Representación esquemática de un nfa que acepta $L(r)$.

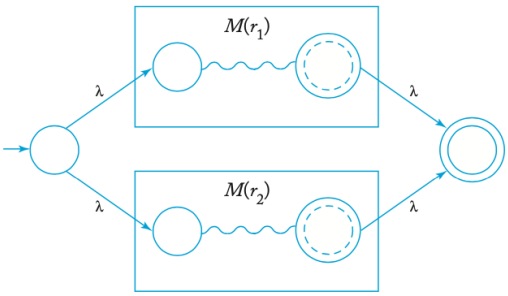


Figura 3.3: Autómata para $L(r_1 + r_2)$.

Como se indica en los dibujos, los estados inicial y final de las máquinas constituyentes pierden su estatus y son reemplazados por nuevos estados inicial y final. Al encadenar varios de estos pasos, podemos construir autómatas para expresiones regulares complejas arbitrarias.

Debe quedar claro de la interpretación de los grafos en las Figuras 3.3 a 3.5 que esta construcción funciona.

Pero, para argumentar más rigurosamente, procedemos por inducción sobre los lenguajes que denotan los diferentes tipos de expresiones regulares r.

Casos base:

$r = \emptyset$**:** formalmente podemos escribir el autómata de la Figura 3.1(a) como $M_\emptyset = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$ con $\delta(q_0, a) = \emptyset$ y $\delta(q_1, a) = \emptyset$, para todo $a \in \Sigma$. Por lo tanto, $L(M_\emptyset) = \emptyset$.

$r = \lambda$**:** formalmente podemos escribir el autómata de la Figura 3.1(b) como $M_\lambda = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$ con $\delta(q_0, \lambda) = \{q_1\}$ y $\delta(q_1, a) = \emptyset$, para todo $a \in \Sigma$. Por lo tanto, $L(M_\lambda) = \{\lambda\}$.

$r = a$**:** formalmente podemos escribir el autómata de la Figura 3.1(c) como

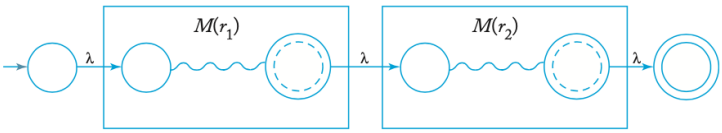


Figura 3.4: Autómata para $L(r_1 r_2)$.

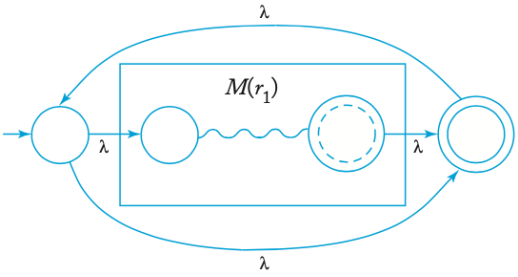


Figura 3.5: Autómata para $L(r_1^*)$.

$M_a = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$ con $\delta(q_0, a) = \{q_1\}$ y $\delta(q_0, b) = \emptyset$, para todo $b \in \Sigma - \{a\}$. Por lo tanto, $L(M_a) = \{a\}$.

Hipótesis de inducción: Sean r_1 y r_2 expresiones regulares, como hipótesis de inducción asumimos que existen nfas

$$M_1 = (Q_1, \Sigma_1, \delta_1, q_{0_1}, \{q_{f_1}\})$$

y

$$M_2 = (Q_2, \Sigma_2, \delta_2, q_{0_2}, \{q_{f_2}\})$$

tales que $L(M_1) = L(r_1)$ y $L(M_2) = L(r_2)$.
Casos inductivos:

$r = r_1 + r_2$**:** Construyamos el autómata para la Figura 3.3

$$M = (Q_1 \cup Q_2 \cup \{q_0, q_f\}, \Sigma_1 \cup \Sigma_2, \delta, q_0, \{q_f\})$$

donde δ está definida por:

1. $\delta(q_0, \lambda) = \{q_{0_1}, q_{0_2}\}$,
2. $\delta(q, a) = \delta_1(q, a)$ para $q \in Q_1 - \{q_{f_1}\}$ y $a \in \Sigma_1 \cup \{\lambda\}$,
3. $\delta(q, a) = \delta_2(q, a)$ para $q \in Q_2 - \{q_{f_2}\}$ y $a \in \Sigma_2 \cup \{\lambda\}$,

4. $\delta(q_{f_1}, \lambda) = \delta(q_{f_2}, \lambda) = \{q_f\}$.

Así, por hipótesis de inducción, si $x \in L(r_1)$ entonces $x \in L(M_1)$, es decir, $\delta_1^*(q_{0_1}, x) = \{q_{f_1}\}$; de tal manera que

$$\begin{aligned}
\delta^*(q_0, x) &= \delta^*(q_0, \lambda x \lambda) \\
&= \delta(\delta^*(\delta(q_0, \lambda), x), \lambda) \\
&\overset{1}{=} \delta(\delta^*(\{q_{0_1}, q_{0_2}\}, x), \lambda) \\
&= \delta(\delta^*(q_{0_1}, x), \lambda) \cup \delta(\delta^*(q_{0_2}, x), \lambda) \\
&\overset{2}{=} \delta(q_{f_1}, \lambda) \cup \delta(\delta^*(q_{0_2}, x), \lambda) \\
&\overset{4}{=} \{q_f\} \cup \delta(\delta^*(q_{0_2}, x), \lambda).
\end{aligned}$$

por lo tanto $x \in L(M)$.

De manera semejante, por hipótesis de inducción, si $x \in L(r_2)$ entonces $x \in L(M_2)$, es decir, $\delta_2^*(q_{0_2}, x) = \{q_{f_2}\}$; de tal manera que

$$\begin{aligned}
\delta^*(q_0, x) &= \delta^*(q_0, \lambda x \lambda) \\
&= \delta(\delta^*(\delta(q_0, \lambda), x), \lambda) \\
&\overset{1}{=} \delta(\delta^*(\{q_{0_1}, q_{0_2}\}, x), \lambda) \\
&= \delta(\delta^*(q_{0_1}, x), \lambda) \cup \delta(\delta^*(q_{0_2}, x), \lambda) \\
&\overset{3}{=} \delta(\delta^*(q_{0_1}, x), \lambda) \cup \delta(q_{f_2}, \lambda) \\
&\overset{4}{=} \delta(\delta^*(q_{0_1}, x), \lambda) \cup \{q_f\}.
\end{aligned}$$

por lo tanto $x \in L(M)$. De tal forma que $L(M) = \{x | x \in L(M_1)$ o $x \in L(M_2)\}$, por lo tanto $L(M) = L(M_1) \cup L(M_2)$.

$r = r_1 r_2$**:** Construyamos el autómata para la Figura 3.4

$$M = (Q_1 \cup Q_2 \cup \{q_0, q_f\}, \Sigma_1 \cup \Sigma_2, \delta, q_0, \{q_f\})$$

donde δ está definida por:

1. $\delta(q_0, \lambda) = \{q_{0_1}\}$,
2. $\delta(q, a) = \delta_1(q, a)$ para $q \in Q_1 - \{q_{f_1}\}$ y $a \in \Sigma_1 \cup \{\lambda\}$,
3. $\delta(q_{f_1}, \lambda) = \{q_{0_2}\}$,

4. $\delta(q,a) = \delta_2(q,a)$ para $q \in Q_2 - \{q_{f_2}\}$ y $a \in \Sigma_2 \cup \{\lambda\}$,
5. $\delta(q_{f_2}, \lambda) = \{q_f\}$.

Sean $x \in L(r_1)$ y $y \in L(r_2)$, por hipótesis de inducción aplicada dos veces, tenemos que $x \in L(M_1)$ y $y \in L(M_2)$. Así, $\delta_1^*(q_{0_1}, x) = \{q_{f_1}\}$ y $\delta_2^*(q_{0_2}, y) = \{q_{f_2}\}$, por lo tanto tenemos:

$$\begin{aligned}
\delta^*(q_0, xy) &= \delta^*(q_0, \lambda x \lambda y \lambda) \\
&= \delta(\delta^*(\delta(\delta^*(\delta(q_0, \lambda), x), \lambda), y), \lambda) \\
&\overset{1}{=} \delta(\delta^*(\delta(\delta^*(q_{0_1}, x), \lambda), y), \lambda) \\
&\overset{2}{=} \delta(\delta^*(\delta(q_{f_1}, \lambda), y), \lambda) \\
&\overset{3}{=} \delta(\delta^*(q_{0_2}, y), \lambda) \\
&\overset{4}{=} \delta(q_{f_2}, \lambda) \\
&\overset{5}{=} \{q_f\}.
\end{aligned}$$

De tal forma que $L(M) = \{xy | x \in L(M_1) \text{ y } y \in L(M_2)\}$, por lo tanto $L(M) = L(M_1)L(M_2)$.

$r = r_1^*$**:** Construyamos el autómata para la Figura 3.5

$$M = (Q_1 \cup \{q_0, q_f\}, \Sigma_1, \delta, q_0, \{q_f\}),$$

donde δ está dada por

1. $\delta(q_0, \lambda) = \{q_{0_1}, q_f\}$,
2. $\delta(q,a) = \delta_1(q,a)$, para todo $q \in Q_1 - \{q_{f_1}\}$ y $a \in \Sigma_1 \cup \{\lambda\}$.
3. $\delta(q_{f_1}, \lambda) = \{q_f\}$,
4. $\delta(q_f, \lambda) = \{q_0\}$,

Recuerde que

$$L^* = \bigcup_{i \geq 0} L^i,$$

donde

$$\begin{aligned}
L^0 &= \{\lambda\}, \\
L^{n+1} &= L^n L, n \geq 0.
\end{aligned}$$

Por ejemplo, L^2 es el lenguaje que se obtine al concatenar dos palabras de L, L^3 es el lenguaje que se obtine al concatenar tres palabras de L y, en general, L^n es el lenguaje que se obtine al concatenar n palabras de L. Por lo tanto, L^* es el lenguaje cuyas palabras son la concatenación de cualquier cantidad de palabras de L.

Demostraremos por inducción sobre n que M acepta cualquier $x \in L(M_1)^*$.

Caso base: Sea $x \in L(M_1)^0$ tenemos que $\delta(q_0, \lambda) \stackrel{1}{=} \{q_{0_1}, q_f\}$, por lo tanto M acepta λ.

Hipótesis de inducción: M acepta cualquier $x \in L(M_1)^n$.

Caso inductivo: Sean $x \in L(M_1)^n$ y $w = xy$, para cualquier $y \in L(M_1)$.

$$\begin{aligned}
\delta^*(q_0, w) &= \delta^*(q_0, x\lambda y\lambda) \\
&= \delta(\delta^*(\delta(\delta^*(q_0, x), \lambda), y), \lambda) \\
&\stackrel{h.i.}{=} \delta(\delta^*(\delta(q_f, \lambda), y), \lambda) \\
&\stackrel{4 \text{ y } 1}{=} \delta(\delta^*(\{q_0, q_{0_1}, q_f\}, y), \lambda) \\
&= \delta(\delta^*(q_{0_1}, y), \lambda) \\
&\stackrel{2}{=} \delta(q_{f_1}, \lambda) \\
&\stackrel{3}{=} q_f.
\end{aligned}$$

Así, M acepta cualquier $w \in L(M_1)^{n+1}$, para $n \geq 0$.

Con esto finalmente hemos demostrado que $L(M) = L(M_1)^*$.

■

Ejemplo 3.7 Encuentre un nfa que acepte $L(r)$ donde

$$r = (a + bb)^*(ba^* + \lambda).$$

Los autómatas para $(a + bb)^*$ y $(ba^* + \lambda)$, construidos directamente a partir de los primeros principios, se muestran en la Figura 3.6. Al juntarlos usando la construcción del teorema 3.1, obtenemos la solución en la Figura 3.7. □

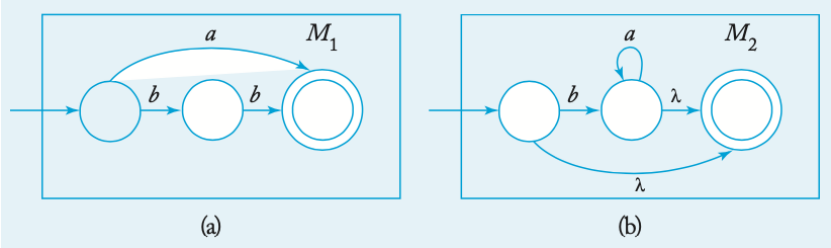


Figura 3.6: (a) M_1 acepta $L(a + bb)$. (b) acepta $L(ba^* + \lambda)$.

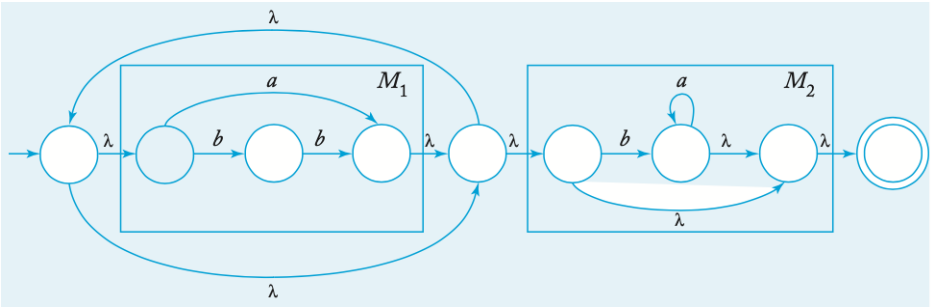


Figura 3.7: El autómata acepta $L((a + bb)^*(ba^* + \lambda))$.

3.2.2. Expresiones regulares para lenguajes regulares

Ahora demostraremos que todo lenguaje aceptado por un autómata finito es denotado por alguna expresión regular.

Observación 3.1 El Teorema 2.2 nos permite obtener un dfa a partir de un nfa, el Teorema 3.2 nos permite obtener una expresión regular a partir de un dfa y el Teorema 3.1 nos permite obtener un nfa a partir de una expresión regular, gracias a estos tres Teoremas concluimos que los tres formalismos son equivalentes.

Teorema 3.2 Si L es aceptado por un dfa, entonces L es denotado por una expresión regular.
Demostración: Sea L el conjunto aceptado por el dfa

$$M = (\{q_1, \ldots, q_n\}, \Sigma, \delta, q_1, F).$$

Sea R_{ij}^k que denota el conjunto de todas las cadenas x tales que $\delta^*(q_i, x) = q_j$ y si $\delta^*(q_i, y) = q_l$ (para cualquier y que es prefijo propio de x, es decir, distinto de x y λ) entonces $l \leq k$.

Esto es, R_{ij}^k es el conjunto de todas las cadenas que llevan al dfa del estado q_i al estado q_j sin pasar por algún estado cuyo número es mayor que k. Note que "pasar por un estado" significa tanto entrar como salir del estado. Así, i y j pueden ser mayores que k.

Ya que no existe un solo estado cuyo número sea mayor que n, R_{ij}^n denota todas las cadenas que llevan al dfa de q_i a q_j. Podemos definir R_{ij}^k recursivamente mediante:

$$R_{ij}^k = R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1} \cup R_{ij}^{k-1}, \tag{3.1}$$

$$R_{ij}^0 = \begin{cases} \{a|\delta(q_i,a) = q_j\} & \text{si } i \neq j, \\ \{a|\delta(q_i,a) = q_j\} \cup \{\lambda\} & \text{si } i = j. \end{cases}$$

Informalmente, la definición de R_{ij}^k significa que las entradas que ocasionan que M vaya de q_i a q_j sin pasar por un estado mayor que q_k están

1. en R_{ij}^{k-1} (es decir, nunca pasan por un estado tan grande como q_k); o

2. compuestas de una cadena en R_{ik}^{k-1} (que lleva a M a q_k por primera vez) seguida por cero o más cadenas en R_{kk}^{k-1} (que lleva a M de q_k de regreso a q_k sin pasar por el estado q_k o algún otro con numeración mayor) seguido por una cadena en R_{kj}^{k-1} (que lleva a M de q_k a q_j).

Debemos demostrar que para cada i, j y k, existe una expresión regular r_{ij}^k que denota al lenguaje R_{ij}^k. Procedemos por inducción sobre k.

Base ($k = 0$). R_{ij}^0 es un conjunto finito de cadenas las cuales son λ o un único símbolo. Así r_{ij}^0 se puede escribir como $a_1 + a_2 + \cdots + a_p$ (o $a_1 + a_2 + \cdots + a_p + \lambda$ si $i = j$), donde $\{a_1, a_2, \ldots, a_p\}$ es el conjunto de símbolos a tales que $\delta(q_i, a) = q_j$. Si no hay tales as, entonces $\emptyset$ (o λ si $i = j$) sirven como r_{ij}^0.

Inducción. La fórmula recursiva para R_{ij}^k dada en (3.1) claramente sólo involucra los operadores de expresiones regulares unión, concatenación y cerradura. Por la hipótesis de inducción, para cada l y m existe una expresión regular r_{lm}^{k-1} tal que $L(r_{lm}^{k-1}) = R_{lm}^{k-1}$. Así, para r_{ij}^k podemos escoger la expresión regular

$$(r_{ik}^{k-1})(r_{kk}^{k-1})^*(r_{kj}^{k-1}) + r_{ij}^{k-1},$$

lo que completa la inducción.

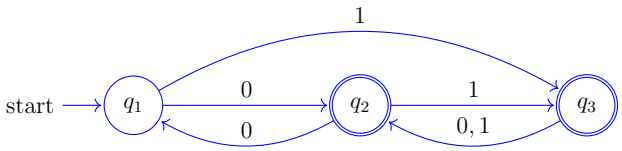


Figura 3.8: Dfa para el Ejemplo 3.8.

Para finalizar la prueba observe que

$$L(M) = \bigcup_{q_j \in F} R^n_{1j}$$

ya que R^n_{1j} denota las etiquetas de todos los caminos de q_1 a q_j. Así, $L(M)$ es denotado por la expresión regular

$$r^n_{1j_1} + r^n_{1j_2} + \cdots + r^n_{1j_p},$$

donde $F = \{q_{j_1}, q_{j_2}, \ldots, q_{j_p}\}$. ■

Ejemplo 3.8 Sea M el dfa de la Figura 3.8, obtenga una expresión regular que denote a $L(M)$.

De acuerdo a la figura tenemos que $F = \{q_2, q_3\}$ y el número de estados es $n = 3$. Por lo tanto la expresión regular que denota a $L(M)$ es

$$r^3_{12} + r^3_{13}.$$

Hagamos algunos cálculos para ilustrar el procedimiento.

$$\begin{aligned}
r_{12}^3 &= r_{13}^2(r_{33}^2)^* r_{32}^2 + r_{12}^2, \\
r_{13}^2 &= r_{12}^1(r_{22}^1)^* r_{23}^1 + r_{13}^1, \\
r_{12}^1 &= r_{11}^0(r_{11}^0)^* r_{12}^0 + r_{12}^0 \\
&= \lambda(\lambda)^* 0 + 0 \\
&= 0 + 0 = 0. \\
r_{22}^1 &= r_{21}^0(r_{11}^0)^* r_{12}^0 + r_{22}^0 \\
&= 0(\lambda)^* 0 + \lambda \\
&= 00 + \lambda. \\
r_{23}^1 &= r_{21}^0(r_{11}^0)^* r_{13}^0 + r_{23}^0 \\
&= 0(\lambda)^* 1 + 1 \\
&= 01 + 1. \\
r_{13}^1 &= r_{11}^0(r_{11}^0)^* r_{13}^0 + r_{13}^0 \\
&= \lambda(\lambda)^* 1 + 1 \\
&= 1 + 1 = 1.
\end{aligned}$$

Una vez hecho estos cálculos ya podemos resolver r_{13}^2.

$$\begin{aligned}
r_{13}^2 &= r_{12}^1(r_{22}^1)^* r_{23}^1 + r_{13}^1 \\
&= 0(00 + \lambda)^*(01 + 1) + 1
\end{aligned}$$

Note que $(00 + \lambda)^*$ es equivalente a $(00)^*$ y que $(01 + 1)$ es equivalente a $(0 + \lambda)1$, así tenemos

$$r_{13}^2 = 0(00)^*(0 + \lambda)1 + 1.$$

Ahora observe que $(00)^*(0+\lambda)$ es equivalente a 0^*. Así que $0(00)^*(0+\lambda)1+1$ es equivalente a $00^*1 + 1$ y ésta es equivalente a 0^*1.

Si bien el procedimiento es bastante claro, también es cierto que es bastante largo hacer todos los cálculos necesarios. □

3.2.3. Ejercicios

1. Use la construcción del Teorema 3.1 para encontrar un nfa para cada uno de los siguientes lenguajes.

a) $L(a^*a + ab)$.

b) $L((aab)^*ab)$.

c) $L(ab^*aa + bba^*ab)$.

2. Encuentre una expresión regular que denote al siguiente lenguaje sobre $\{a, b\}$:
$$L = \{w : n_a(w) \text{ y } n_b(w) \text{ ambos son impares}\}.$$

3. Encuentre un dfa que acepte el siguiente lenguaje.
$$L(aa^* + aba^*b^*).$$

4. Haga los cálculos que faltaron en el Ejemplo 3.8.

3.3. Gramáticas regulares

Una tercera forma de describir los lenguajes regulares es mediante ciertas gramáticas. Las gramáticas son a menudo una forma alternativa de especificar lenguajes. Siempre que definimos una familia de lenguajes a través de un autómata o de alguna otra forma, nos interesa saber qué tipo de gramática podemos asociar con la familia. Primero, veremos las gramáticas que generan lenguajes regulares.

3.3.1. Gramáticas lineales derechas e izquierdas

Definición 3.3 Se dice que una gramática $G = (V, T, S, P)$ es **lineal derecha** si todas las producciones son de la forma

$$A \to xB,$$
$$A \to x,$$

donde $A, B \in V$ y $x \in T^*$. Se dice que una gramática es **lineal izquierda** si todas las producciones son de la forma

$$A \to Bx,$$
$$A \to x.$$

Una **gramática regular** es aquella que es lineal derecha o lineal izquierda.

Tenga en cuenta que en una gramática regular, como máximo, una variable aparece en el lado derecho de cualquier producción. Además, esa variable debe ser consistentemente el símbolo más a la derecha o más a la izquierda del lado derecho de cualquier producción.

Ejemplo 3.9 La gramática $G_1 = (\{S\}, \{a, b\}, S, P_1)$, con P_1 dado como

$$S \to abS|a$$

es lineal derecha. La gramática $G_2 = (\{S, S_1, S_2\}, \{a, b\}, S, P_2)$, con producciones

$$\begin{aligned} S &\to S_1ab, \\ S_1 &\to S_1ab|S_2 \\ S_2 &\to a, \end{aligned}$$

es lineal izquierda. Tango G_1 como G_2 son gramáticas regulare.

La secuencia

$$S \Rightarrow abS \Rightarrow ababS \Rightarrow ababa$$

es una derivación con G_1. A partir de esta única instancia, es fácil conjeturar que $L(G_1)$ es el lenguaje denotado por la expresión regular $r = (ab)^*a$. De manera similar, podemos ver que $L(G_2)$ es el lenguaje regular $L(aab(ab)^*))$. □

Ejemplo 3.10 La gramática $G = (\{S, A, B\}, \{a, b\}, S, P)$ con producciones

$$\begin{aligned} S &\to A, \\ A &\to aB|\lambda, \\ B &\to Ab, \end{aligned}$$

no es regular. Aunque cada producción es lineal derecha o lineal izquierda, la gramática en sí no es lineal derecha ni lineal izquierda y, por lo tanto, no es regular. La gramática es un ejemplo de gramática lineal.

Una gramática lineal es una gramática en la que como máximo una variable puede aparecer en el lado derecho de cualquier producción, sin restricción en la posición de esta variable. Claramente, una gramática regular es siempre lineal, pero no todas las gramáticas lineales son regulares.

□

Nuestro siguiente objetivo será mostrar que las gramáticas regulares están asociadas con los lenguajes regulares y que para cada lenguaje regular hay una gramática regular. Por tanto, las gramáticas regulares son otra forma de hablar de los lenguajes regulares.

3.3.2. Las gramáticas lineales derechas generan lenguajes regulares

Primero, mostramos que un lenguaje generado por una gramática lineal derecha es siempre regular. Para hacerlo, construimos un nfa que imita las derivaciones de una gramática lineal derecha. Tenga en cuenta que las formas oracionales de una gramática lineal derecha tienen la forma especial en la que hay exactamente una variable y aparece como el símbolo más a la derecha. Supongamos ahora que tenemos un paso en una derivación

$$ab \cdots cD \Rightarrow ab \cdots cdE,$$

obtenido mediante el uso de una producción $D \to dE$. El nfa correspondiente puede imitar este paso pasando del estado D al estado E cuando se encuentra un símbolo d. En este esquema, el estado del autómata corresponde a la variable en la forma oracional, mientras que la parte de la cadena ya procesada es idéntica al prefijo terminal de la forma oracional. Esta simple idea es la base del siguiente teorema.

Teorema 3.3 Sea $G = (V, T, S, P)$ una gramática lineal derecha. Entonces $L(G)$ es un lenguaje regular.
Demostración: Suponemos que $V = \{V_0, V_1, \ldots\}$, que $S = V_0$, y que tenemos producciones de la forma $V_0 \to v_1 V_i$, $V_i \to v_2 V_j, \ldots$ o $V_n \to v_l, \ldots$ Si w es una cadena en $L(G)$, entonces debido a la forma de las producciones

$$\begin{aligned} V_0 &\Rightarrow v_1 V_i \\ &\Rightarrow v_1 v_2 V_j \\ &\overset{*}{\Rightarrow} v_1 v_2 \cdots v_k V_n \\ &\Rightarrow v_1 v_2 \cdots v_k v_l = w. \end{aligned} \tag{3.2}$$

El autómata que se construirá reproducirá la derivación consumiendo cada una de estas v por turno. El estado inicial del autómata se etiquetará como

V_0, y para cada variable V_i habrá un estado no final etiquetado como V_i. Para cada producción

$$V_i \to a_1 a_2 \cdots a_m V_j,$$

el autómata tendrá transiciones para conectar V_i y V_j, es decir, δ será definido para que

$$\delta^*(V_i, a_1 a_2 \cdots a_m) = V_j.$$

Para cada producción

$$V_i \to a_1 a_2 \cdots a_m,$$

la correspondiente transición del autómata será

$$\delta^*(V_i, a_1 a_2 \cdots a_m) = V_f,$$

donde V_f es un estado final. Los estados intermedios que se necesitan para hacer esto no son motivo de preocupación y pueden recibir etiquetas arbitrarias. El esquema general se muestra en la Figura 3.9. El autómata completo se ensambla a partir de tales piezas individuales.

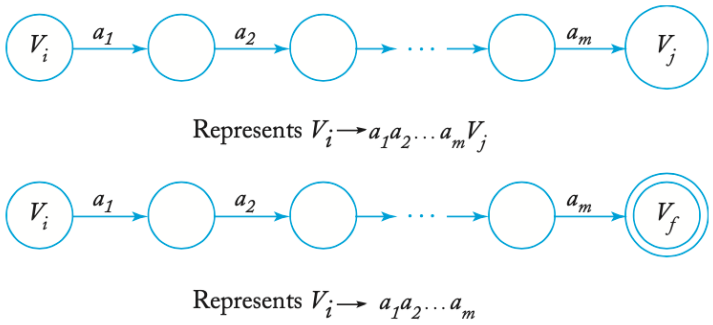


Figura 3.9: Autómatas correspondientes a las producciones de una gramática lineal derecha.

Suponga ahora que $w \in L(G)$ de modo que (3.2) se satisface. En el nfa hay, por construcción, un camino de V_0 a V_i etiquetado v_1, un camino de V_i a V_j etiquetado v_2, y así sucesivamente, de modo que claramente

$$V_f \in \delta^*(V_0, w),$$

y w es aceptado por M.

A la inversa, sup onga que w es aceptado por M. Debido a la forma en que M fue construido, para aceptar w el autómata tiene que pasar por una secuencia de estados $V_0, V_i, \ldots$ a V_f, usando caminos etiquetados como $v_1, v_2, \ldots$. Por lo tanto, w debe tener la forma

$$w = v_1 v_2 \cdots v_k v_l$$

y la derivación

$$V_0 \Rightarrow v_1 V_i \Rightarrow v_1 v_2 V_j \stackrel{*}{\Rightarrow} v_1 v_2 \cdots v_k V_k \Rightarrow v_1 v_2 \cdots v_k v_l$$

es posible. Por tanto, w está en $L(G)$ y se demuestra el teorema.

■

Ejemplo 3.11 Construya un autómata finito que acepte el lenguaje generado por la gramática.

$$V_0 \to aV_1,$$
$$V_1 \to abV_0|b,$$

donde V_0 es la variable inicial. Comenzamos el grafo de transición con los vértices V_0, V_1 y V_f. La primera regla de producción crea una arista etiquetada a entre V_0 y V_1. Para la segunda regla, necesitamos introducir una arista adicional para que haya una ruta etiquetada ab entre V_1 y V_0. Finalmente, necesitamos agregar una arista etiquetado b entre V_1 y V_f, dando el autómata que se muestra en la Figura 3.10. El lenguaje generado por la gramática y aceptado por el autómata es el lenguaje regular $L((aab)^*ab)$.

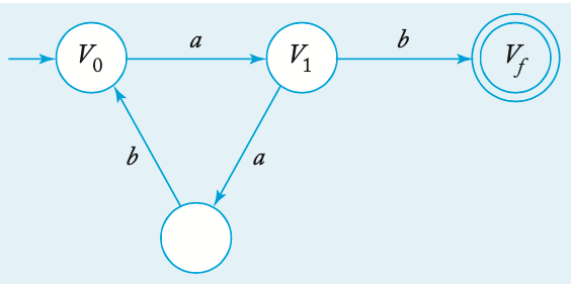


Figura 3.10: Autómata que acepta el lenguaje generado por la gramática lineal derecha del Ejemplo 3.11.

□

3.3.3. Gramáticas lineales derechas para lenguajes regulares

Para mostrar que cada lenguaje regular puede ser generado por alguna gramática lineal derecha, partimos del dfa para el lenguaje e invertimos la construcción mostrada en el Teorema 3.3. Los estados del dfa ahora se convierten en las variables de la gramática, y los símbolos que causan las transiciones se convierten en los terminales de las producciones.

Teorema 3.4 Si L es un lenguaje regular sobre el alfabeto Σ, entonces existe una gramática lineal derecha $G = (V, \Sigma, S, P)$ tal que $L = L(G)$.
Demostración: Sea $M = (Q, \Sigma, \delta, q_0, F)$ un dfa que acepta L. Suponemos que $Q = \{q_0, q_1, \ldots, q_n\}$ y $\Sigma = \{a_1, a_2, \ldots, a_m\}$. Construya la gramática lineal derecha $G = (V, \Sigma, S, P)$ con

$$V = \{q_0, q_1, \ldots, q_n\}$$

y $S = q_0$. Para cada transición

$$\delta(q_i, a_j) = q_k$$

de M, ponemos en P la producción

$$q_i \to a_j q_k. \tag{3.3}$$

Además, si q_k está en F, agregamos a P la producción

$$q_k \to \lambda. \tag{3.4}$$

Primero mostramos que G definido de esta manera puede generar cada cadena en L. Considere $w \in L$ de la forma

$$w = a_i a_j \cdots a_k a_l.$$

Para que M acepte esta cadena, debe hacer movimientos a través de

$$\begin{aligned} \delta(q_0, a_i) &= q_p, \\ \delta(q_p, a_j) &= q_r, \\ &\vdots \\ \delta(q_s, a_k) &= q_t, \\ \delta(q_t, a_l) &= q_f \in F. \end{aligned} \tag{3.5}$$

con la gramática G, y $w \in L(G)$.

Por el contrario, si $w \in L(G)$, entonces su derivación debe tener la forma (3.5). Pero esto implica que

$$\delta^*(q_0, a_i a_j \cdots a_k a_l) = q_f,$$

completando la prueba.

■

Para el propósito de construir una gramática, es útil notar que la restricción de que M sea un dfa no es esencial para la demostración del Teorema 3.4. Con pequeñas modificaciones, se puede usar la misma construcción si M es un nfa.

Ejemplo 3.12 Construya una gramática lineal derecha para $L(aab^*a)$. La función de transición para un nfa, junto con las producciones gramaticales correspondientes, se muestra en la Figura 3.11. El resultado se obtuvo simplemente siguiendo la construcción del Teorema 3.4. La cadena $aaba$ se puede derivar con la gramática construida mediante

$$q_0 \Rightarrow aq_1 \Rightarrow aaq_2 \Rightarrow aabq_2 \Rightarrow aabaq_f \Rightarrow aaba.$$

$\delta(q_0, a) = \{q_1\}$	$q_0 \longrightarrow aq_1$
$\delta(q_1, a) = \{q_2\}$	$q_1 \longrightarrow aq_2$
$\delta(q_2, b) = \{q_2\}$	$q_2 \longrightarrow bq_2$
$\delta(q_2, a) = \{q_f\}$	$q_2 \longrightarrow aq_f$
$q_f \in F$	$q_f \longrightarrow \lambda$

Figura 3.11: Autómata y gramática para el lenguaje del Ejemplo 3.12.

□

3.3.4. Equivalencia de lenguajes regulares y gramáticas regulares

Los dos teoremas anteriores establecen la conexión entre los lenguajes regulares y las gramáticas lineales derechas. Se puede hacer una conexión similar entre los lenguajes regulares y las gramáticas lineales izquierdas, mostrando así la equivalencia completa de las gramáticas regulares y los lenguajes regulares.

Teorema 3.5 Un lenguaje L es regular si y sólo si existe una gramática lineal izquierda G tal que $L = L(G)$.
Demostración: Solo esbozamos la idea principal. Dada cualquier gramática lineal izquierda con producciones de la forma

$$A \to Bv,$$

o

$$A \to v,$$

construimos a partir de ella una gramática lineal a la derecha $\widehat{G}$ reemplazando cada producción de G con

$$A \to v^R B,$$

o

$$A \to v^R,$$

respectivamente. Algunos ejemplos, que el lector debe realizar, aclararán rápidamente que $L(G) = \left(L\left(\widehat{G}\right)\right)^R$. A continuación, recuerde que el reverso de cualquier lenguaje regular también es regular. Dado que $\widehat{G}$ es lineal derecha, $L\left(\widehat{G}\right)$ es regular. Pero también lo son $\left(L\left(\widehat{G}\right)\right)^R$ y $L(G)$.

■

Al juntar los teoremas 3.4 y 3.5, llegamos a la equivalencia de lenguajes regulares y gramáticas regulares.

Teorema 3.6 Un lenguaje L es regular si y sólo si existe una gramática regular G tal que $L = L(G)$.

■

Ahora tenemos varias formas de describir los lenguajes regulares: dfa, nfa, expresiones regulares y gramáticas regulares. Si bien en algunos casos uno u otro de estos pueden ser los más adecuados, todos son igualmente poderosos. Cada uno da una definición completa e inequívoca de un lenguaje regular. La conexión entre todos estos conceptos se establece mediante los cuatro teoremas de este capítulo, como se muestra en la Figura 3.12.

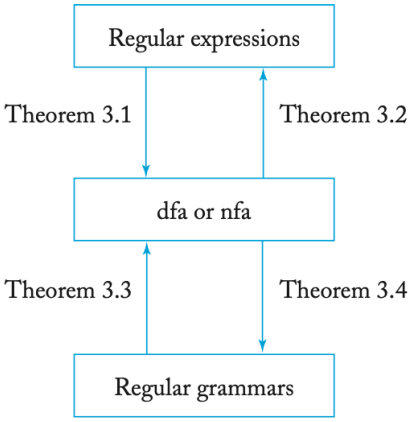


Figura 3.12: La conexión entre las diferentes maneras de definir los lenguajes regulares.

3.3.5. Ejercicios

1. Demuestre que el AFN propuesto en el Ejemplo 3.12 reconoce el lenguaje $\{aab^n a : n \geq 0\}$.

2. Construya un autómata finito que acepte el lenguaje generado por la gramática

$$S \to abA,$$
$$A \to baB,$$
$$B \to aA|bb.$$

3. Construya un autómata finito que acepte el lenguaje generado por la

gramática

$$S \to abS|A,$$
$$A \to baB,$$
$$B \to aA|bb.$$

4. Construya una gramática lineal derecha para el lenguaje $L = \{a^n b^m : n \geq 3, m \geq 2\}$.

5. Construya una gramática lineal derecha para el lenguaje sobre $\Sigma = \{a, b\}$ que consiste de todas las cadenas con no más de dos $a's$.

6. Muestre que para toda gramática lineal derecha que no genere λ existe una gramática lineal derecha cuyas producciones están restringidas a las siguientes dos formas

$$A \to aB \text{ o } A \to a, A, B \in V, a \in T.$$

Capítulo 4

Propiedades de lenguajes regulares

Hemos definido lenguajes regulares, estudiado algunas formas en las que se pueden representar y hemos visto algunos ejemplos de su utilidad. Ahora planteamos la cuestión de cuán generales son los lenguajes regulares. ¿Será que todo lenguaje formal es regular? Quizás cualquier conjunto pueda ser aceptado por algún autómata finito, aunque muy complejo.

Como veremos en breve, la respuesta a esta conjetura es definitivamente no. Pero para comprender por qué esto es así, debemos investigar más profundamente la naturaleza de los lenguajes regulares y ver qué propiedades tiene toda la familia.

La primera pregunta que nos planteamos es qué sucede cuando realizamos operaciones en lenguajes regulares. Las operaciones que consideramos son operaciones de conjuntos simples, como la concatenación, así como operaciones en las que se cambia cada cadena de un lenguaje. ¿El lenguaje resultante sigue siendo regular? Nos referimos a esto como una pregunta de cerradura.

Las propiedades de cerradura, aunque en su mayoría de interés teórico, nos ayudan a discriminar entre las distintas familias de lenguajes que encontraremos.

Un segundo conjunto de preguntas sobre las familias de lenguajes se refiere a nuestra capacidad para decidir sobre determinadas propiedades. Por ejemplo, ¿podemos saber si un lenguaje es finito o no? Como veremos, estas preguntas se responden fácilmente para los lenguajes regulares, pero no son tan fáciles para otras familias de lenguajes.

Finalmente, consideramos la pregunta importante: ¿Cómo podemos saber si un lenguaje determinado es regular o no? Si el lenguaje es de hecho regular, siempre podemos mostrarlo dando algún dfa, expresión regular o gramática regular para él. Pero si no es así, necesitamos otra línea de ataque.

Una forma de mostrar que un lenguaje no es regular es estudiar las propiedades generales de los lenguajes regulares, es decir, las características que comparten todos los lenguajes regulares. Si conocemos alguna propiedad de este tipo, y si podemos demostrar que el lenguaje candidato no la tiene, entonces podemos decir que el lenguaje no es regular.

En este capítulo, examinamos una variedad de propiedades de los lenguajes regulares. Estas propiedades nos dicen mucho sobre lo que los lenguajes regulares pueden y no pueden hacer. Más adelante, cuando analicemos las mismas preguntas para otras familias de lenguajes, las similitudes y diferencias en estas propiedades nos permitirán contrastar las distintas familias de lenguajes.

4.1. Propiedades de cerradura de los lenguajes regulares

Considere la siguiente pregunta: Dados dos lenguajes regulares L_1 y L_2, ¿su unión también es regular? En casos específicos, la respuesta puede ser obvia, pero aquí queremos abordar el problema en general. ¿Es cierto para todos los L_1 y L_2 regulares? Resulta que la respuesta es sí, hecho que expresamos al decir que la familia de los lenguajes regulares es cerrada bajo unión.

Podemos hacer preguntas similares sobre otros tipos de operaciones sobre lenguajes; esto nos lleva al estudio de las propiedades de cerradura de los lenguajes en general.

Las propiedades de cerradura de varias familias de lenguajes bajo diferentes operaciones son de considerable interés teórico. A primera vista, puede que no esté claro qué importancia práctica tienen estas propiedades.

Es cierto que algunos de ellos tienen muy poco, pero muchos resultados son útiles. Al darnos una idea de la naturaleza general de las familias de lenguajes, las propiedades de cerradura nos ayudan a responder otras preguntas más prácticas. Veremos ejemplos de esto (Teorema 4.7 y Ejemplo **??**) más adelante en este capítulo.

4.1.1. Cerradura bajo operaciones simples de conjuntos

Comenzamos mirando la cerradura de lenguajes regulares bajo las operaciones comunes de conjuntos, como unión e intersección.

Teorema 4.1 Si L_1 y L_2 son lenguajes regulares, entonces también lo son $L_1 \cup L_2$, $L_1 \cap L_2$, L_1L_2, $\overline{L_1}$ y L_1^*. Decimos que la familia de los lenguajes regulares es cerrada bajo unión, intersección, concatenación, complemento y cerradura estrella.

Demostración: Si L_1 y L_2 son regulares, entonces existen expresiones regulares r_1 y r_2 tales que $L_1 = L(r_1)$ y $L_2 = L(r_2)$. Por definición, $r_1 + r_2$, r_1r_2 y r_1^* son expresiones regulares que denotan los lenguajes $L_1 \cup L_2$, L_1L_2 y L_1^*, respectivamente. Así, la cerradura bajo unión, concatenación y cerradura estrella es inmediata.

Para mostrar la cerradura bajo complemento, sea $M = (Q, \Sigma, \delta, q_0, F)$ un dfa que acepta L_1. Entonces el dfa

$$\widehat{M} = (Q, \Sigma, \delta, q_0, Q - F)$$

acepta $\overline{L_1}$. Esto es bastante sencillo, tenga en cuenta que en la definición de un dfa, asumimos que δ^* es una función total, de modo que $\delta^*(q_0, w)$ se define para todo $w \in \Sigma^*$. En consecuencia, $\delta^*(q_0, w)$ es un estado final, en cuyo caso $w \in L_1$, o $\delta^*(q_0, w) \in Q - F$ y $w \in \overline{L_1}$.

Demostrar la cerradura en una intersección requiere un poco más de trabajo. Sea $L_1 = L(M_1)$ y $L_2 = L(M_2)$, donde $M_1 = (Q, \Sigma, \delta_1, q_0, F_1)$ y $M_2 = (P, \Sigma, \delta_2, p_0, F_2)$ son dfa. Construimos a partir de M_1 y M_2 un autómata combinado $\widehat{M} = \left(\widehat{Q}, \Sigma, \widehat{\delta}, (q_0, p_0), \widehat{F}\right)$, cuyo conjunto de estados $\widehat{Q} = Q \times P$ consta de pares (q_i, p_j), y cuya función de transición $\widehat{\delta}$ es tal que $\widehat{M}$ está en el estado (q_i, p_j) siempre que M_1 esté en el estado q_i y M_2 esté en el estado p_j. Esto se logra tomando

$$\widehat{\delta}((q_i, p_j), a) = (q_k, p_l),$$

siempre que

$$\delta_1(q_i, a) = q_k$$

y

$$\delta_2(p_j, a) = p_l.$$

$\widehat{F}$ está definido como el conjunto de todos los (q_i, p_j) tales que $q_i \in F_1$ y $p_j \in F_2$. Entonces es muy sencillo demostrar que $w \in L_1 \cap L_2$ si y sólo si es aceptado por $\widehat{M}$. En consecuencia, $L_1 \cap L_2$ es regular.

■

La prueba de cerradura bajo intersección es un buen ejemplo de prueba constructiva. No solo establece el resultado deseado, sino que también muestra explícitamente cómo construir un aceptador finito para la intersección de dos lenguajes regulares.

Las pruebas constructivas ocurren a lo largo de este libro; son importantes porque nos dan una idea de los resultados y, a menudo, sirven como punto de partida para algoritmos prácticos. Aquí, como en muchos casos, hay argumentos más breves pero no constructivos (o al menos no tan obviamente constructivos).

Para la cerradura bajo intersección, comenzamos con la ley de DeMorgan ($\overline{L_1 \cap L_2} = \overline{L_1} \cup \overline{L_2}$) tomando el complemento de ambos lados. Luego

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

para cualesquiera lenguajes L_1 y L_2. Ahora, si L_1 y L_2 son regulares, entonces por cerradura bajo complemento, también lo son $\overline{L_1}$ y $\overline{L_2}$. Usando la cerradura bajo unión, obtenemos a continuación que $\overline{L_1} \cup \overline{L_2}$ es regular. Usando la cerradura bajo complemento una vez más, vemos que

$$\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$$

es regular.

El siguiente ejemplo es una variación de la misma idea.

Ejemplo 4.1 Demuestre que la familia de lenguajes regulares es cerrada bajo la diferencia. En otras palabras, queremos mostrar que si L_1 y L_2 son regulares, entonces $L_1 - L_2$ también es necesariamente regular.

La identidad de conjuntos necesaria es inmediatamente obvia a partir de la definición de diferencia de conjuntos, a saber

$$L_1 - L_2 = L_1 \cap \overline{L_2}.$$

El hecho de que L_2 sea regular implica que $\overline{L_2}$ también lo es. Entonces, debido a la cerradura de los lenguajes regulares bajo intersección, sabemos que $L_1 \cap \overline{L_2}$ es regular y el argumento está completo.

□

Una variedad de otras propiedades de cerradura se pueden derivar directamente mediante argumentos elementales.

Teorema 4.2 La familia de los lenguajes regulares es cerrada bajo reversa.
Demostración: Suponga que L es un lenguaje regular. Luego construimos un nfa con un solo estado final para él. Esto siempre es posible. En el grafo de transición para este nfa, hacemos que el vértice inicial sea un vértice final, el vértice final el vértice inicial, e invertimos la dirección en todas las aristas. Es bastante sencillo mostrar que el nfa modificado acepta w^R si y sólo si el nfa original acepta w. Por lo tanto, el nfa modificado acepta L^R, lo que demuestra la cerradura bajo reversa.

■

4.1.2. Cerradura bajo otras operaciones

Además de las operaciones estándares sobre lenguajes, se pueden definir otras operaciones e investigar propiedades de cerradura para ellas. Hay muchos de estos resultados; seleccionamos sólo dos típicos.

Definición 4.1 Suponga que Σ y Γ son alfabetos. Entonces una función

$$h : \Sigma \to \Gamma^*$$

se llama **homomorfismo**. En palabras, un homomorfismo es una sustitución en la que una sola letra se reemplaza con una cadena. El dominio de la función h se extiende a cadenas de forma obvia; si

$$w = a_1 a_2 \cdots a_n,$$

entonces

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

Si L es un lenguaje sobre Σ, entonces su **imagen homomórfica** se define como

$$h(L) = \{h(w) : w \in L\}.$$

Ejemplo 4.2 Sean $\Sigma = \{a, b\}$ y $\Gamma = \{a, b, c\}$ y defina h por

$$h(a) = ab,$$
$$h(b) = bbc.$$

Entonces $h(aba) = abbbcab$. La imagen homomórfica de $L = \{aa, aba\}$ es el lenguaje $h(L) = \{abab, abbbcab\}$.

□

Si tenemos una expresión regular r para un lenguaje L, entonces se puede obtener una expresión regular para $h(L)$ simplemente aplicando el homomorfismo a cada símbolo Σ de r.

Ejemplo 4.3 Tome $\Sigma = \{a, b\}$ y $\Gamma = \{b, c, d\}$. Defina h por

$$h(a) = dbcc,$$
$$h(b) = bdc.$$

Si L es el lenguaje regular denotado por

$$r = (a + b^*)(aa)^*,$$

entonces

$$r_1 = (dbcc + (bdc)^*)(dbccdbcc)^*$$

denota al lenguaje regular $h(L)$.

□

El resultado general sobre la cerradura de los lenguajes regulares bajo cualquier homomorfismo se deriva de este ejemplo de una manera obvia.

Teorema 4.3 Sea h un homomorfismo. Si L es un lenguaje regular, entonces su imagen homomórfica $h(L)$ también es regular. La familia de los lenguajes regulares es, por tanto, cerrada bajo homomorfismos arbitrarios.
Demostración: Sea L un lenguaje regular denotado por alguna expresión regular r. Encontramos $h(r)$ sustituyendo $h(a)$ por cada símbolo $a \in \Sigma$ de r. Se puede mostrar directamente apelando a la definición de una expresión regular que el resultado es una expresión regular.

Es igualmente fácil ver que la expresión resultante denota $h(L)$. Todo lo que necesitamos hacer es mostrar que para cada $w \in L(r)$, el correspondiente $h(w)$ está en $L(h(r))$ y, a la inversa, para cada v en $L(h(r))$ hay una w en L, tal que $v = h(w)$. Dejando los detalles como ejercicio, afirmamos que $h(L)$ es regular.

■

Definición 4.2 Sean L_1 y L_2 lenguajes sobre el mismo alfabeto. Entonces el cociente derecho de L_1 con L_2 se define como

$$L_1/L_2 = \{x : xy \in L_1 \text{ para algún } y \in L_2\}. \tag{4.1}$$

Para formar el cociente derecho de L_1 con L_2, tomamos todas las cadenas de L_1 que tienen un sufijo que pertenece a L_2. Cada cadena de este tipo, después de eliminar este sufijo, pertenece a L_1/L_2.

Ejemplo 4.4 Si

$$L_1 = \{a^n b^m : n \geq 1, m \geq 0\} \cup \{ba\}$$

y

$$L_2 = \{b^m : m \geq 1\},$$

entonces

$$L_1/L_2 = \{a^n b^m : n \geq 1, m \geq 0\}.$$

Las cadenas de L_2 constan de una o más bs. Por lo tanto, llegamos a la respuesta eliminando una o más bs de esas cadenas en L_1 que terminan con al menos una b.

Tenga en cuenta que aquí L_1, L_2 y L_1/L_2 son todos regulares. Esto sugiere que el cociente derecho de dos lenguajes regulares también es regular. Demostraremos esto en el siguiente teorema mediante una construcción que toma los dfa para L_1 y L_2 y construye a partir de ellos un dfa para L_1/L_2.

Antes de describir la construcción en su totalidad, veamos cómo se aplica a este ejemplo. Comenzamos con un dfa para L_1; digamos que el autómata $M_1 = (Q, \Sigma, \delta, q_0, F)$ en la Figura 4.1. Dado que un autómata para L_1/L_2 debe aceptar cualquier prefijo de cadenas en L_1, intentaremos modificar M_1 para que acepte x si hay alguna y que satisfaga (4.1).

La dificultad radica en encontrar si existe una y tal que $xy \in L_1$ y $y \in L_2$. Para resolverlo, determinamos, para cada $q \in Q$, si hay un camino a un estado final etiquetado v tal que $v \in L_2$. Si es así, cualquier x tal que $\delta^*(q0, x) = q$ estará en L_1/L_2. Modificamos el autómata en consecuencia para hacer q un estado final.

Para aplicar esto a nuestro caso presente, verificamos cada estado q_0, q_1, q_2, q_3, q_4, q_5 para ver si hay un camino etiquetado bb^* a cualquiera de los

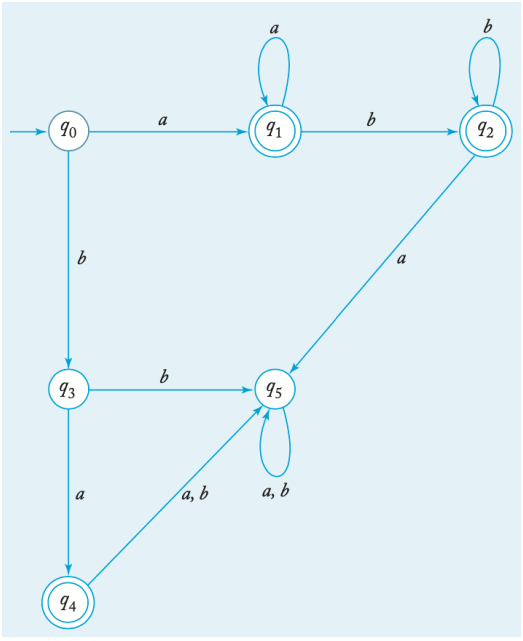


Figura 4.1: DFA que reconoce L_1 para el Ejemplo 4.4.

q_1, q_2 o q_4. Vemos que sólo califican q_1 y q_2; q_0, q_3 y q_4 no. El autómata resultante para L_1/L_2 se muestra en la Figura 4.2. Compruébalo para ver que la construcción funciona. La idea se generaliza en el siguiente teorema. □

Teorema 4.4 Si L_1 y L_2 son lenguajes regulares, entonces L_1/L_2 también lo es. Decimos que la familia de lenguas regulares es cerrada bajo el cociente derecho con un lenguaje regular.

Demostración: Sea $L_1 = L(M)$, donde $M = (Q, \Sigma, \delta, q_0, F)$ es un dfa. Construimos otro dfa $M = (Q, \Sigma, \delta, q_0, \widehat{F})$ como sigue. Para cada $q_i \in Q$, determine si existe una $y \in L_2$ tal que

$$\delta^*(q_i, y) = q_f \in F.$$

Esto se puede hacer mirando $M_i = (Q, \Sigma, \delta, q_i, F)$ del dfa. El autómata M_i es M con el estado inicial q_0 reemplazado por q_i. Ahora determinamos si existe una y en $L(M_i)$ que también esté en L_2.

Para esto, podemos usar la construcción de la intersección de dos lenguajes regulares dada en el Teorema 4.1, encontrando el grafo de transición para

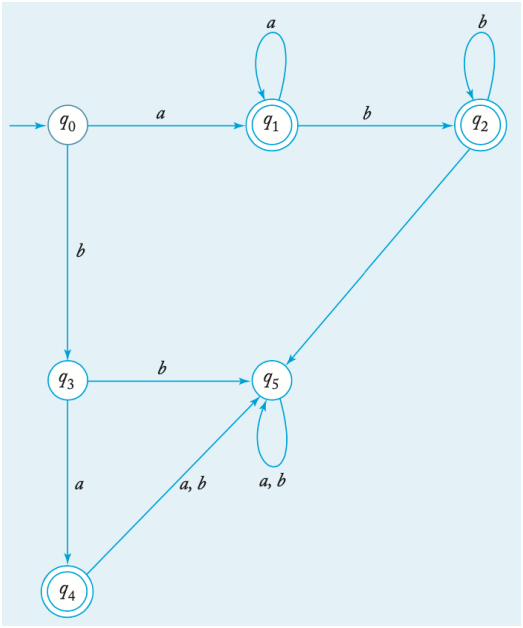


Figura 4.2: DFA que reconoce L_1/L_2 para el Ejemplo 4.4.

$L_2 \cap L(M_i)$. Si hay algún camino entre su vértice inicial y cualquier vértice final, entonces $L_2 \cap L(M_i)$ no está vacío. En ese caso, agregue q_i a $\widehat{F}$. Repitiendo esto para cada $q_i \in Q$, determinamos $\widehat{F}$ y por lo tanto construimos $\widehat{M}$.

Para demostrar que $L\left(\widehat{M}\right) = L_1/L_2$, sea x cualquier elemento de L_1/L_2. Entonces debe haber una $y \in L_2$ tal que $xy \in L_1$. Esto implica que

$$\delta^*(q_0, xy) \in F,$$

así que debe haber algún $q \in Q$ tal que

$$\delta^*(q_0, x) = q$$

y

$$\delta^*(q, y) \in F.$$

Por lo tanto, por construcción, $q \in \widehat{F}$ y $\widehat{M}$ acepta x porque $\delta^*(q_0, x)$ está en $\widehat{F}$.

Por el contrario, para cualquier x aceptada por $\widehat{M}$, tenemos

$$\delta^*(q_0, x) = q \in \widehat{F}.$$

Pero nuevamente por construcción, esto implica que existe un $y \in L_2$ tal que $\delta^*(q, y) \in F$. Por lo tanto, xy está en L_1 y x está en L_1/L_2. Por lo tanto concluimos que

$$L\left(\widehat{M}\right) = L_1/L_2$$

y de esto que L_1/L_2 es regular.

■

Ejemplo 4.5 Encuentre L_1/L_2 para

$$L_1 = L(a^*baa^*),$$
$$L_2 = L(ab^*).$$

Primero encontramos un dfa que acepta L_1. Esto es fácil y se da una solución en la Figura 4.3. El ejemplo es lo suficientemente simple como para que podamos saltarnos las formalidades de la construcción. En el grafo de la Figura 4.3 es bastante evidente que

$$L(M_0) \cap L_2 = \emptyset,$$
$$L(M_1) \cap L_2 = \{a\} \neq \emptyset,$$
$$L(M_2) \cap L_2 = \{a\} \neq \emptyset,$$
$$L(M_3) \cap L_2 = \emptyset.$$

Por tanto, se determina el autómata que acepta L_1/L_2. El resultado se muestra en la Figura 4.4. Acepta el lenguaje denotado por la expresión regular de $a^*b+a^*baa^*$, que se puede simplificar a a^*ba^*. Entonces $L_1/L_2 = L(a^*ba^*)$.
□

4.1.3. Ejercicios

1. Sean $L_1 = L(ab^*aa), L_2 = (a^*bba^*)$. Encuentre una expresión regular para $(L_1 \cup L_2)^*L_2$.

2. Demuestre como a partir de un AFD para L se puede construir un autómata finito para $\overline{L}^*$ (la cerradura estrella del complemento de L).

3. Utilice la construcción del Teorema 4.1 para encontrar nfas que acepten

   a) $L((ab)^*a^*) \cap L(baa^*)$,

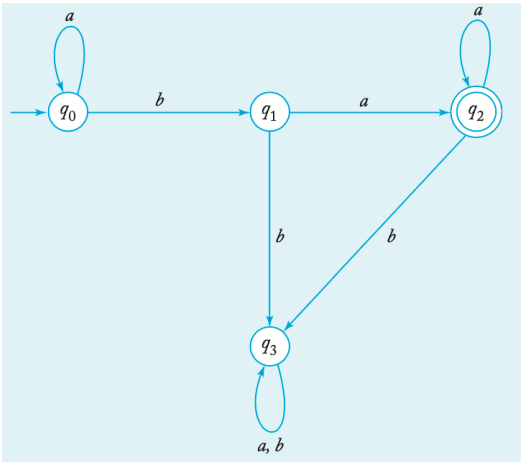


Figura 4.3: DFA que reconoce $L(a^*baa^*)$ para el Ejemplo 4.5.

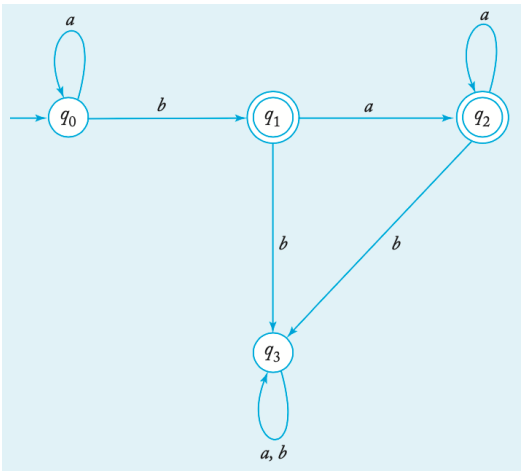


Figura 4.4: DFA que reconoce $L_1/L_2 = L(a^*ba^*)$ para el Ejemplo 4.5.

b) $L(ab^*a^*) \cap L(a^*b^*a)$.

4. La **diferencia simétrica** de dos conjuntos S_1 y S_2 está definida como

$$S_1 \ominus S_2 = \{x : x \in S_1 \text{ o } x \in S_2, \text{ pero } x \text{ no está ni en } S_1 \text{ ni en } S_2\}.$$

Demuestre que la familia de lenguajes regulares es cerrada bajo la diferencia simétrica.

5. El *nor* de dos lenguajes is

$$nor(L_1, L_2) = \{w : w \notin L_1 \text{ y } w \notin L_2\}.$$

6. Sean $L_1 = L(a^*baa^*)$ y $L_2 = L(aba^*)$. Encuentre L_1/L_2.

7. Si L es un lenguaje regular, demuestre que $L_1 = \{uv : u \in L, |v| = 2\}$ también es regular.

8. Si L es un lenguaje regular, demuestre que el lenguaje $\{uv : u \in L, v \in L^R\}$ también es regular.

9. La *cola* de un lenguaje se define como el conjunto de todos los sufijos de sus cadenas, es decir,

$$cola(L) = \{y : xy \in L \text{ para algún } x \in \Sigma^*\}.$$

Demuestre que si L es regular, también lo es $cola(L)$.

10. La *cabeza* de un lenguaje es el conjunto de todos los prefijos de sus cadenas, es decir,

$$cabeza(L) = \{x : xy \in L \text{ para algún } y \in \Sigma^*\}.$$

Muestre que la familia de lenguas regulares es cerrada bajo esta operación.

4.2. Preguntas elementales sobre lenguajes regulares

Llegamos ahora a una cuestión fundamental: dado un lenguaje L y una cadena w, ¿podemos o no determinar si w es un elemento de L? Ésta es la

pregunta de membresía y un método para responderla se llama algoritmo de membresía[1].

Se puede hacer muy poco con lenguajes para los que no podemos encontrar algoritmos de membresía eficientes. La cuestión de la existencia y naturaleza de los algoritmos de membresía será de gran preocupación en discusiones posteriores; es un tema que a menudo es difícil. Sin embargo, para los lenguajes regulares es un asunto fácil.

Primero consideramos qué queremos decir exactamente cuando decimos "dado un lenguaje ..." En muchos argumentos, es importante que esto sea inequívoco. Hemos utilizado varias formas de describir lenguajes regulares: descripciones verbales informales, notación de conjuntos, autómatas finitos, expresiones regulares y gramáticas regulares.

Sólo los tres últimos están suficientemente bien definidos para su uso en teoremas. Por lo tanto, decimos que un lenguaje regular se da en una representación estándar si y solo si está descrito por un autómata finito, una expresión regular o una gramática regular.

Teorema 4.5 Dada una representación estándar de cualquier lenguaje regular L en Σ y cualquier $w \in \Sigma^*$, existe un algoritmo para determinar si w está o no en L.
Demostración: Representamos el lenguaje mediante algún dfa, luego probamos w para ver si es aceptado o no por este autómata.

■

Otras preguntas importantes son si un lenguaje es finito o infinito, si dos lenguajes son iguales y si un lenguaje es un subconjunto de otro. Para los lenguajes regulares, al menos, estas preguntas se responden fácilmente.

Teorema 4.6 Existe un algoritmo para determinar si un lenguaje regular, dado en representación estándar, es vacío, finito o infinito.
Demostración: La respuesta es evidente si representamos el lenguaje como un grafo de transición de un dfa. Si hay una ruta simple desde el vértice inicial a cualquier vértice final, entonces el lenguaje no está vacío.

Para determinar si un lenguaje es infinito o no, encuentre todos los vértices que son la base de algún ciclo. Si alguno de estos está en un camino

[1]Más adelante precisaremos qué significa el término "algoritmo". Por el momento, considérelo un método para escribir un programa de computadora.

desde un vértice inicial hasta un vértice final, el lenguaje es infinito. De lo contrario, es finito.

■

La pregunta de la igualdad de dos lenguajes también es una cuestión práctica importante. A menudo existen varias definiciones de un lenguaje de programación y necesitamos saber si, a pesar de sus diferentes apariencias, especifican el mismo lenguaje.

Este es generalmente un problema difícil; incluso para los lenguajes regulares, el argumento no es obvio. No es posible argumentar sobre una comparación oración por oración, ya que esto funciona solo para lenguajes finitos.

Tampoco es fácil ver la respuesta analizando las expresiones regulares, gramáticas o dfa. Una solución elegante utiliza las propiedades de cerradura ya establecidas.

Teorema 4.7 Dadas las representaciones estándar de dos lenguajes regulares L_1 y L_2, existe un algoritmo para determinar si $L_1 = L_2$ o si no lo es.

Demostración: Usando L_1 y L_2 definimos el lenguaje

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2).$$

Por las propiedades de cerradura, L_3 es regular y podemos encontrar un dfa M que acepta L_3. Una vez que tenemos M, podemos usar el algoritmo del Teorema 4.6 para determinar si L_3 está vacío. Después se puede demostrar que $L_3 = \emptyset$ si y sólo si $L_1 = L_2$.

■

Estos resultados son fundamentales, a pesar de ser obvios y nada sorprendentes. Para los lenguajes regulares, las preguntas planteadas por los teoremas 4.5 a 4.7 pueden responderse fácilmente, pero este no es siempre el caso cuando tratamos con otras familias de lenguajes.

Más adelante nos encontraremos con preguntas como estas en varias ocasiones. Anticipándonos un poco, veremos que las respuestas se vuelven cada vez más difíciles y eventualmente imposibles de encontrar.

4.2.1. Ejercicios

Para todos los ejercicios de esta sección, suponga que los lenguajes regulares se dan en representación estándar.

1. Dado un lenguaje regular L y una cadena $w \in L$, muestre cómo se puede determinar si $w^R \in L$.

2. Exhibir un algoritmo para determinar si un lenguaje regular L contiene alguna cadena w tal que $w^R \in L$.

3. Se dice que un lenguaje es palíndromo si $L = L^R$. Encuentre un algoritmo para determinar si un lenguaje regular dado es un lenguaje palíndromo.

4. Demuestre que existe un algoritmo para determinar si $w \in L_1 - L_2$ para cualquier w dado y cualesquiera lenguajes regulares L_1 y L_2.

5. Demuestre que existe un algoritmo para determinar si $L_1 \subseteq L_2$ para cualesquiera lenguajes regulares L_1 y L_2.

6. Demuestre que existe un algoritmo para determinar si L_1 es un subconjunto propio de L_2 para cualesquiera lenguajes regulares L_1 y L_2.

7. Demuestre que existe un algoritmo para determinar si $\lambda \in L$ para cualquier lenguaje regular L.

8. Demuestre que existe un algoritmo para determinar si $L = \Sigma^*$ para cualquier lenguaje regular L.

9. Muestre un algoritmo que, dados tres lenguajes regulares, L, L_1, L_2, determina si $L = L_1 L_2$ o no.

10. Exhiba un algoritmo que, dado cualquier lenguaje regular L, determina si $L = L^*$ o no.

4.3. Identificación de lenguajes no regulares

Los lenguajes regulares pueden ser infinitos, como han demostrado la mayoría de nuestros ejemplos. Sin embargo, el hecho de que los lenguajes regulares estén asociados con autómatas que tienen memoria finita impone algunos límites a la estructura de un lenguaje regular.

Se deben obedecer algunas restricciones estrictas para mantener la regularidad. La intuición nos dice que un lenguaje es regular sólo si, al procesar cualquier cadena, la información que debe recordarse en cualquier etapa es

estrictamente limitada. Esto es cierto, pero debe demostrarse con precisión para que se utilice de manera significativa. Hay varias formas de hacerlo.

4.3.1. Usando el principio del casillero

Los matemáticos utilizan el término "principio de casillero"para referirse a la siguiente observación simple. Si ponemos n objetos en m cajas (casilleros), y si $n > m$, entonces al menos una caja debe tener más de un elemento. Este es un hecho tan obvio que es sorprendente cuántos resultados profundos se pueden obtener de él.

Ejemplo 4.6 ¿El lenguaje $L = \{a^n b^n : n \geq 0\}$ es regular? La respuesta es no, como mostramos usando una prueba por contradicción.

Suponga que L es regular. Entonces existe algún dfa $M = (Q, \{a, b\}, \delta, q_0, F)$ para él. Ahora mire $\delta^*(q_0, a_i)$ para $i = 1, 2, 3, \ldots$. Dado que hay un número ilimitado de is, pero sólo un número finito de estados en M, el principio de casillero nos dice que debe haber algún estado, digamos q, tal que

$$\delta^*(q_0, a^n) = q$$

y

$$\delta^*(q_0, a^m) = q,$$

con $n \neq m$. Pero ya que M acepta $a^n b^n$ debemos tener

$$\delta^*(q, b^n) = q_f \in F.$$

De aquí podemos concluir que

$$\begin{aligned} \delta^*(q_0, a^m b^n) &= \delta^*(\delta^*(q_0, a^m), b^n) \\ &= \delta^*(q, b^n) \\ &= q_f. \end{aligned}$$

Esto contradice la suposición original de que M acepta $a^m b^n$ sólo si $n = m$, y nos lleva a concluir que L no puede ser regular.

□

En este argumento, el principio del casillero es sólo una forma de enunciar de manera inequívoca lo que queremos decir cuando decimos que un autómata finito tiene una memoria limitada. Para aceptar todos los $a^n b^n$, un autómata

tendría que diferenciar entre todos los prefijos a^n y a^m. Pero como hay sólo un número finito de estados internos con los que hacer esto, hay algunos n y m para los que no se puede hacer la distinción.

Para utilizar este tipo de argumento en una variedad de situaciones, es conveniente codificarlo como un teorema general. Hay varias maneras de hacer esto; el que damos aquí es quizás el más famoso.

Esto es lo que sabemos sobre los grafos de transición para lenguajes regulares:

- Si el grafo de transición no tiene ciclos, el lenguaje es finito y por lo tanto regular.
- Si el grafo de transición tiene un ciclo con una etiqueta no vacía, el lenguaje es infinito. Por el contrario, cada lenguaje regular infinito tiene un dfa con dicho ciclo.
- Si hay un ciclo, este ciclo puede omitirse o repetirse un número arbitrario de veces. Entonces, si el ciclo tiene la etiqueta v y si la cadena w_1vw_2 está en el lenguaje, también deben estar las cadenas w_1w_2, w_1vvw_2, w_1vvvw_2, etc.
- No sabemos en qué parte del dfa se encuentra este ciclo, pero si el dfa tiene m estados, debe ingresarse al ciclo antes de que se hayan leído m símbolos.

Si, para algún lenguaje L, hay incluso una cadena w que no tiene esta propiedad, L no puede ser regular. Esta observación puede enunciarse formalmente como un teorema llamado **lema de bombeo**.

4.3.2. Lema de bombeo

El siguiente resultado, conocido como el **lema de bombeo** para los lenguajes regulares, utiliza el principio de casillero en otra forma. La prueba se basa en la observación de que en un grafo de transición con n vértices, cualquier recorrido de longitud n o más largo debe repetir algún vértice, es decir, contener un ciclo.

Teorema 4.8 Sea L un lenguaje regular infinito. Entonces existe un entero positivo m tal que cualquier $w \in L$ con $|w| \geq m$ se puede descomponer como

$$w = xyz$$

con

$$|xy| \geq m,$$

y

$$|y| \geq 1$$

tal que

$$w^i = xy^iz, \tag{4.2}$$

también está en L para todo $i = 0, 1, 2, \ldots$.

Parafraseando esto, cada cadena suficientemente larga en L puede romperse en tres partes de tal manera que un número arbitrario de repeticiones de la parte del medio produce otra cadena en L. Decimos que la cadena del medio es "bombeada", de ahí el término lema de bombeo para este resultado.
Demostración: Si L es regular, existe un dfa que lo reconoce. Supongamos que tal dfa tiene estados etiquetados como $q_0, q_1, q_2, \ldots, q_n$. Ahora tome una cadena w en L tal que $|w| \geq m = n+1$. Dado que se supone que L es infinito, esto siempre se puede hacer. Considere el conjunto de estados por los que pasa el autómata mientras procesa w, digamos

$$q_0, q_i, q_j, \ldots, q_f.$$

Dado que esta secuencia tiene exactamente $|w| + 1$ entradas, se debe repetir al menos un estado, y dicha repetición debe comenzar a más tardar en el n-ésimo movimiento. Por lo tanto, la secuencia debe verse como

$$q_0, q_i, q_j, \ldots, q_r, \ldots, q_r, \ldots, q_f,$$

indicando que debe haber subcadenas x, y, z de w tales que

$$\begin{aligned} \delta^*(q_0, x) &= q_r, \\ \delta^*(q_r, y) &= q_r, \\ \delta^*(q_r, z) &= q_f, \end{aligned}$$

con $|xy| \leq n + 1 = m$ y $|y| \geq 1$. De esto se sigue inmediatamente que

$$\delta^*(q_0, xz) = q_f,$$

así como

$$\delta^*(q_0, xyz) = q_f,$$
$$\delta^*(q_0, xy^2z) = q_f,$$
$$\delta^*(q_0, xy^3z) = q_f,$$

y así sucesivamente, completando la demostración del teorema.

■

Hemos dado el lema de bombeo solo para lenguajes infinitos. Los lenguajes finitos, aunque siempre regulares, no se pueden bombear, ya que el bombeo crea automáticamente un conjunto infinito. El teorema es válido para lenguajes finitos, pero es vacío. La m en el lema de bombeo debe tomarse más grande que la cadena más larga, de modo que no se pueda bombear ninguna cadena.

El lema de bombeo, como el argumento del casillero en el Ejemplo 4.6, se usa para mostrar que ciertos lenguajes no son regulares. La demostración es siempre por contradicción. No hay nada en el lema de bombeo, como lo hemos dicho aquí, que pueda usarse para demostrar que un lenguaje es regular.

Incluso si pudiéramos mostrar (y esto normalmente es bastante difícil) que cualquier cadena bombeada debe estar en el lenguaje original, no hay nada en el enunciado del Teorema 4.8 que nos permita concluir de esto que el lenguaje es regular.

Ejemplo 4.7 Utilice el lema de bombeo para mostrar que $L = \{a^n b^n : n \geq 0\}$ no es regular. Suponga que L es regular, de modo que el lema de bombeo debe cumplirse. No conocemos el valor de m, pero sea el que sea, siempre podemos elegir $n = m$. Por lo tanto, la subcadena y debe constar completamente de as. Suponga $|y| = k$. Entonces la cadena obtenida usando $i = 0$ en la Ecuación (4.2) es

$$w_0 = a^{m-k}b^m$$

y claramente no está en L. Esto contradice el lema de bombeo y, por lo tanto, indica que la suposición de que L es regular debe ser falsa.

□

Al aplicar el lema de bombeo, debemos tener en cuenta lo que dice el teorema. Se nos garantiza la existencia de una m así como la descomposición xyz, pero no sabemos cuáles son.

No podemos afirmar que hemos llegado a una contradicción solo porque se viola el lema de bombeo para algunos valores específicos de m o xyz.

Por otro lado, el lema de bombeo es válido para cada $w \in L$ y cada i. Por lo tanto, si se viola el lema de bombeo incluso para una w o i, entonces el lenguaje no puede ser regular.

El argumento correcto se puede visualizar como un juego que jugamos contra un oponente. Nuestro objetivo es ganar el juego estableciendo una contradicción del lema de bombeo, mientras el oponente intenta frustrar nuestro intento. Hay cuatro movimientos en el juego.

1. El oponente elige m.

2. Dado m, elegimos una cadena w en L de longitud igual o mayor que m. Somos libres de elegir cualquier w, sujeto a $w \in L$ y $|w| \geq m$.

3. El oponente elige la descomposición xyz, sujeta a $|xy| \leq m$, $|y| \geq 1$. Tenemos que asumir que el oponente toma la decisión que nos dificultará ganar el juego.

4. Tratamos de elegir i de tal manera que la cadena bombeada w_i, definida en la ecuación (4.2), no esté en L. Si podemos hacerlo, ganamos el juego.

   Una estrategia que nos permite ganar cualquiera que sea la elección del oponente equivale a una prueba de que el lenguaje no es regular. En esto, el paso 2 es crucial.

   Si bien no podemos obligar al oponente a elegir una descomposición particular de w, podemos elegir w de modo que el oponente esté muy restringido en el paso 3, forzando una elección de x, y y z que nos permita producir una violación del lema de bombeo en nuestro próximo movimiento.

Ejemplo 4.8 Demuestre que

$$L = \left\{ ww^R : w \in \Sigma^* \right\}$$

no es regular.

Cualquiera que sea la m que elija el oponente en el paso 1, siempre podemos elegir una w como se muestra en la Figura 4.5.

Debido a esta elección y al requisito de que $|xy| \leq m$, el oponente está restringido en el paso 3 a elegir una y que consiste enteramente en as. En el

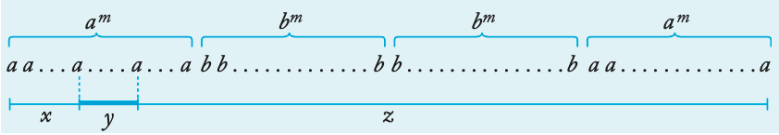


Figura 4.5: Elección de $w = xyz$ para el Ejemplo 4.8.

paso 4, usamos $i = 0$. La cadena obtenida de esta manera tiene menos as a la izquierda que a la derecha y, por lo tanto, no puede tener la forma ww^R. Por tanto, L no es regular.

Tenga en cuenta que si hubiéramos elegido una w demasiado corta, entonces el oponente podría haber elegido una y con un número par de bs. En ese caso, no podríamos haber alcanzado una violación del lema de bombeo en el último paso. También fallaríamos si tuviéramos que elegir una cadena que consta de solamente as, digamos,

$$w = a^{2m},$$

la cual está en L. Para vencernos, el oponente sólo necesita elegir

$$y = aa.$$

Ahora w_i está en L para todo i, y perdemos.

Para aplicar el lema de bombeo, no podemos asumir que el oponente hará un movimiento en falso. Si, en el caso en el que elegimos $w = a^{2m}$ y el oponente eligiera

$$y = a,$$

entonces w_0 es una cadena de longitud impar y, por lo tanto, no está en L. Pero cualquier argumento que asuma que el oponente es tan complaciente es automáticamente incorrecto.

□

Ejemplo 4.9 Sea $\Sigma = \{a, b\}$. El lenguaje

$$L = \{w \in \Sigma^* : n_a(w) < n_b(w)\}$$

no es regular.

Supongamos que se nos da m. Dado que tenemos total libertad para elegir w, elegimos $w = a^m b^{m+1}$. Ahora, ya que $|xy|$ no puede ser mayor que m, el oponente no puede hacer nada más que elegir una y con sólo as, es decir

$$y = a^k, 1 \leq k \leq m.$$

Ahora bombeamos, usando $i = 2$. La cadena resultante

$$w_2 = a^{m+k}b^{m+1}$$

no está en L. Por lo tanto, se viola el lema de bombeo y L no es regular.

□

Ejemplo 4.10 El lenguaje

$$L = \left\{(ab)^n a^k : n > k, k \geq 0\right\}$$

no es regular.

Dado m, elegimos como nuestra cadena

$$w = (ab)^{m+1}a^m,$$

que está en L. Debido a la restricción $|xy| \leq m$, tanto x como y deben estar en la parte de la cadena formada por abs. La elección de x no afecta el argumento, así que veamos qué se puede hacer con y.

Si nuestro oponente elige $y = a$, elegimos $i = 0$ y obtenemos una cadena que no está en $L((ab)^*a^*)$. Si el oponente elige $y = ab$, podemos elegir $i = 0$ nuevamente. Ahora obtenemos la cadena $(ab)^m a^m$, que no está en L. De la misma manera, podemos lidiar con cualquier posible elección del oponente, probando así nuestra afirmación.

□

Ejemplo 4.11 Demuestre que

$$L = \{a^n : n \text{ es un cuadrado perfecto}\}$$

no es regular.

Dada la elección de m del oponente, elegimos

$$w = a^{m^2}.$$

Si $w = xyz$ es la descomposición, entonces claramente

$$y = a^k$$

con $1 \leq k \leq m$. En ese caso,

$$w_0 = a^{m^2-k}.$$

Pero $m^2 - k > (m-1)^2$, por lo que w_0 no puede estar en L. Por lo tanto, el lenguaje no es regular.

□

Ejemplo 4.12 Demuestre que el lenguaje

$$L = \{a^n b^k c^{n+k} : n \geq 0, k \geq 0\}$$

no es regular.

No es difícil aplicar el lema de bombeo directamente, pero es aún más fácil de usar la cerradura bajo homomorfismo. Tome

$$h(a) = a, h(b) = a, h(c) = c,$$

entonces

$$\begin{aligned} h(L) &= \{a^{n+k} c^{n+k} : n + k \geq 0\} \\ &= \{a^i c^i : i \geq 0\}. \end{aligned}$$

Pero sabemos que este lenguaje no es regular; por tanto, L tampoco puede ser regular.

□

El lema de bombeo es difícil de entender y es fácil extraviarse al aplicarlo. A continuación, se muestran algunos errores habituales. Cuidado con ellos.

Un error es intentar usar el lema de bombeo para demostrar que un lenguaje es regular. Incluso si puede demostrar que ninguna cadena en un lenguaje L puede ser bombeada, no puede concluir que L es regular. El lema de bombeo sólo se puede utilizar para demostrar que un lenguaje no es regular.

Otro error es comenzar (generalmente sin darnos cuenta) con una cadena que no esté en L. Por ejemplo, suponga que intentamos mostrar que

$$L = \{a^n : n \text{ es un número primo}\} \tag{4.3}$$

no es regular. Un argumento que comienza con "Dado m, sea $w = a^m \ldots$", es incorrecto ya que m no es necesariamente primo. Para evitar este error, debemos comenzar con algo como "Dado m, sea $w = a^M$, donde M es un número primo mayor que m".

Finalmente, quizás el error más común es hacer algunas suposiciones sobre la descomposición xyz. Lo único que podemos decir sobre la descomposición es lo que nos dice el lema de bombeo, es decir, que y no está vacía y que $|xy| \leq m$; es decir, que y debe estar dentro de m símbolos del extremo izquierdo de la cadena. Cualquier otra cosa invalida el argumento.

Un error típico al intentar demostrar que el lenguaje de la ecuación (4.3) no es regular es decir que $y = a^k$, con k impar. Entonces, por supuesto, $w = xz$ es una cadena de longitud par y, por lo tanto, no está en L. Pero la suposición sobre k no está permitida y la demostración es incorrecta.

Pero incluso si domina las dificultades técnicas del lema de bombeo, puede resultarle difícil saber exactamente cómo utilizarlo. El lema de bombeo es como un juego con reglas complicadas.

El conocimiento de las reglas es esencial, pero eso por sí solo no es suficiente para jugar un buen juego. También necesita una buena estrategia para ganar. Si puede aplicar correctamente el lema de bombeo a algunos de los casos más difíciles de este libro, debe ser felicitado.

4.3.3. Ejercicios

Demuestre que los siguientes lenguajes no son regulares

1. $L = \{a^n b^k c^n : n \geq 0, k \geq 0\}$,
2. $L = \{a^n b^k c^n : n \geq 0, k \geq n\}$,
3. $L = \{a^n b^k c^n d^k : n \geq 0, k > n\}$,
4. $L = \{w : n_a(w) = n_b(w)\}$,
5. $L = \{a^n b^l a^k : k \leq n + l\}$,
6. $L = \{a^n b^l a^k : k \neq n + l\}$,
7. $L = \{a^n b^l a^k : n = l \text{ o } l \neq k\}$,
8. $L = \{a^n b^l : n \geq l\}$,
9. $L = \{w : n_a(w) \neq n_b(w)\}$,
10. $L = \{ww : w \in \{a, b\}^*\}$.

Capítulo 5

Lenguajes libres de contexto

Habiendo descubierto las limitaciones de los lenguajes regulares, ahora vamos a estudiar lenguajes más complicados definiendo un nuevo tipo de gramática, gramáticas libres de contexto y lenguajes libres de contexto asociados.

Si bien las gramáticas libres de contexto aún son bastante simples, pueden tratar con algunos de los lenguajes que sabemos que no son regulares. Esto nos dice que la familia de lenguajes regulares es un subconjunto propio de la familia de lenguajes libres de contexto.

En el último capítulo, descubrimos que no todos los lenguajes son regulares. Si bien los lenguajes regulares son eficaces para describir ciertos patrones simples, no es necesario buscar muy lejos ejemplos de lenguajes no regulares. La relevancia de estas limitaciones para los lenguajes de programación se hace evidente si reinterpretamos algunos de los ejemplos. Si en $L = \{a^*nb^*n : n \geq 0\}$ sustituimos a por un paréntesis izquierdo y b por un paréntesis derecho, entonces cadenas de paréntesis como (()) y ((())) están en L, pero (() no lo está.

Por lo tanto, el lenguaje describe un tipo simple de estructura anidada que se encuentra en los lenguajes de programación, lo que indica que algunas propiedades de los lenguajes de programación requieren algo más que los lenguajes regulares. Para cubrir esta y otras características más complicadas, debemos ampliar la familia de lenguajes. Esto nos lleva a considerar gramáticas y lenguajes libres de contexto.

5.1. Gramáticas libres de contexto

Las producciones en una gramática regular están restringidas de dos maneras: el lado izquierdo debe ser una sola variable, mientras que el lado derecho tiene una forma especial. Para crear gramáticas más poderosas, debemos relajar algunas de estas restricciones. Manteniendo la restricción en el lado izquierdo, pero permitiendo cualquier cosa en el derecho, obtenemos gramáticas libres de contexto.

Definición 5.1 Una gramática $G = (V, T, S, P)$ se dice que es **libre de contexto** si todas las producciones en P tienen la forma

$$A \to x,$$

donde $A \in V$ y $x \in (V \cup T)^*$.

Se dice que un lenguaje L es libre de contexto si y sólo si existe una gramática libre de contexto G tal que $L = L(G)$.

Cada gramática regular es libre de contexto, por lo que un lenguaje regular también lo es. Pero, como sabemos por ejemplos simples como anbn, existen lenguajes no regulares. Ya hemos mostrado en el Ejemplo 1.4 que este lenguaje puede ser generado por una gramática libre de contexto, entonces vemos que la familia de lenguajes regulares es un subconjunto propio de la familia de lenguajes libres de contexto.

Las gramáticas libres de contexto derivan su nombre del hecho de que la sustitución de la variable a la izquierda de una producción puede realizarse en cualquier momento en que dicha variable aparezca en una forma oracional. No depende de los símbolos en el resto de la forma oracional (el contexto). Esta característica es la consecuencia de permitir una sola variable en el lado izquierdo de la producción.

5.1.1. Ejemplos de lenguajes libres de contexto

Ejemplo 5.1 La gramática $G = (\{S\}, \{a, b\}, S, P)$ con producciones

$$S \to aSa,$$
$$S \to bSb,$$
$$S \to \lambda,$$

es libre de contexto. Una derivación típica en la gramática es

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabbaa$$

Esta, y otras derivaciones similares, dejan en claro que

$$L(G) = \{ww^R : w \in \{a, b\}^*\},$$

El lenguaje es libre de contexto, pero como se muestra en el Ejemplo 4.8, no es regular.

□

Ejemplo 5.2 La gramática G con producciones

$$\begin{aligned} S &\to abB, \\ A &\to aaBb, \\ B &\to bbAa, \\ A &\to \lambda, \end{aligned}$$

es libre de contexto. Dejamos que el lector muestre que

$$L(G) = \{ab(bbaa)^n bba(ba)^n : n \geq 0\}.$$

□

Los dos ejemplos anteriores involucran gramáticas que no sólo son libres de contexto, también son lineales. Las gramáticas regulares y lineales son claramente libres de contexto, pero una gramática libre de contexto no es necesariamente lineal.

Ejemplo 5.3 El lenguaje

$$L = \{a^n b^m : n \neq m\}$$

es libre de contexto.

Para mostrar esto, necesitamos producir una gramática libre de contexto para el lenguaje. El caso $n = m$ se resuelve en el Ejemplo 1.4 y podemos basarnos en esa solución. Tomemos el caso $n > m$. Primero generamos una

cadena con un número igual de as y bs, luego agregamos as extras a la izquierda. Esto se hace con

$$\begin{aligned} S &\to AS_1, \\ S_1 &\to aS_1b|\lambda, \\ A &\to aA|a. \end{aligned}$$

Podemos usar un razonamiento similar para el caso $n < m$ y obtenemos la respuesta

$$\begin{aligned} S &\to AS_1|S_1B, \\ S_1 &\to aS_1b|\lambda, \\ A &\to aA|a, \\ B &\to bB|b. \end{aligned}$$

La gramática resultante es libre de contexto, por lo que L es un lenguaje libre de contexto. Sin embargo, la gramática no es lineal.

La forma particular de la gramática dada aquí fue elegida con el propósito de ilustrar; hay muchas otras gramáticas libres de contexto equivalentes. De hecho, hay algunas lineales simples para este lenguaje. Se invita al lector a encontrar alguna.

□

Ejemplo 5.4 Considere la gramática con producciones

$$S \to aSb|SS|\lambda.$$

Esta es otra gramática libre de contexto, pero no lineal. Algunas cadenas de $L(G)$ son *abaabb*, *aababb* y *ababab*. No es difícil conjeturar y demostrar que

$$\begin{aligned} L = \{w \in \{a,b\}^*, n_a(w) = n_b(w) \text{ y } n_a(v) \geq n_b(v), \\ \text{donde } v \text{ es cualquier prefijo de } w\}. \end{aligned} \tag{5.1}$$

Podemos ver claramente la conexión con los lenguajes de programación si reemplazamos a y b con paréntesis izquierdo y derecho, respectivamente. El lenguaje L incluye cadenas tales como (()) y () () () y de hecho es el conjunto de todas las estructuras de paréntesis correctamente anidadas para los lenguajes de programación comunes.

Aquí nuevamente hay muchas otras gramáticas equivalentes. Pero, a diferencia del Ejemplo 5.3, no es tan fácil ver si hay lineales.

□

5.1.2. Derivación más izquierda y derivación más derecha

En una gramática que no es lineal, una derivación puede involucrar formas oracionales con más de una variable. En tales casos, tenemos la opción de elegir el orden en que se reemplazan las variables. Tomemos, por ejemplo, la gramática $G = (\{A, B, S\}, \{a, b\}, S, P)$ con producciones

$$S \to AB, \tag{5.2}$$
$$A \to aaA, \tag{5.3}$$
$$A \to \lambda, \tag{5.4}$$
$$B \to Bb, \tag{5.5}$$
$$B \to \lambda, \tag{5.6}$$

esta gramática genera el lenguaje $L(G) = \{a^{2n}b^m : n \geq 0, m \geq 0\}$. Realice algunas derivaciones para convencerse de esto.

Consideremos ahora las dos derivaciones

$$S \overset{(5,2)}{\Rightarrow} AB \overset{(5,3)}{\Rightarrow} aaAB \overset{(5,4)}{\Rightarrow} aaB \overset{(5,5)}{\Rightarrow} aaBb \overset{(5,6)}{\Rightarrow} aab$$

y

$$S \overset{(5,2)}{\Rightarrow} AB \overset{(5,5)}{\Rightarrow} ABb \overset{(5,3)}{\Rightarrow} aaABb \overset{(5,6)}{\Rightarrow} aaAb \overset{(5,4)}{\Rightarrow} aab.$$

Para mostrar qué producción se aplica, hemos numerado las producciones y escrito el número apropiado sobre el símbolo $\to$. De esto vemos que las dos derivaciones no solo dan la misma oración sino que también usan exactamente las mismas producciones. La diferencia está enteramente en el orden en que se aplican las producciones. Para eliminar tales factores irrelevantes, a menudo requerimos que las variables se reemplacen en un orden específico.

Definición 5.2 Se dice que una derivación es **más izquierda** si en cada paso se reemplaza la variable más a la izquierda en la forma oracional. Si en cada paso se reemplaza la variable más a la derecha, llamamos a la derivación **más derecha**.

Ejemplo 5.5 Considere la gramática con producciones

$$S \to aAB,$$
$$A \to bBb,$$
$$B \to A|\lambda.$$

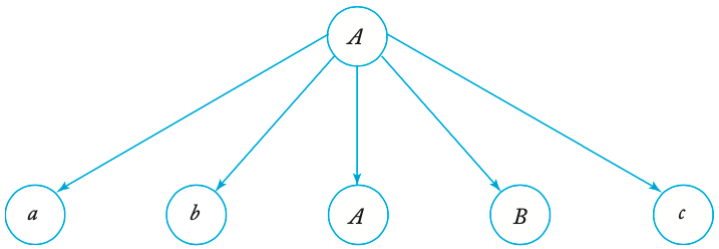


Figura 5.1: Árbol de derivación para la producción $A \to abABc$.

Entonces

$$S \Rightarrow aAB \Rightarrow abBbB \Rightarrow abAbB \Rightarrow abbBbbB \Rightarrow abbbbB \Rightarrow abbbb$$

es una derivación más izquierda de la palabra $abbbb$. Una derivación más derecha de la misma palabra es

$$S \Rightarrow aAB \Rightarrow aA \Rightarrow abBb \Rightarrow abAb \Rightarrow abbBbb \Rightarrow abbbb.$$

□

5.1.3. Árboles de derivación

Una segunda forma de mostrar las derivaciones, independientemente del orden en que se utilicen las producciones, es mediante un árbol de derivación o de análisis sintáctico. Un árbol de derivación es un árbol ordenado en el que los vértices se etiquetan con los lados izquierdos de las producciones y en el que los hijos de un vértice representan sus correspondientes lados derechos. Por ejemplo, la Figura 5.1 muestra parte de un árbol de derivación que representa la producción

$$A \to abABc,$$

En un árbol de derivación, un vértice etiquetado con una variable que aparece en el lado izquierdo de una producción tiene hijos que consisten en los símbolos en el lado derecho de esa producción. Comenzando con la raíz, etiquetada con el símbolo de inicio y terminando en hojas que son terminales, un árbol de derivación muestra cómo se reemplaza cada variable en la derivación. La siguiente definición hace precisa esta noción.

Definición 5.3 Sea $G = (V, T, S, P)$ una gramática libre de contexto. Un árbol ordenado es un árbol de derivación de G si y sólo si tiene las siguientes propiedades.

1. La raíz tiene etiqueta S.

2. Cada hoja tiene una etiqueta en $T \cup \{\lambda\}$.

3. Cada vértice interior tiene una etiqueta en V.

4. Si un vértice tiene una etiqueta $A \in V$ y sus hijos están etiquetados (de izquierda a derecha) $a_1, a_2, \cdots, a_n$, entonces P debe contener una producción de la forma

$$A \to a_1, a_2, \cdots, a_n.$$

5. Una hoja con etiqueta λ no puede tener hermanos, es decir, un vértice con un hijo etiquetado λ no puede tener otro hijo.

A un árbol que tenga las propiedades 3, 4 y 5, que no necesariamente cumpla la propiedad 1 y donde la propiedad 2 se reemplaza por

2a. Toda hoja tiene una etiqueta en $V \cup T \cup \{\lambda\}$,

se le llama **árbol de derivación parcial**.

Se dice que la cadena de símbolos obtenida al leer las hojas del árbol de izquierda a derecha, omitiendo cualquier λ encontrada, es el **producto** del árbol. Al término descriptivo de *izquierda a derecha* se le puede dar un significado preciso. El producto es la cadena de terminales en el orden en que se encuentran cuando se atraviesa el árbol en profundidad, siempre tomando la rama inexplorada más a la izquierda.

Ejemplo 5.6 Considere la gramática con producciones

$$S \to aAB,$$
$$A \to bBb,$$
$$B \to A|\lambda.$$

El árbol de la Figura 5.2 es un árbol de derivación parcial para G, mientras que el árbol de la Figura 5.3 es un árbol de derivación. La cadena $abBbB$, que es el producto del primer árbol, es una forma oracional de G. El producto del segundo árbol, $abbbb$, es una oración de $L(G)$.

□

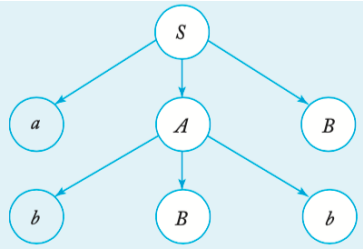


Figura 5.2: Árbol de derivación parcial para la gramática del Ejemplo 5.6.

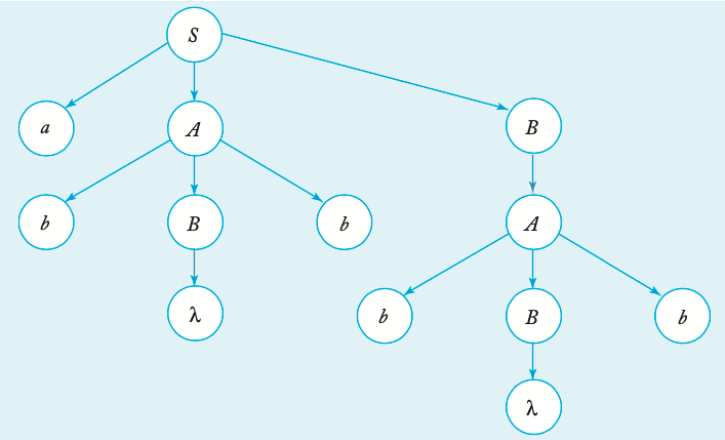


Figura 5.3: Árbol de derivación para la gramática del Ejemplo 5.6.

5.1.4. Relación entre formas sentenciales y árboles de derivación

Los árboles de derivación dan una descripción muy explícita y fácil de comprender de una derivación. Al igual que los grafos de transición para autómatas finitos, esta claridad es de gran ayuda para elaborar argumentos. Sin embargo, primero debemos establecer la conexión entre las derivaciones y los árboles de derivación.

Teorema 5.1 Sea $G = (V, T, S, P)$ una gramática libre de contexto. Entonces para todo $w \in L(G)$, existe un árbol de derivación de G cuyo producto es w. Por el contrario, el producto de cualquier árbol de derivación está en $L(G)$. Además, si t_G es cualquier árbol de derivación parcial de G cuya raíz está etiquetada como S, entonces el producto de t_G es una forma oracional de G.

Demostración: Primero mostramos que para cada forma oracional de $L(G)$ hay un árbol de derivación parcial correspondiente. Hacemos esto por inducción sobre el número de pasos en la derivación. Como base, notamos que el resultado es verdadero para cada forma oracional derivable en un paso. Dado que $S \Rightarrow u$ implica que hay una producción $S \to u$, esto se sigue inmediatamente de la Definición 5.3.

Suponga que para cada forma oracional derivable en n pasos, existe un árbol de derivación parcial correspondiente. Ahora cualquier w derivable en $n + 1$ pasos debe ser tal que

$$S \overset{*}{\Rightarrow} xAy, x, y \in (V \cup T)^*, A \in V,$$

en n pasos, y

$$xAy \Rightarrow xa_1a_2 \cdots a_m y = w; a_i \in V \cup T.$$

Como por la suposición inductiva hay un árbol de derivación parcial con producto xAy, y como la gramática debe tener una producción $A \to a_1a_2 \cdots a_m$, vemos que al expandir la hoja etiquetada como A, obtenemos un árbol de derivación parcial con producto $xa_1a_2 \cdots a_m y = w$. Por inducción, afirmamos que el resultado es verdadero para todas las formas oracionales.

De manera similar, podemos mostrar que todo árbol de derivación parcial representa alguna forma oracional. Dejaremos esto como ejercicio.

Dado que un árbol de derivación es también un árbol de derivación parcial cuyas hojas son terminales, se sigue que cada oración en $L(G)$ es el produc-

to de algún árbol de derivación de G y que el producto de cada árbol de derivación está en $L(G)$.

■

Los árboles de derivación muestran qué producciones se usan para obtener una oración, pero no dan el orden de su aplicación. Los árboles de derivación pueden representar cualquier derivación, lo que refleja el hecho de que este orden es irrelevante, una observación que nos permite cerrar un vacío en la discusión anterior. Por definición, cualquier $w \in L(G)$ tiene una derivación, pero no hemos afirmado que también tenga una derivación más izquierda o más derecha.

Sin embargo, una vez que tenemos un árbol de derivación, siempre podemos obtener una derivación más izquierda pensando en el árbol como construido de tal manera que la variable más a la izquierda en el árbol siempre se expandió primero. Completando algunos detalles, llegamos al resultado no sorprendente de que cualquier $w \in L(G)$ tiene una derivación más izquierda y otra más derecha.

5.1.5. Ejercicios

1. Encuentre gramáticas libres de contexto para los siguientes lenguajes:
   a) $L = \{a^n b^n : n \text{ es par}\}$,
   b) $L = \{a^n b^n : n \text{ es impar}\}$,
   c) $L = \{a^n b^n : n \text{ es múltiplo de tres}\}$,
   d) $L = \{a^n b^m : n \leq m + 3\}$,
   e) $L = \{a^n b^m : n = m - 1\}$.
2. Demuestre que las gramáticas que propuso en el ejercicio anterior generan los respectivos lenguajes.
3. Encuentre una gramática libre de contexto for $\Sigma = \{a, b\}$ para el lenguaje
$$L = \Big\{a^n w w^R b^n : w \in \Sigma^*, n \geq 1\Big\}.$$
4. Sea $L = \{a^n b^n : n \geq 0\}$, demuestre que:
   a) L^2 es libre de contexto.
   b) L^k es libre de contexto para todo $k > 1$.

5.2. Análisis sintáctico

Hasta ahora nos hemos concentrado en los aspectos generativos de las gramáticas. Dada una gramática G, estudiamos el conjunto de cadenas que se pueden derivar usando G. En casos de aplicaciones prácticas, también nos interesa el lado analítico de la gramática: Dada una cadena w de terminales, queremos saber si w está en $L(G)$. Si es así, nos gustaría encontrar una derivación de w.

Un algoritmo que nos puede decir si w está en $L(G)$ es un algoritmo de pertenencia. El término **análisis sintáctico** describe cómo encontrar una secuencia de producciones mediante las cuales se deriva un $w \in L(G)$.

5.2.1. Análisis sintáctico

Dada una cadena w en $L(G)$, podemos analizarla de una manera bastante obvia: construimos sistemáticamente todas las derivaciones posibles (digamos, más a la izquierda) y vemos si alguna de ellas coincide con w. Específicamente, comenzamos en la primera ronda observando todas las producciones de la forma

$$S \to x,$$

encontrar todos las x que se pueden derivar de S en un solo paso. Si ninguna de éstas da como resultado una coincidencia con w, pasamos a la siguiente ronda, en la que aplicamos todas las producciones aplicables a la variable más a la izquierda de cada x. Esto nos da un conjunto de formas oracionales, algunas de las cuales posiblemente conducen a w.

En cada ronda subsiguiente, nuevamente tomamos todas las variables más a la izquierda y aplicamos todas las producciones posibles. Puede ser que algunas de estas formas oracionales puedan ser rechazadas sobre la base de que w nunca puede derivarse de ellas, pero en general, tendremos en cada ronda un conjunto de posibles formas oracionales.

Después de la primera ronda, tenemos las formas oracionales que se pueden derivar aplicando una sola producción, después de la segunda ronda tenemos las formas oracionales que se pueden derivar en dos pasos, y así sucesivamente. Si $w \in L(G)$, entonces debe tener una derivación más izquierda de longitud finita. Por lo tanto, el método eventualmente dará una derivación más izquierda de w.

Como referencia posterior, llamaremos a esto **análisis sintáctico de búsqueda exhaustiva** o **análisis sintáctico de fuerza bruta**. Es una

forma de **análisis sintáctico de arriba hacia abajo**, que podemos ver como la construcción de un árbol de derivación desde la raíz hacia abajo.

Ejemplo 5.7 Considere la gramática

$$S \to SS|aSb|bSa|\lambda$$

y la cadena $w = aabb$. La primera ronda nos da

$$S \Rightarrow SS, \tag{5.7}$$
$$S \Rightarrow aSb, \tag{5.8}$$
$$S \Rightarrow bSa, \tag{5.9}$$
$$S \Rightarrow \lambda. \tag{5.10}$$

Los dos últimos de estos pueden eliminarse de una consideración adicional por razones obvias. La segunda ronda produce las formas oracionales

$$S \Rightarrow SS \Rightarrow SSS,$$
$$S \Rightarrow SS \Rightarrow aSbS,$$
$$S \Rightarrow SS \Rightarrow bSaS,$$
$$S \Rightarrow SS \Rightarrow S,$$

que se obtienen reemplazando la S más a la izquierda en la forma oracional 5.7 con todas las sustituciones aplicables. De manera similar, de la forma oracional 5.8 obtenemos las formas oracionales adicionales

$$S \Rightarrow aSb \Rightarrow aSSb,$$
$$S \Rightarrow aSb \Rightarrow aaSbb,$$
$$S \Rightarrow aSb \Rightarrow abSab,$$
$$S \Rightarrow aSb \Rightarrow ab.$$

Nuevamente, varios de estos pueden eliminarse de la contienda. En la siguiente ronda, encontramos la cadena de destino real a partir de la secuencia

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Por lo tanto, $aabb$ está en el lenguaje generado por la gramática bajo consideración.

□

El análisis sintáctico de búsqueda exhaustiva tiene fallas graves. La más obvia es que es tediosa; no debe usarse cuando se requiere un análisis sintáctico eficiente. Pero incluso cuando la eficiencia es un tema secundario, hay una objeción más pertinente. Si bien el método siempre analiza a $w \in L(G)$, es posible que nunca termine para cadenas que no estén en $L(G)$.

Este es ciertamente el caso en el ejemplo anterior; con $w = abb$, el método continuará produciendo formas oracionales de prueba indefinidamente a menos que incorporemos alguna forma de detenerse.

El problema de la no terminación del análisis sintáctico de búsqueda exhaustiva es relativamente fácil de superar si restringimos la forma que puede tener la gramática. Si examinamos el Ejemplo 5.7, vemos que la dificultad proviene de las producciones $S \to \lambda$; esta producción se puede utilizar para disminuir la longitud de las formas oracionales sucesivas, de modo que no podamos decir fácilmente cuándo detenernos.

Si no tenemos tales producciones, entonces tenemos muchas menos dificultades. De hecho, hay dos tipos de producciones que queremos descartar, las de la forma $A \to \lambda$ así como las de la forma $A \to B$. Como veremos en el próximo capítulo, esta restricción no afecta de forma significativa el poder de las gramáticas resultantes.

Ejemplo 5.8 La gramática

$$S \to SS|aSb|bSa|ab|ba$$

cumple con los requisitos dados. Genera el lenguaje del Ejemplo 5.7 sin la cadena vacía. Dado cualquier $w \in \{a, b\}^+$, el método de análisis sintáctico de búsqueda exhaustiva siempre terminará en no más de $|w|$ rondas. Esto es claro porque la longitud de la forma oracional crece al menos un símbolo en cada ronda. Después de $|w|$ rondas hemos producido un análisis sintáctico o sabemos que $w \notin L(G)$.

□

La idea de este ejemplo se puede generalizar y convertir en un teorema para lenguajes libres de contexto en general.

Teorema 5.2 Supongamos que $G = (V, T, S, P)$ es una gramática libre de contexto que no tiene reglas de la forma

$$A \to \lambda,$$

o

$$A \to B,$$

donde $A, B \in V$. Luego, el método de análisis sintáctico de búsqueda exhaustiva puede convertirse en un algoritmo que, para cualquier $w \in \Sigma^*$, produce un análisis sintáctico de w o nos dice que no es posible realizar ningún análisis.

Demostración: Para cada forma oracional, considere tanto su longitud como el número de símbolos terminales. Cada paso en la derivación aumenta al menos uno de estos. Dado que ni la longitud de una forma oracional ni el número de símbolos terminales pueden exceder $|w|$, una derivación no puede implicar más de $2|w|$ rondas, en cuyo momento tenemos un análisis exitoso o la gramática no puede generar w.

■

Si bien el método de búsqueda exhaustiva ofrece una garantía teórica de que siempre se puede realizar el análisis sintáctico, su utilidad práctica es limitada porque el número de formas oracionales generadas por él puede ser excesivamente grande. Exactamente cuántas formas oracionales se generan difiere de un caso a otro; no se puede establecer un resultado general preciso, pero podemos ponerle algunos límites superiores aproximados.

Si nos restringimos a las derivaciones más a la izquierda, no podemos tener más de $|P|$ formas oracionales después de una ronda, no más de $|P|^2$ formas oracionales después de la segunda ronda, y así sucesivamente. En la demostración del Teorema 5.2, observamos que el análisis sintáctico no puede involucrar más de $2|w|$ rondas; por lo tanto, el número total de formas oracionales no puede exceder

$$\begin{aligned} M &= |P| + |P|^2 + \cdots + |P|^{2|w|} \\ &= O(|P|^{2|w|+1}). \end{aligned} \tag{5.11}$$

Esto indica que el trabajo de análisis sintáctico de búsqueda exhaustiva puede crecer exponencialmente con la longitud de la cadena, lo que hace que el costo del método sea prohibitivo. Por supuesto, la Ecuación (5.11) es solo un límite y, a menudo, el número de formas oracionales es mucho menor. Sin embargo, la observación práctica muestra que el análisis de búsqueda exhaustiva es muy ineficiente en la mayoría de los casos.

La construcción de métodos de análisis sintáctico más eficientes para gramáticas libres de contexto es un asunto que pertenece a un curso sobre compiladores. No lo seguiremos aquí excepto por algunos resultados aislados.

Capítulo 6

Simplificación de gramáticas libres de contexto y formas normales

Antes de que podamos estudiar los lenguajes libres de contexto con mayor profundidad, debemos atender algunos asuntos técnicos. La definición de una gramática libre de contexto no impone restricción alguna sobre el lado derecho de una producción. Sin embargo, la libertad total no es necesaria y, de hecho, es un detrimento en algunos argumentos.

En el Teorema 5.2 vemos la conveniencia de ciertas restricciones sobre las formas gramaticales; eliminar reglas de la forma $A \to \lambda$ y $A \to B$ facilita los argumentos. En muchos casos, es deseable imponer restricciones aún más estrictas a la gramática. Debido a esto, necesitamos buscar métodos para transformar una gramática libre de contexto arbitraria en una equivalente que satisfaga ciertas restricciones en su forma. En este capítulo estudiamos varias transformaciones y sustituciones que serán útiles en discusiones posteriores.

También investigamos **formas normales** para gramáticas libres de contexto. Una forma normal es aquella que, aunque restringida, es lo suficientemente amplia como para que cualquier gramática tenga una versión equivalente en forma normal. Presentamos dos de las más útiles, la **forma normal de Chomsky** y la **forma normal de Greibach**. Ambos tienen muchos usos prácticos y teóricos.

6.1. Métodos para transformar gramáticas

Primero planteamos un problema que es algo molesto con las gramáticas y los lenguajes en general: la presencia de la cadena vacía. La cadena vacía juega un papel bastante singular en muchos teoremas y demostraciones, y a menudo es necesario prestarle especial atención. Preferimos quitarlo de consideración por completo, observando solo los lenguajes que no contienen λ. Al hacerlo, no perdemos generalidad, como vemos a partir de las siguientes consideraciones.

Sea L cualquier lenguaje libre de contexto, y sea $G = (V, T, S, P)$ una gramática libre de contexto para $L - \{\lambda\}$. Entonces la gramática la obtenemos sumando a V la nueva variable S_0, haciendo de S_0 la variable de inicio, y sumando a P las producciones

$$S_0 \to S|\lambda$$

genera L. Por lo tanto, cualquier conclusión no trivial que podamos hacer para $L - \{\lambda\}$ es casi seguro que se transferirá a L. Además, dada cualquier gramática libre de contexto G, hay un método para obtener $\widehat{G}$ tal que $L\left(\widehat{G}\right) = L(G) - \{\lambda\}$.

En consecuencia, a todos los efectos prácticos, no hay diferencia entre los lenguajes libres de contexto que incluyen λ y los que no. Para el resto de este capítulo, a menos que se indique lo contrario, restringiremos nuestra discusión a lenguajes libres de λ.

6.1.1. Una regla de sustitución útil

Muchas reglas gobiernan la generación de gramáticas equivalentes por medio de sustituciones. Aquí damos una que es muy útil para simplificar gramáticas de varias maneras. No definiremos con precisión el término simplificación, pero lo utilizaremos de todos modos. Lo que queremos decir con esto es la eliminación de ciertos tipos de producciones indeseables; el proceso no necesariamente resulta en una reducción real del número de reglas.

Teorema 6.1 Sea $G = (V, T, S, P)$ una gramática libre de contexto. Suponga que P contiene una producción de la forma

$$A \to x_1 B x_2.$$

Asuma que A y B son variables distintas y que

$$B \to y_1|y_2|\cdots|y_n$$

es el conjunto de todas las producciones en P que tienen a B en su lado izquierdo. Sea $\widehat{G} = (V, T, S, \widehat{P})$ la gramática en la que $\widehat{P}$ se construye borrando de P la producción

$$A \to x_1 B x_2 \tag{6.1}$$

y agregando a $\widehat{P}$

$$A \to x_1 y_1 x_2 | x_1 y_2 x_2 | \cdots | x_1 y_n x_2.$$

Entonces, $L\left(\widehat{G}\right) = L(G)$.

Demostración: Suponga que $w \in L(G)$, así que

$$S \stackrel{*}{\Rightarrow}_G w.$$

El subíndice en el signo de derivación $\Rightarrow$ se usa aquí para distinguir entre derivaciones con diferentes gramáticas. Si esta derivación no involucra la producción (6.1), entonces obviamente

$$S \stackrel{*}{\Rightarrow}_{\widehat{G}} w.$$

Si es así, observe la derivación la primera vez que se usa (6.1). La B así introducida eventualmente tiene que ser reemplazada; no perdemos nada al suponer que esto se hace inmediatamente. Por lo tanto

$$S \stackrel{*}{\Rightarrow}_G u_1 A u_2 \Rightarrow_G u_1 x_1 B x_2 u_2 \Rightarrow_G u_1 x_1 y_j x_2 u_2.$$

Pero con la gramática $\widehat{G}$ podemos obtener

$$S \stackrel{*}{\Rightarrow}_{\widehat{G}} u_1 A u_2 \Rightarrow_{\widehat{G}} u_1 x_1 y_j x_2 u_2.$$

Así, podemos alcanzar la misma forma sentencial con G y $\widehat{G}$. Si (6.1) vuelve a aparecer en una forma sentencial posterior, repetimos el argumento. Se sigue, por inducción sobre el número de veces que aparece A en una forma sentencial, que

$$S \stackrel{*}{\Rightarrow}_{\widehat{G}} w.$$

Por lo tanto, si $w \in L(G)$, entonces $w \in L\left(\widehat{G}\right)$.

Por un razonamiento similar, podemos demostrar que si $w \in L\left(\widehat{G}\right)$, entonces $w \in L(G)$, completando la demostración.

■

El Teorema 6.1 es una regla de sustitución simple y bastante intuitiva: una producción $A \to x_1 B x_2$ puede eliminarse de una gramática si ponemos en su lugar el conjunto de producciones en el que B se reemplaza por todas las cadenas que deriva en un solo paso. En este resultado, es necesario que A y B sean variables diferentes.

Ejemplo 6.1 Considere a $G = (\{A, B\}, \{a, b, c\}, A, P)$ con producciones

$$A \to a|aaA|abBc,$$
$$B \to abbA|b.$$

Usando la sustitución sugerida para la variable B, obtenemos la gramática $\widehat{G}$ con producciones

$$A \to a|aaA|ababbAc|abbc,$$
$$B \to abbA|b.$$

La nueva gramática $\widehat{G}$ es equivalente a G. La cadena $aaabbc$ tiene la derivación

$$A \Rightarrow_G aaA \Rightarrow_G aaabBc \Rightarrow_G aaabbc$$

en G, y la correspondiente derivación

$$A \Rightarrow_{\widehat{G}} aaA \Rightarrow_{\widehat{G}} aaabbc$$

en $\widehat{G}$.

Nótese que, en este caso, la variable B y sus producciones asociadas todavía están en la gramática aunque ya no pueden jugar un papel en ninguna derivación. A continuación, mostraremos cómo se pueden eliminar de una gramática tales producciones innecesarias.

□

6.1.2. Eliminación de producciones inútiles

Uno invariablemente quiere quitar producciones de una gramática que nunca puede tomar parte en ninguna derivación. Por ejemplo, en la gramática cuyo conjunto completo de producciones es

$$S \to aSb|\lambda|A,$$
$$A \to aA,$$

la producción $S \to A$ claramente no juega ningún papel, ya que A no puede transformarse en una cadena terminal. Si bien A puede ocurrir en una cadena derivada de S, esto nunca puede conducir a una oración. Eliminar esta producción no afecta el lenguaje y es una simplificación para cualquier definición.

Definición 6.1 Sea $G = (V, T, S, P)$ una gramática libre de contexto. Una variable $A \in V$ se dice que es **útil** si y sólo si existe al menos una $w \in L(G)$ tal que

$$S \overset{*}{\Rightarrow} xAy \overset{*}{\Rightarrow} w, \tag{6.2}$$

con $x, y \in (V \cup T)^*$. En palabras, una variable es útil si y sólo si ocurre en al menos una derivación. A una variable que no es útil se le llama **inútil**. Una producción es inútil si contiene a cualquier variable inútil.

Ejemplo 6.2 Una variable puede ser inútil porque no hay forma de obtener una cadena terminal de ella. El caso que acabamos de mencionar es de este tipo. Otra razón por la que una variable puede ser inútil se muestra en la siguiente gramática. En una gramática con símbolo inicial S y producciones

$$\begin{aligned} &S \to A, \\ &A \to aA|\lambda, \\ &B \to bA, \end{aligned}$$

la variable B es inútil y por lo tanto la producción $B \to bA$. No obstante que a partir de B podemos generar una cadena de terminales, no hay forma de que podamos lograr que $S \overset{*}{\Rightarrow} xBy$.

□

Este ejemplo ilustra las dos razones por las que una variable es inútil: porque no se puede alcanzar desde el símbolo inicial o porque no puede derivar una cadena terminal. Un procedimiento para eliminar variables y producciones inútiles se basa en reconocer estas dos situaciones. Antes de presentar el caso general y el teorema correspondiente, veamos otro ejemplo.

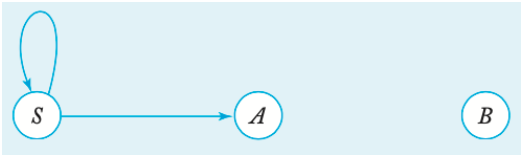


Figura 6.1: Grafo de dependencias para las variables del Ejemplo 6.3.

Ejemplo 6.3 Elimine símbolos inútiles y producciones de $G = (V, T, S, P)$, donde $V = \{S, A, B, C\}$, $T = \{a, b\}$ y P consta de las siguientes producciones

$$S \to aS|A|C,$$
$$A \to a,$$
$$B \to aa,$$
$$C \to aCb.$$

Primero, identificamos el conjunto de variables que pueden conducir a una cadena terminal. Como $A \to a$ y $B \to aa$, las variables A y B pertenecen a este conjunto. S también, porque $S \Rightarrow A \Rightarrow a$. Sin embargo, este argumento no se puede hacer para C, identificándolo así como inútil. Eliminando C y sus correspondientes producciones, llegamos a la gramática G_1 con variables $V_1 = \{S, A, B\}$, terminales $T = \{a\}$ y producciones

$$S \to aS|A,$$
$$A \to a,$$
$$B \to aa.$$

A continuación queremos eliminar las variables que no se pueden alcanzar desde la variable inicial. Para esto, podemos dibujar un **grafo de dependencias** para las variables. Los grafos de dependencias son una forma de visualizar relaciones complejas y se encuentran en muchas aplicaciones.

Para las gramáticas libres de contexto, un grafo de dependencias tiene sus vértices etiquetados con variables, con una arista entre los vértices C y D si y sólo si hay una producción de la forma

$$C \to xDy.$$

En la Figura 6.1 se muestra un grafo de dependencias para V_1. Una variable es útil sólo si hay una ruta desde el vértice etiquetado como S hasta el vértice

etiquetado con esa variable. En nuestro caso, la Figura 6.1 muestra que B es inútil. Quitándola junto con las producciones y terminales afectados, nos quedamos con las producciones

$$S \to aS|A,$$
$$A \to a.$$

La formalización de este proceso conduce a una construcción general y al teorema correspondiente.

□

Teorema 6.2 Sea $G = (V, T, S, P)$ una gramática libre de contexto. Entonces existe una gramática equivalente $\widehat{G} = \left(\widehat{V}, \widehat{T}, S, \widehat{P}\right)$ que no contiene variables ni producciones inútiles.

Demostración: La gramática $\widehat{G}$ puede generarse de G por un algoritmo que consta de dos partes. En la primera parte construimos una gramática intermedia $G_1 = (V_1, T_1, S, P_1)$ tal que V_1 sólo contiene variables A para las cuales

$$A \stackrel{*}{\Rightarrow} w \in T^*$$

es posible. El algoritmo es el siguiente

1. $V_1 \leftarrow \emptyset$.
2. Repita el siguiente paso hasta que no se puedan agregar más variables a V_1. Para todo $A \in V$ para la cual P tenga una producción de la forma

   $$A \to x_1 x_2 \cdots x_n, \text{ con toda } x_i \in V_1 \cup T,$$

   agregue A a V_1.
3. P_1 se compondrá de todas las producciones en P que tengan todos sus símbolos en $V_1 \cup T$.

Claramente este procedimiento termina. Es igualmente claro que si $A \in V_1$, entonces $A \stackrel{*}{\Rightarrow} w \in T^*$ es una derivación posible con G_1. El problema restante es si toda A para la que $A \stackrel{*}{\Rightarrow} w = abc\cdots$ se suma a V_1 antes de que finalice el procedimiento.

Para ver esto, considere cualquier A y mire el árbol de derivación parcial correspondiente a esa derivación (Figura 6.2). En el nivel k, solo hay terminales, por lo que cada variable A_i en el nivel $k-1$ se agregará a V_1 en la primera

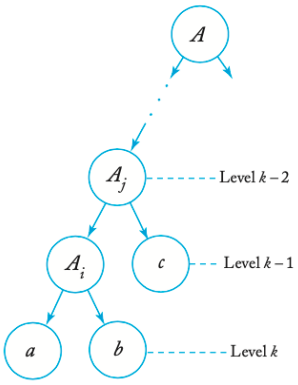


Figura 6.2: Árbol de derivación parcial correspondiente a la derivación $A \overset{*}{\Rightarrow} w = abc\cdots$.

pasada por el Paso 2 del algoritmo. Cualquier variable en el nivel $k-2$ se agregará a V_1 en la segunda pasada por el Paso 2. La tercera vez por el Paso 2, se agregarán todas las variables en el nivel $k-3$, y así sucesivamente. El algoritmo no puede terminar mientras haya variables en el árbol que aún no estén en V_1. Por lo tanto, A eventualmente se agregará a V_1.

En la segunda parte de la construcción, obtenemos la respuesta final $\widehat{G}$ de G_1. Dibujamos el grafo de dependencias para las variables de G_1 y a partir de él encontramos todas las variables que no se pueden alcanzar desde S. Estas se eliminan del conjunto de variables, al igual que las producciones que las involucran. También podemos eliminar cualquier terminal que no se produzca en alguna producción útil. El resultado es la gramática $\widehat{G} = \left(\widehat{V}, \widehat{T}, S, \widehat{P}\right)$.

Debido a la construcción, $\widehat{G}$ no contiene ningún símbolo o producción inútil. Además, para cada $w \in L(G)$ tenemos una derivación

$$S \overset{*}{\Rightarrow}_G xAy \overset{*}{\Rightarrow}_G w.$$

Ya que la construcción de $\widehat{G}$ retiene a A y todas sus producciones asociadas, tenemos todo lo que necesitamos para hacer la derivación

$$S \overset{*}{\Rightarrow}_{\widehat{G}} xAy \overset{*}{\Rightarrow}_{\widehat{G}} w.$$

La gramática $\widehat{G}$ se construye a partir de G mediante la eliminación de producciones, así que $\widehat{P} \subseteq P$, por lo tanto $L\left(\widehat{G}\right) \subseteq L(G)$. Poniendo los dos resultados juntos, vemos que las gramáticas G y $\widehat{G}$ son equivalentes.

■

6.1.3. Eliminación de producciones λ

Un tipo de producción que a veces no es deseable es aquella en la que el lado derecho es la cadena vacía.

Definición 6.2 En una gramática libre de contexto a cualquier producción de la forma

$$A \to \lambda$$

se le llama **producción-λ**. A cualquier variable A para la que es posible la siguiente derivación

$$A \overset{*}{\Rightarrow} \lambda \tag{6.3}$$

se le llama **variable anulable**.

Una gramática puede generar un lenguaje que no contenga λ y, sin embargo, tener algunas producciones-λ o variables anulables. En tales casos, las producciones-λ pueden eliminarse.

Ejemplo 6.4 Considere la gramática dada por las siguientes producciones.

$$S \to aS_1b,$$
$$S_1 \to aS_1b|\lambda,$$

y variable inicial S. Esta gramática genera el lenguaje $\{a^nb^n : n \geq 1\}$ que no contiene a λ. La producción-λ $S_1 \to \lambda$ se puede quitar después de agregar nuevas producciones obtenidas sustituyendo λ por S_1 donde esta variable ocurra a la derecha. Con lo cual obtenemos la gramática dada por las producciones

$$S \to aS_1b|ab,$$
$$S_1 \to aS_1b|ab,$$

Podemos mostrar fácilmente que esta nueva gramática genera el mismo lenguaje que el original.

En situaciones más generales, las sustituciones de producciones-λ se pueden realizar de manera similar, aunque más complicada.

□

Teorema 6.3 Sea G cualquier gramática libre de contexto que no genere a λ. Entonces existe una gramática equivalente $\widehat{G}$ sin producciones-λ.
Demostración: Primero construya el conjunto V_N de todas las variables anulables de G, mediante los siguientes pasos

1. Para todas las producciones $A \to \lambda$ ponga A en V_N.

2. Repita el siguiente paso hasta que no haya más variables que agregar a V_N.

   Para todas las producciones

   $$B \to A_1 A_2 \cdots A_n,$$

   donde $A_1, A_2, \cdots, A_n$ están en V_N, ponga a B en V_N.

Una vez que se ha encontrado el conjunto V_N, estamos listos para construir $\widehat{P}$. Para hacerlo, miramos todas las producciones en P de la forma

$$A \to x_1 x_2 \cdots x_m, m \geq 1,$$

donde cada $x_i \in V \cup T$. Para cada una de esas producciones de P, ponemos en $\widehat{P}$ esa producción, así como todas las generadas reemplazando las variables anulables con λ en todas las combinaciones posibles.

Por ejemplo, si x_i y x_j son ambos anulables, habrá una producción en $\widehat{P}$ con x_i reemplazada por λ, una en la que x_j se reemplaza por λ y otra en la que tanto x_i como x_j se reemplazan por λ. Hay una excepción: si todos los x_i son anulables, la producción $A \to \lambda$ no se pone en $\widehat{P}$. El argumento de que esta gramática $\widehat{G}$ es equivalente a G es sencillo y se deja al lector.

■

Ejemplo 6.5 Encuentre una gramática libre de contexto sin producciones-λ equivalente a la gramática definida mediante

$$\begin{aligned} &S \to ABaC, \\ &A \to BC, \\ &B \to b|\lambda, \\ &C \to D|\lambda, \\ &D \to d. \end{aligned}$$

Del primer paso de la construcción en el Teorema 6.3, encontramos que las variables anulables son A, B, C. Luego, siguiendo el segundo paso de la construcción, obtenemos

$$S \to ABaC|\lambda BaC|A\lambda aC|ABa\lambda|\lambda\lambda aC|A\lambda a\lambda|\lambda Ba\lambda|\lambda\lambda a\lambda,$$
$$A \to B\lambda|\lambda C|BC,$$
$$B \to b,$$
$$C \to D,$$
$$D \to d.$$

□

6.1.4. Eliminación de producciones unitarias

Como hemos visto en el Teorema 5.2, las producciones en las que ambos lados son una sola variable son a veces indeseables.

Definición 6.3 A cualquier producción de una gramática libre de contexto de la forma

$$A \to B,$$

donde $A, B \in V$, se le llama **producción unitaria**.

Para eliminar las producciones unitarias, usamos la regla de sustitución discutida en el Teorema 6.1. Como muestra la construcción del siguiente teorema, esto se puede hacer si procedemos con cierto cuidado.

Teorema 6.4 Sea $G = (V, T, S, P)$ cualquier gramática sin producciones-λ. Entonces existe una gramática libre de contexto $\widehat{G} = \left(\widehat{V}, T, S, \widehat{P}\right)$ que no contiene producciones unitarias y es equivalente a G.
Demostración: Obviamente, cualquier producción unitaria de la forma $A \to A$ se puede quitar sin que haya un efecto, así que sólo es necesario considerar $A \to B$, donde A y B son variables distintas. A primera vista pareciera que podemos usar el Teorema 6.1 directamente con $x_1 = x_2 = \lambda$ para reemplazar

$$A \to B$$

con

$$A \to y_1|y_2|\cdots|y_n.$$

Pero esto no siempre funcionará; en el caso especial

$$A \to B$$
$$B \to A,$$

las producciones unitarias no se eliminan. Para evitar esto, primero encontramos, para cada A, todas las variables B tales que

$$A \stackrel{*}{\Rightarrow} B \tag{6.4}$$

Podemos hacer esto dibujando un grafo de dependencias con un vértice (C, D) siempre que la gramática tenga una producción unitaria $C \to D$; entonces (6.4) se cumple siempre que haya un camino entre A y B. La nueva gramática $\widehat{G}$ se genera poniendo primero en $\widehat{P}$ todas las producciones no unitarias de P. Luego, para toda las A y B que satisfagan (6.4), agregamos a $\widehat{P}$

$$A \to y_1|y_2|\cdots|y_n,$$

donde $B \to y_1|y_2|\cdots|y_n$ son todas las reglas en $\widehat{P}$ que tienen a B del lado izquierdo. Note que como $B \to y_1|y_2|\cdots|y_n$ son tomadas de $\widehat{P}$, ninguna de las y_i puede ser una sola variable, así que no sea crea ninguna producción unitaria en el último paso.

Para demostrar que las gramáticas son equivalentes razonamos como lo hicimos en el Teorema (6.1).

■

Ejemplo 6.6 Quite todas las producciones unitarias de la gramática dada por las producciones

$$S \to Aa|B,$$
$$A \to a|bc|B,$$
$$B \to A|bb.$$

Empezamos agregando a $\widehat{P}$ las producciones no unitarias de P.

$$S \to Aa,$$
$$A \to a|bc,$$
$$B \to bb,$$

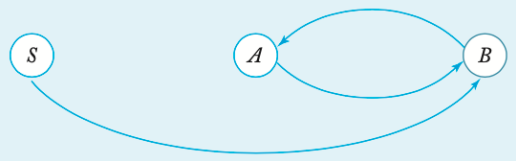


Figura 6.3: Grafo de dependencias para las producciones unitarias del Ejemplo 6.6.

El grafo de dependencias para las producciones unitarias se muestra en la Figura 6.3; vemos que $S \overset{*}{\Rightarrow} A$, $S \overset{*}{\Rightarrow} B$, $A \overset{*}{\Rightarrow} B$ y $B \overset{*}{\Rightarrow} A$. Así que agregamos también a $\widehat{P}$ las nuevas reglas que satisfacen (6.4).

$$S \to a|bc|bb,$$
$$A \to bb,$$
$$B \to a|bc,$$

para obtener la gramática equivalente

$$S \to a|bc|bb|Aa,$$
$$A \to bb|a|bc,$$
$$B \to a|bc|bb,$$

Tenga en cuenta que la eliminación de las producciones unitarias ha hecho que B y las producciones asociadas sean inútiles.

□

Podemos juntar todos estos resultados para mostrar que las gramáticas para lenguajes libres de contexto pueden estar libres de producciones inútiles, producciones-λ y producciones unitarias.

Teorema 6.5 Sea L un lenguaje libre de contexto que no contenga λ. Entonces existe una gramática libre de contexto que general L y que no tiene producciones inútiles, ni producciones-λ, ni producciones unitarias.
Demostración: Los procedimientos dados en los teoremas 6.2, 6.3 y 6.4, eliminan respectivamente esos tipos de producciones. Sin embargo note que

- El procedimiento para eliminar producciones-λ puede crear producciones unitarias.

- El procedimiento para eliminar producciones unitarias requiere que la gramática no tenga producciones-λ.
- Eliminar producciones unitarias no crea producciones-λ.
- Eliminar producciones inútiles no crea ni producciones-λ, ni producciones unitarias.

Por lo tanto, podemos eliminar las producciones indeseables siguiendo la siguiente secuencia de pasos

1. Quite las producciones-λ.
2. Quite las producciones unitarias.
3. Quite las producciones inútiles.

El resultado no tendrá entonces ninguna de estas producciones y el teorema queda demostrado.

■

6.1.5. Ejercicios

1. Elimine la variable B de la gramática

$$S \to aSB|bB,$$
$$B \to aA|b.$$

2. Elimine las producciones inútiles de la gramática

$$S \to aS|AB|\lambda,$$
$$A \to bA,$$
$$B \to AA.$$

3. Elimine las producciones-λ de

$$S \to aSSS,$$
$$S \to bb|\lambda.$$

4. Elimine las producciones-λ de

$$S \to AaB|aaB,$$
$$A \to \lambda,$$
$$B \to bbA|\lambda.$$

5. Elimine las producciones unitarias, las producciones inútiles y las producciones-λ de la gramática dada por las producciones

$$S \to aA|aBB,$$
$$A \to aaA|\lambda,$$
$$B \to bB|bbC,$$
$$C \to B.$$

6.2. Dos formas normales importantes

Hay muchos tipos de formas normales que podemos establecer para gramáticas libres de contexto. Algunas de estas, debido a su amplia utilidad, han sido estudiadas extensamente. Consideramos dos de ellas brevemente.

6.2.1. Forma normal de Chomsky

Un tipo de forma normal que podemos buscar es aquella en la que el número de símbolos a la derecha de una producción está estrictamente limitado. En particular, podemos pedir que la cadena a la derecha de una producción conste de no más de dos símbolos. Un ejemplo de esto es la **forma normal de Chomsky**.

Definición 6.4 Una gramática libre de contexto está en forma normal de Chomsky si todas sus producciones son de la forma

$$A \to BC$$

o

$$A \to a,$$

donde $A, B, C \in V$ y $a \in T$.

Ejemplo 6.7 La gramática dada por las producciones

$$S \to AS|a,$$
$$A \to SA|b$$

está en forma normal de Chomsky. La gramática dada por

$$S \to AS|AAS,$$
$$A \to SA|aa$$

no está en forma normal de Chomsky, las producciones $S \to AAS$ y $A \to aa$ violan las condiciones de la definición 6.4.

□

Teorema 6.6 Cualquier gramática libre de contexto $G = (V, T, S, P)$ con $\lambda \notin L(G)$ tiene una gramática equivalente $\widehat{G} = (\widehat{V}, \widehat{T}, S, \widehat{P})$ en forma normal de Chomsky.

Demostración: Dado el Teorema 6.5, podemos asumir sin pérdida de generalidad que G no tiene producciones-λ, ni producciones unitarias, ni símbolos inútiles. Haremos la construcción de $\widehat{G}$ en dos pasos.

Paso 1: Construya una gramática $G_1 = (V_1, T, S, P_1)$ a partir de G considerando todas las producciones en P de la forma

$$A \to x_1 x_2 \cdots x_n \tag{6.5}$$

donde cada x_i es un símbolo en V o en T. Si $n = 1$, entonces x_1 debe ser un terminal ya que G no tiene producciones unitarias. En este caso, ponga la producción en P_1.

Si $n \geq 2$, introduzca nuevas variables B_a para cada $a \in T$. Para cada producción de P en la forma (6.5) ponga en P_1 la producción

$$A \to C_1 C_2 \cdots C_n$$

donde

$$C_i = x_i, \text{ si } x_i \in V,$$

y

$$C_i = B_a, \text{ si } x_i = a.$$

Para todo B_a también ponga en P_1 la producción

$$B_a \to a.$$

Esta parte del algoritmo quita todos los terminales de las producciones cuyos lados derechos tienen longitud mayor a uno, reemplazándolos con variables nuevas. Al final de este paso tenemos una gramática G_1 donde todas sus producciones tienen la forma

$$A \to a, \tag{6.6}$$

o

$$A \to C_1 C_2 \cdots C_n, \tag{6.7}$$

donde $C_i \in V_1$.

Como consecuencia fácil del Teorema 6.1 (Regla de sustitución) tenemos que

$$L(G_1) = L(G)$$

Paso 2: Se introducen variables adicionales para reducir la longitud de los lados derechos de las producciones, donde sea necesario. Primero, todas las producciones de la forma (6.6) así como las de la forma (6.7) con $n = 2$ se ponen dentro de $\widehat{P}$.

Para cada producción de la forma (6.7) con $n > 2$, se introducen variables $D_1, D_2, \cdots$ y se ponen en $\widehat{P}$ las producciones

$$\begin{aligned} A &\to C_1 D_1 \\ D_1 &\to C_2 D_2 \\ &\vdots \\ D_{n-2} &\to C_{n-1} C_n. \end{aligned}$$

La gramática resultante $\widehat{G}$ está en forma normal de Chomsky. Aplicando varias veces el Teorema 6.1 (Regla de sustitución) se demuestra que $L(G_1) = L\left(\widehat{G}\right)$, así que

$$L\left(\widehat{G}\right) = L(G).$$

Este argumento un tanto informal puede precisarse fácilmente. Dejaremos esto al lector. ■

Ejemplo 6.8 Convierta la grámatica, con las producciones siguientes, a forma normal de Chomsky.

$$\begin{aligned} S &\to ABa, \\ A &\to aab, \\ B &\to Ac. \end{aligned}$$

Como se requiere por la construcción del Teorema 6.6, la gramática no tiene producciones-λ, ni producciones unitarias, ni símbolos inútiles.

En el primer paso introducimos las variables nuevas B_a, B_b y B_c, y hacemos las sustituciones adecuadas para obtener

$$\begin{aligned} S &\to ABB_a, & B_a &\to a, \\ A &\to B_aB_aB_b, & B_b &\to b, \\ B &\to AB_c, & B_c &\to c. \end{aligned}$$

En el segundo paso, se introducen variables adicionales para que las dos primeras producciones queden en forma normal de Chomsky y así obtener el resultado final.

$$\begin{aligned} S &\to AD_1, & B &\to AB_c, \\ D_1 &\to BB_a, & B_a &\to a, \\ A &\to B_aD_2, & B_b &\to b, \\ D_2 &\to B_aB_b, & B_c &\to c. \end{aligned}$$

□

6.2.2. Forma normal de Greibach

Otra forma gramatical útil es la forma normal de Greibach. Aquí ponemos restricciones no sobre la longitud de los lados derechos de una producción, sino sobre las posiciones en las que pueden aparecer terminales y variables.

Los argumentos que justifican la forma normal de Greibach son un poco complicados y no muy transparentes. De manera similar, construir una gramática en la forma normal de Greibach equivalente a una gramática libre de contexto dada es tedioso.

Por lo tanto, tratamos este asunto muy brevemente. Sin embargo, la forma normal de Greibach tiene muchas consecuencias teóricas y prácticas.

Definición 6.5 Una gramática libre de contexto se dice que **está en forma normal de Greibach** si todas las producciones tienen la forma

$$A \to ax,$$

donde $a \in T$ y $x \in V^*$.

Si una gramática no está en la forma normal de Greibach, podemos reescribirla en esta forma con algunas de las técnicas mencionadas anteriormente. Aquí hay dos ejemplos simples.

Ejemplo 6.9 La gramática

$$\begin{aligned} &S \to AB, \\ &A \to aA|bB|b, \\ &B \to b \end{aligned}$$

no está en forma normal de Greibach. Pero aplicando las sustituciones del Teorema 6.1, obtenemos la gramática equivalente

$$\begin{aligned} &S \to aAB|bBB|bB, \\ &A \to aA|bB|b, \\ &B \to b \end{aligned}$$

□

Ejemplo 6.10 Convierta la gramática

$$S \to abSb|aa$$

a su forma normal de Greibach.

Aquí podemos usar un dispositivo similar al introducido en la construcción de la forma normal de Chomsky. Introducimos nuevas variables A y B como sinónimos para los terminales a y b, por supuesto también agregamos las producciones correspondientes, Sustituyendo los terminales por sus respectivas variables obtenemos la gramática

$$\begin{aligned} &S \to aBSB|aA, \\ &A \to a, \\ &B \to b \end{aligned}$$

que está en forma normal de Greibach.

□

En general ni la conversión de una gramática dada a la forma normal de Greibach ni la prueba de que esto siempre se puede hacer es un asunto

sencillo. Introducimos aquí la forma normal de Greibach porque simplificará la discusión técnica de un resultado importante en el próximo capítulo.

Sin embargo, desde un punto de vista conceptual, la forma normal de Greibach no juega ningún otro papel en nuestra discusión, por lo que solo citamos el siguiente resultado general sin demostración.

Teorema 6.7 Para cada gramática libre de contexto G con $\lambda \notin L(G)$, existe una gramática equivalente $\widehat{G}$ en forma normal de Greibach.

■

6.2.3. Ejercicios

1. Convierta las siguientes gramáticas a forma normal de Chomsky.

   a) $S \to aSS|a|b.$

   b) $S \to aSb|Sab|ab.$

   c)

   $$S \to aSaaA|A,$$
   $$A \to abA|bb.$$

   d)

   $$S \to baAB,$$
   $$A \to bAB|\lambda$$
   $$B \to BAa|A|\lambda.$$

   e)

   $$S \to AB|aB,$$
   $$A \to abb|\lambda$$
   $$B \to bbA.$$

2. Convierta las siguientes gramáticas a forma normal de Greibach.

   a)

   $$S \to aSb|bSa|a|b|ab$$

b)

$$S \to aSb|ab|bb$$

c)

$$S \to ab|aS|aaS|aSS$$

d)

$$\begin{aligned} S &\to ABb|a|b, \\ A &\to aaA|B, \\ B &\to bAb. \end{aligned}$$

6.3. Un algoritmo de membresía para gramáticas libres de contexto

En la Sección 5.2, afirmamos, sin ninguna elaboración, que existen algoritmos de membresía y análisis sintáctico para gramáticas libres de contexto que requieren aproximadamente $|w|^3$ pasos para analizar una cadena w. Ahora estamos en condiciones de justificar esta afirmación.

El algoritmo que describiremos aquí se llama algoritmo CYK, en honor a sus creadores J. Cocke, D. H. Younger y T. Kasami. El algoritmo funciona sólo si la gramática está en la forma normal de Chomsky y tiene éxito al dividir un problema en una secuencia de problemas más pequeños de la siguiente manera.

Asuma que tenemos una gramática libre de contexto $G = (V, T, S, P)$ en forma normal de Chomsky y una cadena

$$w = a_1 a_2 \cdots a_n.$$

Defina subcadenas

$$w_{ij} = a_i \cdots a_j,$$

y subconjuntos de V

$$V_{ij} = \left\{ A \in V : A \stackrel{*}{\Rightarrow} w_{ij} \right\}.$$

Claramente, $w \in L(G)$ si y sólo si $S \in V_{1n}$.

Para calcular V_{ij}, observe que $A \in V_{ii}$ si y sólo si G contiene una producción $A \to a_i$. Por lo tanto, V_{ii} se puede calcular para todo $1 \leq i \leq n$

inspeccionando a w y las producciones de la gramática.Para continuar, note que para todo $j > i$, A deriva w_{ij} si y sólo si existe una producción $A \to BC$, con $B \overset{*}{\Rightarrow} w_{ik}$ y $C \overset{*}{\Rightarrow} w_{k+1j}$ para algún k con $i \leq k, k < j$. En otras palabras

$$V_{ij} = \bigcup_{k \in \{i, i+1, \cdots, j-1\}} \left\{ A : A \to BC, \text{ con } B \in V_{ik}, C \in V_{k+1j} \right\} \tag{6.8}$$

Una inspección a los índices en (6.8) muestra que se pueden usar para calcular todos los V_{ij} si procedemos en la secuencia.

1. Calcule $V_{11}, V_{22}, \cdots, V_{nn}$,
2. Calcule $V_{12}, V_{23}, \cdots, V_{n-1n}$,
3. Calcule $V_{13}, V_{24}, \cdots, V_{n-2n}$,
4. Calcule $V_{14}, V_{25}, \cdots, V_{n-3n}$,
5. $\vdots$
6. Calcule $V_{1n-2}, V_{2n-1}, V_{3n}$,
7. Calcule V_{1n-1}, V_{2n},
8. Calcule V_{1n}.

Ejemplo 6.11 Determine si la cadena $w = aabbb$ está en el lenguaje generado por la gramática dada por las producciones

$$\begin{aligned} S &\to AB, \\ A &\to BB|a, \\ B &\to AB|b. \end{aligned}$$

Primero note que $w_{11} = a$ el conjunto V_{11} de todas las variables que derivan inmediatamente a a es $V_{11} = \{A\}$. Ya que $w_{22} = a$, también tenemos que $V_{22} = \{A\}$ y, similarmente tenemos que

$$V_{11} = \{A\}, V_{22} = \{A\}, V_{33} = \{B\}, V_{44} = \{B\}, V_{55} = \{B\}.$$

Ahora usamos (6.8) para obtener

$$V_{12} = \{A : A \to BC, B \in V_{11}, C \in V_{22}\},$$

la única variable de V_{11} es A y la única variable de V_{22} también es A, por lo tanto buscamos AA como parte derecha de alguna de las reglas, como no está entonces $V_{12} = \emptyset$.

Para V_{23}, buscamos AB del lado derecho de las producciones, vemos que tanto S como B derivan AB, por lo tanto $V_{23} = \{S, B\}$.

Para V_{34}, buscamos BB del lado derecho de las producciones, vemos que sólo A deriva a BB, por lo tanto $V_{34} = \{A\}$.

Para V_{45}, buscamos BB del lado derecho de las producciones, vemos que sólo A deriva a BB, por lo tanto $V_{45} = \{A\}$.

Para V_{13} buscamos del lado derecho de las producciones a AS y AB, la primera no se encuentra pero a la segunda la pueden derivar S y B, así $V_{13} = \{S, B\}$. Siguiendo de esta manera obtenemos:

$$V_{13} = \{S, B\}, V_{24} = \{A\}, V_{35} = \{S, B\}.$$

Para V_{14} se tienen que analizar $V_{11}V_{24}, V_{12}V_{34}, V_{13}V_{44}$.
Para V_{25} se tienen que analizar $V_{22}V_{35}, V_{23}V_{45}, V_{24}V_{55}$.
Para V_{15} se tienen que analizar $V_{11}V_{25}, V_{12}V_{35}, V_{13}V_{45}, V_{14}V_{55}$.
Obteniendo finalmente

$$V_{11} = \{A\}, V_{22} = \{A\}, V_{33} = \{B\}, V_{44} = \{B\}, V_{55} = \{B\}.$$
$$V_{12} = \emptyset, V_{23} = \{S, B\}, V_{34} = \{A\}, V_{45} = \{A\}.$$
$$V_{13} = \{S, B\}, V_{24} = \{A\}, V_{35} = \{S, B\}.$$
$$V_{14} = \{A\}, V_{25} = \{S, B\}.$$
$$V_{15} = \{S, B\}.$$

Y como $S \in V_{15}$ entonces $aabbb \in L(G)$.

□

El algoritmo CYK, como se describe aquí, determina la pertenencia a cualquier lenguaje generado por una gramática en la forma normal de Chomsky. Con algunas adiciones para realizar un seguimiento de cómo se derivan los elementos de V_{ij}, se puede convertir en un método de análisis.

Para ver que el algoritmo de membresía CYK requiere $O(n^3)$ pasos, observe que se deben calcular exactamente $n(n+1)/2$ conjuntos de V_{ij}. Cada uno implica la evaluación de, como máximo, n términos en (6.8), por lo que se sigue el resultado declarado.

6.3.1. Ejercicios

1. Use el algoritmo CYK para saber si las siguientes palabras están en el lenguaje generado por la gramática del Ejemplo 6.11.

   a) abb, *c*) $aabba$, *e*) $aaabbb$,
   b) bbb, *d*) $abbbb$, *f*) $aaaabbb$.

2. Utilice el método CYK para determinar si la cadena $w = aaabbbb$ está en el lenguaje generado por la gramática $S \to aSb|b$.

Capítulo 7

Autómatas de pila

La descripción de lenguajes libres de contexto por medio de gramáticas libres de contexto es conveniente, como lo ilustra el uso de BNF en la definición de lenguajes de programación. La siguiente pregunta es si existe una clase de autómatas que se pueda asociar con lenguajes libres de contexto. Como hemos visto, los autómatas finitos no pueden reconocer todos los lenguajes libres de contexto.

Intuitivamente, entendemos que esto se debe a que los autómatas finitos tienen memorias estrictamente finitas, mientras que el reconocimiento de un lenguaje libre de contexto puede requerir el almacenamiento de una cantidad ilimitada de información. Por ejemplo, al escanear una cadena del lenguaje $L = \{a^n b^n : n \geq 0\}$, no sólo debemos verificar que todas las a preceden a la primera b, también debemos contar el número de a. Dado que n no está acotado, este conteo no se puede realizar con una memoria finita.

Queremos una máquina que pueda contar sin límite. Pero como vemos en otros ejemplos, como $\{ww^R\}$, necesitamos más que una capacidad de conteo ilimitada: necesitamos la capacidad de almacenar y combinar una secuencia de símbolos en orden inverso. Esto sugiere que podríamos probar una pila como mecanismo de almacenamiento, lo que permite un almacenamiento ilimitado que se restringe a operar como una pila. Esto nos da una clase de máquinas llamadas **autómatas de pila (pda)**.

En este capítulo, exploramos la conexión entre los autómatas de pila y los lenguajes libres de contexto. Primero mostramos que si permitimos que los autómatas de pila actúen de manera no determinista, obtenemos una clase de autómatas que acepta exactamente la familia de lenguajes libres de contexto.

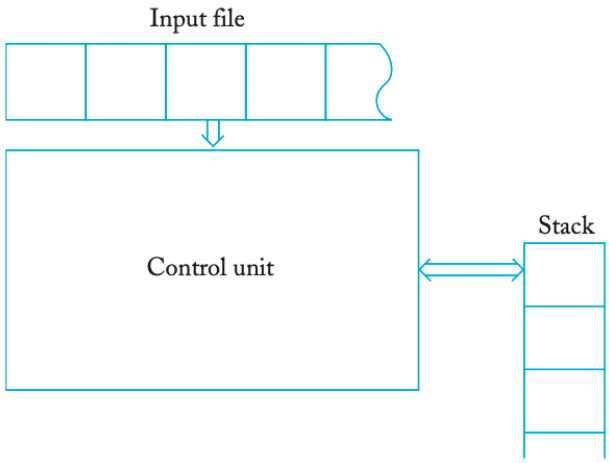


Figura 7.1: Representación esquemática de un autómata de pila.

7.1. Autómatas de pila no deterministas

En la Figura 7.1 se da una representación esquemática de un autómata de pila. Cada movimiento de la unidad de control lee un símbolo del archivo de entrada, mientras que al mismo tiempo cambia el contenido de la pila a través de las operaciones de pila habituales. Cada movimiento de la unidad de control está determinado por el símbolo de entrada actual, así como por el símbolo actualmente en la parte superior de la pila. El resultado del movimiento es un nuevo estado de la unidad de control y un cambio en la parte superior de la pila.

7.1.1. Definición de un autómata de pila

Formalizar esta noción intuitiva nos da una definición precisa de un autómata de pila.

Definición 7.1 Un **aceptador de pila no determinista (npda)** se define por la septupla

$$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F),$$

donde

Q es un conjunto de estados internos de la unidad de control,

Σ es el alfabeto de entrada,

Γ es un conjunto finito de de símbolos llamado el **alfabeto de la pila**,

$\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to R \subseteq (Q \times \Gamma^*)$ es la función de transición, donde R debe ser un conjunto finito,

$q_0 \in Q$ es el estado inicial de la unidad de control,

$z \in \Gamma$ es el **símbolo inicial de la pila**,

F $\subseteq Q$ es el conjunto de estados finales.

La complicada apariencia formal del dominio y rango de δ merece un examen más detenido. Los argumentos de δ son el estado actual de la unidad de control, el símbolo de entrada actual y el símbolo actual en la parte superior de la pila. El resultado es un conjunto de pares (q, x), donde q es el siguiente estado de la unidad de control y x es una cadena que se coloca en la parte superior de la pila en lugar del único símbolo anterior.

Tenga en cuenta que el segundo argumento de δ puede ser λ, lo que indica que es posible un movimiento que no consume un símbolo de entrada. Llamaremos a tal movimiento una transición λ. Tenga en cuenta también que δ se define de modo que necesita un símbolo de pila; ningún movimiento es posible si la pila está vacía.

Finalmente, el requisito de que los elementos del rango de δ sean un subconjunto finito es necesario porque $Q \times \Gamma^*$ es un conjunto infinito y por lo tanto tiene infinitos subconjuntos. Si bien un npda puede tener varias opciones para sus movimientos, esta elección debe estar restringida a un conjunto finito de posibilidades.

Ejemplo 7.1 Suponga que el conjunto de reglas de transición de un npda contiene:

$$\delta(q_1, a, b) = \{(q_2, cd), (q_3, \lambda)\}.$$

Si en algún momento el autómata entra al estado q_1, lee el símbolo de entrada a y el símbolo en el tope de la pila es b, entonces ocurren dos cosas:

1. el autómata entra al estado q_2 y la cadena cd reemplaza a b en el tope de la pila, es decir, b se elimina del tope de la pila y se empila la cadena cd.

2. el autómata entra al estado q_3 y la cadena λ reemplaza a b en el tope de la pila, es decir, b se elimina del tope de la pila y ninguna cadena se empila.

En nuestra notación asumimos que insertar una cadena en la pila se hace símbolo por símbolo, empezando por la parte más derecha de la cadena.

□

Ejemplo 7.2 Considere el siguiente npda:

$$\begin{aligned} Q &= \{q_0, q_1, q_2, q_3\}, \\ \Sigma &= \{a, b\}, \\ \Gamma &= \{0, 1\}, \\ z &= 0, \\ F &= \{q_3\}. \end{aligned}$$

con estado inicial q_0 y

$$\begin{aligned} &\delta(q_0, a, 0) = \{(q_1, 10), (q_3, \lambda)\}, && \delta(q_1, b, 1) = \{(q_2, \lambda)\}, \\ &\delta(q_0, \lambda, 0) = \{(q_3, \lambda)\}, && \delta(q_2, b, 1) = \{(q_2, \lambda)\}, \\ &\delta(q_1, a, 1) = \{(q_1, 11)\}, && \delta(q_2, \lambda, 0) = \{(q_3, \lambda)\}. \end{aligned}$$

¿Qué podemos decir de la acción de este autómata?

Primero, observe que las transiciones no se especifican para todas las combinaciones posibles de símbolos de entrada y de pila. Por ejemplo, no se da ninguna entrada para $\delta(q_0, b, 0)$. La interpretación de esto es la misma que usamos para los autómatas finitos no deterministas: una transición no especificada es el conjunto vacío y representa una configuración muerta para el npda. Las transiciones cruciales son

$$\delta(q_1, a, 1) = \{(q_1, 11)\}$$

que agrega un 1 a la pila cuando se lee una a, y

$$\delta(q_2, b, 1) = \{(q_2, \lambda)\}$$

que elimina un 1 cuando se encuentra una b. Estos dos pasos cuentan el número de a que empatan con el número de b. La unidad de control está en el estado q_1 hasta que se encuentra la primera b, en cuyo momento pasa

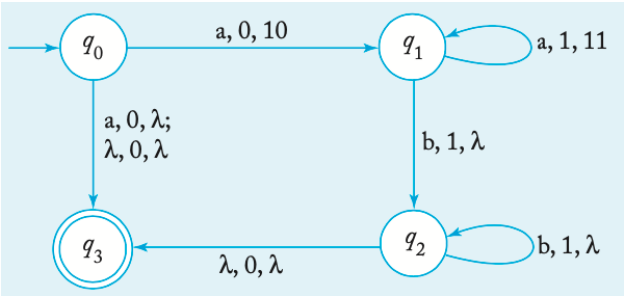


Figura 7.2: Grafo de transiciones para el npda del Ejemplo 7.2.

al estado q_2. Esto asegura que ninguna b precede a la última a. Después de analizar las transiciones restantes, vemos que el npda terminará en el estado final q_3 si y sólo si la cadena de entrada está en el lenguaje

$$L = \{a^n b^n : n \geq 0\} \cup \{a\}.$$

Como analogía con los autómatas finitos, podríamos decir que el npda acepta el lenguaje anterior. Por supuesto, antes de hacer tal afirmación, debemos definir qué entendemos por un npda que acepta un lenguaje.

□

También podemos usar grafos de transiciones para representar los npda. En esta representación, etiquetamos las aristas del grafo con tres cosas: el símbolo de entrada actual, el símbolo en la parte superior de la pila y la cadena que reemplaza la parte superior de la pila.

Ejemplo 7.3 El npda del Ejemplo 7.2 está representado por el grafo de transiciones de la Figura 7.2. □

Si bien los grafos de transiciones son convenientes para describir los npda, son menos útiles para generar argumentos. El hecho de que tengamos que realizar un seguimiento, no solo de los estados internos, sino también del contenido de la pila, limita la utilidad de los grafos de transiciones para el razonamiento formal.

En cambio, presentamos una notación sucinta para describir las configuraciones sucesivas de un npda durante el procesamiento de una cadena. Los factores relevantes en cualquier momento son el estado actual de la unidad de control, la parte no leída de la cadena de entrada y el contenido actual de

la pila. Juntos, estos determinan por completo todas las formas posibles en que puede proceder el npda. Al triple

$$(q, w, u),$$

donde q es el estado en el que se encuentra el autómata, w es la parte que no se ha leído de la cadena de entrada y u es el contenido de la pila (el símbolo más a la izquierda de u está en el tope de la pila), se le llama **descripción instantánea** del autómata de pila.

Un movimiento en un paso de una descripción intantánea a otra se denota mediante el símbolo $\vdash$; así

$$(q_1, aw, bx) \vdash (q_2, w, yx)$$

es posible si y sólo si

$$(q_2, y) \in \delta(q_1, a, b).$$

Movimientos que toman un número arbitrario de pasos se denotan mediante $\overset{*}{\vdash}$. Así, la expresión

$$(q_1, w_1, x_1) \overset{*}{\vdash} (q_2, w_2, x_2)$$

indica un posible cambio de configuración en cierto número de pasos.

En ocasiones en las que se están considerando varios autómatas, usaremos $\vdash_M$ para enfatizar que el movimiento lo realiza el autómata M.

7.1.2. El lenguaje aceptado por un autómata de pila

Definición 7.2 Sea $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ un autómata de pila no determinista. El lenguaje aceptado por M es el conjunto

$$L(M) = \Big\{ w \in \Sigma^* : (q_0, w, z) \overset{*}{\vdash}_M (p, \lambda, u), p \in F, u \in \Gamma^* \Big\}.$$

En palabras, el lenguaje aceptado por un npda es el conjunto de palabras que al ser leídas por el autómata empezando en el estado inicial lo llevan a un estado final, sin importar lo que quede en la pila.

Ejemplo 7.4 Construya un npda que acepte el lenguaje

$$L = \Big\{ w \in \{a, b\}^* : n_a(w) = n_b(w) \Big\}.$$

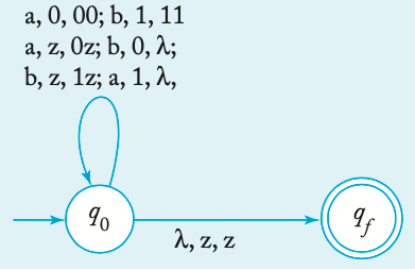


Figura 7.3: Grafo de transiciones para el npda del Ejemplo 7.4.

Como en el Ejemplo 7.2, la solución a este problema implica contar el número de a y b, lo cual se hace fácilmente con una pila. Aquí ni siquiera tenemos que preocuparnos por el orden de las a y las b.

Podemos insertar un símbolo contador, digamos 0, en la pila cada vez que se lee una a, luego sacar un símbolo contador de la pila cuando se encuentra una b. La única dificultad con esto es que si hay un prefijo de w con más b que a, no encontraremos un 0 en el tope de la pila. Pero esto es fácil de arreglar; podemos usar un símbolo contador negativo, digamos 1, para contar las b que se compararán con las a más tarde. La solución completa se da en el grafo de transiciones de la Figura 7.3.

□

Ejemplo 7.5 Construya un npda que acepte el lenguaje:

$$L = \left\{ ww^R : w \in \{a, b\}^+ \right\},$$

usamos el hecho de que los símbolos se recuperan de una pila en el orden inverso al de su inserción. Al leer la primera parte de la cadena, empilamos símbolos consecutivos en la pila.

Para la segunda parte, comparamos el símbolo de entrada actual con la parte superior de la pila, continuando mientras los dos coincidan. Dado que los símbolos se recuperan de la pila en orden inverso al que se insertaron, se logrará una coincidencia completa si y sólo si la entrada tiene la forma ww^R.

Una dificultad aparente con esta sugerencia es que no conocemos la mitad de la cadena, es decir, dónde termina w y comienza w^R. Pero la naturaleza no determinista del autómata nos ayuda con esto; el npda adivina correctamente dónde está el medio y cambia de estado en ese punto. Una solución al

problema viene dada por $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$, donde

$$\begin{aligned} Q &= \{q_0, q_1, q_2\}, \\ \Sigma &= \{a, b\}, \\ \Gamma &= \{a, b, z\}, \\ F &= \{q_2\}. \end{aligned}$$

La función de transición se puede visualizar como si tuviera varias partes: un conjunto para empilar w en la pila empezando en q_0,

$$\begin{aligned} \delta(q_0, a, a) &= \{(q_0, aa)\}, & \delta(q_0, b, a) &= \{(q_0, ba)\}, \\ \delta(q_0, a, b) &= \{(q_0, ab)\}, & \delta(q_0, b, b) &= \{(q_0, bb)\}, \\ \delta(q_0, a, z) &= \{(q_0, az)\}, & \delta(q_0, b, z) &= \{(q_0, bz)\}. \end{aligned}$$

un conjunto para adivinar la mitad de la cadena, donde el npda cambia del estado q_0 a q_1,

$$\delta(q_0, \lambda, a) = \{(q_1, a)\}, \qquad \delta(q_0, \lambda, b) = \{(q_1, b)\}.$$

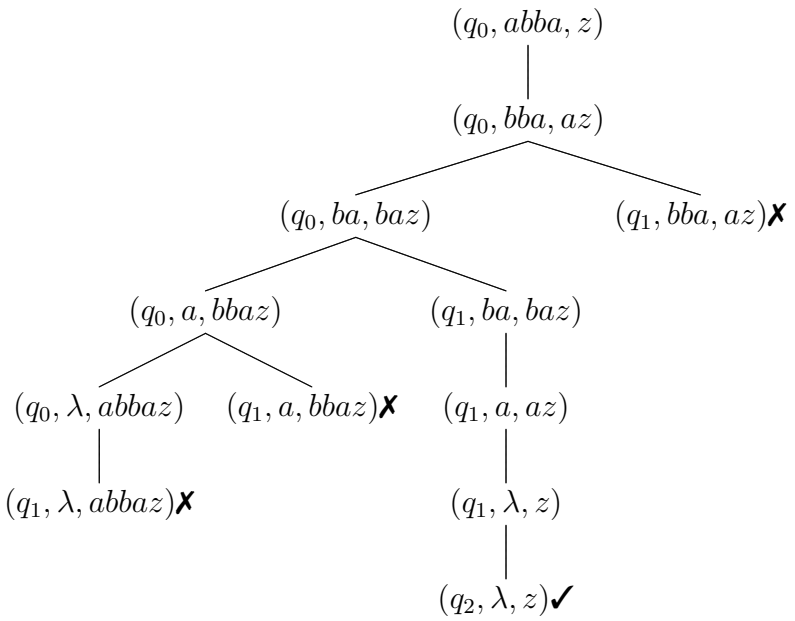
Un conjunto que empata w^R con el contenido de la pila:

$$\delta(q_1, a, a) = \{(q_1, \lambda)\}, \qquad \delta(q_1, b, b) = \{(q_1, \lambda)\}.$$

Finalmente, la transición que reconoce un empate exitoso:

$$\delta(q_1, \lambda, z) = \{(q_2, z)\}.$$

El siguiente árbol describe los movimientos que realiza el npda para reconocer la palabra *abba*, note que sólo tiene éxito la rama que adivina cual es la mitad de la cadena. Las bifurcaciones del árbol representan los movimientos no deterministas del npda.



□

7.1.3. Ejercicios

Construya npda que acepten los siguientes lenguajes:

1. $L = \{a^n b^{3n} : n \geq 0\}$.
2. $L = \{wcw^R : w \in \{a, b\}^*\}$.
3. $L = \{a^n b^m c^{n+m} : n \geq 0, m \geq 0\}$.
4. $L = \{a^n b^{n+m} c^m : n \geq 0, m \geq 1\}$.
5. $L = \{a^3 b^n c^n : n \geq 0\}$.
6. $L = \{a^n b^m : n \leq m \leq 3n\}$.
7. $L = \{w : n_a(w) = n_b(w) + 1\}$.
8. $L = \{w : n_a(w) = 2n_b(w)\}$.
9. $L = \{w : n_a(w) + n_b(w) = n_c(w)\}$.
10. $L = \{w : 2n_a(w) \leq n_b(w) \leq 3n_a(w)\}$.

11. $L = \{w : n_a(w) < n_b(w)\}$.

Capítulo 8

Máquinas de Turing

En nuestra discusión hasta ahora hemos encontrado algunas ideas fundamentales, en particular los conceptos de lenguajes regulares y libres de contexto y su asociación con autómatas finitos y aceptadores de pila. Nuestro estudio ha revelado que los lenguajes regulares forman un subconjunto propio de los lenguajes libres de contexto y, por lo tanto, que los autómatas de pila son más poderosos que los autómatas finitos.

También vimos que los lenguajes libres de contexto, si bien son fundamentales para el estudio de los lenguajes de programación, tienen un alcance limitado. Esto quedó claro en el último capítulo, donde nuestros resultados mostraron que algunos lenguajes simples, como $\{a^n b^n c^n\}$ y $\{ww\}$, no son libres de contexto.

Esto nos impulsa a mirar más allá de los lenguajes libres de contexto e investigar cómo se podrían definir nuevas familias de lenguajes que incluyan estos ejemplos. Para ello, volvemos al cuadro general de un autómata. Si comparamos autómatas finitos con autómatas de pila, vemos que la naturaleza del almacenamiento temporal crea la diferencia entre ellos.

Si no hay almacenamiento, tenemos un autómata finito; si el almacenamiento es una pila, tenemos el autómata de pila más poderoso. Extrapolando a partir de esta observación, podemos esperar descubrir familias de lenguajes aún más poderosas si le damos al autómata un almacenamiento más flexible. Por ejemplo, ¿qué sucedería si, en el esquema general de la Figura 1.1, usáramos dos pilas, tres pilas, una cola o algún otro dispositivo de almacenamiento? ¿Cada dispositivo de almacenamiento define un nuevo tipo de autómata y, a través de él, una nueva familia de lenguajes?

Este enfoque plantea una gran cantidad de preguntas, la mayoría de las

cuales resultan ser poco interesantes. Es más instructivo hacer una pregunta más ambiciosa y considerar hasta dónde se puede llevar el concepto de autómata. ¿Qué podemos decir sobre los autómatas más poderosos y los límites de la computación? Esto lleva al concepto fundamental de una **máquina de Turing** y, a su vez, a una definición precisa de la idea de un cómputo mecánico o algorítmico.

Comenzamos nuestro estudio con una definición formal de una máquina de Turing, luego desarrollamos una idea de lo que implica haciendo algunos programas simples. A continuación argumentamos que, si bien el mecanismo de una máquina de Turing es bastante rudimentario, el concepto es lo suficientemente amplio como para abarcar procesos muy complejos. La discusión culmina con la tesis de Turing, que sostiene que cualquier proceso computacional, como los que llevan a cabo las computadoras actuales, puede realizarse en una máquina de Turing.

8.1. La máquina de Turing estándar

Una máquina de Turing es un autómata cuyo almacenamiento temporal es una cinta. Esta cinta está dividida en celdas, cada una de las cuales puede contener un símbolo. Asociado con la cinta hay una cabeza de lectura y escritura que puede viajar hacia la derecha o hacia la izquierda en la cinta y que puede leer y escribir un solo símbolo en cada movimiento.

Para desviarnos un poco del esquema general del Capítulo 1, el autómata que usamos como máquina de Turing no tendrá un archivo de entrada ni ningún mecanismo especial de salida. Cualquier entrada y salida que sea necesaria se hará en la cinta de la máquina. Veremos más adelante que esta modificación de nuestro modelo general en la Sección 1.3 es de poca importancia.

Podríamos retener el archivo de entrada y un mecanismo de salida específico sin afectar ninguna de las conclusiones que estamos a punto de sacar, pero los omitimos porque el autómata resultante es un poco más fácil de describir.

En la figura 9.1 se muestra un diagrama que ofrece una visualización intuitiva de una máquina de Turing. La definición 9.1 precisa la noción.

Definición 8.1 Una máquina de Turing se define por la septupla

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F),$$

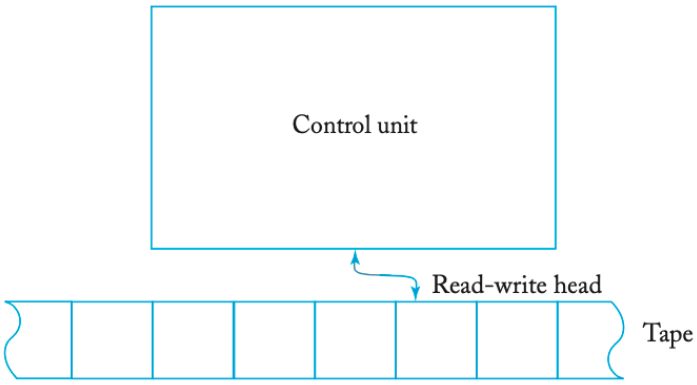


Figura 8.1: Diagrama de una máquina de Turing.

donde

Q es el conjunto de estados internos de la unidad de control,

Σ es el alfabeto de entrada,

Γ es un conjunto finito de de símbolos llamado el **alfabeto de la cinta**,

$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ es la función de transición,

$q_0 \in Q$ es el estado inicial de la unidad de control,

$\square \in \Gamma$ es un símbolo especial llamado **blanco**,

$F \subseteq Q$ es el conjunto de estados finales.

En la definición de una máquina de Turing, asumimos que $\Sigma \subseteq \Gamma - \{\square\}$, es decir, que el alfabeto de entrada es un subconjunto del alfabeto de cinta, sin incluir el espacio en blanco. Los espacios en blanco se descartan como entrada por razones que se harán evidentes en breve.

En general, δ es una función parcial sobre $Q \times \Gamma$; su interpretación da el principio por el cual opera una máquina de Turing. Los argumentos de δ son el estado actual de la unidad de control y el símbolo de cinta actual que se lee.

El resultado es un nuevo estado de la unidad de control, un símbolo nuevo de cinta, que reemplaza al antiguo, y un símbolo de movimiento, L o R. El

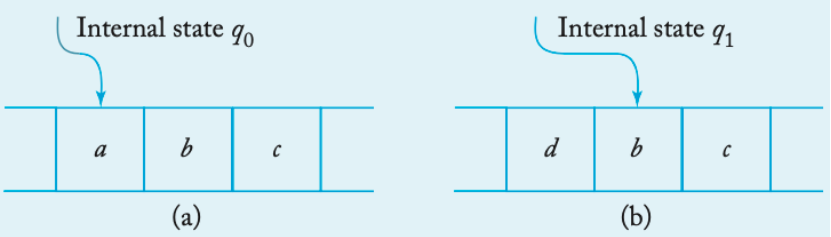


Figura 8.2: La situación (a) antes del movimiento y (b) después del movimiento.

símbolo de movimiento indica si la cabeza de lectura y escritura se mueve hacia la izquierda o hacia la derecha una celda, después de que el símbolo nuevo ha sido escrito en la cinta.

Ejemplo 8.1 La Figura 8.2 muestra la situación antes y después del movimiento

$$\delta(q_0, a) = (q_1, d, R)$$

□

Podemos pensar en una máquina de Turing como una computadora bastante simple. Tiene una unidad de procesamiento, que tiene una memoria finita, y en su cinta tiene un almacenamiento secundario de capacidad ilimitada. Las instrucciones que puede llevar a cabo una computadora de este tipo son muy limitadas: puede detectar un símbolo en su cinta y usar el resultado para decidir qué hacer a continuación.

Las únicas acciones que puede realizar la máquina son reescribir el símbolo actual, cambiar el estado del control y mover la cabeza de lectura y escritura. Este pequeño conjunto de instrucciones puede parecer inadecuado para hacer cosas complicadas, pero no es así. Las máquinas de Turing son bastante poderosas en principio. La función de transición δ define cómo actúa esta computadora, y a menudo la llamamos el "programa" de la máquina.

Como siempre, el autómata comienza en el estado inicial dado con alguna información en la cinta. Luego pasa por una secuencia de pasos controlados por la función de transición δ. Durante este proceso, el contenido de cualquier celda de la cinta se puede examinar y cambiar muchas veces. Eventualmente, todo el proceso puede terminar, lo que logramos en una máquina de Turing al ponerla en un **estado de paro**.

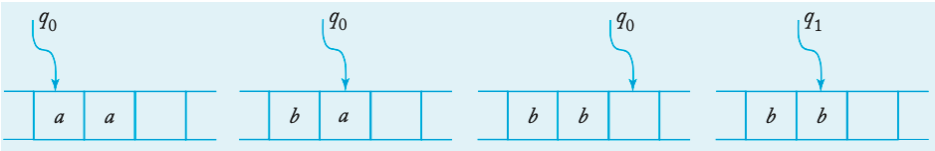


Figura 8.3: Una secuencia de movimientos para el Ejemplo 8.2.

Se dice que una máquina de Turing se detiene cada vez que alcanza una configuración para la que δ no está definido; esto es posible porque δ es una función parcial. De hecho, supondremos que no se definen transiciones para ningún estado final, por lo que la máquina de Turing se detendrá cada vez que entre en un estado final.

Ejemplo 8.2 Considere la máquina de Turing definida mediante:

$$\begin{aligned} Q &= \{q_0, q_1\}, & \delta(q_0, a) &= (q_0, b, R), \\ \Sigma &= \{a, b\}, & \delta(q_0, b) &= (q_0, b, R), \\ \Gamma &= \{a, b, \square\}, & \delta(q_0, \square) &= (q_1, \square, L). \\ F &= \{q_1\}, \end{aligned}$$

Si esta máquina de Turing se inicia en el estado q_0 con el símbolo a debajo de la cabeza de lectura y escritura, la regla de transición aplicable es $\delta(q_0, a) = (q_0, b, R)$. Por lo tanto, la cabeza de lectura y escritura reemplazará la a con una b, luego se moverá hacia la derecha en la cinta. La máquina permanecerá en el estado q_0.

Cualquier a posterior también se reemplazará con una b, pero las b no se modificarán. Cuando la máquina encuentra el primer espacio en blanco, se moverá una celda hacia la izquierda y luego se detendrá en el estado final q_1.

La Figura 8.3 muestra varias etapas del proceso para una configuración inicial simple.

□

Como antes, podemos usar grafos de transiciones para representar máquinas de Turing. Ahora etiquetamos las aristas del grafo con tres elementos: el símbolo de cinta actual, el símbolo que lo reemplaza y la dirección en la que se debe mover la cabeza de lectura y escritura. La máquina de Turing del Ejemplo 8.2 está representada por el grafo de transiciones de la Figura 8.4.

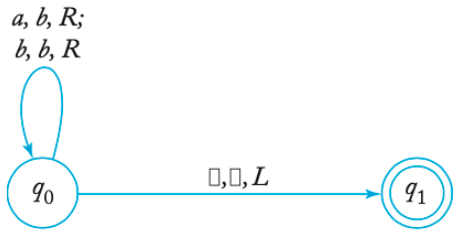


Figura 8.4: Grafo de transiciones para la máquina de Turing del Ejemplo 8.2.

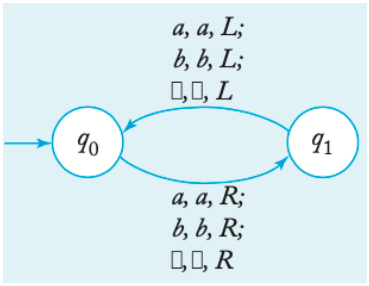


Figura 8.5: Grafo de transiciones para la máquina de Turing del Ejemplo 8.3.

Ejemplo 8.3 Mire la máquina de Turing en la Figura 8.5. Para ver qué sucederá, podemos trazar un caso típico. Suponga que la cinta inicialmente contiene$ab\cdots$, con la cabeza de lectura y escritura en a.

La máquina luego lee la a, pero no la cambia. Su siguiente estado es q_1 y la cabeza de lectura y escritura se mueve hacia la derecha, de modo que ahora está sobre b. Este símbolo también se lee y se deja sin cambios. La máquina vuelve al estado q_0 y la cabeza de lectura y escritura se mueve hacia la izquierda.

Ahora estamos exactamente en el estado original y la secuencia de movimientos comienza de nuevo. De esto queda claro que la máquina, cualquiera que sea la información inicial en su cinta, funcionará para siempre, con la cabeza de lectura y escritura moviéndose alternativamente a la derecha y luego a la izquierda, pero sin modificar la cinta.

Este es un ejemplo de una máquina de Turing que no se detiene. En analogía con la terminología de programación, decimos que la máquina de Turing está en un **ciclo infinito**.

□

Dado que se pueden hacer varias definiciones diferentes de una máquina

de Turing, vale la pena resumir las características principales de nuestro modelo, al que llamaremos **máquina de Turing estándar**:

1. La máquina de Turing tiene una cinta ilimitada en ambas direcciones, lo que permite cualquier número de movimientos hacia la izquierda y hacia la derecha.

2. La máquina de Turing es determinista en el sentido de que δ define como máximo un movimiento para cada configuración.

3. No hay un archivo de entrada especial. Suponemos que en el momento inicial la cinta tiene algún contenido específico. Algo de esto puede considerarse entrada. Del mismo modo, no hay ningún dispositivo de salida especial. Cada vez que la máquina se detiene, algunos o todos los contenidos de la cinta pueden verse como salida.

Estas convenciones se eligieron principalmente por la conveniencia de la discusión posterior. En el Capítulo 9, veremos otras versiones de máquinas de Turing y discutiremos su relación con nuestro modelo estándar.

Aquí, como en el caso de los pda, la forma más conveniente de exhibir una secuencia de configuraciones de una máquina de Turing utiliza la idea de una descripción instantánea. Cualquier configuración está completamente determinada por el estado actual de la unidad de control, el contenido de la cinta y la posición de la cabeza de lectura y escritura. Usaremos la notación en la que

$$x_1 q x_2$$

o

$$a_1 a_2 \cdots a_{k-1} q a_k a_{k+1} \cdots a_n$$

es la descripción instantánea de una máquina en el estado q con la cinta que se muestra en la Figura 8.6. Los símbolos $a_1, \ldots, a_n$ muestran el contenido de la cinta, mientras que q define el estado de la unidad de control. Esta convención se elige de modo que la posición de la cabeza de lectura y escritura esté sobre la celda que contiene el símbolo inmediatamente después de q.

La descripción instantánea brinda sólo una cantidad finita de información a la derecha y a la izquierda de la cabeza de lectura y escritura. Se supone que la parte no especificada de la cinta contiene todos los espacios en blanco; normalmente tales espacios en blanco son irrelevantes y no se muestran explícitamente en la descripción instantánea.

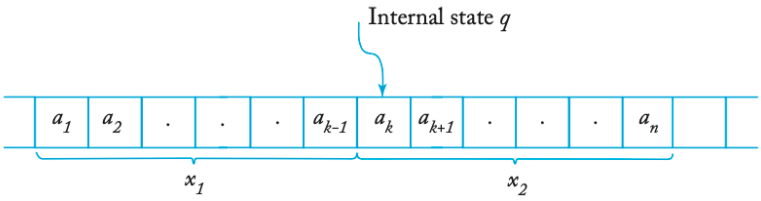


Figura 8.6: Configuración de una máquina de Turing con descripción instantánea $a_1a_2 \cdots a_{k-1}qa_ka_{k+1} \cdots a_n$.

Sin embargo, si la posición de los espacios en blanco es relevante para la discusión, el símbolo en blanco puede aparecer en la descripción instantánea. Por ejemplo, la descripción instantánea $q\Box w$ indica que la cabeza de lectura y escritura está en la celda inmediatamente a la izquierda del primer símbolo de w y que esta celda contiene un espacio en blanco.

Ejemplo 8.4 Las imágenes dibujadas en la Figura 8.3 corresponden a la secuencia de descripciones instantáneas q_0aa, bq_0a, $bbq_0\Box$, bq_1b.

□

Un movimiento de una configuración a otra se denotará con $\vdash$. Así, si

$$\delta(q_1, c) = (q_2, e, R),$$

entonces el movimiento

$$abq_1cd \vdash abeq_2d$$

se realiza siempre que el estado interno sea q_1, la cinta contenga $abcd$ y la cabeza de lectura y escritura esté en c. El símbolo $\overset{*}{\vdash}$ tiene el significado habitual de un número arbitrario de movimientos. Los subíndices, como $\vdash_M$, se usan en argumentos para distinguir entre varias máquinas.

Ejemplo 8.5 La acción de la máquina de Turing en la Figura 8.3 se puede representar por

$$q_0aa \vdash bq_0a \vdash bbq_0\Box \vdash bq_1b$$

o

$$q_0aa \overset{*}{\vdash} bq_1b$$

□

Para una mayor discusión, es conveniente resumir las diversas observaciones que acabamos de hacer de manera formal.

Definición 8.2 Sea $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ una máquina de Turing. Entonces cualquier cadena $a_1 \cdots a_{k-1} q_1 a_k a_{k+1} \cdots a_n$, con $a_i \in \Gamma$ y $q_1 \in Q$, es una descripción instantánea de M. Un movimiento

$$a_1 a_2 \cdots a_{k-1} q a_k a_{k+1} \cdots a_n \vdash a_1 a_2 \cdots a_{k-1} a'_k q' a_{k+1} \cdots a_n$$

es posible si y sólo si

$$\delta(q, a_k) = (q', a'_k, R)$$

Un movimiento

$$a_1 a_2 \cdots a_{k-1} q a_k a_{k+1} \cdots a_n \vdash a_1 a_2 \cdots q' a_{k-1} a'_k a_{k+1} \cdots a_n$$

es posible si y sólo si

$$\delta(q, a_k) = (q', a'_k, L).$$

Se dice que una máquina de Turing M **para** iniciando en alguna configuración inicial $x_1 q_i x_2$ si

$$x_1 q_i x_2 \overset{*}{\vdash} y_1 q_j a y_2$$

para cualquier q_j y a para los que $\delta(q_j, a)$ está indefinida. A la secuencia de configuraciones que llevan a la máquina M a un estado de paro se le llama **cálculo**.

El ejemplo 8.3 muestra la posibilidad de que una máquina de Turing nunca se detenga, avanzando en un ciclo sin fin del que no puede escapar. Esta situación juega un papel fundamental en la discusión de las máquinas de Turing, por lo que utilizamos una notación especial para ello. Lo representaremos por

$$x_1 q x_2 \overset{*}{\vdash} \infty$$

indicando que, a partir de la configuración inicial $x_1 q x_2$, la máquina entra en un ciclo y nunca se detiene.

8.1.1. Máquinas de Turing como aceptadoras de lenguajes

Las máquinas de Turing pueden verse como aceptadoras en el siguiente sentido. Se escribe una cadena w en la cinta, con espacios en blanco que completan las partes no utilizadas. La máquina se pone en marcha en el estado inicial q_0 con la cabeza de lectura y escritura colocada en el símbolo más a la izquierda de w. Si, después de una secuencia de movimientos, la máquina de Turing entra en un estado final y se detiene, entonces se considera que w es aceptada.

Definición 8.3 Sea $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ una máquina de Turing. Entonces el lenguaje aceptado por M es

$$L(M) = \left\{ w \in \Sigma^+ : q_0 w \overset{*}{\vdash} x_1 q_f x_2, q_f \in F, x_1, x_2 \in \Gamma^* \right\}.$$

Esta definición indica que la entrada w está escrita en la cinta con espacios en blanco a ambos lados. La razón para excluir espacios en blanco de la entrada ahora se vuelve clara: nos asegura que toda la entrada está restringida a una región bien definida de la cinta, delimitada por espacios en blanco a la derecha y a la izquierda.

Sin esta convención, la máquina no podría limitar la región en la que debe buscar la entrada; sin importar cuántos espacios en blanco viera, nunca podría estar seguro de que no hubiera alguna entrada que no estuviera en blanco en algún otro lugar de la cinta.

La Definición 8.3 nos dice qué debe suceder cuando $w \in L(M)$. No dice nada sobre el resultado de cualquier otra entrada. Cuando w no está en $L(M)$, puede suceder una de dos cosas: la máquina puede detenerse en un estado no final o puede entrar en un ciclo infinito y nunca detenerse. Cualquier cadena para la cual M no se detiene, por definición, no está en $L(M)$.

Ejemplo 8.6 Para $\Sigma = \{0, 1\}$ diseñe una máquina de Turing que acepte el lenguaje denotado por la expresión regular 00^*.

Este es un ejercicio sencillo de programación de máquinas de Turing. Comenzando en el extremo izquierdo de la entrada, leemos cada símbolo y verificamos que sea un 0. Si lo es, continuamos moviéndolo hacia la derecha.

Si llegamos a un espacio en blanco sin encontrar nada más que 0, terminamos y aceptamos la cadena.

Si la entrada contiene un 1 en cualquier parte, la cadena no está en $L(00^*)$, y nos detenemos en un estado no final. Para realizar un seguimiento del cálculo, son suficientes dos estados internos $Q = \{q_0, q_1\}$ y un estado final $F = \{q_1\}$. Como función de transición podemos tomar

$$\delta(q_0, 0) = (q_0, 0, R),$$
$$\delta(q_0, \square) = (q_1, \square, R).$$

Siempre que aparezca un 0 debajo de la cabeza de lectura y escritura, la cabeza se moverá hacia la derecha. Si en algún momento se lee un 1, la máquina se detendrá en el estado no final q_0, ya que $\delta(q_0, 1)$ no está definida.

Tenga en cuenta que la máquina de Turing también se detiene en un estado final si se inicia en el estado q_0 en un espacio en blanco. Podríamos interpretar esto como la aceptación de λ, pero por razones técnicas la cadena vacía no está incluida en la Definición 8.3.

□

El reconocimiento de lenguajes más complicados es más difícil. Dado que las máquinas de Turing tienen un conjunto de instrucciones primitivo, los cálculos que podemos programar fácilmente en un lenguaje de nivel superior suelen ser engorrosos en una máquina de Turing. Aún así, es posible, y el concepto es fácil de entender, como ilustran los siguientes ejemplos.

Ejemplo 8.7 Para $\Sigma = \{a, b\}$ diseñe una máquina de Turing que acepte

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

Intuitivamente, resolvemos el problema de la siguiente manera. Comenzando en la a más a la izquierda, la marcamos reemplazándola con algún símbolo, digamos x. Luego dejamos que la cabeza de lectura y escritura viaje hacia la derecha para encontrar la b más a la izquierda, que a su vez se marca reemplazándola con otro símbolo, digamos y.

Después de eso, volvemos a la izquierda hacia la a más a la izquierda, la reemplazamos con una x, luego nos movemos hacia la b más a la izquierda y la reemplazamos con y, y así sucesivamente. Viajando de un lado a otro de esta manera, empatamos cada a con una b correspondiente. Si después de algún tiempo no quedan a o b, entonces la cadena debe estar en L.

Trabajando los detalles, llegamos a una solución completa para la cual $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $F = \{q_4\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b, x, y, \square\}$. Las transiciones se pueden dividir en varias partes. El conjunto

$$\begin{aligned}\delta(q_0, a) &= (q_1, x, R),\\ \delta(q_1, a) &= (q_1, a, R),\\ \delta(q_1, y) &= (q_1, y, R),\\ \delta(q_1, b) &= (q_2, y, L),\end{aligned}$$

reemplaza la a más a la izquierda con una x, luego hace que la cabeza de lectura y escritura viaje hacia la derecha hasta la primera b, reemplazándola con una y. Cuando se escribe y, la máquina entra en un estado q_2, lo que indica que una a se ha emparejado con éxito con una b.

El siguiente conjunto de transiciones invierte la dirección hasta que se encuentra una x, reposiciona la cabeza de lectura y escritura sobre la a más a la izquierda y devuelve el control al estado inicial.

$$\begin{aligned}\delta(q_2, y) &= (q_2, y, L),\\ \delta(q_2, a) &= (q_2, a, L),\\ \delta(q_2, x) &= (q_0, x, R),\end{aligned}$$

Después de pasar por esta parte del cómputo, la máquina habrá realizado el cómputo parcial.

$$q_0 aa \cdots abb \cdots b \overset{*}{\vdash} xq_0 a \cdots ayb \cdots b,$$

de modo que una sola a se ha emparejado con una sola b. Después de dos pasadas, habremos completado el cómputo parcial.

$$q_0 aa \cdots abb \cdots b \overset{*}{\vdash} xxq_0 \cdots ayy \cdots b,$$

y así sucesivamente, indicando que el proceso de emparejamiento se está realizando correctamente. Cuando la entrada es una cadena $a^n b^n$, la reescritura continúa de esta manera, deteniéndose sólo cuando no hay más a para borrar.

Al buscar la a más a la izquierda, la cabeza de lectura y escritura se desplaza hacia la izquierda con la máquina en el estado q_2. Cuando se encuentra una x, la dirección se invierte para obtener la a. Pero ahora, en lugar de encontrar una a, encontrará una y.

Para terminar, se realiza una verificación final para ver si todas las letras a y b han sido reemplazadas (para detectar entradas donde una a sigue a una b). Esto se puede hacer mediante

$$\begin{aligned}\delta(q_0, y) &= (q_3, y, R),\\ \delta(q_3, y) &= (q_3, y, R),\\ \delta(q_3, \square) &= (q_4, \square, R).\end{aligned}$$

Si ingresamos una cadena que no está en el lenguaje, el cálculo se detendrá en un estado no final. Por ejemplo, si le damos a la máquina una cadena $a^n b^m$, con $n > m$, la máquina eventualmente encontrará un espacio en blanco en el estado q_1. Se detendrá porque no se especifica ninguna transición para este caso. Otra entrada que no esté en el lenguaje también conducirá a un estado de detención no final.

La entrada particular *aabb* da las siguientes descripciones instantáneas sucesivas:

$$\begin{aligned}q_0aabb &\vdash xq_1abb \vdash xaq_1bb \vdash xq_2ayb\\ &\vdash q_2xayb \vdash xq_0ayb \vdash xxq_1yb\\ &\vdash xxyq_1b \vdash xxq_2yy \vdash xq_2xyy\\ &\vdash xxq_0yy \vdash xxyq_3y \vdash xxyyq_3\square\\ &\vdash xxyy\ \square q_4\square.\end{aligned}$$

En este punto, la máquina de Turing se detiene en un estado final, por lo que se acepta la cadena *aabb*.

Se le sugiere trazar este programa con varias cadenas más en L, así como con algunas que no están en L.

□

Ejemplo 8.8 Diseñe una máquina de Turing que acepte

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

Las ideas utilizadas en el Ejemplo 8.7 se trasladan fácilmente a este caso. Hacemos coincidir cada a, b y c reemplazándolos en orden por x, y y z, respectivamente. Al final, verificamos que todos los símbolos originales hayan sido reescritos.

Aunque conceptualmente es una simple extensión del ejemplo anterior, escribir el programa real es tedioso. Lo dejamos como un ejercicio algo largo, pero directo. Tenga en cuenta que aunque $\{a^n b^n\}$ es un lenguaje libre de contexto y $\{a^n b^n c^n\}$ no lo es, las máquinas de Turing pueden aceptarlos con estructuras muy similares.

□

Una conclusión que podemos sacar de este ejemplo es que una máquina de Turing puede reconocer algunos lenguajes que no son libres de contexto, una primera indicación de que las máquinas de Turing son más poderosas que los autómatas de pila.

8.1.2. Máquinas de Turing como transductores

Hemos tenido pocas razones hasta ahora para estudiar transductores; en la teoría de lenguajes, los aceptores son bastante adecuados. Pero como veremos en breve, las máquinas de Turing no sólo son interesantes como aceptadoras de lenguajes, sino que también nos proporcionan un modelo abstracto simple para las computadoras digitales en general.

Dado que el propósito principal de una computadora es transformar la entrada en salida, actúa como un transductor. Si queremos modelar computadoras usando máquinas de Turing, tenemos que mirar más de cerca este aspecto.

La entrada para un cálculo serán todos los símbolos que no estén en blanco en la cinta en el momento inicial. Al final del cálculo, la salida será lo que esté en la cinta. Por lo tanto, podemos ver una máquina de Turing transductoa M como una implementación de una función f definida por

$$\widehat{w} = f(w)$$

siempre que

$$q_0 w \overset{*}{\vdash}_M q_f \widehat{w},$$

para algún estado final q_f.

Definición 8.4 Una función f con dominio D se dice que es **Turing computable** o sólo **computable** si existe alguna máquina de Turing

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$$

tal que

$$q_0 w \overset{*}{\vdash}_M q_f f(w), q_f \in F,$$

para todo $w \in D$.

Como afirmaremos en breve, todas las funciones matemáticas comunes, sin importar cuán complicadas sean, son Turing computables. Comenzamos observando algunas operaciones simples, como la suma y la comparación aritmética.

Ejemplo 8.9 Dados dos enteros positivos x y y, diseñe una máquina de Turing que calcule $x + y$.

Primero tenemos que elegir alguna convención para representar números enteros positivos. Para simplificar, usaremos una notación unaria en la que cualquier entero positivo x se representa por $w(x) \in \{1\}^+$, tal que

$$|w(x)| = x.$$

También debemos decidir cómo se colocan inicialmente x y y en la cinta y cómo aparecerá su suma al final del cálculo. Supondremos que $w(x)$ y $w(y)$ están en la cinta en notación unaria, separados por un solo 0, con la cabeza de lectura y escritura en el símbolo más a la izquierda de $w(x)$.

Después del cálculo, $w(x+y)$ estará en la cinta seguido de un solo 0, y la cabeza de lectura y escritura se colocará en el extremo izquierdo del resultado. Por lo tanto, queremos diseñar una máquina de Turing para realizar el cálculo

$$q_0 w(x)0w(y) \overset{*}{\vdash} q_f w(x+y)0,$$

donde q_f es un estado final. Construir un programa para esto es relativamente simple. Todo lo que tenemos que hacer es mover el 0 de separación al extremo derecho de $w(y)$, de modo que la suma no sea más que la fusión de las dos cadenas. Para lograr esto, construimos $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$, con $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $F = \{q_4\}$, y

$$\begin{aligned}
\delta(q_0, 1) &= (q_0, 1, R), & \delta(q_2, 1) &= (q_3, 0, L),\\
\delta(q_0, 0) &= (q_1, 1, R), & \delta(q_3, 1) &= (q_3, 1, L),\\
\delta(q_1, 1) &= (q_1, 1, R), & \delta(q_3, \square) &= (q_4, \square, R).\\
\delta(q_1, \square) &= (q_2, \square, L), &&
\end{aligned}$$

Tenga en cuenta que al mover el 0 a la derecha creamos temporalmente un 1 adicional, un hecho que se recuerda al poner la máquina en el estado q_1. Se necesita la transición $\delta(q_2, 1) = (q_3, 0, R)$ para eliminar esto al final del cálculo. Esto se puede ver en la secuencia de descripciones instantáneas para sumar 111 a 11:

$$\begin{aligned} q_0111011 &\vdash 1q_011011 \vdash 11q_01011 \vdash 111q_0011 \\ &\vdash 1111q_111 \vdash 11111q_11 \vdash 111111q_1\square \\ &\vdash 11111q_21 \vdash 1111q_310 \vdash 111q_3110 \\ &\vdash 11q_31110 \vdash 1q_311110 \vdash q_3111110 \\ &\vdash q_3\square111110 \vdash q_4111110 = q_4w(x+y)0. \end{aligned}$$

La notación unaria, aunque engorrosa para los cálculos prácticos, es muy conveniente para programar máquinas de Turing. Los programas resultantes son mucho más cortos y sencillos que si hubiésemos usado otra representación, como binaria o decimal.

□

Sumar números es una de las operaciones fundamentales de cualquier computadora, que juega un papel en la síntesis de instrucciones más complicadas. Otras operaciones básicas son la copia de cadenas y las comparaciones simples. Estos también se pueden hacer fácilmente en una máquina de Turing.

Ejemplo 8.10 Diseñe una máquina de Turing que copie cadenas de 1s. Más precisamente, construya una máquina que realice el siguiente cálculo:

$$q_0w \overset{*}{\vdash} q_fww,$$

para cualquier $w \in \{1\}^+$.

Para resolver el problema, implementamos el siguiente proceso intuitivo:

1. Reemplaza cada 1 por una x.
2. Encuentra la x más a la derecha y reemplázala con 1.
3. Vaya al extremo derecho de la región actual que no está en blanco y cree un 1 allí.
4. Repita los pasos 2 y 3 hasta que no haya más x.

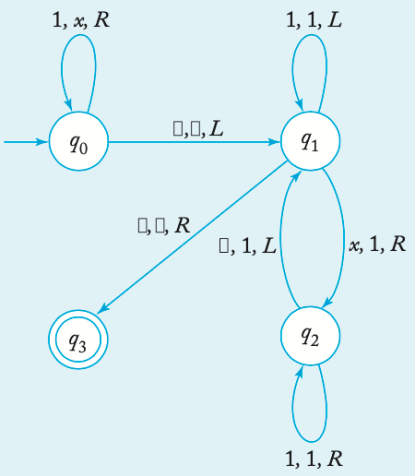


Figura 8.7: Grafo de transiciones para la máquina de Turing del Ejemplo 8.10.

La solución se muestra en el grafo de transiciones de la Figura 8.7. Y su función de transición está dada por

$$\begin{aligned} \delta(q_0, 1) &= (q_0, x, R), & \delta(q_0, \square) &= (q_1, \square, L), \\ \delta(q_1, 1) &= (q_1, 1, L), & \delta(q_1, x) &= (q_2, 1, R), \\ \delta(q_1, \square) &= (q_3, \square, R), & \delta(q_2, 1) &= (q_2, 1, R), \\ \delta(q_2, \square) &= (q_1, 1, L). \end{aligned}$$

Puede ser un poco difícil ver al principio que la solución es correcta, así que sigamos el programa con la cadena simple 11. El cálculo realizado en este caso es

$$\begin{aligned} q_0 11 &\vdash x q_0 1 \vdash x x q_0 \square \vdash x q_1 x \\ &\vdash x 1 q_2 \square \vdash x q_1 11 \vdash q_1 x 11 \\ &\vdash 1 q_2 11 \vdash 11 q_2 1 \vdash 111 q_2 \square \\ &\vdash 11 q_1 11 \vdash 1 q_1 111 \vdash q_1 1111 \\ &\vdash q_1 \square 1111 \vdash q_3 1111. \end{aligned}$$

□

Ejemplo 8.11 Sean x y y dos enteros positivos representados en notación unaria. Construya una máquina de Turing que pare en un estado final q_y si

$x \geq y$ y que pare en un estado no final q_n si $x < y$. Específicamente, una máquina que realice el siguiente cálculo:

$$q_0 w(x)0w(y) \overset{*}{\vdash} q_y w(x)0w(y) \text{ si } x \geq y,$$
$$q_0 w(x)0w(y) \overset{*}{\vdash} q_n w(x)0w(y) \text{ si } x < y.$$

Para resolver este problema, podemos usar la idea del Ejemplo 8.7 con algunas modificaciones menores. En lugar de empatar a y b, empatamos cada 1 a la izquierda del 0 divisorio con el 1 a la derecha. Al final del empate, tendremos en la cinta ya sea

$$xx \cdots 110xx \cdots x\Box$$

o

$$xx \cdots xx0xx \cdots x11\Box,$$

dependiendo de si $x > y$ o $y > x$. En el primer caso, cuando intentamos hacer coincidir otro 1, encontramos el espacio en blanco a la derecha del espacio de trabajo. Esto puede usarse como una señal para ingresar al estado q_y.

En el segundo caso, todavía encontramos un 1 a la derecha cuando todos los 1 a la izquierda han sido reemplazados. Usamos esto para entrar en el otro estado q_n. El programa completo para esto es sencillo y se deja como ejercicio.

Este ejemplo destaca el punto importante de que una máquina de Turing puede programarse para tomar decisiones basadas en comparaciones aritméticas. Este tipo de decisión simple es común en el lenguaje de máquina de las computadoras, donde se ingresan secuencias de instrucciones alternas, dependiendo del resultado de una operación aritmética.

□

8.1.3. Ejercicios

1. Construya máquinas de Turing que acepten los siguientes lenguajes sobre $\{a, b\}$, verifique sus máquinas con dos ejemplos.

   a) $L = \{w : |w| \text{ es impar}\}$.

   b) $L = \{w : |w| \text{ es múltiplo de } 4\}$.

   c) $L = \{w : n_a(w) \neq n_b(w)\}$.

d) $L = \{a^n b^n a^n b^n : n \neq 0\}$.

e) $L = \{a^n b^{2n} : n \geq 1\}$.

2. Construya la máquina de Turing especificada en el Ejemplo 8.11, verifique su máquina con dos ejemplos.

3. Construya máquinas de Turing que calculen las siguientes funciones para x y y enteros positivos representados en unario, verifique sus máquinas con dos ejemplos.

   a) $f(w) = w^R$, donde $w \in \{0, 1\}^+$.

   b) $f(x) = 2x + 1$.

   c) $f(x, y) = 2x + 3y$.

   d) $f(x) = x \mod 5$.

8.2. Tesis de Turing

La discusión anterior no solo muestra cómo se puede construir una máquina de Turing a partir de piezas más simples, sino que también ilustra un aspecto negativo de trabajar con autómatas de tan bajo nivel.

Si bien se necesita muy poca imaginación o ingenio para traducir un diagrama de bloques o un pseudocódigo en el programa de la máquina de Turing correspondiente, en realidad hacerlo lleva mucho tiempo, es propenso a errores y agrega poco a nuestra comprensión. El conjunto de instrucciones de una máquina de Turing es tan restringido que cualquier argumento, solución o prueba para un problema no trivial es bastante tedioso.

Ahora nos enfrentamos a un dilema: queremos afirmar que las máquinas de Turing pueden realizar no solo las operaciones simples para las que proporcionamos programas explícitos, sino también procesos más complejos, describibles mediante diagramas de bloques o pseudocódigo. Para defender tales afirmaciones contra el desafío, debemos mostrar los programas relevantes explícitamente.

Pero hacerlo es desagradable y distrae y debe evitarse si es posible. De alguna manera, nos gustaría encontrar una manera de llevar a cabo una discusión razonablemente rigurosa de las máquinas de Turing sin tener que escribir un código largo y de bajo nivel. Desafortunadamente, no existe una forma completamente satisfactoria de salir del aprieto; lo mejor que podemos

hacer es llegar a un compromiso razonable. Para ver cómo podemos lograr tal compromiso, pasamos a un tema un tanto filosófico.

Podemos sacar algunas conclusiones simples de los ejemplos de la sección anterior. La primera es que las máquinas de Turing parecen ser más poderosas que los autómatas de pila. En el Ejemplo 8.8, esbozamos la construcción de una máquina de Turing para un lenguaje que no es libre de contexto y para el cual, en consecuencia, no existe ningún autómata de pila. Los Ejemplos 8.9, 8.10 y 8.11 muestran que las máquinas de Turing pueden realizar algunas operaciones aritméticas simples, realizar manipulaciones de cadenas y hacer algunas comparaciones simples.

Supongamos que hiciéramos la conjetura de que, en cierto sentido, las máquinas de Turing tienen la misma potencia que una computadora digital típica. ¿Cómo podríamos defender o refutar tal hipótesis?

Para defenderlo, podríamos tomar una secuencia de problemas cada vez más difíciles y mostrar cómo los resuelve alguna máquina de Turing. También podríamos tomar el conjunto de instrucciones de lenguaje de máquina de una computadora específica y diseñar una máquina de Turing que pueda realizar todas las instrucciones del conjunto.

Sin duda, esto pondría a prueba nuestra paciencia, pero en principio debería ser posible si nuestra hipótesis es correcta. Aún así, mientras cada éxito en esta dirección fortalecería nuestra convicción de la verdad de la hipótesis, no conduciría a una prueba. La dificultad radica en el hecho de que no sabemos exactamente qué se entiende por "una computadora digital típica" y que no tenemos medios para hacer una definición precisa.

También podemos abordar el problema desde el otro lado. Podríamos tratar de encontrar algún procedimiento para el cual podamos escribir un programa de computadora, pero para el cual podamos demostrar que no puede existir una máquina de Turing. Si esto fuera posible, tendríamos una base para rechazar la hipótesis.

Pero nadie ha sido capaz todavía de producir un contraejemplo; el hecho de que todos esos intentos hayan fracasado debe tomarse como evidencia circunstancial de que no se puede hacer. Todo indica que las máquinas de Turing son, en principio, tan poderosas como cualquier computadora.

Argumentos de este tipo llevaron a A. M. Turing y otros a mediados de la década de 1930 a la célebre conjetura llamada **tesis de Turing**. Esta hipótesis establece que cualquier cálculo que pueda llevarse a cabo por medios mecánicos puede ser realizado por alguna máquina de Turing.

Esta es una declaración general, por lo que es importante tener en cuen-

ta cuál es la tesis de Turing. No es algo que se pueda probar. Para ello, tendríamos que definir con precisión el término "medios mecánicos".

Esto requeriría algún otro modelo abstracto y no nos dejaría más adelante que antes. La tesis de Turing se ve más correctamente como una definición de lo que constituye un cálculo mecánico: un cálculo es mecánico si y solo si puede ser realizado por alguna máquina de Turing.

Si tomamos esta actitud y consideramos la tesis de Turing simplemente como una definición, planteamos la cuestión de si esta definición es suficientemente amplia. ¿Es lo suficientemente amplio como para cubrir todo lo que hacemos ahora (y posiblemente podríamos hacer en el futuro) con computadoras? Un "sí" inequívoco no es posible, pero la evidencia a su favor es muy fuerte.

Algunos argumentos para aceptar la tesis de Turing como la definición de un cálculo mecánico son

1. Cualquier cosa que se pueda hacer en cualquier computadora digital existente también se puede hacer con una máquina de Turing.

2. Nadie ha sido aún capaz de sugerir un problema, solucionable por lo que intuitivamente consideramos un algoritmo, para el cual no se puede escribir un programa de máquina de Turing.

3. Se han propuesto modelos alternativos para el cálculo mecánico, pero ninguno de ellos es más potente que el modelo de la máquina de Turing.

Estos argumentos son circunstanciales y la tesis de Turing no puede ser probada por ellos. A pesar de su plausibilidad, la tesis de Turing sigue siendo una suposición. Pero ver la tesis de Turing simplemente como una definición arbitraria pasa por alto un punto importante. En cierto sentido, la tesis de Turing juega el mismo papel en la computación que las leyes básicas de la física y la química.

La física clásica, por ejemplo, se basa en gran medida en las leyes del movimiento de Newton. Aunque las llamemos leyes, no tienen necesidad lógica; más bien, son modelos plausibles que explican gran parte del mundo físico. Los aceptamos porque las conclusiones que sacamos de ellos concuerdan con nuestra experiencia y nuestras observaciones.

Tales leyes no pueden ser probadas como verdaderas, aunque posiblemente pueden ser invalidadas. Si un resultado experimental contradice una

conclusión basada en las leyes, podríamos comenzar a cuestionar su validez. Por otro lado, el fracaso repetido en invalidar una ley fortalece nuestra confianza en ella.

Esta es la situación de la tesis de Turing, por lo que tenemos alguna razón para considerarla una ley básica de la computación. Las conclusiones que sacamos de ella concuerdan con lo que sabemos sobre las computadoras reales y, hasta ahora, todos los intentos de invalidarla han fallado.

Siempre existe la posibilidad de que a alguien se le ocurra otra definición que dé cuenta de algunas situaciones sutiles que no están cubiertas por las máquinas de Turing pero que aún están dentro del alcance de nuestra noción intuitiva de computación mecánica.

En tal eventualidad, algunas de nuestras discusiones posteriores tendrían que modificarse significativamente. Sin embargo, la probabilidad de que esto suceda parece ser muy pequeña.

Habiendo aceptado la tesis de Turing, estamos en condiciones de dar una definición precisa de un algoritmo.

Definición 8.5 Un **algoritmo** para una función $f : D \to R$ es una máquina de Turing M, a la que se le da como entrada en su cinta cualquier $d \in D$ y eventualmente para con la respuesta correcta $f(d) \in R$ sobre su cinta. Específicamente, se requiere que:

$$q_0 d \overset{*}{\vdash}_M q_f f(d), q_f \in F,$$

para todo $d \in D$.

Identificar un algoritmo con un programa de máquina de Turing nos permite probar rigurosamente afirmaciones tales como "existe un algoritmo $\cdots$" o "no hay algoritmo $\cdots$ ". Sin embargo, construir explícitamente un algoritmo incluso para problemas relativamente simples es una tarea muy larga.

Para evitar perspectivas tan desagradables, podemos apelar a la tesis de Turing y afirmar que cualquier cosa que podamos hacer en cualquier computadora también se puede hacer en una máquina de Turing. En consecuencia, podríamos sustituir "programa en C" por "máquina de Turing" en la Definición 8.5.

Esto aliviaría considerablemente la carga de exhibir algoritmos. En realidad, como ya hemos hecho, daremos un paso más y aceptaremos las descripciones verbales o los diagramas de bloques como algoritmos suponiendo

que podríamos escribir un programa de máquina de Turing para ellos si nos retaran a hacerlo.

Esto simplifica mucho la discusión, pero obviamente nos deja abiertos a la crítica. Si bien el "programa en C" está bien definido, la "descripción verbal clara" no lo está, y corremos el peligro de afirmar la existencia de algoritmos inexistentes. Pero este peligro está más que compensado por el hecho de que podemos mantener la discusión simple e intuitivamente clara y que podemos dar descripciones concisas de algunos procesos bastante complejos.

El lector que tenga dudas sobre la validez de estas afirmaciones puede disiparlas escribiendo un programa adecuado en algún lenguaje de programación.

8.2.1. Ejercicios

1. En la discusión anterior, afirmamos en un momento que las máquinas de Turing parecen ser más poderosas que los autómatas de pila. Debido a que siempre se puede hacer que la cinta de una máquina de Turing se comporte como una pila, parecería que podríamos haber afirmado que las máquinas de Turing son más poderosas. ¿Qué factor importante no se tuvo en cuenta en el argumento que debe abordarse antes de que se justifique tal afirmación?

Capítulo 9

Otros modelos de Máquinas de Turing

Nuestra definición de una Máquina de Turing estándar no es la única posible; hay definiciones alternativas que podrían servir igualmente bien. Las conclusiones que podemos sacar sobre el poder de una Máquina de Turing son en gran medida independientes de la estructura específica elegida para ella.

En este capítulo examinamos algunas variaciones, mostrando que la Máquina de Turing estándar es equivalente en cierto sentido a otros modelos más complejos. Si aceptamos la tesis de Turing, esperamos que complicar la Máquina de Turing estándar dándole un dispositivo de almacenamiento más complejo no tendrá ningún efecto sobre el poder del autómata.

Cualquier cálculo que se pueda realizar en una nueva disposición de este tipo seguirá perteneciendo a la categoría de cálculo mecánico y, por lo tanto, se puede realizar mediante un modelo estándar. No obstante, es instructivo estudiar modelos más complejos, aunque sólo sea por el hecho de que una demostración explícita del resultado esperado demostrará el poder de la Máquina de Turing y, por lo tanto, aumentará nuestra confianza en la tesis de Turing.

Son posibles muchas variaciones del modelo básico de la Definición 9.1. Por ejemplo, podemos considerar máquinas de Turing con más de una cinta o con cintas que se extienden en varias dimensiones. Consideraremos variantes que serán útiles en discusiones posteriores.

9.1. Variaciones menores a las Máquinas de Turing

Primero consideramos algunos cambios relativamente menores en la Definición 9.1 e investigamos si estos cambios hacen alguna diferencia en el concepto general. Siempre que cambiamos una definición, introducimos un nuevo tipo de autómatas y planteamos la cuestión de si estos nuevos autómatas son en algún sentido real diferentes de los que ya hemos encontrado.

¿Qué entendemos por diferencia esencial entre una clase de autómatas y otra? Aunque puede haber claras diferencias en sus definiciones, estas diferencias pueden no tener consecuencias interesantes. Hemos visto un ejemplo de esto en el caso de autómatas finitos deterministas y no deterministas. Estos tienen definiciones bastante diferentes, pero son equivalentes en el sentido de que ambos se identifican exactamente con la familia de lenguas regulares.

Extrapolando esto, podemos definir equivalencia o no equivalencia para clases de autómatas en general.

9.1.1. Equivalencia de Clases de Autómatas

Siempre que definamos la equivalencia para dos autómatas o clases de autómatas, debemos establecer cuidadosamente qué debe entenderse por esta equivalencia. Para el resto de este capítulo, seguimos la precedencia establecida para nfa y dfa y definimos la equivalencia con respecto a la capacidad de aceptar lenguajes.

Definición 9.1 Dos autómatas son equivalentes si aceptan el mismo lenguaje. Considere dos clases de autómatas C_1 y C_2. Si por cada autómata M_1 en C_1 hay un autómata M_2 en C_2 tal que

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

decimos que C_2 es al menos tan poderosa como C_1. Si lo contrario también se cumple y para cada M_2 en C_2 hay un M_1 en C_1 tal que $L(M_1) = L(M_2)$, decimos que C_1 y C_2 son equivalentes.

Hay muchas formas de establecer la equivalencia de los autómatas. La construcción del Teorema 2.2 hace esto para dfas y nfas. Para demostrar la

equivalencia en conexión con las máquinas de Turing, a menudo usamos una técnica importante llamada **simulación**.

Sea M un autómata. Decimos que otro autómata $\widehat{M}$ puede simular un cálculo de M si $\widehat{M}$ puede imitar el cálculo de M de la siguiente manera. Sea $d_0, d_1, \ldots$ la secuencia de descripciones instantáneas de un cálculo de M, es decir,

$$d_0 \vdash_M d_1 \vdash_M \cdots \vdash_M d_n \cdots .$$

Entonces $\widehat{M}$ simula este cálculo si lleva a cabo un

$$\widehat{d_0} \overset{*}{\vdash}_{\widehat{M}} \widehat{d_1} \overset{*}{\vdash}_{\widehat{M}} \cdots \overset{*}{\vdash}_{\widehat{M}} \widehat{d_n} \cdots ,$$

donde $\widehat{d_0}, \widehat{d_1}, \ldots$ son descripciones instantáneas, de manera que cada una de ellas está asociada a una configuración única de M. En otras palabras, si conocemos el cálculo realizado por $\widehat{M}$, podemos determinar a partir de él exactamente qué cálculos habría realizado M, dada la configuración inicial correspondiente.

Tenga en cuenta que la simulación de un solo movimiento $d_i \vdash_M d_{i+1}$ de M puede implicar varios movimientos de $\widehat{M}$. Las configuraciones intermedias en $\widehat{d_i} \overset{*}{\vdash}_{\widehat{M}} \widehat{d_{i+1}}$ pueden no corresponder a ninguna configuración de M, pero esto no afecta nada si podemos decir qué configuraciones de $\widehat{M}$ son relevantes.

Siempre que podamos determinar a partir del cálculo de $\widehat{M}$ lo que habría hecho M, la simulación es adecuada. Si $\widehat{M}$ puede simular todos los cálculos de M, decimos que $\widehat{M}$ puede simular M.

Debería quedar claro que si $\widehat{M}$ puede simular M, entonces las cosas se pueden arreglar para que M y $\widehat{M}$ acepten el mismo lenguaje, y los dos autómatas sean equivalentes.

Para demostrar la equivalencia de dos clases de autómatas, mostramos que por cada máquina en una clase, hay una máquina en la segunda clase capaz de simularla, y viceversa.

9.1.2. Máquinas de Turing con opción de permanencia

En nuestra definición de una Máquina de Turing estándar, la cabeza de lectura y escritura debe moverse hacia la derecha o hacia la izquierda. A veces es conveniente proporcionar una tercera opción, que la cabeza de lectura y escritura permanezca en su lugar después de reescribir el contenido de la

celda. Por lo tanto, podemos definir una Máquina de Turing con una opción de permanencia reemplazando δ en la Definición 9.1 por

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

con la interpretación de que S significa que no hay movimiento de la cabeza de lectura-escritura. Esta opción no amplía la potencia del autómata.

Teorema 9.1 La clase de máquinas Turing con opción de permanencia es equivalente a la clase de máquinas Turing estándar.

Demostración: Dado que una Máquina de Turing con una opción de permanencia es claramente una extensión del modelo estándar, es obvio que cualquier Máquina de Turing estándar puede ser simulada por una con una opción de permanencia.

Para mostrar el sentido contrario, sea $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ una Máquina de Turing con una opción de permanencia para ser simulada por una Máquina de Turing estándar $\widehat{M} = (\widehat{Q}, \Sigma, \Gamma, \widehat{\delta}, \widehat{q}_0, \Box, \widehat{F})$. Para cada movimiento de M, la máquina de simulación $\widehat{M}$ hace lo siguiente. Si el movimiento de M no implica la opción de permanecer, la máquina simuladora realiza un movimiento, esencialmente idéntico al movimiento que se va a simular.

Si S está involucrado en el movimiento de M, entonces $\widehat{M}$ hará dos movimientos: el primero reescribe el símbolo y mueve la cabeza de lectura-escritura hacia la derecha; el segundo mueve la cabeza de lectura y escritura hacia la izquierda, dejando inalterado el contenido de la cinta. La máquina de simulación se puede construir a partir de M definiendo $\widehat{\delta}$ de la siguiente manera: Para cada transición

$$\delta(q_i, a) = (q_j, b, L \text{ o } R),$$

ponemos en $\widehat{\delta}$

$$\widehat{\delta}(\widehat{q}_i, a) = (\widehat{q}_j, b, L \text{ o } R).$$

Para cada transición-S

$$\delta(q_i, a) = (q_j, b, S),$$

ponemos en $\widehat{\delta}$ las transiciones correspondientes

$$\widehat{\delta}(\widehat{q}_i, a) = (\widehat{q}_{j_S}, b, R)$$

y

$$\widehat{\delta}(\widehat{q}_{j_S}, c) = (\widehat{q}_j, c, L),$$

para todo $c \in \Gamma$.

Es razonablemente obvio que cada cálculo de M tiene un cálculo correspondiente de $\widehat{M}$, de modo que $\widehat{M}$ puede simular M.

■

La simulación es una técnica estándar para mostrar la equivalencia de autómatas, y el formalismo que hemos descrito lo hace posible, como se muestra en el teorema anterior, hablar sobre el proceso con precisión y demostrar teoremas sobre equivalencia. En nuestra discusión posterior, usamos la noción de simulación con frecuencia, pero generalmente no intentamos describir todo de una manera rigurosa y detallada.

Las simulaciones completas con máquinas de Turing suelen ser engorrosas. Para evitar esto, mantenemos nuestra discusión descriptiva, más que en forma de prueba de teoremas. Las simulaciones se dan sólo a grandes rasgos, pero no debería ser difícil ver cómo se pueden hacer rigurosas. Al lector le resultará instructivo esbozar cada simulación en algún lenguaje de nivel superior o en pseudocódigo.

Antes de presentar otros modelos, hacemos un comentario sobre la Máquina de Turing estándar. Está implícito en la Definición 9.1 que cada símbolo de cinta puede estar compuesto de caracteres en lugar de uno solo. Esto puede hacerse más explícito dibujando una versión ampliada de la Figura 9.1 (Figura 9.1), en la que los símbolos de la cinta son tripletes de algún alfabeto más simple.

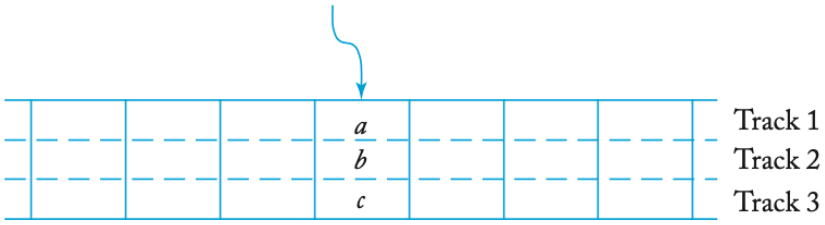


Figura 9.1: Máquina de Turing con tres pistas.

En la imagen, hemos dividido cada celda de la cinta en tres partes, llamadas **pistas**, cada una de las cuales contiene un miembro del triplete. Con base en esta visualización, este autómata a veces se denomina Máquina de Turing con **múltiples pistas**, pero tal vista de ninguna manera extiende la Definición 9.1, ya que todo lo que tenemos que hacer es que cada símbolo en el alfabeto Γ se componga de varias partes.

Sin embargo, otros modelos de máquinas de Turing implican un cambio de definición, por lo que debe demostrarse la equivalencia con la máquina estándar.

9.1.3. La Máquina de Turing fuera de línea

La definición general de un autómata en el Capítulo 1 contenía un archivo de entrada así como un almacenamiento temporal. En la Definición 9.1 descartamos el archivo de entrada por razones de simplicidad, alegando que esto no hace ninguna diferencia en el concepto de Máquina de Turing. Ahora ampliamos esta afirmación.

Si volvemos a poner el archivo de entrada en la discusión, obtenemos lo que se conoce como una **Máquina de Turing fuera de línea**. En una máquina de este tipo, cada movimiento se rige por el estado interno, lo que se lee actualmente en el archivo de entrada y lo que ve la cabeza de lectura y escritura. En la Figura 9.2 se muestra una representación esquemática de una máquina fuera de línea.

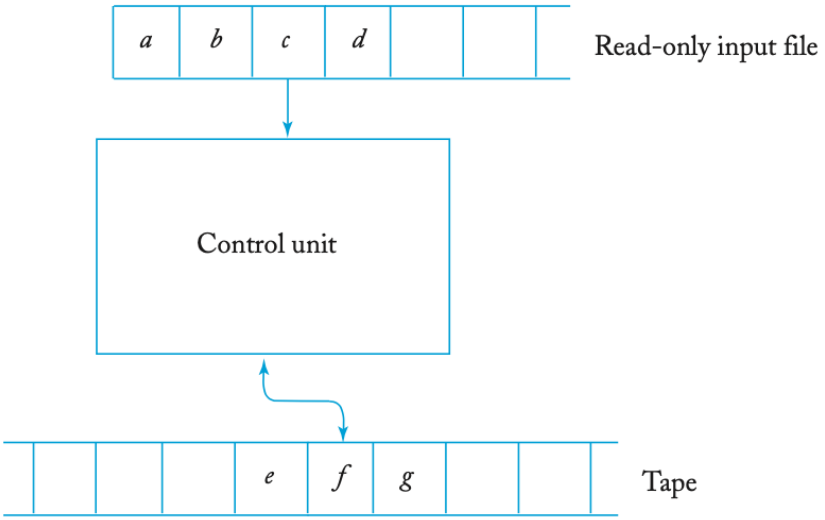


Figura 9.2: Máquina de Turing fuera de línea.

Es fácil hacer una definición formal de una Máquina de Turing fuera de línea, pero dejaremos esto como un ejercicio. Lo que queremos hacer brevemente es indicar por qué la clase de máquinas de Turing fuera de línea es

equivalente a la clase de máquinas estándar.

Primero, el comportamiento de cualquier Máquina de Turing estándar puede simularse mediante algún modelo fuera de línea. Todo lo que necesita hacer la máquina de simulación es copiar la entrada del archivo de entrada a la cinta. Entonces puede proceder de la misma manera que la máquina estándar.

La simulación de una máquina M fuera de línea por una máquina estándar $\widehat{M}$ requiere una descripción más extensa. Una máquina estándar puede simular el cálculo de una máquina fuera de línea utilizando la disposición de cuatro pistas que se muestra en la Figura 9.3. En esa imagen, el contenido de la cinta que se muestra representa la configuración específica de la Figura 9.2.

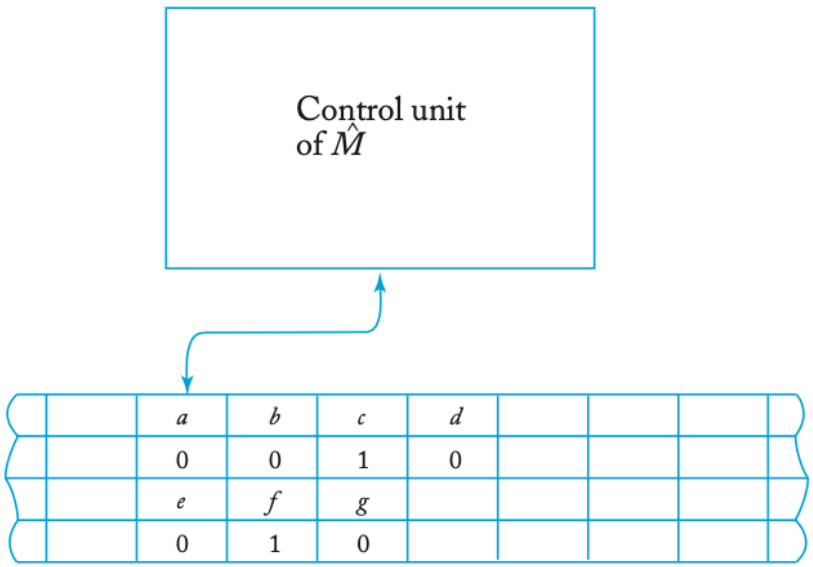


Figura 9.3: Máquina de Turing estándar $\widehat{M}$ que simula la Máquina de Turing fuera de línea M.

Cada una de las cuatro pistas de $\widehat{M}$ juega un papel específico en la simulación. La primera pista tiene la entrada, la segunda marca la posición en la que se lee la entrada, la tercera representa la cinta de M y la cuarta muestra la posición de la cabeza de lectura y escritura de M.

La simulación de cada movimiento de M requiere un número de movimientos de $\widehat{M}$. Comenzando desde alguna posición estándar, digamos el

extremo izquierdo, y con la información relevante marcada por marcadores de fin especiales, $\widehat{M}$ busca en la pista 2 para ubicar la posición en la que se lee el archivo de entrada de M.

El símbolo que se encuentra en la celda correspondiente en la pista 1 se recuerda poniendo la unidad de control de $\widehat{M}$ en un estado elegido para este propósito. A continuación, se busca en la pista 4 la posición de la cabeza de lectura-escritura de M. Con la entrada recordada y el símbolo en la pista 3, ahora sabemos que debe hacer M. $\widehat{M}$ recuerda nuevamente esta información con un estado interno apropiado.

A continuación, las cuatro pistas de la cinta de $\widehat{M}$ se modifican para reflejar el movimiento de M. Finalmente, la cabeza de lectura y escritura de $\widehat{M}$ vuelve a la posición estándar para la simulación del siguiente movimiento.

9.1.4. Ejercicios

1. Considere una Máquina de Turing que, en cualquier movimiento en particular, puede cambiar el símbolo de la cinta o mover la cabeza de lectura y escritura, pero no ambos.

   a) Dé una definición formal de tal máquina.

   b) Demuestre que la clase de tales máquinas es equivalente a la clase de máquinas de Turing estándar.

2. Muestre cómo una máquina del tipo definido en el ejercicio anterior podría simular los movimientos

$$\delta(q_i, a) = (q_j, b, R)$$
$$\delta(q_i, b) = (q_k, a, L)$$

   de una Máquina de Turing estándar.

9.2. Máquinas de Turing con almacenamiento más complejo

El dispositivo de almacenamiento de una máquina Turing estándar es tan simple que uno podría pensar que es posible ganar poder mediante el uso de dispositivos de almacenamiento más complejos. Pero este no es el caso, como ilustramos ahora.

9.2.1. Máquinas de Turing Multicintas

Una Máquina de Turing multicintas es una Máquina de Turing con varias cintas, cada una con su propia cabeza de lectura y escritura controlada independientemente (Figura 9.4).

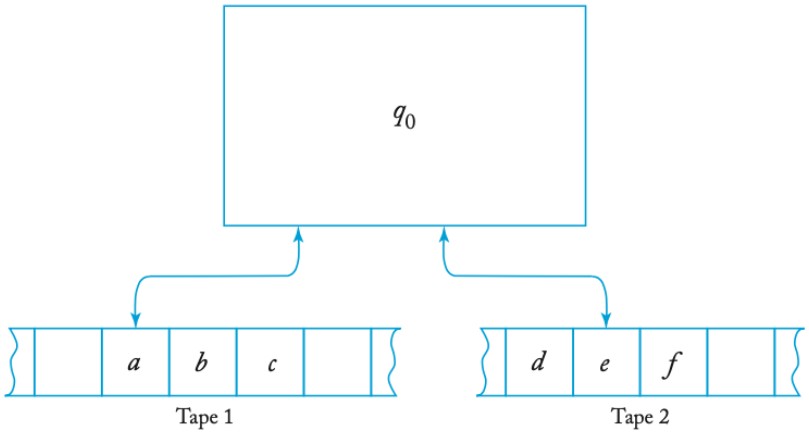


Figura 9.4: Máquina de Turing con dos cintas.

La definición formal de una Máquina de Turing de varias cintas va más allá de la definición 9.1, ya que requiere una función de transición modificada. Normalmente, definimos una máquina de n cintas por $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$, donde $Q, \Sigma, \Gamma, q_0, F$ son como en la Definición 9.1, pero

$$\delta : Q \times \Gamma^n \to Q \times \Gamma^n \times \{L, R\}^n$$

especifica lo que sucede en todas las cintas. Por ejemplo, si $n = 2$, con la configuración actual que se muestra en la Figura 9.4, entonces

$$\delta(q_0, a, e) = (q_1, x, y, L, R)$$

se interpreta de la siguiente manera. La regla de transición se puede aplicar sólo si la máquina está en el estado q_0 y la primera cabeza de lectura y escritura ve una a y la segunda una e.

El símbolo de la primera cinta será reemplazado por una x y su cabeza de lectura y escritura se moverá hacia la izquierda. Al mismo tiempo, el símbolo de la segunda cinta se reescribe como y y la cabeza de lectura y escritura se

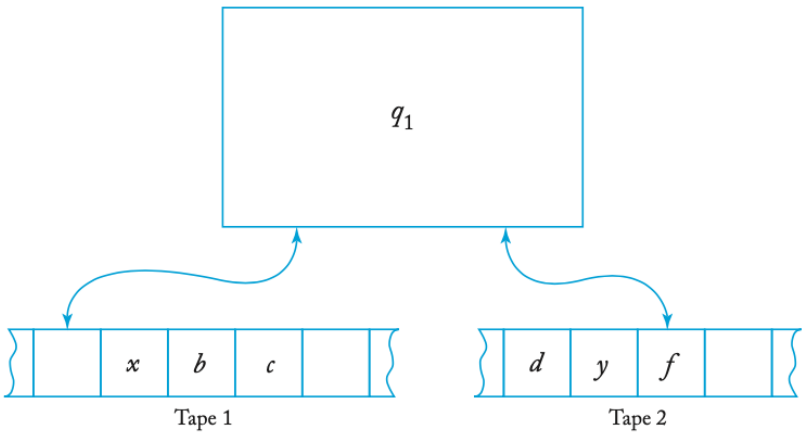


Figura 9.5: Resultado de aplicar la regla de transición $\delta(q_0, a, e) = (q_1, x, y, L, R)$ a la configuración de la Figura 9.4.

mueve hacia la derecha. Luego, la unidad de control cambia su estado a q_1 y la máquina pasa a la nueva configuración que se muestra en la Figura 9.5.

Para mostrar la equivalencia entre las máquinas de Turing estándar y de múltiples cintas, argumentamos que cualquier Máquina de Turing de múltiples cintas M puede ser simulada por una Máquina de Turing estándar $\widehat{M}$ y, a la inversa, que cualquier Máquina de Turing estándar puede ser simulada por una de múltiples cintas.

La segunda parte de esta afirmación no necesita elaboración, ya que siempre podemos optar por ejecutar una máquina de varias cintas con sólo una de sus cintas haciendo un trabajo útil. La simulación de una máquina de varias cintas por una con una sola cinta es un poco más complicada, pero conceptualmente sencilla.

Considere, por ejemplo, la máquina de dos cintas en la configuración que se muestra en la Figura 9.6. La máquina de simulación de una sola cinta tendrá cuatro pistas (Figura 9.7).

La primera pista representa el contenido de la cinta 1 de M. La parte que no está en blanco de la segunda pista tiene todos ceros, excepto por un solo 1 que marca la posición de la cabeza de lectura y escritura de M.

Las pistas 3 y 4 juegan un papel similar para la cinta 2 de M. La figura 9.7 deja claro que, para las configuraciones relevantes de $\widehat{M}$ (es decir, las que tienen la forma indicada), existe una única configuración correspondiente de

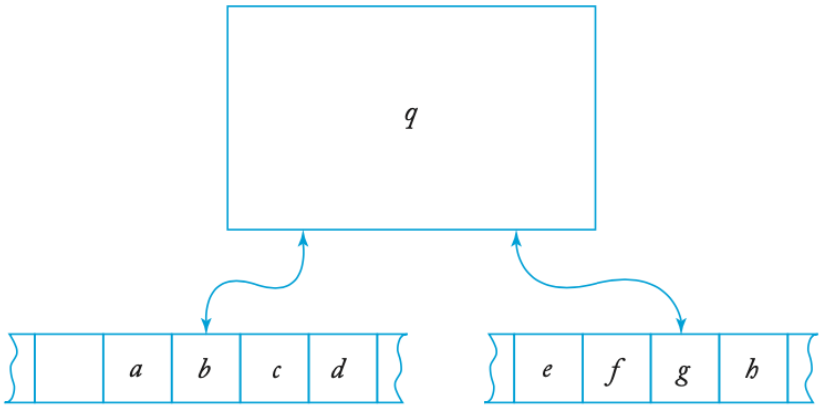


Figura 9.6: Máquina de Turing con dos cintas.

M.

La representación de una máquina de varias cintas mediante una máquina de una sola cinta es similar a la que se utiliza en la simulación de una máquina fuera de línea. Los pasos reales de la simulación también son muy parecidos, con la única diferencia de que hay que considerar más cintas.

El esquema dado para la simulación de máquinas fuera de línea se traslada a este caso con modificaciones menores y sugiere un procedimiento mediante el cual la función de transición $\widehat{\delta}$ de $\widehat{M}$ puede construirse a partir de la función de transición δ de M.

Para que la construcción sea precisa, se necesita mucha escritura. Ciertamente, los cálculos de $\widehat{M}$ dan la apariencia de ser largos y elaborados, pero esto no influye en la conclusión. Todo lo que se puede hacer en M también se puede hacer en $\widehat{M}$.

Es importante tener en cuenta el siguiente punto. Cuando afirmamos que una Máquina de Turing con múltiples cintas no es más poderosa que una estándar, solo estamos haciendo una declaración sobre lo que pueden hacer estas máquinas, en particular, qué lenguajes pueden ser aceptados.

Ejemplo 9.1 Considere el lenguaje $\{a^n b^n\}$. En el ejemplo 9.7, describimos un método laborioso por el cual este lenguaje puede ser aceptado por una Máquina de Turing con una cinta. El uso de una máquina de dos cintas facilita mucho el trabajo.

Suponga que se escribe una cadena inicial $a^n b^m$ en la cinta 1 al comienzo del cálculo. Luego leemos todas las as y las copiamos en la cinta 2. Cuando

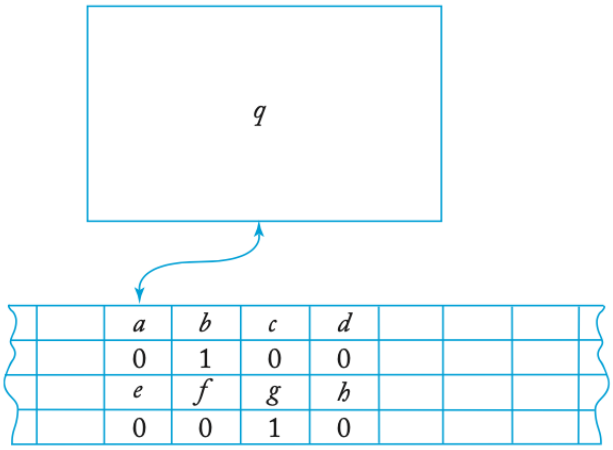


Figura 9.7: Máquina de simulación con una sola cinta y cuatro pistas correspondiente a la Figura 9.6.

llegamos al final de las as, comparamos las bs de la cinta 1 con las as copiadas de la cinta 2. De esta manera, podemos determinar si hay un número igual de as y bs sin movimientos repetidos hacia adelante y hacia atrás de la cabeza de lectura y escritura.

□

Recuerde que los diversos modelos de máquinas de Turing se consideran equivalentes sólo con respecto a su capacidad para hacer cosas, no con respecto a la facilidad de programación o cualquier otra medida de eficiencia que podamos considerar.

9.2.2. Ejercicios

1. Defina lo que se podría llamar una Máquina de Turing fuera de línea de varias cintas y describa cómo puede ser simulada por una Máquina de Turing estándar.

9.3. Máquinas de Turing No Deterministas

Si bien la tesis de Turing hace plausible que la estructura específica de la cinta sea irrelevante para el poder de la Máquina de Turing, no se puede

decir lo mismo del no determinismo. Dado que el no determinismo implica un elemento de elección y, por lo tanto, tiene un sabor no mecánico, apelar a la tesis de Turing es inapropiado.

Debemos analizar el efecto del no determinismo con más detalle si queremos argumentar que el no determinismo no agrega nada al poder de una Máquina de Turing. Nuevamente recurrimos a la simulación, mostrando que el comportamiento no determinista puede manejarse de manera determinista.

Definición 9.2 Una Máquina de Turing no determinista es un autómata según la Definición 9.1, excepto que δ es ahora una función

$$\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{L,R\}}.$$

Como siempre, cuando está involucrado el no determinismo, el rango de δ es un conjunto de posibles transiciones, cualquiera de las cuales puede ser elegida por la máquina.

Ejemplo 9.2 Si una Máquina de Turing tiene transiciones especificadas por

$$\delta(q_0, a) = \{(q_1, b, R), (q_2, c, L)\},$$

es no determinista. Los dos movimientos

$$q_0aaa \vdash bq_1aa$$

y

$$q_0aaa \vdash q_2\Box caa$$

son posibles.

□

Dado que no está claro qué papel juega el no determinismo al calcular funciones, los autómatas no deterministas generalmente se consideran aceptadores. Se dice que una Máquina de Turing no determinista acepta w si hay alguna secuencia posible de movimientos tal que

$$q_0w \overset{*}{\vdash} x_1q_fx_2,$$

con $q_f \in F$. Una máquina no determinista puede tener movimientos disponibles que conducen a un estado no final o a un bucle infinito. Pero, como

siempre ocurre con el no determinismo, estas alternativas son irrelevantes; todo lo que nos interesa es la existencia de una secuencia de movimientos que conducen a la aceptación.

Para mostrar que una Máquina de Turing no determinista no es más poderosa que una determinista, necesitamos proporcionar una equivalente determinista para la no determinista. Ya hemos aludido a uno. El no determinismo puede verse como un algoritmo de retroceso determinista, y una máquina determinista puede simular una no determinista siempre que pueda manejar la contabilidad involucrada en el retroceso.

Para ver cómo se puede hacer esto simplemente, consideremos una visión alternativa del no determinismo, una que es útil en muchos argumentos: una máquina no determinista puede verse como una que tiene la capacidad de replicarse a sí misma siempre que sea necesario. Cuando es posible más de un movimiento, la máquina produce tantas réplicas como sea necesario y asigna a cada réplica la tarea de realizar una de las alternativas.

Esta visión del no determinismo puede parecer particularmente no mecanicista, ya que la replicación ilimitada ciertamente no está al alcance de las computadoras actuales. Sin embargo, es posible una simulación.

Una forma de visualizar la simulación es usar una Máquina de Turing estándar, manteniendo todas las posibles descripciones instantáneas de la máquina no determinista en su cinta, separadas por alguna convención.

La figura 9.8 muestra la forma en que pueden aparecer las dos configuraciones aq_0aa y bbq_1a. El símbolo $\times$ se utiliza para delimitar el área de interés, mientras que $+$ separa descripciones instantáneas individuales. La máquina de simulación mira todas las configuraciones activas y las actualiza de acuerdo con el programa de la máquina no determinista.

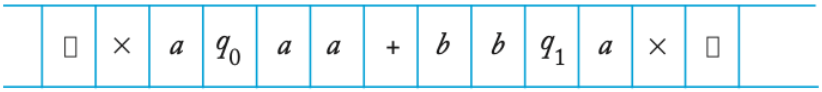


Figura 9.8: Representación en la cinta de las configuraciones aq_0aa y bbq_1a.

Las nuevas configuraciones o la expansión de descripciones instantáneas implicarán mover los marcadores $\times$. Los detalles son ciertamente tediosos, pero no difíciles de visualizar. Con base en esta simulación, llegamos a la conclusión de que para cada Máquina de Turing no determinista existe una máquina estándar determinista equivalente.

Teorema 9.2 La clase de máquinas de Turing deterministas y la clase de máquinas de Turing no deterministas son equivalentes.

Demostración: Utilice la construcción sugerida anteriormente para mostrar que cualquier Máquina de Turing no determinista puede ser simulada por una determinista.

■

Más adelante reconsideraremos el efecto del no determinismo en situaciones prácticas, por lo que necesitamos agregar algunos comentarios. Como siempre, el no determinismo puede verse como una elección entre alternativas. Esto se puede visualizar como un árbol de decisiones (Figura 9.9).

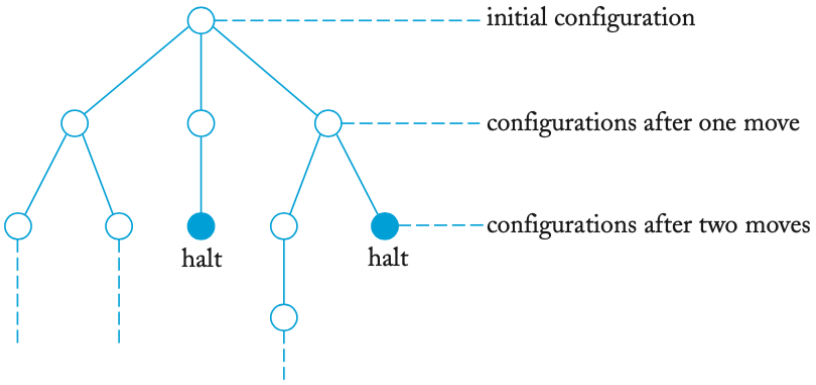


Figura 9.9: Árbol de decisiones para los movimientos de una Máquina de Turing No Determinista.

El ancho de dicho árbol de configuración depende del factor de ramificación, es decir, la cantidad de opciones disponibles en cada movimiento. Si k denota la ramificación máxima, entonces

$$M = k^n \tag{9.1}$$

es el número máximo de configuraciones que pueden existir después de n movimientos. Para fines posteriores, es necesario profundizar en la definición de aceptación del lenguaje y también incluir el asunto de la membresía.

Definición 9.3 Se dice que una Máquina de Turing no determinista M *acepta* un lenguaje L si, para todo $w \in L$, al menos una de las configuraciones posibles acepta w. Puede haber ramificaciones que lleven a configuraciones

de no aceptación, mientras que algunas pueden poner la máquina en un bucle infinito. Pero estos son irrelevantes para la aceptación.

Se dice que una Máquina de Turing no determinista M *decide* un lenguaje L si, para todo $w \in \Sigma^*$, hay un camino que conduce a la aceptación o al rechazo.

9.3.1. Ejercicios

1. Escriba programas para máquinas de Turing no deterministas que acepten los siguientes lenguajes. En cada caso, explique si el no determinismo simplifica la tarea y cómo.

   a) $L = \{ww : w \in \{a, b\}^+\}$.

   b) $L = \{ww^R w : w \in \{a, b\}^+\}$.

   c) $L = \{xww^R y : x, y, w \in \{a, b\}^+, |x| \geq |y|\}$.

   d) $L = \{w : n_a(w) = n_b(w) = n_c(w)\}$.

9.4. Máquina de Turing Universal

Considere el siguiente argumento en contra de la tesis de Turing: "Una Máquina de Turing como se presenta en la Definición 9.1 es una computadora de propósito especial. Una vez que se define δ, la máquina se limita a realizar un tipo particular de cálculo.

Las computadoras digitales, por otro lado, son máquinas de uso general que se pueden programar para realizar diferentes trabajos en diferentes momentos. En consecuencia, las máquinas de Turing no pueden considerarse equivalentes a las computadoras digitales de uso general".

Esta objeción se puede superar diseñando una Máquina de Turing reprogramable, llamada **Máquina de Turing Universal**. Una Máquina de Turing Universal M_u es un autómata que, dada como entrada la descripción de cualquier Máquina de Turing M y una cadena w, puede simular el cálculo de M sobre w. Para construir tal M_u, primero elegimos una forma estándar para describir las máquinas de Turing.

Podemos, sin perdida de generalidad, suponer que

$$Q = \{q_1, q_2, \ldots, q_n\},$$

con q_1 el estado inicial, q_2 el estado final único y

$$\Gamma = \{a_1, a_2, \ldots, a_m\},$$

donde a_1 representa el espacio en blanco. Luego seleccionamos una codificación en la que q_1 está representado por 1, q_2 está representado por 11, y así sucesivamente. De manera similar, a_1 se codifica como 1, a_2 como 11, etc. El símbolo 0 se utilizará como separador entre los unos.

Con el estado inicial y final y el blanco definidos por esta convención, cualquier Máquina de Turing puede describirse completamente solamente con δ. La función de transición se codifica de acuerdo con este esquema, con los argumentos y el resultado en alguna secuencia prescrita.

Por ejemplo, $\delta(q_1, a_2) = (q_2, a_3, L)$ podría aparecer como

$$\cdots 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 0 \cdots$$

De esto se deduce que cualquier Máquina de Turing tiene una codificación finita como una cadena en $\{0,1\}^+$ y que, dada cualquier codificación de M, podemos decodificarla de forma única.

Algunas cadenas no representarán ninguna Máquina de Turing (por ejemplo, la cadena 00011), pero podemos detectarlas fácilmente, por lo que no son motivo de preocupación. Una Máquina de Turing Universal M_u tiene entonces un alfabeto de entrada que incluye $\{0,1\}$ y la estructura de una máquina de varias cintas, como se muestra en la Figura 9.10.

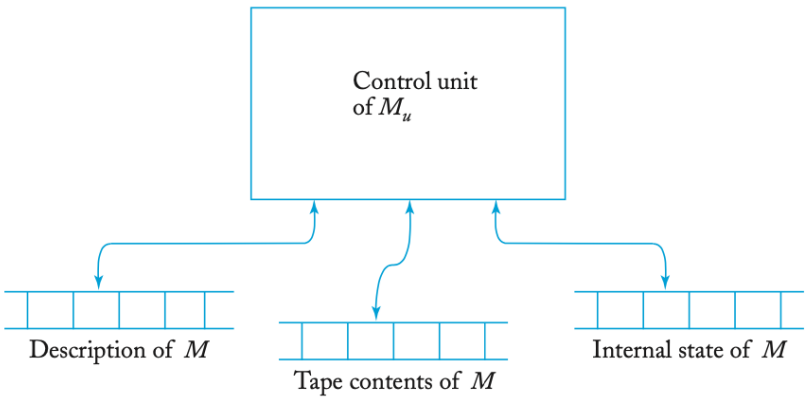


Figura 9.10: Máquina de Turing Universal M_u con tres cintas.

Para cualquier entrada M y w, la cinta 1 mantendrá una definición codificada de M. La cinta 2 contendrá el contenido de la cinta M, y la cinta 3 el estado interno de M. M_u examina primero el contenido de las cintas 2 y 3 para determinar la configuración de M. A continuación, consulta la cinta 1 para ver qué haría M en esta configuración. Finalmente, las cintas 2 y 3 se modificarán para reflejar el resultado del movimiento.

Está dentro de lo razonable construir una Máquina de Turing Universal real (ver, por ejemplo, Denning, Dennis y Qualitz 1978), pero el proceso carece de interés. En cambio, preferimos apelar a la hipótesis de Turing. La implementación claramente se puede hacer usando algún lenguaje de programación.

Por lo tanto, esperamos que también se pueda realizar con una máquina Turing estándar. Entonces, tenemos justificación para afirmar la existencia de una Máquina de Turing que, dado cualquier programa, puede realizar los cálculos especificados por ese programa y que, por lo tanto, es un modelo adecuado para una computadora de propósito general.

La observación de que cada Máquina de Turing puede representarse mediante una cadena de ceros y unos tiene implicaciones importantes. Pero antes de explorar estas implicaciones, necesitamos revisar algunos resultados de la teoría de conjuntos.

Algunos conjuntos son finitos, pero la mayoría de los conjuntos (y lenguajes) interesantes son infinitos. Para conjuntos infinitos, distinguimos entre conjuntos que son contables y conjuntos que son incontables. Se dice que un conjunto es contable si sus elementos se pueden poner en una correspondencia uno a uno con los números enteros positivos.

Con esto queremos decir que los elementos del conjunto se pueden escribir en algún orden, digamos, $x_1, x_2, x_3, \ldots$, de modo que cada elemento del conjunto tenga algún índice finito. Por ejemplo, el conjunto de todos los enteros pares se puede escribir en el orden $0, 2, 4, \ldots$. Dado que cualquier entero positivo $2n$ aparece en la posición $n + 1$, el conjunto es contable.

Esto no debería ser demasiado sorprendente, pero hay ejemplos más complicados, algunos de los cuales pueden parecer contradictorios. Tome el conjunto de todos los cocientes de la forma p/q, donde p y q son números enteros positivos.

¿Cómo deberíamos ordenar este conjunto para demostrar que es contable? No podemos usar la secuencia

$$\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots$$

porque entonces el $\frac{2}{3}$ nunca aparecería. Esto no implica que el conjunto sea incontable; en este caso, hay una forma inteligente de ordenar el conjunto para demostrar que, de hecho, es contable.

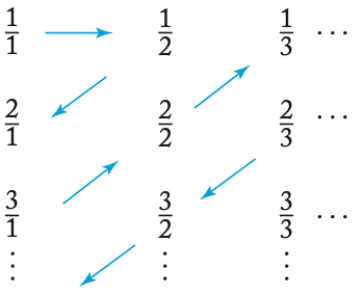


Figura 9.11: Diagonalización del conjunto de cocientes de enteros positivos.

Observe el esquema que se muestra en la Figura 9.11 y escriba el elemento en el orden en que se encuentran siguiendo las flechas. Esto nos da

$$\frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{3}{1}, \frac{2}{2}, \frac{1}{3}, \frac{2}{3} \cdots$$

Aquí el elemento $\frac{2}{3}$ aparece en el séptimo lugar y cada elemento tiene alguna posición en la secuencia. Por tanto, el conjunto es contable.

Vemos en este ejemplo que podemos probar que un conjunto es contable si podemos producir un método mediante el cual sus elementos se pueden escribir en alguna secuencia. A este método lo llamamos **procedimiento de enumeración**. Dado que un procedimiento de enumeración es una especie de proceso mecánico, podemos usar un modelo de Máquina de Turing para definirlo formalmente.

Definición 9.4 Sea S un conjunto de cadenas en algún alfabeto Σ. Entonces, un procedimiento de enumeración para S es una Máquina de Turing que puede realizar la secuencia de pasos

$$q_0\square \overset{*}{\vdash} q_s x_1 \# s_1 \overset{*}{\vdash} q_s x_2 \# s_2 \overset{*}{\vdash} \cdots,$$

con $x_i \in \Gamma^* - \{\#\}$, $s_i \in S$, de tal manera que cualquier s en S se produce en un número finito de pasos. El estado q_s es un estado que significa pertenencia a S; es decir, siempre que se ingrese q_s, la cadena que sigue a # debe estar en S.

No todos los conjuntos son contables. Como veremos en el próximo capítulo, hay algunos conjuntos incontables. Pero cualquier conjunto para el que existe un procedimiento de enumeración es contable porque la enumeración proporciona la secuencia requerida.

Estrictamente hablando, un procedimiento de enumeración no puede llamarse algoritmo ya que no terminará cuando S sea infinito. Sin embargo, puede considerarse un proceso significativo, porque produce resultados bien definidos y predecibles.

Ejemplo 9.3 Sea $\Sigma = \{a, b, c\}$. Demuestre que $S = \Sigma^+$ es contable.

Solución: Podemos demostrar que $S = \Sigma^+$ es contable si podemos encontrar un procedimiento de enumeración que produzca sus elementos en algún orden, digamos en el orden en que aparecerían en un diccionario.

Sin embargo, el orden utilizado en los diccionarios no es adecuado sin modificaciones. En un diccionario, todas las palabras que comienzan con a se enumeran antes de la cadena b. Pero cuando hay un número infinito de palabras a, nunca llegaremos a b, violando así la condición de la Definición 9.4 de que cualquier cadena dada se enumere después de un número finito de pasos.

En su lugar, podemos usar un orden modificado, en el que tomamos la longitud de la cadena como primer criterio, seguido de un orden alfabético de todas las cadenas de igual longitud. Este es un procedimiento de enumeración que da la secuencia

$$a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \ldots$$

Como tendremos varios usos para tal orden, lo llamaremos el **orden propio**. □

Una consecuencia importante de la discusión anterior es que las máquinas de Turing son contables.

Teorema 9.3 El conjunto de todas las máquinas de Turing, aunque infinito, es contable.

Demostración: Podemos codificar cada Máquina de Turing usando 0 y 1. Con esta codificación, construimos el siguiente procedimiento de enumeración.

1. Genere la siguiente cadena en $\{0, 1\}^+$ en el orden propio.

2. Verifique la cadena generada para ver si define una Máquina de Turing. Si es así, escríbalo en la cinta en la forma requerida por la Definición 9.4. Si no es así, ignore la cadena.

3. Regrese al paso 1.

■

Dado que cada Máquina de Turing tiene una descripción finita, este proceso eventualmente generará cualquier máquina específica.

El orden particular de las máquinas de Turing depende de la codificación que usemos; si usamos una codificación diferente, debemos esperar un orden diferente. Sin embargo, esto no tiene importancia y muestra que el orden en sí no es importante. Lo que importa es la existencia de algún orden.

9.4.1. Ejercicios

1. Proporcione la codificación, utilizando el método sugerido, para

$$\delta(q_1, a_1) = (q_1, a_1, R),$$
$$\delta(q_1, a_2) = (q_3, a_1, L),$$
$$\delta(q_3, a_1) = (q_2, a_2, L).$$

2. Si a está codificado como 1, b como 11, R como 1, L como 11, decodifique la cadena 011010111011010.

9.5. Autómatas acotados linealmente

Si bien no es posible extender el poder de la máquina de Turing estándar complicando la estructura de la cinta, es posible limitarla restringiendo la forma en que se puede usar la cinta. Ya hemos visto un ejemplo de esto con los autómatas de pila.

Un autómata de de pila puede considerarse como una máquina de Turing no determinista con una cinta que está restringida a usarse como una pila. También podemos restringir el uso de la cinta de otras formas; por ejemplo, podríamos permitir que sólo una parte finita de la cinta se utilice como espacio de trabajo.

Se puede demostrar que esto nos lleva de regreso a los autómatas finitos, por lo que no necesitamos seguir adelante. Pero hay una forma de limitar el uso de la cinta que conduce a una situación más interesante: permitimos que la máquina utilice sólo la parte de la cinta ocupada por la entrada.

Por lo tanto, hay más espacio disponible para cadenas de entrada largas que para cadenas cortas, lo que genera otra clase de máquinas, los **autómatas acotados linealmente** (o **lba**).

Un autómata acotado linealmente, como una máquina de Turing estándar, tiene una cinta ilimitada, pero la cantidad de cinta que se puede utilizar depende de la entrada. En particular, restringimos la parte utilizable de la cinta a exactamente las celdas tomadas por la entrada[1].

Para hacer cumplir esto, podemos imaginar la entrada entre corchetes con dos símbolos especiales, el **marcador del extremo izquierdo** [y el **marcador del extremo derecho**]. Para una entrada w, la configuración inicial de la máquina de Turing viene dada por la descripción instantánea $q_0[w]$.

Los marcadores de finalización no se pueden reescribir y la cabeza de lectura y escritura no se puede mover a la izquierda de [ni a la derecha de]. A veces decimos que la cabeza de lectura y escritura "rebota" en los marcadores finales.

Definición 9.5 Un autómata acotado linealmente es una máquina de Turing no determinista $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$, como en la Definición 9.2, sujeto a la restricción de que Σ debe contener dos símbolos especiales [y], tal que $\delta(q_i, [)$ sólo puede contener elementos de la forma $(q_j, [, R)$, y $\delta(q_i,])$ sólo puede contener elementos de la forma $(q_j,], L)$.

Definición 9.6 Una cadena w es aceptada por un autómata acotado linealmente si hay una posible secuencia de movimientos

$$q_0[w] \overset{*}{\vdash} [x_1 q_f x_2]$$

para algunos $q_f \in F, x_1, x_2 \in \Gamma^*$. El lenguaje aceptado por el lba es el conjunto de todas las cadenas aceptadas.

[1] En algunas definiciones, la parte utilizable de la cinta es un múltiplo de la longitud de entrada, donde el múltiplo puede depender del lenguaje, pero no de la entrada. Aquí usamos sólo la longitud exacta de la cadena de entrada, pero permitimos máquinas multipistas, con la entrada en una sola pista.

Tenga en cuenta que en esta definición se supone que un autómata acotado linealmente no es determinista. Esto no es solo una cuestión de conveniencia, sino que es esencial para la discusión de este tipo de autómatas.

Ejemplo 9.4 El lenguaje

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

es aceptado por algún autómata acotado linealmente. Una estrategia para reconocerlo es cambiar los símbolos a, b y c, por x, y y z, al final si todos los símbolos de entrada fueron reemplazados entonces se acepta la cadena, de lo contrario se rechaza.

Note que este cálculo no requiere más espacio que el ocupado por la cadena de entrada, así que puede ser realizado por un autómata acotado linealmente.

□

Ejemplo 9.5 Encuentre un autómata acotado linealmente que acepte el lenguaje

$$L = \{a^{n!} : n \geq 0\}.$$

Una forma de resolver el problema es dividir el número de símbolos a sucesivamente por $2, 3, 4, \ldots$, hasta que podamos aceptar o rechazar la cadena.

Si la entrada está en L, eventualmente quedará una sola a; si no, en algún momento surgirá un resto distinto de cero. Esbozamos la solución para señalar una implicación tácita de la Definición 9.5. Dado que la cinta de un autómata acotado linealmente puede ser multipista, las pistas adicionales se pueden utilizar como espacio de trabajo.

Para este problema, podemos utilizar una cinta de dos pistas. La primera pista contiene el número de símbolos a que quedan durante el proceso de división, y la segunda pista contiene el divisor actual (Figura 9.12). La solución real es bastante simple. Usando el divisor en la segunda pista, dividimos el número de símbolos a en la primera pista, por ejemplo, eliminando todos los símbolos excepto aquellos en múltiplos del divisor.

Después de esto, incrementamos el divisor en uno y continuamos hasta que encontremos un resto distinto de cero o nos quedemos con una sola a.

□

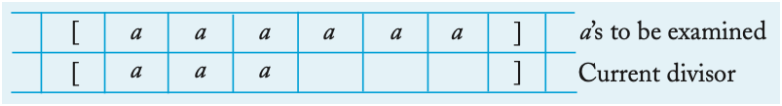


Figura 9.12: Las dos pistas del autómata acotado linealmente para el Ejemplo 9.5.

Los dos últimos ejemplos sugieren que los autómatas acotados linealmente son más poderosos que los autómatas de pila, ya que ninguno de dichos lenguajes es libre de contexto.

Para probar tal conjetura, todavía tenemos que demostrar que cualquier lenguaje libre de contexto puede ser aceptado por un autómata acotado linealmente. Haremos esto más tarde de una manera un tanto indirecta.

No es tan fácil hacer una conjetura sobre la relación entre las máquinas de Turing y los autómatas acotados linealmente. Problemas como el Ejemplo 9.5 se pueden resolver invariablemente mediante un autómata acotado linealmente, ya que se dispone de una cantidad de espacio temporal proporcional a la longitud de la entrada.

De hecho, es bastante difícil llegar a un lenguaje concreto y explícitamente definido que no pueda ser aceptado por ningún autómata acotado linealmente. En el Capítulo 10 mostraremos que la clase de autómatas acotados linealemente es menos poderosa que la clase de máquinas de Turing no restringidas, pero una demostración de esto requiere mucho más trabajo.

9.5.1. Ejercicios

Explique cómo se podrían construir autómatas acotados linealmente para aceptar los siguientes lenguajes:

1. $L = \{a^n : n = m^2, m \geq 1\}$.
2. $L = \{a^n : n$ es un número primo$\}$.
3. $L = \{ww : w \in \{a, b\}^+\}$.

Capítulo 10

Una Jerarquía de Lenguajes Formales y Autómatas

Volvamos ahora nuestra atención a nuestro principal interés, el estudio de los lenguajes formales. Nuestro objetivo inmediato será examinar los lenguajes asociados con las máquinas de Turing y algunas de sus restricciones.

Debido a que las máquinas de Turing pueden realizar cualquier tipo de cálculo algorítmico, esperamos encontrar que la familia de lenguajes asociados con ellas sea bastante amplia. Incluye no solo lenguajes regulares y libres de contexto, sino también los diversos ejemplos que hemos encontrado que se encuentran fuera de estas familias.

La pregunta no trivial es si hay algún lenguaje que no sea aceptado por alguna máquina de Turing. Primero responderemos a esta pregunta mostrando que hay más lenguajes que las máquinas de Turing, por lo que debe haber algunos lenguajes para los que no hay máquinas de Turing. La demostración es breve y elegante, pero no constructiva, y da poca información sobre el problema.

Por esta razón, estableceremos la existencia de lenguajes no reconocibles por las máquinas de Turing a través de ejemplos más explícitos que realmente nos permitan identificar uno de esos lenguajes. Otra vía de investigación será observar la relación entre las máquinas de Turing y ciertos tipos de gramáticas y establecer una conexión entre estas gramáticas y las gramáticas regulares y libres de contexto.

Esto conduce a una jerarquía de gramáticas y, a través de ella, a un método para clasificar familias de lenguajes. Algunos diagramas de teoría de conjuntos ilustran claramente las relaciones entre varias familias de lenguas.

Estrictamente hablando, muchos de los argumentos de este capítulo son válidos solo para lenguajes que no incluyen la cadena vacía. Esta restricción surge del hecho de que las máquinas de Turing, como las hemos definido, no pueden aceptar la cadena vacía.

Para evitar tener que reformular la definición o tener que agregar un descargo de responsabilidad repetido, asumimos tácitamente que los lenguajes discutidos en este capítulo, a menos que se indique lo contrario, no contienen λ.

10.1. Lenguajes recursivos y recursivamente enumerables

Comenzamos con algo de terminología para los lenguajes asociados con las máquinas de Turing. Al hacerlo, debemos hacer la importante distinción entre lenguajes para los que existe una máquina de Turing de aceptación y lenguajes para los que existe un algoritmo de pertenencia. Debido a que una máquina de Turing no se detiene necesariamente en una entrada que no acepta, la primera no implica la segunda.

Definición 10.1 Se dice que un lenguaje L es recursivamente enumerable si existe una máquina de Turing que lo acepta.

Esta definición implica sólo que existe una máquina de Turing M, tal que, para cada $w \in L$,

$$q_0 w \overset{*}{\vdash}_M x_1 q_f x_2,$$

con q_f un estado final. La definición no dice nada sobre lo que sucede para $w \notin L$; puede ser que la máquina se detenga en un estado no final o que nunca se detenga y entre en un bucle infinito.

Podemos ser más exigentes y pedir que la máquina nos diga si una entrada determinada está en su lenguaje o no.

Definición 10.2 Se dice que un lenguaje L sobre Σ es recursivo si existe una máquina de Turing M que acepta L y que se detiene sobre cada w en Σ^+. En otras palabras, un lenguaje es recursivo si y sólo si existe un algoritmo de pertenencia para él.

Si un lenguaje es recursivo, existe un procedimiento de enumeración fácil de construir. Supongamos que M es una máquina de Turing que determina la pertenencia a un lenguaje recursivo L. Primero construimos otra máquina de Turing, digamos $\widehat{M}$, que genera todas las cadenas en Σ^+ en el orden correcto, digamos $w_1, w_2, \ldots$. Como estas cadenas se generan, se convierten en la entrada a M, que se modifica para que escriba cadenas en su cinta sólo si están en L.

No es tan fácil de ver que existe también un procedimiento de enumeración para cada lenguaje recursivamente enumerable. No podemos usar el argumento anterior tal como está, porque si alguna w_j no está en L, la máquina M, cuando se inicia con w_j en su cinta, puede que nunca se detenga y, por lo tanto, nunca llegue a las cadenas de L que siguen a w_j en la enumeración.

Para asegurarse de que esto no suceda, el cálculo se realiza de una manera diferente. Primero obtenemos $\widehat{M}$ para generar w_1 y dejamos que M ejecute un movimiento en él. Luego dejamos que $\widehat{M}$ genere w_2 y dejamos que M ejecute un movimiento en w_2, seguido del segundo movimiento en w_1.

Después de esto, generamos w_3 y damos un paso en w_3, el segundo paso en w_2, el tercer paso en w_1, y así sucesivamente. El orden de ejecución se muestra en la Figura 10.1. A partir de esto, está claro que M nunca entrará en un bucle infinito. Dado que cualquier $w \in L$ es generado por $\widehat{M}$ y aceptado por M en un número finito de pasos, cada cadena en L es finalmente producida por M.

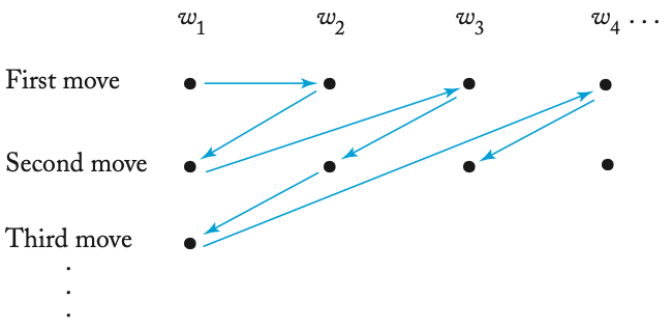


Figura 10.1: Orden de ejecución para enumerar un lenguaje recursivamente enumerable.

Es fácil ver que todos los lenguajes para los que existe un procedimiento de enumeración son enumerables de forma recursiva. Simplemente comparamos la cadena de entrada dada con cadenas sucesivas generadas por el

procedimiento de enumeración. Si $w \in L$, eventualmente obtendremos una coincidencia y el proceso se puede terminar.

Las definiciones 10.1 y 10.2 nos dan muy poca información sobre la naturaleza de los lenguajes recursivos o recursivamente enumerables. Estas definiciones adjuntan nombres a las familias de lenguajes asociadas con las máquinas de Turing, pero no arrojan luz sobre la naturaleza de los lenguajes representativos en estas familias. Tampoco nos dicen mucho sobre las relaciones entre estos lenguajes o su conexión con las familias de lenguajes que hemos encontrado antes.

Por lo tanto, nos enfrentamos de inmediato a preguntas como "¿Hay lenguajes que son recursivamente enumerables pero no recursivos?" y "¿Hay lenguajes, que se puedan describir de alguna manera, que no sean enumerables de forma recursiva?" Si bien podremos proporcionar algunas respuestas, no podremos producir ejemplos muy explícitos para ilustrar estas preguntas, especialmente la segunda.

10.1.1. Lenguajes que no son recursivamente enumerables

Podemos establecer la existencia de lenguajes que no son enumerables de forma recursiva de diversas formas. Una es muy corta y utiliza un resultado muy fundamental y elegante de las matemáticas.

Teorema 10.1 Sea S un conjunto numerable infinito. Entonces su conjunto potencia 2^S no es contable.

Demostración: Sea $S = \{s_1, s_2, s_3, \ldots\}$. Entonces, cualquier elemento t de 2^S puede representarse mediante una secuencia de ceros y unos, con un 1 en la posición i si y sólo si s_i está en t. Por ejemplo, el conjunto $\{s_2, s_3, s_6\}$ está representado por 011001, mientras que $\{s_1, s_3, s_5, \ldots\}$ está representado por $10101\ldots$.

Claramente, cualquier elemento de 2^S puede ser representado por tal secuencia, y cualquier secuencia representa un elemento único de 2^S. Supongamos que 2^S fuera contable; entonces sus elementos podrían escribirse en algún orden, digamos $t_1, t_2, \ldots$, y podríamos ingresarlos en una tabla, como se muestra en la Figura 10.2.

En esta tabla, tome los elementos de la diagonal principal y complemente cada entrada, es decir, reemplace 0 con 1 y viceversa. En el ejemplo de la

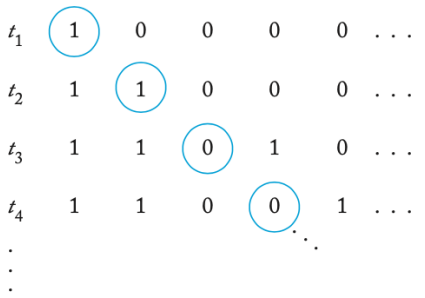


Figura 10.2: Diagonalización de 2^S.

Figura 11.2, los elementos son $1100\ldots$, por lo que obtenemos $0011\ldots$ como resultado.

La nueva secuencia a lo largo de la diagonal representa algún elemento de 2^S, digamos t_i para alguna i. Pero no puede ser t_1 porque difiere de t_1 en s_1. Por la misma razón, no puede ser t_2, t_3 ni ninguna otra entrada en la enumeración. Esta contradicción crea un callejón sin salida lógico que sólo puede eliminarse descartando la suposición de que 2^S es contable. ■

Este tipo de argumento, debido a que implica una manipulación de los elementos diagonales de una tabla, se llama *diagonalización*. La técnica se atribuye al matemático G. F. Cantor, quien la utilizó para demostrar que el conjunto de números reales no es contable.

En los próximos capítulos, veremos un argumento similar en varios contextos. El Teorema 10.1 es la diagonalización en su forma más pura.

Como consecuencia inmediata de este resultado, podemos mostrar que, en cierto sentido, hay menos máquinas de Turing que lenguajes, por lo que debe haber algunos lenguajes que no sean recursivamente enumerables.

Teorema 10.2 Para cualquier Σ no vacío, existen lenguajes que no son enumerables de forma recursiva.

Demostración: Un lenguaje es un subconjunto de Σ^*, y cada uno de esos subconjuntos es un lenguaje. Por lo tanto, el conjunto de todos los lenguajes es exactamente 2^{Σ^*}. Dado que Σ^* es infinito, el Teorema 10.1 nos dice que el conjunto de todos los lenguajes sobre Σ no es contable.

Pero el conjunto de todas las máquinas de Turing se puede enumerar, por lo que el conjunto de todos los lenguajes enumerables de forma recursiva es contable. Esto implica que debe haber algunos lenguajes sobre Σ que no se

puedan enumerar de forma recursiva.

■

Esta demostración, aunque breve y sencilla, es insatisfactoria en muchos sentidos. Es completamente no constructiva y, si bien nos informa de la existencia de algunos lenguajes que no son enumerables de forma recursiva, no nos da ningún indicio sobre cómo podrían ser estos lenguajes. En el siguiente conjunto de resultados, investigamos la conclusión de manera más explícita.

10.1.2. Un lenguaje que no es recursivamente enumerable

Dado que todo lenguaje que puede describirse de forma algorítmica directa puede ser aceptado por una máquina de Turing y, por tanto, es recursivamente enumerable, la descripción de un lenguaje que no sea recursivamente enumerable debe ser indirecta. Sin embargo, es posible. El argumento implica una variación del tema de la diagonalización.

Teorema 10.3 Existe un lenguaje recursivamente enumerable cuyo complemento no es recursivamente enumerable.

Demostración: Sea $\Sigma = \{a\}$, y considere el conjunto de todas las máquinas de Turing con este alfabeto de entrada. Según el teorema 9.3, este conjunto es contable, por lo que podemos asociar un orden $M_1, M_2, \ldots$ con sus elementos.

Para cada máquina de Turing M_i, hay un lenguaje asociado $L(M_i)$ enumerable recursivamente. Reciprocamente, para cada lenguaje enumerable recursivamente sobre Σ, hay alguna máquina de Turing que lo acepta.

Ahora consideramos un nuevo lenguaje L definido como sigue. Para cada $i \geq 1$, la cadena a^i está en L si y sólo si $a^i \in L(M_i)$. Está claro que el lenguaje L está bien definido, ya que el enunciado $a^i \in L(M_i)$, y por lo tanto $a^i \in L$, debe ser verdadero o falso. A continuación, consideramos el complemento de L,

$$\overline{L} = \{a^i : a^i \notin L(M_i)\}, \tag{10.1}$$

que también está bien definido pero, como mostraremos, no es recursivamente enumerable.

Mostraremos esto por contradicción, partiendo del supuesto de que $\overline{L}$ es recursivamente enumerable. Si es así, entonces debe haber alguna máquina

de Turing, digamos M_k, tal que

$$\overline{L} = L(M_k). \tag{10.2}$$

Considere la cadena a^k. ¿Está en L o en $\overline{L}$? Suponga que $a^k \in \overline{L}$. Por (10.2) esto implica que

$$a^k \in L(M_k).$$

Pero (10.1) ahora implica que

$$a^k \notin \overline{L}.$$

Alternativamente, si asumimos que a^k está en L, entonces $a^k \notin \overline{L}$ y (10.2) implica que

$$a^k \notin L(M_k).$$

Pero luego de (10.1) obtenemos que

$$a^k \in \overline{L}.$$

La contradicción es ineludible y debemos concluir que nuestra suposición de que L es recursivamente enumerable es falsa.

Para completar la demostración del teorema como se indica, aún debemos demostrar que L es recursivamente enumerable. Para ello, podemos utilizar el procedimiento de enumeración conocido para las máquinas de Turing.

Dada a^i, primero encontramos i contando el número de símbolos a. Luego usamos el procedimiento de enumeración de las máquinas de Turing para encontrar M_i. Finalmente, damos su descripción junto con a^i a una máquina universal de Turing M_u que simula la acción de M sobre a^i. Si a^i está en L, el cálculo realizado por M_u finalmente se detendrá. El efecto combinado de esto es una máquina de Turing que acepta cada $a^i \in L$. Por lo tanto, según la definición 10.1, L es recursivamente enumerable.

■

La demostración de este teorema muestra explícitamente, a través de (10.1), un lenguaje bien definido que no es recursivamente enumerable. Esto no quiere decir que haya una interpretación fácil e intuitiva de $\overline{L}$. Sería difícil exhibir más que unos pocos miembros triviales de este lenguaje. Sin embargo está correctamente definido.

10.1.3. Un lenguaje que es recursivamente enumerable pero no es recursivo

A continuación, mostramos que hay algunos lenguajes que se pueden enumerar de forma recursiva pero que no son recursivos. Una vez más, debemos hacerlo de una manera bastante indirecta. Comenzamos por establecer un resultado subsidiario.

Teorema 10.4 Si un lenguaje L y su complemento $\overline{L}$ son recursivamente enumerables, ambos lenguajes son recursivos. Si L es recursivo, entonces $\overline{L}$ también es recursivo y, en consecuencia, ambos son recursivamente enumerables.

Demostración: Si L y $\overline{L}$ son recursivamente enumerables, entonces existen máquinas de Turing M y $\widehat{M}$ que sirven como procedimientos de enumeración para L y $\overline{L}$, respectivamente. El primero producirá $w_1, w_2, \ldots$ en L, el segundo $\widehat{w}_1, \widehat{w}_2, \ldots$ en $\overline{L}$. Supongamos que ahora se nos da cualquier $w \in \Sigma^+$.

Primero dejamos que M genere w_1 y lo comparamos con w. Si no son iguales, dejamos que $\widehat{M}$ genere $\widehat{w}_1$ y se vuelve a comparar. Si necesitamos continuar, luego dejamos que M genere w_2, luego $\widehat{M}$ genera $\widehat{w}_2$, y así sucesivamente.

Cualquier $w \in \Sigma^+$ será generado por M o $\widehat{M}$, por lo que eventualmente obtendremos una coincidencia. Si la cadena coincidente es producida por M, w pertenece a L, de lo contrario está en $\overline{L}$. El proceso es un algoritmo de pertenencia tanto para L como para $\overline{L}$, por lo que ambos son recursivos.

Por lo contrario, suponga que L es recursivo. Entonces existe un algoritmo de membresía para él. Pero esto se convierte en un algoritmo de pertenencia para $\overline{L}$ simplemente complementando su conclusión. Por tanto, $\overline{L}$ es recursivo. Dado que cualquier lenguaje recursivo es recursivamente enumerable, la prueba está completa.

■

De esto, concluimos directamente que la familia de lenguajes recursivamente enumerables y la familia de lenguajes recursivos no son idénticas. El lenguaje L del Teorema 10.3 pertenece a la primera familia, pero no a la segunda.

Teorema 10.5 Existe un lenguaje recursivamente enumerable que no es recursivo; es decir, la familia de lenguajes recursivos es un subconjunto propio de la familia de lenguajes recursivamente enumerables.

Demostración: Considere el lenguaje L del Teorema 10.3. Este lenguaje es enumerable de forma recursiva, pero su complemento no lo es. Por lo tanto, según el Teorema 10.4, no es recursivo, lo que nos da el ejemplo buscado. ■

Vemos de esto que, de hecho, hay lenguajes bien definidos para los que no se puede construir un algoritmo de pertenencia.

10.1.4. Ejercicios

1. Sea L un lenguaje finito. Demuestre que entonces L^+ es recursivamente enumerable. Sugiera un procedimiento de enumeración para L^+.

2. Sea L un lenguaje libre de contexto. Muestre que L^+ es recursivamente enumerable y sugiera un procedimiento de enumeración para éste.

3. Muestre que si un lenguaje no es recursivamente enumerable, su complemento no puede ser recursivo.

4. Demuestre que la familia de lenguajes recursivamente enumerables es cerrada bajo unión.

5. ¿La familia de lenguajes recursivamente enumerables es cerrada bajo intersección?

6. Demuestre que la familia de lenguajes recursivos es cerrada bajo unión e intersección.

10.2. Gramáticas no restringidas

Para investigar la conexión entre lenguajes y gramáticas recursivamente enumerables, volvemos a la definición general de una gramática en el Capítulo 1. En la Definición 1.1 se permitió que las reglas de producción tomaran cualquier forma, pero luego se establecieron varias restricciones para obtener tipos de gramáticas específicas. Si tomamos la forma general y no imponemos restricciones, obtenemos gramáticas no restringidas.

Definición 10.3 Una gramática $G = (V, T, S, P)$ se llama **no restringida** (**sin restricciones**) si todas las producciones son de la forma

$$u \to v,$$

donde u está en $(V \cup T)^+$ y v está en $(V \cup T)^*$.

En una gramática no restringida, esencialmente no se imponen condiciones a las producciones. Cualquier número de variables y terminales puede estar a la izquierda o a la derecha, y estos pueden ocurrir en cualquier orden. Solo hay una restricción: λ no está permitido como el lado izquierdo de una producción.

Como veremos, las gramáticas no restringidas son mucho más poderosas que las formas restringidas como las gramáticas regulares y libres de contexto que hemos estudiado hasta ahora.

De hecho, las gramáticas no restringidas corresponden a la familia más grande de lenguajes, por lo que podemos esperar reconocerlas por medios mecánicos; es decir, las gramáticas no restringidas generan exactamente la familia de lenguajes enumerables de forma recursiva.

Mostramos esto en dos partes; el primero es bastante sencillo, pero el segundo implica una construcción larga.

Teorema 10.6 Cualquier lenguaje generado por una gramática sin restricciones es enumerable de forma recursiva.

Demostración: La gramática en efecto define un procedimiento para enumerar todas las cadenas en el lenguaje de forma sistemática. Por ejemplo, podemos enumerar todos las w en L de manera que

$$S \Rightarrow w,$$

es decir, w se deriva en un paso. Dado que el conjunto de producciones de la gramática es finito, habrá un número finito de tales cadenas. A continuación, enumeramos todos los w en L que se pueden derivar en dos pasos

$$S \Rightarrow x \Rightarrow w,$$

y así. Podemos simular estas derivaciones en una máquina de Turing y, por lo tanto, tener un procedimiento de enumeración para el lenguaje. Por tanto, es recursivamente enumerable.

■

Esta parte de la correspondencia entre lenguajes enumerables de forma recursiva y gramáticas no restringidas no es sorprendente. La gramática genera cadenas mediante un proceso algorítmico bien definido, por lo que las derivaciones se pueden realizar en una máquina de Turing.

Para mostrar la recíproca, describimos cómo cualquier máquina de Turing puede ser imitada por una gramática sin restricciones.

Se nos da una máquina de Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ y queremos producir una gramática G tal que $L(G) = L(M)$. La idea detrás de la construcción es relativamente simple, pero su implementación se vuelve notoriamente engorrosa.

Dado que el cálculo de la máquina de Turing puede describirse mediante la secuencia de descripciones instantáneas

$$q_0 w \overset{*}{\vdash} x q_f w, \tag{10.3}$$

intentaremos ordenarlo para que la gramática correspondiente tenga la propiedad de que

$$q_0 w \overset{*}{\Rightarrow} x q_f w, \tag{10.4}$$

si y sólo si (10.3) se cumple. Esto no es difícil de hacer; lo que es más difícil de ver es cómo hacer la conexión entre (10.4) y lo que realmente queremos, es decir,

$$S \Rightarrow w$$

para toda w que satisfaga (10.3). Para lograrlo, construimos una gramática que, a grandes rasgos, tiene las siguientes propiedades:

1. S puede derivar $q_0 w$ para toda $w \in \Sigma^+$.
2. (10.4) es posible si y sólo si (10.3) se cumple.
3. Cuando se genera una cadena $x q_f y$ con $q_f \in F$, la gramática transforma esta cadena en la w original.

La secuencia completa de derivaciones es entonces

$$S \overset{*}{\Rightarrow} q_0 w \overset{*}{\Rightarrow} x q_f y \overset{*}{\Rightarrow} w. \tag{10.5}$$

El tercer paso en la derivación anterior es el problemático. ¿Cómo puede la gramática recordar w si se modifica durante el segundo paso? Resolvemos

esto codificando cadenas de modo que la versión codificada originalmente tenga dos copias de w.

La primera se guarda, mientras que la segunda se usa en los pasos de (10.4). Cuando se ingresa una configuración final, la gramática borra todo excepto la w guardada.

Para producir dos copias de w y manejar el símbolo de estado de M (que eventualmente tiene que ser eliminado por la gramática), introducimos las variables V_{ab} y V_{aib} para todo $a \in \Sigma \cup \{\square\}$, $b \in \Gamma$ y todo i tal que $q_i \in Q$. La variable V_{ab} codifica los dos símbolos a y b, mientras que V_{aib} codifica a y b, así como el estado q_i.

El primer paso en (10.5) se puede lograr (en la forma codificada) mediante

$$S \to V_{\square\square}S|SV_{\square\square}|T, \tag{10.6}$$

$$T \to TV_{aa}|V_{a0a}, \tag{10.7}$$

para todo $a \in \Sigma$. Estas producciones permiten que la gramática genere una versión codificada de cualquier cadena q_0w con un número arbitrario de espacios en blanco al principio y al final.

Para el segundo paso, para cada transición

$$\delta(q_i, c) = (q_j, d, R)$$

de M, ponemos en la gramática producciones de la forma

$$V_{aic}V_{pq} \to V_{ad}V_{pjq}, \tag{10.8}$$

para todo $a, p \in \Sigma \cup \{\square\}$, $q \in \Gamma$. Para cada

$$\delta(q_i, c) = (q_j, d, L)$$

de M, incluimos en G

$$V_{pq}V_{aic} \to V_{pjq}V_{ad}, \tag{10.9}$$

para todo $a, p \in \Sigma \cup \{\square\}, q \in \Gamma$.

Si en el segundo paso, M entra en un estado final, la gramática debe entonces deshacerse de todo excepto w, que se guarda en los primeros índices de las V. Por lo tanto, para cada $q_j \in F$, incluimos producciones

$$V_{ajb} \to a, \tag{10.10}$$

para todo $a \in \Sigma \cup \{\square\}, b \in \Gamma$. Esto crea el primer terminal en la cadena, que luego provoca una reescritura en el resto mediante

$$cV_{ab} \to ca, \tag{10.11}$$
$$V_{ab}c \to ac, \tag{10.12}$$

para todo $a, c \in \Sigma \cup \{\square\}, b \in \Gamma$. También necesitamos una producción especial

$$\square \to \lambda. \tag{10.13}$$

Esta última producción se ocupa del caso en que M se mueva fuera de la parte de la cinta ocupada por la entrada w. Para que las cosas funcionen en este caso, primero debemos usar (10.6) y (10.7) para generar

$$\square \cdots \square q_0 w \square \cdots \square,$$

que representa toda la región de la cinta utilizada. Los espacios en blanco extraños se eliminan al final por (10.13).

El siguiente ejemplo ilustra esta complicada construcción. Revise cuidadosamente cada paso en el ejemplo para ver qué hacen las distintas producciones y por qué son necesarias.

Ejemplo 10.1 Sea $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ una máquina de Turing con

$$\begin{aligned} Q &= \{q_0, q_1\}, \\ \Gamma &= \{a, b, \square\}, \\ \Sigma &= \{a, b\}, \\ F &= \{q_1\}, \end{aligned}$$

y

$$\begin{aligned} \delta(q_0, a) &= (q_0, a, R), \\ \delta(q_0, \square) &= (q_1, \square, L). \end{aligned}$$

Esta máquina acepta $L(aa^*)$.

Considere ahora el cálculo

$$q_0aa \vdash aq_0a \vdash aaq_0\square \vdash aq_1a\square, \tag{10.14}$$

que acepta la cadena aa. Para derivar esta cadena con G, primero usamos reglas de la forma (10.6) y (10.7) para obtener la cadena inicial apropiada,

$$S \Rightarrow SV_{\square\square} \Rightarrow TV_{\square\square} \Rightarrow TV_{aa}V_{\square\square} \Rightarrow V_{a0a}V_{aa}V_{\square\square}.$$

La última forma sentencial es el punto de partida para la parte de la derivación que imita el cálculo de la máquina de Turing. Contiene la entrada original $aa\square$ en la secuencia de los primeros índices y la descripción instantánea inicial $q_0aa\square$ en los índices restantes. A continuación, aplicamos

$$V_{a0a}V_{aa} \rightarrow V_{aa}V_{a0a},$$

y

$$V_{a0a}V_{\square\square} \rightarrow V_{aa}V_{\square 0\square},$$

que son instancias específicas de (10.8), y

$$V_{aa}V_{\square 0\square} \rightarrow V_{a1a}V_{\square\square}$$

procedente de (10.9). Entonces los siguientes pasos en la derivación son

$$V_{a0a}V_{aa}V_{\square\square} \Rightarrow V_{aa}V_{a0a}V_{\square\square} \Rightarrow V_{aa}V_{aa}V_{\square 0\square} \Rightarrow V_{aa}V_{a1a}V_{\square}.$$

La secuencia de los primeros índices sigue siendo la misma, siempre recordando la entrada inicial. La secuencia de los otros índices es

$$0aa\square, a0a\square, aa0\square, a1a\square,$$

que es equivalente a la secuencia de descripciones instantáneas en (10.14).

Finalmente, (10.10) a (10.13) se utilizan en los últimos pasos

$$V_{aa}V_{a1a}V_{\square\square} \Rightarrow V_{aa}aV_{\square\square} \Rightarrow V_{aa}a\square \Rightarrow aa\square \Rightarrow aa.$$

La construcción descrita en (10.6) a (10.13) es la base de la prueba del siguiente resultado.

□

Teorema 10.7 Para cada lenguaje L recursivamente enumerable, existe una gramática G sin restricciones, tal que $L = L(G)$.

Demostración: La construcción descrita garantiza que si

$$x \vdash y,$$

entonces

$$e(x) \Rightarrow e(y)$$

donde $e(x)$ denota la codificación de una cadena de acuerdo con la convención dada. Mediante inducción sobre el número de pasos, podemos demostrar que

$$e(q_0 w) \overset{*}{\Rightarrow} e(y)$$

si y sólo si

$$q_0 w \overset{*}{\vdash} y.$$

También debemos mostrar que podemos generar todas las configuraciones iniciales posibles y que w se reconstruye adecuadamente si y sólo si M ingresa a una configuración final. Los detalles, que no son demasiado difíciles, se dejan como ejercicio.

■

10.2.1. Ejercicios

1. ¿Qué lenguaje deriva la siguiente gramática sin restricciones?

$$\begin{aligned} S &\to S_1 B, \\ S_1 &\to a S_1 b, \\ bB &\to bbbB, \\ aS_1 b &\to aa, \\ B &\to \lambda. \end{aligned}$$

2. Construya una máquina de Turing para $L(01(01)^*)$, luego encuentre una gramática sin restricciones para ella usando la construcción del Teorema 10.7. Dé una derivación para 0101 usando la gramática resultante.

10.3. Gramáticas sensibles al contexto y lenguajes

Entre las gramáticas restringidas y las gramáticas sin restricciones, se pueden definir una gran variedad de gramáticas "restringidas en algún sentido".

No todos los casos arrojan resultados interesantes; entre las que sí lo hacen, las gramáticas sensibles al contexto han recibido una atención considerable.

Estas gramáticas generan lenguajes asociados con una clase restringida de máquinas de Turing, los autómatas acotados linealmente, que presentamos en la Sección 9.5.

Definición 10.4 Se dice que una gramática $G = (V, T, S, P)$ es sensible al contexto si todas las producciones son de la forma

$$x \to y,$$

donde $x, y \in (V \cup T)^+$ y

$$|x| \leq |y|. \tag{10.15}$$

Esta definición muestra claramente un aspecto de este tipo de gramáticas; no se contraen, en el sentido de que la longitud de las sucesivas formas sentenciales nunca puede disminuir.

Es menos obvio por qué tales gramáticas deben llamarse sensibles al contexto, pero se puede demostrar (ver, por ejemplo, Salomaa 1973) que todas esas gramáticas pueden reescribirse en una forma normal en la que todas las producciones son de la forma

$$xAy \to xvy.$$

Esto equivale a decir que la producción

$$A \to v$$

se puede aplicar solo en la situación en la que A ocurre en un contexto donde la cadena x está a la izquierda y la cadena y a la derecha.

Si bien usamos la terminología que surge de esta interpretación en particular, la forma en sí es de poco interés para nosotros aquí, y nos basaremos completamente en la Definición 10.4.

10.3.1. Lenguajes sensibles al contexto y autómatas acotados linealmente

Como sugiere la terminología, las gramáticas sensibles al contexto están asociadas con una familia de lenguajes con el mismo nombre.

Definición 10.5 Se dice que un lenguaje L es sensible al contexto si existe una gramática sensible al contexto G, tal que $L = L(G)$ o $L = L(G) \cup \{\lambda\}$.

En esta definición, reintroducimos la cadena vacía. La Definición 10.4 implica que $x \to \lambda$ no está permitida, por lo que una gramática sensible al contexto nunca puede generar un lenguaje que contenga la cadena vacía.

Sin embargo, cada lenguaje libre de contexto sin λ puede ser generado por un caso especial de una gramática sensible al contexto, digamos por una en la forma normal de Chomsky o Greibach, las cuales satisfacen las condiciones de la Definición 10.4.

Al incluir la cadena vacía en la definición de un lenguaje sensible al contexto (pero no en la gramática), podemos afirmar que la familia de lenguajes libres de contexto es un subconjunto de la familia de lenguajes sensibles al contexto.

Ejemplo 10.2 El lenguaje $L = \{a^n b^n c^n : n \geq 1\}$ es un lenguaje sensible al contexto. Demostramos esto exhibiendo una gramática sensible al contexto para el lenguaje. Tal gramática es

$$\begin{aligned} S &\to abc|aAbc, \\ Ab &\to bA, \\ Ac &\to Bbcc, \\ bB &\to Bb, \\ aB &\to aa|aaA. \end{aligned}$$

Podemos ver cómo funciona esto al observar una derivación de $a^3b^3c^3$.

$$\begin{aligned} S &\Rightarrow aAbc \Rightarrow abAc \Rightarrow abBbcc \\ &\Rightarrow aBbbcc \Rightarrow aaAbbcc \Rightarrow aabAbcc \\ &\Rightarrow aabbAcc \Rightarrow aabbBbccc \\ &\Rightarrow aabBbbccc \Rightarrow aaBbbbccc \\ &\Rightarrow aaabbbccc. \end{aligned}$$

La solución utiliza eficazmente las variables A y B como mensajeros. Se crea una A a la izquierda, viaja a la derecha hasta la primera c, donde crea otra b y c. Luego envía al mensajero B de regreso a la izquierda para crear la correspondiente a. El proceso es muy similar a la forma en que se puede programar una máquina de Turing para que acepte el lenguaje L.

□

Dado que el lenguaje del ejemplo anterior no es libre de contexto, vemos que la familia de lenguajes libres de contexto es un subconjunto propio de la familia de lenguajes sensibles al contexto.

El Ejemplo 10.2 también muestra que no es fácil encontrar una gramática sensible al contexto, incluso para ejemplos relativamente simples. A menudo, la solución se obtiene más fácilmente comenzando con un programa de máquina de Turing y luego encontrando una gramática equivalente para él.

Algunos ejemplos mostrarán que, siempre que el lenguaje sea sensible al contexto, la máquina de Turing correspondiente tiene requisitos de espacio predecibles; en particular, puede verse como un autómata acotado linealmente.

Teorema 10.8 Para cada lenguaje sensible al contexto L sin incluir λ, existe algún autómata acotado linealmente M tal que $L = L(M)$.

Demostración: Si L es sensible al contexto, entonces existe una gramática sensible al contexto para $L - \{\lambda\}$. Mostramos que las derivaciones en esta gramática pueden ser simuladas por un autómata acotado linealmente.

El autómata acotado linealmente tendrá dos pistas, una que contiene la cadena de entrada w, la otra que contiene las formas sentenciales derivadas usando G. Un punto clave de este argumento es que ninguna forma sentencial puede tener una longitud mayor que $|w|$. Otro punto a tener en cuenta es que un autómata acotado linealmente es, por definición, no determinista.

Esto es necesario en el argumento, ya que podemos afirmar que siempre se puede adivinar la producción correcta y que no es necesario buscar alternativas improductivas. Por lo tanto, el cálculo descrito en el Teorema 10.6 se puede realizar sin usar espacio excepto el que originalmente ocupaba w; es decir, se puede realizar mediante un autómata acotado linealmente.

■

Teorema 10.9 Si un lenguaje L es aceptado por algún autómata acotado linealmente M, entonces existe una gramática sensible al contexto que genera a L.

Demostración: La construcción aquí es similar a la del Teorema 10.7. Todas las producciones generadas en el Teorema 10.7 no se contraen excepto (10.13),

$$\square \to \lambda$$

Pero esta producción puede omitirse. Sólo es necesaria cuando la máquina de Turing se mueve fuera de los límites de la entrada original, lo cual no es el caso aquí. La gramática obtenida por la construcción sin esta producción innecesaria no se contrae, completando el argumento.

■

10.3.2. Relación entre lenguajes recursivos y sensibles al contexto

El teorema 10.9 nos dice que cada lenguaje sensible al contexto es aceptado por alguna máquina de Turing y, por lo tanto, es recursivamente enumerable. El teorema 10.10 se sigue fácilmente de esto.

Teorema 10.10 Todo lenguaje L sensible al contexto es recursivo.

Demostración: Considere el lenguaje sensible al contexto L con una gramática sensible al contexto asociada G, y observe una derivación de w

$$S \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \cdots \Rightarrow x_n \Rightarrow w.$$

Podemos suponer sin ninguna pérdida de generalidad que todas las formas sentenciales en una sola derivación son diferentes; es decir, $x_i \neq x_j$ para todo $i \neq j$. El quid de nuestro argumento es que el número de pasos en cualquier derivación es una función acotada de $|w|$. Sabemos que

$$|x_j| \leq |x_{j+1}|,$$

porque G no se contrae. Lo único que necesitamos agregar es que existe algún m, dependiendo sólo de G y w, de manera que

$$|x_j| < |x_{j+m}|,$$

para todo j, con $m = m(|w|)$ una función acotada de $|V \cup T|$ y $|w|$. Esto se debe a que la finitud de $|V \cup T|$ implica que sólo hay un número finito de cadenas de una longitud determinada. Por lo tanto, la longitud de una derivación de $w \in L$ es como máximo $|w|m(|w|)$.

Esta observación nos da inmediatamente un algoritmo de pertenencia para L. Comprobamos todas las derivaciones de longitud hasta $|w|m(|w|)$. Dado que el conjunto de producciones de G es finito, solo hay un número finito de estos. Si alguno de ellos da w, entonces $w \in L$, de lo contrario no está en L.

■

Teorema 10.11 Existe un lenguaje recursivo que no es sensible al contexto.

Demostración: Considere el conjunto de todas las gramáticas sensibles al contexto sobre $T = \{a, b\}$. Podemos usar una convención en la que cada gramática tiene un conjunto de variables de la forma

$$V = \{V_0, V_1, V_2, \ldots\}.$$

Cada gramática sensible al contexto está completamente especificada por sus producciones; podemos pensar en ellas como si estuvieran escritas en una sola cadena

$$x_1 \to y_1; x_2 \to y_2; \ldots; x_m \to y_m.$$

A esta cadena aplicamos ahora el homomorfismo

$$\begin{aligned} h(a) &= 010, \\ h(b) &= 01^20, \\ h(\to) &= 01^30, \\ h(;) &= 01^40, \\ h(V_i) &= 01^{i+5}0. \end{aligned}$$

Por lo tanto, cualquier gramática sensible al contexto se puede representar de forma única mediante una cadena de $L((011^*0)^*)$. Además, la representación es invertible en el sentido de que, dada dicha cadena, hay como máximo una gramática sensible al contexto correspondiente.

Introduzcamos un orden propio en $\{0, 1\}^+$, de modo que podamos escribir cadenas en el orden w_1, w_2, etc. Una cadena dada w_j puede no definir una gramática sensible al contexto; pero si lo hace, llame a la gramática G_j.

A continuación, definimos un lenguaje L mediante

$$L = \{w_i : w_i \text{ define una gramática sensible al contexto } G_i \text{ y } w_i \notin L(G_i)\}.$$

L está bien definido y de hecho es recursivo.

Para ver esto, construimos un algoritmo de pertenencia. Dado w_i, comprobamos para ver si define un gramatica sensible al contexto G_i. Si no, entonces $w_i \notin L$. Si la cadena define una gramática, entonces $L(G_i)$ es recursivo, y podemos usar el algoritmo de pertenencia del Teorema 10.10 para averiguar si $w_i \in L(G_i)$. Si no es así, entonces w_i pertenece a L.

Pero L no es sensible al contexto. Si lo fuera, existiría un w_j tal que $L = L(G_j)$. Entonces podemos preguntar si w_j está en $L(G_j)$. Si asumimos que $w_j \in L(G_j)$, entonces, por definición, w_j no está en L. Pero $L = L(G_j)$, entonces tenemos una contradicción. Por el contrario, si asumimos que $w_j \notin L(G_j)$, entonces, por definición, $w_j \in L$ y tenemos otra contradicción. Por lo tanto, debemos concluir que L no es sensible al contexto. ■

El resultado del Teorema 10.11 indica que los autómatas acotados linealmente son de hecho menos poderosos que las máquinas de Turing, ya que sólo aceptan un subconjunto propio de los lenguajes recursivos.

Se sigue del mismo resultado que los autómatas acotados linealmente son más poderosos que los autómatas de pila. Los lenguajes libres de contexto, que se generan mediante gramáticas libres de contexto, son un subconjunto de los lenguajes sensibles al contexto. Como muestran varios ejemplos, son un subconjunto propio.

Debido a la equivalencia esencial de los autómatas acotados linealmente y los lenguajes sensibles al contexto por un lado, y los autómatas de pila y los lenguajes libres de contexto por el otro, vemos que cualquier lenguaje aceptado por un autómata de pila también es aceptado por algún autómata acotado linealmente, pero que hay lenguajes aceptados por algunos autómatas acotados linealmente para los que no hay autómatas de pila.

10.3.3. Ejercicios

Construya gramáticas sensibles al contexto para los siguientes lenguajes:

1. $L = \{a^{n+1}b^nc^{n-1} : n \geq 1\}$.
2. $L = \{a^nb^na^{2n} : n \geq 1\}$.
3. $L = \{a^nb^mc^nd^m : n \geq 1, m \geq 1\}$.
4. $L = \{ww : w \in \{a, b\}^+\}$.
5. $L = \{a^nb^nc^nd^n : n \geq 1\}$.

10.4. La Jerarquía de Chomsky

Ahora nos hemos encontrado con una serie de familias de lenguajes, entre ellos los lenguajes recursivamente enumerables (L_{RE}), los lenguajes sensibles al contexto (L_{CS}), los lenguajes libres de contexto (L_{CF}) y los lenguajes regulares (L_{REG}).

Una forma de mostrar la relación entre estas familias es mediante la **Jerarquía de Chomsky**. Noam Chomsky, uno de los fundadores de la teoría de lenguajes formales, proporcionó una clasificación inicial con cuatro tipos de lenguajes, del tipo 0 al tipo 3. Esta terminología original ha persistido y se encuentran frecuentes referencias a ella, pero los tipos numéricos son en realidad nombres diferentes para las familias de lenguajes que hemos estudiado.

Los lenguajes de tipo 0 son los generados por gramáticas sin restricciones, es decir, los lenguajes enumerables recursivamente. El tipo 1 consta de los lenguajes sensibles al contexto, el tipo 2 consta de los lenguajes libres de contexto y el tipo 3 consta de los lenguajes regulares.

Como hemos visto, cada familia de lenguajes de tipo i es un subconjunto propio de la familia de tipo $i - 1$. La figura 10.3 muestra la jerarquía de Chomsky original.

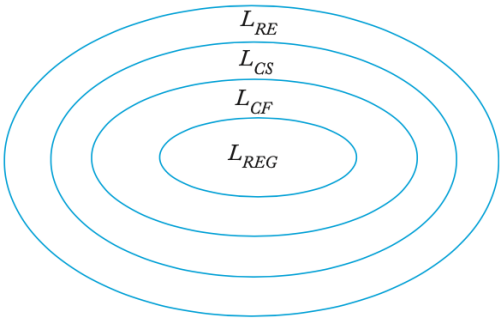


Figura 10.3: Jerarquía de Chomsky original.

También hemos conocido otras familias de lenguajes que pueden encajar en esta imagen. Incluyendo las familias de lenguajes deterministas libres de contexto (L_{DCF}) y lenguajes recursivos (L_{REC}), llegamos a la jerarquía extendida que se muestra en la Figura 10.4.

Para resumir, hemos explorado las relaciones entre varias familias de lenguajes y sus autómatas asociados. Al hacerlo, establecimos una jerarquía de

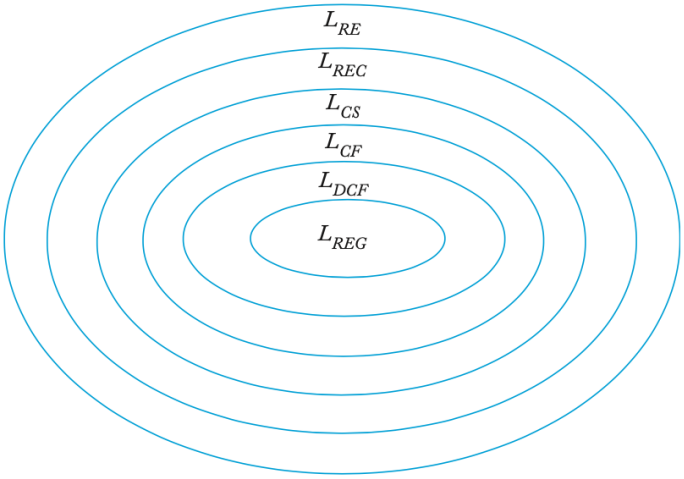


Figura 10.4: Jerarquía de Chomsky extendida.

lenguajes y clasificamos a los autómatas por su poder como aceptadores de lenguajes.

Las máquinas de Turing son más poderosas que los autómatas acotados linealmente. Estos, a su vez, son más poderosos que los autómatas de pila. En la parte inferior de la jerarquía se encuentran los aceptadores finitos, con los que comenzamos nuestro estudio.

Capítulo 11

Límites del Cálculo Algorítmico

Hemos hablado sobre lo que pueden hacer las máquinas de Turing, ahora veremos lo que no pueden hacer. Aunque la tesis de Turing nos lleva a creer que existen pocas limitaciones en el poder de una máquina de Turing, hemos afirmado en varias ocasiones que no podría existir ningún algoritmo para la solución de ciertos problemas.

Ahora hacemos más explícito lo que queremos decir con esta afirmación. Algunos de los resultados se obtuvieron de forma bastante sencilla; si un lenguaje no es recursivo, entonces, por definición, no existe un algoritmo de pertenencia para él. Si esto fuera todo lo que hay en este tema, no sería muy interesante; los lenguajes no recursivos tienen poco valor práctico.

Pero el problema es más profundo. Por ejemplo, hemos declarado (pero aún no probado) que no existe un algoritmo para determinar si una gramática libre de contexto no es ambigua. Esta pregunta es claramente de importancia práctica en el estudio de los lenguajes de programación.

Primero definimos los conceptos de decidibilidad y computabilidad para precisar lo que queremos decir cuando decimos que una máquina de Turing no puede hacer algo. A continuación, examinamos varios problemas clásicos de este tipo, entre ellos el conocido problema de paro de las máquinas de Turing. De esto se siguen una serie de problemas relacionados con las máquinas de Turing y los lenguajes enumerables de forma recursiva.

11.1. Algunos problemas que no se pueden resolver con Máquinas de Turing

El argumento de que el poder de los cálculos mecánicos es limitado no es sorprendente. Intuitivamente sabemos que muchas preguntas vagas y especulativas requieren una comprensión y un razonamiento especiales mucho más allá de la capacidad de cualquier computadora que ahora podamos construir o incluso prever.

Lo que es más interesante para los científicos de la computación es que hay cuestiones que pueden plantearse de forma clara y sencilla, con una aparente posibilidad de una solución algorítmica, pero que se sabe que no pueden ser resueltas por ninguna computadora.

11.1.1. Computabilidad y decidibilidad

Dijimos que se dice que una función f en un dominio determinado es computable si existe una máquina de Turing que calcula el valor de f para todos los argumentos en su dominio. Una función no es computable si no existe tal máquina de Turing.

Puede haber una máquina de Turing que pueda calcular f en parte de su dominio, pero llamamos a la función computable sólo si hay una máquina de Turing que calcula la función en la totalidad de su dominio. Vemos de esto que, cuando clasificamos una función como computable o no computable, debemos tener claro cuál es su dominio.

Nuestra preocupación aquí será la configuración algo simplificada donde el resultado de un cálculo es un simple "sí" o "no". En este caso, hablamos de que un problema es **decidible** o **indecidible**.

Por problema entenderemos un conjunto de afirmaciones relacionadas, cada una de las cuales debe ser verdadera o falsa. Por ejemplo, consideramos la afirmación "Para una gramática G libre de contexto, el lenguaje $L(G)$ es ambiguo". Para algunas G esto es cierto, para otras es falso, pero claramente debemos tener uno u otro. El problema es decidir si el enunciado es verdadero para cualquier G que se nos dé.

Nuevamente, hay un dominio subyacente, el conjunto de todas las gramáticas libres de contexto. Decimos que un problema es decidible si existe una máquina de Turing que da la respuesta correcta para cada enunciado en el dominio del problema.

Cuando enunciamos resultados de decidibilidad o indecidibilidad, siempre debemos saber cuál es el dominio, porque esto puede afectar la conclusión. El problema puede resolverse en algún dominio pero no en otro.

Específicamente, una sola instancia de un problema siempre es decidible, ya que la respuesta es verdadera o falsa. En el primer caso, una máquina de Turing que siempre responde "verdadero"da la respuesta correcta, mientras que en el segundo caso es apropiada una que siempre responde "falso".

Esto puede parecer una respuesta graciosa, pero enfatiza un punto importante. El hecho de que no sepamos cuál es la respuesta correcta no hace ninguna diferencia; lo que importa es que existe alguna máquina de Turing que da la respuesta correcta.

11.1.2. El problema de paro de la Máquina de Turing

Partimos de algunos problemas que tienen trascendencia histórica y que a la vez nos dan un punto de partida para desarrollar resultados posteriores. El más conocido de ellos es el problema de paro de la máquina de Turing.

En pocas palabras, el problema es: Dada la descripción de una máquina de Turing M y una entrada w, ¿M, cuando se inicia en la configuración inicial q_0w, realiza un cálculo que finalmente se detiene?

Usando una forma abreviada de hablar sobre el problema, preguntamos si M aplicada a w, o simplemente (M, w), se detiene o no. El dominio de este problema debe tomarse como el conjunto de todas las máquinas de Turing y todas las w; es decir, estamos buscando una sola máquina de Turing que, dada la descripción de una M y w arbitrarias, predecirá si el cálculo de M aplicada a w se detendrá o no.

No podemos encontrar la respuesta simulando la acción de M sobre w, digamos realizándola en una máquina de Turing universal, porque no hay límite en la duración del cálculo. Si M entra en un ciclo infinito, no importa cuánto esperemos, nunca podremos estar seguros de que M está de hecho en un ciclo. Puede ser simplemente un caso de un cálculo muy largo.

Lo que necesitamos es un algoritmo que pueda determinar la respuesta correcta para cualquier M y w realizando un análisis de la descripción de la máquina y la entrada. Pero, como mostraremos, no existe tal algoritmo.

Para una discusión posterior, es conveniente tener una idea precisa de lo que entendemos por problema de paro; por esta razón, hacemos una definición específica de lo que dijimos en forma un tanto vaga.

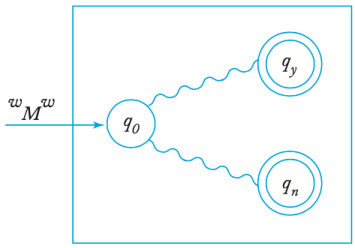


Figura 11.1: H se inicia en el estado q_0 con la entrada $w_M w$ y eventualmente se detiene en el estado q_y o q_n.

Definición 11.1 Sea w_M una cadena que describe una máquina de Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$, y sea w una cadena en el alfabeto de M. Supondremos que w_M y w están codificados como una cadena de ceros y unos, como se sugiere en la Sección 9.4. Una solución del problema de paro es una máquina de Turing H, que para cualquier w_M y w realiza el cálculo

$$q_0 w_M w \overset{*}{\vdash} x_1 q_y x_2$$

si M aplicada a w para, y

$$q_0 w_M w \overset{*}{\vdash} y_1 q_n y_2$$

si M aplicada a w no para. Aquí q_y y q_n son estados finales de H.

Teorema 11.1 No existe ninguna máquina de Turing H que se comporte como lo requiere la Definición 11.1. Por tanto, el problema de paro es indecidible.

Demostración: Suponemos lo contrario, es decir, que existe un algoritmo y, en consecuencia, alguna máquina de Turing H, que resuelve el problema de paro. La entrada a H será la cadena $w_M w$.

El requisito es entonces que, dado cualquier $w_M w$, la máquina de Turing H se detendrá con una respuesta de sí o no. Logramos esto pidiendo que H se detenga en uno de los dos estados finales correspondientes, digamos, q_y o q_n. La situación se puede visualizar mediante un diagrama de bloques como el de la Figura 11.1.

La intención de este diagrama es indicar que, si H se inicia en el estado q_0 con la entrada $w_M w$, eventualmente se detendrá en el estado q_y o q_n.

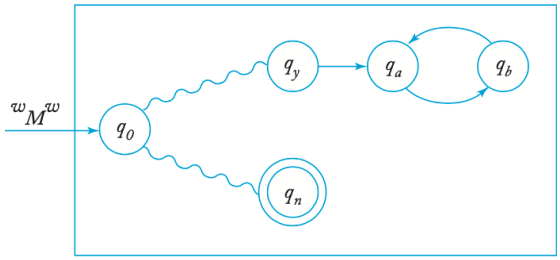


Figura 11.2: H' se comporta como H excepto que cuando alcanza el estado q_y independientemente de lo que haya en la cinta no lo cambia y se cicla.

Como lo requiere la Definición 11.1, queremos que H opere de acuerdo con las siguientes reglas:

$$q_0 w_M w \overset{*}{\vdash}_H x_1 q_y x_2$$

si M aplicada a w para, y

$$q_0 w_M w \overset{*}{\vdash}_H y_1 q_n y_2$$

si M aplicada a w no para.

A continuación, modificamos H para producir una máquina de Turing H' con la estructura que se muestra en la Figura 11.2. Con los estados agregados en la Figura 11.2 queremos expresar que las transiciones entre el estado q_y y los nuevos estados q_a y q_b deben realizarse, independientemente del símbolo de la cinta, de tal manera que la cinta permanezca sin cambios.

La forma en que se hace esto es sencilla. Comparando H y H' vemos que, en situaciones donde H alcanza q_y y se detiene, la máquina modificada H' entrará en un ciclo infinito. Formalmente, la acción de H' se describe por

$$q_0 w_M w \overset{*}{\vdash}_{H'} \infty$$

si M aplicada a w para, y

$$q_0 w_M w \overset{*}{\vdash}_{H'} y_1 q_n y_2$$

si M aplicada a w no para.

A partir de H' construimos otra máquina de Turing $\widehat{H}$. Esta nueva máquina toma como entrada w_M y la copia, terminando en su estado inicial q_0.

Después de eso, se comporta exactamente como H'. Entonces la acción de $\widehat{H}$ es tal que

$$q_0 w_M \overset{*}{\vdash}_{\widehat{H}} q_0 w_M w_M \overset{*}{\vdash}_{\widehat{H}} \infty$$

si M aplicada a w_M para, y

$$q_0 w_M \overset{*}{\vdash}_{\widehat{H}} q_0 w_M w_M \overset{*}{\vdash}_{\widehat{H}} y_1 q_n y_2$$

si M aplicada a w_M no para.

Ahora, $\widehat{H}$ es una máquina de Turing, por lo que tiene una descripción en $\{0,1\}^*$, digamos, $\widehat{w}$. Esta cadena, además de ser la descripción de $\widehat{H}$, también se puede utilizar como cadena de entrada. Por lo tanto, podemos preguntarnos legítimamente qué sucedería si $\widehat{H}$ se aplicara a $\widehat{w}$. De lo anterior, identificando M con $\widehat{H}$, obtenemos

$$q_0 \widehat{w} \overset{*}{\vdash}_{\widehat{H}} \infty$$

si $\widehat{H}$ aplicada a $\widehat{w}$ para, y

$$q_0 \widehat{w} \overset{*}{\vdash}_{\widehat{H}} y_1 q_n y_2$$

si $\widehat{H}$ aplicada a $\widehat{w}$ no para.

Esto es claramente un sinsentido. La contradicción nos dice que nuestra suposición de la existencia de H, y por tanto la suposición de la decidibilidad del problema de paro, debe ser falsa.

■

Uno puede objetar la Definición 11.1, ya que requerimos que, para resolver el problema de paro, H tuviera que comenzar y terminar en configuraciones muy específicas.

Sin embargo, no es difícil ver que estas condiciones elegidas arbitrariamente juegan sólo un papel menor en el argumento, y que esencialmente el mismo razonamiento podría usarse con cualquier otra configuración inicial y final. Hemos vinculado el problema a una definición específica por el bien de la discusión, pero esto no afecta la conclusión.

Es importante tener en cuenta lo que dice el Teorema 11.1. No excluye la solución del problema de paro para casos específicos; a menudo podemos decir mediante un análisis de M y w si la máquina de Turing parará o no. Lo que dice el teorema es que esto no siempre se puede hacer; no existe un algoritmo que pueda tomar una decisión correcta para todas las w_M y w.

Los argumentos para probar el Teorema 11.1 se dieron porque son clásicos y de interés histórico. La conclusión del teorema está en realidad implícita en resultados anteriores, como muestra el siguiente argumento.

Teorema 11.2 Si el problema de paro fuera decidible, entonces todo lenguaje enumerable recursivamente sería recursivo. En consecuencia, el problema de paro es indecidible.

Demostración: Para ver esto, sea L un lenguaje recursivamente enumerable sobre Σ, y sea M una máquina de Turing que acepta L. Sea H la máquina de Turing que resuelve el problema de paro. Construimos a partir de esto el siguiente procedimiento:

1. Aplique H a $w_M w$. Si H dice "no", entonces, por definición, w no está en L.

2. Si H dice "sí", aplique M a w. Pero M debe parar, entonces eventualmente nos dirá que w está en L.

Esto constituye un algoritmo de pertenencia que hace que L sea recursivo. Pero ya sabemos que hay lenguajes enumerables recursivamente que no son recursivos. La contradicción implica que H no puede existir, es decir, que el problema de paro es indecidible.

■

11.1.3. Reducción de un problema indecidible a otro

El argumento anterior, que conecta el problema de paro con el problema de membresía, ilustra la técnica de reducción que es muy importante en Teoría de la Computación. Decimos que un problema A se reduce a un problema B si la decidibilidad de A se deriva de la decidibilidad de B. Entonces, si sabemos que A es indecidible, podemos concluir que B también es indecidible. Veamos algunos ejemplos para ilustrar esta idea.

Ejemplo 11.1 El **problema de entrar en un estado** es el siguiente. Dada cualquier máquina de Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ y cualquier $q \in Q$, $w \in \Sigma^+$, decida si se entra o no en el estado q cuando M se aplica a w. Este problema es indecidible.

Para reducir el problema de paro al problema de entrar en un estado, suponga que tenemos un algoritmo A que resuelve el problema de entrar en

un estado. Entonces podríamos usarlo para resolver el problema de paro. Por ejemplo, dado cualquier M y w, primero modificamos M para obtener $\widehat{M}$ de tal manera que $\widehat{M}$ se detenga en el estado q si y sólo si M se detiene.

Podemos hacer esto simplemente mirando la función de transición δ de M. Si M se detiene, lo hace porque alguna $\delta(q_i, a)$ no está definida. Para obtener $\widehat{M}$, cambiamos cada δ indefinida a

$$\delta(q_i, a) = (q, a, R),$$

donde q es un estado final. Aplicamos el algoritmo de entrar en un estado A a $\left(\widehat{M}, q, w\right)$. Si A responde sí, es decir, se ingresa al estado q, entonces (M, w) se detiene. Si A dice que no, entonces (M, w) no se detiene.

Por tanto, la suposición de que el problema de entrar en un estado es decidible nos da un algoritmo para el problema de paro. Debido a que el problema de paro es indecidible, el problema de entrar en un estado también tiene que ser indecidible.

□

Ejemplo 11.2 El **problema de paro de la cinta en blanco** es otro problema al que se puede reducir al problema de paro. Dada una máquina de Turing M, determine si M se detiene o no si se inicia con una cinta en blanco. Esto es indecidible.

Para mostrar cómo se logra esta reducción, suponga que se nos da alguna M y alguna w. Primero construimos a partir de M una nueva máquina M_w que comienza con una cinta en blanco, escribe w en ella y luego se posiciona en una configuración q_0w. Después de eso, M_w actúa como M. Claramente, M_w se detendrá con una cinta en blanco si y sólo si M se detiene con w.

Supongamos ahora que el problema de paro de la cinta en blanco fuera decidible. Dado cualquier (M, w), primero construimos M_w, luego le aplicamos el algoritmo del problema de paro de la cinta en blanco. La conclusión nos dice si M aplicada a w se detendrá.

Dado que esto se puede hacer para cualquier M y w, un algoritmo para el problema de paro de la cinta en blanco se puede convertir en un algoritmo para el problema de paro. Dado que se sabe que este último es indecidible, lo mismo debe ser cierto para el problema de paro de la cinta en blanco.

□

La construcción en los argumentos de estos dos ejemplos ilustra un enfoque común para establecer resultados de indecidibilidad. Un diagrama de

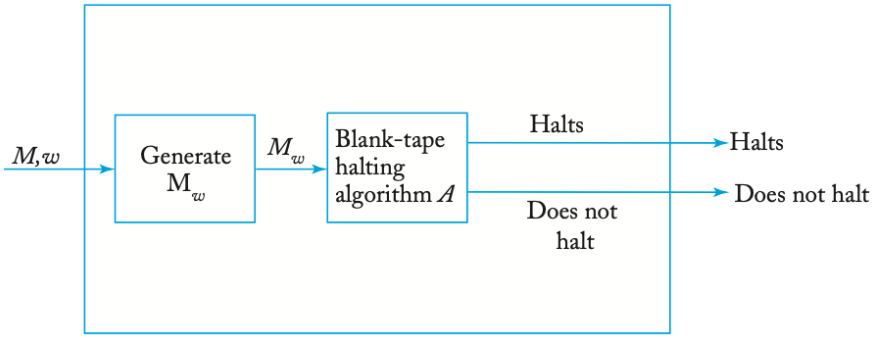


Figura 11.3: Algoritmo para el problema de paro.

bloques a menudo nos ayuda a visualizar el proceso. La construcción del ejemplo 11.2 se resume en la Figura 11.3. En ese diagrama, primero usamos un algoritmo que transforma (M, w) en M_w; tal algoritmo existe claramente.

A continuación, usamos el algoritmo para resolver el problema de paro de la cinta en blanco, que asumimos que existe. Poner los dos juntos produce un algoritmo para el problema de paro. Pero esto es imposible y podemos concluir que A no puede existir.

Un problema de decisión es efectivamente una función con un rango $\{0, 1\}$, es decir, una respuesta verdadera o falsa. También podemos mirar funciones más generales para ver si son computables; para ello, seguimos el método establecido y reducimos el problema de paro (o cualquier otro problema conocido como indecidible) al problema de calcular la función en cuestión.

Debido a la tesis de Turing, esperamos que las funciones que se encuentran en circunstancias prácticas sean computables, por lo que para obtener ejemplos de funciones no computables debemos buscar un poco más. La mayoría de los ejemplos de funciones no computables se asocian con intentos de predecir el comportamiento de las máquinas de Turing.

Ejemplo 11.3 Sea $\Gamma = \{0, 1, \square\}$. Considere la función $f(n)$ cuyo valor es el número máximo de movimientos que puede realizar cualquier máquina de Turing de n estados que se detiene cuando se inicia con una cinta en blanco. Esta función no es computable.

Antes de que nos propongamos demostrar esto, asegurémonos de que $f(n)$ esté definida para todo n. Observe primero que sólo hay un número finito de máquinas de Turing con n estados. Esto se debe a que Q y Γ son finitos, por lo que δ tiene un dominio y un rango finitos. Esto, a su vez, implica que

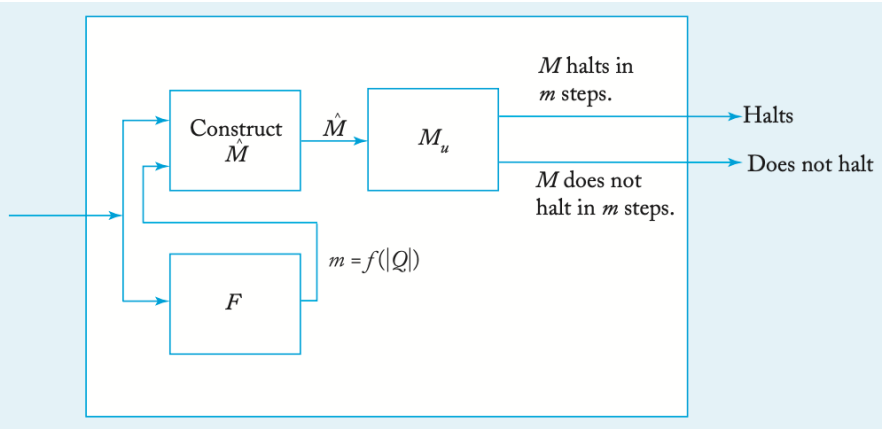


Figura 11.4: Algoritmo para el problema de paro de la cinta en blanco.

sólo hay un número finito de δ diferentes y, por lo tanto, un número finito de máquinas de Turing de n estados diferentes.

De todas las máquinas de n estados, hay algunas que siempre se detienen, por ejemplo, las máquinas que sólo tienen estados finales y, por lo tanto, no hacen ningún movimiento. Algunas de las máquinas de n estados no se detienen cuando se inician con una cinta en blanco, pero no entran en la definición de f. Cada máquina que se detiene ejecutará un cierto número de movimientos; de estos, tomamos el más grande para dar $f(n)$.

Tome cualquier máquina de Turing M y número positivo m. Es fácil modificar M para producir $\widehat{M}$ de tal manera que este último siempre se detendrá con una de dos respuestas: M aplicada a una cinta en blanco se detiene en no más de m movimientos, o M aplicada a una cinta en blanco hace más de m movimientos.

Todo lo que tenemos que hacer para esto es hacer que M cuente sus movimientos y termine cuando este recuento exceda m. Suponga ahora que $f(n)$ es calculable por alguna máquina de Turing F. Entonces podemos poner $\widehat{M}$ y F juntos como se muestra en la Figura 11.4. Primero calculamos $f(|Q|)$, donde Q es el conjunto de estados de M. Esto nos dice el número máximo de movimientos que M puede hacer si se detiene.

El valor que obtenemos se usa entonces como m para construir $\widehat{M}$ como se describe, y se da una descripción de $\widehat{M}$ a una máquina de Turing universal para su ejecución. Esto nos dice si M aplicada a una cinta en blanco se detiene o no en menos de $f(|Q|)$ pasos.

Si encontramos que M aplicada a una cinta en blanco hace más de $f(|Q|)$ movimientos, entonces, debido a la definición de f, la implicación es que M nunca se detiene. Por tanto, tenemos una solución al problema de paro de la cinta en blanco. La imposibilidad de la conclusión nos obliga a aceptar que f no es computable.

□

11.1.4. Ejercicios

1. Muestre que la pregunta "¿Acepta una máquina de Turing todas las cadenas de longitud par?" es indecidible.

2. Demuestre que el siguiente problema es indecidible. Dada cualquier máquina de Turing M, $a \in \Gamma$ y $w \in \Sigma^+$, determine si el símbolo a se escribe alguna vez cuando M se aplica a w.

3. Demuestre que no existe un algoritmo para decidir si una máquina de Turing arbitraria se detiene o no en todas las entradas.

4. Muestre que no existe un algoritmo para decidir si dos máquinas de Turing M_1 y M_2 aceptan el mismo lenguaje.

5. Muestre que el problema de determinar si una máquina de Turing para con cualquier entrada es indecidible.