J. Glenn Brookshear

TEORÍA DE LA COMPUTACIÓN

Lenguajes formales, autómatas y complejidad

ADDISON-WESLEY IBEROAMERICANA

TEORÍA DE LA COMPUTACIÓN

Lenguajes formales, autómatas y complejidad

TEORÍA DE LA COMPUTACIÓN

Lenguajes formales, autómatas y complejidad

J. Glenn Brookshear
Marquette University

Versión en español de
Ernesto Morales Peake
México, D.F.

Con la colaboración de
Jesús Luis Díaz Labrador
Universidad de Deusto
Bilbao, España
y
Guillermo Levine
Universidad de Guadalajara
Jalisco, México

ADDISON-WESLEY IBEROAMERICANA
Argentina • Brasil • Chile • Ecuador • España
Estados Unidos • México • Perú • Puerto Rico • Venezuela

Versión en español de la obra titulada *Theory of Computation. Formal Languages, Automata, and Complexity*, de J. Glenn Brookshear, publicada originalmente en inglés por The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, E.U.A. © 1989

Esta edición en español es la única autorizada.

Portada: Pintor Claudio Cué, *Cempasúchil, la flor de los muertos.* México, 1992 (100 x 80 cm), técnica mixta sobre tela. Colección particular. Reproducido con autorización.

© 1993 por **Addison-Wesley Iberoamericana, S.A.**
Wilmington, Delaware, E.U.A.

Impreso en Estados Unidos. *Printed in U.S.A.*

ISBN 0-201-60119-2
12345678910-AL-96959493

A Earlene y Cheryl

PREFACIO

He diseñado este libro con el fin de que sirva como texto para un curso introductorio de un semestre de teoría de la computación. El libro cubre los temas tradicionales de lenguajes formales, autómatas, computabilidad y complejidad computacional. Así, esta obra constituiría un texto adecuado para cursos como el CS16 del currículum 1987 del ACM y el CO3 del currículum de ciencias de la computación del ACM (*Communications of the ACM*, marzo de 1986).

FILOSOFÍA SUBYACENTE

Mi objetivo al escribir este libro fue presentar los fundamentos de la ciencia teórica de la computación de una manera accesible para los estudiantes universitarios de ciencias de la computación e informática. Mi deseo fue que los estudiantes vieran las ideas teóricas como la base para la resolución de problemas reales, no como abstracciones inútiles.

Me dediqué a esta tarea adoptando dos principios básicos. El primero fue subrayar la relación entre los aspectos teóricos que se presentan y los aspectos prácticos con los que están familiarizados los estudiantes de este nivel. Si no se hiciera hincapié en esto, los estudiantes cuyos antecedentes consisten en programación, ingeniería de software y estructuras de datos tendrían problemas para establecer una relación con la teoría de autómatas, los lenguajes formales y la teoría de funciones recursivas. Por consiguiente, los aspectos teóricos del libro se vinculan estrechamente a los temas prácticos. Por ejemplo, la jerarquía de la teoría de autómatas y de los lenguajes formales se plantea en el contexto de la construcción de rutinas de análisis sintáctico bien diseñadas para un compilador. Como otro ejemplo, el tema de la computabilidad se relaciona con la búsqueda de las estructuras que se requieren en un lenguaje de programación para asegurar que un programa en dicho lenguaje pueda resolver cualquier problema que tenga una solución algorítmica.

El otro principio al cual me he ajustado al escribir este libro fue evitar los aspectos de menor importancia. Se trata de un texto para un curso introductorio de un semestre, no de una enciclopedia sobre el tema. Mi deseo es que los estudiantes vean el panorama general, sin que se enreden en el estudio de los detalles. Por ejemplo, esta perspectiva se pierde si uno trata de presentar todos los temas relacionados con los lenguajes regulares antes de pasar a la jerarquía de lenguajes. Si se avanza con demasiada lentitud, los estudiantes pierden su perspectiva del tema.

Por lo tanto, he decidido evitar los aspectos que no contribuyen de manera directa a la perspectiva global y que no son material previo necesario para abordar otros temas del libro. Así, hay material suficiente para un curso introductorio de un semestre, al mismo tiempo que es posible avanzar con rapidez por el texto, de manera que su estructura no se oculte debido a los detalles; es posible cubrir los aspectos detallados en otros cursos. En resumen, este libro analiza los aspectos siguientes:

- lenguajes regulares, independientes del contexto y generales estructurados por frases, junto con los autómatas relacionados con ellos;
- la computabilidad en el contexto de las máquinas de Turing, las funciones recursivas parciales y los lenguajes de programación sencillos; y
- la teoría de la complejidad, con una introducción a algunos de los problemas de clasificación pendientes relacionados con las clases *P* y *NP*.

APOYOS PEDAGÓGICOS

Clasifico los apoyos pedagógicos como implícitos o explícitos. En lo referente a la primera categoría, he tratado de proporcionar más explicaciones que demostraciones. Por supuesto, presento demostraciones rigurosas en el texto, pero también he incluido una cantidad considerable de diálogos y motivación intuitiva. Mi objetivo es que los estudiantes no sólo conozcan el material, sino que lo aprendan, por lo que el libro contiene más presentaciones informales que las que normalmente se encuentran en un formato de definición, teorema y demostración.

En lo que corresponde a la categoría de apoyos pedagógicos explícitos, incluyo, al final de cada capítulo, una sección llamada "Comentarios finales", donde repaso el material y analizo las interrelaciones. Asimismo, he incluido gran cantidad de ejercicios en el texto, y cada sección (después del capítulo introductorio) concluye con algunos ejercicios diseñados para repasar el material recién analizado. Después de cada capítulo presento diversos problemas, los cuales no sólo repasan el material del capítulo sino que se relacionan además con el material de otros capítulos e incluyen conceptos que no se presentan en el texto.

AGRADECIMIENTOS

Son varias las personas que han contribuido a la elaboración de este libro. Doy gracias a todos ellos, en especial a las siguientes personas, quienes actuaron como revisores: J. Mack Adams, Eric Bach, T. M. Phillips, Donald Friesen, Tim Long, Matthias Stallmann, George Struble y Allen Tucker. Además, Richard Iltis utilizó una de las primeras versiones del manuscrito para su cátedra y proporcionó muchos comentarios de gran utilidad. Las discusiones con Walter Savitch ayudaron a eliminar varias partes difíciles del texto final.

También quiero agradecer a Paul Bankston, Mark Barnard, Phillip Bender, Madhu Deshpande, Douglas Harris, Gary Krenz, John Moyer, Francis Pastijn, John Simms y Michael Slattery, miembros de mi departamento en Marquette University, quienes aportaron al proyecto sus ideas y su tiempo. Son unos colegas maravillosos; de lo contrario, desde mucho tiempo atrás habrían comenzado a cerrar las puertas de sus oficinas al escuchar que me acercaba por el corredor.

Otro agradecimiento muy especial va dirigido a los estudiantes que recibieron cursos basados en las primeras versiones del manuscrito y me proporcionaron sus comentarios. Muchos de estos estudiantes mostraron una gran capacidad (y placer) al señalar mis errores, y su entusiasmo ha mejorado este libro.

Por último, un reconocimiento a la paciencia de mi esposa Earlene y mi hija Cheryl, a quienes dedico este libro.

ÍNDICE GENERAL

Capítulo 0 Preliminares **1**
0.1 Repaso de la teoría de conjuntos 2
0.2 Base gramatical de la traducción de lenguajes 12
0.3 Antecedentes históricos 14
0.4 Esbozo del resto del texto 16
Problemas de repaso del capítulo 17

Capítulo 1 Autómatas finitos y lenguajes regulares **21**
1.1 Análisis léxico 22
1.2 Autómatas finitos deterministas 28
1.3 Límites de los autómatas finitos deterministas 36
1.4 Autómatas finitos no deterministas 42
1.5 Gramáticas regulares 50
1.6 Expresiones regulares 57
1.7 Comentarios finales 68
Problemas de repaso del capítulo 69

Capítulo 2 Autómatas de pila y lenguajes independientes del contexto **75**
2.1 Autómatas de pila 76
2.2 Gramáticas independientes del contexto 82
2.3 Límites de los autómatas de pila 102
2.4 Analizadores sintácticos *LL*(*k*) 114
2.5 Analizadores sintácticos *LR*(*k*) 121
2.6 Comentarios finales 134
Problemas de repaso del capítulo 135

Capítulo 3 Máquinas de Turing y lenguajes estructurados por frases **141**
3.1 Máquinas de Turing 142

3.2 Construcción modular de máquinas de Turing 148
3.3 Máquinas de Turing como aceptadores de lenguajes 158
3.4 Lenguajes aceptados por máquinas de Turing 171
3.5 Más allá de los lenguajes estructurados por frases 181
3.6 Comentarios finales 195
Problemas de repaso del capítulo 196

Capítulo 4 Computabilidad 203
4.1 Fundamentos de la teoría de funciones recursivas 204
4.2 Alcance de las funciones recursivas primitivas 211
4.3 Funciones recursivas parciales 219
4.4 Poder de los lenguajes de programación 233
4.5 Comentarios finales 240
Problemas de repaso del capítulo 241

Capítulo 5 Complejidad 247
5.1 Complejidad de los cálculos 248
5.2 Complejidad de los algoritmos 251
5.3 Complejidad de los problemas 258
5.4 Complejidad temporal de los problemas de reconocimiento de lenguajes 267
5.5 Complejidad temporal de máquinas no deterministas 278
5.6 Comentarios finales 291
Problemas de repaso del capítulo 292

Apéndices 297
A. Más acerca de la construcción de tablas de análisis sintáctico *LR*(1) 297
B. Más acerca de la función de Ackerman 305
C. Algunos problemas importantes sin solución 313
D. Acerca de la complejidad del problema de comparación de cadenas 319
E. Muestras de problemas *NP* 323

Lecturas adicionales 327
Índice de materias 329
Vocabulario técnico bilingüe 333

CAPÍTULO 0

Preliminares

0.1 Repaso de la teoría de conjuntos
Definiciones básicas
Demostración por inducción
Conjuntos contables e incontables
¿Y qué?
0.2 Base gramatical de la traducción de lenguajes
0.3 Antecedentes históricos
0.4 Esbozo del resto del texto

El concepto de proceso computacional, también llamado proceso algorítmico o algoritmo, es fundamental para la ciencia de la computación. Al ejecutar estos procesos, los computadores son capaces de resolver problemas; por el contrario, un computador no puede resolver un problema que no tenga una solución algorítmica. Así, cualquier limitación de las capacidades de los procesos computacionales es también una limitación de las capacidades de una máquina computadora.

Este texto es una introducción al estudio de los procesos computacionales. Nuestro objetivo es explorar el alcance de estos procesos utilizando métodos matemáticos rigurosos, y al mismo tiempo relacionar los resultados teóricos con aspectos de interés práctico. Comenzaremos con un estudio teórico de las técnicas de reconocimiento de patrones que asociamos en lo general con los aspectos prácticos del procesamiento de lenguajes y en lo particular con la construcción de compiladores. Esto nos llevará a ciertos límites aparentes con respecto al poder de los procesos computacionales y luego a un estudio teórico de la computabilidad. Una vez más, no sólo veremos los aspectos teóricos; encontraremos que existen muchos problemas que teóricamente tienen una solución algorítmica, aun cuando no cuenten con soluciones algorítmicas prácticas. Para ser más precisos, muchos de los problemas que en teoría pueden resolverse con algoritmos requieren tal cantidad de recursos (en lo que se refiere a tiempo o espacio de almacenamiento) que, desde el punto de vista práctico, se debe considerar que aún no tienen solución. Por lo tanto, nuestro

estudio concluirá con una exploración de los trabajos de investigación actuales encaminados a identificar aquellos problemas que tienen soluciones prácticas.

Para motivar aún más nuestro estudio de la computabilidad, comparemos a un físico que resuelve problemas utilizando las leyes físicas descubiertas por Isaac Newton con un programador que resuelve problemas expresando sus soluciones en un lenguaje de programación. Existen varios problemas que el físico puede resolver dentro del marco newtoniano; sin embargo, también existen problemas para los cuales las leyes físicas de Newton son demasiado restrictivas y, para darles solución, el físico debe emplear las leyes más generales de la teoría de la relatividad de Einstein. La ciencia de la física incorpora la búsqueda de leyes más poderosas con las cuales explicar fenómenos físicos cada vez más complejos. No obstante, sin importar los principios fundamentales sobre los cuales se apoya su estudio, existirá un límite para las preguntas que pueden responder los físicos. Por ejemplo, si suponemos que la teoría de la gran explosión es verdadera, es poco probable que los físicos puedan explorar alguna vez la porción del universo que yace más allá de la distancia que ha viajado la luz desde la explosión.

Como punto de comparación, existen varios problemas que el programador de computadores puede resolver utilizando un lenguaje de programación determinado, pero este lenguaje quizá imponga limitaciones a los problemas que el programador puede resolver. ¿Es posible superar estas limitaciones desarrollando lenguajes de programación más poderosos? Además, como sucede con la teoría de la gran explosión, ¿existe algún punto en el cual la adición de características a un lenguaje de programación, o el cambio a otro lenguaje, no incremente el poder de solución de problemas del programador? Estas preguntas son parte medular de nuestro estudio.

0.1 REPASO DE LA TEORÍA DE CONJUNTOS

Gran parte de nuestro estudio comprenderá los conceptos básicos de la teoría de conjuntos, por lo que repasaremos estas ideas antes de proseguir.

Definiciones básicas

Intuitivamente, un **conjunto** es una colección de objetos que por lo general se denominan **elementos** (o **miembros**) del conjunto. Con frecuencia se emplea una notación con corchetes para describir un conjunto; por ejemplo, podemos escribir $A = \{a, b, c\}$ para indicar que A es el conjunto cuyos elementos son a, b, y c. Otro ejemplo es $W = \{x: x \text{ es un entero positivo}\}$, que se lee "$W$ equivale al conjunto de todo x tal que x es un entero positivo", lo cual indica que W es el conjunto de todos los enteros positivos. Un caso algo especial es el **conjunto vacío**, representado por $\{\ \}$ y también por $\varnothing$, que no contiene elementos.

A un conjunto también se le puede llamar colección o familia, y podemos referirnos a un conjunto de números, a una colección de números o a una familia de números. Otro término que se emplea con frecuencia es **clase**, aunque técnicamente se trata de un término más general que un conjunto. De hecho, el concepto de clase se desarrolló para evitar paradojas como "el conjunto de todos los conjuntos que no pertenecen a sí mismos" (¿se contendría a sí mismo un conjunto de este tipo?). Sin embargo, no tenemos que preocuparnos por estos detalles; sencillamente, usaremos el término "clase" en aquellos casos donde se acostumbra hacerlo.

Un conjunto que aparece con frecuencia en los capítulos siguientes es el conjunto de los **números naturales**, también conocido como conjunto de los enteros no negativos $\{0, 1, 2, 3, \cdots\}$, al cual representamos con el símbolo $\mathbb{N}$. Por desgracia, la definición de los números naturales varía un poco. Algunas personas no consideran que el cero se encuentre en el conjunto de números naturales, y definen a estos números como el conjunto de los enteros positivos $\{1, 2, 3, \cdots\}$. Este conjunto se representa con $\mathbb{N}^+$. Tal distinción no constituye más que un pequeño inconveniente. La tendencia a incluir el cero como número natural es un reflejo de que resulta más conveniente para los científicos de la computación comenzar a contar en cero que en uno.

El símbolo $\in$ se lee como "es un elemento de". Así, $x \in X$ significa que x es un elemento de X. Por otra parte, $\notin$ se lee como "no es un elemento de".

La **unión** de dos conjuntos A y B, representada por $A \cup B$, es la colección de todos los elementos que se encuentran en A o en B. Así, $A \cup B = \{x: x \in A \text{ o } x \in B\}$. La **intersección** de dos conjuntos A y B, representada por $A \cap B$, es la colección de objetos que son elementos tanto de A como de B; por consiguiente, $A \cap B = \{x: x \in A \text{ y } x \in B\}$. Por ejemplo, si $A = \{a, b, c\}$ y $B = \{b, c, d\}$, entonces $A \cup B$ sería $\{a, b, c, d\}$ y $A \cap B$ sería $\{b, c\}$.

Utilizamos el signo menos para representar la resta de conjuntos. Es decir, $\{a, b, c\} - \{a, c\} = \{b\}$, y $\{a, b\} - \{d, e\} = \{a, b\}$. El conjunto $A - B$ se llama **complemento** de B con respecto a A. Algunas veces se da por sentado que los elementos de todos los conjuntos considerados pertenecen a un conjunto universal de gran tamaño. En estos casos, nos referimos al complemento de un conjunto X con relación a este conjunto universal sencillamente como el complemento de X. De esta manera, al hablar de números naturales, el complemento del conjunto $\{1, 5\}$ se entendería como el complemento de $\{1, 5\}$ con relación a $\mathbb{N}$; si habláramos de símbolos alfabéticos, el complemento de $\{a, b, c\}$ sería el conjunto de símbolos distintos de a, b y c.

Decimos que A es un **subconjunto** de B, representado por $A \subseteq B$, si todos los elementos de A son también elementos de B (así, el conjunto vacío es un subconjunto de cualquier conjunto). El conjunto A es un **subconjunto propio** de B si $A \subseteq B$ y $B - A \neq \varnothing$. Los conjuntos A y B son iguales si se cumple que $A \subseteq B$ y $B \subseteq A$.

El **producto cartesiano** de dos conjuntos A y B, representado por $A \times B$, es el conjunto de todos los pares ordenados de la forma (a, b), donde $a \in A$ y $b \in B$ (observe que, por lo general, $A \times B \neq B \times A$ ya que la disposición de los

componentes en un par ordenado es significativa). Es posible generalizar el concepto del producto de conjuntos para obtener el producto de más de dos conjuntos. Por ejemplo, $A \times B \times C = \{(a, b, c): a \in A, b \in B \text{ y } c \in C\}$. Es común utilizar la notación abreviada A^2 para representar $A \times A$, A^3 para representar $A \times A \times A$ y, de manera general, A^n para representar la colección de todas las n-tuplas de elementos de A.

Dados dos conjuntos A y B, una **función** de A a B es un subconjunto de $A \times B$ tal que cada elemento de A aparece como el primer elemento de uno y sólo uno de los pares ordenados de la función. Decimos que una función establece una **correspondencia** entre A y B. Si f es una función de A a B, entonces A es el **dominio** de f y se dice que f es una función de A. El conjunto de elementos que aparece como el segundo componente de los pares ordenados de una función se denomina **contradominio** de la función. Si cada elemento de B se halla en el contradominio de f, decimos que f es una función de A **sobre** B (o que f establece una correspondencia de A sobre B). Si cada miembro del contradominio de f está asociado a sólo un elemento del domi-nio, decimos que f es una función **uno a uno**, o sencillamente que f es uno a uno.

Por lo general, los elementos del dominio de una función se consideran como entradas para la función y los elementos del contradominio como los valores de salida. Este concepto de entrada y salida origina la notación $f: A \to B$, lo cual representa una función f de A a B. Dada una función $f: A \to B$, representamos como $f(x)$ al valor de salida asociado a la entrada x.

Por ejemplo, suponga que $f: \mathbb{N} \to \mathbb{N}$ es la función que consiste en los pares ordenados de la forma (n, n^2), es decir, $f(n) = n^2$ para cada $n \in \mathbb{N}$. Esta función es uno a uno, pero no se encuentra sobre $\mathbb{N}$ ya que el contradominio de f sólo contiene cuadrados perfectos.

Demostración por inducción

Algunas veces queremos demostrar que una proposición es verdadera para una serie de enteros. Por ejemplo, si $\mathcal{P}(X)$ representa el **conjunto potencia** de X (es decir, la colección de todos los subconjuntos de X) y $|X|$ representa el número de elementos del conjunto X, quizá querríamos demostrar que

$$|\mathcal{P}(X)| = 2^{|X|}$$

para todos los conjuntos finitos X con tamaño mayor o igual que uno.

En la figura 0.1 se confirma esta fórmula para todos los conjuntos de tamaño uno, dos y tres, presentando exhaustivamente todos los subconjuntos de cada caso (observe que los nombres asignados a los elementos de los conjuntos no tienen importancia; aquí empleamos la notación $\{1, 2, \cdots, n\}$ para representar un conjunto genérico de tamaño n). No obstante, esta estrategia explícita de caso por caso nunca podrá demostrar que la fórmula es verdadera para todos los casos posibles, ya que existe un número

El conjunto X	Todos los subconjuntos de X	
$\{1\}$	$\{\ \}$	$\{1\}$
$\{1, 2\}$	$\{\ \}$	$\{2\}$
	$\{1\}$	$\{1, 2\}$
$\{1, 2, 3\}$	$\{\ \}$	$\{3\}$
	$\{1\}$	$\{1, 3\}$
	$\{2\}$	$\{2, 3\}$
	$\{1, 2\}$	$\{1, 2, 3\}$

Figura 0.1 Conjuntos de cardinalidad 1, 2 y 3 con todos sus subconjuntos

infinito de casos que habría que evaluar. Lo que necesitamos es un argumento general que cubra al mismo tiempo todos los casos.

En nuestro ejemplo, podemos obtener este argumento identificando un método genérico para construir la colección de subconjuntos del conjunto $\{1, 2, \cdots, n+1\}$ a partir de la colección de subconjuntos del conjunto $\{1, 2, \cdots, n\}$. De manera específica, una vez generados todos los subconjuntos del conjunto $\{1, 2, \cdots, n\}$, podemos generar todos los subconjuntos de $\{1, 2, \cdots, n, n+1\}$ presentando una lista de todos los subconjuntos que se encuentran en el caso anterior (en una columna) y luego repitiendo esta lista (en una segunda columna) e insertando el elemento adicional, $n+1$, en cada subconjunto (ésta fue la técnica empleada para construir la figura 0.1). A partir de esta observación general, llegamos a la conclusión de que existen dos veces más subconjuntos para un conjunto de tamaño $n + 1$ que para un conjunto de tamaño n. Es decir, para cualquier $n \in \mathbb{N}^+$,

$$|\mathcal{P}(\{1, 2, \cdots, n+1\})| = 2\,|\mathcal{P}(\{1, 2, \cdots, n\})|$$

Aquí lo importante es que se trata de una conclusión genérica, ya que no se basa en un valor específico de n: tiene la misma validez para el valor $n = 3$ que para $n = 153$ o para cualquier otro $n \in \mathbb{N}^+$. Por consiguiente, si podemos demostrar con un argumento específico que $|\mathcal{P}(X)| = 2^{|X|}$ para un conjunto X, esta proposición genérica indicaría que la ecuación puede ampliarse para cada conjunto finito mayor sin tener que validar cada caso con un argumento aparte. De hecho, si siempre existen dos veces más subconjuntos para un conjunto de tamaño $n + 1$ que para un conjunto de tamaño n y si $|\mathcal{P}(X)| = 2^{|X|}$ para un conjunto finito X, entonces, para cualquier conjunto Y donde $|Y| = |X| + 1$, tenemos

$$|\mathcal{P}(Y)| = 2\,|\mathcal{P}(X)| = 2\,(2^{|X|}) = 2^{|Y|}$$

Entonces, si confirmamos la ecuación $|\mathcal{P}(X)| = 2^{|X|}$ para el caso de un conjunto de tamaño uno (como se hizo en el primer renglón de la figura 0.1), la ecuación debe ser válida para cualquier conjunto de tamaño $n \in \mathbb{N}^+$.

Esta técnica se conoce como demostración por **inducción**; resulta útil cuando se desea demostrar un enunciado que tiene que ver con una serie de enteros. En vez de tratar de usar un argumento caso por caso, lo que nunca sería suficiente si existiera un número infinito de casos, se proporciona un argumento específico que contemple, el menor valor considerado (**caso básico**) y luego se proporciona un argumento genérico para demostrar que, si el enunciado es verdadero en un caso, también deberá ser verdadero para el siguiente caso mayor.

Hemos empleado el ejemplo del conjunto potencia ya que la fórmula obtenida será de utilidad en análisis posteriores. Como un ejemplo más vinculado al ambiente de programación, se aplicará la inducción para demostrar que el procedimiento recursivo clasificación, que se esboza en la figura 0.2, obtiene la versión clasificada de su lista de entrada. Utilizaremos la inducción en el número de entradas de la lista.

Primero se confirma que clasificación opera correctamente para el caso básico, donde la lista de entrada no contiene dato alguno (lista vacía), rastreando explícitamente la ejecución del procedimiento para ese caso. Luego se supone que el procedimiento da como resultado la versión clasificada de listas con n elementos, donde n es un entero positivo genérico, y se demuestra que, entonces, debe producir la versión clasificada de cualquier lista que contenga $n + 1$ elementos.

Para este fin observamos que una llamada de la forma clasificación (ListaEntrada), donde ListaEntrada es una lista que contiene $n + 1$ elementos, ocasionaría la ejecución de la porción then del procedimiento. Allí, otra activación de clasificación produce la versión clasificada de la sublista que consiste en los elementos posteriores al primero de la lista original ListaEntrada. Puesto que esta sublista sólo contiene n elementos, la activación de clasificación se ejecutará correctamente (por nuestra suposición respecto a la lista con sólo n elementos) y, por lo tanto, la lista de entrada original aparecerá como el primer elemento original seguido por la versión clasificada del resto de la lista. Por ende, la siguiente instrucción del procedimiento, que mueve el primer elemento de la lista a su lugar correcto entre los demás, dará fin a la tarea de clasificar toda la lista. Concluimos que si el procedimiento clasificación clasifica correctamente listas de n elementos, entonces también debe clasificar de manera correcta listas con $n + 1$ elementos. Al combinar esto con la primera conclusión anterior (que clasificación opera en forma correcta si se le proporciona la lista vacía), demostramos que el procedimiento clasificará listas finitas de cualquier tamaño.

Conjuntos contables e incontables

Consideremos ahora el concepto intuitivo del número de elementos en un conjunto. A primera vista, no parece haber nada complicado con respecto a


```
procedure clasificación (lista);
begin
    if (lista no está vacía) then
        begin
            clasificación (porción de la lista que sigue al primer elemento);
            mover el primer elemento a su lugar correcto
                dentro de la parte restante de la lista
        end
end;
```

Figura 0.2 Esbozo del procedimiento ***clasificación***

este concepto. Es obvio que en el conjunto $\{a, b, c\}$ el número de elementos es tres, y que en $\{x, y\}$ es dos. Sin embargo, ¿cuántos elementos hay en el conjunto $\mathbb{N}^+$? Una persona común diría que una "infinidad", pero esto, para nuestros propósitos, es demasiado simplista. El infinito no es un número de nuestro sistema contable normal, sino que el término refleja el concepto de un conjunto con más elementos que los que pueden representarse con un número tradicional. Además, en seguida veremos que los conjuntos con un número infinito de elementos, llamados **conjuntos infinitos**, no contienen la misma cantidad de elementos. El tamaño de dos conjuntos infinitos puede variar, al igual que el tamaño de dos conjuntos finitos. Por esto, existen varios niveles de infinitud, donde algunos conjuntos infinitos son mayores que otros.

Ya que en nuestro sistema contable tradicional no existen números que representen el tamaño de un conjunto infinito, es necesario ampliar nuestro sistema para tener en cuenta estos conjuntos. En este sistema ampliado nos referimos al tamaño de un conjunto como su **cardinalidad**. La cardinalidad de un conjunto finito corresponde al concepto tradicional del número de elementos en un conjunto, pero la cardinalidad de un conjunto infinito no es un número en el sentido habitual. Más bien, la cardinalidad de un conjunto infinito es un número, llamado **número cardinal**, que existe únicamente en nuestro sistema de numeración ampliado. La cardinalidad de un conjunto A se representa con $|A|$.

Para completar nuestro sistema de cardinalidad, es necesario establecer una forma de comparar las cardinalidades de dos conjuntos diferentes, es decir, se requiere definir qué significa el hecho de que una cardinalidad sea menor que otra. Para esto, se establece que la cardinalidad de un conjunto X es menor o igual que la cardinalidad del conjunto Y, lo cual se representa con $|X| \leq |Y|$, si y sólo si puede establecerse una correspondencia de cada elemento de X con un elemento único de Y. En otras palabras, $|X| \leq |Y|$ si y sólo si hay una función uno a uno de X a Y. Observe que si X y Y son conjuntos finitos, esta definición de menor o igual equivale a la definición tradicional: $|\{x, y\}| \leq |\{r, s, t\}|$ ya que puede establecerse una correspondencia entre x y r y entre y y s.

A continuación, se dice que la cardinalidad del conjunto X es igual a la cardinalidad del conjunto Y, representado por $|X| = |Y|$, si y sólo si existe una función uno a uno que establezca la correspondencia de X *sobre* Y (el teorema de Schröder-Bernstein, cuyos detalles no son importantes ahora, confirma que estas definiciones están de acuerdo con nuestra intuición; establece que si $|X| \leq |Y|$ y $|Y| \leq |X|$, entonces $|X| = |Y|$). Observe que $|\mathbb{N}| = |\mathbb{N}^+|$, donde una función uno a uno adecuada es aquella que asocia cada x de $\mathbb{N}$ con el número $x + 1$ de $\mathbb{N}^+$. Por último, definimos $|X| < |Y|$ para representar $|X| \leq |Y|$ y $|X| \neq |Y|$.

En nuestro análisis, es muy importante la relación entre un conjunto y su conjunto potencia. En el caso de un conjunto finito X, ya se ha mostrado que hay más elementos en $\mathcal{P}(X)$ que en X (de hecho, si X contiene n elementos, entonces $\mathcal{P}(X)$ contendrá 2^n elementos). El teorema siguiente extiende a los conjuntos infinitos esta relación entre un conjunto y su conjunto potencia.

RELEER.

TEOREMA 0.1
Si X es un conjunto cualquiera, entonces $|X| < |\mathcal{P}(X)|$.

DEMOSTRACIÓN
Obviamente, $|X| \leq |\mathcal{P}(X)|$ pues la función que establece la correspondencia entre cada elemento x de X y el conjunto $\{x\}$ proporciona la función uno a uno que se requiere (si X está vacío, entonces cada elemento de X puede corresponderse con un elemento de $\mathcal{P}(X)$ ya que no hay elementos en X).

Nuestra tarea, entonces, es mostrar que $|X| \neq |\mathcal{P}(X)|$. Esto se puede lograr demostrando que no existe función alguna de X *sobre* $\mathcal{P}(X)$. Suponga que f es cualquier función de X a $\mathcal{P}(X)$ y considere el conjunto $Y = \{x : x \in X \text{ y } x \notin f(x)\}$. Observe que Y es un subconjunto de X y, por lo tanto, un elemento de $\mathcal{P}(X)$. Luego, si f es sobre $\mathcal{P}(X)$, entonces en X debe existir una y tal que $f(y) = Y$.

Sin embargo, si existe tal y, entonces debe cumplirse $y \in f(y)$ o $y \notin f(y)$, las cuales conducen a contradicciones. Si $y \in f(y)$, entonces la definición de Y significaría que $y \notin Y$, lo que contradice la afirmación de que $f(y) = Y$. De manera parecida, si $y \notin f(y)$, entonces la definición de Y implicaría que $y \in Y$, lo cual también contradice la suposición de que $f(y) = Y$.

Concluimos que no existe un $y \in X$ tal que $f(y) = Y$. Por consiguiente f no puede ser *sobre* $\mathcal{P}(X)$ y, por lo tanto, $|X| < |\mathcal{P}(X)|$.

∎

Ahora nos encontramos en un punto donde podemos apoyar nuestra afirmación de que los conjuntos infinitos pueden tener cardinalidades distintas. Basta con seleccionar un conjunto infinito X y comparar su cardinalidad con la de su conjunto potencia $\mathcal{P}(X)$. Por el teorema 0.1, la cardinalidad del segundo conjunto debe ser mayor que la del primero. De hecho, el proceso de formación de conjuntos potencia nos lleva a una jerarquía de conjuntos infinitos ya que la cardinalidad de $\mathcal{P}(\mathcal{P}(X))$ debe ser mayor que la cardinalidad de $\mathcal{P}(X)$, etcétera. Es decir,

$$|X| < |\mathcal{P}(X)| < |\mathcal{P}(\mathcal{P}(X))| < \cdots$$

Una consecuencia de esta jerarquía es que no existe ningún cardinal infinito mayor. Dado un conjunto infinito, su potencia será siempre mayor.

Existe, no obstante, un cardinal infinito menor: la cardinalidad de $\mathbb{N}$. Esta afirmación va de acuerdo con nuestra percepción intuitiva de que si X es un conjunto infinito, entonces es posible extraer elemento por elemento sin agotar el conjunto; es decir, se puede extraer un elemento "cero", luego un "primer" elemento seguido por un "segundo", etc.; por lo tanto, X debe contener por lo menos $|\mathbb{N}|$ elementos. Sin embargo, este argumento intuitivo se basa en un enfoque de caso por caso que, como ya establecimos en nuestro análisis de la inducción, no proporciona una demostración válida. Por lo tanto, presentaremos el siguiente teorema formal y su demostración; se basa en el axioma de elección, el cual establece que, dada una colección cualquiera de conjuntos no vacíos, existe una función g cuyo dominio es la colección de conjuntos, y que para cada conjunto X de la colección, $g(X) \in X$. Es decir, si se toma como entrada un conjunto X de la colección, la función dará como resultado un elemento de X.

TEOREMA 0.2
La cardinalidad del conjunto de los números naturales es menor o igual que la cardinalidad de cualquier conjunto infinito.

DEMOSTRACIÓN
Debemos mostrar que, dado un conjunto infinito X, existe una función f uno a uno de $\mathbb{N}$ a X. Para esto, hagamos que g represente una función cuyo dominio es la colección de los subconjuntos no vacíos de X, y que para cada subconjunto no vacío Y de X, $g(Y)$ es un elemento de Y (la existencia de esta función se basa en el axioma de elección). Mediante esta asociación entre subconjuntos no vacíos de X y elementos, definimos ahora la función f por inducción. Definimos que $f(0)$ es $g(X)$. Entonces, si n es un número natural para el cual se han definido $f(n), f(n-1), \cdots, f(0)$, definimos $f(n+1)$ como el elemento $g(X - \{f(0), f(1), \cdots, f(n)\})$. Observe que, de acuerdo con nuestra definición de g, $f(n+1)$ se toma del conjunto $X - \{f(0), f(1), \cdots, f(n)\}$, el cual es no vacío ya que X es infinito y $\{f(0), f(1), \cdots, f(n)\}$ es finito.

Por lo tanto, $f(n+1)$ debe ser distinto de $f(0), f(1), \cdots, f(n)$. Entonces, f es una función uno a uno de $\mathbb{N}$ en X. Concluimos que la cardinalidad de los números naturales no es mayor que la de X.

∎

El teorema 0.2 muestra que el conjunto $\mathbb{N}$ es uno de los conjuntos infinitos más pequeños. Decimos que es *uno* de los más pequeños ya que existen varios conjuntos del mismo tamaño (de la misma manera que existen varios conjuntos de tamaño tres). Por ejemplo, ya se vio que $\mathbb{N}^+$ es del mismo tamaño que $\mathbb{N}$.

Finalmente, presentaremos los conceptos de conjuntos contables e incontables, como antes prometimos. Aquí resulta útil recordar el proceso al que normalmente nos referimos como contar. Contar los elementos de un conjunto significa asignar los valores $1, 2, 3, \cdots$, a los elementos del conjunto. Incluso podemos pronunciar las palabras "uno, dos, tres,⋯" conforme señalamos los objetos durante el recuento. En resumen, el proceso de recuento asigna enteros positivos a los elementos de un conjunto. Sin embargo, hemos visto que existen conjuntos infinitos con más elementos que la cantidad de enteros positivos. Por consiguiente, no es posible contar estos conjuntos ya que no hay enteros positivos suficientes. Se dice que estos conjuntos son **incontables**. Por otra parte, se dice que un conjunto es **contable** (o **enumerable**) si su cardinalidad es menor o igual que la de los enteros positivos. El conjunto de los números naturales es contable; no así su conjunto potencia. De hecho, la combinación de los teoremas 0.1 y 0.2 indica que *el conjunto potencia de cualquier conjunto infinito es incontable.*

Una forma de mostrar que un conjunto infinito es contable consiste en demostrar que existe un proceso para presentar una lista en secuencia de sus elementos de manera que cada elemento del conjunto aparezca en la lista. La elaboración de esta lista es una forma de contar los elementos: a la primera entrada de la lista se le asigna uno; a la segunda se le asigna dos; a la tercera, tres, etcétera.

Como ejemplo, se mostrará que el conjunto $\mathbb{N}^+ \times \mathbb{N}^+$ es contable. Esto se logra describiendo la elaboración de una lista de todos los elementos de $\mathbb{N}^+ \times \mathbb{N}^+$. Esta lista consiste en todos los pares cuyos componentes suman dos, a continuación todos los pares cuyos componentes suman tres, en seguida todos los pares cuyos componentes suman cuatro, etcétera. Los pares de cada una de estas agrupaciones se colocan en orden, de acuerdo con sus primeros componentes. Así, la lista tiene la forma (1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1), ⋯ (véase Fig. 0.3). Observe que, a final de cuentas, esta lista contendrá cada uno de los elementos de $\mathbb{N}^+ \times \mathbb{N}^+$ (el par (m, n) aparecerá como el elemento n la porción de la lista que contiene los pares cuyos componentes suman $m + n$). Por último, observamos que la elaboración de esta lista es, en realidad, un recuento de los elementos de $\mathbb{N}^+ \times \mathbb{N}^+$: hemos asignado uno a (1, 1), dos a (1, 2), tres a (2, 1), etcétera. Por consiguiente, podemos concluir que $\mathbb{N}^+ \times \mathbb{N}^+$ es contable.

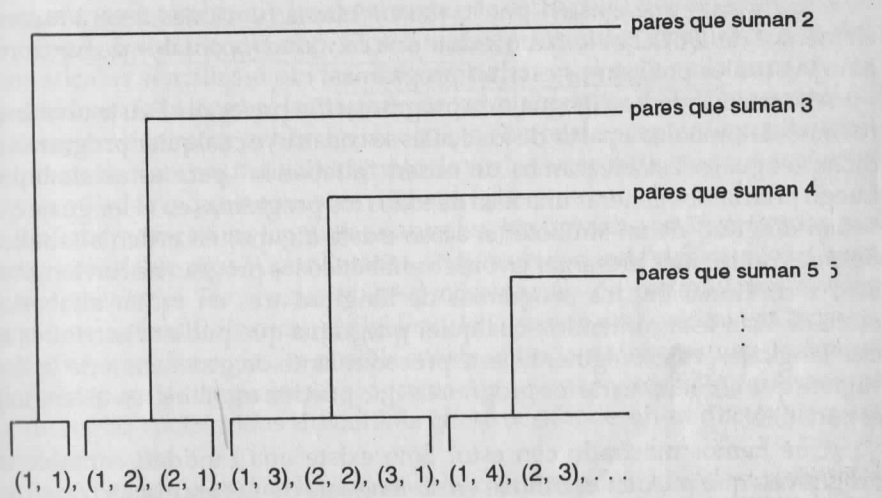


Figura 0.3 Elaboración de la lista de elementos de $\mathbb{N}^+ \times \mathbb{N}^+$ en secuencia

¿Y qué?

Concluyamos esta sección relacionando nuestro análisis con un ambiente de computación típico. Suponga que tenemos un programa listo para ejecutarse en un computador. El programa está diseñado para aceptar cierta entrada (quizás una lista de nombres que debe clasificarse) y producir determinada salida (como sería la lista ordenada). Sin importar cuáles sean la entrada y la salida, desde el punto de vista de la máquina no son más que una cadena de bits. Por lo tanto, podemos considerar a la entrada y a la salida como números naturales representados en notación binaria (podemos imaginar un 1 adicional colocado en el extremo izquierdo de estas cadenas de bits, de modo que los 0 iniciales de las cadenas originales sean significativos). Desde esta perspectiva, el programa acepta un número de entrada y, con base en el valor de ese número, produce un número de salida. Es decir, la acción del programa es calcular una función de $\mathbb{N}$ a $\mathbb{N}$. A la vez, cada función de $\mathbb{N}$ a $\mathbb{N}$ sugiere un programa que en alguna ocasión podríamos escribir.

¿Cuántas funciones de éstas existen? Obviamente, no hay más funciones de $\mathbb{N}$ a $\{0, 1\}$ que de $\mathbb{N}$ a $\mathbb{N}$, ya que éstas proporcionan más valores de salida potenciales. Sin embargo, hay tantas funciones de $\mathbb{N}$ a $\{0, 1\}$ como elementos en el conjunto potencia de $\mathbb{N}$. Incluso, para cada subconjunto X de $\mathbb{N}$ hay una función f, llamada **función característica** de X, de $\mathbb{N}$ a $\{0, 1\}$, definida por

$$f(n) = \begin{cases} 1 \text{ si } n \in X \\ 0 \text{ si } n \notin X \end{cases}$$

Concluimos que existen por lo menos tantas funciones de $\mathbb{N}$ a $\mathbb{N}$ como elementos de $\mathcal{P}(\mathbb{N})$. Por lo tanto, existe una cantidad incontable de funciones para las cuales podríamos escribir programas.

Ahora, considere su lenguaje de programación preferido. Existe un número finito de símbolos a partir de los cuales se construye cualquier programa en dicho lenguaje. Establezcamos un orden "alfabético" para estos símbolos. Luego podríamos generar una lista de todos los programas en el lenguaje que tienen longitud de un símbolo (si acaso existe alguno) en orden alfabético; después podríamos presentar en orden alfabético los programas con longitud dos; a continuación, los programas de longitud tres en orden alfabético, etcétera. Esta lista contendría cualquier programa que pudiera escribirse en este lenguaje. Por consiguiente, este procedimiento de generación de la lista sugiere que el conjunto de los programas que pueden escribirse en el lenguaje es contable.

¿Qué hemos mostrado con esto? Sólo existe una cantidad contable de programas que pueden escribirse en su lenguaje de programación favorito, pero hay una cantidad incontable de funciones que podrían calcularse con un programa. Puesto que cada programa calcula sólo una función, se quedará corto. Por ende, existen problemas que se quisieran resolver utilizando un computador, ¡pero para los cuales no puede escribirse un programa en su lenguaje de programación preferido!

¿Cuáles son algunos de estos problemas? ¿Es posible resolver alguno de ellos cambiando a otro lenguaje de programación? ¿Podrían resolverse estos problemas si construimos un computador más grande? Incluso si usted puede producir un programa para resolver un problema, ¿sería capaz una máquina actual de ejecutarlo con la rapidez suficiente para producir una respuesta mientras usted esté vivo? ¿Será capaz el computador del mañana de producir una respuesta durante la vida de sus hijos? Estas son algunas de las preguntas que trataremos directa o indirectamente en los capítulos siguientes.

0.2 BASE GRAMATICAL DE LA TRADUCCIÓN DE LENGUAJES

Gran parte de las primeras etapas de nuestro estudio se basará en problemas que tienen que ver con el procesamiento del lenguaje. Por lo anterior, tomemos un momento para repasar la terminología y algunos de los conceptos asociados al proceso de traducción de lenguajes.

En primer lugar, debemos esclarecer los términos **lenguaje formal** y **lenguaje natural**. Es posible distinguir a ambos por medio de la pregunta "¿qué surgió primero, el lenguaje o sus reglas gramaticales?" En general, un lenguaje natural es aquel que ha evolucionado con el paso del tiempo para fines de la comunicación humana, por ejemplo el español, el inglés o el alemán. Estos lenguajes continúan su evolución sin tomar en cuenta reglas gramaticales formales; cualquier regla se desarrolla después de que sucede el hecho, en

un intento por explicar, y no determinar, la estructura del lenguaje. Como resultado de esto, pocas veces los lenguajes naturales se ajustan a reglas gramaticales sencillas u obvias. Esta situación es el pelo en la sopa de los académicos de la lengua, la mayoría de los cuales trata de convencernos de que nuestra forma de hablar es incorrecta en vez de aceptar la posibilidad de que no han encontrado el conjunto de reglas adecuado, que sus reglas ya no son válidas o que no existen dichas reglas.

En contraste con los lenguajes naturales, los formales están definidos por reglas preestablecidas y, por lo tanto, se ajustan con todo rigor a ellas. Como ejemplo tenemos los lenguajes de programación de computadores y los lenguajes matemáticos como el álgebra y la lógica de proposiciones. Gracias a esta adhesión a las reglas, es posible construir traductores computadorizados eficientes para los lenguajes de programación, a la vez que la falta de observancia de reglas establecidas dificulta la construcción de un traductor para un lenguaje natural.

Para realzar el papel de la gramática en el proceso de traducción, revisemos los pasos que sigue un típico traductor de lenguajes, como un compilador para determinado lenguaje de programación. Una de las tareas fundamentales de este traductor es reconocer los bloques de construcción individuales del lenguaje que se traduce. En su nivel más bajo, esto implica reconocer que ciertas cadenas de símbolos del programa que se traduce (llamado **programa fuente**) deben considerarse como la representación de un solo objeto. Por ejemplo, hay que reconocer la variable LISTA como un identificador, en vez de cinco letras por separado; la cadena de dígitos 524 debe reconocerse como un solo número; la cadena: = (que se utiliza en varios lenguajes de programación) debe reconocerse como la operación de asignación.

El proceso de identificación de estas secuencias multisímbolos dentro del programa fuente es manejada por una porción del compilador llamada **analizador léxico**. En esencia, se trata de un submódulo que recibe el programa fuente como una cadena de símbolos y produce una cadena de **componentes léxicos** *(tokens)*, cada uno de los cuales representa un solo objeto que pudo haberse representado con más de un símbolo en la cadena de entrada (un componente léxico puede representarse como un valor entero que es posible almacenar en una sola celda de la memoria de la máquina).

Después de reducir a un componente léxico cada secuencia multisímbolos que representa a un objeto, la siguiente tarea del traductor es analizar el patrón de componentes léxicos. Al llegar a este punto, un compilador típico tendría que reconocer que los componentes léxicos de if, then y else representan conjuntamente una estructura if-then-else, o que la cadena que se encuentra entre el componente léxico repeat y el componente léxico until correspondiente representa el cuerpo de una estructura de ciclo. Este análisis se conoce como **análisis sintáctico** y es realizado por un módulo del compilador que se llama **analizador sintáctico**.

Conforme el analizador sintáctico reconoce las distintas estructuras del programa, solicita los servicios de otro módulo, llamado **generador de código**,

para producir la versión traducida de esa parte específica del programa. Al formar estas porciones traducidas del programa, empieza a constituirse la versión traducida de éste (llamada **programa objeto**).

En resumen, el proceso de traducción consta de tres componentes básicos: análisis léxico, análisis sintáctico y generación de código, como se muestra en la figura 0.4. Los primeros dos componentes dependen en gran medida de la capacidad para reconocer patrones. Si éstos se basan en reglas gramaticales, entonces el proceso de reconocimiento también puede apoyarse en estas reglas. Lo anterior sucede al procesar un lenguaje formal. Sin embargo, si el lenguaje que se traduce no se adhiere estrictamente a reglas gramaticales bien definidas, puede ser muy difícil construir un segmento de programa para el análisis sintáctico del lenguaje.

0.3 ANTECEDENTES HISTÓRICOS

La mayor parte del conocimiento científico es el resultado de muchos años de investigación, con frecuencia sobre temas aparentemente no relacionados. Esto sucede incluso en un campo tan joven como el de la ciencia de la computación. Una gran parte de los resultados que se presentan en los capítulos siguientes fue descubierta a principios del siglo XX por matemáticos que trabajaban en áreas como la lógica y los sistemas axiomáticos. Por lo tanto, es relevante para nuestro estudio hacer una breve introducción a este periodo matemático.

Al iniciar el siglo XX, las matemáticas estaban a punto de efectuar grandes descubrimientos. Los logros de los siguientes 40 años estaban destinados a sacudir las bases de las matemáticas y tuvieron consecuencias que se extendieron al campo de la ciencia de la computación, la cual se hallaba por nacer. Para poder comprender este periodo de transición, consideremos el enfoque axiomático para las matemáticas, un tema que recibió mucha atención al comenzar el siglo.

Poniéndolo de manera sencilla, el método axiomático requiere una colección de enunciados básicos, llamados **axiomas**, que describen las propiedades fundamentales del sistema que se estudia. Apartir de estos **axiomas**, se derivan

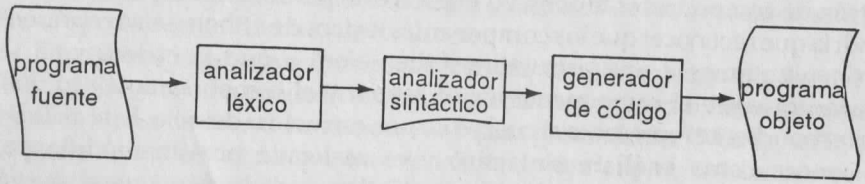


Figura 0.4 Proceso básico de traducción

enunciados adicionales, llamados **teoremas**, aplicando secuencias finitas de **reglas de inferencia**. Estas reglas están diseñadas para que cualquier teorema que se derive sea un enunciado que constituya una consecuencia lógica de los axiomas. Por ejemplo, una regla de inferencia bastante conocida es *modus ponens*, que establece que a partir de dos enunciados de la forma

$$\text{si } X \text{ entonces } Y$$

y

$$X$$

podemos derivar el enunciado Y. Por medio del *modus ponens*, podemos concluir "sopla el viento" a partir de los enunciados "los árboles se mueven" y "si los árboles se mueven, sopla el viento".

Una ventaja del método axiomático es que ofrece un modelo de razonamiento deductivo en el cual todas las suposiciones están aisladas en los axiomas iniciales y las reglas de inferencia. Se asegura que cualquier enunciado que se derive posteriormente sea una consecuencia de estas suposiciones, y sólo de estas suposiciones. Así, el método axiomático proporciona una importante herramienta que puede emplearse para pesquisas científicas.

A principios de este siglo, muchos matemáticos creían que todas las matemáticas podían basarse en un solo sistema axiomático, y su meta era encontrar el conjunto de axiomas y reglas de inferencias correctos. Así, cualquier teorema matemático podría derivarse de los axiomas con sólo aplicar una serie finita de reglas de inferencia. De esta manera, las matemáticas se verían reducidas a un sistema computacional en el cual podría determinarse algorítmicamente la veracidad o falsedad de cualquier enunciado matemático.

En 1931, surgió un descubrimiento sorprendente con la publicación del teorema de incompleción de Kurt Gödel. En esencia, este teorema establece que para cualquier sistema axiomático válido con la riqueza suficiente para la parte de las matemáticas que consiste en los números naturales y las operaciones de suma y multiplicación, deben existir enunciados matemáticos cuya veracidad o falsedad no pueda probarse a partir del sistema. Es decir, la axiomatización de las matemáticas estaba condenada al fracaso.

El teorema de Gödel generó un gran interés por el poder de los métodos axiomáticos y los procesos computacionales. Si no existe ningún algoritmo general para demostrar teoremas matemáticos, entonces, ¿qué puede lograrse con los medios axiomáticos? Para responder esta pregunta, los matemáticos comenzaron a desarrollar máquinas computacionales teóricas (esto sucedió antes de que la tecnología fuera capaz de producir máquinas reales) y a estudiar las capacidades y limitaciones de estas máquinas. Los resultados de esta labor forman el núcleo del tema conocido como teoría de la computación, que hoy en día abarca subtemas como la teoría de lenguajes formales, la teoría de la computabilidad y la teoría de la complejidad, objetos de nuestro estudio.

Por lo tanto, no es ninguna sorpresa que la ciencia computacional teórica tenga un fuerte sabor matemático. Las preguntas de computabilidad a las

que se enfrentan los científicos teóricos de la computación son las mismas preguntas de demostrabilidad a las que se enfrentan los matemáticos, y viceversa. Básicamente, la diferencia se debe a la perspectiva.

0.4 ESBOZO DEL RESTO DEL TEXTO

Damos fin a este capítulo preliminar con un breve resumen de los capítulos restantes. En los capítulos 1 a 3 estudiamos tres clases de máquinas teóricas: autómatas finitos, autómatas de pila y máquinas de Turing, cada una de las cuales es más poderosa que la anterior. Consideramos el poder de estas máquinas en el contexto de la resolución de problemas de reconocimiento de patrones. Es decir, nuestro interés se centrará en la identificación de los tipos de patrones de símbolos que pueden reconocer las distintas clases de máquinas.

Como ya hemos visto, los problemas de reconocimiento de patrones son parte medular de los sistemas de procesamiento de lenguajes, como los compiladores de los lenguajes de programación de alto nivel que existen en la actualidad. Por lo tanto, no nos sorprende que el material que se analiza en los tres capítulos siguientes proporcione la base teórica para la construcción de compiladores modernos, una relación entre la teoría y la práctica que examinaremos.

El material que se cubre en los capítulos 1 a 3 también se aplica en el campo del procesamiento de lenguajes naturales. Aprenderemos que la jerarquía de máquinas que se estudiará en los capítulos siguientes corresponde a la jerarquía de lenguajes descubierta en la década de los cincuenta por el lingüista N. Chomsky, quien se interesó por establecer una base formal para los estudios lingüísticos. Por esto, nuestro estudio de las máquinas de reconocimiento de patrones está estrechamente relacionado con la lingüística.

Los tres capítulos siguientes también nos llevarán a establecer un límite aparente para las capacidades computacionales de las máquinas. Nadie ha sido capaz de diseñar un tipo de máquinas (reales o teóricas) con mayor poder computacional que la clase de máquinas de Turing. Es más, una conjetura generalmente aceptada, conocida como tesis de Turing, dice que las máquinas de Turing poseen la capacidad suficiente para resolver cualquier problema para el cual existe una solución por medios computacionales. La tesis de Turing ofrece el contexto para formular nuestro estudio de la computabilidad.

En el capítulo 4 analizaremos evidencias que apoyan la tesis de Turing. Teniendo presente este objetivo, consideraremos otras áreas de investigación que han expuesto límites aparentes para los procesos computacionales y que muestran cómo estos límites son equivalentes a los de las máquinas de Turing. En las primeras secciones del capítulo 4 presentaremos el tema de la teoría de funciones recursivas, en el cual se identifica el límite aparente del poder computacional en una conjetura conocida como tesis de Church. A continuación mostraremos que la tesis de Church equivale a la tesis de Turing. Como

evidencia adicional, consideraremos los límites de los procesos computacionales en el contexto del diseño de lenguajes de programación, donde una vez más encontraremos un límite que está de acuerdo con la tesis de Turing.

Por último, en el capítulo 5 abordaremos la cuestión de la computabilidad desde un punto de vista práctico en vez de teórico. Aquí analizaremos algunos de los retos a los que se enfrentan los investigadores que intentan clasificar los problemas en función de los recursos requeridos para resolverlos por medios algorítmicos. Encontraremos que las soluciones algorítmicas de muchos problemas requieren tales cantidades de tiempo o espacio de almacenamiento que, desde un punto de vista práctico, los problemas permanecen sin solución, aunque existe una solución en las capacidades teóricas de los procesos computacionales. También veremos que los investigadores todavía no han podido clasificar gran cantidad de problemas. Por lo tanto, aún no existe respuesta a la pregunta de si estos problemas tienen o no solución.

En resumen, los capítulos siguientes presentan una amplia gama de temas. Estudiaremos temas tan abstractos como el poder computacional de las máquinas teóricas y las capacidades de los procesos computacionales. También veremos la relación entre estas ideas abstractas y situaciones tan reales como las técnicas de construcción de compiladores, el poder computacional de los lenguajes de programación, y preguntas con respecto a qué tan práctico es resolver ciertos problemas utilizando computadores.

Problemas de repaso del capítulo

1. Muestre que las siguientes igualdades (conocidas como leyes de DeMorgan) son verdaderas para tres conjuntos X, Y y Z cualesquiera. (Para demostrar que $A = B$, demuestre que $A \subseteq B$ y $B \subseteq A$).

$$(X - Y) \cap (X - Z) = X - (Y \cup Z)$$
$$(X - Y) \cup (X - Z) = X - (Y \cap Z)$$

2. Demuestre que para tres conjuntos X, Y y Z cualesquiera:
   a. $X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z)$
   b. $X \cup (Y \cap Z) = (X \cup Y) \cap (X \cup Z)$

3. Suponga que $X = \{x: x \in \mathbb{N} \text{ y } x \text{ es impar}\}$, $Y = \{y: y \in \mathbb{N} \text{ y } y \text{ es primo}\}$, y $Z = \{z: z \in \mathbb{N} \text{ y } z \text{ es múltiplo de tres}\}$. Describa cada uno de los conjuntos siguientes.

a. $X \cap Y$	d. $Z - Y$	g. $(Y \cap Z) - X$	j. $X \cup (Y \cap Z)$
b. $X \cap Z$	e. $X - Y$	h. $(X \cap Y) \cap Z$	
c. $Y \cap Z$	f. $X - (Y \cap Z)$	i. $X \cup Y$	

4. Presente una lista con todos los elementos de cada uno de los conjuntos siguientes.

   a. $\{x, y\} \times \{a, b, c\}$
   b. $\{a, b, c\} \times \{x, y\}$
   c. $\{x, y\} \times \{k\} \times \{a, b\}$
   d. $\{x, y\} \times \{\ \}$

5. Encuentre ejemplos de conjuntos no vacíos para los cuales sea verdadero cada uno de los enunciados siguientes.

   a. $X - Y = Y$ b. $A \times B = B \times A$ c. $P - Q = Q - P$

6. a. Defina una función $f: \mathbb{N} \to \mathbb{N}$ que sea uno a uno pero no sobre.
   b. Defina una función $f: \mathbb{N} \to \mathbb{N}$ que sea sobre pero no uno a uno.

7. Demuestre que $0 + 1 + 2 + \cdots + n = \dfrac{n(n+1)}{2}$ para todos los valores de n que pertenezcan a $\mathbb{N}$.

8. Demuestre que $1^2 + 2^2 + \cdots + n^2 = \dfrac{n(n+1)(2n+1)}{6}$ para todos los valores de n que pertenezcan a $\mathbb{N}^+$.

9. Demuestre que $1^3 + 2^3 + \cdots + n^3 = \dfrac{n^2(n+1)^2}{4}$ para todos los valores de n que pertenezcan a $\mathbb{N}^+$.

10. Demuestre que $2^2 + 4^2 + \cdots + (2n)^2 = \dfrac{2n(n+1)(2n+1)}{3}$ para todos los valores de n que pertenezcan a $\mathbb{N}^+$.

11. Demuestre que $1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$ para todos los valores de n que pertenezcan a $\mathbb{N}$.

12. Muestre que $x^n - y^n$ es divisible por $x - y$ para todo n en $\mathbb{N}^+$.

13. Muestre que si x es un número no negativo, entonces $(1 + X)^n - 1 \geq nx$ para todos los valores de n en $\mathbb{N}$.

14. Demuestre que $\dfrac{1}{2} + \dfrac{1}{6} + \cdots + \dfrac{1}{n(n+1)} = \dfrac{n}{n+1}$ para todos los valores de n que pertenezcan a $\mathbb{N}^+$.

15. Presente una lista de todos los elementos de $\mathcal{P}(\{x, y\})$. Luego, genere una lista de todos los elementos de $\mathcal{P}(\mathcal{P}(\{x, y\}))$. Muestre que $|\{x, y\}| \leq |\mathcal{P}(\{x, y\})| \leq |\mathcal{P}(\mathcal{P}(\{x, y\}))|$ identificando funciones uno a uno de $\{x, y\}$ a $\mathcal{P}(\{x, y\})$ y $\mathcal{P}(\{x, y\})$ a $\mathcal{P}(\mathcal{P}(\{x, y\}))$.

16. Muestre que la cardinalidad del conjunto de personas registradas en el sistema de seguridad social de Estados Unidos no es mayor que el conjunto de enteros de 1 000 000 000 a 1 999 999 999.

17. Muestre que la cardinalidad de $\mathcal{P}(X)$ es igual a la cardinalidad del conjunto de todas las funciones de X a $\{0, 1\}$.

18. Suponga que $f_1, f_2, \cdots$ es una lista infinita contable de funciones de $\mathbb{N}^+$ a $\mathbb{N}^+$. Defina la función $f: \mathbb{N} \to \mathbb{N}$ de manera que $f(n) = f_n(n) + 1$ para cada $n \in \mathbb{N}^+$. Demuestre que la función f no es una de las funciones de la lista original.

19. a. Proporcione un ejemplo de dos conjuntos disjuntos, no vacíos, A y B tales que $|A| < |B| < |A \cup B|$.
    b. Proporcione un ejemplo de dos conjuntos disjuntos, no vacíos, A y B tales que $|A| < |B| = |A \cup B|$.
    c. Proporcione un ejemplo de dos conjuntos disjuntos, no vacíos, A y B tales que $|A| = |B| = |A \cup B|$.

20. Tome su lenguaje de programación favorito e identifique los primeros tres programas que aparecerían en la lista descrita en los últimos párrafos de la sección 0.1.

21. Muestre que existe una cantidad por lo menos contable de frases en el idioma español.

22. Muestre que $|\mathbb{N}| = |\mathbb{N} \times \mathbb{N} \times \mathbb{N}|$.

23. Muestre que el conjunto de los números naturales pares tiene la misma cardinalidad que el conjunto de números naturales impares.

24. Muestre que la cardinalidad del conjunto de números primos es igual que la cardinalidad de $\mathbb{N}^+$.

25. Proporcione ejemplos para mostrar que la intersección de dos conjuntos incontables puede ser:
    a. finita
    b. infinita contable
    c. incontable

26. Muestre que si X es un conjunto incontable y Y es un conjunto contable, entonces $X - Y$ debe ser incontable.

27. Muestre que la unión de dos conjuntos contables es contable.

28. Muestre que un conjunto puede tener la misma cardinalidad que un subconjunto propio de sí mismo.

29. Muestre que el conjunto de todos los subconjuntos finitos de $\mathbb{N}$ es contable.

30. Muestre que el conjunto de todos los números reales es incontable.

31. Muestre que cualquier conjunto infinito contable tiene una cantidad de subconjuntos infinitos de los que dos cualesquiera tienen sólo un número finito de elementos en común.

CAPITULO 1

Autómatas finitos y lenguajes regulares

1.1 **Análisis léxico**
Diagramas de transiciones
Tablas de transiciones
1.2 **Autómatas finitos deterministas**
Definiciones básicas
Diagramas de transiciones deterministas
Ejemplos de autómatas finitos deterministas
1.3 **Límites de los autómatas finitos deterministas**
Autómatas finitos deterministas como aceptadores de lenguajes
Un lenguaje no regular
1.4 **Autómatas finitos no deterministas**
1.5 **Gramáticas regulares**
1.6 **Expresiones regulares**
1.7 **Comentarios finales**

En este capítulo definimos y estudiamos la clase de máquinas teóricas conocida como autómatas finitos. Aunque su poder es limitado, encontraremos que estas máquinas son capaces de reconocer numerosos patrones de símbolos, los cuales identificamos con la clase de los lenguajes regulares.

Los autómatas finitos y los lenguajes regulares se encuentran en el nivel más bajo de la jerarquía de máquinas y lenguajes que estudiaremos en los tres capítulos siguientes. Esta base establece la importancia de tales máquinas y lenguajes desde una perspectiva teórica, pero su importancia no se limita a la teoría. Los conceptos que presentan los autómatas finitos y los lenguajes regulares son de interés fundamental para la mayoría de las aplicaciones que requieren técnicas de reconocimiento de patrones.

Una de estas aplicaciones es la construcción de compiladores. Por ejemplo,

un compilador debe ser capaz de reconocer cuáles son las cadenas de símbolos del programa fuente que deben considerarse como representaciones de objetos individuales, por ejemplo nombres de variables, constantes numéricas y palabras reservadas. Esta tarea de reconocimiento de patrones es manejada por el analizador léxico del compilador. En este capítulo se considerarán los principios básicos que rigen la construcción de analizadores léxicos; estos principios serán la base para una gran parte de nuestro estudio posterior.

1.1 ANÁLISIS LÉXICO

Abordemos ahora un problema al que se enfrenta un compilador: detectar si una cadena del programa fuente representa o no un nombre de variable aceptable. En un lenguaje de programación típico, estos nombres comienzan con una letra, seguida por una combinación arbitraria (pero finita) de letras y dígitos. Así, *X25*, *PepeRosas* y *x2y3z* serían nombres aceptables, mientras que *25*, *x.h* y *Pepe-Rosas* no lo serían. Además, cualquier estructura léxica en un lenguaje de programación termina con un conjunto de símbolos reconocido como fin de la estructura. A estos símbolos los llamaremos **marcas de fin de cadena**. En el caso de nombres de variables, estas marcas podrían ser espacios, puntos y comas, y retorno de carro.

Diagramas de transiciones

Para desarrollar una unidad de programa que reconozca las ocurrencias de nombres de variables, nuestro primer paso podría ser representar de una manera concisa, no ambigua, la estructura de un nombre aceptable. Para este fin podemos utilizar la forma gráfica de un **diagrama de transiciones** (también llamado **diagrama de estado** o, en el campo del procesamiento de lenguajes naturales, **red de transiciones**), como se muestra en la figura 1.1. Un diagrama de transiciones es una colección finita de círculos, los cuales se pueden rotular para fines de referencia, conectados por flechas que reciben el nombre de **arcos**. Cada uno de estos arcos se etiqueta con un símbolo o categoría de símbolos (p. ej., "dígito" o "letra") que podría presentarse en la cadena de entrada que se analiza. Uno de los círculos se designa con un apuntador, y representa una posición inicial. Además, por lo menos uno de los círculos se representa como un círculo doble; estos círculos dobles designan posiciones del diagrama en las cuales se ha reconocido una cadena válida.

Decimos que una cadena de símbolos es aceptada por un diagrama de transiciones si los símbolos que aparecen en la cadena (de izquierda a derecha) corresponden a una secuencia de arcos rotulados que conducen del círculo designado por el apuntador a un círculo doble. Así, al analizar la cadena *X25* utilizando la figura 1.1, partimos del círculo inicial siguiendo el arco con

Análisis léxico
(tokenizador)

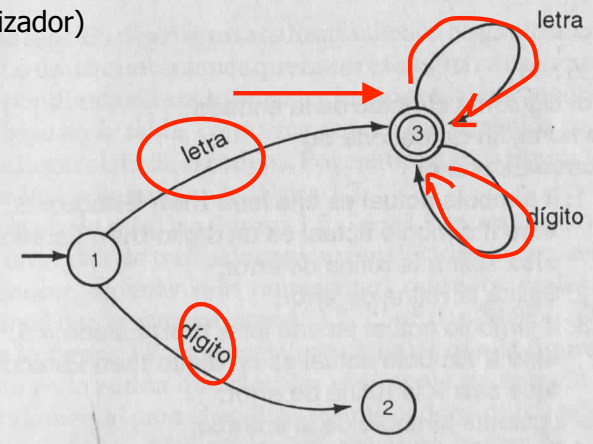


Figura 1.1 Diagrama de transiciones que representa la sintaxis de un nombre de variable

etiqueta "letra" (ya que la cadena comienza con *X*). A partir del círculo 3, recorremos dos veces (una para el 2 y otra para el 5) el arco llamado "dígito y nos encontramos en el doble círculo 3. Por el contrario, si utilizamos la cadena 25, comenzamos en el mismo círculo inicial, pero esta vez seguimos el arco rotulado "dígito" y nos encontramos en un callejón sin salida. Concluimos entonces que *X25* es un nombre de variable aceptable, no así 25.

Observe que los círculos en un diagrama de transiciones representan posiciones, o estados, donde nos podemos encontrar al evaluar una cadena de símbolos. Haciendo referencia a la figura 1.1, vemos que el círculo 1 se puede indentificar como el estado inicial del análisis de una cadena que el círculo 2 representa el estado de haber localizado un dígito como primer símbolo de la cadena, y que el círculo 3 representa el estado de haber encontrado una cadena que comienza con una letra. Por esto, es común llamar **estados** a los círculos de un diagrama de transiciones. El círculo de partida se llama **estado inicial** y los círculos dobles, estados de **aceptación**.

Una vez que hemos desarrollado un diagrama de transiciones que acepta únicamente las estructuras sintácticas que constituyen un nombre de variable válido, no es difícil escribir un programa que reconozca estas estructuras. Por ejemplo, el diagrama de la figura 1.1 sugiere el segmento de programa que se muestra en la figura 1.2. Observe que la estructura básica del algoritmo producido es la de un solo enunciado case que dirige las actividades de acuerdo con el estado actual. Las opciones posibles para cada estado se manejan utilizando estructuras condicionales adicionales, como los enunciados if-then-else anidados. En el caso de una cadena inaceptable, el segmento de programa sale a una rutina de error que, por medio de un parámetro, puede emplearse para imprimir un mensaje adecuado al respecto, como "El nombre de la variable debe comenzar con una letra" o "Identificador no válido".

```
Estado := 1;
Leer el siguiente símbolo de la entrada;
while no es fin-de-cadena do
    case Estado of
      1: if símbolo actual es una letra then Estado:= 3,
         else if símbolo actual es un dígito then Estado:= 2,
         else salir a la rutina de error;
      2: Salir a la rutina de error;
      3: if símbolo actual es una letra then Estado:= 3,
         else if símbolo actual es un dígito then Estado:= 3,
         else salir a la rutina de error;
Leer el siguiente símbolo de la entrada;
end while;
if Estado no es 3 then salir a la rutina de error;
```

Figura 1.2 Serie de instrucciones sugerida por el diagrama de transiciones de la figura 1.1

Vemos entonces que los diagramas de transiciones se pueden emplear como herramientas de diseño para producir rutinas de análisis léxico, de manera similar a como los ingenieros de software utilizan los diagramas de flujo y de estructura. No obstante, el código generado directamente a partir de un diagrama de transiciones no siempre representa la mejor solución al problema; se obtiene una solución más elegante si se emplean tablas de transiciones.

Tablas de transiciones

Una **tabla de transiciones** es un arreglo (o matriz) bidimensional cuyos elementos proporcionan el resumen de un diagrama de transiciones correspondiente. Para elaborar una tabla de este tipo construimos primero un arreglo, con una fila para cada estado del diagrama de transiciones y una columna para cada símbolo o categoría de símbolos que podría ocurrir en la cadena de entrada. El elemento que se encuentra en la fila m y la columna n es el estado que se alcanzaría en el diagrama de transiciones al dejar el estado m a través de un arco con etiqueta n. Si no existe arco n alguno que salga del estado m, entonces la casilla correspondiente de la tabla se marca como un estado de error. Con la finalidad de completar la tabla de transiciones, agregamos una columna rotulada "FDC" para el fin de cadena. La casilla en la columna FDC contiene el valor "aceptar" si la fila corresponde a un estado de aceptación del diagrama, o contiene el valor "error", en caso contrario. La figura 1.3 representa una tabla de transiciones obtenida a partir del diagrama de transiciones de la figura 1.1.

Es bastante sencillo diseñar un analizador léxico basándose en una tabla de transiciones. Lo único que tenemos que hacer es asignar a una variable un valor inicial, correspondiente al estado inicial, y luego actualizar repetidamente esta variable con base en la tabla, conforme se leen los símbolos de la cadena de entrada, hasta llegar al fin de la cadena. Por ejemplo, en la figura 1.4 se presenta un analizador léxico basado en la figura 1.3.

Como un ejemplo más, las figuras 1.5, 1.6 y 1.7 muestran un diagrama de transiciones, una tabla de transiciones y un analizador léxico, respectivamente, para reconocer cadenas que representan números reales positivos en notación decimal o exponencial, como 35.7, 2.56*E*10 o 34.0*E*-7. Observe que la estructura de la figura 1.7 es esencialmente igual que la figura 1.4; la única diferencia está en la rutina que clasifica el símbolo de entrada. De hecho, el algoritmo fundamental para el análisis de una cadena utilizando una tabla de transiciones es el mismo para cualquier estructura léxica.

En resumen, hemos presentado algunas técnicas bastante sencillas para desarrollar analizadores léxicos. Ahora, lo que necesitamos descubrir es el grado en el cual pueden aplicarse estas técnicas. ¿Es posible utilizar segmentos de programa basados en diagramas de transiciones para analizar cualquier estructura sintáctica, o existen estructuras que no pueden reconocerse con estos diagramas? Si existen límites para estas técnicas, ¿qué se requiere para manejar los casos más complejos? Atenderemos estas preguntas en las secciones restantes de este capítulo, y continuaremos nuestra búsqueda por los capítulos 2 y 3.

	letra	dígito	FDC
1	3	2	error
2	error	error	error
3	3	3	aceptar

Figura 1.3 Tabla de transiciones construida a partir del diagrama de transiciones de la figura 1.1

```
Estado : = 1;
repeat
    Leer el siguiente símbolo del flujo de entrada;
    case símbolo of
      letra: Entrada : = "letra";
      dígito: Entrada : = "dígito";
      marca de fin de cadena: Entrada : = "FDC";
      ninguno de los anteriores: salir a la rutina de error;
    Estado : = Tabla [Estado, Entrada];
    if Estado = "error" then salir a la rutina de error;
  until Estado = "aceptar"
```

Figura 1.4 Analizador léxico basado en la tabla de transiciones de la figura 1.3

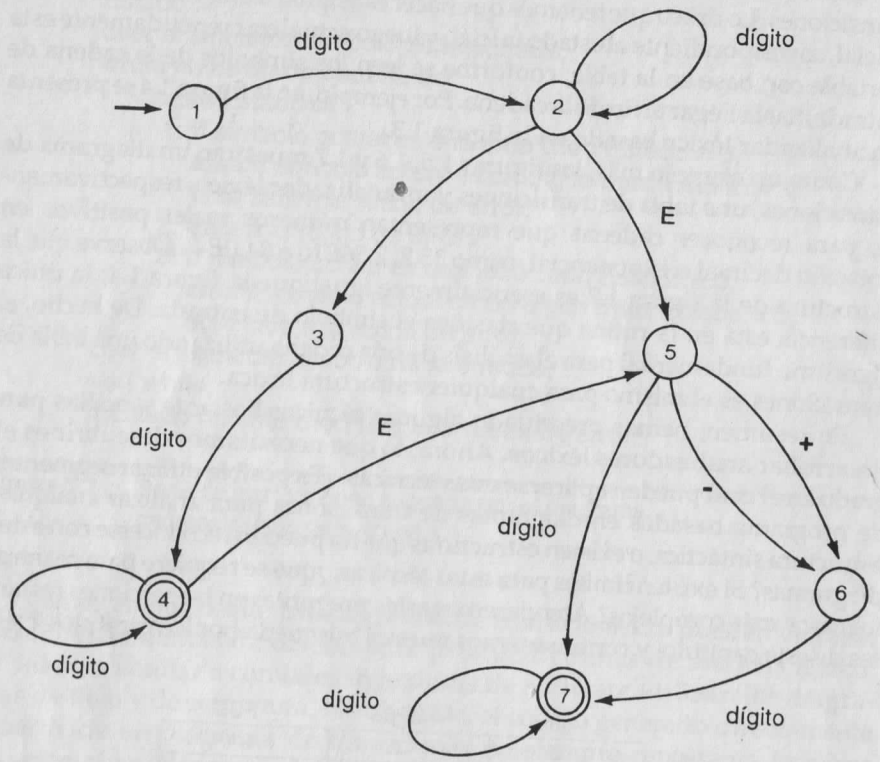


Figura 1.5 Diagrama de transiciones que representa la sintaxis de un número real

	dígito	•	E	+	-	FDC
1	2	error	error	error	error	error
2	2	3	5	error	error	error
3	4	error	error	error	error	error
4	4	error	5	error	error	aceptar
5	7	error	error	6	6	error
6	7	error	error	error	error	error
7	7	error	error	error	error	aceptar

Figura 1.6 Tabla de transiciones construida a partir del diagrama de transiciones de la figura 1.5


```
Estado: = 1;
repeat
   Leer el siguiente símbolo del flujo de entrada;
   case símbolo of
      0 a 9: Entrada := "dígito";
      Marca de fin de cadena: Entrada : = "FDC";
      •, E, +, - Entrada := símbolo;
      Ninguno de los anteriores: salir a la rutina de error;
   Estado : = Tabla [Estado, Entrada];
   if Estado = "error" then salir a la rutina de error;
until Estado = "aceptar"
```

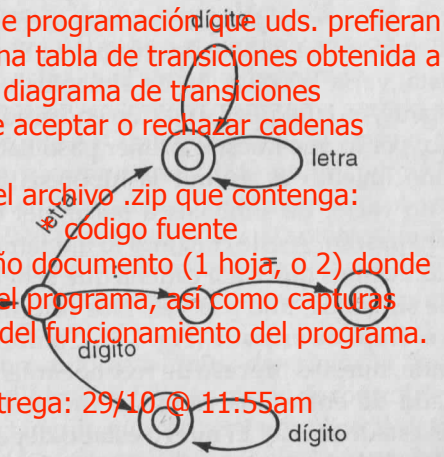
Figura 1.7 Analizador léxico basado en la tabla de transiciones de la figura 1.6

Tarea 1.3. Escribir un programa que reconozca un AFD a partir de su tabla de transiciones.

Ejercicios

1. Diseñe un diagrama de transiciones para reconocer expresiones aritméticas de longitud arbitraria que comprenden enteros positivos separados por signos de suma, resta, multiplicación o división.
2. Escriba un analizador léxico directamente a partir del siguiente diagrama de transiciones.

* Lenguaje de programación que uds. prefieran
* Debe contener una tabla de transiciones obtenida a partir del diagrama de transiciones
* Debe aceptar o rechazar cadenas



Subir el archivo .zip que contenga:
código fuente
* un pequeño documento (1 hoja, o 2) donde describa el programa, así como capturas de pantalla del funcionamiento del programa.

Entrega: 29/10 @ 11:55am

3. Construya una tabla de transiciones a partir del diagrama del ejercicio 2 y escriba un analizador léxico basado en esa tabla.
4. Muestre que se puede modificar un diagrama de transiciones que contiene un arco rotulado por una cadena de símbolos de longitud dos o más (lo cual significa que para recorrer el arco se requiere la cadena completa y no un solo símbolo) para que contenga sólo arcos rotulados con símbolos sencillos, pero de manera que siga aceptando las mismas cadenas que antes.

1.2 AUTÓMATAS FINITOS DETERMINISTAS

Concluimos la sección anterior preguntando si los diagramas de transiciones proporcionan una herramienta con el poder suficiente para desarrollar programas que reconozcan estructuras sintácticas de complejidad arbitraria. En esta sección atendemos esta pregunta, formalizando con mayor precisión el proceso de reconocimiento de estructuras por medio de diagramas de transiciones. Nuestra meta es identificar las características pertinentes de un sistema de reconocimiento de patrones construido sobre el principio de los diagramas de transiciones, para que podamos estudiar de manera genérica el potencial de un sistema de reconocimiento de patrones en vez de presentar cada ejemplo como un caso aislado

Definiciones básicas

Para iniciar esta tarea de formalización, reconocemos que las cadenas que deben analizarse en una aplicación están construidas a partir de un conjunto de símbolos. En el caso de un analizador léxico en un compilador, estas cadenas generalmente están formadas por los símbolos disponibles en el teclado de una terminal de computador. Sin embargo, en un computador digital moderno todos estos símbolos están representados por patrones de ceros y unos. A partir de esta perspectiva más elemental, cualquier cadena consiste en una combinación de sólo dos símbolos. Por lo tanto, dependiendo del punto de vista, varía la colección de símbolos utilizada para construir cadenas. No obstante, en cualquier situación encontramos que el conjunto de símbolos es finito, por lo que nuestro primer paso hacia la formalización del proceso de reconocimiento es asumir la hipótesis de la existencia de un conjunto finito, no vacío, de símbolos a partir del cual se construyen las cadenas que se analizarán. A este conjunto lo llamamos **alfabeto.**

A continuación vemos que cada cadena que se recibe se analizará como una secuencia de símbolos, uno a la vez. Nos referimos a la fuente de esta secuencia como el **flujo de entrada** (Fig. 1.8). Conforme llega cada símbolo del flujo de entrada, nuestro proceso de reconocimiento implica cambiar de un estado, tomado de entre una cantidad finita de ellos, a otro, o bien permanecer en el estado actual. El nuevo estado dependerá únicamente del estado actual y del símbolo que se recibe. Para subrayar este punto, observe que, una vez que el proceso de reconocimiento ha llegado al estado 5 de la figura 1.5, su estado siguiente no dependerá de si ha llegado al estado 5 proveniente del estado 2 o del estado 4; en cambio, el estado siguiente estará determinado sólo por el siguiente símbolo que se reciba y por estar en el estado 5. En análisis subsecuentes se verá la importancia de que los sistemas de reconocimiento basados en estos diagramas de transiciones no puedan recordar cómo llegaron al lugar donde se encuentran.

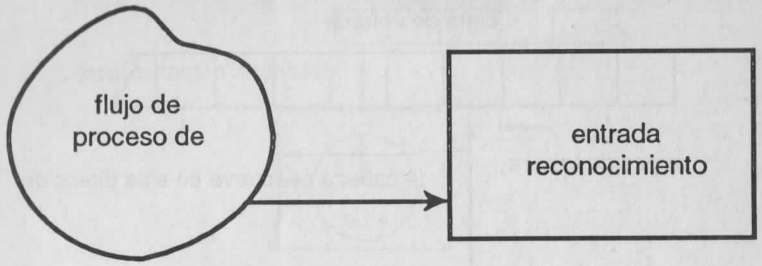


Figura 1.8 Flujo de entrada como fuente de símbolos.

Los conceptos que tenemos hasta ahora se fortalecen definiendo una máquina conceptual conocida como **autómata finito determinista.** Esta máquina consiste en un dispositivo que puede estar en cualquiera de un número finito de estados, uno de los cuales es el estado inicial y por lo menos uno es un estado de aceptación. A este dispositivo está unido un flujo de entrada por medio del cual llegan en secuencia los símbolos de un alfabeto determinado. La máquina tiene la capacidad para detectar los símbolos conforme llegan y, basándose en el estado actual y el símbolo recibido, ejecutar una **transición** (de estado) que consiste en un cambio a otro estado o la permanencia en el estado actual. La determinación de cuál será precisamente la transición que ocurra al recibir un símbolo depende de un **mecanismo de control** de la máquina, programado para conocer cuál debe ser el nuevo estado dependiendo de la combinación del estado actual y el símbolo de entrada.

Debemos hacer hincapié en que el programa de un autómata finito determinista no debe contener ambigüedades, lo mismo que sucede con cualquier programa para computador. Éste es un punto importante que analizaremos pronto con mayor detalle. Por el momento, observamos que ésta es la razón detrás de la palabra "determinista" en "autómata finito determinista". La palabra "finito" se refiere a que la máquina sólo tiene un número finito de estados. En ocasiones se hace mención de estas máquinas como autómatas deterministas de estado finito.

Tradicionalmente, un autómata finito determinista se visualiza de la manera presentada en la figura 1.9. El mecanismo de control de la máquina está representado por un rectángulo que contiene una especie de carátula de reloj. Los estados posibles están representados en la circunferencia de este reloj y un apuntador indica el estado actual. El flujo de entrada a la máquina aparece como una cinta de papel dividida en celdas, cada una de las cuales es capaz de almacenar un solo símbolo. Consideramos que la cinta tiene un extremo izquierdo, pero que se extiende infinitamente hacia la derecha. La máquina es capaz de leer los símbolos de esta cinta, de izquierda a derecha, por medio de una cabeza de lectura cuya posición en la figura 1.9 está indicada por una flecha. Cada vez que se lee un símbolo, la cabeza de lectura se mueve a la

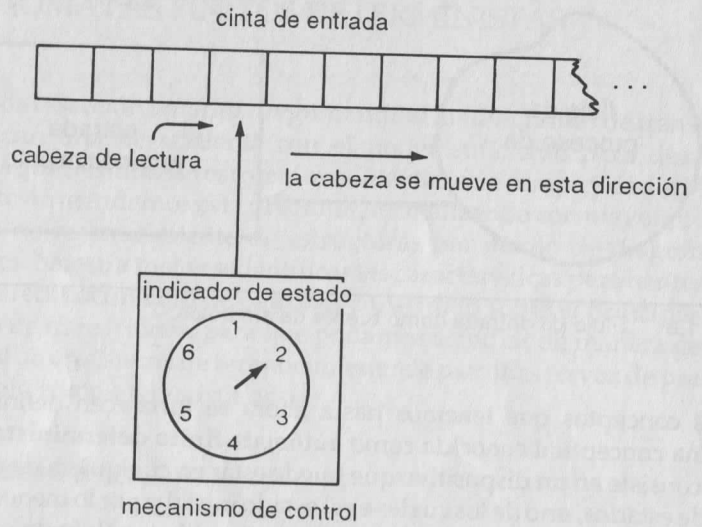


Figura 1.9 Representación de un autómata finito determinista

siguiente posición de la cinta. Así, la posición donde descansa la cabeza de lectura corresponde a la siguiente celda que se leerá.

Para representar un programa en el mecanismo de control, utilizamos un diagrama de transiciones cuyos estados representan los estados de la máquina y cuyos arcos representan una posible transición de la máquina. En este contexto, los estados de inicio y aceptación del diagrama corresponden a los estados de inicio y de aceptación del autómata.

Decimos que un autómata finito determinista acepta su cadena de entrada si, después de comenzar sus cálculos en el estado inicial con la cabeza de lectura sobre el primer símbolo de la entrada, la máquina cambia a un estado de aceptación después de leer el último símbolo de la cadena (véase Fig.1.10). Si después de leer el último símbolo de la cadena la máquina no queda en un estado de aceptación, decimos que la cadena ha sido rechazada. Por ende, un autómata finito determinista es, en esencia, una máquina analizadora de cadenas que acepta aquellas cadenas aceptadas por su diagrama de transiciones interno y rechaza todas las demás.

Surge un caso ligeramente distinto si la máquina llega al final de su entrada antes de leer algún símbolo (es decir, si la entrada comienza con una marca de fin de cadena). En este caso decimos que la entrada es una cadena vacía (una cadena que no contiene símbolos). Por desgracia, una cadena vacía no aparece muy bien en una página impresa, por lo que la representamos con el símbolo λ. Recalcamos que la cadena vacía no contiene símbolos; por lo tanto, un autómata finito determinista aceptará λ si y sólo si su estado inicial es también un estado de aceptación.

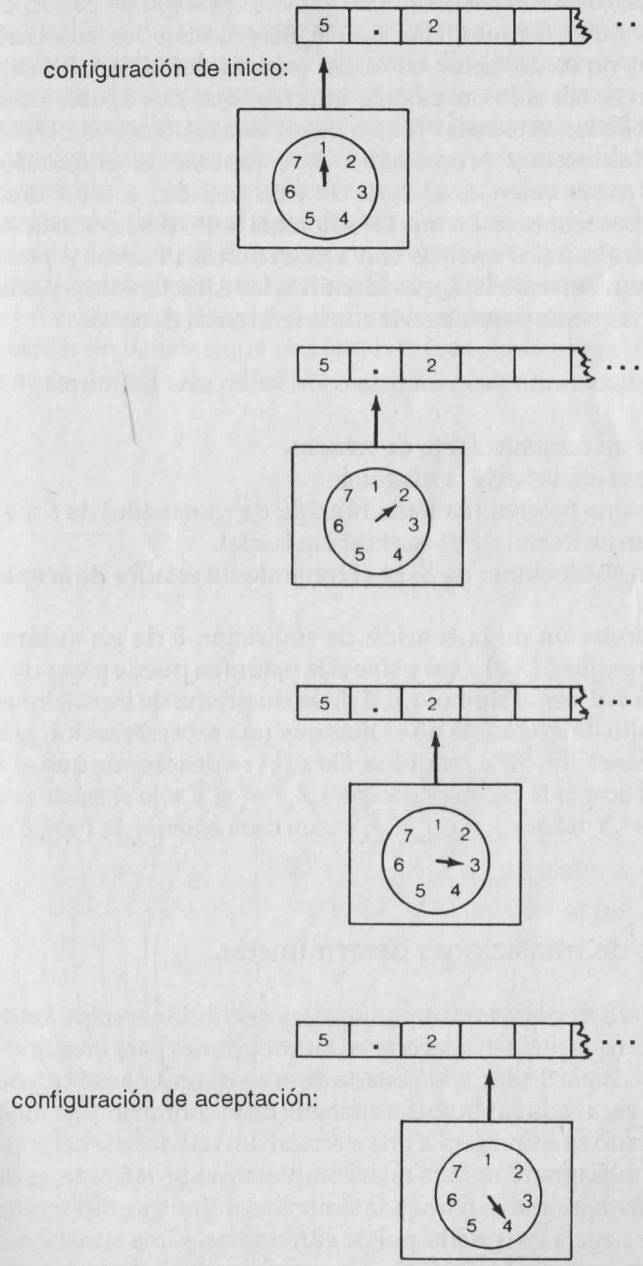


Figura 1.10 Pasos efectuados por un autómata finito determinista (programado por el diagrama de la Fig. 1.5) al procesar la cadena "5.2"

Por último, debemos hacer notar que la tecnología de cintas, cabezas de lectura y carátulas de reloj con la cual describimos los autómatas finitos deterministas no es un factor crítico en nuestra definición. En cambio, esta implantación es tan sólo un modelo informal que nos ayuda a recordar las propiedades de los autómatas finitos deterministas. Pueden construirse máquinas con las mismas propiedades de computación utilizando diversas tecnologías, como veremos al final de esta sección, y cada una de estas máquinas debe reconocerse como un autómata finito determinista. Así, reanudaremos ahora nuestro análisis con una definición formal y precisa de un autómata finito determinista, que identifica las características pertinentes de estas máquinas y que puede servir como referencia decisiva.

Un autómata finito determinista consiste en una quíntupla $(S, \Sigma, \delta, \iota, F)$ donde:

a. S es un conjunto finito de estados.
b. Σ es el alfabeto de la máquina.
c. δ es una función (llamada **función de transición**) de $S \times \Sigma$ a S.
d. ι (un elemento de S) es el estado inicial.
e. F (un subconjunto de S) es el conjunto de estados de aceptación.

La interpretación de la función de transición δ de un autómata finito determinista es que $\delta(p, x) = q$ si y sólo si la máquina puede pasar de un estado p a un estado q al leer el símbolo x. Así, un diagrama de transiciones para un autómata finito determinista no es más que una representación gráfica de la función de transición de la máquina. De aquí se desprende que el autómata $(S, \Sigma, \delta, \iota, F)$ acepta la cadena no vacía $x_1x_2 \cdots x_n$ si y sólo si existe una serie de estados $s_0, s_1, \cdots, s_n$ tal que $s_0 = \iota$, $s_n \in F$, y para cada entero j de 1 a n, $\delta(s_{j-1}, x_j) = s_j$.

Diagramas de transiciones deterministas

El requisito del determinismo impone ciertas restricciones sobre los diagramas de transiciones que pueden aparecer en los programas para un autómata finito determinista. En particular, cada estado de estos diagramas sólo debe tener un arco que sale para cada símbolo del alfabeto; de lo contrario, una máquina que llega a ese estado se enfrentará a una elección de cuál debe ser el arco a seguir. Además, dicho diagrama deberá estar **completamente definido**, es decir, debe existir por lo menos un arco para cada símbolo del alfabeto; de lo contrario, una máquina que llega a ese estado puede enfrentarse a una situación donde no puede aplicarse ninguna transición. Observe que estos requisitos están reflejados en nuestra definición formal, primero porque δ es una función y, segundo porque su dominio es $S \times \Sigma$.

Se dice que un diagrama de transiciones es **determinista** si cumple estas dos condiciones. Entonces, desde un punto de vista técnico, el diagrama de la figura 1.1 no es determinista ya que no está completamente definido: no representa cuál será la acción que debe ocurrir si se recibe una letra o un dígito mientras se encuentre en el estado 2. El diagrama de la figura 1.5 tiene problemas similares ya que, entre otras cosas, no describe lo que deberá suceder si recibe un punto mientras se encuentra en el estado 1. No obstante, los dos diagramas no tienen más de un arco de salida de un estado para cada símbolo y, por consiguiente, pueden modificarse para ajustarse a los requisitos del determinismo. Primero añadimos un estado adicional que representará un papel de captación global. Luego, para cada símbolo del alfabeto, dibujamos un arco, rotulado con dicho símbolo, que empieza y termina en este nuevo estado. Por último, agregamos arcos de los otros estados a este nuevo, hasta que cada uno de los estados sea el origen de un arco para cada símbolo del alfabeto.

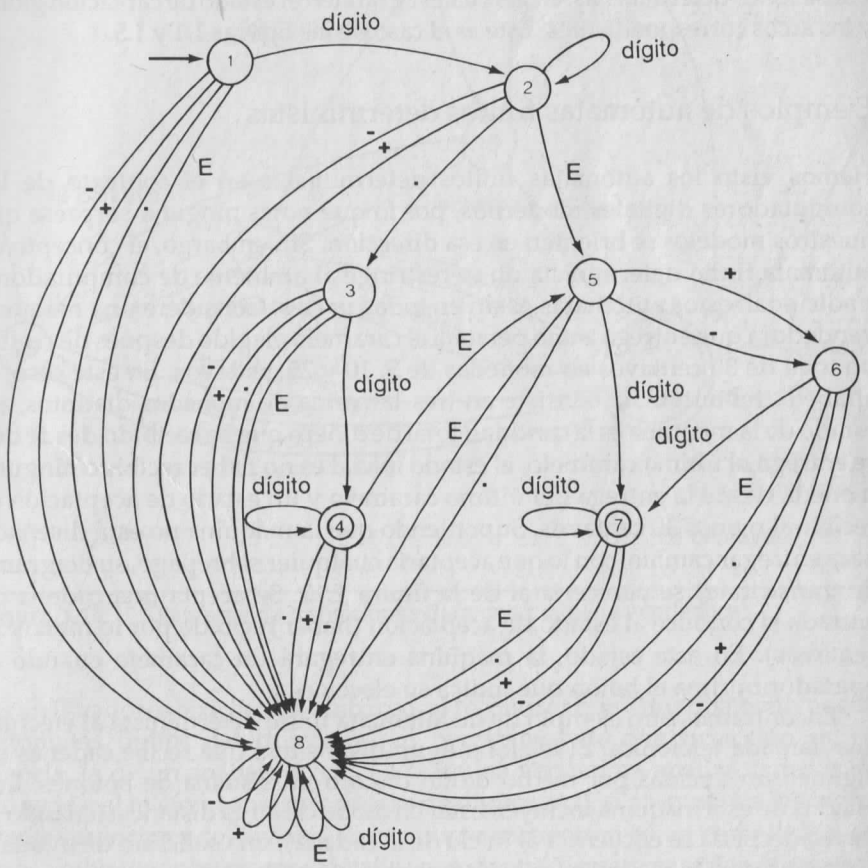


Figura 1.11 Diagrama "completo" de la figura 1.5

En la figura 1.11 se muestra el resultado de aplicar esta técnica al diagrama de transiciones de la figura 1.5. El nuevo estado es el número 8. Observe que en el diagrama original la ocurrencia de una cadena inaceptable ocasionaba un error al solicitar el recorrido de un arco inexistente. En el diagrama modificado, una cadena inaceptable ocasiona que la máquina recorra un arco al estado 8, donde permanece hasta alcanzar el final de la cadena de entrada. Al llegar a este punto se rechazará la cadena, ya que el estado 8 no es de aceptación. Por esto, los dos diagramas son equivalentes en lo que se refiere a que aceptan las mismas cadenas; difieren tan sólo en la manera en que llegan a sus conclusiones. La figura 1.11 también demuestra por qué, al dibujar diagramas de transiciones deterministas, nos vemos tentados a no representar todas las transiciones. De hecho, si se incluyen todas las transiciones, fácilmente saturamos los diagramas, lo que a su vez oculta la estructura significativa del autómata que se representa. Por consiguiente, con frecuencia se dibujan versiones parciales, o esqueletos, de los diagramas de transiciones deterministas, en los cuales se omiten el estado de captación global y los arcos correspondientes. Este es el caso de las figuras 1.1 y 1.5.

Ejemplos de autómatas finitos deterministas

Hemos visto los autómatas finitos deterministas en el contexto de los computadores digitales modernos, por lo que no es ninguna sorpresa que nuestros modelos se orienten en esa dirección. Sin embargo, el concepto de autómata finito determinista no se restringe al ambiente de computadores tradicional: estos autómatas están en todas partes. Considere una máquina vendedora que entrega a una persona el caramelo elegido después de recibir un total de 30 centavos en monedas de 5, 10 y 25 centavos. En este caso, el alfabeto del autómata consiste en tres tamaños de monedas distintos, un estado de la máquina es la cantidad total de dinero que ha recibido desde que se entregó el último caramelo, el estado inicial es no haber recibido ninguna moneda desde la entrega del último caramelo y un estado de aceptación es recibir al menos 30 centavos. Suponiendo que la máquina no está diseñada para entregar cambio, con lo que aceptaría cualquier sobrepago, su diagrama de transiciones se parecería al de la figura 1.12. Se acepta una cadena de entrada si conduce al estado de aceptación (haber recibido por lo menos 30 centavos). En este estado, la máquina entregará un caramelo cuando el operador oprima el botón que indica su elección.

Encontramos otro ejemplo de un autómata finito determinista al efectuar una llamada telefónica. El teléfono es un dispositivo que recibe cadenas de dígitos introducidas por medio de un disco o un sistema de botones. Los estados de esta máquina incluyen estar en modo de larga distancia (cuando la clave adecuada se encuentra al inicio de la cadena), una solicitud de ayuda a la operadora o la recepción de un número telefónico válido.

Hemos considerado estos ejemplos de autómatas finitos deterministas distintos a los computadores para tratar de ampliar nuestra perspectiva más allá de

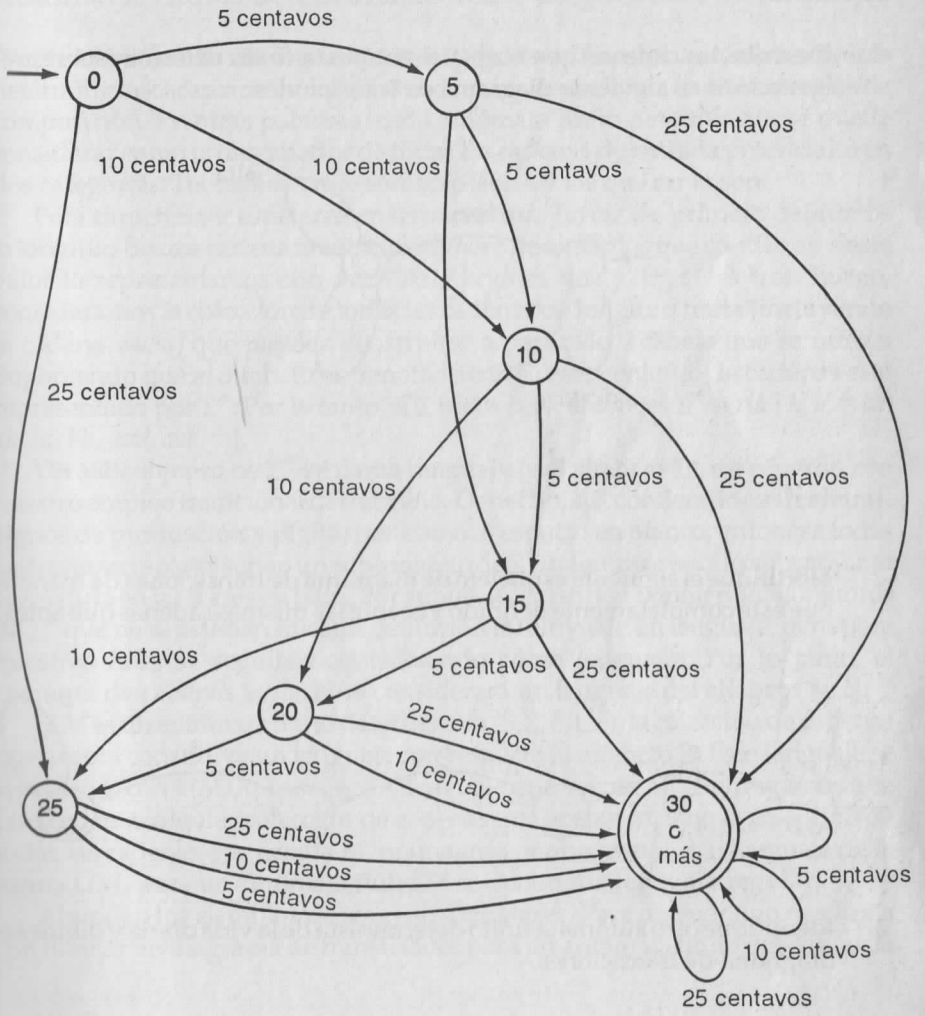
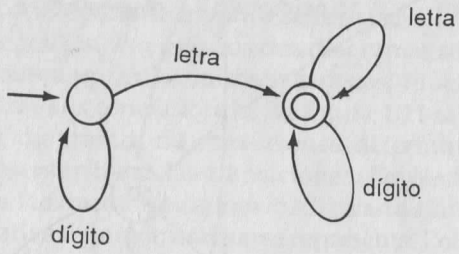


Figura 1.12 Diagrama de transiciones para una máquina vendedora

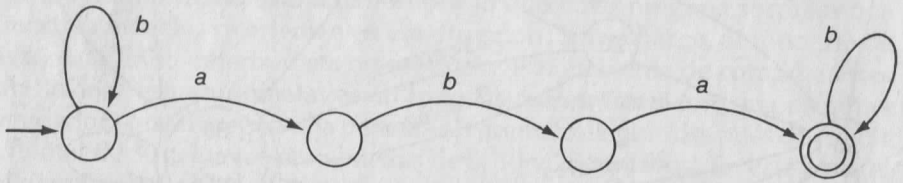
los analizadores léxicos. Sin embargo, al reconocer que estos dispositivos son autómatas finitos deterministas nos percatamos de que su acción es, en esencia, la de un análisis léxico. La máquina vendedora analiza cadenas de monedas; el teléfono analiza cadenas de dígitos. Así, si abstraemos los ingredientes esenciales del análisis léxico que se encuentran en un compilador, en otros ambientes observamos detalles que de otra manera pasarían desapercibidos. Aunque este ejemplo específico no tiene una trascendencia significativa, sí constituye una muestra del poder de las teorías abstractas.

Ejercicios

1. Describa las cadenas que acepta el autómata finito determinista representado en el siguiente diagrama de transiciones.



2. Modifique el siguiente esqueleto de diagrama de transiciones de manera que esté completamente definido y acepte las mismas cadenas que antes.



3. Identifique otro autómata finito determinista de la vida diaria y dibuje su diagrama de transiciones.

1.3 LÍMITES DE LOS AUTÓMATAS FINITOS DETERMINISTAS

Hasta ahora hemos presentado la clase de máquinas conocidas como autómatas finitos deterministas y hemos analizado su relación con los analizadores léxicos. Ahora, nuestro problema es determinar el grado hasta el cual pueden construirse algoritmos de reconocimiento de patrones a partir de estos principios. De manera específica, lo que nos preguntamos es si el empleo de tablas de transiciones, como se presentaron en la sección 1.1, ofrece la flexibilidad suficiente para el procesamiento general de cadenas.

Autómatas finitos deterministas como aceptadores de lenguajes

Reconsideremos la tarea que queremos que realicen los autómatas finitos deterministas: aceptar sólo aquellas cadenas que se adhieran a ciertas reglas de composición. En otras palabras, cada autómata finito determinista se puede considerar como un agrupador de todas las cadenas de entrada potenciales en dos categorías: las cadenas que son aceptables y las que no lo son.

Para caracterizar esta tarea en términos más formales, primero definimos la longitud de una cadena w como el número de símbolos que contiene y a este valor lo representamos con $|w|$. Así, $|xy|$ es dos y $|xyz|$ es tres. Luego, consideramos la colección de todas las cadenas de longitud finita (incluyendo la cadena vacía) que pueden construirse a partir del alfabeto que se utiliza. Suponiendo que el alfabeto es denotado con Σ, este conjunto de cadenas está representado por Σ^*. Por lo tanto, si Σ fuera $\{a,b\}$, entonces Σ^* sería $\{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \cdots\}$.

Un subconjunto de Σ^* se llama **lenguaje** (del alfabeto Σ), de acuerdo con nuestro empleo tradicional del término. De hecho, si Σ contiene todas las letras, signos de puntuación y dígitos, así como el espacio en blanco, entonces todas las frases en español serían un subconjunto de Σ^*, mientras que otro subconjunto consistiría en las frases en latín. Por supuesto, existirían también subconjuntos de Σ^* que no satisfarían nuestra definición intuitiva de un lenguaje, pero para nuestros fines se seguirán considerando como lenguajes. Por lo tanto, el conjunto de cadenas $\{a, ab, b\}$ se considerará un lenguaje del alfabeto $\{a, b\}$.

Si M es un autómata finito determinista $(S,\Sigma,\delta,\iota,F)$, la colección de cadenas que acepta constituye un lenguaje con respecto al alfabeto Σ. Este lenguaje se representa con $L(M)$, que se lee "el lenguaje que acepta M". Subrayamos que $L(M)$ no es cualquier colección de cadenas que acepta M, sino la colección de todas las cadenas que acepta M, ni una más ni una menos. Un lenguaje de la forma $L(M)$ para un autómata finito M se denomina **lenguaje regular**.

Abundan los ejemplos de lenguajes regulares. Para obtener uno nos basta con dibujar un diagrama de transiciones para un autómata finito determinista

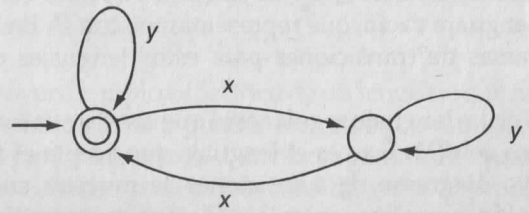


Figura 1.13 Diagrama de transiciones para un autómata finito determinista que acepta exactamente aquellas cadenas de x y y que contienen un número par de x

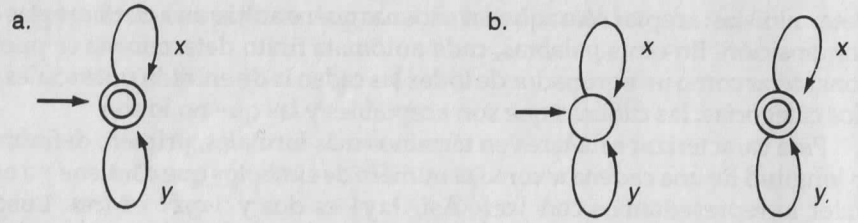


Figura 1.14 Diagramas de transiciones para autómatas finitos deterministas que aceptan a) el lenguaje $\{x, y\}^*$ y b) el lenguaje $\varnothing$

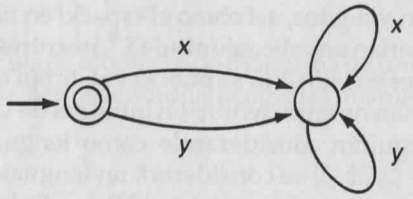


Figura 1.15 Diagrama de transiciones para un autómata finito determinista que acepta el lenguaje $\{\lambda\}$

y luego identificar el lenguaje que acepta. Como muestra, la figura 1.13 presenta un diagrama de transiciones que acepta exactamente aquellas cadenas de x y y que contienen un número par de x y cualquier número de y. Así, este lenguaje (del alfabeto $\{x,y\}$) es regular. Un par de lenguajes regulares algo especiales es la colección de todas las cadenas de longitud finita de un alfabeto Σ, que ya representamos con Σ^*, y la colección que no contiene cadenas, conocida como **lenguaje vacío**, que representamos con $\varnothing$. En la figura 1.14 se presentan diagramas de transiciones para estos lenguajes con respecto al alfabeto $\{x, y\}$.

Otro ejemplo de un lenguaje regular es el que consiste en una cadena vacía, el cual denotamos con $\{\lambda\}$. Éste es el lenguaje que acepta el autómata finito determinista cuyo diagrama de transiciones se muestra en el figura 1.15. Observe la distinción que se hace entre $\{\lambda\}$ y $\varnothing$: el primero contiene una cadena, mientras que el segundo no contiene ninguna.

Nótese que los lenguajes regulares son aquellos para los cuales son aplicables las técnicas de análisis léxico de la sección 1.1. Por esto, el teorema siguiente, que establece la existencia de lenguajes no regulares, tiene importancia con respecto a nuestra pregunta acerca del poder de esta técnicas. En resumen, las técnicas de análisis léxico de la sección 1.1 no tienen la flexibilidad suficiente para el procesamiento general de cadenas.

TEOREMA 1.1
Para cualquier alfabeto Σ, existe un lenguaje que no es igual a $L(M)$ para cualquier autómata finito determinista M.

DEMOSTRACION
Puesto que cualquier arco de un autómata finito determinista que esté rotulado con un símbolo fuera de Σ no tendrá efecto alguno sobre el procesamiento de una cadena en Σ^*, podemos considerar sólo las máquinas con alfabeto Σ. Sin embargo, la colección de autómatas finitos deterministas con alfabeto Σ es contable ya que podemos elaborar en forma sistemática una lista de todas las máquinas posibles con un estado, seguidas por todas las máquinas con dos estados, luego por aquéllas con tres estados, etcétera. Por otra parte, el número de lenguajes con respecto al alfabeto Σ es incontable, ya que el conjunto infinito Σ^* tiene un número incontable de subconjuntos. Por lo tanto, hay más lenguajes que autómatas finitos deterministas. Por consiguiente, ya que cada autómata finito determinista acepta sólo un lenguaje, deben existir lenguajes que no son aceptados por una máquina de este tipo.

ver página 8 teorema 0.1

∎

El teorema 1.1 nos dice que existen conjuntos de cadenas que no pueden ser identificados por los autómatas finitos deterministas, y por lo tanto nuestras técnicas de análisis léxico no pueden reconocerlos. Sin embargo, el hecho de que los autómatas finitos deterministas no puedan resolver todos estos problemas de reconocimientos no es más que la punta del iceberg. Más adelante veremos que cualquier problema algorítmico tiene limitaciones similares.

Un lenguaje no regular

Mostremos ahora un ejemplo específico de un lenguaje que no es regular. Este tipo de ejemplos indicará cómo pueden mejorarse las técnicas de análisis léxico de la sección 1.1 para permitirnos una mayor gama de estructuras de cadenas.

Teniendo en cuenta este objetivo, consideremos el problema de analizar expresiones matemáticas donde se usan paréntesis para hacer más claro el orden de ejecución o para alterar la precedencia normal de los operadores, como sucede en las expresiones $(a+b)+c$ y $a+(b+c)$. En estas expresiones debe existir el mismo número de paréntesis izquierdos y derechos. Además, al recorrer una expresión de izquierda a derecha, el número de paréntesis

derechos detectados no debe exceder el número de paréntesis izquierdos encontrados hasta ese punto. Con estas observaciones, nuestra intuición nos dice que el análisis de estas expresiones aritméticas requiere la habilidad para recordar cuántos paréntesis izquierdos se han encontrado sin que exista un paréntesis derecho correspondiente. Puesto que un autómata finito determinista no tiene manera de almacenar este recuento para una referencia posterior, podemos adivinar que el análisis de expresiones aritméticas que contienen paréntesis es una tarea que rebasa las capacidades de los autómatas finitos. Sin embargo, para demostrarlo requerimos ciertos antecedentes.

En primer lugar, presentamos la notación w^n, donde w es una cadena de Σ^* y n es un entero no negativo. Se trata de una forma abreviada para representar una cadena de n copias del patrón w. Así, si $\Sigma = \{x, y\}$, entonces $x^4 = xxxx$, $x^2y^2 = xxyy$, $(xy)^3 = xyxyxy$ y $y^0 = \lambda$ (la cadena vacía).

Ahora necesitamos el siguiente teorema.

TEOREMA 1.2

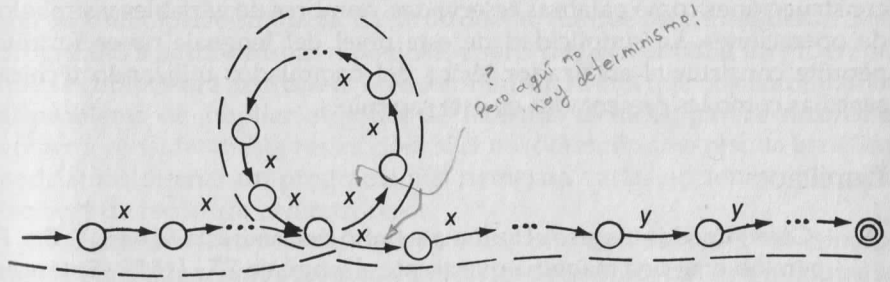
Si un lenguaje regular contiene cadenas de la forma x^ny^n para enteros arbitrariamente grandes, n, entonces debe contener cadenas de la forma x^my^n, donde m y n no son iguales.

DEMOSTRACIÓN

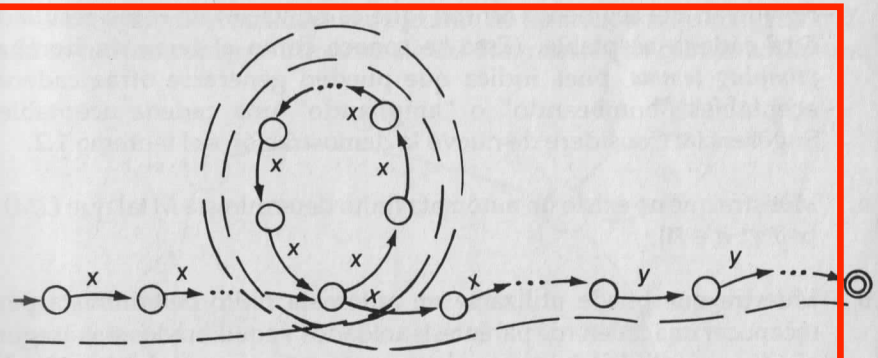
Suponga que M es un autómata finito determinista tal que $L(M)$ contiene x^ny^n para una n arbitrariamente grande. Entonces, debe existir un entero positivo k mayor que el número de estados en M y tal que x^ky^k se encuentre en $L(M)$. Puesto que existen más símbolos en x^k que estados en M, el proceso de aceptación de x^ky^k dará como resultado que se recorra más de una vez alguno de los estados de M antes de llegar a alguna de las y de la cadena. Es decir, durante la lectura de algunas de las x se recorrerá una ruta circular del diagrama de transiciones de la máquina. Si j es el número de x leídas al recorrer esta ruta, entonces la máquina puede aceptar la cadena $x^{k+j}y^k$ recorriendo esta ruta una vez más (véase Fig 1.16). Por lo tanto, existe un entero positivo m (específicamente $k + j$) que no es igual a k, tal que x^my^k se encuentra en $L(M)$.

∎

Una consecuencia inmediata del teorema 1.2 es que el lenguaje $\{x^ny^n: n \in \mathbb{N}\}$ no es regular. Una consecuencia un poco más sutil es que los autómatas finitos deterministas carecen de poder suficiente para analizar expresiones aritméticas que contienen paréntesis. Si un autómata finito determinista aceptara tales expresiones, entonces tendría que aceptar expresiones de la forma $(^n)^n$ para enteros n arbitrariamente grandes. Sin embargo, el teorema 1.2 nos dice que un autómata de este tipo también



a. Ruta que se recorre al aceptar x^ky^k



b. Ruta que se recorre al aceptar $x^{k+j}y^k$

Figura 1.16 Porción de un diagrama de transiciones que acepta tanto x^ky^k como $x^{k+j}y^k$

debe aceptar expresiones en donde el número de paréntesis izquierdos no sea igual al número de paréntesis derechos. Así, tal máquina aceptaría expresiones tanto incorrectas como correctas.

Para concluir, debemos notar que los autómatas finitos deterministas, aunque tengan un poder limitado, no son inútiles. Si usted repasa la sintaxis de su lenguaje de programación de alto nivel preferido, quizá encuentre que un autómata finito no puede reconocer todos los detalles de la estructura de su programa ya que, entre otras cosas, es probable que el lenguaje permita emplear paréntesis en expresiones aritméticas. Por otra parte, también

encontrará que los autómatas finitos pueden procesar la estructura de construcciónes como palabras reservadas, nombres de variables y símbolos de operaciones. La simplicidad de este nivel del lenguaje no es fortuita: permite construir el analizador léxico del compilador utilizando técnicas sencillas como las presentadas en este capítulo.

Ejercicios

1. ¿Cómo puede alterarse el autómata finito determinista $M = (S, \Sigma, \delta, \iota, F)$ para obtener una máquina que acepte el lenguaje $\Sigma^* - L(M)$? (Entonces, el complemento de un lenguaje regular con respecto a Σ^* es regular.)
   Cambiando los estados ○ por ◎ y ◎ por ○
2. Elabore una lista de los autómatas finitos deterministas basados en el alfabeto $\{x, y\}$ que tengan un estado y luego los que tengan dos estados.
   2 de 1 edo, 52, de 2 edos.
3. Muestre que si un autómata finito determinista es capaz de aceptar un número de cadenas infinito, entonces debe aceptar una cadena que consista en la concatenación de tres segmentos tales que cualquier repetición del segmento central (que es no vacío) dé como resultado otra cadena aceptable. (Esto se conoce como el **lema de bombeo** *pumping lemma*, pues indica que pueden generarse otras cadenas aceptables "bombeando" o "ampliando" una cadena aceptable.) Sugerencia: Considere de nuevo la demostración del teorema 1.2.
4. Muestre que no existe un autómata finito determinista M tal que $L(M) = \{x^n y^n z^n : n \in \mathbb{N}\}$.
5. Muestre que puede utilizarse un autómata finito determinista para reconocer una cadena de paréntesis anidados y equilibrados si se asegura que la profundidad del anidamiento no excederá un nivel determinado.

1.4 AUTÓMATAS FINITOS NO DETERMINISTAS

Una vez que nos hemos percatado de las limitantes de los autómatas finitos deterministas, examinar ahora algunas modificaciones que pueden incrementar su potencial. Aquí y en los capítulos subsecuentes, nuestro enfoque será considerar una reducción de las restricciones que hemos impuesto sobre las máquinas. Después de todo, la intuición nos indica que una máquina más flexible debería ser capaz de desempeñar tareas más variadas y, por lo tanto, aceptar un lenguaje que no podrían aceptar las versiones más restringidas. Sin embargo, estamos a punto de ver que nuestra intuición no siempre es correcta.

En lo que llevamos de nuestro estudio, hemos insistido en que los diagramas de transiciones de los autómatas que consideramos deben ser

deterministas. A partir de un estado sólo puede existir un solo arco rotulado con un símbolo determinado. La razón de esto fue que pensamos desarrollar programas a partir de estas máquinas, y sería de poca utilidad un programa que se comportara de manera no determinista. Ahora que nos enfrentamos al problema de ampliar el poder de nuestras técnicas, parece razonable volver a considerar esta restricción; si el no determinismo resulta benéfico, podríamos diseñar un programa que manejara varias opciones aplicando técnicas de recorrido con retrocesos.

Tomando esto en cuenta, consideremos otro tipo de máquinas conceptuales, conocidas como **autómatas finitos no deterministas**. Esta máquina se parece mucho a un autómata finito determinista pues también analiza cadenas construidas a partir de un alfabeto finito y sólo puede tener un número finito de estados, algunos de los cuales son de aceptación y uno es el estado inicial. Sin embargo, a diferencia de los autómatas finitos deterministas, la transición que se ejecuta en una etapa dada de un autómata finito no determinista puede ser incierta. Esto quizás se debera a que es posible aplicar más de una transición, o a que ninguna es aplicable, como sucede con una máquina que no está completamente definida.

Como ejemplo, considere el diagrama de transiciones de la figura 1.17, el cual es intrínsecamente diferente de los diagramas que dibujamos antes.

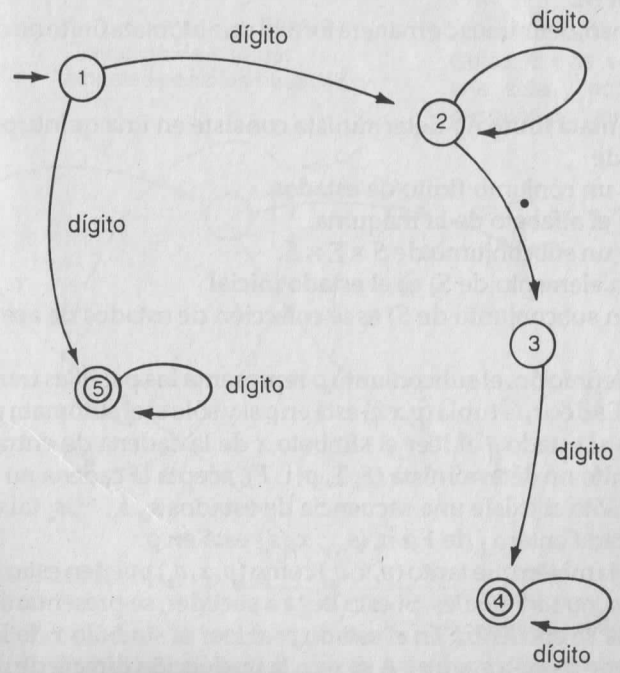


Figura 1.17 Diagrama de transiciones que acepta cadenas que representan enteros o cadenas que representan números reales en notación decimal

Para ser más precisos, aunque una cadena comience con un dígito, esto no nos indica con exactitud cuál es el arco que debe seguirse a partir del estado inicial, pues existen dos posibilidades: una que conduce a la sección del diagrama que describe la estructura de un número en notación decimal y otra que lleva a la descripción de un entero. Por lo tanto, un autómata finito programado con este diagrama sería no determinista.

Recuerde que en el caso de un autómata finito determinista, decimos que una cadena se acepta si al analizarla la máquina queda en un estado de aceptación. Sin embargo, en el caso de un autómata finito no determinista, el intento de análisis de una cadena puede llevar un error simplemente porque se tomaron las decisiones incorrectas en los puntos donde existían varias opciones (si tomamos una decisión incorrecta en el primer paso del análisis de la cadena 352 utilizando la figura 1.17, no llegaríamos a un estado de aceptación, a pesar de que la cadena en cuestión es compatible con otras rutas del diagrama). Por esto, decimos que un autómata finito no determinista acepta una cadena si es *posible* que su análisis deje a la máquina en un estado de aceptación.

Como sucede en el caso de los autómatas finitos deterministas, el conjunto de todas las cadenas aceptadas por un autómata finito no determinista M es un lenguaje que representamos con $L(M)$, y nos referimos a él como el lenguaje aceptado por M.

En resumen, definimos de manera formal un autómata finito no determinista como sigue:

Un autómata finito no determinista consiste en una quíntupla $(S, \Sigma, \rho, \iota, F)$, donde

a. S es un conjunto finito de estados.
b. Σ es el alfabeto de la máquina.
c. ρ es un subconjunto de $S \times \Sigma \times S$.
d. ι (un elemento de S) es el estado inicial
e. F (un subconjunto de S) es la colección de estados de aceptación.

En esta definición, el subconjunto ρ representa las posibles transiciones de la máquina. Es decir, la tupla (p,x,q) está en ρ si y sólo si el autómata puede pasar del estado p al estado q al leer el símbolo x de la cadena de entrada. Así, un autómata finito no determinista $(S, \Sigma, \rho, \iota, F)$, acepta la cadena no vacía $x_1x_2 \cdots x_n \in \Sigma^*$ si y sólo si existe una secuencia de estados $s_0, s_1, \cdots, s_n$ tal que $s_0 = \iota$, $s_n \in F$ y para cada entero j de 1 a n, (s_{j-1}, x_j, s_j) está en ρ.

Observe también que tanto (p, x, q_1) como (p, x, q_2) pueden estar en ρ incluso cuando q_1 y q_2 no son iguales. Si esto llega a suceder, se presentará una opción si la máquina se encuentra en el estado p: al leer el símbolo x de la cadena de entrada: puede pasar a q_1 o a q_2. A su vez, la traducción directa de un autómata finito no determinista a formato de programa puede producir un analizador léxico no determinista. Para corregir esta situación, tendríamos que modificar el analizador a fin de que no rechace una cadena de entrada hasta que haya

probado todas las rutas posibles del diagrama de transiciones y encontrado que todas fallan. Este sistema de retrocesos haría más complejo al analizador y sólo valdría la pena si obtuviéramos más poder de procesamiento. Así, el teorema 1.3 explica por qué rehusamos utilizar un autómata finito no determinista como base para un analizador léxico.

Para motivar este teorema, imaginemos la estrategia que seguiríamos si utilizáramos un diagrama de transiciones no determinista para analizar una cadena. En esta situación, podríamos evitar ajustarnos a una sola posibilidad y tratar de seguir todas las opciones en paralelo; es decir, atravesaríamos al mismo tiempo más de una ruta a través del diagrama de transiciones. Al obtener cada símbolo de la entrada, cada una de estas rutas avanzaría de manera independiente. En este contexto, aceptaríamos una cadena si alguna de las rutas terminara en un estado de aceptación al llegar al final de la entrada. Las otras rutas representarían únicamente opciones que, de haberlas seguido, habrían llevado al fracaso y a la necesidad de retroceder para seguir otras opciones.

Cuando seguimos este enfoque de procesamiento en paralelo, nuestro estado en un momento dado ya no se hallará determinado por un solo

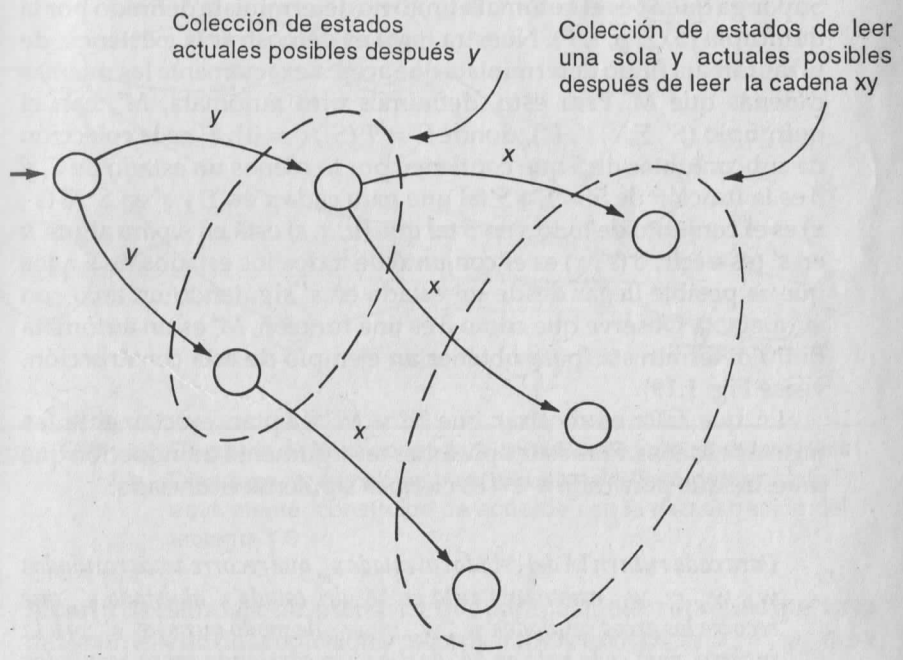


Figura 1.18 Análisis de una cadena con un autómata finito no determinista, siguiendo las opciones en paralelo

estado del diagrama de transiciones, sino por la colección de los estados actuales de todas las rutas posibles que se consideran (véase Fig. 1.18). Además, si K es esta colección de estados actuales, entonces, al leer un símbolo x del flujo de entrada obtendríamos un nuevo estado actual, representado por la colección de los estados a los que es posible llegar desde un estado K siguiendo un arco con etiqueta x.

Observe que este enfoque en paralelo, que pasa de colecciones de estados a colecciones de estados, elimina la necesidad de tomar decisiones cuando se presentan varias opciones: basta con seguir en paralelo todas las opciones. Así, parecería que con este enfoque se podría superar el no determinismo en un autómata finito. Esta observación es la que da origen a nuestra demostración del siguiente teorema.

TEOREMA 1.3

Para cada autómata finito no determinista, existe un autómata finito determinista que acepta exactamente el mismo lenguaje.

DEMOSTRACIÓN

Suponga que M es el autómata finito no determinista definido por la quíntupla $(S, \Sigma, \rho, \iota, F)$. Nuestra tarea es demostrar la existencia de un autómata finito determinista que acepta exactamente las mismas cadenas que M. Para esto, definimos otro autómata, M', con el quíntuplo $(S', \Sigma, \delta, \iota', F')$, donde $S' = \mathcal{P}(S)$, $\iota' = \{\iota\}$, F' es la colección de subconjuntos de S que contienen por lo menos un estado de F, y δ es la función de $S' \times \Sigma$ a S' tal que para cada x en Σ y s' en S', $\delta(s', x)$ es el conjunto de todo s en S tal que (u, x, s) está en ρ para algun u en s' (es decir, $\delta(s', x)$ es el conjunto de todos los estados de S a los que es posible llegar desde un estado en s' siguiendo un arco con etiqueta x). Observe que como δ es una función, M' es un autómata finito determinista (para obtener un ejemplo de esta construcción, véase Fig. 1.19).

Lo que falta es mostrar que M y M' aceptan exactamente las mismas cadenas. Para esto, aplicamos un argumento de inducción que muestre que para cada $n \in \mathbb{N}$ es cierto el siguiente enunciado.

> *Para cada ruta en* M *del estado* ι *al estado* s_n, *que recorre arcos rotulados* $w_1, w_2, \cdots, w_n$, *existe una ruta en* M′ *del estado* ι' *al estado* s'_n *que recorre los arcos rotulados* $w_1, w_2, \cdots, w_n$, *de modo que* $s_n \in s'_n$; *y a la inversa, para cada ruta en* M′ *de* ι' *a* s'_n *recorriendo arcos rotulados* $w_1, w_2, \cdots, w_n$ *y cada* $s_n \in s'_n$, *existe una ruta en* M *de* ι *a* s_n *que recorre arcos rotulados* $w_1, w_2, \cdots, w_n$.

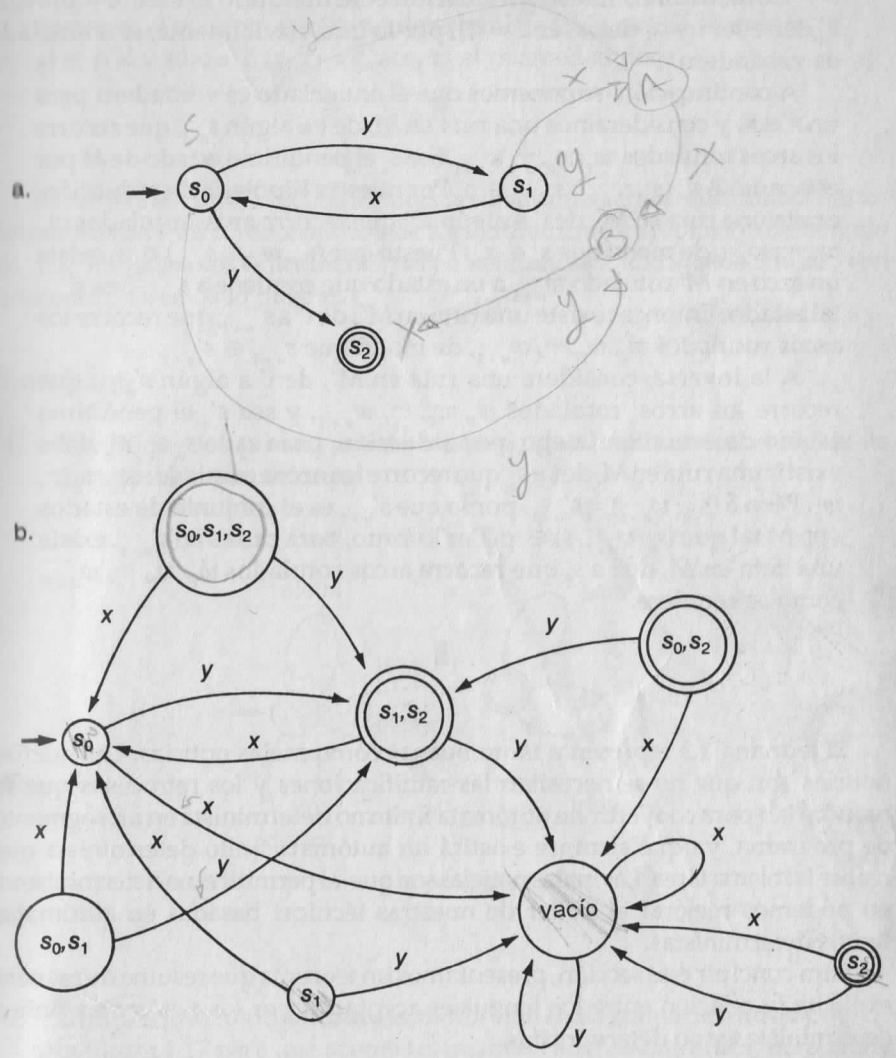


Figura 1.19 a. Diagrama de transiciones para un autómata finito no determinista
b. Diagrama de transición para el autómata finito determinista equivalente, construido de acuerdo con la demostración del teorema 1.3

A partir de esto, debe desprenderse que para cualquier ruta en M que vaya de ι a un estado de aceptación y recorra arcos rotulados $w_1, w_2, \cdots, w_n$, debe existir una ruta en M', de ι' a un estado de aceptación, que siga los arcos rotulados $w_1, w_2, \cdots, w_n$ y viceversa. Por lo tanto, M y M' deben aceptar el mismo lenguaje.

Comenzamos nuestra inducción considerando el caso $n = 0$. Aquí s_n debe ser ι y s'_n debe ser $\iota' = \{\iota\}$, por lo que, trivialmente, el enunciado es verdadero.

A continuación, suponemos que el enunciado es verdadero para un $n \in \mathbb{N}$ y consideramos una ruta en M, de ι a algún s_{n+1}, que recorre los arcos rotulados $w_1, w_2, \cdots, w_{n+1}$. Sea s_n el penúltimo estado de M por esta ruta. Así, $(s_n, w_{n+1}, s_{n+1}) \in \rho$. Por nuestra hipótesis de inducción, existe una ruta en M', de ι' a algún s'_n, que recorre arcos rotulados $w_1, w_2, \cdots, w_{n+1}$ de modo que $s_n \in s'_n$. Puesto que $(s_n, w_{n+1}, s_{n+1}) \in \rho$, existe un arco en M' rotulado w_{n+1} a un estado que contiene a s_{n+1}. Sea s'_{n+1} tal estado. Entonces existe una ruta en M', de ι' a s'_{n+1}, que recorre los arcos rotulados $w_1, w_2, \cdots, w_{n+1}$, de modo que $s_{n+1} \in s'_{n+1}$.

A la inversa, considere una ruta en M', de i' a algún s'_{n+1}, que recorre los arcos rotulados $w_1, w_2, \cdots, w_{n+1}$, y sea s'_n el penúltimo estado de esta ruta. Luego, por inducción, para cada $s_n \in s'_n$ debe existir una ruta en M, de ι a s_n, que recorre los arcos rotulados $w_1, w_2, \cdots, w_n$. Pero $\delta(s'_n, w_{n+1}) = s'_{n+1}$, por lo que s'_{n+1} es el conjunto de estados s en M tal que $(s_n, w_{n+1}, s) \in \rho$. Por lo tanto, para cada s en s'_{n+1}, existe una ruta en M, de i a s, que recorre arcos rotulados $w_1, w_2, \cdots, w_{n+1}$, como se requiere.

∎

El teorema 1.3 representa tanto buenas como malas noticias. Las buenas noticias son que no se necesitan las ramificaciones y los retrocesos que se requerirían para convertir un autómata finito no determinista en un segmento de programa, ya que siempre existirá un autómata finito determinista que realice la misma tarea. Las malas noticias son que, al permitir el no determinismo, no podemos mejorar el poder de nuestras técnicas basadas en autómatas finitos deterministas.

Para concluir esta sección, presentamos un teorema que resume de manera explícita la relación entre los lenguajes aceptados por los autómatas finitos deterministas y no deterministas.

TEOREMA 1.4

Para cualquier alfabeto Σ, $\{L(M)$: M es un autómata finito determinista con alfabeto $\Sigma\} = \{L(M)$: M es un autómata finito no determinista con alfabeto $\Sigma\}$.

DEMOSTRACION

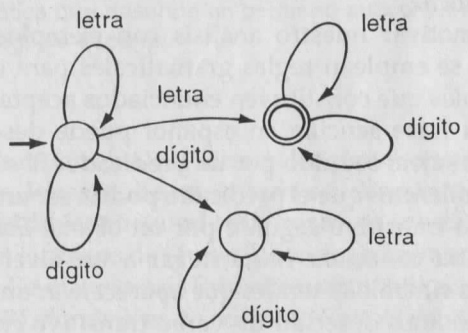
El hecho de que los lenguajes aceptados por los autómatas finitos no deterministas sean también aceptados por los autómatas finitos deterministas no es más que una repetición del teorema 1.3. A la inversa, si $M = (S, \Sigma, \delta, \iota, F)$ es un autómata finito determinista, entonces el autómata finito no determinista $(S, \Sigma, \rho, \iota, F)$, donde $(p, x, q) \in \rho$ si y sólo si $\delta(p, x) = q$, acepta el mismo lenguaje.

∎

Por el teorema 1.4, muchos autores no distinguen entre autómatas finitos deterministas y los no deterministas cuando analizan la aceptación de lenguajes. Por el contrario, únicamente hacen referencia a autómatas finitos; con frecuencia haremos lo mismo.

Ejercicios

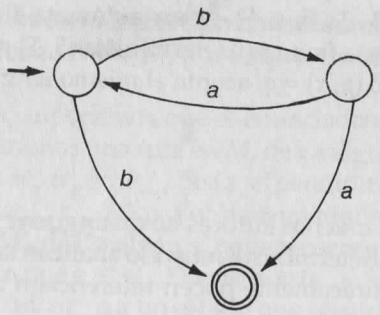
1. Identifique los puntos donde existe no determinismo en el siguiente diagrama de transiciones.



2. Utilizando métodos intuitivos, modifique el diagrama de transiciones de la figura 1.17 para que acepte las mismas cadenas que antes, pero ahora sin ramificaciones no deterministas.

3. Muestre que cada autómata finito determinista $(S, \Sigma, \delta, \iota, F)$ es un autómata finito no determinista, demostrando que la función de transición δ puede representarse como un subconjunto de $S \times \Sigma \times S$.

4. Empleando las técnicas presentadas en la demostración del teorema 1.3, dibuje un diagrama de transiciones para un autómata finito determinista que acepte las mismas cadenas que el autómata finito no determinista representado en el siguiente diagrama de transiciones.



1.5 GRAMÁTICAS REGULARES

Nuestro objetivo para lo que resta del capítulo es aprender más acerca de los lenguajes regulares. En esta sección analizaremos su estructura gramatical y en la siguiente su composición algebraica. Comenzaremos con el concepto general de gramática.

Resulta útil motivar nuestro análisis con ejemplos de los lenguajes naturales, donde se emplean reglas gramaticales para distinguir aquellas cadenas de símbolos que constituyen enunciados aceptables de las que no. Por ejemplo, una frase sencilla en español puede describirse como una estructura con un sujeto seguido por un predicado. El sujeto podría ser un nombre sencillo, mientras que el predicado podría ser un verbo intransitivo o quizás un verbo transitivo seguido por un objeto. Esta descomposición descendente podría continuar hasta llegar a un nivel de detalle donde describiríamos los símbolos literales que aparecen en una frase en español. Por ejemplo, podríamos describir un verbo transitivo como "golpear" o la palabra "quiere". En la figura 1.20 se muestra esta descomposición.

Hay que señalar varios puntos con respecto a la figura 1.20. En primer lugar, usted observará que algunos de los términos se encuentran encerrados entre corchetes agudos, mientras que otros no lo están. Los términos que no aparecen entre corchetes se llaman **terminales**, o símbolos que pueden aparecer en una frase. Los términos que se hallan encerrados entre corchetes se denominan **no terminales**. Uno de estos no terminales (en la Fig. 1.20 es <frase>) se considera como **símbolo de inicio**. Cada una de las líneas de la figura 1.20 se llama **regla de reescritura** consiste en una parte izquierda y una derecha, conectadas por una flecha. La parte derecha de estas reglas representa una descripción más detallada de la parte izquierda. Por ejemplo, la primera regla de reescritura de la figura 1.20 indica que una <frase> tiene la estructura de un <sujeto> seguido por un <predicado> seguido por un <punto>. Más adelante en la misma figura, aparece una descripción más detallada de estos no terminales. Así, la figura 1.20 describe de manera jerárquica la estructura de una frase, comenzando con el símbolo de inicio <frase>.

<frase> → <sujeto> <predicado> <punto>

<sujeto> → <sustantivo>

<sustantivo> → María

<sustantivo> → Juan

<predicado> → <verbo intransitivo>

<predicado> → <verbo transitivo> <objeto>

<verbo intransitivo> → patinar

<verbo transitivo> → golpear

<verbo transitivo> → quiere

<objeto> → a <sustantivo>

<punto> → .

Figura 1.20 Gramática que describe un pequeño subconjunto del lenguaje español

Esta colección de no terminales y terminales, junto con un símbolo de inicio y un conjunto finito de reglas de reescritura, se denomina gramática o, más precisamente, **gramática estructurada por frases**, ya se está basa en la composición de cadenas en términos de "frases", donde cada frase está representada por un no terminal. De manera más formal, una gramática se define como una cuádrupla (V, T, S, R), donde V es un conjunto finito de no terminales, T es un conjunto finito de terminales, S (un elemento de N) es el símbolo inicial y R es un conjunto finito de reglas de reescritura. En general, los lados derecho e izquierdo de las reglas de reescritura de una gramática pueden ser cualquier combinación de terminales y no terminales, siempre y cuando el lado izquierdo contenga por lo menos un no terminal. Además, el lado derecho de algunas reglas de reescritura puede consistir en una cadena vacía (en este caso, la regla significa que el patrón representado por el lado derecho de la regla puede ser la cadena vacía). Esta regla se llama **regla λ**.

A fin de evitar confusiones, se emplean corchetes para distinguir los terminales de los no terminales. Por ejemplo, en una gramática que describiera la estructura de las frases en español, el término *frase* podría aparecer como un no terminal, representando el símbolo de inicio, y como un terminal, representando una palabra en una frase. En este caso, los corchetes nos permitirían diferenciar usos duales del término. En nuestro estudio, sin embargo, distinguiremos entre terminales y no terminales con una notación menos engorrosa. Salvo que se especifique lo contrario, representaremos a

los no terminales con letras mayúsculas y a los terminales con letras minúsculas. De esta manera, una regla de la forma $S \rightarrow xN$ significará el no terminal S se puede refinar como el terminal x seguido por el no terminal N.

Se dice que una gramática genera una cadena de terminales si, al comenzar con el símbolo de inicio, se puede producir esa cadena sustituyendo sucesivamente los patrones que se encuentran en el lado izquierdo de las reglas de reescritura de la gramática con las expresiones correspondientes de la derecha, hasta que sólo queden terminales. La secuencia de pasos de este proceso se le conoce como **derivación** de la cadena. Por ejemplo, la cadena (la frase).

María quiere a Juan.

puede generarse a partir de la gramática de la figura 1.20 usando la derivación de la figura 1.21. Como un ejemplo, más la figura 1.22 muestra otra gramática, con el símbolo inicial S, y una derivación que muestra cómo puede generarse la cadena xyz.

Si los terminales de una gramática G son símbolos del alfabeto Σ, decimos que G es una gramática del alfabeto Σ. En este caso, las cadenas generadas por G son en realidad cadenas de Σ^*. Por lo tanto, una gramática G de un alfabeto Σ especifica un lenguaje de Σ que consiste en las cadenas generadas por G; a este lenguaje se le representa con $L(G)$. Si lo desea, puede confirmar qué el lenguaje generado por la gramática de la figura 1.22 es $\{x^m y^m z^m : m, n \in \mathbb{N}\}$.

<frase>
⇓
<sujeto> <predicado> <punto>
⇓
<sustantivo> <predicado> <punto>
⇓
María <predicado> <punto>
⇓
María <verbo transitivo> <objeto> <punto>
⇓
María quiere <objeto> <punto>
⇓
María quiere a <sustantivo> <punto>
⇓
María quiere a Juan <punto>
⇓
María quiere a Juan.

Figura 1.21 Derivación de la frase "María quiere a Juan."

$$S \rightarrow XSZ$$
$$S \rightarrow Y$$
$$Y \rightarrow yY$$
$$Y \rightarrow \lambda$$
$$X \rightarrow x$$
$$Z \rightarrow z$$

$$S \Rightarrow XSZ \Rightarrow xSZ \Rightarrow xyYZ \Rightarrow xyZ \Rightarrow xyz$$

Figura 1.22 Gramática y una derivación asociada de la cadena xyz

$S \rightarrow 0T$	$T \rightarrow 0T$	$U \rightarrow 0V$	$V \rightarrow 0V$
$S \rightarrow 1T$	$T \rightarrow 1T$	$U \rightarrow 1V$	$V \rightarrow 1V$
$S \rightarrow 2T$	$T \rightarrow 2T$	$U \rightarrow 2V$	$V \rightarrow 2V$
$S \rightarrow 3T$	$T \rightarrow 3T$	$U \rightarrow 3V$	$V \rightarrow 3V$
$S \rightarrow 4T$	$T \rightarrow 4T$	$U \rightarrow 4V$	$V \rightarrow 4V$
$S \rightarrow 5T$	$T \rightarrow 5T$	$U \rightarrow 5V$	$V \rightarrow 5V$
$S \rightarrow 6T$	$T \rightarrow 6T$	$U \rightarrow 6V$	$V \rightarrow 6V$
$S \rightarrow 7T$	$T \rightarrow 7T$	$U \rightarrow 7V$	$V \rightarrow 7V$
$S \rightarrow 8T$	$T \rightarrow 8T$	$U \rightarrow 8V$	$V \rightarrow 8V$
$S \rightarrow 9T$	$T \rightarrow 9T$	$U \rightarrow 9V$	$V \rightarrow 9V$
	$T \rightarrow .U$		$V \rightarrow \lambda$

Figura 1.23 Gramática regular que genera cadenas que representan números racionales en notación decimal

Ahora definimos una **gramática regular** como aquella gramática cuyas reglas de reescritura se adhieren a las siguientes restricciones. El lado izquierdo de cualquier regla de reescritura de una gramática regular debe consistir en un solo no terminal, mientras que el lado derecho debe ser un terminal seguido por un no terminal, o un solo terminal, o la cadena vacía. Así, las reglas de reescritura de la forma

$$Z \rightarrow yX$$
$$Z \rightarrow x$$
$$W \rightarrow 1$$

$$S \to xX$$
$$X \to yY$$
$$Y \to xX$$
$$Y \to \lambda$$

Figura 1.24 Gramática regular que genera cadenas consistentes en una o más copias del patrón *xy*

$$S \to xX$$
$$S \to yY$$
$$X \to yY$$
$$X \to \lambda$$
$$Y \to xX$$
$$Y \to \lambda$$

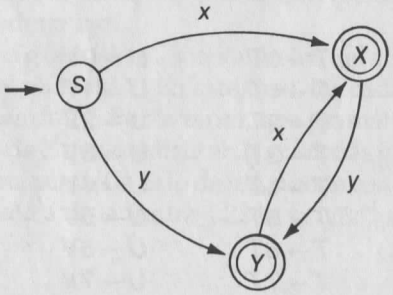


Figura 1.25 Gramática regular *G* y diagrama de transiciones para un autómata finito *M* tal que *L(G) = L(M)*

estarían permitidas en una gramática regular, mientras que las reglas de la forma

$$yW \to X$$
$$X \to xZy$$
$$YX \to WvZ$$

no lo estarían.

La figura 1.23 presenta una gramática regular que genera cadenas que representan números racionales en notación decimal, y la figura 1.24 presenta una gramática regular que genera cadenas consistentes en una o más copias del patrón *xy*. En los dos ejemplos, el símbolo inicial es *S*. Usted observará que ninguna de estas gramáticas utiliza una regla cuyo lado derecho consista en un solo terminal. Aunque lo permite la definición de una gramática regular, en realidad no se requieren reglas de este tipo. De hecho, en una gramática regular cualquier regla de la forma

$$N \to x$$

podría reemplazarse con el par de reglas

$$N \to xX$$
$$X \to \lambda$$

donde *X* es un no terminal que no aparece en ningún otro lugar de la gramática, sin alterar el conjunto de cadenas que podría generar la gramática.

La importancia de las gramáticas regulares reside en que los lenguajes generados por ellas son exactamente aquellos que reconocen los autómatas finitos, como lo muestra el teorema 1.5

TEOREMA 1.5
Para cada alfabeto Σ $\{L(G)$: *G* es una gramática regular de $\Sigma\} = \{L(M)$: *M* es un autómata finito de $\Sigma\}$.

DEMOSTRACIÓN
Si *G* es una gramática regular de Σ, podemos convertirla en una gramática regular *G'* que genera el mismo lenguaje pero que no contiene reglas de reescritura cuyo lado derecho consiste en un solo terminal (véanse los comentarios que preceden al Teorema 1.5). Entonces podemos definir *M* como el autómata finito no determinista $(S, \Sigma, \rho, \iota, F)$, donde *S* es la colección de no terminales de *G'*, ι es el símbolo inicial de *G'*, *F* es la colección de no terminales de *G'* que aparecen en el lado izquierdo de alguna regla λ, y ρ consiste en la tripleta (P, x, Q) para el cual *G'* contiene una regla de reescritura de la forma $P \to xQ$. A la inversa, si *M* es el autómata finito no determinista $(S, \Sigma, \rho, \iota, F)$, entonces podemos definir *G'* como la gramática regular de Σ para la cual los no terminales son los estados de *S*, el símbolo inicial es ι y las reglas de reescritura son de la forma $P \to xQ$ si (P, x, Q) está en ρ y $Q \to \lambda$ si Q está en *F*.

En ambos casos, $L(M) = L(G')$ ya que la derivación de una cadena de la gramática G' corresponde directamente a una ruta en el diagrama de transiciones de *M* que conduce del estado inicial a un estado de aceptación y viceversa. Como ejemplo, véase la figura 1.25.

∎

La importancia del enfoque gramatical de la especificación de lenguajes se hace aparente en los dos capítulos siguientes, donde veremos que el empleo de gramáticas lleva a un esquema de clasificación para diversos tipos de lenguajes. Por el momento, la composición de gramáticas se puede comparar con la de los diagramas de sintaxis que se utilizan con frecuencia en la actualidad al describir la sintaxis de lenguajes de programación como Pascal, Modula-2 y Ada. Encontrará que estos diagramas de sintaxis son, en esencia, otra forma de representación de las reglas de reescritura de una gramática (véase Fig. 1.26).

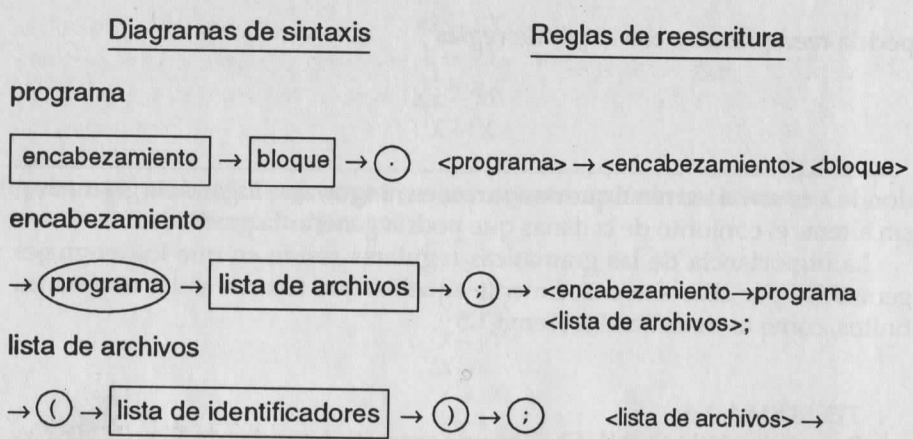


Figura 1.26 Algunos diagramas de sintaxis que describen partes del lenguaje de programación Pascal, comparados con sus formas gramaticales equivalentes

Ejercicios

1. Elabore una lista con todas las frases generadas por la gramática de la figura 1.20.

2. Muestre que es posible modificar la gramática que se presenta a continua ción (con símbolo inicial S) para formar una gramática regular, sin cambiar el lenguaje que genera. ¿Cuál de los ejercicios de la sección anterior es en esencia el mismo problema, aunque en un contexto diferente?

$$S \to yX$$
$$X \to xxX$$
$$X \to yY$$
$$Y \to \lambda$$

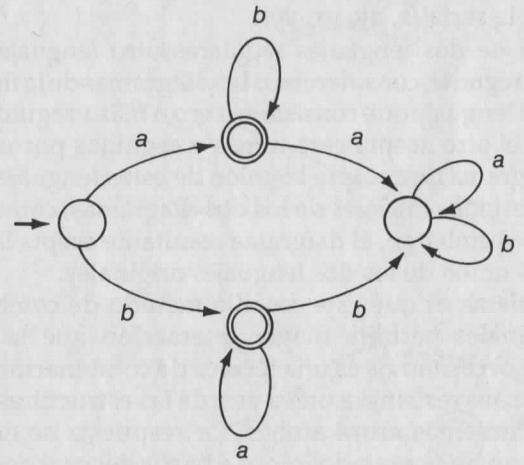
3. Convierta la siguiente gramática regular en otra gramática regular que no contenga reglas de reescritura cuyos lados derechos consistan en un solo terminal, pero que a la vez genere el mismo lenguaje. Describa con sus propias palabras el lenguaje generado.

$$S \to xS$$
$$S \to y$$
$$S \to z$$

4. Dibuje un diagrama de transiciones para un autómata finito que acepte el lenguaje generado por la gramática regular (con símbolo inicial S) que se presenta a continuación. Describa el lenguaje con sus propias palabras.

$$S \to \lambda$$
$$S \to xX$$
$$S \to yY$$
$$Y \to yY$$
$$Y \to \lambda$$
$$X \to xX$$
$$X \to \lambda$$

5. Presente una gramática regular que genere el lenguaje aceptado por el autómata finito cuyo diagrama de transiciones se presenta a continuación.



1.6 EXPRESIONES REGULARES

Hemos definido los lenguajes regulares como aquellos que son reconocidos por autómatas finitos, y hemos mostrado que son también aquellos lenguajes generados por gramáticas regulares. En esta sección desarrollaremos otra caracterización de los lenguajes regulares, la cual proporciona mayores detalles acerca de la composición de dichos lenguajes.

Aquí nuestro enfoque será mostrar cómo se pueden construir los lenguajes regulares a partir de pequeños bloques de construcción. Comenzamos por considerar los lenguajes más sencillos que pueden formarse con el

alfabeto Σ. Es decir, nos interesan los subconjuntos más simples de Σ^*. Quizás usted conjeturará que serían los subconjuntos que consisten en cadenas simples de longitud uno, y de hecho éstos son los lenguajes que utilizaremos como algunos de nuestros bloques de construcción. Después de todo, si $\Sigma = \{x, y\}$, entonces los lenguajes $\{x\}$ y $\{y\}$ parecerían ser los bloques de construcción naturales para la construcción de otros lenguajes de Σ.

Existen, sin embargo dos lenguajes que no podríamos construir utilizando estos bloques. Uno de ellos es el lenguaje $\{\lambda\}$; el otro es $\varnothing$. Por razones que serán obvias en unos momentos, no empleamos $\{\lambda\}$ como uno de nuestros bloques de construcción. En vez de esto, nuestra colección de bloques contiene el lenguaje vacío y todos los lenguajes de cadenas simples con longitud uno.

Una vez establecida esta base, nos centramos en las maneras como se pueden construir lenguajes más complicados a partir de los más básicos. Quizás la técnica más directa sea combinar dos lenguajes, utilizando la operación de unión de la teoría de conjuntos. Esta operación entre dos lenguajes se representan con el símbolo tradicional para la unión de conjuntos, $\cup$. Así, si L_1 fuese el lenguaje $\{x, xy\}$ y L_2 fuera $\{yx, yy\}$, entonces el lenguaje $L_1 \cup L_2$ sería $\{x, xy, yx, yy\}$.

¿Es la unión de dos lenguajes regulares otro lenguaje regular? Para responder esta pregunta, consideremos los diagramas de la figura 1.27a. Uno de ellos acepta el lenguaje que consiste en cero o más x seguidas por una sola y, mientras que el otro acepta cero o más y seguidas por una sola x. Para construir un diagrama que acepte la unión de estos lenguajes, uno tendería a identificar los estados iniciales de los dos diagramas, como se muestra en la figura 1.27b. Sin embargo, el diagrama resultante acepta la cadena $xyxyx$, que no está en la unión de los dos lenguajes originales.

Aquí el problema es que este sencillo método de combinación de los diagramas originales permite mayor interacción que la que podemos permitir. Lo que necesitamos es una técnica de combinación que produzca un diagrama que nos restrinja a una u otra de las estructuras originales, sin permitir que alternemos entre ambas. La respuesta se encuentra en la introducción de un nuevo estado inicial a partir del cual podamos entrar a uno de los diagramas originales sin poder regresar, como se muestra en la figura 1.27c. La construcción general es la siguiente: dibuje un nuevo estado inicial; declárelo como un estado de aceptación si y sólo si uno de los estados iniciales era de aceptación; de los estados iniciales originales dibuje un arco, con la misma etiqueta, a partir del nuevo estado inicial; para cada estado que sea el punto de destino de un arco de alguno elimine la característica de inicio de los estados iniciales originales.

Este proceso, cuando se aplica a cualquier par de diagramas de transiciones para autómatas finitos, producirá otro diagrama de transiciones para el cual cualquier cálculo es una copia del cálculo que habría efectuado una de las máquinas originales, y viceversa. Por lo tanto, *la unión de dos lenguajes regulares cualesquiera también es regular.*

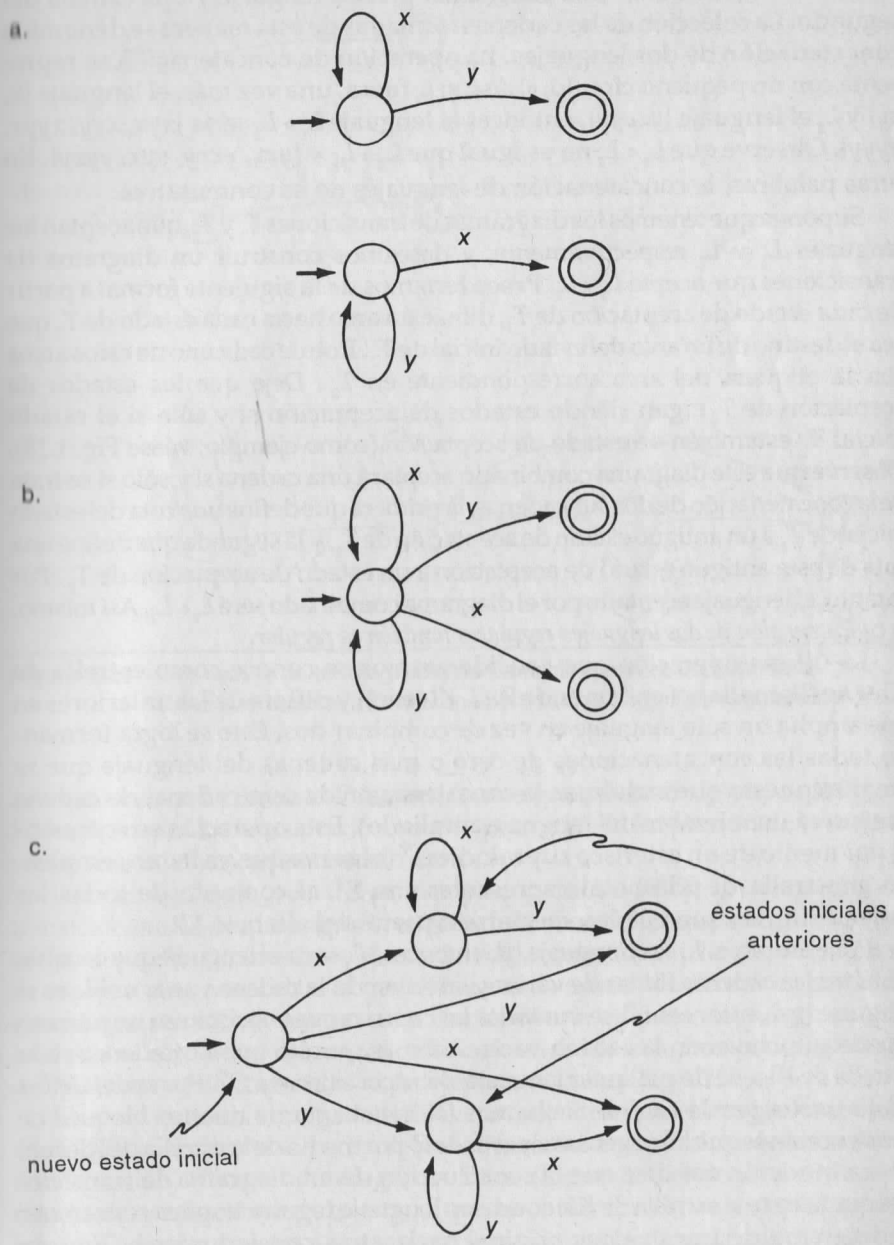


Figura 1.27 Formación de un diagrama de transiciones para un autómata finito que acepta la unión de dos lenguajes regulares

Otra técnica para combinar dos lenguajes es recopilar todas las cadenas formadas al concatenar una cadena del primer lenguaje y una cadena del segundo. La colección de las cadenas formadas de esta manera se denomina **concatenación** de dos lenguajes. La operación de concatenación se representa con un pequeño círculo, $\circ$. Así, si L_1 fuera, una vez más, el lenguaje $\{x, xy\}$ y L_2 el lenguaje $\{yx, yy\}$, entonces el lenguaje $L_1 \circ L_2$ sería $\{xyx, xyy, xyyx, xyyy\}$. Observe que $L_1 \circ L_2$ no es igual que $L_2 \circ L_1 = \{yxx, yxxy, yyx, yyxy\}$. En otras palabras, la concatenación de lenguajes no es conmutativa.

Suponga que tenemos los diagramas de transiciones T_1 y T_2, que aceptan los lenguajes L_1 y L_2 respectivamente, y deseamos construir un diagrama de transiciones que acepte $L_1 \circ L_2$. Procederíamos de la siguiente forma: a partir de cada estado de aceptación de T_1, dibuje un arco hacia cada estado de T_2 que sea el destino de un arco del estado inicial de T_2. Rotule cada uno de estos arcos con la etiqueta del arco correspondiente en T_2. Deje que los estados de aceptación de T_1 sigan siendo estados de aceptación si y sólo si el estado inicial T_2 estambién un estado de aceptación (como ejemplo, véase Fig. 1.28). Observe que este diagrama combinado aceptará una cadena si y sólo si se trata de la concatenación de dos subcadenas, la primera que define una ruta del estado inicial de T_1 a un antiguo estado de aceptación de T_1, y la segunda que define una ruta de este antiguo estado de aceptación a un estado de aceptación de T_2. Por lo tanto, el lenguaje aceptado por el diagrama combinado será $L_1 \circ L_2$. Así mismo, *la concatenación de dos lenguajes regulares también es regular.*

La última operación que consideraremos se conoce como **estrella de Kleene** (llamada así en honor de S. C. Kleene), y difiere de las anteriores en que amplía un solo lenguaje en vez de combinar dos. Esto se logra formando todas las concatenaciones de cero o más cadenas del lenguaje que se amplía (puesto que incluimos la concatenación de cero cadenas, la cadena vacía será un miembro del lenguaje ampliado). Esta operación se representa por medio de un asterisco supraíndice, * (observe que ya hemos empleado la estrella de Kleene al representar con Σ^* el conjunto de todas las cadenas finitas que pueden formarse a partir del alfabeto Σ).

Por ejemplo, si L_1 es el lenguaje $\{y\}$, entonces L^*_1 sería el lenguaje que consiste en todas las cadenas finitas de varias y, incluyendo la cadena vacía, o si L_2 es el lenguaje $\{yy\}$, entonces L^*_2 serían todas las cadenas que consisten en un número par de y, incluyendo la cadena vacía. Además, puesto que λ pertenece a la estrella de Kleene de cualquier lenguaje, debe pertenecer a $\varnothing^*$. Por ende, $\varnothing^{**} = \{\delta\}$ (la razón por la cual no incluimos $\{\delta\}$ como uno de nuestro bloques de construcción es que $\{\delta\}$ se genera a partir de $\varnothing$ por medio de la estrella de Kleene)

La intuición nos dice que la construcción de un diagrama de transiciones que acepte la estrella de Kleene de un lenguaje regular implica concatenar el diagrama de transiciones original hacia atrás consigo mismo. En este caso, nuestra intuición es correcta salvo por un pequeño detalle: el nuevo diagrama de transiciones debe aceptar la cadena vacía. Para lograr esto, nuestra intuición nos podría indicar que basta con designar el estado inicial original como un nuevo estado de aceptación, pero el problema no

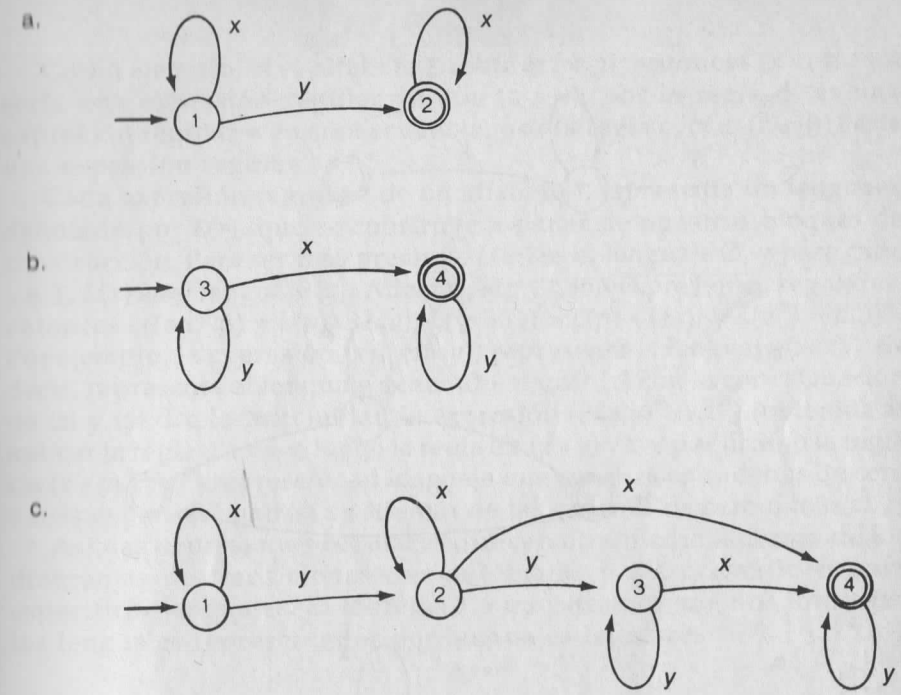


Figura 1.28 Formación de un diagrama de transiciones para un autómata finito que acepta la concatenación de dos lenguajes regulares

se soluciona tan fácilmente. Por ejemplo, designar el estado inicial de la figura 1.29a como estado de aceptación no sólo permitiría la aceptación de la cadena vacía, sino además cadenas como $xxxx$.

Por consiguiente, nuestro primer paso para modificar un diagrama de transiciones a fin de que acepte la estrella de Kleene del lenguaje originalmente aceptado es dibujar un nuevo estado inicial y conectarlo al diagrama original, de manera muy parecida a lo que se hace en la operación de unión: por cada estado que sea dibujamos un arco con la misma etiqueta del nuevo estado inicial hacia el destino de un arco del estado inicial del diagrama original luego designamos a éste como estado de aceptación (véase Fig.1.29b).

Una vez que se ha agregado el nuevo estado inicial, nuestra siguiente tarea es modificar el diagrama para que podamos establecer un ciclo de los estados de aceptación al "inicio" del diagrama. Esto se logra añadiendo un arco de cada estado de aceptación a cada estado que es el destino de un arco del estado inicial. Cada uno de estos nuevos arcos se rotula con la etiqueta que corresponda al arco del estado inicial. El resultado es un diagrama de transiciones que acepta una cadena no vacía si y sólo si esa cadena es la concatenación de cadenas aceptadas por el diagrama original (como ejemplo, véase Fig. 1.29c). Concluimos que *la estrella de Kleene de cualquier lenguaje regular es regular.*

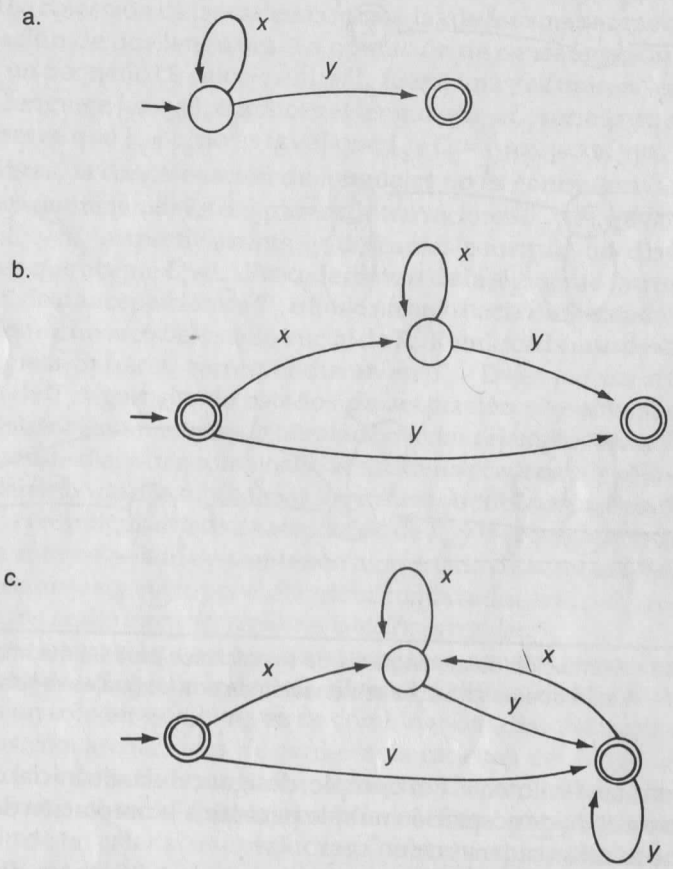


Figura 1.29 Formación de un diagrama de transiciones para un autómata finito que acepta la estrella de Kleene de un lenguaje regular

Dado un alfabeto particular y usando las operaciones de unión, concatenación y estrella de Kleene, podemos construir muchos lenguajes a partir de nuestros bloques de construcción. Para presentar este proceso con mayor precisión, establecemos la siguiente definición.

Una **expresión regular** (para un alfabeto Σ) se define como sigue:

a. $\varnothing$ es una expresión regular.
b. Cada miembro de Σ es una expresión regular.
c. Si p y q son expresiones regulares, entonces también lo es $(p \cup q)$.
d. Si p y q son expresiones regulares, entonces también lo es $(p \circ q)$.
e. Si p es una expresión regular, entonces también lo es p^*.

Como ejemplo, si el alfabeto Σ fuera $\{x, y, z\}$, entonces $(x \cup (z \circ y))$ sería una expresión regular ya que $(z \circ y)$, por la regla d, es una expresión regular y en consecuencia, por la regla c, $(x \cup (z \circ y))$ sería una expresión regular.

Cada expresión regular r de un alfabeto Σ representa un lenguaje, denotado por $L(r)$, que se construye a partir de nuestros bloques de construcción. Para ser más precisos, $L(\varnothing)$ es el lenguaje $\varnothing$, y para cada $x \in \Sigma$, $L(x)$ es el lenguaje $\{x\}$. Además, si p y q son expresiones regulares, entonces $L((p \cup q)) = L(p) \cup L(q)$, $L((p \circ q)) = L(p) \circ L(q)$, y $L(p^*) = L(p)^*$. Por ejemplo, la expresión $(x \cup (z \circ y))$ representa el lenguaje $\{x, zy\}$. Es decir, representa el lenguaje generado al unir $\{x\}$ con la concatenación de $\{z\}$ y $\{y\}$. De forma similar, la expresión $((x \circ y)^* \cup z^*)$ (obtenida al aplicar la regla d a x y y, luego la regla e a $(x \circ y)$ y z, y por último la regla c a $(x \circ y)^*$ y z^*) representa el lenguaje que consiste en cadenas de cero o más copias del patrón xy además de las cadenas de cero o más z.

Así, las expresiones regulares ofrecen otro medio, además de los diagramas de transiciones, los autómatas y las gramáticas, para especificar lenguajes. El teorema 1.6 muestra por qué nos interesan los lenguajes representados por lenguajes regulares.

TEOREMA 1.6
Dado un alfabeto Σ, los lenguajes regulares de Σ son exactamente los lenguajes representados por las expresiones regulares de Σ.

DEMOSTRACIÓN
Primero debemos mostrar que un lenguaje representado por una expresión regular es regular. Ya hemos visto que el lenguaje $\varnothing$ es regular, así como cualquier lenguaje que contenga sólo una cadena de longitud uno. Además, hemos visto que la unión y la concatenación de dos lenguajes son regulares siempre y que la estrella de Kleene de cualquier lenguaje regular también es regular. Así, el primer paso de nuestra demostración está completo.

A continuación mostramos que cualquier lenguaje aceptado por un autómata finito puede representarse con una expresión regular. Suponemos que T es el diagrama de transiciones para un autómata finito y mostramos, por inducción para el número de estados en T que no son el inicial o los de aceptación, que existe una expresión regular que representa los lenguajes aceptados por T. Los diagramas que consideraremos son un poco más generales que los diagramas de transiciones tradicionales, ya que sus arcos pueden estar rotulados con expresiones regulares (para recorrer uno de estos arcos se requiere que la máquina lea un patrón compatible con la expresión). Si generalizados los lenguajes aceptados por estos

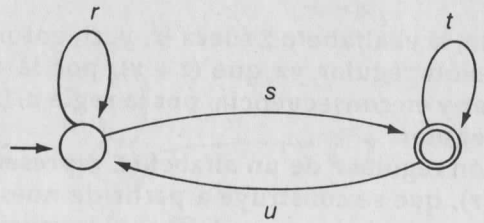


Figura 1.30 Arcos posibles en un diagrama de transiciones con dos estados, uno inicial y otro de aceptación

diagramas pueden representarse con expresiones regulares, entonces también puede representarse con ellas cualquier lenguaje aceptado por un diagrama de transiciones tradicional para un autómata finito.

Nuestro proceso de inducción supone que T sólo tiene un estado de aceptación. De lo contrario, para cada estado de aceptación podríamos hacer una copia aparte de T donde sólo ese estado sería de aceptación, encontrar expresiones regulares relacionadas con estos diagramas y luego formar la unión de estas expresiones a fin de obtener la expresión deseada para T.

Para comenzar nuestro proceso de inducción, supongamos entonces que T es un diagrama de transiciones generalizado con sólo un estado de aceptación, y que cada estado de T es un estado inicial o de aceptación. Entonces existen dos posibilidades: T contiene sólo un estado que es tanto el inicial como el de aceptación, o T contiene dos estados, de los cuales uno es el inicial y el otro es el de aceptación. En el primer caso, la expresión deseada no es más que la estrella de Kleene de la unión de las etiquetas que aparecen en los arcos del diagrama.

El segundo caso es un poco más complejo. Si existen varios arcos que conectan los mismos estados en la misma dirección, los sustituimos con un solo arco rotulado con la expresión regular que representa la unión de las etiquetas originales. Al llegar a este punto, T puede contener, a lo sumo, cuatro arcos: uno del estado inicial al estado inicial, uno del estado inicial al de aceptación, uno del estado de aceptación a sí mismo y uno del estado de aceptación al inicial. Representamos estos arcos con las etiquetas r, s, t y u respectivamente (véase Fig. 1.30).

Si el arco s no está presente en el diagrama, entonces la expresión regular relacionada es $\varnothing$ ya que no habría manera de llegar al estado de aceptación desde el inicial. De lo contrario, la estructura de la expresión regular relacionada dependerá de la ausencia o existencia del arco con etiqueta u. Si no existe este arco, entonces la expresión regular deseada es

$$((r^* \circ s) \circ t^*)$$

donde r y t son sustituidos por $\varnothing$ si no existe en el diagrama el arco correspondiente. Si existe un arco del estado de aceptación al estado inicial, entonces la expresión deseada es

$$(((r^* \circ s) \circ t^*) \circ (u \circ ((r^* \circ s) \circ t^*))^*)$$

donde, una vez más, r y t son sustituidos por $\varnothing$ si no existe en T el arco correspondiente.

Supongamos ahora que $n \in \mathbb{N}$ y que pueda representarse con una expresión regular el lenguaje aceptado por cualquier diagrama de transiciones generalizado que no tenga más de n estados que no sean el inicial o los de aceptación. Para cualquier diagrama de transiciones generalizado con $n + 1$ estados que no sean el inicial o los de aceptación, proseguimos de la siguiente manera (véase Fig. 1.31): seleccione un estado s_0 de T que no sea inicial o de aceptación; elimine de T ese estado y todos sus arcos incidentes; para cada par de arcos eliminados (uno que conduce de otro estado a s_0 y otro que sale de s_0 a otro estado, con etiquetas p y q respectivamente), dibuje un arco desde el origen del arco con etiqueta p al destino del arco con etiqueta q, y rotule este nuevo arco como $((p \circ r^*) \circ q)$, donde r es la unión de las etiquetas de los arcos que originalmente iban de s_0 a s_0, o bien $\varnothing$ si no existían tales arcos. El diagrama resultante es otro diagrama de transiciones generalizado que aceptará el mismo lenguaje que T y que sólo tiene n estados que no sean iniciales o de aceptación. Entonces, por nuestra hipótesis de inducción, es posible representar el lenguaje en cuestión con una expresión regular.

∎

El teorema 1.6 proporciona una forma más de caracterizar lenguajes regulares: son los lenguajes aceptados por autómatas finitos deterministas, los lenguajes aceptados por autómatas finitos no deterministas, los lenguajes generados por gramáticas regulares y los lenguajes representados por expresiones regulares.

De lo anterior podemos concluir que el poder de las expresiones regulares es un tanto limitado, pero no significa que estas sean inútiles. Por el contrario, dichas expresiones proporcionan detalles de la estructura de los lenguajes regulares que quizá no sean aparentes en otras caracterizaciones. Asimismo, las expresiones regulares proporcionan una manera relativamente concisa para expresar y comunicar lenguajes regulares. Por ejemplo, para describir un lenguaje regular utilizando un autómata finito, se requiere una definición de la función de transición, quizá en forma de diagrama o de tabla de transiciones; para describir un lenguaje por medio de una gramática, se

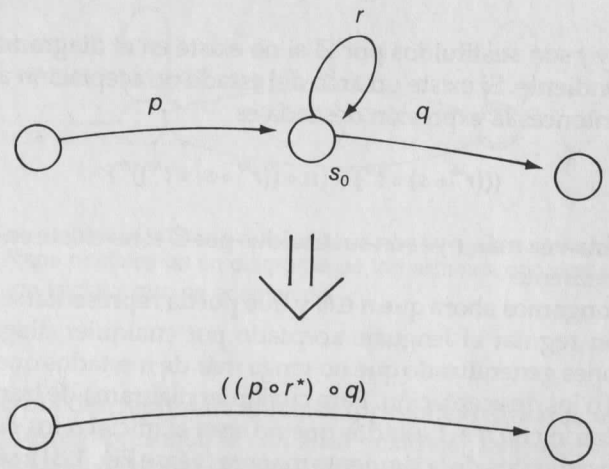


Figura 1.31 Eliminación de un estado s_0 de un diagrama de transiciones generalizado para un autómata finito

requiere una lista de reglas de reescritura; pero, utilizando una expresión regular, se puede describir un lenguaje en una sola línea.

Uno de los principales ejemplos de las ventajas de esta notación concisa se puede encontrar en muchos de los editores de texto que se emplean actualmente, los cuales permiten buscar un patrón específico en un archivo, para eliminarlo o reemplazarlo por otra cadena. En esta situación se requiere un sistema de notación para comunicar la forma del patrón a encontrar. Aunque la sintaxis que se emplee puede ser distinta de la que aquí se ha presentado, en estas aplicaciones es fundamental el tema de la expresión del patrón por medio de las operaciones de unión, concatenación y estrella de Kleene. Como ejemplo específico, al utilizar el editor *ed* de UNIX, el mandato

$$s/xx^*/z/$$

indica al sistema que sustituya la primera ocurrencia de una o más x por una sola z. En este caso, la expresión xx^* representa una sola x concatenada con la estrella de Kleene de x.

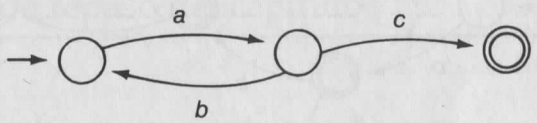
Por último, una consecuencia de los teoremas 1.3 y 1.6 es que los sistemas de comparación de patrones que se basan exclusivamente en la unión, la concatenación y la estrella de Kleene para la representación de patrones, son un tanto restrictivos en cuanto a los patrones que se pueden solicitar. Por otra parte, como testimonio del alcance de los lenguajes regulares, se han diseñado muchos sistemas de gran utilidad a pesar de estas restricciones.

Ejercicios

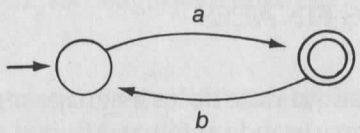
1. ¿Cuáles de los lenguajes descritos por las siguientes expresiones regulares para el alfabeto $\{x, y, z\}$ son infinitos? Describa, en una sola frase, el contenido de cada uno de estos lenguajes infinitos y elabore una lista exhaustiva de las cadenas de los lenguajes que sean finitos.

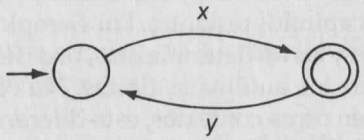
a. $(x \circ (y \circ z^*))$	e. $(y \circ y)^*$
b. $(x^* \circ (y \circ z))$	f. $(x^* \cup y^*)$
c. $((z \cup y) \circ x)$	g. $((x \circ x) \cup z)$
d. $(z \cup y)^*$	h. $((z \cup y) \cup x)$

2. Dibuje un diagrama de transiciones que acepte la estrella de Kleene del lenguaje aceptado por el siguiente diagrama.

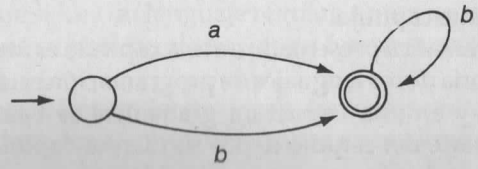


3. Dibuje un diagrama de transiciones que acepte la unión de los lenguajes aceptados por los siguientes diagramas.

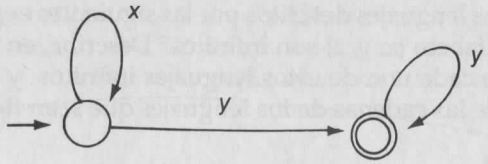




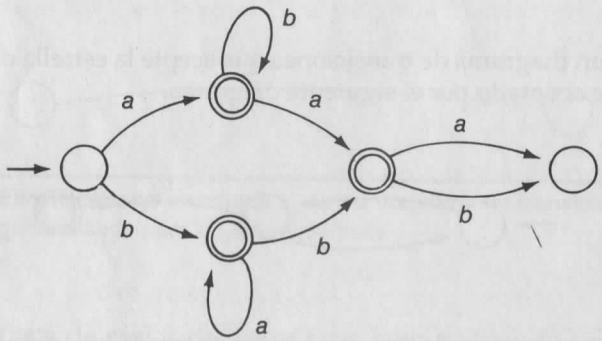
4. Dibuje un diagrama de transiciones que acepte la concatenación del lenguaje aceptado por



seguido por el lenguaje aceptado por



5. Construya una expresión regular que describa el lenguaje aceptado por el siguiente diagrama de transiciones.



1.7 COMENTARIOS FINALES

Este capítulo ha presentado la clase de los lenguajes regulares y ha mostrado cómo pueden caracterizarse usando autómatas finitos, gramáticas regulares y expresiones regulares. Además de ser importantes en sí, muchos de los conceptos que se presentan en este capítulo también sirven como introducción de los conceptos de los capítulos restantes. Por ejemplo, hemos presentado las máquinas deterministas y las no deterministas, una distinción que no afecta el poder computacional de los autómatas finitos. No obstante, en el siguiente capítulo veremos que, en otros contextos, esta diferencia puede tener ramificaciones importantes. También hemos visto cómo la tarea de análisis léxico de un compilador puede reducirse a un sencillo proceso de consulta de una tabla. Aunque las tablas tienen una utilidad potencial en este contexto, su empleo es más común en las rutinas de análisis sintáctico de los compiladores, algo que veremos en el siguiente capítulo.

El concepto de gramática presentado en este capítulo es importante ya que la sintaxis de la mayoría de los lenguajes de programación está descrita hoy en día por gramáticas, y en esta estructura gramatical se basa el proceso de compilación. Gran parte del estudio de los siguientes capítulos implicará la cuestión de cuáles son los lenguajes que pueden generar las gramáticas

cuyas reglas de reescritura se adhieren a distintas restricciones. Esto tiene relevancia en la medida en que, al hacerse más complejas las reglas de reescritura, también el algoritmo de reconocimiento del lenguaje asociado se vuelve más complicado. Así, si nos asignan la tarea de diseñar un compilador para un lenguaje de programación (o algún tipo de procesador para un lenguaje natural), nos gustaría basar nuestro diseño en la gramática más sencilla posible.

Sin embargo, nuestro objetivo final es descubrir hasta dónde pueden emplearse las máquinas para resolver problemas; hasta ahora, el problema que hemos elegido es el del análisis sintáctico de lenguajes. Como veremos en los capítulos siguientes, este problema tiene la complejidad suficiente para que la búsqueda de su solución nos lleve a los límites aparentes de los procesos computacionales.

Problemas de repaso del capítulo

1. Muestre que la colección de todas las cadenas de x, y y z, que contienen un número impar de x, un número impar de y y un número par de z es un lenguaje regular del alfabeto $\{x, y, z\}$.

2. Muestre que si L_1 y L_2 son lenguajes regulares, entonces $L_1 \cap L_2$ es regular.

3. Muestre que si L_1 y L_2 son lenguajes regulares, entonces $L_1 - L_2$ es regular.

4. a. Si Σ es un alfabeto, ¿es la colección de palíndromos de Σ^* un lenguaje regular? ¿Por qué? no.

   b. Si L es un lenguaje regular, ¿es la colección de palíndromos de L un lenguaje regular? ¿Por qué? sí.

5. Muestre que si L es un lenguaje regular del alfabeto Σ, entonces también es regular el lenguaje que consiste en todas las cadenas de la forma wv, donde $w \in L$ y $v \in \Sigma^* - L$.

6. Muestre que si L es un lenguaje regular, entonces también es regular el lenguaje que se obtiene al escribir en forma inversa las cadenas de L.

7. Muestre que si L es un lenguaje regular, entonces también es regular la colección de cadenas cuyas inversas se encuentran así mismo en L.

8. La cadena 527943001 es una lista de los dígitos que aparecen en el problema de adición presentado a continuación, si leyéramos las columnas en forma descendente y de derecha a izquierda (colocamos los ceros iniciales que se supone que están en la columna extrema izquierda). Diseñe un diagrama de transiciones para un autómata finito que acepte únicamente aquellas cadenas de dígitos que pueden obtenerse de un problema de adición de este tipo.

$$\begin{array}{r} 95 \\ +42 \\ \hline 137 \end{array}$$

9. Se dice que una regla de reescritura de una gramática, que sustituye un no terminal por un par terminal-no terminal, es **lineal por la izquierda** o **lineal por la derecha**, dependiendo de si el no terminal está a la izquierda o a la derecha (respectivamente) del terminal (la regla $N \to Mx$ sería lineal por la izquierda, mientras que la regla $N \to xM$ sería lineal por la derecha). Observe que nuestra definición de una gramática regular permitía reglas lineales por la derecha pero no por la izquierda. Muestre que los lenguajes regulares se generan de nuevo si cambiamos la definición para que permita reglas lineales por la izquierda, a la vez que se excluyen las lineales por la derecha.

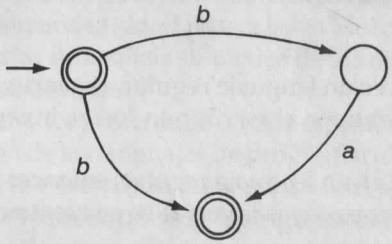
   Explique cómo se relaciona este problema con el problema 6.

10. Muestre que no es regular el lenguaje de $\{x, y\}$ que consiste en aquellas cadenas con el mismo número de x y y.

11. Muestre que es regular el lenguaje de $\{x, y\}$ que consiste en aquellas cadenas que no contienen tres x consecutivas.

12. Muestre que cualquier lenguaje finito es regular.

13. Diseñe un diagrama de transiciones para un autómata finito determinista que acepte las mismas cadenas que el autómata finito no determinista representado a continuación:



14. Suponga que permitimos que los autómatas finitos cambien de un estado a otro sin leer un símbolo de sus cintas de entrada. En un diagrama de transiciones, este tipo de transiciones normalmente se representa como un arco con etiqueta λ en vez de un símbolo del alfabeto, y se le conoce como **transición λ**. Muestre que cualquier diagrama de transiciones que tenga transiciones λ se puede modificar para que ya no contenga este tipo de transiciones pero que a la vez acepte el mismo lenguaje (el diagrama modificado puede ser no determinista).

15. Describa el lenguaje representado por cada una de las siguientes expresiones regulares.
   a. $(z \cup y)^* \circ x$
   b. $((x \circ x^*) \circ y \circ y^*))$
   c. $((x \circ x^*) \cup (y \circ y^*))$
   d. $((x^* \circ y^*) \circ z^*)$

16. Escriba expresiones regulares que describan los siguientes lenguajes.
   a. Todas las cadenas que consisten en un número impar de x.
   b. Todas las cadenas que consisten en un número impar de x y un número par de y.
   c. Todas las cadenas de x y y tales que cada y esté inmediatamente precedida por una x e inmediatamente seguida por una x.

17. Desarrolle una gramática para generar las cadenas que constituyen literales de tipo real en el lenguaje de programación Pascal.

18. Suponga que L es un lenguaje regular del alfabeto $\Sigma = \{x, y\}$. Muestre que $\{w: w \in \Sigma^* \text{ y } xw \in L\}$ y $\{w: w \in \Sigma^* \text{ y } wx \in L\}$ son también lenguajes regulares de Σ.

19. Encuentre un lenguaje regular (que no contenga λ) que no pueda ser el lenguaje aceptado por un autómata finito determinista con un solo estado de aceptación.

20. Muestre que si un autómata finito determinista M acepta una cadena que contiene más símbolos que estados tiene M, entonces M debe aceptar un número infinito de cadenas.

21. Diseñe un algoritmo que convierta diagramas de transiciones de autómatas finitos a expresiones regulares equivalentes.

22. Muestre que el lenguaje $\{x^n: n \text{ es primo}\}$ no es regular.

23. Muestre que el lenguaje $\{x^n: n = m^2 \text{ para algún } m \in \mathbb{N}\}$ no es regular.

24. Encuentre una expresión regular que represente la intersección de los lenguajes representados por cada uno de los siguientes pares de expresiones regulares.

x2

{lambda, x, xxxx, xxxxxxxxx, xxxxxxxxxxxxxxx, ...}

a. $(x \cup y^*)$ y $(x \cup y)^*$
b. $(x \circ (x \cup y)^*)$ y $((x \cup y)^* \circ y)$
c. $(((x \cup y) \circ y) \circ (x \cup y)^*)$ y $(y \circ (x \cup y)^*)$
d. $((x \cup y) \circ (x \cup \varnothing))$ y $((x \cup y) \circ (x \circ y)^*)$

Hacer por lo menos uno usando el método por leyes de DeMorgan

25. Muestre que si L es un lenguaje regular, entonces el conjunto de cadenas en L de longitud impar es también un lenguaje regular.
¿Se aplica lo mismo al conjunto de cadenas de longitud par? Justifique su respuesta.

26. Muestre que si L es un lenguaje regular que no contiene λ, entonces L puede ser aceptado por un autómata finito no determinista con un solo estado de aceptación.

27. Hemos mostrado que la unión de dos lenguajes regulares es regular. ¿Es siempre regular la unión de una colección de lenguajes regulares? Justifique su respuesta. sí.

28. Encuentre un diagrama de transiciones que acepte el lenguaje generado por la gramática regular que se presenta a continuación (el símbolo inicial es S). Luego, encuentre una expresión regular que represente el mismo lenguaje.

$$
\begin{aligned}
S &\to xN \\
S &\to x \\
N &\to yM \\
N &\to y \\
M &\to zN \\
M &\to z
\end{aligned}
$$

$L \to \lambda$

29. Muestre que un lenguaje regular que no contiene λ puede ser generada una gramática ~~por~~ regular que no contenga ninguna regla λ.

30. Muestre que si el lenguaje aceptado por un autómata finito contiene una cadena no vacía, entonces debe contener una cadena no vacía cuya longitud no sea mayor que el número de estados en el autómata.

31. Diseñe un diagrama de transiciones para un autómata finito que acepte el lenguaje consistente en las cadenas del alfabeto $\{w, x, y, z\}$ en donde el patrón xy siempre esté seguido por una w y donde al patrón yx siempre siga una z.

32. ¿Es cada lenguaje regular el lenguaje aceptado por un autómata finito determinista cuyo diagrama de transiciones pueda dibujarse sobre una superficie plana sin que alguno de sus arcos se crucen?

Jueves 12 a las 11:55am

Problemas de programación

1. Escriba un programa que evalúe cadenas de entrada para ver si se adhieren a los patrones descritos por la figura 1.17.

2. Diseñe e implante un algoritmo para evaluar gramáticas regulares y determinar si generan por lo menos una cadena no vacía (puede comenzar por traducir el problema 30 de repaso del capítulo a un enunciado acerca de gramáticas).

3. Desarrolle un paquete de software que genere automáticamente analizadores sintácticos para lenguajes regulares. En primer lugar, escriba un programa que acepte gramáticas regulares como entrada y genere como salida las tablas de transiciones correspondientes. Luego, escriba un programa que analice su cadena de entrada de acuerdo con la tabla de transiciones generada por el programa anterior.

4. Desarrolle un paquete similar al del problema de programación 3, pero que acepte expresiones regulares en vez de gramáticas regulares.

CAPÍTULO 2

Autómatas de pila y lenguajes independientes del contexto

2.1 Autómatas de pila
Definición de los autómatas de pila
Autómatas de pila como aceptadores de lenguajes
2.2 Gramáticas independientes del contexto
Definición de las gramáticas independientes del contexto
Gramáticas independientes del contexto y autómatas de pila
Forma normal de Chomsky
2.3 Límites de los autómatas de pila
Alcance de los lenguajes independientes del contexto
Autómatas de pila deterministas
Principio de preanálisis
2.4 Analizadores sintácticos *LL*(*k*)
Proceso de análisis sintáctico *LL*
Aplicación del principio de preanálisis
Tablas de análisis sintáctico *LL*
2.5 Analizadores sintácticos *LR*(*k*)
Proceso de análisis sintáctico *LR*
Implantación de analizadores sintácticos *LR*(*k*)
Tablas de análisis sintáctico *LR*
Comparación entre los analizadores sintácticos *LR*(*k*) y *LL*(*k*)
2.6 Comentarios finales

En el capítulo anterior vimos cómo se pueden construir analizadores léxicos utilizando los principios de los autómatas finitos, e investigamos los lenguajes regulares que estos autómatas son capaces de reconocer. También vimos que las técnicas sencillas relacionadas con los autómatas finitos son limitadas; específicamente, encontramos que los sistemas de

análisis sintáctico basados en estos autómatas no eran capaces de manejar gran parte de las estructuras sintácticas que se presentan en los lenguajes de programación generales.

En este capítulo generalizamos los conceptos de autómatas finitos y gramáticas regulares a fin de obtener técnicas para el análisis sintáctico de una mayor gama de lenguajes, conocidos como lenguajes independientes del contexto. Para esto, agregamos al autómata un sistema de memoria interna (en forma de pila). Esta adición incrementa de manera considerable el potencial de procesamiento de lenguaje del autómata y proporciona un marco en el cual se formulan varios algoritmos eficientes para el análisis sintáctico; el capítulo concluye con un estudio de las técnicas de análisis sintáctico que se encuentran en los compiladores modernos.

Para clasificar los lenguajes reconocidos por los autómatas mejorados, utilizaremos de nuevo el concepto de gramática. Al permitir mayor complejidad en la estructura de las reglas de reescritura de la gramática, seremos capaces de identificar las gramáticas que generan los lenguajes reconocidos por nuestras máquinas mejoradas. Esta caracterización gramatical será de utilidad en varias situaciones; con estas gramáticas se describe la sintaxis de la mayoría de los lenguajes de programación modernos.

2.1 AUTÓMATAS DE PILA

Como vimos en el capítulo 1, no existe ningún autómata finito que pueda reconocer el lenguaje $\{x^n y^n : n \in \mathbb{N}\}$. De hecho, este lenguaje (donde x era un paréntesis izquierdo y y un paréntesis derecho) fue nuestro principal ejemplo de las limitaciones de los autómatas finitos. Presentamos la hipótesis de que este problema ocurre porque los autómatas finitos no tienen forma de recordar cuántas x se detectaron en la primera parte de la cadena, por lo que eran incapaces de verificar si existía el mismo número de y. Por consiguiente, ahora especulamos que el problema podría resolverse añadiendo algún tipo de memoria a la máquina conceptual en consideración.

Definición de los autómatas de pila

Después de esta introducción, presentamos la clase de máquinas conocidas como **autómatas de pila**; una máquina de este tipo se representa en la figura 2.1. Al igual que un autómata finito, un autómata de pila cuenta con un flujo de entrada y un mecanismo de control que puede encontrarse en uno de entre un número finito de estados. Uno de estos estados se designa como el inicial y por lo menos un estado se designa como estado de aceptación. La principal diferencia entre los autómatas de pila y los finitos es que los primeros cuentan con una pila en donde pueden almacenar información para recuperarla más tarde.

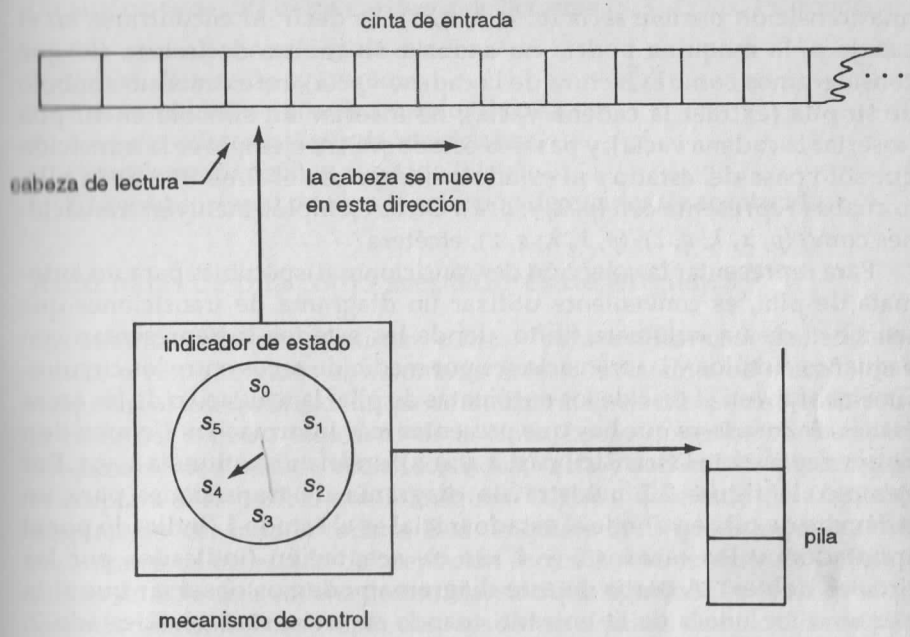


Figura 2.1 Autómata de pila

Los símbolos que pueden almacenarse en esta pila (conocidos como **símbolos de pila** de la máquina) constituyen un conjunto finito que puede incluir algunos o todos los símbolos del alfabeto de la máquina y quizá algunos símbolos adicionales que la máquina utiliza como marcas internas. Por ejemplo, una máquina podrá almacenar símbolos especiales en su pila para separar secciones que tengan interpretaciones distintas. Para ser más precisos, si una máquina inserta un símbolo especial en la pila antes de efectuar algún otro cálculo, entonces la presencia de ese símbolo en la cima de la pila puede usarse como indicador de "pila vacía" para cálculos posteriores. En nuestros ejemplos adoptamos el símbolo # para este fin.

Las transiciones que ejecutan los autómatas de pila deben ser variantes de la siguiente secuencia básica: leer un símbolo de la entrada, extraer un símbolo de la pila, insertar un símbolo en la pila y pasar a un nuevo estado. Este proceso se representa con la notación $(p, x, s; q, y)$, donde p, x, s, q y y son, respectivamente el estado actual, el símbolo del alfabeto que se lee de la entrada, el símbolo que se extrae de la pila, el nuevo estado y el símbolo que se inserta en la pila. Esta notación está diseñada para indicar que, el estado actual, el símbolo de entrada y el símbolo en la cima de la pila ayudan a determinar conjuntamente el nuevo estado y el símbolo que deberá insertarse en la pila.

Se obtienen variantes de este proceso básico de transición permitiendo que las transiciones lean, extraigan o inserten la cadena vacía. Por ejemplo,

una transición posible sería $(p, \lambda, \lambda; q, \lambda)$. Es decir, al encontrarse en el estado p, la máquina podría no avanzar su cabeza de lectura (lo que consideramos como la lectura de la cadena vacía), no extraer un símbolo de su pila (extraer la cadena vacía), no insertar un símbolo en su pila (insertar la cadena vacía), y pasar al estado q. Otro ejemplo es la transición que sólo pasa del estado p al estado q extrayendo el símbolo s de la pila, lo cual se representa con $(p, \lambda, s; q, \lambda)$. Otros ejemplos incluyen transiciones como $(p, x, \lambda; q, z)$, $(p, \lambda, \lambda; q, z)$, etcétera.

Para representar la colección de transiciones disponibles para un autómata de pila, es conveniente utilizar un diagrama de transiciones que semeje el de un autómata finito, donde los estados se representan con pequeños círculos y las transiciones por medio de arcos entre los círculos. Sin embargo, en el caso de los autómatas de pila, la rotulación de los arcos es más elaborada ya que hay que presentar más información. Un arco de p a q que representa la transición $(p, x, y; q, z)$ tendría una etiqueta $x, y; z$. Por ejemplo, la figura 2.2 muestra un diagrama de transiciones para un autómata de pila en donde el estado inicial es el estado 1 (indicado por el apuntador) y los estados 1 y 4 son de aceptación (indicados por los círculos dobles). A partir de este diagrama podemos observar que si la máquina lee una x de la entrada cuando se encuentra en el estado 2, insertará una x en la pila y regresará al estado 2; si la máquina lee una y de la entrada y puede extraer una x de la pila cuando se encuentra en el estado 3, regresará al estado 3; o si el símbolo # se encuentra en la cima de la pila cuando la máquina se halla en el estado 3, la máquina puede extraer este símbolo y pasar al estado 4.

Como sucede con los autómatas finitos, es posible implantar los autómatas de pila con varias tecnologías, por lo que aislamos las propiedades definitivas de los autómatas de pila en la siguiente definición formal.

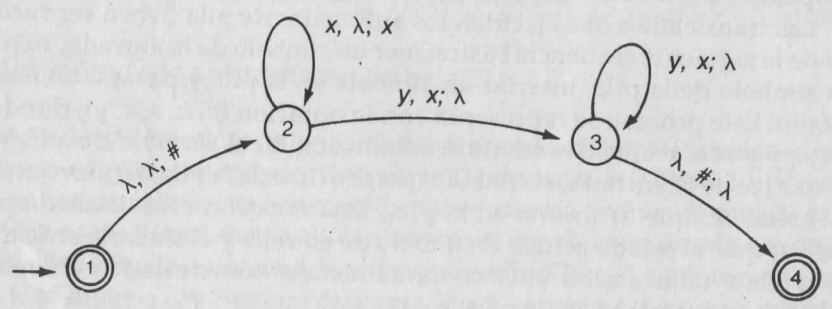


Figura 2.2 Diagrama de transiciones para un autómata de pila

Un autómata de pila es una séxtupla de la forma $(S, \Sigma, \Gamma, T, \iota, F)$, donde:

a. S es una colección finita de estados.
b. Σ es el alfabeto de la máquina.
c. Γ es la colección finita de símbolos de pila.
d. T es una colección finita de transiciones.
e. ι (un elemento de S) es el estado inicial.
f. F (un subconjunto de S) es la colección de estados de aceptación.

Autómatas de pila como aceptadores de lenguajes

Los autómatas de pila se pueden utilizar para analizar cadenas, en forma similar a como se usan los autómatas finitos. En la cinta de entrada de la máquina colocamos la cadena que se analizará, con la cabeza de lectura sobre la celda del extremo izquierdo de la cinta. Luego ponemos en marcha la máquina desde su estado inicial, con la pila vacía, y declaramos que la cadena se aceptará si es posible que la máquina llegue a un estado de aceptación después de leer toda la cadena. Esto no quiere decir que la máquina deba encontrarse en un estado de aceptación inmediatamente después de leer el último símbolo de la cadena de entrada, sino que significa que ya no tiene que leer más de la cinta. Por ejemplo, después de leer el último símbolo de la entrada, un autómata de pila puede ejecutar varias transiciones de la forma $(p, \lambda, x; q, y)$ antes de aceptar la cadena.

Usamos la palabra "posible" al definir la aceptación de cadenas en un autómata de pila ya que los autómatas de este tipo que aquí se consideran son no deterministas; no hemos establecido restricción alguna con respecto al número de transiciones que son aplicables en un instante determinado. Por lo tanto, como sucede con los autómatas finitos no deterministas, pueden existir varias secuencias de transiciones que se recorran a partir de la configuración inicial de la máquina, de las cuales sólo una debe conducir a un estado de aceptación para que declaremos la aceptación de la cadena (es lamentable que la terminología común no haga hincapié en la naturaleza no determinista de los autómatas de pila. Técnicamente, deberían llamarse autómatas de pila no deterministas).

Siguiendo los caminos establecidos para los autómatas finitos, nos referimos a la colección de todas las cadenas aceptadas por un autómata de pila M como el lenguaje aceptado por la máquina, representado por $L(M)$. Subrayamos de nuevo que el lenguaje $L(M)$ no es cualquier colección de cadenas aceptadas por M, sino *la* colección de *todas* las cadenas que acepta M.

Se puede obtener una importánte clase de máquinas restringiendo las transiciones disponibles para un autómata de pila a las de la forma $(p, x, \lambda; q, \lambda)$. Los pasos de esta forma ignoran que la máquina tiene una pila y, por consiguiente, las actividades de la máquina dependen exclusivamente del estado y el símbolo de entrada actuales. Por ende, la clase de máquinas construidas de esta manera es la clase de los autómatas finitos. Por lo tanto, *los lenguajes aceptados por los autómatas de pila incluyen los lenguajes regulares.*

Los autómatas de pila también pueden aceptar lenguajes que no pueden aceptar los autómatas finitos, por ejemplo el lenguaje $\{x^n y^n : n \in \mathbb{N}\}$. De hecho, la figura 2.2 es un diagrama de transiciones de dicha máquina. El primer paso es marcar la parte inferior de la pila con el símbolo # y luego insertar en la pila las x conforme se lean de la entrada. Luego la máquina extrae una x de la pila cada vez que se lee una y. De esta manera, cuando el símbolo # reaparece en la parte superior de la pila, se ha leído el mismo número de y y x. Observe que, como el estado inicial es también un estado de aceptación, se permite que la máquina acepte la cadena x^0y^0, que es λ.

Antes de concluir esta sección, se necesitan algunos comentarios adicionales. Recuerde que el criterio de aceptación que se proporcionó antes permite que un autómata de pila declare la aceptación de una cadena sin que tenga que vaciar antes su pila. Por ejemplo, un autómata de pila basado en el diagrama de la figura 2.3 aceptará el lenguaje $\{x^m y^n : m, n \in \mathbb{N}^+ \text{ y } m \geq n\}$, pues se aceptarán aquellas cadenas con más x que y aunque queden x en la pila (observe que este autómata no aceptaría cadenas con más y que x, pues no podría leer todos los símbolos de dicha cadena).

Se podría conjeturar que la aplicación de una teoría basada en estos autómatas nos llevaría a módulos de programa que devolvieran el control a otros módulos dejando en la pila residuos de sus cálculos que podrían ocasionar confusiones en cálculos posteriores. Por esto, es frecuente que se prefiera considerar únicamente los autómatas de pila que vacían sus pilas antes de llegar a un estado de aceptación. En el teorema 2.1 podemos ver que esta restricción no reduce el poder de estas máquinas.

TEOREMA 2.1

Para cada autómata de pila que acepte cadenas sin vaciar su pila, existe un autómata que acepta el mismo lenguaje pero que vacía su pila antes de llegar a un estado de aceptación.

DEMOSTRACIÓN

Suponga que $M = (S, \Sigma, \Gamma, T, \iota, F)$ es un autómata de pila que acepta cadenas sin tener que vaciar necesariamente su pila. Podemos modificar M de la manera siguiente:

1. Elimine la designación "inicial" del estado inicial de M. Luego, añada un nuevo estado inicial y una transición que permita a M pasar del nuevo estado inicial al anterior a la vez que inserta en la pila un símbolo especial # (que no se encontraba anteriormente en Γ).

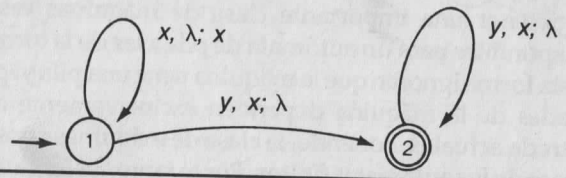


Figura 2.3 Autómata de pila que puede aceptar cadenas sin una pila vacía

2. Elimine la característica de aceptación de cada estado de aceptación de M. Luego, añada un estado p junto con las transiciones que permiten a la máquina pasar de cada uno de los antiguos estados de aceptación a p sin leer, extraer o insertar un símbolo.

3. Para cada x en Γ (sin incluir #), introduzca la transición $(p, \lambda, x; p, \lambda)$.

4. Añada un nuevo estado de aceptación q y la transición $(p, \lambda, \#; q, \lambda)$.

Observe que la versión modificada de M sólo marca el fondo de su pila antes de efectuar algún cálculo y luego simula los cálculos de la máquina original hasta el punto donde la máquina original habría declarado la aceptación de la entrada. Aquí la máquina modificada pasa al estado p, vacía su pila y luego pasa a su estado de aceptación q quitando la marca de fin de pila. Así, tanto la máquina original como la modificada aceptan las mismas cadenas, aunque la versión modificada llega a su estado de aceptación únicamente cuando su pila está vacía.

∎

La figura 2.4 muestra el resultado de aplicar la técnica de la demostración anterior al diagrama de la figura 2.3. Un autómata de pila basado en este nuevo diagrama aceptará exactamente las mismas cadenas que el original, pero no puede aceptar una cadena a menos que su pila se encuentre vacía.

Por último, es importante recordar que los autómatas que aquí se consideran son no deterministas. El proceso de modificación descrito en la demostración del teorema 2.1 puede introducir varios puntos de no determinismo por medio de las transiciones que conducen de los antiguos estados de aceptación al nuevo estado p.

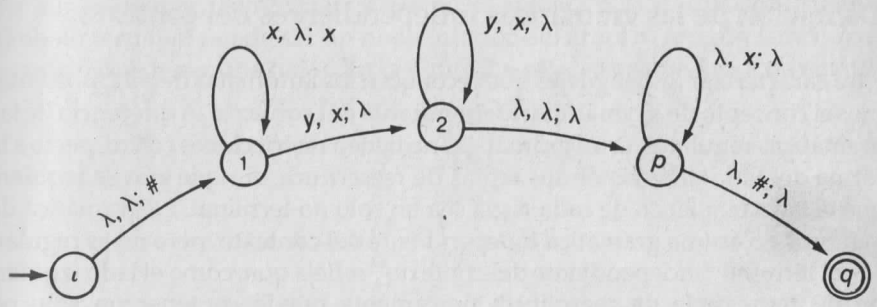
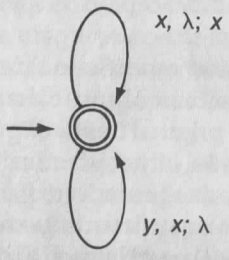


Figura 2.4 El diagrama de la figura 2.3 después de modificarlo para que vacíe su pila antes de aceptar una cadena

Ejercicios

1. Diseñe un autómata de pila M tal que $L(M) = \{x^n y^m x^n : m, n \in \mathbb{N}\}$

2. ¿Cuál es el lenguaje que acepta el autómata de pila cuyo diagrama de transiciones se presenta a continuación?



3. Modifique el diagrama de transiciones del ejercicio 2 para que el autómata de pila acepte el mismo conjunto de cadenas pero con su pila vacía.

4. Muestre cómo pueden combinarse dos autómatas de pila M_1 y M_2 para formar un solo autómata de pila que acepte el lenguaje $L(M_1) \cup L(M_2)$.

2.2 GRAMÁTICAS INDEPENDIENTES DEL CONTEXTO

Ahora que hemos extendido las máquinas en consideración de autómatas finitos a autómatas de pila, nuestra investigación se centra en cuáles son los lenguajes que estos autómatas extendidos pueden reconocer. Comenzamos la búsqueda de la respuesta regresando al concepto de las gramáticas.

Definición de las gramáticas independientes del contexto

Para caracterizar los lenguajes que reconocen los autómatas de pila, presentamos el concepto de **gramática independiente del contexto.** A diferencia de las gramáticas regulares, estas gramáticas no tienen restricciones con respecto a la forma del lado derecho de sus reglas de reescritura, aunque aún se requiere que el lado izquierdo de cada regla sea un solo no terminal. La gramática de la figura 2.5 es una gramática independiente del contexto, pero no es regular.

El término "independiente del contexto" refleja que, como el lado izquierdo de cada regla de reescritura únicamente puede contener un solo no terminal, la regla puede aplicarse sin importar el contexto donde se encuentre dicho no terminal. Por el contrario, considere una regla de reescritura cuyo lado izquierdo contenga más de un no terminal, como $xNy \rightarrow xzy$. Esta regla

dice que el no terminal N puede sustituirse con el terminal z sólo cuando esté rodeado por los terminales x y y. Por lo tanto, la capacidad de eliminar N aplicando la regla dependerá del contexto en vez de ser independiente.

Al igual que las gramáticas regulares, las gramáticas independientes del contexto generan cadenas por medio de derivaciones. No obstante, en el caso de las gramáticas independientes del contexto pueden surgir dudas con respecto a cuál será el no terminal que deberá reemplazarse en un paso específico de la derivación. Por ejemplo, al generar una cadena con la gramática de la figura 2.5, el primer paso produce la cadena $zMNz$, que presenta la opción de reemplazar el no terminal M o el N en el siguiente paso. Por consiguiente, para generar la cadena $zazabzbz$, se podría producir la derivación

$$S \Rightarrow zMNz \Rightarrow zaMaNz \Rightarrow zazaNz \Rightarrow zazabNbz \Rightarrow zazabzbz$$

siguiendo la regla rutinaria de aplicar siempre una regla de reescritura al no terminal situado más a la izquierda en la cadena actual (esto se llama **derivación por la izquierda**). También podría producirse la derivación

$$S \Rightarrow zMNz \Rightarrow zMbNbz \Rightarrow zMbzbz \Rightarrow zaMabzbz \Rightarrow zazabzbz$$

aplicando siempre la regla de reescritura al no terminal situado más a la derecha, lo cual daría como resultado una **derivación por la derecha.** Incluso se podrían seguir otros patrones y obtener otras derivaciones de la misma cadena.

Lo cierto es que el orden en que se apliquen las reglas de reescritura no afecta la determinación de si una cadena puede generarse a partir de cierta gramática independiente del contexto. Esto resulta obvio cuando reconocemos que *si una cadena puede generarse a partir de alguna derivación, entonces puede ser generada por una derivación por la izquierda*. Para ver esto, primero consideramos el árbol de análisis sintáctico asociado a una derivación.

Un **árbol de análisis sintáctico** no es más que un árbol cuyos nodos representan terminales y no terminales de la gramática, donde el nodo raíz es el símbolo de inicio de la gramática y los hijos de cada nodo no terminal son los símbolos que reemplazan a ese no terminal en la derivación (ningún símbolo terminal puede ser un nodo interior del árbol ni ningún símbolo no terminal puede ser una hoja). En la figura 2.6 se presenta un árbol de análisis

$$\begin{aligned} S &\rightarrow zMNz \\ M &\rightarrow aMa \\ M &\rightarrow z \\ N &\rightarrow bNb \\ N &\rightarrow z \end{aligned}$$

Figura 2.5 Gramática independiente del contexto que genera cadenas de la forma $za^n za^n b^m zb^m z$, donde $m, n \in \mathbb{N}$

sintáctico para la cadena *zazabzbz* usando la gramática de la figura 2.5 y cualquiera de las derivaciones anteriores.

Ahora, para ver que cualquier cadena generada por una gramática independiente del contexto se puede generar con una derivación por la izquierda, observamos que las derivaciones que corresponden al mismo árbol de análisis sintáctico únicamente difieren en cuanto al orden en que se aplican las reglas de reescritura. El orden de aplicación de las reglas sólo refleja el orden de construcción de las ramas del árbol. Una derivación por la izquierda corresponde a la construcción del árbol sintáctico comenzando por la rama izquierda, mientras que una derivación por la derecha corresponde a una construcción que se inicia por la rama derecha. Sin embargo, el orden de construcción de las ramas no afecta la estructura final del árbol, ya que cada rama es independiente de las demás. Por lo tanto, dada una derivación que no es por la izquierda, se puede construir el árbol de análisis sintáctico asociado y luego elaborar un derivación por la izquierda de la misma cadena, aplicando una "evaluación por la izquierda" sistemática del árbol.

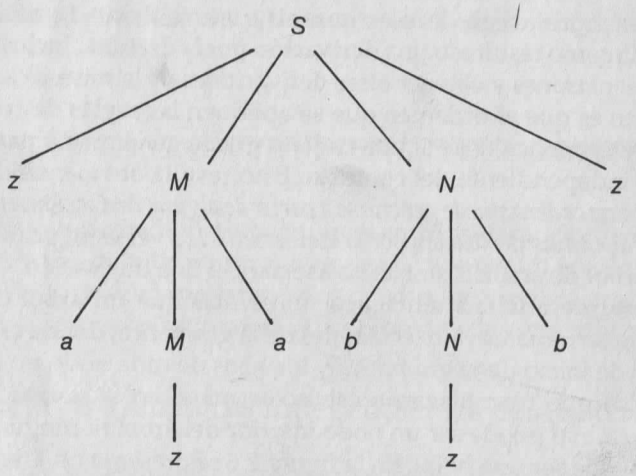


Figura 2.6 Árbol de análisis sintáctico para la cadena *zazabzbz* utilizando la gramática de la figura 2.5

$$S \to xSy$$
$$S \to \lambda$$

Figura 2.7 Gramática independiente del contexto que genera cadenas de la forma $x^n y^n$ donde $n \in \mathbb{N}$

Finalmente, observamos que la flexibilidad que ofrecen las gramáticas independientes del contexto permite la construcción de una gramática que genera el lenguaje $\{x^n y^n : n \in \mathbb{N}\}$, que se presenta en la figura 2.7. Este ejemplo, combinado con el hecho de que cualquier gramática regular es una gramática independiente del contexto, nos permite llegar a la conclusión de que las gramáticas independientes del contexto generan una mayor colección de lenguajes que las gramáticas regulares. Los lenguajes generados por gramáticas independientes del contexto se denominan **lenguajes independientes del contexto.**

Gramáticas independientes del contexto y autómatas de pila

Antes de considerar la relación entre las gramáticas independientes del contexto y los autómatas de pila, debemos esclarecer un aspecto de la notación. En los análisis que se presentan a continuación, será conveniente considerar transiciones únicas que insertan más de un símbolo en la pila, como $(p, a, s; q, xyz)$. En este caso, se insertan en la pila los símbolos z, y y x (en ese orden). Así, después de efectuar la transición, x se hallará en la cima de la pila (con y debajo y z en el fondo). Observe que las transiciones de este tipo sólo representan una forma conveniente de notación y no añaden capacidades adicionales a la máquina. De hecho, la transición de inserción múltiple $(p, a, s; q, xyz)$ podría simularse con la secuencia de transiciones tradicionales $(p, a, s; q_1, z)$, $(q_1, \lambda, \lambda; q_2, y)$ y $(q_2, \lambda, \lambda; q, x)$, donde q_1 y q_2 son estados adicionales a los que no puede llegar ninguna otra secuencia de transiciones.

Ahora debemos mostrar que los *lenguajes generados por gramáticas independientes del contexto son exactamente los mismos lenguajes que aceptan los autómatas de pila.* Esto se hará en dos etapas. Primero mostramos que para cualquier gramática G independiente del contexto existe un autómata de pila M tal que $L(M) = L(G)$ (Teorema 2.2). Luego mostramos que para cualquier autómata de pila M existe una gramática G independiente del contexto tal que $L(G) = L(M)$ (Teorema 2.3).

TEOREMA 2.2
Para cada gramática G independiente del contexto, existe un autómata de pila M tal que $L(G) = L(M)$.

DEMOSTRACIÓN
Dada una gramática G independiente del contexto, construimos un autómata de pila M de la manera siguiente:

1. Designe el alfabeto de M como los símbolos terminales de G, y los símbolos de pila de M como los símbolos terminales y no terminales de G, junto con el símbolo especial # (podemos suponer que # no es un símbolo terminal o no terminal de G).

2. Designe los estados de M como ι, p, q y f, donde ι es el estado inicial y f es el único estado de aceptación.
3. Introduzca la transición $(\iota, \lambda, \lambda; p, \#)$.
4. Introduzca una transición $(p, \lambda, \lambda; q, S)$ donde S es el símbolo inicial de G.
5. Introduzca una transición de la forma $(q, \lambda, N; q, w)$ para cada regla de reescritura $N \rightarrow w$ en G (aquí empleamos nuestra nueva convención que permite que una sola transición inserte más de un símbolo de pila. Específicamente, w puede ser una cadena de cero o más símbolos, incluyendo terminales y no terminales).
6. Introduzca una transición de la forma $(q, x, x; q, \lambda)$ para cada terminal x de G (es decir, para cada símbolo del alfabeto de M).
7. Introduzca la transición $(q, \lambda, \#; f, \lambda)$.

Un autómata de pila construido de esta manera analizará una cadena de entrada marcando primero el fondo de la pila con el símbolo #, luego insertando en la pila el símbolo inicial de la gramática y después entrando al estado q. De ahí y hasta que el símbolo # vuelva a aparecer en la cima de la pila, el autómata extraerá un no terminal de la pila y lo reemplazará con el lado derecho de una regla de reescritura aplicable, o extraerá un terminal de la pila a la vez que lee el mismo terminal en la entrada. Una vez que el símbolo # regresa a la cima de la pila, el autómata cambiará a su estado de aceptación f, indicando que la entrada recibida hasta ese punto es aceptable.

Observe que la cadena de símbolos que integran la parte derecha de una regla de reescritura se inserta en la pila de derecha a izquierda. Así, el no terminal situado mas a la izquierda será el primero en surgir en la cima de la pila; por tanto, también será el primer no terminal de la pila que se reemplazará. Por consiguiente, el autómata analiza su entrada efectuando una derivación por la izquierda de acuerdo con las reglas de la gramática en la cual se basa. Sin embargo, como ya vimos, las cadenas generadas por una gramática independiente del contexto son exactamente aquellas que tienen una derivación por la izquierda. Entonces, el autómata acepta exactamente el mismo lenguaje que genera la gramática.

∎

Quizás con un ejemplo pueda comprenderse mejor la función de las distintas transiciones construidas en la demostración del teorema 2.2. Consideremos los pasos que ejecuta el autómata de pila descrito en el diagrama de transiciones de la figura 2.8, construido a partir de la gramática independiente del contexto de la figura 2.5. Para evaluar la cadena *zazabzbz*, iniciamos la máquina con la configuración representada en la figura 2.9a. A partir de esta

configuración, la máquina marca el fondo de la pila con el símbolo # y cambia al estado q a la vez que inserta el no terminal S en la pila, para llegar a la configuración representada en la figura 2.9 b. A partir de este punto, la pila se emplea para contener una descripción de la estructura que la máquina espera encontrar en la parte restante de su flujo de entrada. Así, el símbolo S que se encuentra actualmente en la parte superior de la pila indica que la máquina espera que los símbolos restantes de su entrada constituyan una estructura que pueda generarse a partir del no terminal S.

Sin embargo, puesto que S no es un terminal, la máquina no puede esperar que el contenido de su pila aparezca explícitamente en la entrada, por lo que hay que reemplazar el no terminal antes de que la máquina intente comparar su pila directamente con el flujo de entrada. De hecho, se trata de una regla general que sigue la máquina: en cualquier instante, la cima de la pila contiene un no terminal que debe reemplazarse con una descripción equivalente, aunque más detallada, de la estructura. Este es el propósito de las transiciones incluidas en la regla 5 de la demostración del teorema 2.2. Con su ejecución se reemplaza un no terminal de la cima de la pila por una descripción más detallada de la estructura, de acuerdo con alguna regla de reescritura de la gramática original. Así, la máquina de nuestro ejemplo continúa con la ejecución de la transición $(q, \lambda, S; q, zMNz)$ para llegar a la configuración representada en la figura 2.9c.

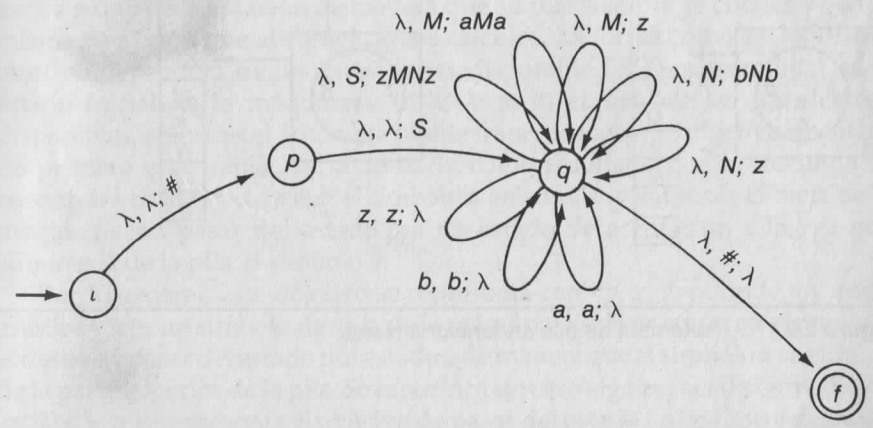


Figura 2.8 Diagrama de transiciones de un autómata de pila construido a partir de la gramática de la figura 2.5 utilizando las técnicas presentadas en la demostración del teorema 2.2

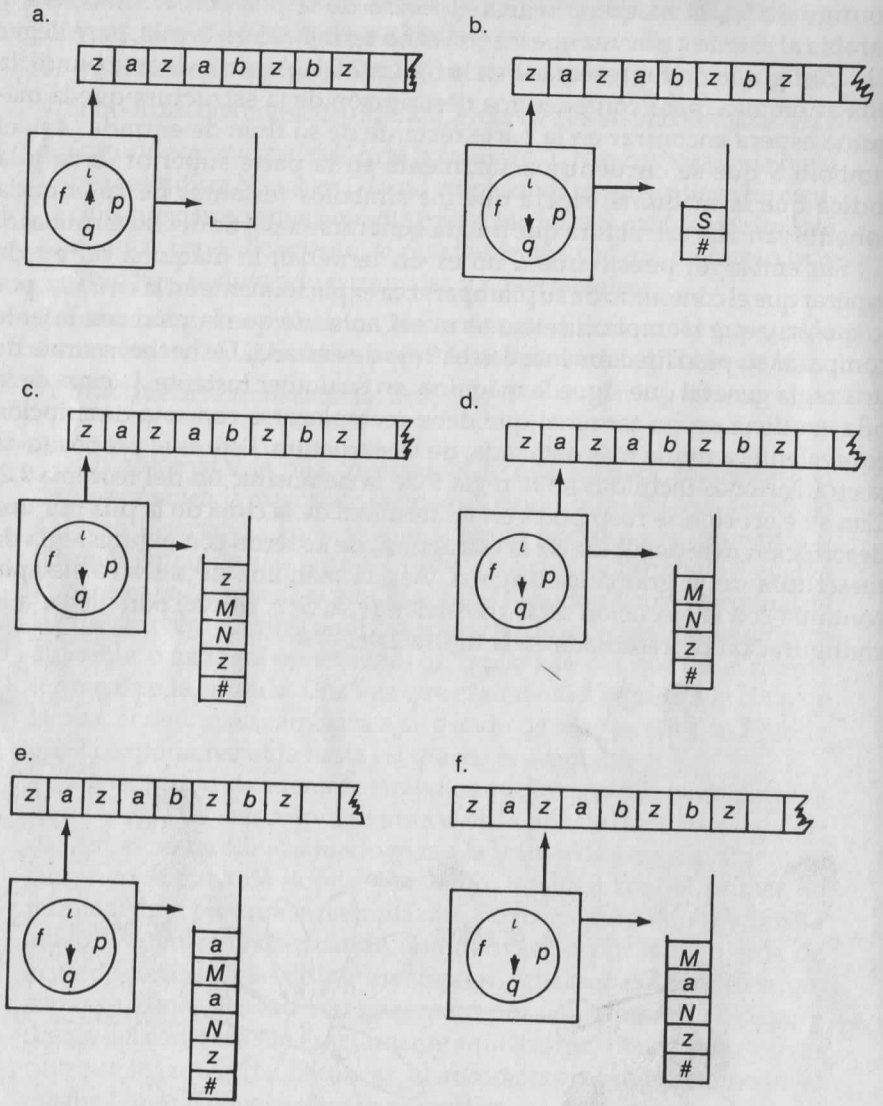


Figura 2.9 Autómata de pila en funcionamiento

Ahora la cima de la pila de la máquina contiene el terminal z y, por tanto, puede compararse con la cadena de entrada a través de la transición $(q, z, z; q, \lambda)$. Después de ejecutar esta transición, la pila de la máquina contiene MNz y los símbolos restantes de la cadena de entrada son $azabzbz$, como se muestra en la figura 2.9d.

De nuevo, la cima de la pila contiene un no terminal que la máquina debe reemplazar. Sin embargo, a diferencia de la sustitución anterior del no terminal S, existen dos reglas de reescritura que pueden aplicarse para sustituir el no terminal M. El autómata de pila podría ejecutar la transición $(q, \lambda, M; q, z)$, lo que ocasionaría que la máquina fallara en su intento por aceptar la cadena, o podría ejecutar $(q, \lambda, M; q, aMa)$, que en este caso sería la opción correcta (este autómata es no determinista). Supongamos que se elige la opción correcta ya que nuestro objetivo es observar cómo la máquina podría aceptar la cadena de entrada (recuerde que una cadena se encuentra en el lenguaje de una máquina no determinista si es *posible* que la máquina acepte la cadena). Después de ejecutar $(q, \lambda, M; q, aMa)$ la máquina aparece como se muestra en la figura 2.9e, con el terminal a en la cima de la pila. Entonces, este terminal se compara con la cadena de entrada a través de la transición $(q, a, a; q, \lambda)$, lo que deja la pila con $MaNz$ y los símbolos $zabzbz$ por leerse de la cadena de entrada, como se ilustra en la figura 2.9f.

En la figura 2.10 se resumen las actividades restantes de la máquina. Observe que la lectura del último símbolo de la entrada descubre la marca # en la pila. Entonces, la transición $(q, \lambda, \#; f, \lambda)$ transfiere la máquina al estado de aceptación f, donde se declara la aceptación de la entrada.

Preparémonos ahora para el teorema 2.3. Suponga que se nos da un autómata de pila que acepta cadenas sólo cuando su pila está vacía. Entonces, desde el punto de vista del autómata, su tarea es tratar de pasar de su estado inicial a uno de aceptación de manera que su pila siempre se encuentre en la misma condición que al comenzar los cálculos. La forma cómo se logre este objetivo dependerá de las transiciones disponibles. Si, por ejemplo, ι es el estado inicial de la máquina y $(\iota, \lambda, \lambda; p, \#)$ es una de las transiciones disponibles, entonces el autómata puede tratar de lograr su objetivo ejecutando primero esta transición. Esto daría como resultado que la máquina se encontrara en el estado p con el símbolo # en su pila, y entonces la meta de la máquina sería pasar del estado p a un estado de aceptación a la vez que eliminaría de la pila el símbolo #.

Por lo general, este objetivo se representa con $<p, x, q>$, donde p y q son estados y x es un símbolo de pila de la máquina. Es decir, $<p, x, q>$ representa el deseo de pasar del estado p al estado q de manera que el símbolo x se elimine de la parte superior de la pila. Se especifica un caso algo especial por medio de $<p, \lambda, q>$, que representa el objetivo de pasar del estado p al estado q de modo que la pila no se altere. En otras palabras, al llegar al estado q, la pila será la misma que existía cuando se encontraba en el estado p. Un ejemplo de esta situación es el objetivo original, antes mencionado, de pasar del estado inicial a uno de aceptación. De hecho, para cada estado de aceptación f, el autómata

Contenido de la pila	Resto de la entrada	Transición ejecutada
λ	*zazabzbz*	$(\iota, \lambda, \lambda; p, \#)$
#	*zazabzbz*	$(p, \lambda, \lambda; q, S)$
S#	*zazabzbz*	$(q, \lambda, S; q, zMNz)$
zMNz#	*zazabzbz*	$(q, z, z; q, \lambda)$
MNz#	*azabzbz*	$(q, \lambda, M; q, aMa)$
aMaNz#	*azabzbz*	$(q, a, a; q, \lambda)$
MaNz#	*zabzbz*	$(q, \lambda, M; q, z)$
zaNz#	*zabzbz*	$(q, z, z; q, \lambda)$
aNz#	*abzbz*	$(q, a, a; q, \lambda)$
Nz#	*bzbz*	$(q, \lambda, N; q, bNb)$
bNbz#	*bzbz*	$(q, b, b; q, \lambda)$
Nbz#	*zbz*	$(q, \lambda, N; q, z)$
zbz#	*zbz*	$(q, z, z; q, \lambda)$
bz#	*bz*	$(q, b, b; q, \lambda)$
z#	*z*	$(q, z, z; p, \lambda)$
#	λ	$(q, \lambda, \#; f, \lambda)$

Figura 2.10 Análisis completo de la cadena *zazabzbz* que efectúa el autómata de pila descrito en la figura 2.8

tiene un objetivo principal representado por $<\iota, \lambda, f>$, donde ι es el estado inicial de la máquina.

Ahora estamos listos para mostrar que los lenguajes aceptados por un autómata de pila son independientes del contexto.

TEOREMA 2.3

Para cada autómata de pila M, existe una gramática G independiente del contexto tal que $L(M) = L(G)$.

DEMOSTRACIÓN

Dado un autómata de pila M, nuestra tarea es producir una gramática G independiente del contexto que genere el lenguaje $L(M)$. Como resultado del teorema 2.1, podemos suponer que el autómata de pila M acepta cadenas únicamente cuando su pila está vacía. Una vez hecha esta observación, construimos G de tal manera que sus no terminales representen los objetivos de M, como se mencionó antes, y que sus reglas de reescritura representen refinamientos de objetivos mayores, presentados en términos de objetivos más pequeños.

Para ser más precisos, especificamos que los no terminales de G consisten en el símbolo de inicio S más todos los objetivos de M, es decir, todas las estructuras sintácticas de la forma $<p, x, q>$ donde

p y q son estados de M y x es λ o un símbolo de pila de M (así, G puede contener un gran número de no terminales incluso si M es un autómata pequeño). Los terminales de G son los símbolos del alfabeto de M.

Ahora estamos listos para introducir la primera colección de reglas de reescritura de G. Estas reglas se obtienen de la siguiente manera:

1. Para cada estado de aceptación f de M, forme la regla de reescritura $S \rightarrow <\iota, \lambda, f>$, donde ι es el estado inicial de M.

Las reglas de reescritura obtenidas en el paso 1 aseguran que cualquier derivación que utilice esta gramática comenzará sustituyendo el símbolo inicial de la gramática por un objetivo principal del autómata.

Se obtiene otra colección de reglas de reescritura con

2. Para cada estado p en M, forme la regla de reescritura $<p, \lambda, p> \rightarrow \lambda$.

Las reglas obtenidas en el paso 2 son reflejo de que puede eliminarse el objetivo de pasar de un estado a sí mismo sin modificar la pila.

Cada una de las demás reglas de reescritura de G se construye a partir de una transición en M, siguiendo el paso 3 o el paso 4 descritos ahora.

3. Para cada transición $(p, x, y; q, z)$ de M (donde y no es λ), genere una regla de reescritura $<p, y, r> \rightarrow x<q, z, r>$ para cada estado r de M.

Las reglas generadas por el paso 3 indican que el objetivo de pasar de un estado p a un estado r eliminando y de la pila puede lograrse si se pasa primero a un estado q mientras que se lee x de la entrada y se intercambia z por y en la pila (usando la transición $(p, x, y; q, z)$) y luego intentando pasar del estado q al estado r a la vez que se elimina z de la pila.

4. Para cada transición de la forma $(p, x, \lambda; q, z)$, genere todas las reglas de reescritura de la forma $<p, w, r> \rightarrow x <q, z, k> <k, w, r>$, donde w es un símbolo de pila o λ, mientras que k y r (que pueden ser iguales) son estados de M.

Las reglas de reescritura construidas en el paso 4 reflejan que el objetivo de pasar de un estado p a un estado r a la vez que se elimina w de la pila puede lograrse si primero se pasa al estado q mientras se lee x de la entrada y se inserta z en la pila (por medio de la transición $(p, x, \lambda; q, z)$) y luego se intenta pasar del estado q al estado r a través de un estado k, a la vez que se eliminan z y w de la pila.

Observe que las reglas de reescritura construidas en los pasos 1 a 4 forman una gramática independiente del contexto. Sólo resta mostrar que esta gramática genera el mismo lenguaje que acepta el autómata: debemos mostrar que este último puede aceptar cualquier cadena generada por la gramática y que la gramática puede generar cualquier cadena que acepte el autómata. Ambos enunciados serán verdaderos si demostramos la siguiente afirmación.

Aplicando reglas de la gramática, se puede reescribir un no terminal de la forma <p, α, q> *(donde* α *es* λ *o un símbolo de pila) como una cadena de terminales* w, *si y sólo si el autómata puede pasar de un estado* p *a un estado* q *siguiendo una ruta cuyo recorrido da como resultado que se elimine* α *de la pila y que se lea la cadena* w *de la cinta de la máquina.*

Comencemos por demostrar la parte "sólo si" de esta afirmación, aplicando la inducción en el número de pasos requeridos para reescribir $<p, \alpha, q>$ como w. Si se requiere sólo un paso, entonces el no terminal $<p, \alpha, q>$ debe ser en realidad $<p, \lambda, q>$ ya que ésta es la única forma de un no terminal que puede reescribirse como cadena terminal en un solo paso. A su vez, esto significa que w debe ser λ. Así, nuestra afirmación es verdadera para el caso básico, ya que el autómata siempre puede pasar de un estado p a p sin eliminar nada de la pila ni leer nada de la cinta.

Ahora, supongamos que la ruta deseada en el autómata existe para cualquier caso donde, en n o menos pasos, el no terminal que representa un objetivo puede reescribirse como una cadena de terminales, y suponga que en $n+1$ pasos puede reescribirse $<p, \alpha, q>$ como la cadena de terminales w. Observe que el primer paso de este proceso debe ser la aplicación de una regla de reescritura del paso 3 o del paso 4 anteriores. Supongamos que se obtiene del paso 3, con la transición $(p, w_1, \alpha; r, \beta)$. (El caso del paso 4 no es más que una leve generalización de este caso.) El primer paso del proceso de reescritura aparece como $<p, \alpha, q> \Rightarrow w_1 < r, \beta, q>$. Esto significa que el resto del proceso de reescritura convierte $<r, \beta, q>$ en una cadena w_2, tal que $w = w_1 w_2$, en sólo n pasos. Entonces, por nuestra hipótesis de inducción, el autómata puede pasar del estado r al estado q leyendo w_2 a la vez que elimina β de su pila. Si hacemos que la transición $(p, w_1, \alpha; r, \beta)$ anteceda a este cálculo, obtenemos un proceso que comienza en el estado p y pasa al estado q a la vez que lee de la cinta la cadena w y elimina α de la pila. Por lo tanto, podemos concluir que la parte "sólo si" de nuestra afirmación es verdadera.

Ahora, consideremos la parte "si" de nuestra afirmación. Una vez más aplicamos la inducción, pero esta vez sobre la longitud de la ruta de p a q. Si la longitud de esta ruta es cero, la ruta no requiere transiciones y q debe ser igual a p. Por lo tanto, basta con la regla de reescritura $<p, \alpha, p> \rightarrow \lambda$.

Luego suponemos que si el autómata puede pasar de cualquier estado r a un estado s, siguiendo una ruta que consiste en no más de n transiciones y que da como resultado la lectura de la cadena v de la cinta y la eliminación de α de la pila (donde α es λ o un símbolo de pila), entonces hay en la gramática reglas de reescritura que permiten reescribir $<p, \alpha, q>$ como w. Considere una ruta de $n+1$ pasos del estado p a q cuyo recorrido da como resultado la lectura de cinta de la cadena w y la eliminación de α de la pila (donde α es λ o un símbolo de pila). Supongamos que el primer paso de esta ruta es la ejecución de la transición $(p, w_1, \alpha; t, z)$, donde $\alpha \neq \lambda$ (los otros primeros pasos posibles se manejan de manera semejante). Entonces, la porción restante de la ruta debe pasar del estado t al estado q en sólo n transiciones, sin eliminar nada de la pila al leer de la cinta la cadena w_2, donde $w = w_1 w_2$. Sin embargo, por nuestra hipótesis de inducción, esto significa que deben existir en la gramática reglas de reescritura que permitan reescribir $< t, \lambda, q >$ como w_2. Además, la existencia de la transición $(p, w_1, \alpha; t, z)$ implica la existencia de la regla $<p, \alpha, q> \rightarrow w_1 < t, \lambda, q >$. Por lo tanto, el no terminal $<p, \alpha, q>$ puede reescribirse como w aplicando primero esta regla y luego reescribiendo $< t, \lambda, q >$ como w_2.

Concluimos que las partes "si" y "sólo si" de nuestra afirmación deben ser verdaderas, y por lo tanto la gramática independiente del contexto construida a partir de los pasos 1, 2, 3 y 4 debe generar el mismo lenguaje que acepta el autómata de pila M.

∎

De nuevo, un ejemplo debe ayudar a esclarecer varios de los puntos de la demostración anterior. Considere la construcción de una gramática independiente del contexto que acepte el lenguaje $\{cb^n c : n \in \mathbb{N}^+\}$ del autómata de pila de la figura 2.11. Nuestra primera observación es que en la gramática resultante existen numerosos no terminales, incluyendo el símbolo de inicio S más un no terminal de la forma $<p, x, q>$ para cada tripleta (p, x, q), donde x es λ o c (puesto que c es el único símbolo de pila), y p y q (que pueden ser iguales) son estados de la máquina. Además como se muestra en la figura 2.12 se forma un gran número de reglas de reescritura, que se presentan, junto con sus transiciones asociadas. Por último, en la figura 2.13 se muestra una derivación de la cadena $cbbc$. Aquí vemos que las reglas de reescritura asociadas a las transiciones que leen símbolos de la entrada de la máquina introducen estos mismos símbolos en la cadena que se deriva. Así, una vez que el proceso de derivación ha eliminado todos los no terminales, los símbolos en la cadena restante son exactamente los mismos que habría leído el cálculo correspondiente del autómata de pila.

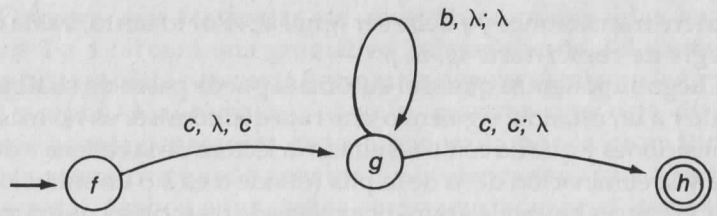


Figura 2.11 Diagrama de transiciones para un autómata de pila que acepta $\{cb^nc: n \in \mathbb{N}^+\}$

En resumen, contamos ahora con dos caracterizaciones para los lenguajes independientes del contexto: son los lenguajes aceptados por autómatas de pila, así como los lenguajes generados por gramáticas independientes del contexto.

Forma normal de Chomsky

Una de las ventajas de clasificar los lenguajes en función de su gramática es que estas clasificaciones proporcionan detalles con respecto a las estructuras de cadenas que pueden aparecer en los lenguajes correspondientes. Sin embargo, a primera vista parece que la flexibilidad que permiten las gramáticas independientes del contexto impone pocas restricciones a las posibles estructuras de cadenas que pueden encontrarse en los lenguajes independientes del contexto. Por esto, quizás le sorprenda saber que los lenguajes independientes del contexto sí tienen gramáticas cuyas reglas de reescritura se adhieren a formatos extremadamente rígidos. Por lo tanto, investigaremos con mayor detenimiento la estructura de las reglas de reescritura que se encuentran en las gramáticas independientes del contexto.

Comenzamos nuestra investigación considerando la necesidad de tener reglas λ en una gramática independiente del contexto. Si el lenguaje generado por la gramática contiene la cadena vacía, entonces debe aparecer cuando menos una regla λ en la gramática; de lo contrario, no habría manera de derivar la cadena vacía del símbolo inicial de la gramática. Sin embargo, podríamos preguntar hasta qué punto se requieren las reglas λ si el lenguaje que se genera no contiene la cadena vacía.

Para responder esta pregunta, démonos cuenta primero que la existencia de una regla λ puede permitir que más no terminales que el que aparece en la regla puedan reescribirse como la cadena vacía. Por ejemplo, si una gramática contiene las reglas $N \to \lambda$ y $M \to N$, entonces tanto M como N se podrían reescribir como λ, aunque la regla $M \to \lambda$ no se encuentre en la gramática. Para identificar el efecto de las reglas λ en una gramática independiente del

contexto, debemos aislar todos los no terminales que pueden reescribirse como la cadena vacía. Para esto, definimos un encuadernamiento λ de longitud n como la secuencia de reglas de la forma $N_n \to N_{n-1}, N_{n-1} \to N_{n-2}, \cdots, N_0 \to \lambda$, y definimos el no terminal N_n como el origen del encuadernamiento.

encadenamiento

Del paso 1

$S \to \langle f, \lambda, h\rangle$

Del paso 2

$\langle f, \lambda, f\rangle \to \lambda$
$\langle g, \lambda, g\rangle \to \lambda$
$\langle h, \lambda, h\rangle \to \lambda$

Del paso 3, transición $(g, c, c; h, \lambda)$

$\langle g, c, f\rangle \to c\langle h, \lambda, f\rangle$
$\langle g, c, g\rangle \to c\langle h, \lambda, g\rangle$
$\langle g, c, h\rangle \to c\langle h, \lambda, h\rangle$

Del paso 4, transición $(f, c, \lambda; g, c)$

$\langle f, \lambda, f\rangle \to c\ \langle g, c, f\rangle\ \langle f, \lambda, f\rangle$	$\langle f, c, f\rangle \to c\ \langle g, c, f\rangle\ \langle f, c, f\rangle$
$\langle f, \lambda, f\rangle \to c\ \langle g, c, g\rangle\ \langle g, \lambda, f\rangle$	$\langle f, c, f\rangle \to c\ \langle g, c, g\rangle\ \langle g, c, f\rangle$
$\langle f, \lambda, f\rangle \to c\ \langle g, c, h\rangle\ \langle h, \lambda, f\rangle$	$\langle f, c, f\rangle \to c\ \langle g, c, h\rangle\ \langle h, c, f\rangle$
$\langle f, \lambda, g\rangle \to c\ \langle g, c, f\rangle\ \langle f, \lambda, g\rangle$	$\langle f, c, g\rangle \to c\ \langle g, c, f\rangle\ \langle f, c, g\rangle$
$\langle f, \lambda, g\rangle \to c\ \langle g, c, g\rangle\ \langle g, \lambda, g\rangle$	$\langle f, c, g\rangle \to c\ \langle g, c, g\rangle\ \langle g, c, g\rangle$
$\langle f, \lambda, g\rangle \to c\ \langle g, c, h\rangle\ \langle h, \lambda, g\rangle$	$\langle f, c, g\rangle \to c\ \langle g, c, h\rangle\ \langle h, c, g\rangle$
$\langle f, \lambda, h\rangle \to c\ \langle g, c, f\rangle\ \langle f, \lambda, h\rangle$	$\langle f, c, h\rangle \to c\ \langle g, c, f\rangle\ \langle f, c, h\rangle$
$\langle f, \lambda, h\rangle \to c\ \langle g, c, g\rangle\ \langle g, \lambda, h\rangle$	$\langle f, c, h\rangle \to c\ \langle g, c, g\rangle\ \langle g, c, h\rangle$
$\langle f, \lambda, h\rangle \to c\ \langle g, c, h\rangle\ \langle h, \lambda, h\rangle$	$\langle f, c, h\rangle \to c\ \langle g, c, h\rangle\ \langle h, c, h\rangle$

Del paso 4, transición $(g, b, l; g, \lambda)$

$\langle g, \lambda, f\rangle \to b\ \langle g, \lambda, f\rangle\ \langle f, \lambda, f\rangle$	$\langle g, c, f\rangle \to b\ \langle g, \lambda, f\rangle\ \langle f, c, f\rangle$
$\langle g, \lambda, f\rangle \to b\ \langle g, \lambda, g\rangle\ \langle g, \lambda, f\rangle$	$\langle g, c, f\rangle \to b\ \langle g, \lambda, g\rangle\ \langle g, c, f\rangle$
$\langle g, \lambda, f\rangle \to b\ \langle g, \lambda, h\rangle\ \langle h, \lambda, f\rangle$	$\langle g, c, f\rangle \to b\ \langle g, \lambda, h\rangle\ \langle h, c, f\rangle$
$\langle g, \lambda, g\rangle \to b\ \langle g, \lambda, f\rangle\ \langle f, \lambda, g\rangle$	$\langle g, c, g\rangle \to b\ \langle g, \lambda, f\rangle\ \langle f, c, g\rangle$
$\langle g, \lambda, g\rangle \to b\ \langle g, \lambda, g\rangle\ \langle g, \lambda, g\rangle$	$\langle g, c, g\rangle \to b\ \langle g, \lambda, g\rangle\ \langle g, c, g\rangle$
$\langle g, \lambda, g\rangle \to b\ \langle g, \lambda, h\rangle\ \langle h, \lambda, g\rangle$	$\langle g, c, g\rangle \to b\ \langle g, \lambda, h\rangle\ \langle h, c, g\rangle$
$\langle g, \lambda, h\rangle \to b\ \langle g, \lambda, f\rangle\ \langle f, \lambda, h\rangle$	$\langle g, c, h\rangle \to b\ \langle g, \lambda, f\rangle\ \langle f, c, h\rangle$
$\langle g, \lambda, h\rangle \to b\ \langle g, \lambda, g\rangle\ \langle g, \lambda, h\rangle$	$\langle g, c, h\rangle \to b\ \langle g, \lambda, g\rangle\ \langle g, c, h\rangle$
$\langle g, \lambda, h\rangle \to b\ \langle g, \lambda, h\rangle\ \langle h, \lambda, h\rangle$	$\langle g, c, h\rangle \to b\ \langle g, \lambda, h\rangle\ \langle h, c, h\rangle$

Figura 2.12 Reglas de reescritura obtenidas a partir del autómata de pila de la figura 2.11

$$
\begin{aligned}
S &\Rightarrow \langle f, \lambda, h\rangle \\
&\Rightarrow c\langle g, c, h\rangle\langle h, \lambda, h\rangle \\
&\Rightarrow cb\langle g, \lambda, g\rangle\langle g, c, h\rangle\langle h, \lambda, h\rangle \\
&\Rightarrow cb\langle g, c, h\rangle\langle h, \lambda, h\rangle \\
&\Rightarrow cbb\langle g, \lambda, g\rangle\langle g, c, h\rangle\langle h, \lambda, h\rangle \\
&\Rightarrow cbb\langle g, c, h\rangle\langle h, \lambda, h\rangle \\
&\Rightarrow cbbc\langle h, \lambda, h\rangle\langle h, \lambda, h\rangle \\
&\Rightarrow cbbc\langle h, \lambda, h\rangle \\
&\Rightarrow cbbc
\end{aligned}
$$

Figura 2.13 Derivación de *cbbc* utilizando las reglas de reescritura de la figura 2.12

Ahora, si G es una gramática independiente del contexto que no genera la cadena vacía, definimos U_0 como el conjunto de los no terminales que aparecen como origen de los encadenamientos λ de longitud cero, o sea, los no terminales que aparecen del lado izquierdo de las reglas λ. A este conjunto le añadimos los orígenes de todos los encadenamientos λ de longitud uno para formar otro conjunto, U_1. Luego, a U_1 le agregamos los orígenes de todos los encadenamientos λ de longitud dos para obtener un conjunto llamado U_2, etcétera (si G fuera la gramática en la figura 2.14a, entonces U_0 sería $\{Q\}$ y U_1 sería $\{P, Q\}$).

Puesto que en una gramática sólo hay un número finito de no terminales, debe existir un punto en el cual este proceso deje de introducir no terminales adicionales. Al llegar a este punto hemos recopilado todos los no terminales de G que pueden reescribirse como la cadena vacía; este conjunto se representa con U (observe que, como G no genera la cadena vacía, U no puede contener el símbolo inicial de G).

Ahora, para cada regla de reescritura de la forma $N \rightarrow w$, donde w es una cadena de terminales y no terminales, agregamos a G todas las reglas de la forma $N \rightarrow w'$, donde w' es cualquier cadena *no vacía* obtenida al eliminar de w una o más ocurrencias de no terminales en U. (Una vez más, haciendo referencia a la figura 2.14a, el conjunto U sería $\{P, Q\}$ por lo que agregamos a la gramática las reglas siguientes:

$$
\left.\begin{array}{l} S \rightarrow zPzz \\ S \rightarrow zzQz \\ S \rightarrow zzz \end{array}\right\} \textit{a partir de } S \rightarrow zPzQz
$$

$$
\begin{array}{ll}
P \rightarrow xx & \textit{a partir de } P \rightarrow xPx \\
Q \rightarrow yy & \textit{a partir de } Q \rightarrow yPy
\end{array}
$$

a.
$$
\begin{array}{l}
S \rightarrow zPzQz \\
P \rightarrow xPx \\
P \rightarrow Q \\
Q \rightarrow yPy \\
Q \rightarrow \lambda
\end{array}
$$

b.
$$
\begin{array}{l}
S \rightarrow zPzQz \\
S \rightarrow zzQz \\
S \rightarrow zPzz \\
S \rightarrow zzz \\
P \rightarrow zPx \quad xPx \\
P \rightarrow xx \\
P \rightarrow Q \\
Q \rightarrow yPz \quad y \\
Q \rightarrow yy
\end{array}
$$

Figura 2.14 Una gramática independiente del contexto (que no genera la cadena vacía) y otra gramática que genera el mismo lenguaje sin utilizar reglas λ

Observe que no agregamos la regla $P \rightarrow \lambda$ que se obtendría de la regla $P \rightarrow Q$. (Véase Fig. 2.14b.)

Una vez que agregamos estas nuevas reglas a la gramática, ya no necesitamos las reglas λ. Suponga que la derivación de una cadena, empleando la gramática original, requiere la aplicación de una regla λ, y sea N el origen del encadenamientos λ más largo que aparece en la derivación que termina con esta regla λ. Entonces, la ocurrencia de N debe introducirse en la derivación aplicando alguna regla de la forma $M \rightarrow w_1Nw_2$, donde w_1 y w_2 son cadenas de terminales y no terminales, una de las cuales debe ser no vacía (se escogió N como el origen del encadenamiento λ que da fin a la regla λ; además, N no es el símbolo inicial, ya que $N \in U$ y $S \in U$). Esto significa que la regla $M \rightarrow w_1w_2$ es una de las reglas que hemos agregado a la gramática. Así, podemos eliminar de la derivación el empleo de la regla λ si aplicamos la regla $M \rightarrow w_1w_2$ en vez de $M \rightarrow w_1Nw_2$. De esta manera podemos eliminar cualquier utilización de reglas λ en una derivación. Además, podemos deshacernos de todas las reglas λ de la gramática sin reducir sus poderes generativos.

Como ejemplo, considere la derivación de la figura 2.15a, basada en la gramática de la figura 2.14a. El último paso de la derivación utiliza la regla λ, $Q \rightarrow \lambda$. El origen del encadenamientos λ que condujo al uso de esta regla es el no terminal Q introducido en el primer paso de la derivación. Por lo tanto, podemos cambiar este paso utilizando la nueva regla $S \rightarrow zPzz$ para obtener la derivación de la figura 2.15b. Esta derivación usa la regla λ, $Q \rightarrow \lambda$, en su último paso. El no terminal P introducido en el segundo paso es el origen del encadenamiento λ que lleva a esta regla. Por ello, alteramos este paso para aprovechar la nueva regla $P \rightarrow xx$, obteniendo así la derivación de la figura 2.15c que sólo usa reglas de la gramática modificada de la figura 2.14b.

Por último, observamos que la gramática que se desprende de este proceso de eliminación de reglas λ no puede generár cadenas que no generaba la gramática original. Después de todo, es posible simular cualquiera de las reglas añadidas por medio de cortas secuencias de reglas de la gramática original. Por

a. $S \Rightarrow zPzQz$
$\Rightarrow zxPxzQz$
$\Rightarrow zxQxzQz$
$\Rightarrow zxxzQz$
$\Rightarrow zxxzz$

b. $S \Rightarrow zPzz$
$\Rightarrow zxPxzz$
$\Rightarrow zxQxzz$
$\Rightarrow zxxzz$

c. $S \Rightarrow zPzz$
$\Rightarrow zxxzz$

Figura 2.15 Modificación de una derivación basada en la gramática de la figura 2.14a para obtener una derivación basada en la gramática de la figura 2.14b

lo tanto, concluimos que *cualquier lenguaje independiente del contexto que no contiene la cadena vacía se puede generar por medio de una gramática independiente del contexto que no tenga reglas* λ.

Tomando como punto de partida esta conclusión, podemos demostrar el siguiente teorema.

TEOREMA 2.4
Si L es un lenguaje independiente del contexto que no contiene la cadena vacía, entonces existe una gramática G independiente del contexto tal que $L(G) = L$ y el lado derecho de cualquier regla de reescritura en G consiste en un solo terminal o exactamente dos no terminales.

DEMOSTRACIÓN
Sea L un lenguaje independiente del contexto que no contiene la cadena vacía. Ya sabemos que una gramática G independiente del contexto que no contiene reglas λ puede generar L. Nuestro enfoque es modificar esta gramática para que se adhiera a las restricciones del teorema.

Para cada terminal x en G, introducimos un nuevo no terminal único X y la regla de reescritura $X \to x$, y luego reemplazamos las ocurrencias del terminal x en todas las demás reglas de G con X. Esto

produce una gramática G' independiente del contexto en la cual el lado derecho de cada regla de reescritura es un solo terminal o una cadena de no terminales. Además, $L(G') = L(G)$.

Ahora reemplazamos cada regla de G' de la forma

$$N \to N_1 N_2 \cdots N_n$$

donde $n > 2$, con la colección de reglas

$$\begin{aligned} N &\to N_1 R_1 \\ R_1 &\to N_2 R_2 \\ &\vdots \\ R_{n-1} &\to N_{n-1} N_n \end{aligned}$$

donde cada R_k es un no terminal único que no aparece en ningún otro lugar de la gramática. Obviamente, esta modificación no cambia el lenguaje generado por la gramática.

Al llegar a esta etapa tenemos una gramática G' independiente del contexto que genera L, para la cual el lado derecho de cada regla de reescritura es un solo terminal, dos no terminales o un solo no terminal. Entonces, lo que queda por hacer es eliminar las reglas de la última forma. Para esto, consideramos cada secuencia de reglas de reescritura de la forma $N_n \to N_{n-1}, N_{n-1} \to N_{n-2}, \cdots, N_2 \to N_1$, introducimos la regla $N_n \to x$ si $N_1 \to x$ es una regla de G', e introducimos la regla $N_n \to AB$ si $N_1 \to AB$ está en G'. Una vez que se han agregado estas reglas, podemos eliminar aquellas donde el lado derecho contiene un solo no terminal, sin reducir los poderes generativos de la gramática. Después de todo, cualquier derivación que usa las reglas de la forma $M \to N$ puede modificarse para que utilice en cambio las reglas que acabamos de introducir. Así, la gramática resultante genera L y satisface las restricciones del teorema.

∎

Para hacer más claros los pasos de la demostración anterior, considere cómo afectarían a la gramática de la figura 2.16a, con símbolo inicial S. El primer paso sería introducir los nuevos no terminales X, Y y Z y convertir la gramática en la gramática G' que se presenta en la figura 2.16 b. A continuación, la regla $S \to ZMZ$ se reemplazaría por el par de reglas $S \to ZR_1$ y $R_1 \to MZ$, mientras que $M \to YMY$ se reemplazaría por $M \to YP_1$ y $P_1 \to MY$, para obtener la gramática de la figura 2.16 c. Finalmente, la secuencia $N \to X$ y la secuencia $M \to N$ y $N \to X$ darían origen a las reglas $N \to x$ y $M \to x$, produciendo así la gramática de la figura 2.16 d.

a. $S \to zMz$
$M \to N$
$M \to yMy$
$N \to x$

b. $S \to ZMZ$
$M \to N$
$M \to YMY$
$N \to X$
$X \to x$
$Y \to y$
$Z \to z$

c. $S \to ZR_1$
$R_1 \to MZ$
$M \to N$
$M \to YP_1$
$P_1 \to MY$
$N \to X$
$X \to x$
$Y \to y$
$Z \to z$

d. $S \to ZR_1$
$R_1 \to MZ$
$M \to x$
$M \to YP_1$
$P_1 \to MY$
$N \to x$
$X \to x$
$Y \to y$
$Z \to z$

Figura 2.16 Aplicación del proceso descrito en la demostración del teorema 2.4

Se dice que una gramática cuyas reglas de reescritura se adhieren a las restricciones del teorema 2.4 tiene la **forma normal de Chomsky** (llamada así en honor de N Chomsky). Por ejemplo, la gramática

$$S \to XM$$
$$M \to SY$$
$$X \to x$$
$$Y \to y$$

cuyo símbolo inicial es S tiene la forma normal de Chomsky, mientras que

$$S \to xSy$$
$$S \to xy$$

que genera el mismo lenguaje, no la tiene.

En resumen, el teorema 2.4 indica que cualquier lenguaje independiente del contexto que no contenga la cadena vacía puede ser generado por una gramática independiente del contexto que tenga la forma normal de Chomsky. Aunque se limita a un subconjunto de los lenguajes independientes del contexto, esta caracterización sigue siendo bastante general. Por ejemplo, la mayoría de los lenguajes en ambientes de aplicación, como los lenguajes de programación, no contienen la cadena vacía; incluso para aquellos lenguajes que sí contienen la cadena vacía, la caracterización tiene su mérito. De hecho,

Gramática para $L - \{\lambda\}$ con símbolo inicial S	Gramática para L con símbolo inicial S'
	$S' \to \lambda$ } Nuevas reglas
	$S' \to MN$ } Nuevas reglas
$S \to MN$	$S \to MN$
$N \to MS$	$N \to MS$
$N \to x$	$N \to x$
$M \to x$	$M \to x$

Figura 2.17 Modificación de una gramática con forma normal de Chomsky que genera $L - \{\lambda\}$, para obtener una nueva gramática que genere L

si un lenguaje L independiente del contexto contiene λ, podemos encontrar una gramática independiente del contexto que "casi" tiene la forma normal de Chomsky pero que aún genera L. Para lograr esto, primero encontramos una gramática G independiente del contexto con forma normal de Chomsky que genere $L - \{\lambda\}$. Luego, modificamos G agregando un nuevo no terminal S' que se convierte en el símbolo inicial de la nueva gramática; después añadimos la regla $S' \to \lambda$ (para que la nueva gramática genere λ); y, por último, para cada regla de G cuyo lado izquierdo consiste en el antiguo símbolo inicial de G, agregamos una nueva regla en donde éste se sustituye con el nuevo no terminal S' en ese lado izquierdo (véase Fig. 2.17). Es evidente que esta gramática modificada generará las cadenas del lenguaje L, y sólo esas cadenas (al introducir el nuevo símbolo inicial S' se elimina la posibilidad de que la regla λ interactúe con otras reglas de la gramática). Por consiguiente, incluso los lenguajes independientes del contexto que contienen la cadena vacía pueden ser generados por gramáticas que casi tienen la forma normal de Chomsky.

Ejercicios

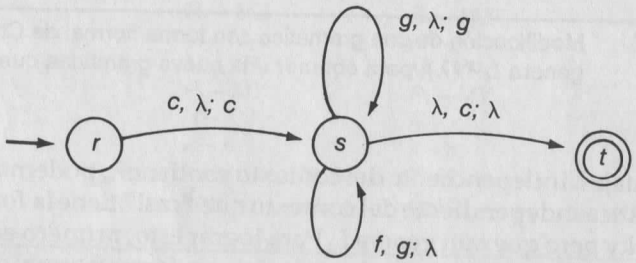
1. Muestre que toda cadena derivada por la izquierda de una gramática independiente del contexto puede derivarse también por la derecha.

2. Se dice que una gramática es ambigua cuando permite más de un árbol de análisis sintáctico para una sola cadena (véase Ap. C, Sec. C.1). Demuestre que la gramática que se presenta a continuación es ambigua, mostrando que la cadena *ictictses* tiene derivaciones que producen distintos árboles de análisis sintáctico.

$$S \to ictS$$
$$S \to ictSeS$$
$$S \to s$$

(Quizás ya haya observado antes este problema si asocia las siguientes palabras con los símbolos terminales de la gramática anterior: i = if, c = condición, t = then, e = else y s = enunciado.)

3. Utilice el procedimiento descrito en el teorema 2.3 para construir una gramática independiente del contexto que genere el lenguaje aceptado por el autómata de pila cuyo diagrama de transiciones es el siguiente.



4. Demuestre que la unión de dos lenguajes independientes del contexto también es independiente del contexto mostrando cómo pueden combinarse dos gramáticas independientes del contexto de los lenguajes originales para formar una gramática independiente del contexto que genere la unión.

5. Convierta la siguiente gramática, con símbolo inicial S, en una gramática con forma normal de Chomsky que genere el mismo lenguaje.

$$S \to xSy$$
$$S \to wNz$$
$$N \to S$$
$$N \to \lambda$$

2.3 LÍMITES DE LOS AUTÓMATAS DE PILA

Hasta ahora hemos caracterizado los lenguajes independientes del contexto como aquellos generados por gramáticas independientes del contexto y como aquellos aceptados por los autómatas de pila. Sin embargo, no hemos considerado el alcance de estos lenguajes: no nos hemos preguntado si existen lenguajes que no son independientes del contexto. Además, los autómatas de pila que hemos considerado hasta ahora son no deterministas y, ya que nuestro plan es desarrollar herramientas de diseño de compiladores basadas en las propiedades de los autómatas de pila, es necesario comprender el papel del determinismo en los autómatas de pila. Éstos serán los aspectos que trataremos en esta sección.

Alcance de los lenguajes independientes del contexto

Primero presentaremos un lenguaje que no es independiente del contexto. Para esto utilizaremos el teorema siguiente, conocido como **lema de bombeo** ya que muestra cómo en algunos lenguajes independientes del contexto pueden producirse cadenas "bombeando" ("ampliando") porciones de otras cadenas.

TEOREMA 2.5
Si L es un lenguaje independiente del contexto que contiene un número infinito de cadenas, entonces debe existir en L una cadena que tenga la forma $svuwt$, donde s, v, u, w y t son subcadenas, por lo menos una de v y w es no vacía, y sv^nuw^nt está en L para cada $n \in \mathbb{N}^+$.

DEMOSTRACIÓN
Sea L un lenguaje independiente del contexto que contiene un número infinito de cadenas, y sea G una gramática independiente del contexto tal que $L(G) = L$. Sea m el número máximo de símbolos (terminales y no terminales) que se encuentran en el lado derecho de cualquier regla de reescritura en G; es decir, m es la longitud del lado derecho más largo de las reglas de reescritura de G. Entonces, cada nodo de un árbol de análisis sintáctico basado en G puede tener cuando mucho m hijos. A su vez, cualquier árbol de análisis sintáctico de profundidad d puede producir una cadena de longitud máxima m^d (donde la profundidad del árbol es el número de aristas en la ruta más larga de la raíz a una hoja).

Sea ahora j el número de símbolos no terminales de G, y elija una cadena de L con longitud mayor que m^j. Entonces, el árbol de análisis sintáctico T para dicha cadena debe tener una profundidad mayor que j. Esto indica que existe una ruta en T, que va de la raíz a una hoja, la cual contiene más de j no terminales. Por consiguiente, algún no terminal N debe aparecer por lo menos dos veces en la ruta. Consideremos el subárbol de T cuyo nodo raíz es la ocurrencia más alta de N en esta ruta, y en el cual la siguiente ocurrencia de N es una hoja (como lo indica la región sombreada de la Fig. 2.18). En otras palabras, consideramos el subárbol de T cuya raíz es la ocurrencia más alta de N y luego descartamos todo lo que queda debajo de la siguiente ocurrencia de N.

Este subárbol indica que el patrón vNw se deriva de la primera ocurrencia de N, donde v y w son concatenaciones de las hojas del subárbol a la izquierda y a la derecha de la segunda N, respectivamente (véase de nuevo la Fig. 2.18). Podemos suponer que v o w debe ser no vacía, pues de lo contrario podríamos eliminar la región sombreada de la figura 2.18 para obtener el árbol de la figura 2.19a, recortando así la ruta elegida. Sin embargo si pudieran recortarse así todas las rutas con

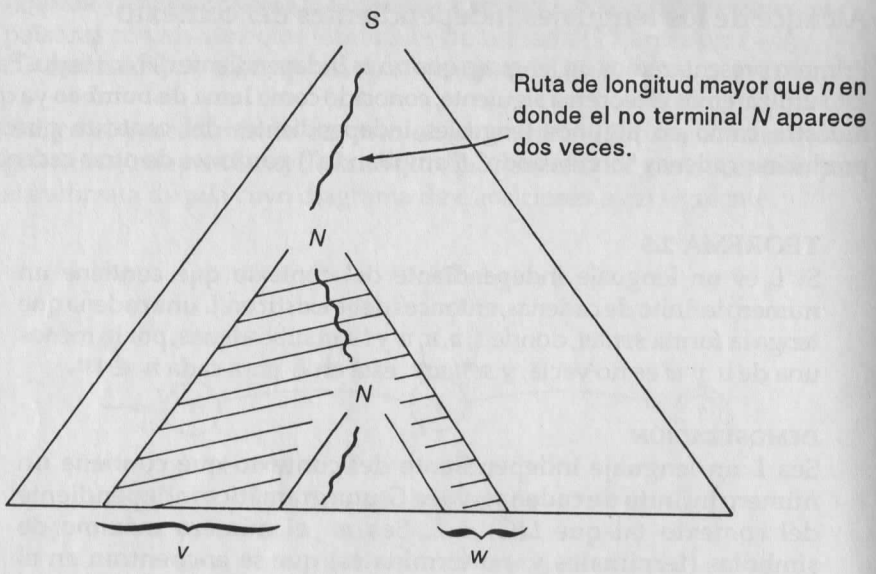


Figura 2.18 Representación gráfica del árbol de derivación T

longitud mayor que j, produciríamos un árbol de análisis sintáctico para la cadena elegida que tuviera una profundidad menor que j, lo cual sería una contradicción.

Observe que pueden construirse otros árboles de análisis sintáctico repitiendo un número arbitrario de veces las copias del subárbol seleccionado, como se muestra en la figura 2.19b. Cada uno de estos árboles de análisis sintáctico representa una cadena que la gramática G puede generar. Por lo tanto, G genera cadenas que contienen estructuras de la forma v^iNw^i para cada entero positivo i. A su vez, debe existir una cadena en L de la forma $svuwt$ donde sv^nuw^nt se encuentre en L para cada $n \in \mathbb{N}^+$, como se afirma en el teorema.

∎

Una consecuencia del teorema 2.5 es que el lenguaje $\{x^ny^nz^n: n \in \mathbb{N}^+\}$ no es independiente del contexto. De hecho, este lenguaje contiene una cantidad infinita de cadenas, pero no existe en el lenguaje cadena alguna que tenga un segmento con posibilidades de repetirse según lo establecido en el teorema y aún así producir cadenas en el lenguaje (si cada uno de los dos segmentos repetidos consiste solo en las x, sólo en las y o sólo en las z, entonces el resultado no contendrá el mismo número para cada símbolo. Además, si repetimos un segmento con más de un tipo de símbolo, entonces el resultado tendrá las y antes de las x o z antes de las y).

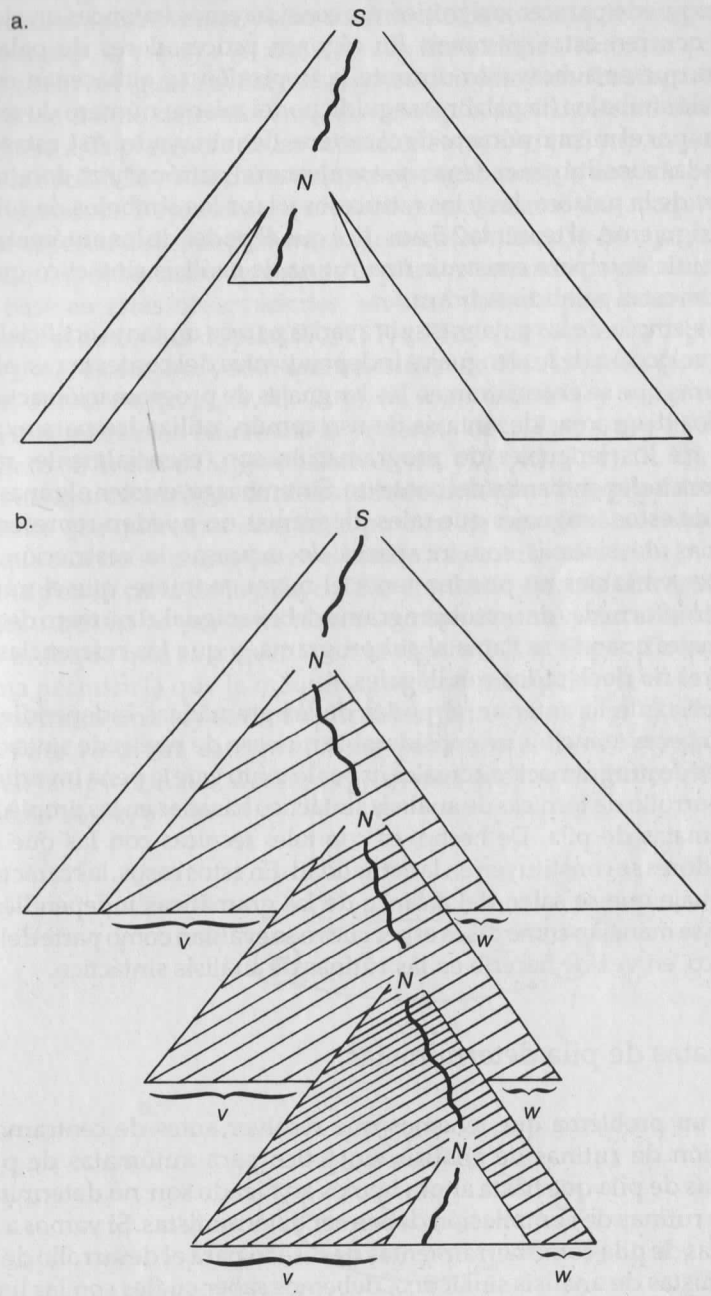


Figura 2.19 Árboles sintácticos que pueden construirse modificando el árbol de la figura 2.17

El hecho de que el lenguaje $\{x^n y^n z^n : n \in \mathbb{N}^+\}$ no sea independiente del contexto puede parecer insignificante; consideremos entonces un ejemplo en el cual ocurren estos patrones. En algunos procesadores de palabras, las palabras que se subrayarán durante la impresión se almacenan como una cadena de símbolos (la palabra) seguida por el mismo número de retrocesos, seguidos por el mismo número de caracteres de subrayado. Así, estas palabras subrayadas constituyen cadenas que se ajustan al patrón $x^n y^n z^n$, donde las x son las letras de la palabra, las y los retrocesos y las z los símbolos de subrayado. Por consiguiente, el teorema 2.5 nos dice que el poder de los autómatas de pila sería insuficiente para construir una rutina de análisis sintáctico que pueda reconocer estas palabras subrayadas.

Si el ejemplo de las palabras subrayadas parece un tanto artificial, se debe a que la colección de los lenguajes independientes del contexto casi abarca las estructuras que se encuentran en los lenguajes de programación actuales. De hecho, los diagramas de sintaxis de uso común, utilizados para expresar la sintaxis de los lenguajes de programación son, esencialmente, reglas de reescritura independientes del contexto. Sin embargo, existen algunas características de estos lenguajes que tales diagramas no pueden representar. Los diagramas de sintaxis son incapaces de expresar la restricción de que diferentes variables no pueden tener el mismo nombre, que el número de parámetros formales de un subprograma debe ser igual al número de parámetros actuales cuando se llama al subprograma, y que las referencias a identificadores no declarados son ilegales.

No obstante lo anterior, el poder de las gramáticas independientes del contexto permite incluir un considerable número de reglas de sintaxis de los lenguajes de programación actuales, y por lo tanto vale la pena invertir tiempo en el desarrollo de técnicas de análisis sintáctico basadas en las propiedades de los autómatas de pila. De hecho, es con tales técnicas con las que muchos compiladores se constituyen en la actualidad. En estos casos, las características del lenguaje que se salen del alcance de las gramáticas independientes del contexto se manejan como casos especiales o se evalúan como parte del análisis semántico, en vez de hacerlo en las rutinas de análisis sintáctico.

Autómatas de pila deterministas

Persiste un problema que tenemos que resolver antes de centrarnos en la producción de rutinas de análisis sintáctico para autómatas de pila. Los autómatas de pila que hasta ahora hemos analizado son no deterministas, y nuestras rutinas de compilación deben ser deterministas. Si vamos a utilizar autómatas de pila como herramientas de diseño para el desarrollo de rutinas deterministas de análisis sintáctico, debemos saber cuáles son las limitaciones, si existen, que pueden surgir al requerir un comportamiento determinista. Nuestro primer paso en esa dirección es presentar el concepto de autómatas de pila deterministas.

De manera general, un **autómata de pila determinista** es un autómata de pila en el cual es aplicable una, y sólo una, transición en cualquier instante. Esto implica que si $(p, x, y; q, z)$ y $(p, x, y; r, w)$ son transiciones, entonces q debe ser igual a r y z debe ser igual a w. Sin embargo, esta condición no basta para excluir la posibilidad del no determinismo. Por ejemplo, si el siguiente símbolo de entrada fuera x, la presencia de las transiciones $(p, \lambda, y; q, z)$ y $(p, x, y; q, z)$ nos ofrecería la opción de pasar del estado p a otro ignorando la entrada o salir del estado p leyéndola. Se presentaría un problema similar si una máquina con y en la parte superior de la pila pudiera elegir entre $(p, \lambda, \lambda; q, z)$ y $(p, x, y; q, z)$. ¿Debe salir la máquina del estado p ignorando la pila o extrayendo de ella un símbolo?

Con base en estas observaciones, un autómata de pila determinista se define como el autómata de pila $(S, \Sigma, \Gamma, T, \iota, F)$ tal que para cada tripleta (p, x, y) en $S \times \Sigma \times \Gamma$, existe una y sólo una transición en T de la forma $(p, u, v; q, z)$, donde (u, v) está en $\{(x, y), (x, \lambda), (\lambda, y), (\lambda, \lambda)\}$, q está en S y z está en Γ. Por ejemplo, esta restricción eliminaría la presencia de $(p, \lambda, y; q, z)$ y $(p, x, \lambda; r, s)$, ya que tanto (x, λ) como (λ, y) se hallan en $\{(x, y), (x, \lambda), (\lambda, y), (\lambda, \lambda)\}$.

Para subrayar las sutilezas del desarrollo de un autómata de pila determinista, supongamos que queremos diseñar la porción de una de estas máquinas de manera que cuando se encuentre en el estado p ejecute la transición $(p, x, y; q, \lambda)$ si hay una y en la cima de la pila, o de lo contrario ejecuta la transición $(p, x, \lambda; q, \lambda)$. Aquí no se presenta incertidumbre alguna, pero sí la hay en el sencillo enfoque que se presenta en la figura 2.20a. Específicamente, el diagrama permitiría que la máquina tuviera una opción si en la cima de la pila existiera una y, y serían aplicables tanto $(p, x, y; q, \lambda)$ como $(p, x, \lambda; q, \lambda)$. Para resolver este problema pasamos a la estrategia que se ilustra en la figura 2.20b, donde suponemos que los símbolos de pila de la máquina son x, y y #.

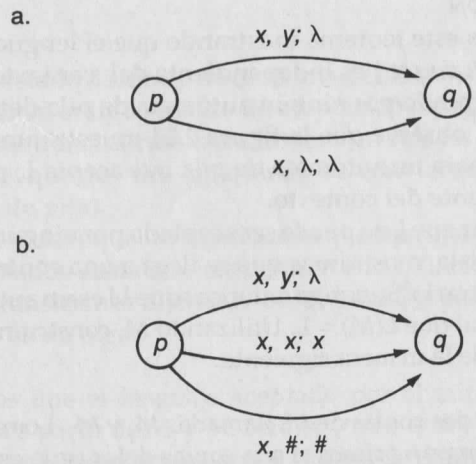


Figura 2.20 Porción del diagrama de transiciones para un autómata de pila que presenta no determinismo y su contrapartida determinista

Allí hemos introducido opciones explícitas para la máquina en caso de que el símbolo en la cima de la pila no fuera y. En estos casos se indica a la máquina que extraiga un símbolo de la pila y luego lo inserte de nuevo. Por supuesto, este enfoque requiere que la pila no se encuentre vacía cuando la máquina se halle en el estado p, pues de lo contrario ninguna de las transiciones sería aplicable. Es por esto que hemos introducido el símbolo # que se supone está en el fondo de la pila al iniciar cualquier cálculo, y que se conserva allí hasta el final.

Al igual que en el capítulo anterior, nuestro uso del término determinista implica que el sistema está completamente definido. Sin embargo, las afirmaciones que estamos a punto de efectuar con respecto a los autómatas de pila deterministas serán válidas, sin importar si los autómatas en cuestión se encuentran o no completamente definidos. Además, si se obliga a definir por completo a los autómatas, se obtienen diagramas más bien saturados. Como vimos en el caso de los autómatas finitos, la inclusión de los arcos para todos los casos es una cuestión más de paciencia que de contenido (compare la Fig. 1.5 con la 1.11). Entonces, al dibujar diagramas para autómatas de pila deterministas, con frecuencia representaremos sólo aquellas transiciones pertinentes para la tarea en cuestión, entendiéndose que si es necesario puede completarse el diagrama parcial.

Nuestra siguiente tarea es decidir si el requisito del determinismo reduce o no el poder de un autómata de pila. Por desgracia para nosotros, así es, como se muestra en el teorema siguiente.

TEOREMA 2.6

Existe un lenguaje independiente del contexto que no es el lenguaje aceptado por ningún autómata de pila determinista.

DEMOSTRACIÓN

Demostramos este teorema mostrando que el lenguaje $L = \{x^n y^n : n \in \mathbb{N}^+\} \cup \{x^n y^{2n} : n \cup \mathbb{N}^+\}$ es independiente del contexto pero que no puede ser aceptado por ningún autómata de pila determinista. En primer lugar, observe que la figura 2.21 muestra un diagrama de transiciones para un autómata de pila que acepta L; por lo tanto, L es independiente del contexto.

Para probar que L no puede ser aceptado por ningún autómata de pila determinista, mostramos que se llega a una contradicción si se supone lo contrario. Suponga entonces que M es un autómata de pila determinista tal que $L(M) = L$. Utilizando M, construimos otro autómata de pila de la manera siguiente:

1. Construya dos copias de M, llamadas M_1 y M_2. Los estados de M_1 y M_2 se llamarán primos si son copias del mismo estado en M.
2. Elimine la característica de aceptación de los estados de aceptación de M_1 y la característica de inicio del estado inicial de M_2.

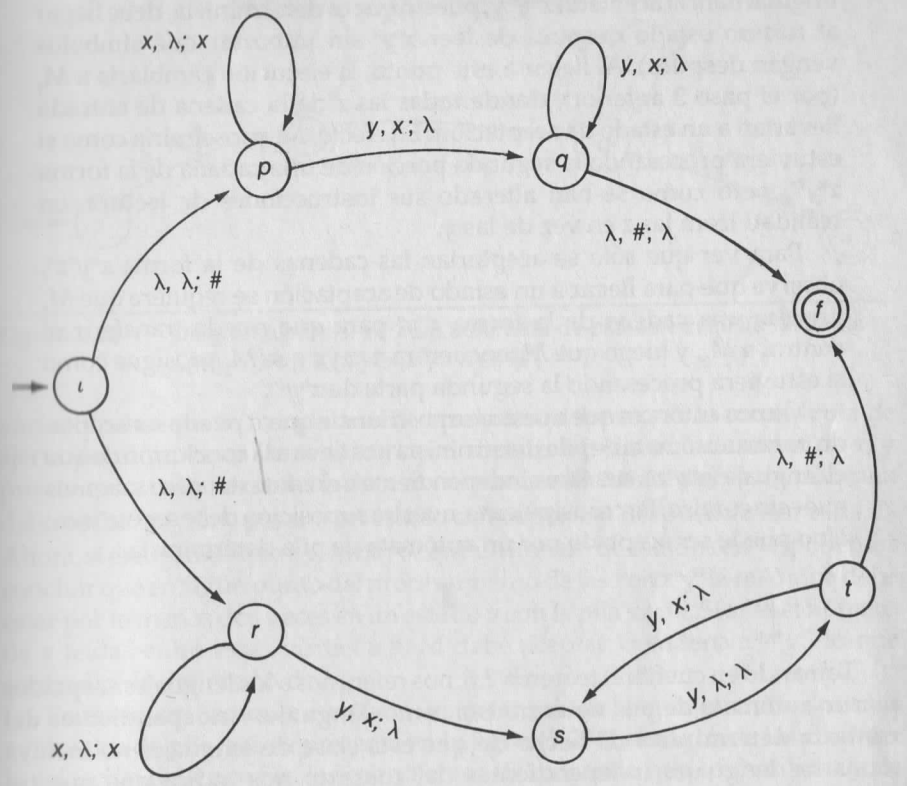


Figura 2.21 Diagrama de transiciones para un autómata de pila M para el cual $L(M) = \{x^n y^n : n \in \mathbb{N}^+\} \cup \{x^n y^{2n} : n \in \mathbb{N}^+\}$

3. Cambie el estado destino p de cada una de las transiciones que se originan en un antiguo estado de aceptación de M_1 al primo de p en M_2. (Estas transiciones alteradas forman enlaces entre M_1 y M_2, de manera que las dos máquinas se convierten en un solo autómata de pila).
4. Modifique todas aquellas transiciones que lean una y de la entrada y cuyos estados destino se encuentren en M_2 (incluyendo quizá, aquellas transiciones alteradas en el paso 3), para que lean el símbolo z en su lugar.

Afirmamos que el lenguaje aceptado por el autómata de pila construido así a partir de M_1 y M_2 será $\{x^n y^n z^n : n \in \mathbb{N}^+\}$. De hecho, si a esta máquina se le proporcionara una cadena de entrada de la forma $x^n y^n z^n$, tendría que llegar a uno de los antiguos estados de aceptación de M_1 después de leer las x y las y pero antes de leer alguna z (la M_1

original habría aceptado $x^n y^n$ y, puesto que es determinista, debe llegar al mismo estado después de leer $x^n y^n$ sin importar qué símbolos vengan después). Al llegar a este punto, la ejecución cambiaría a M_2 (por el paso 3 anterior), donde todas las z de la cadena de entrada llevarían a un estado de aceptación. En efecto, M_2 proseguiría como si estuviera procesando la segunda porción de una cadena de la forma $x^n y^{2n}$, pero como se han alterado sus instrucciones de lectura, en realidad leerá las z en vez de las y.

Para ver que sólo se aceptarían las cadenas de la forma $x^n y^n z^n$, observe que para llegar a un estado de aceptación se requiere que M_1 procese una cadena de la forma $x^n y^n$ para que pueda transferir el control a M_2, y luego que M_2 encuentra n z, ya que M_2 prosigue como si estuviera procesando la segunda parte de $x^n y^{2n}$.

Vemos entonces que nuestra suposición de que L puede ser aceptado por un autómata de pila determinista nos lleva a la conclusión de que el lenguaje $\{x^n y^n z^n : \mathbb{N} \in \mathbb{N}\}$ es independiente del contexto; pero sabemos que esto es falso. Por consiguiente, nuestra suposición debe ser errónea: L no puede ser aceptado por un autómata de pila determinista.

∎

Tomando en cuenta el teorema 2.6, nos referimos a los lenguajes aceptados por un autómata de pila determinista como **lenguajes independientes del contexto determinista**. El hecho de que esta clase de lenguajes no incluya todos los lenguajes independientes del contexto nos indica que nuestro objetivo de construir rutinas deterministas de análisis sintáctico y basadas en autómatas de pila no se alcanzará para todos de los lenguajes independientes del contexto, sino únicamente en el caso, más reducido, de los lenguajes independientes del contexto determinista.

De hecho, el panorama es aún más desalentador. Los autómatas de pila deterministas que consideramos aceptan cadenas sin que necesariamente tengan que vaciar sus pilas y, como se mencionó antes, es probable que este modelo dé origen a segmentos de programa que saturen la memoria de un computador con los residuos de cálculos anteriores. Por desgracia, este problema es más serio para los autómatas de pila deterministas que para los autómatas de pila ordinarios, ya que no podemos modificar cada autómata de pila determinista para que vacíe su pila antes de llegar al estado de aceptación.

Para convencernos de esta situación, considere el lenguaje L obtenido de la unión de $\{x^n : n \in \mathbb{N}\}$ y $\{x^n y^n : n \in \mathbb{N}\}$, que es aceptado por el autómata de pila determinista representado en el diagrama parcial de la figura 2.22, y que por lo tanto es un lenguaje independiente del contexto determinista. Afirmamos que L no puede ser aceptado por un autómata de pila determinista al cual se le requiera que vacíe su pila antes de llegar a un estado de aceptación. De hecho, si M fuera dicha máquina y recibiera una entrada de la forma $x^n y^n$, después de leer cada x tendría que llegar a un estado de aceptación con la pila

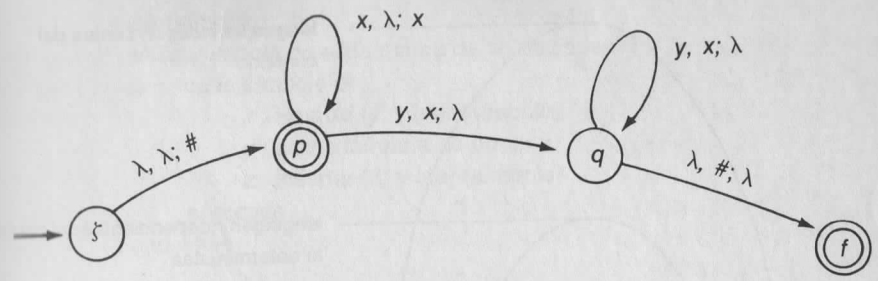


Figura 2.22 Diagrama parcial para un autómata de pila determinista M donde $L(M) = \{x^n : n \in \mathbb{N}^+\} \cup \{x^n y^n : n \in \mathbb{N}^+\}$

vacía. (Puesto que la máquina es determinista, debe seguir la misma ruta de ejecución al procesar las x en $x^n y^n$ que al procesar cualquier cadena de x, y puesto que cualquier cadena de x es en sí una cadena aceptable, la máquina debe vaciar su pila y pasar a un estado de aceptación después de leer cada x). Ahora, si escogemos que n sea mayor que el número de estados en M, podemos concluir que en algún punto del procesamiento de las x en $x^n y^n$ la máquina debe estar por lo menos dos veces en un estado p con la pila vacía. Si m es el número de x leídas entre estas visitas a p, M debe aceptar la cadena $x^{n+m} y^n$, lo que contradice la definición de M.

En resumen, hemos encontrado una jerarquía de lenguajes asociados a los autómatas de pila que se presenta en la figura 2.23. La mayor de las clases es la colección de lenguajes independientes del contexto que los autómatas de pila generales aceptan. Dentro de esta clase se encuentran propiamente incluidos los lenguajes independientes del contexto deterministas aceptados por los autómatas de pila deterministas (aquellos que no tienen el requisito de vaciar sus pilas). Luego, propiamente contenidos en la clase de los lenguajes independientes del contexto deterministas, se hallan los lenguajes que pueden ser aceptados por los autómatas de pila deterministas que vacían sus pilas antes de aceptar una cadena de entrada.

Principio de preanálisis

Al llegar a este punto es probable que comience a desvanecerse nuestra esperanza de desarrollar segmentos de programa utilizables para el análisis sintáctico de una amplia clase de lenguajes. Parece que cualquier técnica que se desarrolle estará restringida a la menor clase de lenguajes de la figura 2.23. Por fortuna, esta situación mejora si consideramos de nuevo la forma en que los autómatas aceptan sus cadenas de entrada. Recuerde que en la sección 2.1 vimos que, para aceptar una cadena, un autómata de pila debe cumplir con su criterio de aceptación después de leer la cadena pero sin desplazar más allá su cabeza de lectura por la cinta. Esto significa que un autómata de pila debe

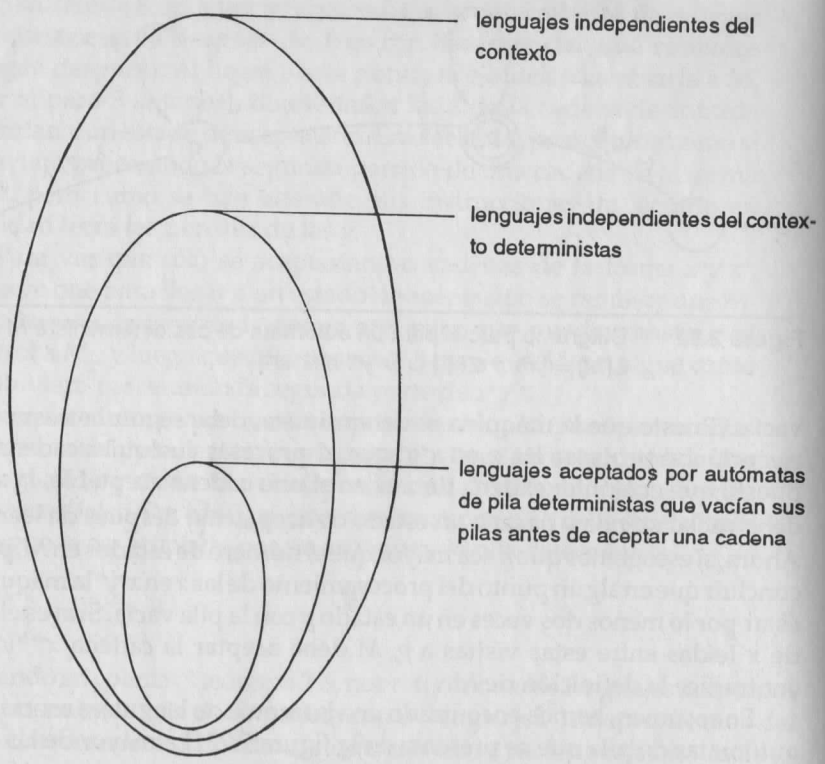


Figura 2.23 Clasificación de los lenguajes independientes del contexto

llegar a un estado de aceptación sin realmente leer la marca de fin de cadena. El resultado de esto es lo que podríamos considerar como un proceso de aceptación pasivo en el cual la máquina debe "caer" en un estado de aceptación sin que se lo indique una marca de fin de cadena. Por el contrario, si imaginamos un sistema mediante el cual la máquina pudiera observar el siguiente símbolo de la cinta (sin tener que avanzar su cabeza de lectura), podría detectar la marca de fin de cadena próxima y ejecutar actividades especiales "de cierre" que de otra manera no ejecutaría.

Nos referiremos a la técnica de observar los símbolos siguientes sin leerlos como **principio de preanálisis**. Su aplicación se ejemplifica con la estructura del ciclo while del segmento de programa de la figura 2.24. En esencia, este segmento observa el siguiente símbolo y emplea la información obtenida para decidir si procesa o no el símbolo dentro de la estructura while. En las distintas opciones del enunciado case, la rutina finalmente decide si consume el símbolo y se prepara para procesar otro. Así, la variable símbolo es en realidad un área de almacenamiento temporal, o *buffer*, para el símbo-


```
leer(símbolo);
while (símbolo no es la marca de fin de cadena)
    case símbolo of
        x : insertar (y) y leer (símbolo)
        y : salir a la rutina de error
        z : insertar (#) y leer (símbolo)
    end case
end while
```

Figura 2.24 Ciclo ***while*** típico

lo siguiente. Para tomar una decisión, se puede considerar un símbolo allí almacenado, sin que tenga que consumirse (o leerse oficialmente) hasta tomar la decisión de procesarlo. De modos específico, esta rutina no consumirá la marca de fin de cadena sino que permanecerá en el buffer tras terminar la estructura while, donde estará disponible como entrada para la siguiente rutina. Después de todo, en un compilador real es posible que la marca de fin de cadena sea el primer símbolo de la siguiente estructura que haya que analizar.

Vemos entonces que la aplicación del principio de preanálisis es una técnica de programación común. Estamos a punto de ver también que al aplicar este principio durante la conversión de autómatas de pila a segmentos de programa podemos construir programas que superen el no determinismo de algunos autómatas de pila. Por lo tanto, la teoría de autómatas de pila proporciona la base para construir rutinas de análisis sintáctico destinadas a una amplia gama de lenguajes independientes del contexto. Cómo logra esto es el tema de las dos secciones siguientes.

Ejercicios

1. Aplique el teorema 2.5 para demostrar que el lenguaje $\{x^m y^n x^m y^n x^m y^n : m, n \in \mathbb{N}^+\}$ no es independiente del contexto.

2. Proporcione ejemplos de los siguientes lenguajes:
   a. Un lenguaje que no es independiente del contexto.
   b. Un lenguaje independiente del contexto pero no determinista.
   c. Un lenguaje que es independiente del contexto determinista pero que no es aceptado por un autómata de pila determinista que tiene que vaciar su pila.
   d. Un lenguaje que es aceptado por un autómata de pila determinista que tiene que vaciar su pila pero que no es un lenguaje regular.

3. Dibuje la porción pertinente de un diagrama de transiciones para un autómata de pila determinista que pasará del estado p al q leyendo una x, extrayendo una y e insertando una z, si la cima de la pila contiene una y, o pasará al estado q leyendo una x, sin extraer nada e insertando una z si la cima de la pila no contiene una y.

4. Identifique las situaciones que darían origen a incertidumbres en la ejecución de un autómata de pila que contuviera las transiciones $(p, \lambda, y; q, z)$ y $(p, x, \lambda; r, w)$.

2.4 ANALIZADORES SINTÁCTICOS *LL(k)*

Ahora es el momento de considerar cómo pueden desarrollarse las rutinas de análisis sintáctico a partir de los autómatas de pila. Tradicionalmente, este problema surge cuando se describe un lenguaje en función de reglas de reescritura gramaticales. Después de ello, se desarrollan las rutinas de análisis sintáctico para el lenguaje empleando la teoría de los autómatas de pila como herramienta de desarrollo. Éste será el contexto para nuestro análisis.

Proceso de análisis sintáctico *LL*

Una técnica para traducir gramáticas independientes del contexto a autómatas de pila es seguir el proceso descrito en la demostración del teorema 2.2. Esta construcción produce un autómata de pila que analiza su cadena de entrada marcando antes el fondo de la pila e insertando en la pila el símbolo inicial de la gramática. Luego repite los tres pasos siguientes, según resulte aplicable.

1. Si la cima de la pila contiene un no terminal de la gramática, reemplace ese no terminal de acuerdo con una de las reglas de reescritura de la gramática.
2. Si la cima de la pila contiene un terminal, elimínelo de la pila a la vez que lee de la entrada el mismo terminal. Si el símbolo en la entrada no equivale al símbolo de la pila, se declara que la entrada es una cadena ilegal.
3. Si aparece en la superficie de la pila la marca de fondo de pila, elimínela y declare que es aceptable la porción de la cadena de entrada procesada hasta el momento.

Recuerde que este proceso analiza la sintaxis de la cadena de entrada produciendo una derivación por la izquierda conforme lee la cadena de izquierda a derecha. Por consiguiente, actuará de la misma manera un segmento de programa que se obtenga traduciendo directamente el autómata a enunciados de programa. Los analizadores sintácticos desarrollados de esta manera se conocen como **analizadores sintácticos *LL***. La primera *L* denota que el analizador lee su entrada de izquierda a derecha (*Left to right*, en inglés); la segunda *L* indica que el objetivo del analizador sintáctico es producir una derivación por la izquierda (*Leftmost derivation*, en inglés).

La figura 2.25b muestra un diagrama de transiciones construido a partir de la gramática de la figura 2.25a usando el proceso descrito en la demostración del teorema 2.2 (se trata de la gramática independiente del contexto que vimos en la figura 2.7, la cual genera cadenas de la forma $x^n y^n$ para enteros no negativos n). Para producir una rutina de análisis sintáctico a partir de esta gramática, podemos convertir las transiciones de la máquina directamente a enunciados de programa, obteniendo así la rutina de la figura 2.26, donde hemos empleado la estructura while tradicional para simular las actividades disponibles para la máquina cuando se encuentra en el estado q (mientras el símbolo en la cima de la pila no sea la marca de fondo de pila, la máquina permanecerá en el estado q).

Obviamente, el segmento de la figura 2.26 no es un producto terminado. En primer lugar, no hemos considerado los errores ocasionados por entradas inválidas. Por ejemplo, si aparece una x en la superficie de la pila durante el ciclo while, nuestra rutina supone que el siguiente símbolo de la entrada es una x y ejecuta la instrucción "leer una x de la cadena de entrada". En realidad, el siguiente símbolo puede ser distinto de x, por lo que nuestra rutina debe tomar en cuenta esta posibilidad. Debido a ello debemos ampliar la instrucción "leer una x de la cadena de entrada" para obtener el par de instrucciones que se muestra a continuación:

```
leer(símbolo);
if símbolo no es x then salir a la rutina de error;
```

Otro problema de menor magnitud que existe en la rutina de la figura 2.26 es que puede llegar al estado f con una pila vacía sin haber leído toda la cadena de entrada. Por ejemplo, la cadena xyx no existe en el lenguaje descrito por la gramática original, pero nuestra rutina nunca se percatará de esto. En cambio, leerá la entrada hasta xy, donde se detendrá suponiendo que la cadena de entrada es válida. Este problema se puede corregir agregando las instrucciones

```
leer(símbolo);
if símbolo no es la marca de fin de cadena then salir a la rutina de error
```

al final de la rutina.

Sin embargo, en la rutina existe un problema que es más severo que los anteriores: en algunos casos las órdenes presentan opciones sin resolver. De hecho, si el estado actual es q y el símbolo en la cima de la pila es S, la rutina ofrece la opción de reemplazar la S con xSy o simplemente eliminar la S de la pila. Este problema es fundamentalmente distinto de los temas ya mencionados, ya que implica la selección de instrucciones en vez de una mera clarificación o refinamiento de los detalles de las instrucciones.

a. $S \rightarrow xSy$
$S \rightarrow \lambda$

b.

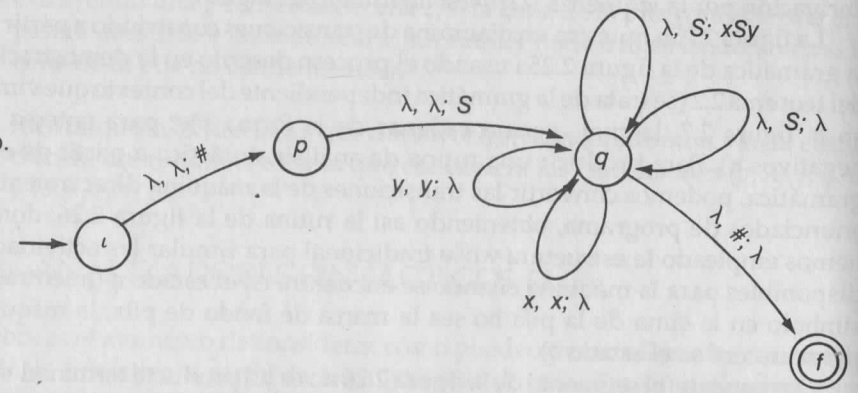


Figura 2.25 Gramática independiente del contexto y diagrama de transiciones asociado para un autómata de pila

```
Estado : = ι;
insertar (#);
Estado : = p;
insertar (S);
Estado : = q ;
while cima-de-la-pila ≠ # do
    case cima-de-la-pila of
        S: extraer (S) e insertar (xSy), o extraer (S);
        x: extraer (x), leer una x de la entrada;
        y: extraer (y), leer una y de la entrada;
    end case
end while;
extraer (#);
Estado : = f
```

Figura 2.26 Segmento de "programa" obtenido al traducir a enunciados el diagrama de la figura 2.25

Aplicación del principio de preanálisis

Por fortuna, el no determinismo de nuestra rutina se puede resolver utilizando el principio de preanálisis presentado en la sección anterior. Si


```
Estado : = ι;
insertar (#);
Estado : = p;
insertar (S);
Estado : = q;
leer (Símbolo);
while cima-de-la-pila ≠ # do
    case cima-de-la-pila of
        S : if Símbolo ≠ x then extraer (S)
                else extraer (S), insertar (xSy);
        x : if Símbolo no es x then salir a la rutina de error
                else extraer (x), leer (Símbolo);
        y : if Símbolo no es y then salir a la rutina de error
                else extraer (y), leer (Símbolo);
    end case
end while;
extraer (#)
if Símbolo no es la marca de fin de cadena then salir a la rutina de error;
Estado : = f
```

Figura 2.27 Rutina de análisis sintáctico basada en la gramática de la figura 2.25

encontramos una x al observar el siguiente símbolo de la entrada, entonces debemos reemplazar la S por la cadena xSy; de lo contrario, debemos sustituirla por la cadena vacía. (Si insertamos xSy en la pila sabiendo que el siguiente símbolo de la entrada no es una x, estamos condenados al fracaso. Una vez que insertamos un símbolo terminal en la pila, para poder extraerlo debe ser igual a un símbolo de la entrada. Si colocamos xSy en la pila cuando observamos en la entrada un símbolo distinto de x, el símbolo de la entrada no será igual a la x en la cima de la pila, y nunca podríamos vaciar la pila y pasar al estado de aceptación.)

Con base en lo anterior, podemos convertir el diagrama no determinista de la figura 2.25 en el segmento de programa determinista que se muestra en la figura 2.27. Aquí hemos utilizado la variable Símbolo como un almacenamiento temporal para el siguiente símbolo de la entrada. A partir de este almacenamiento temporal es posible interrogar cuál es el símbolo cada vez que se tengan que tomar decisiones, pero sin procesarlo antes de que sea necesario. Sobre todo, observe que la rutina no consume la marca de fin de cadena, aunque la detecte; permanece en el almacenamiento temporal, donde puede usarse como el primer símbolo de la siguiente estructura que será analizada por el sistema de análisis sintáctico.

El problema que surge en el ejemplo anterior es un fenómeno común en los analizadores sintácticos *LL*, pues se origina cuando la gramática propone más de una forma de reescribir el mismo no terminal. Estas opciones múltiples son esenciales para las gramáticas que deben generar lenguajes que contienen más de una cadena (una gramática independiente del contexto que sólo ofrece una manera de reescribir cada no terminal sólo puede generar una cadena). Por

$S \rightarrow xSz$
$S \rightarrow xyTyz$
$T \rightarrow \lambda$

Figura 2.28 Gramática independiente del contexto que requiere una analizador sintáctico $LL(2)$

esto, la actividad básica de los analizadores sintácticos LL es predecir cuál de las distintas reglas de reescritura es la que debe usarse para procesar los símbolos de entrada restantes. Por consiguiente, a estos analizadores se les llama **analizadores sintácticos predictivos.**

Si se aplica el principio de preanálisis, pueden resolverse muchas de las incertidumbres que se presentan en los analizadores sintácticos predictivos. Sin embargo, incluso en aquellos casos donde el principio de preanálisis es la técnica indicada, es posible que su aplicación no sea tan directa como en nuestro ejemplo. Si fuéramos a construir un analizador sintáctico a partir de la gramática de la figura 2.28, encontraríamos que la decisión con respecto a la reescritura de S no puede resolverse con sólo observar el siguiente símbolo de entrada (el conocimiento de que el siguiente símbolo es x no nos indica que debemos aplicar $S \rightarrow xSy$ en vez de $s \rightarrow xyTyz$). En cambio, la decisión depende de los dos símbolos siguientes. Entonces, para desarrollar una rutina determinista de análisis sintáctico debemos contar con espacio de almacenamiento temporal para dos símbolos de entrada.

Como resultado, existe una jerarquía de analizadores sintácticos LL cuya característica distintiva es el número de símbolos de entrada que comprende su sistema de preanálisis. Estos analizadores se llaman **analizadores sintácticos** $LL(k)$, donde k es un entero que indica el número de símbolos preanalizados por el analizador sintáctico. El ejemplo de la figura 2.27 es un analizador sintáctico $LL(1)$, mientras que un analizador sintáctico basado en la gramática de la figura 2.28 sería un analizador sintáctico $LL(2)$.

Usted quizá piense (correctamente) que la carga de predicciones que se coloca sobre los analizadores sintácticos $LL(k)$ restringe a fin de cuentas, los lenguajes que pueden analizarse. En efecto, existen lenguajes dentro de los límites de los analizadores sintácticos de pila que no puede reconocer ningún analizador sintáctico $LL(k)$, sin importar la magnitud de k. Un ejemplo es el lenguaje $\{x^n: n \in \mathbb{N}\} \cup \{x^n y^n: n \in \mathbb{N}\}$, que ya sabemos es determinista independiente del contexto. Por intuición, cualquier gramática independiente del contexto que genere este lenguaje debe permitir que se reescriba un no terminal con una cadena que contiene sólo x o una cadena que contiene una combinación equilibrada de x y y. Esto quiere decir que existirán por lo menos dos reglas para reescribir este no terminal. A su vez, cualquier analizador sintáctico $LL(k)$ se enfrentará al problema de decidir cuál de estas reglas será la que se aplique cuando surja el no terminal en la superficie de la pila. Por desgracia, independientemente de la magnitud de k, existen cadenas en el lenguaje en las cuales no es posible detectar la existencia de o la ausencia de y posteriores sin antes observar más de k x de

preanálisis. Entonces, cualquier analizador sintáctico $LL(k)$ será incapaz de manejar las decisiones necesarias para analizar la sintaxis de este lenguaje.

La existencia de un lenguaje determinista independiente del contexto que no puede ser analizado por un analizador sintáctico $LL(k)$ sugiere que pueden existir analizadores sintácticos basados en la teoría de los autómatas de pila más poderosos que estos analizadores sintácticos predictivos, hipótesis que nos conduce a la siguiente sección. No obstante, al incrementar el poder aumenta también la complejidad, por lo que la sencillez de los analizadores sintácticos $LL(k)$ hace de ellos una opción popular cuando son capaces de manejar el lenguaje que se considera. Por ahora, dejamos a un lado nuestra búsqueda de potencia y consideramos cómo pueden simplificarse los analizadores sintácticos $LL(k)$ utilizando tablas de análisis sintáctico.

Tablas de análisis sintáctico LL

Una tabla de análisis sintáctico para un analizador sintáctico $LL(1)$ es una matriz bidimensional. Las filas se etiquetan con los no terminales de la gramática sobre la cual se basa el analizador sintáctico. Las columnas se etiquetan con los terminales de la gramática más una columna adicional FDC (que representa la marca de fin de cadena). El elemento (m, n) de la tabla indica la acción que debe seguirse cuando el no terminal m aparece en la cima de la pila y el símbolo de preanálisis es n. Si en esta situación debiera reemplazarse el no terminal m siguiendo una regla de reescritura, el lado derecho de la regla aparecería en la posición (m, n). De lo contrario, la casilla contiene un indicador de error. Por ejemplo, en la figura 2.29 se presenta una tabla de análisis sintáctico para la gramática de la figura 2.5.

Una vez que se ha construido la tabla de análisis sintáctico, la tarea de escribir un segmento de programa que efectúe el análisis sintáctico del lenguaje es bastante sencilla. Lo único que tiene que hacer el segmento es insertar en la pila el símbolo inicial de la gramática y luego, mientras no se vacíe la pila, igualar los terminales de la cima de la pila con los de la entrada o reemplazar los no terminales de la cima de la pila siguiendo las directrices de la tabla de análisis sintáctico. Por ejemplo, la rutina de la figura 2.30 es un analizador sintáctico $LL(1)$ que utiliza la tabla de la figura 2.29.

	a	*b*	*z*	FDC
S	error	error	*zMNz*	error
m	*aMa*	error	*z*	error
N	error	*bNb*	*z*	error

Figura 2.29 Tabla de análisis sintáctico $LL(1)$ para la gramática de la figura 2.5

```
insertar (S);
leer (Símbolo);
while pila no está vacía do
    case cima-de-la-pila of
        terminal: if cima-de-la-pila = Símbolo
                then extraer de la pila y leer (Símbolo)
                else salir a la rutina de error;
        no terminal: if tabla [cima-de-la-pila, Símbolo] ≠ error
                then reemplazar cima-de-la-pila por tabla
                [cima-de-la-pila, Símbolo]
                else salir a la rutina de error;
    end case
end while
if Símbolo no es la marca de fin de cadena then salir a la rutina de error
```

Figura 2.30 Rutina genérica de análisis sintáctico *LL*(1)

	x	*y*	*FDC*
S	*xSy*	λ	λ

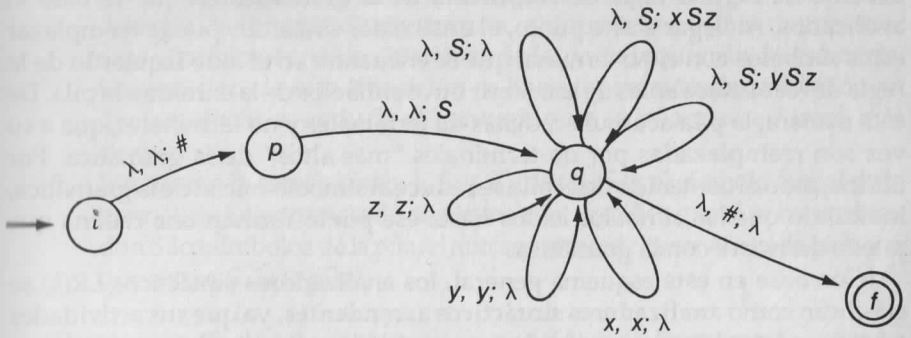
Figura 2.31 Tabla de análisis sintáctico *LL*(1) para la gramática de la figura 2.7

Otra ventaja de utilizar tablas de análisis sintáctico es que permiten normalizar el algoritmo de análisis. Cualquier analizador sintáctico *LL*(1) puede emplear el mismo algoritmo; si se desea obtener un analizador sintáctico para otro lenguaje, basta con sustituir la tabla de análisis sintáctico por una nueva. Para subrayar este punto, concluimos observando que la combinación del segmento de programa de la figura 2.30 con la tabla de análisis sintáctico de la figura 2.31 produce un analizador sintáctico para el lenguaje generado por la gramática de la figura 2.7.

Ejercicios

1. Reescriba el segmento de programa de la figura 2.30 utilizando una estructura repeat ... until en vez de una estructura while.

2. Traduzca el diagrama de transiciones que se muestra a continuación directamente a un segmento de programa que analice el lenguaje en cuestión. ¿Cuáles son las incertidumbres que deben resolverse para obtener una rutina determinista?



3. Diseñe una tabla de análisis sintáctico *LL*(1) para la gramática siguiente.

$$S \rightarrow xSz$$
$$S \rightarrow ySz$$
$$S \rightarrow \lambda$$

4. ¿Cuántos símbolos de preanálisis requeriría un analizador sintáctico *LL* al analizar la sintaxis de cadenas basadas en la gramática siguiente? Diseñe una tabla de análisis sintáctico correspondiente.

$$S \rightarrow xSy$$
$$S \rightarrow xy$$

2.5 ANALIZADORES SINTÁCTICOS *LR*(*k*)

En la sección anterior mencionamos que la naturaleza predictiva de los analizadores sintácticos *LL*(*k*) restringe la clase de lenguajes que pueden manejar estos analizadores. En esta sección presentamos una clase de analizadores sintácticos que evitan muchos de los problemas relacionados con sus homólogos predictivos. Estos analizadores se conocen como **analizadores sintácticos *LR*(*k*)**, ya que leen su entrada de izquierda a derecha (*Left to right*, en inglés) mientras construyen una derivación por la derecha (*Right derivation*, en inglés) de sus cadenas de entrada utilizando un sistema de preanálisis que comprende *k* símbolos.

Proceso de análisis sintáctico *LR*

En términos generales, un analizador sintáctico *LR* transfiere símbolos de su entrada a la pila hasta que los símbolos superiores de la pila sean iguales al lado

derecho de alguna regla de reescritura de la gramática en que se basa el analizador. Al llegar a este punto, el analizador sintáctico puede reemplazar estos símbolos con el no terminal que se encuentra en el lado izquierdo de la regla de reescritura antes de transferir otros símbolos de la entrada a la pila. De esta manera, la pila acumula cadenas de terminales y no terminales, que a su vez son reemplazadas por no terminales "más altos" de la gramática. Por último, todo el contenido de la pila se reduce al símbolo inicial de la gramática, indicando que los símbolos leídos hasta ese punto forman una cadena que puede derivarse con la gramática.

Con base en este esquema general, los analizadores sintácticos *LR(k)* se clasifican como **analizadores sintácticos ascendentes**, ya que sus actividades corresponden a la construcción de ocurrencias de no terminales a partir de sus componentes, hasta generar el símbolo inicial de la gramática. En comparación, los analizadores sintácticos *LL(k)* se conocen como **analizadores sintácticos descendentes** pues comienzan con el símbolo inicial en la pila y repetidamente dividen los no terminales de la pila en sus componentes, hasta generar una cadena de símbolos equivalente a la cadena de entrada.

Demos marcha atrás por un momento y desarrollemos los detalles específicos de los analizadores sintácticos *LR(k)*. Recuerde que un analizador *LL(k)* se basa en un autómata de pila construido a partir de una gramática independiente del contexto; esta construcción se basa en el proceso descrito en la demostración del teorema 2.2. De manera similar, un analizador sintáctico *LR(k)* se basa en un autómata de pila construido a partir de una gramática independiente del contexto, con la excepción de que el autómata se construye de la manera descrita en los cinco pasos siguientes.

1. Establezca cuatro estados: uno inicial llamado i, un estado de aceptación llamado f y otros dos estados llamados p y q.
2. Introduzca las transiciones $(\iota, \lambda, \lambda; p, \#)$ y $(q, \lambda, \#; f, \lambda)$, donde suponemos que # es un símbolo que no se presenta en esta gramática.
3. Para cada símbolo terminal x en la gramática, introduzca la transición $(p, x, \lambda; p, x)$. Estas transiciones permiten que el autómata transfiera a la pila los símbolos de entrada mientras se encuentra en el estado p. La ejecución de esta transición se llama **operación de desplazamiento** ya que su efecto es desplazar un símbolo de la cadena de entrada a la pila.
4. Para cada regla de reescritura $N \rightarrow w$ (donde w representa una cadena de uno o más símbolos) que exista en la gramática, introduzca la transición $(p, \lambda, w; p, N)$. (Aquí permitimos que una transición elimine más de un símbolo de la pila. Para ser más precisos, la transición $(p, \mathrm{l}, xy; p, z)$ es una forma abreviada de transición $(p, \lambda, y; p_1 \lambda)$ seguida por $(p_1, \lambda, x; p, z)$, donde p_1 es un estado al que no puede llegar ninguna otra transición. Así, para ejecutar la transición $(p, \lambda, xy; p, z)$, un autómata debe tener una y en la cima de la pila con una x inmediatamente debajo). La presencia de estas transiciones significa que si los símbolos de la porción superior de la pila concuerdan con los del lado derecho de la regla de reescritura, entonces es posible reemplazar esos símbolos con el no terminal de la parte izquierda de esa regla. La ejecución de esta transición se llama **operación de reducción** ya que su efecto es reducir el contenido de la pila a una forma más simple.
5. Introduzca la transición $(p, \lambda, S; q, \lambda)$, donde S es el símbolo inicial de la gramática. La presencia de esta transición significa que si se han reducido a S los símbolos de la pila, el autómata puede pasar al estado q a la vez que extrae S de la pila.

Como ejemplo de esta construcción, considere el diagrama de la figura 2.32, el cual se construyó a partir de la gramática de la figura 2.5. Un autómata de pila basado en este diagrama de transiciones analizaría la cadena *zazabzbz* tal como se muestra en el resumen de la figura 2.33. Comenzaría por marcar el fondo de la pila con el símbolo # y pasar al estado p, teniendo la cadena *zazabzbz* en la entrada, como lo indica la primera fila de la figura. A partir de esta configuración, el autómata desplazaría la primera z de la entrada a la pila a través de la transición $(p, z, \lambda; p, z)$, para llegar a la configuración que se presenta en la segunda fila de la figura 2.33. En esta etapa, el autómata tendría la opción de ejecutar $(p, \lambda, z; p, M)$, $(p, \lambda, z; p, N)$ o $(p, a, \lambda; p, a)$. (Este autómata es no determinista.) En nuestro ejemplo, las primeras dos opciones representan decisiones incorrectas; deben tomarse si hay que analizar la z en la cima de la pila como un refinamiento del no terminal M o N, respectivamente. La tercera opción es la correcta en nuestro caso, pues hay que leer más de la cadena antes de ejecutar la operación de reducción.

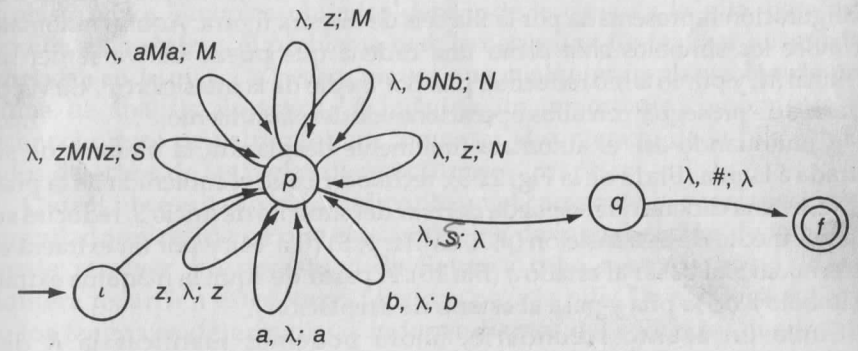


Figura 2.32 Otro diagrama de transiciones para un autómata de pila basado en la gramática de la figura 2.5

(Números de fila)	Estado actual	Contenido de la pila	Resto de la entrada
1	*p*	#	*zazabzbz*
2	*p*	#*z*	*azabzbz*
3	*p*	#*za*	*zabzbz*
4	*p*	#*zaz*	*abzbz*
5	*p*	#*zaM*	*abzbz*
6	*p*	#*zaMa*	*bzbz*
7	*p*	#*zM*	*bzbz*
8	*p*	#*zMb*	*zbz*
9	*p*	#*zMbz*	*bz*
10	*p*	#*zMbN*	*bz*
11	*p*	#*zMbNb*	*z*
12	*p*	#*zMN*	*z*
13	*p*	#*zMNz*	
14	*p*	#*S*	
15	*q*	#	
16	*f*	vacío	

Figura 2.33 Análisis completo de la cadena *zazabzbz* efectuado por el autómata de la figura 2.32

Puesto que el objetivo es mostrar cómo puede aceptar el autómata la cadena *zazabzbz*, seguimos esta última opción. De hecho, el autómata debe continuar con la ejecución de operaciones de desplazamiento hasta que llegue a la configuración representada por la cuarta fila de la figura 2.33. En esta etapa, la acción correcta es ejecutar la transición $(p, \lambda, z; p, M)$, reconociendo que la z en la cima de la pila se deriva de una ocurrencia del no terminal *M*. Después de esta transición, el autómata debe ejecutar la operación de desplazamiento $(p, a, \lambda; p, a)$ y llegar a la configuración representada por la fila seis de nuestra figura. Aquí el autómata reconoce los símbolos *aMa* como una cadena que puede derivarse del no terminal *M*, y por lo tanto reduce su pila por medio de la transición $(p, \lambda, aMa; p, M)$ antes de proseguir con otras operaciones de desplazamiento.

Continuando así, el autómata finalmente desplazará la última z de su entrada a la pila (fila13 de la Fig. 2.33); reconocerá que el contenido de la pila, *zMNz*, es una cadena que se puede derivar del símbolo de inicio *S*, reducirá su pila por medio de la transición $(p, \lambda, zMNz; p, S)$ (fila 14); y por fin extraerá el no terminal *S* al pasar al estado *q* (fila 15). A partir de aquí, la máquina extrae el símbolo # de la pila y pasa al estado de aceptación *f*.

Como un asunto secundario, ahora podemos justificar la *R* del término analizador sintáctico *LR(k)*, la cual, según dijimos, indica que en estos analizadores del análisis de las entradas se lleva a cabo construyendo derivaciones por la derecha. Recuerde que una derivación es una

secuencia de reglas de reescritura que transforma el símbolo inicial en la cadena derivada. Sin embargo, el proceso ascendente, cuyo resumen se presenta en la figura 2.33, lo hace a la inversa: genera el símbolo inicial a partir de la cadena derivada. Por consiguiente, la derivación implicada aparece en orden inverso. Para encontrarla, leemos en forma ascendente la columna "contenido de la pila" de la figura 2.33 a la vez que registramos las reglas de reescritura que aplicó el autómata. En nuestro ejemplo, esto revela la derivación

$$S \Rightarrow zMNz \Rightarrow zMbNbz \Rightarrow zMbzbz \Rightarrow zaMabzbz \Rightarrow zazabzbz$$

la cual, como se dijo, es una derivación por la derecha.

Implantación de analizadores sintácticos *LR(k)*

Se presentan dos problemas principales al tratar de convertir los autómatas de pila, como el que se muestra en la figura 2.32, a un formato de programa más tradicional. El primero tiene que ver con el no determinismo, de manera similar a lo que sucede con los analizadores sintácticos *LL*: si se presenta una opción, ¿cómo saber si debemos desplazar o reducir? Además, si nuestra opción es reducir, puede existir más de una reducción posible (*si z* está en la cima de la pila, ¿reducimos con $(p, \lambda, z; p, M)$ o con $(p, \lambda, z; p, N)$? Como podrá suponer, estos asuntos se resuelven con la aplicación del principio de preanálisis.

El segundo problema tiene que ver con los aspectos técnicos de la interrogación de la pila. Por ejemplo, antes de que podamos decidir si ejecutamos la transición $(p, \lambda, aMa; p, M)$, debemos ser capaces de deducir que los tres símbolos superiores de la pila son *a*, *M* y *a*. Sin embargo, en un momento determinado sólo está disponible para observación el símbolo que se encuentra en la cima de la pila. Éste parece ser un problema que puede solucionarse si implantamos la pila como una estructura híbrida que permita observar los símbolos ubicados debajo de la cima de la pila, pero esta modificación soslaya el problema real: la necesidad de realizar búsquedas repetidas en la pila. De hecho, parece que cualquier implantación de una rutina de análisis sintáctico *LR* incluiría un importante componente de reconocimiento de patrones para comparar el contenido de la pila con los lados derechos de las reglas de reescritura.

Quizás el resultado más fascinante en el campo de la construcción de compiladores resida en que este problema de interrogación de la pila se pueda resolver sin necesidad de llevar a cabo costosas búsquedas o siquiera recurrir a estructuras híbridas para la pila. De hecho, en el caso de los lenguajes deterministas independientes del contexto, puede integrarse en una sola tabla de análisis sintáctico toda la información necesaria para resolver este problema de interrogación de la pila, así como el problema de elección de opciones.

En la figura 2.34 se muestra el ejemplo de una tabla para un analizador sintáctico *LR*(1) basado en la gramática de la figura 2.5. Las columnas de esta tabla están rotuladas con los símbolos de la gramática (incluyendo tanto terminales como no terminales) junto con una marca de fin de cadena (representada por FDC). Las filas se etiquetan con números que representan símbolos (componentes léxicos) especiales (veremos que estos símbolos especiales se utilizan para representar patrones que pueden aparecer en la pila).

Para describir los elementos de la tabla, consideremos el proceso donde se usa la tabla. El análisis sintáctico de cualquier cadena comienza asignando el valor uno a una variable de símbolo especial e insertando este valor en la pila vacía. (Desde este momento, a la inserción de un símbolo terminal o no terminal en la pila le seguirá la inserción sobre él valor actual de la variable de símbolo especial. Esto quiere decir que el contenido de la pila alternará entre símbolos terminales o no terminales y símbolos especiales, donde cada uno de estos últimos representa el patrón de lo que yace debajo de él. Por lo tanto, podemos investigar la estructura interna de la pila observando el símbolo especial de la cima.)

	a	b	z	FDC	*S*	*M*	*N*
1			desplazar 2		14		
2	desplazar 3		desplazar 7			4	
3	desplazar 3		desplazar 7			8	
4		desplazar 5	desplazar 9				6
5		desplazar 5	desplazar 9				10
6			desplazar 11				
7	$M \to z$	$M \to z$	$M \to z$				
8	desplazar 12						
9		$N \to z$	$N \to z$				
10		desplazar 13					
11				$S \to zMNz$			
12	$M \to aMa$	$M \to aMa$	$M \to aMa$				
13		$N \to bNb$	$N \to bNb$				
14				aceptar			

Figura 2.34 Tabla de análisis sintáctico *LR*(1) basada en la gramática de la figura 2.5

Una vez que el símbolo especial se ha establecido y se ha almacenado en la pila, hacemos referencia a la tabla de análisis sintáctico. La fila que nos interesa está determinada por el símbolo especial actual y la columna por el símbolo de preanálisis. Los casos más sencillos se presentan cuando la casilla correspondiente de la tabla está vacía o contiene la palabra aceptar. En el primer caso se considera que la cadena es inválida y que debe ejecutarse la rutina de error adecuada. En el segundo caso se indica que la cadena leída de la entrada es aceptable y que el proceso de análisis sintáctico debe concluir.

Otra posibilidad es que la casilla de la tabla contenga desplazar, lo cual indica que debe ejecutarse la operación de desplazamiento. En este caso el siguiente símbolo debe leerse (el símbolo de preanálisis) de la entrada y colocarse en la pila; a la variable de símbolo especial se le debe asignar el valor que existe en la casilla de la tabla junto con la operación de desplazamiento y este nuevo valor de símbolo especial debe insertarse en la pila; por último, debe actualizarse el símbolo de preanálisis.

La última posibilidad es que la entrada de la tabla contenga una regla de reescritura de la gramática, lo cual indica que se trata una operación de reducción. El proceso que aquí se requiere implica la sustitución de una cadena de símbolos de la pila (el lado derecho de la regla de reescritura) con un solo no terminal (el lado izquierdo de la regla). Sin embargo, se requiere un poco más de detalle para manejar los valores de los símbolos especiales que también se encuentran almacenados en la pila. En primer lugar hay que eliminar dos símbolos de la pila por cada símbolo del lado derecho de la regla de reescritura. Esto elimina cada uno de los símbolos del lado derecho de la regla, así como el valor de símbolo especial que se encuentra almacenado encima del símbolo. En este punto, la cima de la pila contendrá el valor del símbolo especial que se colocó después de crear la porción inferior (la que queda) de la pila. Hay que recordar este valor como "símbolo especial temporal" e insertar encima el no terminal del lado izquierdo de la regla de reescritura. Luego, este no terminal se emplea para identificar una columna de la tabla de análisis sintáctico, a la vez que el símbolo especial temporal determina una fila. El valor que se encuentra en este lugar en la tabla de análisis sintáctico debe asignarse a la variable de símbolo especial y además debe insertarse en la pila.

Así, al emplear una tabla de análisis sintáctico, el analizador *LR* sencillamente hace referencia de manera cíclica a la tabla hasta encontrar una entrada en blanco o de aceptación. En la figura 2.35 se presenta un algoritmo de análisis sintáctico *LR*(1) que utiliza la tabla de la figura 2.34. La figura 2.36 resume las actividades del analizador durante el procesamiento de la cadena *zazabzbz*. Observe que esta figura es esencialmente igual que la figura 2.33, excepto por los valores adicionales de símbolos especiales en la pila.

```
Símbolo Especial := 1;
insertar (Símbolo Especial);
leer (Símbolo);
Valor Tabla := Tabla [Símbolo Especial, Símbolo];
while Valor Tabla no es "aceptar" do
    if Valor Tabla es un desplazamiento
      then begin
            insertar (Símbolo); Símbolo Especial := Valor Tabla. Estado;
            insertar (Símbolo Especial); leer (Símbolo)
            end
    else if Valor Tabla es una reducción
      then begin
            extraer (lado derecho de Valor Tabla.ReglaReescritura);
            Símbolo Especial := cima-de-la-pila; (* Esto no es una
      extracción *)
            insertar (lado izquierdo de Valor Tabla.ReglaReescritura);
            Símbolo Especial := Tabla [Símbolo Especial
            lado izquierdo de Valor Tabla.ReglaReescritura];
            insertar (Símbolo Especial)
            end
    else if Valor Tabla está en blanco then salir a la rutina de error;
    Valor Tabla := Tabla [Símbolo Especial, Símbolo];
end while;
if Símbolo no es FDC then salir a la rutina de error;
vaciar pila
```

Figura 2.35 Algoritmo de análisis sintáctico *LR*(1)

Tablas de análisis sintáctico *LR*

Aunque la construcción de una tabla de análisis sintáctico *LR* corresponde más a la construcción de compiladores que a la teoría de los lenguajes formales, no debemos ignorar por completo este tema. De hecho, la construcción de estas tablas es una importante aplicación de la teoría de autómatas finitos. La tabla de un analizador sintáctico *LR*(1) se basa en la existencia de un autómata finito que acepta exactamente las cadenas de símbolos de la gramática (terminales y no terminales) que conducen a operaciones de reducción. Aquí presentamos un breve ejemplo de esto; en el apéndice A se ofrecen mayores detalles.

El diagrama de la figura 2.37 es el diagrama de transiciones del autómata finito a partir del cual se construyó la tabla de análisis sintáctico de la figura 2.34. Observe que acepta cadenas como *zaMa*, *zMbNb* y *zMNz*, las cuales, cuando se encuentran en la pila del analizador, deben reducirse a *zM*, *zMN* y *S*, respectivamente. En términos de este diagrama, el objetivo del analizador sintáctico *LR* es llegar al estado 14 recorriendo el arco con etiqueta *S*.

Contenido de la pila	Resto de la entrada
vacía	*zazabzbz*
①	*zazabzbz*
① *z* ②	*azabzbz*
① *z* ② *a* ③	*zabzbz*
① *z* ② *a* ③ *z* ⑦	*abzbz*
① *z* ② *a* ③ *M* ⑧	*abzbz*
① *z* ② *a* ③ *M* ⑧ *a* ⑫	*bzbz*
① *z* ② *M* ④	*bzbz*
① *z* ② *M* ④ *b* ⑤	*zbz*
① *z* ② *M* ④ *b* ⑤ *z* ⑨	*bz*
① *z* ② *M* ④ *b* ⑤ *N* ⑩	*bz*
① *z* ② *M* ④ *b* ⑤ *N* ⑩ *b* ⑬	*z*
① *z* ② *M* ④ *N* ⑥	*z*
① *z* ② *M* ④ *N* ⑥ *z* ⑪	
① *S* ⑭	
vacía	

Figura 2.36 Análisis sintáctico de la cadena *zazabzbz* con el algoritmo de la figura 2.35 y la tabla de la figura 2.34

Para lograrlo, ejecuta lo que podría parecer un proceso de prueba y error donde el analizador sigue repetidamente una ruta hacia un estado de aceptación, retrocede a un estado anterior siguiendo la misma ruta en sentido inverso y a partir de allí se encamina hacia una nueva dirección.

No obstante, este proceso no es de prueba y error, sino una secuencia bien definida de eventos guiados por los símbolos que se detectan en la cadena analizada. La idea es comenzar el proceso de análisis sintáctico siguiendo la ruta determinada por la cadena de entrada hasta encontrar un estado de aceptación. Al llegar a este punto, la ruta recorrida por el autómata finito corresponde al patrón de símbolos que el analizador ha desplazado a la pila. Entonces, el proceso de análisis retrocede por esta ruta recorriendo los símbolos que deben eliminarse de la pila del analizador durante la operación de reducción. A partir de este punto, el analizador sintáctico recorre el arco del diagrama de transiciones del autómata finito que tenga etiqueta equivalente al no terminal colocado en la pila por la operación de reducción correspondiente. Así, una vez que se ha completado la operación de reducción, los símbolos en la pila del analizador corresponderán una vez más a los símbolos de la ruta que recorre el autómata finito.

Para ver cómo funciona este proceso, consideremos de nuevo la tarea de análisis sintáctico de la cadena *zazabzbz*. Conforme el analizador lee los

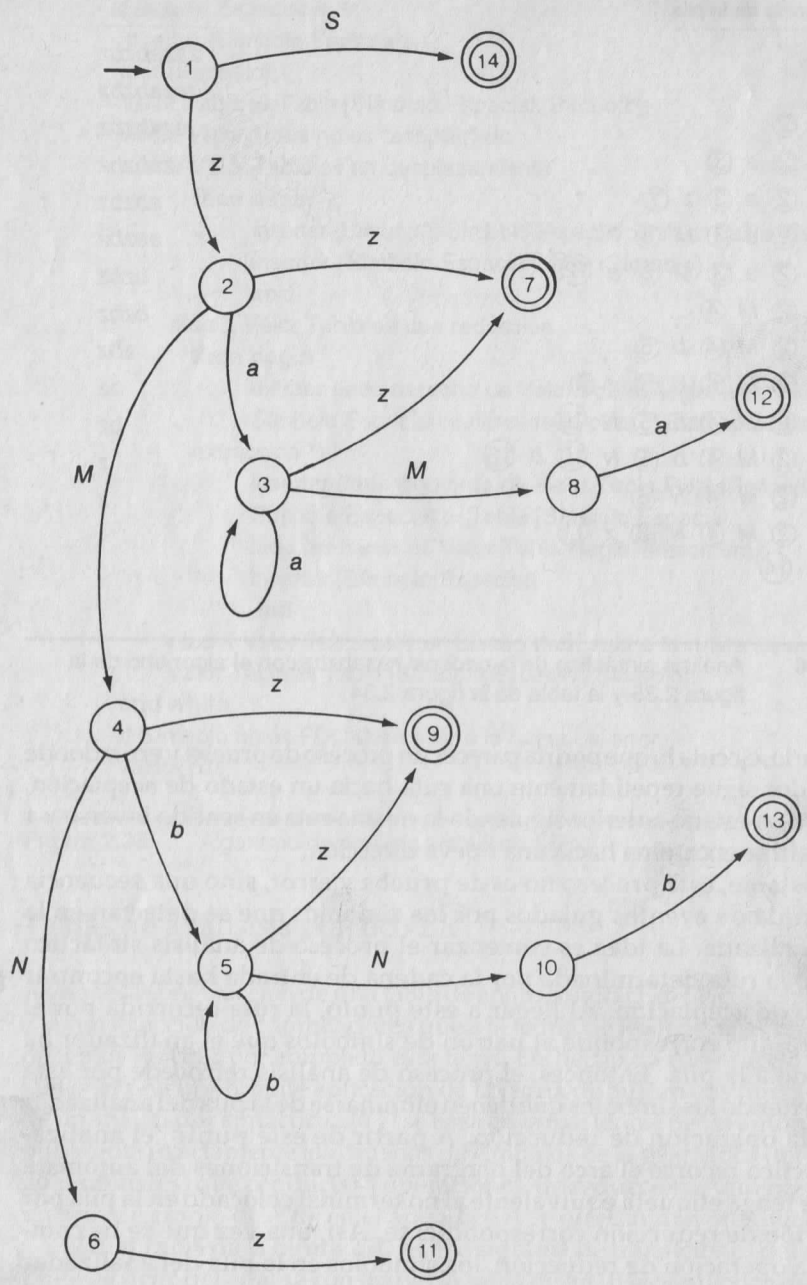


Figura 2.37 Autómata finito a partir del cual se construyó la tabla de la figura 2.34

símbolos *zaz*, el autómata finito se mueve por la secuencia de estados 1, 2, 3 y 7. Al llegar a este punto, se requiere una operación de reducción basada en la regla $M \rightarrow z$. Entonces, el autómata retrocede al estado 3 (siguiendo el arco z) y del estado 3 pasa al estado 8 por el arco con etiqueta *M*. Al mismo tiempo, el analizador sintáctico *LR* extrae la *z* superior de la pila y la reemplaza con el no terminal *M*. Así, la nueva ruta que recorre el autómata finito (a través de los estados 1, 2, 3 y 8) corresponde de nuevo al patrón de símbolos en la pila del analizador. La lectura del siguiente símbolo de la entrada, *a*, lleva al autómata finito al estado 12, donde el analizador retrocederá al estado 2 y seguirá el arco con etiqueta *M* hasta el estado 4. De esta manera, el proceso de análisis sintáctico finalmente llegará al estado 11 a través de los arcos *z*, *M*, *N* y *z*. De aquí, el proceso de análisis retrocederá al estado 1 y pasará al estado 14 ya que la cadena *zMNz* se reduce al símbolo inicial *S*.

Tomando en cuenta esta situación, no es difícil comprender la construcción de un analizador sintáctico *LR*(1). Los valores de los símbolos especiales representan los estados del autómata finito. Las casillas de desplazamiento de la tabla corresponden a los arcos rotulados con terminales, mientras que el símbolo especial que se encuentra en esa casilla representa el estado al final del arco. Las casillas de reducción indican que el analizador ha llegado a un estado de aceptación en el autómata finito, y también proporcionan la información necesaria para realizar el proceso de retrocesos. (Observe que el estado de un autómata finito donde deberá detenerse el retroceso está representado por el símbolo especial que aparece en la cima de la pila después de extraer el lado derecho de la regla de reescritura.) Las casillas de las columnas etiquetadas con no terminales permiten al analizador sintáctico establecer una nueva dirección en el diagrama después del retroceso.

Hay que hacer un comentario final antes de abandonar nuestro tratamiento de los analizadores sintácticos *LR*. Para completar un compilador es necesario combinar un analizador con un generador de código que produzca instrucciones en el lenguaje objeto que reflejen las estructuras detectadas por el analizador. Al emplear un analizador sintáctico *LR*, este enlace entre el analizador y el generador de código se presenta en relación con cada operación de reducción. De hecho, es en esta etapa del proceso de análisis sintáctico donde se ha reconocido una estructura, por lo que es el momento indicado para solicitar al generador de código que construya el código para esa estructura. En resumen, los cálculos efectuados por el analizador sintáctico y el generador de código siguen el patrón básico.

```
repeat
    analizar la sintaxis hasta ejecutar una reducción
    generar código
until termina el análisis sintáctico
```

Comparación entre los analizadores sintácticos *LR*(*k*) y *LL*(*k*)

Para concluir, debemos confirmar que la colección de analizadores sintácticos *LR*(*k*) es más poderosa que la de los analizadores *LL*(k). Ya hemos planteado que ningún analizador sintáctico *LL*(*k*) puede manejar el lenguaje $\{x^n: n \in \mathbb{N}\} \cup \{x^n y^n: n \in \mathbb{N}\}$. Sin embargo, en la figura 2.38 se presenta una gramática para este lenguaje y su tabla de análisis sintáctico *LR*(1) correspondiente. Si se combina esta tabla con el algoritmo de la figura 2.35, se obtiene un analizador *LR*(1)que reconoce el lenguaje (esto se logra desplazando las *x* de la entrada a la pila hasta alcanzar una *y* o una marca de fin de cadena. En ese momento, el analizador es capaz de aplicar las reglas de reescritura apropiadas para analizar correctamente la cadena).

Existen, no obstante, lenguajes independientes del contexto que ningún analizador sintáctico *LR*(*k*) puede reconocer. De hecho, la clase de lenguajes

$$S \to X$$
$$S \to Y$$
$$S \to \lambda$$
$$X \to xX$$
$$Y \to xYy$$
$$Y \to xy$$

	x	y	FDC	S	X	Y
1	desplazar 2		$S \to \lambda$	9	8	7
2	desplazar 2	desplazar 6	$X \to x$		3	4
3			$X \to xX$			
4		desplazar 5				
5		$Y \to xXy$				
6		$Y \to xy$				
7						
8						
9			aceptar			

Figura 2.38 Gramática independiente del contexto y su tabla de análisis sintáctico *LR*(1)

que pueden ser analizados por los analizadores *LR*(*k*) es precisamente la clase de los lenguajes independientes del contexto deterministas. Aunque no lo demostraremos, por lo menos debemos señalar que este límite del poder de los analizadores *LR*(*k*) está de acuerdo con nuestra intuición: un analizador sintáctico *LR*(*k*) debe ser determinista y, puesto que su estructura se basa en un autómata de pila, debe deducirse que los analizadores sintácticos *LR*(*k*) sólo pueden analizar aquellos lenguajes que aceptan los autómatas de pila deterministas.

Un ejemplo de lenguaje independiente del contexto que un analizador sintáctico *LR*(*k*) no puede analizar el lenguaje

$$\{x^n y^n: n \in \mathbb{N}^+\} \cup \{x^n y^{2n}: n \in \mathbb{N}^+\}$$

Intuitivamente, el problema aquí es que una vez que se llega a la primera *y* de la entrada, el analizador sintáctico debe decidir cuál es la regla de reescritura que debe aplicarse conociendo sólo los siguientes *k* símbolos de la cadena. Si *n* es mayor que *k*, conocer sólo los siguientes *k* símbolos no basta para que el analizador detecte si la entrada contendrá *n* y o 2*n y*, y por lo tanto el analizador es incapaz de seleccionar la regla de reescritura correcta.

Ejercicios

1. Con el proceso alternativo descrito en esta sección para construir un autómata de pila a partir de una gramática independiente del contexto, construya un diagrama de transiciones para un autómata de pila utilizando la gramática que se presenta a continuación.

$$S \to xSz$$
$$S \to ySz$$
$$S \to \lambda$$

2. Identifique la derivación obtenida por el autómata de pila construido en el ejercicio 1 al analizar la cadena *yxyzzz*.

3. Construya una tabla de análisis *LR*(1) para la gramática siguiente (existen procesos algorítmicos que hacen esto y que no hemos analizado [para obtener más información consulte el apéndice A], pero esta gramática es lo suficientemente sencilla para que pueda desarrollarse una tabla con sólo saber cómo se usan las tablas de análisis sintáctico *LR*(1)).

$$S \to xSy$$
$$S \to \lambda$$

2.6 COMENTARIOS FINALES

Primero debemos aclarar la relación entre la jerarquía asociada con los lenguajes independientes del contexto y la clase de lenguajes regulares presentados en el capítulo anterior. Ya hemos visto que los lenguajes regulares son independientes del contexto, pero la clasificación puede ser más precisa. Los lenguajes regulares están propiamente contenidos en la clase de los lenguajes aceptados por los autómatas de pila deterministas que vacían sus pilas antes de aceptar una cadena. Esto es evidente si se observa que cualquier lenguaje regular puede ser aceptado por un autómata finito determinista, que es en esencia un autómata de pila determinista cuyas transiciones nunca utilizan la pila, y como la pila nunca se emplea, debe estar vacía cuando se llega a un estado de aceptación. Además, la inclusión es propia puesto que el lenguaje $\{x^y y^n: n \in \mathbb{N}\}$ no es regular pero sí aceptado por un autómata de pila determinista que vacía su pila antes de aceptar una cadena (Fig. 2.2).

Así, en la figura 2.39, una ampliación de la figura 2.23, podemos resumir lo que hasta ahora ha cubierto nuestro estudio de los lenguajes.

Por último, como primer paso para introducir algunos de los temas de los capítulos restantes, observe que los lenguajes independientes del contexto no son cerrados para la intersección. Por ejemplo, los lenguajes $\{x^n y^n z^m: m, n \in \mathbb{N}\}$ y $\{x^m y^n z^n: m, n \in \mathbb{N}\}$ son independientes del contexto (el primero es generado por $S \to TZ$, $T \to \lambda$, $T \to xTy$, $Z \to \lambda$, $Z \to zZ$, y el segundo es similar), pero su intersección es el lenguaje $\{x^n y^n z^n: m, n \in \mathbb{N}\}$, el cual, como hemos visto, no es independiente del contexto. Al combinar esto con el hecho de que los lenguajes independientes del contexto son cerrados para la unión finita, y aplicar la leyes de DeMorgan, resulta que los lenguajes independientes del contexto no son cerrados para la complementación. Es decir, existen subconjuntos de Σ^* que son independientes del contexto pero cuyos complementos en Σ^* no lo son.

Esto significa que la capacidad de un autómata de pila para aceptar cadenas de un lenguaje no es simétrica con la capacidad para rechazar las cadenas que no se encuentran en el lenguaje (la capacidad para rechazar las cadenas que no están en el lenguaje equivaldría a la capacidad para aceptar el complemento del lenguaje). Así, existe una importante diferencia entre la capacidad para responder sí cuando la cadena está en el lenguaje y la capacidad para responder sí o no cuando la cadena está o no está en el lenguaje (en el primer caso, si la entrada no se halla en el lenguaje, la máquina puede quedar atrapada en un ciclo y nunca responder. En el segundo caso, la máquina debe, a fin de cuentas, responder de manera correcta sin importar si la respuesta es sí o no). La carencia de simetría será un factor importante en análisis subsecuentes.

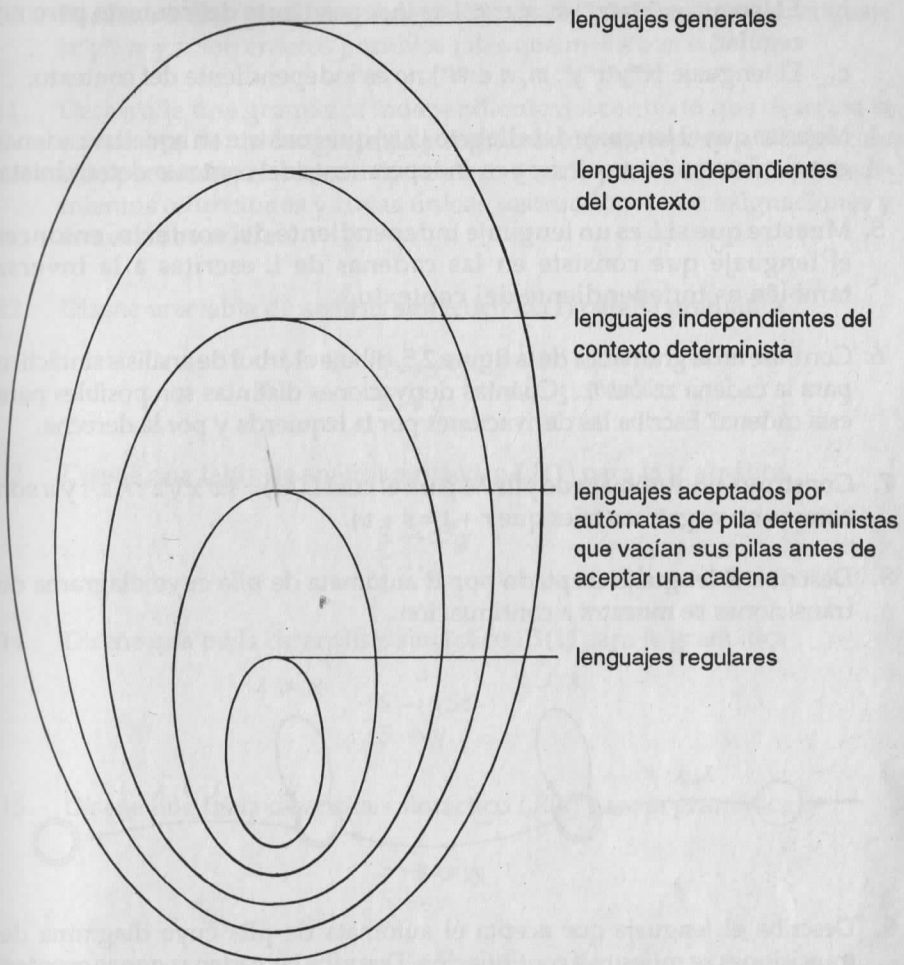


Figura 2.39 La jerarquía de los lenguajes que hemos presentado hasta ahora

Problemas de repaso del capítulo

1. Para cada nivel en la jerarquía de la figura 2.39, proporcione un ejemplo de un lenguaje que exista en ese nivel pero no en el nivel inmediato inferior.

2. Muestre que el lenguaje $\{x^r y^s z^t: s = r + t\}$ es independiente del contexto.

3. Muestre que:
   a. El lenguaje $\{x^m y^n: m, n \in \mathbb{N}^+\}$ es regular.

b. El lenguaje $\{x^m y^n x^m : m, n \in \mathbb{N}^+\}$ es independiente del contexto pero no regular.

c. El lenguaje $\{x^m y^n x^m y^n : m, n \in \mathbb{N}^+\}$ no es independiente del contexto.

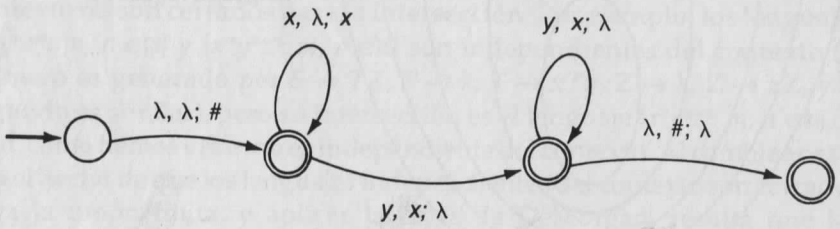
4. Muestre que el lenguaje del alfabeto $\{x, y\}$ que consiste en aquellas cadenas con el mismo número de x y y es independiente del contexto determinista.

5. Muestre que si L es un lenguaje independiente del contexto, entonces el lenguaje que consiste en las cadenas de L escritas a la inversa también es independiente del contexto.

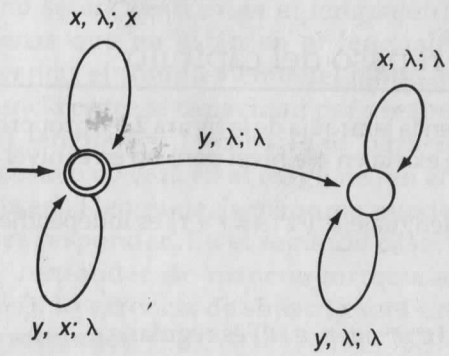
6. Con base en la gramática de la figura 2.5, dibuje el árbol de análisis sintáctico para la cadena *zzbbzbbz*. ¿Cuántas derivaciones distintas son posibles para esta cadena? Escriba las derivaciones por la izquierda y por la derecha.

7. Construya un autómata de pila M para el cual $L(M) = \{w^r x^s y^t z^u : r, s, t$ y u son enteros no negativos tales que $r + t = s + u\}$.

8. Describa el lenguaje aceptado por el autómata de pila cuyo diagrama de transiciones se muestra a continuación.



9. Describa el lenguaje que acepta el autómata de pila cuyo diagrama de transiciones se muestra a continuación. Describa las cadenas que se aceptan cuando la pila de la máquina no está vacía. Modifique el diagrama para que acepte las mismas cadenas que antes pero sólo después de vaciar la pila.



10. Encuentre una gramática independiente del contexto para el lenguaje $\{x^n y^m : m$ y n son enteros positivos tales que $m = n$ o $m = 2n\}$.

11. Desarrolle una gramática independiente del contexto que describa la estructura de un lenguaje de programación semejante a Pascal que sólo permite variables de tipo entero o real, que no contiene procedimientos o funciones y cuyas únicas instrucciones son asignaciones y enunciados while.

12. Diseñe una tabla de análisis sintáctico $LL(1)$ para la gramática

$$S \to xSz$$
$$S \to y$$

13. Diseñe una tabla de análisis sintáctico $LL(1)$ para la gramática

$$S \to xSy$$
$$S \to y$$

14. Diseñe una tabla de análisis sintáctico $ID(1)$ para la gramática

$$S \to xSz$$
$$S \to y$$

15. Diseñe una tabla de análisis sintáctico $LR(1)$ para la gramática

$$S \to xSy$$
$$S \to y$$

16. Diseñe un autómata de pila determinista M para el cual $L(M)$ sea el lenguaje generado por la gramática

$$S \to xSx$$
$$S \to y$$

17. Diseñe un autómata de pila determinista M para el cual $L(M)$ sea el lenguaje generado por la gramática

$$S \to xSy$$
$$S \to y$$

18. Diseñe un autómata de pila determinista M para el cual $L(M)$ sea el lenguaje generado por la gramática siguiente, con símbolo inicial S.

$$S \to xyN$$
$$N \to zS$$
$$N \to z$$
$$N \to \lambda$$

{x y ou *) (n}

19. Sea $\Sigma = \{x, y, «, », (,)\}$. Diseñe un autómata de pila M tal que $L(M)$ sea el conjunto de todas las cadenas que contituyan expresiones regulares del alfabeto $\{x, y\}$.

20. Construya una gramática G independiente del contexto tal que $L(G)$ sea el conjunto de todas las cadenas formadas por la concatenación de una cadena en $\{x, y\}^*$ con la misma cadena escrita a la inversa.

21. Muestre que un lenguaje que consiste en los palíndromos de $\{x, y\}^*$ es independiente del contexto (un palíndromo es una cadena que es idéntica a sí misma escrita a la inversa).

22. a. Diseñe una gramática independiente del contexto que genere ellenguaje $\{x^m y^n$: m y n son enteros no negativos donde $n < m\}$.

    b. Diseñe una gramática independiente del contexto que genere el lenguaje $\{x^m y^n$: m y n son enteros no negativos donde $n > m\}$.

23. a. Diseñe un autómata de pila M tal que $L(M) = \{x^m y^n$: m y n son enteros no negativos y $m \le n \le 2m\}$.

    b. Diseñe un autómata de pila M tal que $L(M) = \{x^m y^n$: m y n son enteros no negativos y $n < m$ o $2m < n\}$.

24. Muestre que el lenguaje $\{x^n$: n es un entero positivo primo$\}$ no es independiente del contexto.

25. Muestre que el lenguaje $\{x^n$: $n = m^2$ para una $m \in \mathbb{N}\}$ no es independiente del contexto.

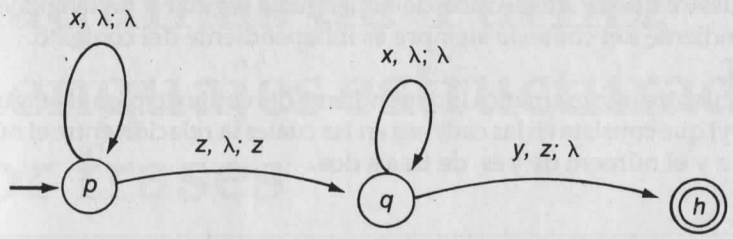
26. Muestre que el lenguaje $\{x^n y^n z x^n y^n$: $n \in \mathbb{N}\}$ no es independiente del contexto.

27. ¿Es el lenguaje $\{x^m y^n$: $m, n \in \mathbb{N}$ y $m \le n \le 2m\}$ independiente del contexto? Justifique su respuesta.

28. ¿Es siempre independiente del contexto la unión de una colección de lenguajes independientes del contexto? Justifique su respuesta.

29. Utilizando un argumento de cardinalidad, muestre que deben existir lenguajes que no son independientes del contexto.

30. Aplique la técnica utilizada en la demostración del teorema 2.3 para construir una gramática independiente del contexto que genere el lenguaje aceptado por el autómata de pila descrito a continuación.



31. Aplique la técnica utilizada en la demostración del teorema 2.3 para construir una gramática independiente del contexto que genere el lenguaje aceptado por el autómata de pila descrito en la figura 2.2.

32. Convierta la gramática siguiente, con símbolo inicial S, a la forma normal de Chomsky.

$$S \to MzN$$
$$M \to xM$$
$$M \to \lambda$$
$$N \to yN$$
$$N \to \lambda$$

33. Encuentre una gramática independiente del contexto en forma normal de Chomsky que genere el mismo lenguaje que la gramática de la figura 2.5.

34. ¿Por qué se preferiría utilizar la colección de reglas de reescritura

$$X \to xY$$
$$Y \to yZ$$
$$Z \to z$$
$$Z \to \lambda$$

en vez de

$$X \to xyz$$
$$X \to x$$

al construir un analizador sintáctico?

35. Muestre que si L es un lenguaje independiente del contexto que no contiene la cadena vacía, entonces existe una gramática G independiente

del contexto en la cual el lado derecho de cada regla de reescritura consiste en un terminal seguido por cero o más no terminales (se dice que estas gramáticas tienen la forma normal de Griebach).

36. Muestre que la intersección de un lenguaje regular y un lenguaje independiente del contexto siempre es independiente del contexto.

37. Encuentre una gramática independiente del contexto para el lenguaje de $\{x, y\}$ que consiste en las cadenas en las cuales la relación entre el número de x y el número de y es de tres a dos.

Problemas de programación

1. Realice el algoritmo de análisis sintáctico $LL(1)$ que se muestra en la figura 2.30. Aplique los resultados a las tablas de análisis sintáctico de las figuras 2.29 y 2.31, así como a su respuesta al problema 13 de repaso del capítulo.

2. Realice el algoritmo de análisis sintáctico $LR(1)$ que se muestra en la figura 2.35. Aplique los resultados a las tablas de análisis sintáctico de las figuras 2.34 y 2.38 así como a su respuesta al problema 15 de repaso del capítulo.

3. Escriba un programa para convertir gramáticas independientes del contexto que no generan la cadena vacía en gramáticas con la forma normal de Chomsky.

CAPÍTULO 3

Máquinas de Turing y lenguajes estructurados por frases

3.1 **Máquinas de Turing**
Propiedades básicas de las máquinas de Turing
Los orígenes de las máquinas de Turing
3.2 **Construcción modular de máquinas de Turing**
Combinación de máquinas de Turing
Bloques de construcción básicos
3.3 **Máquinas de Turing como aceptadores de lenguajes**
Procedimientos de evaluación de cadenas
Máquinas de Turing de varias cintas
Máquinas de Turing no deterministas
3.4 **Lenguajes aceptados por máquinas de Turing**
Comparación entre lenguajes aceptados por máquinas de Turing y lenguajes estructurados por frases
Alcance de los lenguajes estructurados por frases
3.5 **Más allá de los lenguajes estructurados por frases**
Sistema de codificación de máquinas de Turing
Un lenguaje no estructurado por frases
Máquinas de Turing universales
Comparación entre lenguajes aceptables y decidibles
El problema de la parada
3.6 **Comentarios finales**

Hasta ahora hemos presentado las clases de autómatas finitos y de pila. En ambos casos nuestro objetivo ha sido comprender el potencial de reconocimiento de lenguajes de estas máquinas teóricas. En este capítulo presentaremos otra clase de autómatas, todavía más generales, conocidos como máquinas de Turing, y estudiaremos el poder computacional de

estas máquinas en el contexto de la solución de problemas de reconocimiento de lenguajes.

Uno de los principales resultados será que las máquinas de Turing son capaces de aceptar exactamente los mismos lenguajes que pueden generar las gramáticas menos restrictivas: las gramaticas estructuradas por frases y así, el poder de procesamiento de lenguajes de esta clase de máquinas está limitado por las mismas fronteras que delimitan los poderes generativos de las gramáticas. La importancia de esta equivalencia aumenta por el hecho de que nadie ha podido definir una clase de máquinas computacionales más poderosas que las máquinas de Turing. De hecho, aprenderemos que en la actualidad los científicos de la computación aceptan de manera general la conjetura de que la clase de máquinas de Turing encierra el poder de cualquier proceso computacional (esta conjetura se conoce como tesis de Turing). Por esto, se cree que la correspondencia entre el potencial de reconocimiento de lenguajes de las máquinas de Turing y el poder generativo de las gramáticas que veremos en este capítulo define los límites máximos de cualquier sistema computacional de reconocimiento de lenguajes.

En resumen, la presentación de las máquinas de Turing en este capítulo servirá como culminación de nuestro estudio de las gramáticas, los lenguajes y los algoritmos de análisis sintáctico. Empero, esto no quiere decir que habrá concluido todo nuestro estudio; más bien, las limitaciones aparentes de los poderes computacionales que descubriremos en este capítulo serán el punto de partida para el estudio de la computabilidad.

3.1 MÁQUINAS DE TURING

La clase de autómatas que ahora se conoce como **máquinas de Turing** fue propuesta por Alan M. Turing en 1936. (La idea básica de Turing fue estudiar los procesos algorítmicos utilizando un modelo computacional. En ese mismo año, Emil L. Post presentó un enfoque similar, y más tarde se mostró que ambas estrategias son equivalentes en cuanto a poder computacional.) Para nuestros fines es conveniente considerar a las máquinas de Turing como una versión generalizada de los autómatas que hemos visto en capítulos anteriores. Las máquinas de Turing se asemejan a los autómatas finitos en que constan de un mecanismo de control y un flujo de entrada que concebimos como una cinta; la diferencia es que las máquinas de Turing pueden mover sus cabezas de lectura hacia adelante y hacia atrás y pueden leer o escribir en la cinta. Veremos que estas características aumentan en gran medida la capacidad de las máquinas.

Propiedades básicas de las máquinas de Turing

Al igual que las demás máquinas que hemos estudiado, la máquina de Turing contiene un mecanismo de control que en cualquier momento puede encontrarse en uno de entre un número finito de estados. Uno de estos estados

se denomina estado inicial y representa el estado en el cual la máquina comienza los cálculos. Otro de los estados se conoce como **estado de parada**; una vez que la máquina llega a ese estado, terminan todos los cálculos. De esta manera, el estado de parada de una máquina de Turing difiere de los estados de aceptación de los autómatas finitos y de pila en que éstos pueden continuar sus cálculos después de llegar a un estado de aceptación, mientras que una máquina de Turing debe detenerse en el momento en que llegue a su estado de parada. (nota: con base en la definición anterior, el estado inicial de una máquina de Turing no puede ser a la vez el estado de parada; por lo tanto, toda máquina de Turing debe tener cuando menos dos estados).

Una diferencia más importante entre una máquina de Turing y los autómatas de los capítulos anteriores es que la máquina de Turing puede leer y escribir en su medio de entrada. Para ser más precisos, la máquina de Turing está equipada con una cabeza que puede emplearse para leer y escribir símbolos en la cinta de la máquina (como sucede con los otros autómatas, esta cinta tiene un extremo izquierdo pero se extiende indefinidamente hacia la derecha). Así, una máquina de Turing puede emplear su cinta como almacenamiento auxiliar, de la misma forma en que otra clase de autómatas utiliza su pila. Sin embargo, con este almacenamiento, una máquina de Turing no se limita a las operaciones de inserción y extracción, sino que puede rastrear los datos de la cinta y modificar las celdas que desee sin alterar las demás.

Al utilizar la cinta para fines de almacenamiento auxiliar, es conveniente que una máquina de Turing emplee marcas especiales para distinguir porciones de la cinta. Para esto, permitimos que una máquina de Turing lea y escriba símbolos que no aparecen en los datos de entrada; es decir, hacemos una distinción entre el conjunto (finito) de símbolos, llamado alfabeto de la máquina, en el que deben estar codificados los datos de entrada iniciales, y un conjunto, posiblemente mayor (también finito), de **símbolos de cinta**, que la máquina puede leer y escribir (esta distinción es similar a la que se establece entre el alfabeto de un autómata y sus símbolos de pila). De esta manera, los símbolos de cinta de una máquina de Turing pueden incluir marcas especiales que no sean símbolos del alfabeto de la máquina.

El espacio en blanco es un símbolo que cae en esta categoría; se trata de un símbolo que se supone está en cualquier celda de la cinta que no esté ocupada. Por ejemplo, si una máquina de Turing leyera más allá de los símbolos de entrada de su cinta, encontraría y leería las celdas en blanco que allí se encuentran. Además, puede ser necesario que una máquina de Turing tenga que borrar una celda, escribiendo en ella un espacio en blanco. Por esto, con frecuencia se considera que el espacio en blanco pertenece al conjunto de símbolos de cinta de la máquina de Turing, pero no forma parte del alfabeto de la máquina.

Es fácil que el símbolo de espacio en blanco ocasione confusiones y malas interpretaciones en una página impresa. Por cuestiones de comunicación, adoptamos el símbolo $\triangle$ para representar el espacio en blanco. Esta convención elimina la ambigüedad entre las cadenas xy y $x\ y$ ya que podemos expresar la primera como $x\triangle y$, y la segunda, como $x\triangle\triangle y$.

Las acciones específicas que puede realizar una máquina de Turing consisten en operaciones de escritura y de movimiento. La operación de escritura consiste en reemplazar un símbolo en la cinta con otro símbolo y luego cambiar a un nuevo estado (el cual puede ser el mismo donde se encontraba antes). La operación de movimiento comprende mover la cabeza una celda a la derecha o a la izquierda y luego pasar a un nuevo estado (que, una vez más, puede ser igual al de partida). La acción que se ejecutará en un momento determinado dependerá del símbolo (el símbolo actual) que está en la celda visible en ese momento para la cabeza (la celda actual), así como del estado actual del mecanismo de control de la máquina.

Si representamos con Γ el conjunto de símbolos de cinta de una máquina de Turing, con S, el conjunto de estados y con S', el conjunto de estados que no son el de parada, entonces es posible representar las transiciones de la máquina por medio de una función, llamada **función de transición** de la máquina, de la forma $\delta:(S' \times \Gamma) \rightarrow S \times (\Gamma \cup \{L, R\})$, donde suponemos que los símbolos L y R no pertenecen a Γ.

La semántica de esta representación funcional es la siguiente:

a. $\delta(p, x) = (q, y)$ significa "si el estado actual es p y el símbolo actual es x, reemplazar la x con el símbolo y y pasar al estado q".
b. $\delta(p, x) = (q, L)$ significa "si el estado actual es p y el símbolo actual es x, mover la cabeza una celda a la izquierda y pasar al estado q".
c. $\delta(p, x) = (q, R)$ significa "si el estado actual es p y el símbolo actual es x, mover la cabeza una celda a la derecha y pasar al estado q".

Observe que, al describir con una función las transiciones de una máquina de Turing, la máquina es determinista. Para ser más precisos, existe una, y sólo una, transición asociada a cada par estado–símbolo donde el estado no es el de detención.

Durante la operación normal, una máquina de Turing ejecuta transiciones repetidamente hasta llegar al estado de parada (esto quiere decir que en ciertas condiciones es posible que nunca se detengan los cálculos de una máquina de Turing, ya que su programa interno puede quedar atrapado en un ciclo sin fin). Existe, sin embargo, una anomalía que puede ocurrir durante este proceso: la cabeza de la máquina puede "rebasar" el extremo izquierdo de la cinta. En este caso, la máquina abandonará los cálculos y decimos que la ejecución de la máquina sufrió una **terminación anormal.**

Como sucede con otros autómatas, es útil representar visualmente una máquina de Turing, como se hace en la figura 3.1. Allí, el mecanismo de control de la máquina está representado por un rectángulo con una especie de carátula de reloj que indica el estado actual de la máquina. Encima de este rectángulo se encuentra la cinta de la máquina; la posición de la cabeza de la máquina está indicada con un apuntador, como se hizo en los autómatas finitos y de pila.

Como también sucede con los otros autómatas, la colección de transiciones de una máquina de Turing se puede representar de manera conveniente por medio de un diagrama de transiciones en el cual se ilustran con pequeños

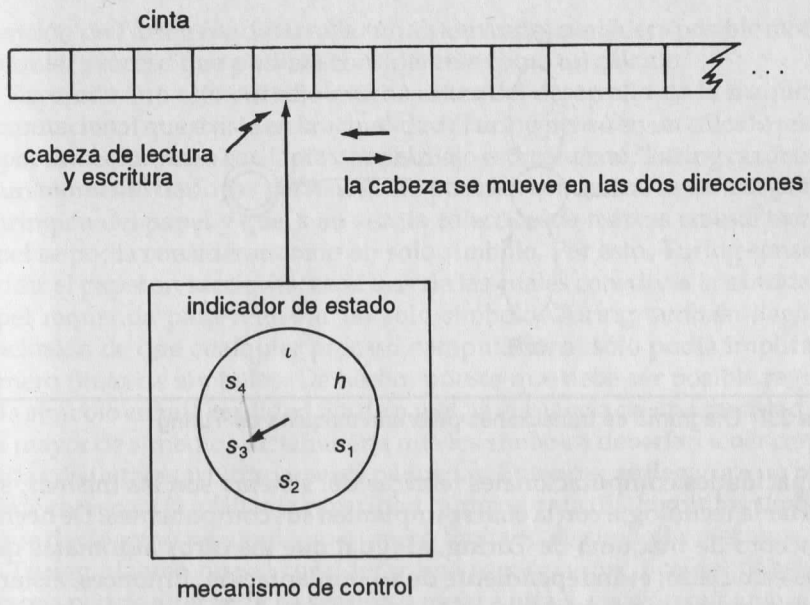


Figura 3. 1 Representación de una máquina de Turing

círculos los estados de la máquina (identificando los estados inicial y de parada con un apuntador y un doble círculo respectivamente) conectados por arcos que representan las transiciones posibles. Cada uno de los arcos del diagrama de transiciones de una máquina de Turing se etiqueta con un par de símbolos separados por una diagonal. El primer símbolo del par representa el símbolo de la cinta que debe existir en la celda actual para que la transición sea aplicable; el segundo símbolo es el que se escribirá en la celda actual (si la transición es una operación de escritura) o de lo contrario uno de los símbolos, L o R (si la transición es una operación de movimiento). De esta manera, la transición $\delta(p, x) = (q, y)$ estaría representada por un arco de p a q con etiqueta x/y; o $\delta(p, x) = (q, L)$ aparecería como un arco de p a q con etiqueta x/L. Como muestra, la figura 3.2 es un diagrama de transiciones completo para una máquina de Turing con símbolos de cinta $\{a, b, \triangle\}$ que mueve su cabeza hacia la derecha hasta encontrar un espacio en blanco.

Con base en nuestras experiencias con otros autómatas, no debe sorprender que muchas veces resulta difícil manejar y comprender un diagrama de transiciones completo para una máquina de Turing. Por esto, en ocasiones dibujamos esqueletos de los diagramas, que sólo contienen los arcos pertinentes para el análisis, dándose por sentado que, de ser necesario, podrían añadirse los demás arcos.

Como pronto veremos, la "máquina" computacional original de Turing era una computación humana con lápiz y papel. Con la llegada de los equipos de cálculo electrónico, este modelo dio lugar al concepto de máquina electromecánica que lee y escribe en una cinta magnética. Lo importante es que

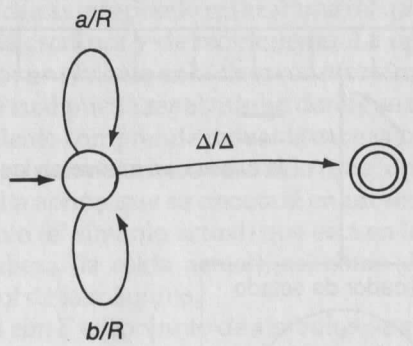


Figura 3.2 Diagrama de transiciones para una máquina de Turing

las capacidades computacionales teóricas del sistema son las mismas, sin importar la tecnología con la cual se implanten sus componentes. De hecho, el concepto de máquina de Turing, al igual que los otros autómatas que hemos estudiado, es independiente de su implantación. Entonces, aislaremos las características definitivas de una máquina de Turing en la siguiente definición formal.

Una máquina de Turing es una séxtupla de la forma $(S, \Sigma, \Gamma, \delta, \iota, h)$, donde:

a. S es una colección finita de estados.
b. Σ es un conjunto finito de símbolos distintos de espacio en blanco, llamado alfabeto de la máquina.
c. Γ es un conjunto finito de símbolos, incluidos los de Σ, que se conocen como símbolos de la cinta de la máquina.
d. δ es la función de transición de la máquina.
e. ι es un elemento de S llamado estado inicial.
f. h es un elemento de S llamado estado de parada.

En ocasiones es conveniente contar con una notación concisa que represente la configuración de la cinta de una máquina de Turing, incluyendo el contenido de sus celdas y la posición de la cabeza. En estos casos presentaremos la lista del contenido de las celdas de la cinta, subrayando la posición de la cabeza. De esta manera, $\triangle x\underline{y}z\triangle\triangle\cdots$ representará una cinta que contiene un espacio en blanco, seguido por los símbolos x, y y z, seguidos a su vez por espacios en blanco, con la cabeza sobre la celda que contiene la z.

Los orígenes de las máquinas de Turing

El propósito para el cual se desarrollaron las máquinas de Turing es distinto del de los otros autómatas que hemos estudiado, ya que se diseñaron para contener todo el poder de los procesos computacionales. En otras palabras, la

intención de Turing fue desarrollar un sistema en el cual fuera posible modelar cualquier proceso que pudiera considerarse como un cálculo.

Recuerde que esto sucedió mucho antes del desarrollo de la maquinaria computacional que existe en la actualidad. Turing pensó en un cálculo realizado por un ser humano con lápiz y papel. Bajo este contexto, Turing razonó que, en un momento dado las personas sólo podrían concentrarse en una porción restringida del papel y que, a su vez, la colección de marcas en este trozo de papel se podía considerar como un solo símbolo. Por esto, Turing consideró dividir el papel en secciones, cada una de las cuales constituía la cantidad de papel requerida para registrar un solo símbolo. Turing también llegó a la conclusión de que cualquier proceso computacional sólo podía implicar un número finito de símbolos. De hecho, puesto que debe ser posible registrar cada símbolo en una cantidad fija de papel, la existencia de una cantidad cada vez mayor de símbolos dictaminaría que los símbolos deberían tener características distintivas arbitrariamente pequeñas. Entonces, se llegaría a un punto donde comenzaría a fallar la capacidad humana para distinguir los símbolos, lo que daría como resultado un número limitado de símbolos efectivos.

Turing planteó que al considerar una sección específica del papel, una persona podría alterar dicha sección o pasar a otra. La acción por emprender y sus detalles dependerían del símbolo existente en la sección y del estado de la mente de la persona. Turing concluyó que, al igual que con el número de símbolos, los seres humanos poseían, sólo una cantidad finita de estados de la mente distinguibles. Turing consideró también que la persona se hallaría en un estado especial inicial al comenzar los cálculos y en un estado designado como de parada al completar los cálculos.

Para evitar que la disponibilidad de papel restringiera el poder del modelo, Turing propuso que la cantidad de papel disponible para los cálculos fuera limitada.

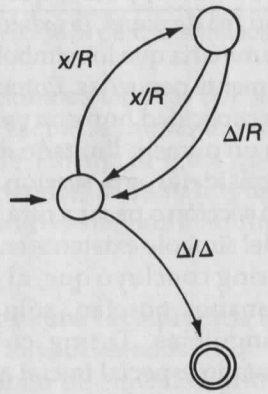
Es muy significativo que nadie haya podido producir un modelo computacional ampliamente aceptado que supere el poder del modelo de Turing. Este modelo es incluso más general que los computadores actuales, ya que una máquina de Turing nunca se ve restringida por la carencia de espacio de almacenamiento, algo que finalmente debe ocurrir en una máquina real. Por esto, la mayoría de los científicos de la computación aceptan la **tesis de Turing:** *el poder computacional de una máquina de Turing es tan grande como el de cualquier sistema computacional posible.* Además, esta tesis y los resultados que la apoyan hoy en día tienen importantes implicaciones con respecto a las dudas sobre la computabilidad. Para resolver cualquier problema con un computador, se requiere desarrollar un proceso computacional (o un algoritmo) que resuelva el problema. Entonces, las respuestas a las preguntas que se planteaban en la época de Turing, como "¿qué puede hacerse con un proceso computacional?" nos ayudan a responder las preguntas actuales, como "¿qué puede hacer un computador moderno?".

En el capítulo 4 volveremos a hablar de la función de las máquinas de Turing en el estudio de los procesos computacionales, además de investigar

la relación entre estas nociones abstractas y el diseño de lenguajes de programación. Sin embargo, en este capítulo continuaremos con la presentación de una técnica útil para construir máquinas de Turing y luego veremos la relación entre máquinas de Turing, gramáticas y lenguajes.

Ejercicios

1. ¿Con qué configuración de cinta se detendrá la máquina de Turing que se presenta a continuación si inicia con la cinta configurada como $\underline{x}xx\triangle\triangle\triangle\cdots$?



2. Diseñe una máquina de Turing con símbolos de cinta x, y y $\triangle$ que busque en la cinta el patrón $xyxy$ y se detenga si y sólo si encuentra ese patrón.

3. Diseñe una máquina de Turing que, al iniciar con la cabeza sobre la celda del extremo izquierdo de la cinta, tenga una terminación anormal si y sólo si hay una x registrada en algún lugar de la cinta. Si aplicara su máquina a una cinta que no contiene una x, ¿detectaría la máquina esta situación?

4. Muestre que las capacidades computacionales de una máquina de Turing no aumentarían si se permitiera que tuvieran más de un estado de parada, ya que esta máquina se podría simular con una máquina de Turing con un solo estado de parada.

3.2 CONSTRUCCIÓN MODULAR DE MÁQUINAS DE TURING

Aunque el propósito principal de este capítulo es estudiar la capacidad de aceptación de lenguajes por parte de las máquinas de Turing, un efecto

secundario importante es la presentación de los fundamentos de las máquinas de Turing, ya que estas máquinas servirán como herramientas en los capítulos subsecuentes. Por esto investigaremos las máquinas de Turing con mayor detenimiento antes de continuar con nuestro estudio de los lenguajes. En esta sección, nuestro objetivo es desarrollar técnicas por medio de las cuales puedan construirse máquinas de Turing complejas a partir de bloques elementales. Este enfoque facilitará la construcción y la comprensión de las máquinas que veremos más adelante.

Combinación de máquinas de Turing

Aunque hablaremos en términos de la combinación de máquinas de Turing para formar otras máquinas de Turing mayores, nuestra estrategia será en realidad combinar los programas de las máquinas de Turing en una forma muy semejante a como se combinan módulos de programas para desarrollar grandes sistemas de software. Para esto, es útil representar los programas de las máquinas de Turing por medio de diagramas de transiciones y combinarlos de manera parecida a lo que hicimos para formar la unión y la concatenación de autómatas finitos.

Suponga que tenemos dos máquinas de Turing, M_1 y M_2, con diagramas de transiciones T_1 y T_2, respectivamente, y símbolos de cinta del conjunto Γ. Si queremos desarrollar un diagrama de transiciones para otra máquina que simule las actividades de M_1 seguidas por las de M_2, bastaría con eliminar la designación de parada del estado de parada de T_1 y la característica de inicio del estado inicial de T_2, y luego dibujar una arco con etiqueta x/x para cada x en Γ, del antiguo estado de parada de T_1 al antiguo estado inicial de T_2.

Obviamente, el resultado se comportaría de la manera esperada, pero este sencillo enfoque tiene varias desventajas. Por ejemplo, en potencia podría introducirse un gran número de transiciones que fueran simples reescrituras del contenido de la celda (esta deficiencia presentaría obstáculos para nuestro análisis de la complejidad en el capítulo 5). Además, suponga que queremos que la máquina compuesta se detenga después de simular M_1, a menos que el símbolo actual, al llegar al estado de parada, sea z, en cuyo caso nos gustaría que la nueva máquina simulara las acciones de M_2; o suponga que necesitamos combinar tres diagramas y controlar el paso de uno a otro dependiendo del valor del símbolo actual al llegar a cada uno de los antiguos estados de parada. En estos casos se requiere un enlace un poco más complicado.

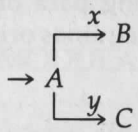
Suponga entonces que necesitamos combinar los diagramas de transiciones de varias máquinas de Turing para obtener una máquina que simule alguna combinación de las máquinas originales. Lo hacemos de la siguiente manera:

1. Elimine la característica de inicio de los estados iniciales de todas las máquinas, excepto la de aquél donde iniciará la máquina compuesta.

2. Elimine la característica de detención de los estados de parada de todas las máquinas e introduzca un nuevo estado de parada que no se encuentre en ninguno de los diagramas que se combinan.
3. Para cada uno de los antiguos estados de parada p y cada x en Γ, dibuje un arco de la siguiente forma:
   a. Si la máquina compuesta debe detenerse al llegar a p con el símbolo actual x, dibuje un arco con etiqueta x/x de p al nuevo estado de parada.
   b. Si al llegar al estado p con el símbolo actual x, la máquina compuesta debe transferir el control a la máquina $M = (S, \Sigma, \Gamma, \delta, \iota, h)$, dibuje entonces un arco con etiqueta x/z de p al estado q de M, donde $\delta(\iota, x) = (q, z)$.

La figura 3.3 proporciona un ejemplo de esta construcción. Aquí hemos comenzado con los diagramas de transiciones para tres máquinas distintas, cada una con los símbolos de cinta x, y y $\triangle$. Una mueve su cabeza una celda a la derecha, otra encuentra la primera x a la derecha de la celda actual, y la tercera encuentra la primera y a la derecha de la celda actual. Con el proceso de composición antes mencionado y empleando estas máquinas como bloques de construcción, hemos construido una máquina compuesta que encuentra la segunda ocurrencia del símbolo distinto de espacio en blanco que se halla a la derecha de la posición inicial de la cabeza.

Es conveniente evitar los detalles interiores de los bloques de construcción a partir de los cuales construimos máquinas de Turing más complejas, como sucede en el caso de grandes sistemas de software. Si sabemos cuál es la tarea que realiza cada una de las máquinas más pequeñas, entonces nuestra preocupación será el adecuado enlace de estas máquinas y no cómo cada una lleva a cabo su tarea específica. Por esto, evitaremos el uso de diagramas de transiciones detallados como los que se muestran en la figura 3.3, y en vez de esto utilizaremos diagramas compuestos. En estos diagramas cada uno de los bloques de construcción se representa como un nodo, con flechas entre los nodos para indicar las transiciones entre bloques. Estas flechas se etiquetan de acuerdo con el valor que debe aparecer en la celda actual para que se recorra la flecha. Es decir, si una flecha del nodo A al nodo B tiene una etiqueta x, entonces la ejecución se transferirá a la máquina B si A llega a su antiguo estado de parada con una x en la celda actual. Como hicimos con los diagramas de transiciones, debe indicarse con un apuntador el nodo del diagrama compuesto donde debe comenzar la ejecución. Por ejemplo, la máquina compuesta presentada en la figura 3.3 podría resumirse con el diagrama compuesto



donde el nodo A representa la máquina que mueve su cabeza una celda a la derecha, B la máquina que busca una x y C la máquina que busca una y.

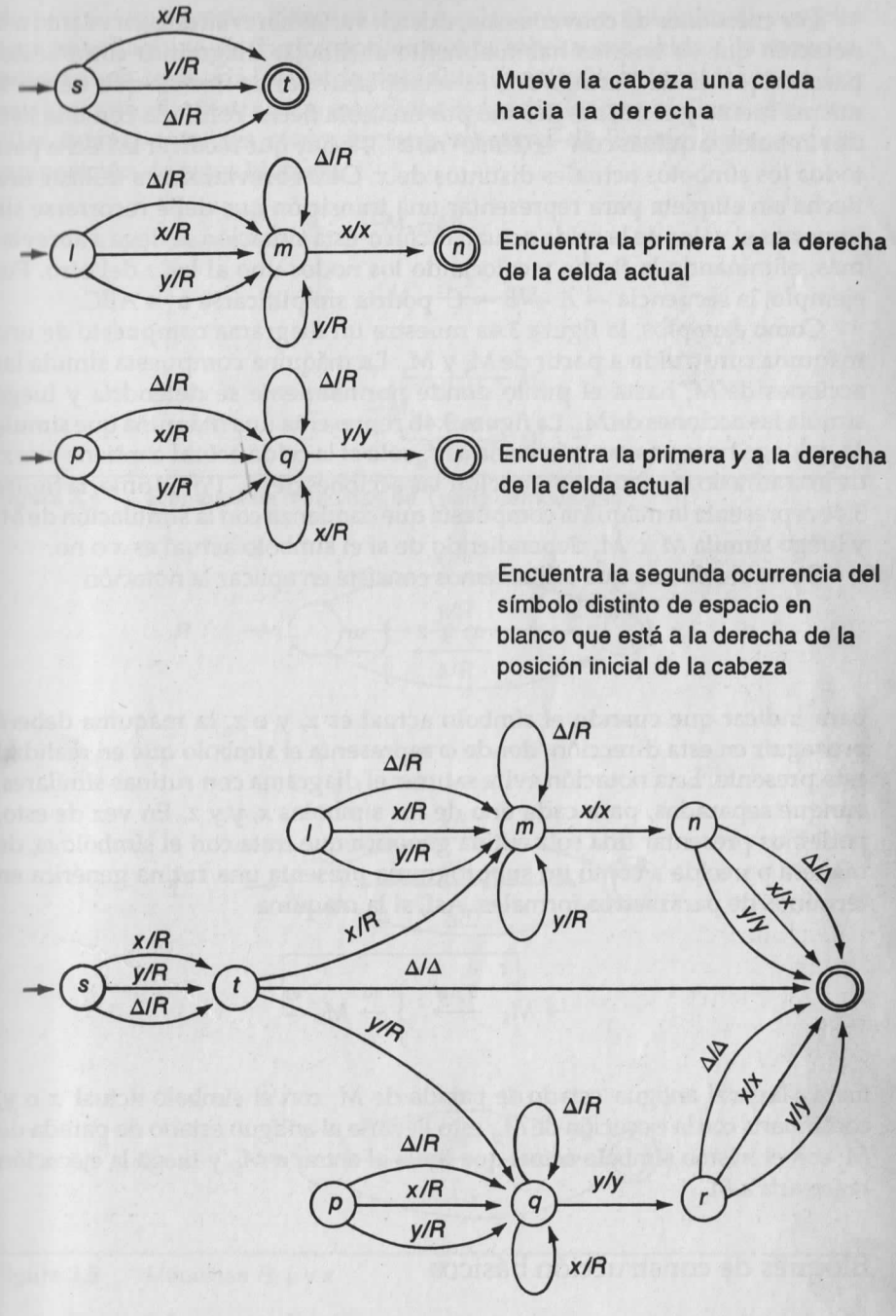


Figura 3.3 Construcción de una máquina de Turing compuesta a partir de máquinas más pequeñas

Por cuestiones de conveniencia, existen varias abreviaturas en cuanto a la notación que se emplea habitualmente al dibujar diagramas compuestos para máquinas de Turing. Uno es reemplazar varias flechas que tienen la misma fuente y el mismo destino por una sola flecha rotulada con una lista de símbolos, o quizás con $\neg x$ (léase "no x"), si hay que recorrer la flecha para todos los símbolos actuales distintos de x. Otra abreviatura es utilizar una flecha sin etiqueta para representar una transición que debe recorrerse sin importar el valor de la celda actual. Incluso esta situación se llega a abreviar más, eliminando la flecha y colocando los nodos uno al lado del otro. Por ejemplo, la secuencia $\rightarrow A \rightarrow B \rightarrow C$ podría simplificarse a $\rightarrow ABC$.

Como ejemplos, la figura 3.4a muestra un diagrama compuesto de una máquina construida a partir de M_1 y M_2. La máquina compuesta simula las acciones de M_1 hasta el punto donde normalmente se detendría y luego simula las acciones de M_2. La figura 3.4b representa una máquina que simula M_1 y luego hace una transferencia a M_2 sólo si la celda actual contiene una x; de lo contrario, se detiene al concluir las acciones de M_1. Por último, la figura 3.4c representa la máquina compuesta que comienza con la simulación de M_1 y luego simula M_2 o M_3 dependiendo de si el símbolo actual es x o no.

Otra abreviatura que utilizaremos consiste en aplicar la notación

$$\left.\xrightarrow{x,\ y,\ z}\right\}\xrightarrow{\omega}$$

para indicar que cuando el símbolo actual es x, y o z, la máquina deberá proseguir en esta dirección, donde ω representa el símbolo que en realidad está presente. Esta notación evita saturar el diagrama con rutinas similares, aunque separadas, para cada uno de los símbolos x, y y z. En vez de esto, podemos presentar una sola rutina genérica que trata con el símbolo ω, de manera parecida a como un subprograma presenta una rutina genérica en términos de parámetros formales. Así, si la máquina

$$\rightarrow M_1 \left.\xrightarrow{x,\ y}\right\}\xrightarrow{\omega} M_2 \xrightarrow{\omega}$$

fuera a llegar al antiguo estado de parada de M_1 con el símbolo actual x o y, continuaría con la ejecución de M_2; esto llevaría al antiguo estado de parada de M_2 con el mismo símbolo actual que tenía al entrar a M_2, y luego la ejecución regresaría a M_1.

Bloques de construcción básicos

Una vez establecido este sistema de notación, consideremos ahora las máquinas de Turing elementales que emplearemos como bloques de construcción en análisis posteriores. Obtendremos una pista de cuáles deberán ser estos

bloques de construcción si recordamos que las únicas actividades disponibles para una máquina de Turing son mover la cabeza una celda a la derecha, moverla una celda a la izquierda y escribir un símbolo en la celda actual. Por consiguiente, si construimos máquinas individuales que realizan estas sencillas tareas, entonces cualquier otra máquina de Turing debe ser una composición de estos bloques.

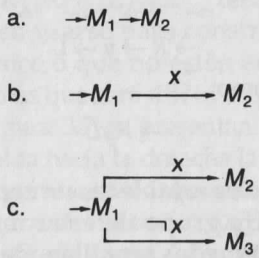


Figura 3.4 Ejemplos de máquinas de Turing compuestas

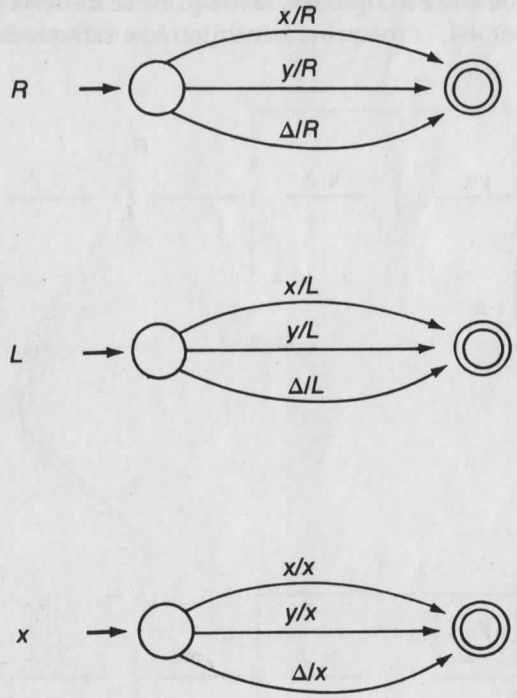


Figura 3.5 Máquinas *R*, *L* y *x*

La figura 3.5 muestra los diagramas de transiciones de las máquinas de Turing que efectúan cada una de estas actividades rudimentarias, suponiendo que los símbolos de la cinta son x, y y $\triangle$ (las máquinas para otros

símbolos no son más que simples generalizaciones de estas máquinas). Representamos con R la máquina que mueve su cabeza una celda a la derecha, con L la que la mueve una celda a la izquierda y con x la máquina que escribe un símbolo en la celda actual. De esta manera, una máquina de Turing que se mueve una celda a la derecha, escribe el símbolo y, se mueve una celda hacia la izquierda y se detiene, podría representarse con el diagrama compuesto

$$\rightarrow R \rightarrow y \rightarrow L$$

o, en forma condensada, como

$$\rightarrow RyL$$

Nuestro siguiente paso es establecer un repertorio de máquinas ligeramente más complejas. Un grupo de estas máquinas, presentado en la figura 3.6, lleva a cabo búsquedas sencillas. De manera más precisa, para cualquier símbolo x la máquina representada por R_x recorre la cinta a la derecha de su posición inicial en busca de una celda que contenga el símbolo x. Si encuentra ese símbolo, la máquina se detiene y esa celda será la actual; de no ser así, la máquina continuará su búsqueda eternamente.

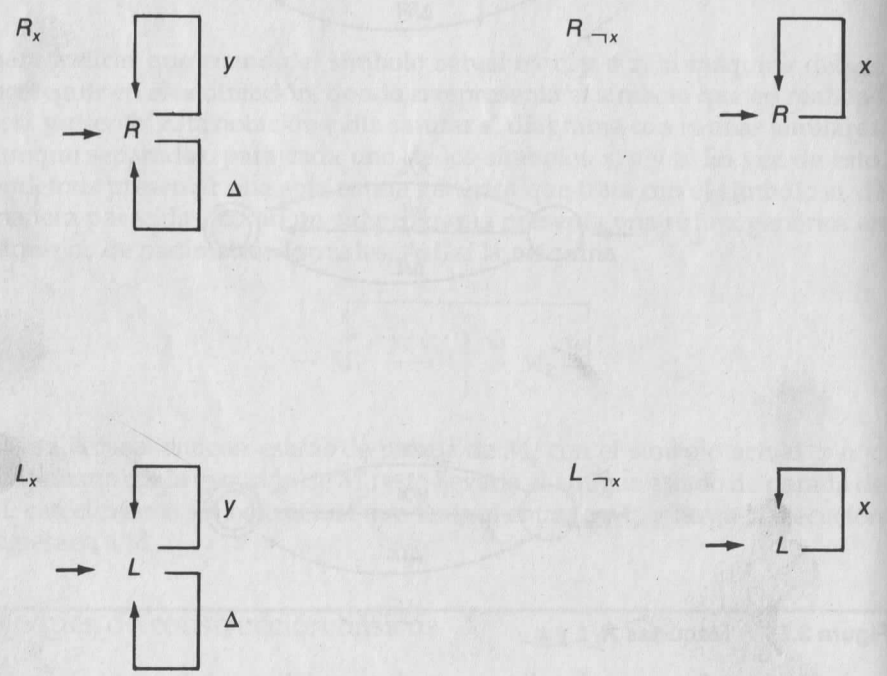


Figura 3.6 Máquinas R_x, $R_{\neg x}$, L_x y $L_{\neg x}$

De modo similar, la máquina $R_{\neg x}$ buscará a la derecha de la posición inicial cualquier símbolo que no sea x. Las máquinas L_x y $L_{\neg x}$ efectúan estas mismas búsquedas hacia la izquierda de la posición inicial (observe que, a diferencia de la búsqueda hacia la derecha desde la posición inicial, la búsqueda hacia la izquierda puede ocasionar una terminación anormal si llega al extremo izquierdo de la cinta sin encontrar el objetivo de la búsqueda). Las máquinas R_Δ, $R_{\neg\Delta}$, L_Δ y $L_{\neg\Delta}$ tendrán una utilidad especial para nosotros, pues pueden usarse para construir máquinas compuestas que busquen celdas en blanco o que no estén en blanco.

Otra colección de máquinas que será útil es la que lleva a cabo operaciones de desplazamiento; en la figura 3.7 se presentan dos de estas máquinas. La máquina S_R desplaza una celda hacia la derecha la cadena de símbolos que no están en blanco y que se encuentran a la izquierda de la celda actual. Así, si la cinta de S_R tuviera la configuración inicial $\Delta xyy\underline{x}x\Delta\Delta\Delta\Delta\cdots$, entonces S_R se detendría con la cinta configurada como $\Delta\Delta xyy\underline{x}\Delta\Delta\Delta\cdots$; o, si se aplicara a $\Delta yx\underline{y}\Delta\Delta xy\Delta\Delta\cdots$, S_R produciría $\Delta\Delta yx\underline{y}\Delta xx\underline{y}\Delta\Delta\cdots$. Como un caso algo especial, si la celda a la izquierda de la actual contiene un espacio en blanco, S_R únicamente borraría la celda actual. Así, S_R convertiría la configuración $xy\Delta\underline{y}x\Delta\Delta\Delta\cdots$ de la cinta a $xy\Delta\underline{\Delta}x\Delta\Delta\Delta\cdots$.

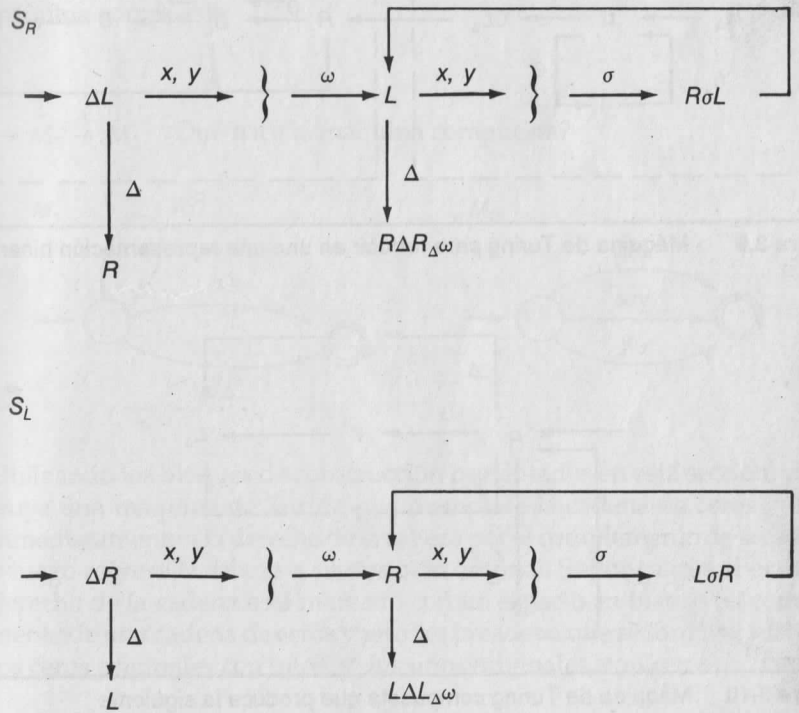


Figura 3.7 Máquinas S_R y S_L

En forma parecida, S_L realiza un desplazamiento hacia la izquierda. Si la cinta de S_L tuviera la configuración inicial $\triangle\underline{x}yyxx\triangle\triangle\triangle\cdots$, entonces S_L se detendría con su cinta configurada como $\triangle\underline{y}yxx\triangle\triangle\triangle\triangle\cdots$; o si se aplicara a $\triangle yxy\triangle\underline{\triangle}xxy\triangle\triangle\cdots$, S_L produciría $\triangle yxy\triangle\underline{x}xy\triangle\triangle\triangle\cdots$.

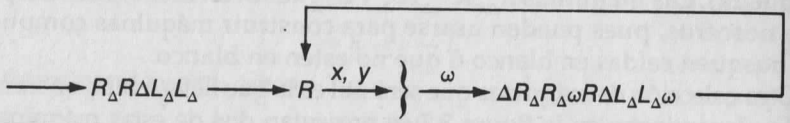


Figura 3.8 Máquina copiadora que transforma un patrón de la forma $\underline{\triangle}w\triangle$ $\triangle w\underline{\triangle}w\triangle$

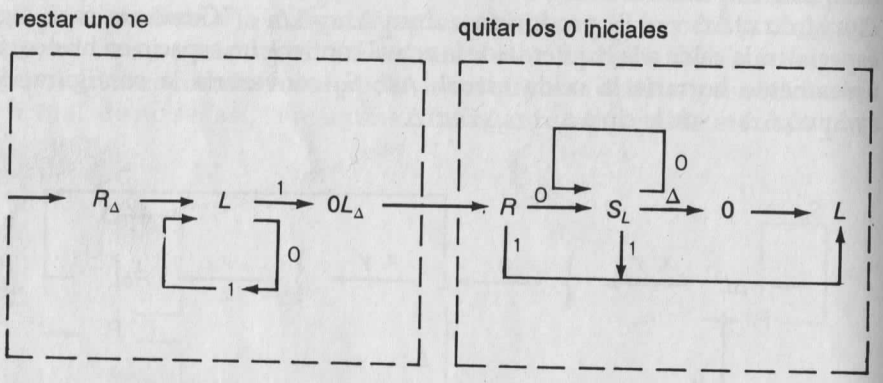


Figura 3.9 Máquina de Turing para reducir en uno una representación binaria

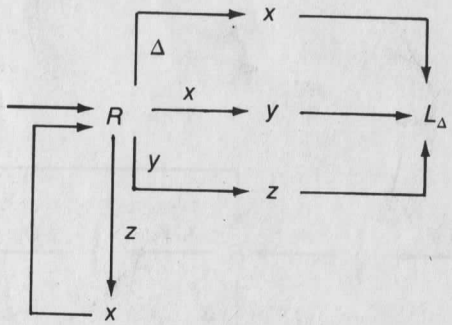
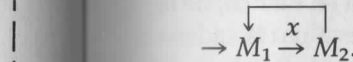


Figura 3.10 Máquina de Turing compuesta que produce la siguiente cadena en la secuencia λ, *x, y, z, xx, yx, zx, xy, yy, zy, xz, yz, zz, xxx, yxx,* ···

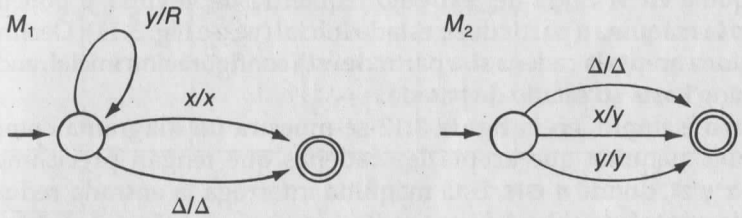
Concluimos con varios ejemplos de cómo pueden combinarse las máquinas elementales presentadas hasta ahora para formar máquinas compuestas más complejas. La figura 3.8 define una máquina copiadora que transforma un patrón de la forma $\underline{\triangle}w\triangle$ a la forma $\triangle w\underline{\triangle}w\triangle$, donde w es cualquier cadena (posiblemente de longitud cero) de símbolos que no son espacios en blanco. La máquina que se presenta en la figura 3.9 supone que su entrada representa un entero positivo en notación binaria y reduce en uno el valor representado. Por último, la máquina de la figura 3.10 modifica cadenas del alfabeto $\{x, y, z\}$. Específicamente, si iniciamos la máquina con su cinta configurada como $\underline{\triangle}w_1\triangle\triangle\triangle\cdots$, donde w_1 es una cadena en $\{x, y, z\}^*$, entonces la máquina se detendrá con la cinta configurada como $\underline{\triangle}w_2\triangle\triangle\triangle\cdots$, donde w_2 es la cadena que sigue a w_1 en la secuencia λ, *x, y, z, xx, yx, zx, xy, yy, zy, xz, yz, zz, xxx, yxx,* ···. Más adelante veremos la utilidad de esta máquina como bloque de construcción.

Ejercicios

1. Combine los diagramas de transiciones de las máquinas M_1 y M_2 que se muestran a continuación para formar el diagrama de transiciones de la máquina compuesta



¿Qué hace la máquina compuesta?



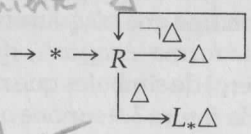
2. Utilizando los bloques de construcción presentados en esta sección, construya una máquina de Turing que reemplace la cadena de ceros y unos inmediatamente a la derecha de la cabeza por el complemento de la cadena y luego regrese la cabeza a su posición original. Suponga que el extremo derecho de la cadena está marcado con un espacio en blanco (el complemento de una cadena de ceros y unos es la cadena que se forma al sustituir los ceros originales con unos, y los unos originales, con ceros).

3. ¿En qué condiciones se presentaría una terminación anormal durante la ejecución de la máquina S_R?

4. Describa los cálculos efectuados por la siguiente máquina de Turing.

borra hasta el siguiente Δ

ej: ΔabnxΔmaΔΔ



ΔΔaΔΔΔΔma ΔΔ ✓

3.3 MÁQUINAS DE TURING COMO ACEPTADORES DE LENGUAJES

Hemos presentado a los autómatas de los capítulos anteriores como aceptadores de lenguajes y los hemos utilizado para evaluar cadenas y determinar su pertenencia a un lenguaje específico. Por lo tanto, es natural que estudiemos a las máquinas de Turing desde la misma perspectiva. En esta sección analizaremos los aspectos de la aceptación de cadenas realizada por máquinas de Turing; empezaremos considerando el contexto en el cual una máquina de Turing acepta una cadena de entrada.

Procedimientos de evaluación de cadenas

Para evaluar una cadena de algún alfabeto Σ con una máquina de Turing, registramos la cadena en la cinta (que de otra manera estaría en blanco) de la máquina, comenzando por la segunda celda (si fuéramos a evaluar la cadena *xxyy*, la cinta aparecería como $\triangle xxyy\triangle\triangle\triangle\cdots$). Luego colocamos la cabeza de la máquina en la celda del extremo izquierdo de la cinta y ponemos en marcha la máquina a partir de su estado inicial (véase Fig. 3.11). Decimos que la máquina acepta la cadena si, a partir de esta configuración inicial, encuentra su camino hasta su estado de parada.

Como ejemplo, en la figura 3.12 se muestra un diagrama compuesto para una máquina que acepta las cadenas que tengan precisamente la forma $x^n y^n z^n$, donde $n \in \mathbb{N}$. Está máquina interroga la entrada reduciendo repetidamente la longitud de una cadena no vacía de la forma $x^n y^n z^n$ a través de la secuencia $x^{n-1}y^n z^n$, $x^{n-1}y^{n-1}z^n$ y $x^{n-1}y^{n-1}z^{n-1}$. La máquina se detiene si y sólo si este proceso de reducción produce una cadena vacía como resultado de la eliminación del mismo número de *x*, *y* y *z* (se emplea # como símbolo de cinta para ayudar a encontrar el extremo izquierdo de la cinta después de completar cada secuencia de reducciones).

Al igual que con los demás autómatas, la colección de cadenas aceptadas por una máquina de Turing *M* se llama lenguaje aceptado por la máquina y se representa con *L*(*M*). Se dice que un lenguaje *L* es un **lenguaje aceptado por una máquina de Turing** si existe una máquina de Turing *M* tal que *L* = *L*(*M*).

La figura 3.12 tiene una relevancia especial para nuestros fines, pues muestra que las máquinas de Turing son capaces de aceptar lenguajes que no

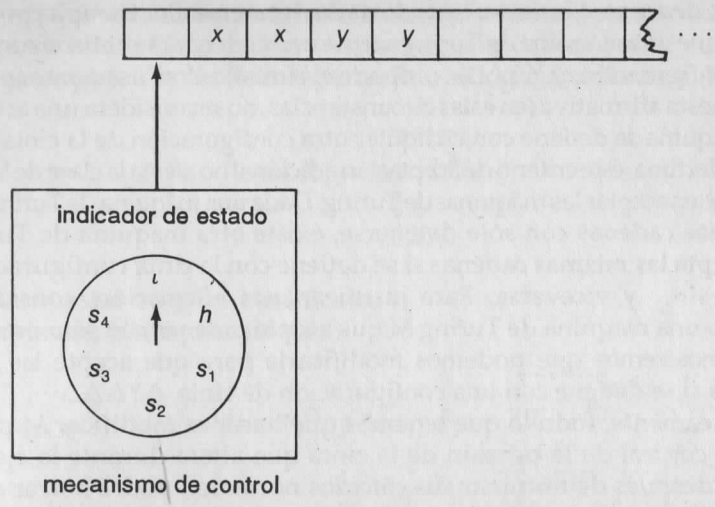


Figura 3.11 Configuración inicial de una máquina de Turing al evaluar la cadena *xxyy*.

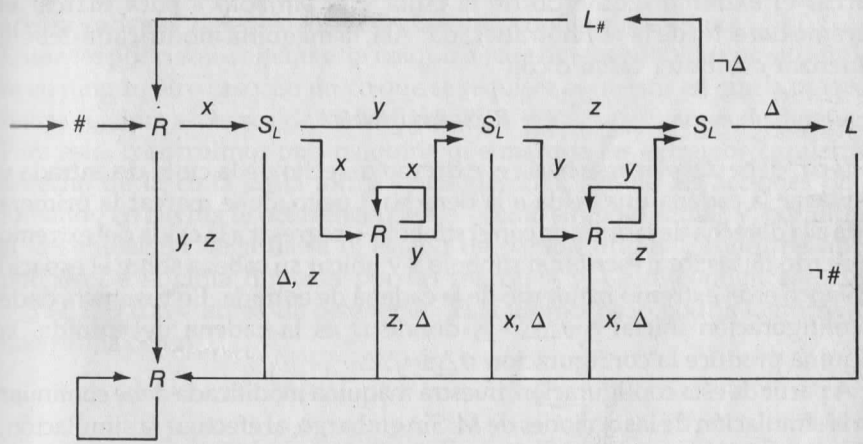


Figura 3.12 Máquina de Turing *M* para la cual $L(M) = \{x^n y^n z^n : n \in \mathbb{N}\}$

pueden aceptar los autómatas de pila (recuerde que $\{x^n y^n z^n : n \in \mathbb{N}\}$ no es un lenguaje independiente del contexto). De hecho, encontraremos que la clase de los lenguajes aceptados por máquinas de Turing contiene propiamente todos los lenguajes independientes del contexto, aunque requeriremos un poco de trabajo preliminar para demostrarlo.

Hemos definido la aceptación de cadenas de las máquinas de Turing de una manera que se ajusta a la de los demás autómatas. Sin embargo, en ocasiones es conveniente requerir que una máquina de Turing escriba un

mensaje de aceptación en su cinta antes de detenerse. Por ejemplo, podríamos desear que una máquina de Turing acepte una cadena si se detiene con su cinta en la configuración $\underline{\triangle}Y\triangle\triangle\triangle\cdots$, donde el símbolo Y se usa para representar la respuesta afirmativa (en estas circunstancias, no se considera una aceptación si la máquina se detiene con cualquier otra configuración de la cinta).

Por fortuna, este criterio de aceptación adicional no afecta la clase de lenguajes que pueden aceptar las máquinas de Turing. Dada una máquina de Turing M que acepte las cadenas con sólo detenerse, existe otra máquina de Turing M' que acepta las mismas cadenas si se detiene con la cinta configurada como $\underline{\triangle}Y\triangle\triangle\triangle\cdots$, y viceversa. Para justificar esta afirmación, consideremos primero una máquina de Turing M que acepte cadenas con sólo detenerse y luego mostremos que podemos modificarla para que acepte las mismas cadenas si se detiene con una configuración de cinta $\underline{\triangle}Y\triangle\triangle\triangle\cdots$.

Básicamente, todo lo que tenemos que hacer es modificar M para que lleve el control de la porción de la cinta que altera durante la ejecución. Luego, después de terminar sus cálculos normales, podrá borrar esa porción de la cinta y escribir el mensaje adecuado en la cinta en blanco antes de detenerse. Para llevar este control de la porción alterada de la cinta, utilizamos los símbolos de cinta especiales # y $*$. El símbolo # se usará para marcar el extremo izquierdo de la cinta y el símbolo $*$ para marcar el extremo derecho de la porción alterada. Así, la máquina modificada deberá comenzar cualquier cálculo con

$$\rightarrow R_{\triangle}S_R R{*}L_{\triangle}L\#R$$

Es decir, debe desplazarse hacia el extremo derecho de la cinta de entrada y desplazar la cadena una celda a la derecha. Luego, debe marcar la primera celda a la derecha de la entrada con el símbolo $*$, regresar a la celda del extremo izquierdo de la cinta, escribir el símbolo # y ubicar su cabeza sobre el espacio en blanco en el extremo izquierdo de la cadena de entrada. En resumen, dada la configuración inicial $\underline{\triangle}w\triangle\triangle\cdots$, donde w es la cadena de entrada, la máquina produce la configuración $\#\underline{\triangle}w{*}\triangle\triangle\cdots$.

A partir de esta configuración, nuestra máquina modificada debe continuar con la simulación de las acciones de M. Sin embargo, al efectuar la simulación, la máquina debe estar pendiente de la ocurrencia de dos circunstancias especiales. La primera ocurre si el símbolo # se convierte en el símbolo actual. Observe que, como toda la entrada se ha desplazado una celda a la derecha, no se leerá la celda que contiene # durante la simulación, a menos que el proceso simulado sufra una terminación anormal. Entonces, si el símbolo actual fuera #, nuestra máquina modificada debería mover su cabeza una celda más hacia la izquierda para que también experimentara una terminación anormal.

La otra circunstancia especial es la ocurrencia del símbolo $*$ como símbolo actual. Esta situación indica que el cálculo simulado ha movido su cabeza a la derecha, hacia una celda de la cinta que no se había alterado antes. En este caso, nuestra máquina modificada debe empujar la marca $*$ una celda a la derecha ejecutando $\rightarrow R{*}L\triangle$ antes de continuar con la simulación.

Una vez que se han simulado los cálculos de M hasta el punto donde M ha llegado a su estado de parada, nuestra máquina deberá borrar su cinta y escribir el mensaje Y antes de detenerse. Esto se logra añadiendo el bloque

$$\begin{array}{l} \rightarrow R_* \rightarrow \triangle L \begin{array}{l} \downarrow \\ \neg\# \end{array} \\ \quad\quad\quad\quad \downarrow\# \\ \quad\quad\quad\quad \triangle RYL \end{array}$$

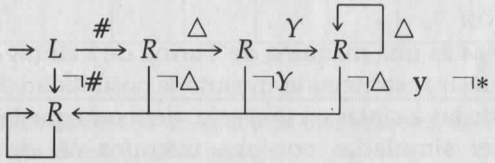
al final de nuestra máquina modificada.

En resumen, la máquina final es

$$\rightarrow R_{\triangle}S_R R{*}L_{\triangle}L\#R \rightarrow M_0 \rightarrow R_* \rightarrow \triangle L \begin{array}{l} \downarrow \\ \neg\# \end{array}$$
$$\downarrow\#$$
$$\triangle RYL$$

donde M_0 es la máquina que simula M, excepto por las dos circunstancias especiales en los cuales surge # o $*$ como el símbolo actual.

A la inversa, suponga que comenzamos con una máquina de Turing M' que acepta cadenas deteniéndose con la cinta configurada como $\underline{\triangle}Y\triangle\triangle\triangle\cdots$. Entonces podríamos alterar esta máquina para obtener otra que no se detuviera en ningún otro caso. Lo único que se requiere es insistir en que la máquina revise su cinta en busca de la configuración $\underline{\triangle}Y\triangle\triangle\triangle\cdots$ antes de detenerse. Para esto, construimos otra máquina que marque los extremos izquierdo y derecho de la cinta en la forma antes descrita; simule las acciones de M' tomando en cuenta la ocurrencia de # o $*$ como símbolo actual; y, por último, si la simulación llega hasta el estado de parada original, confirme que la porción de la cinta limitada por las marcas # y $*$ esté configurada como $\#\underline{\triangle}Y\triangle\triangle\cdots\triangle\triangle*$ antes de detenerse. Este último paso podría lograrse con una rutina como



Concluimos que se pueden establecer varias maneras para que una máquina de Turing acepte sus cadenas de entrada. Podemos permitir que simplemente se detengan o insistir en que respondan con un mensaje de aceptación. Ambas estrategias tienen sus ventajas y desventajas pero, ya que cuentan con el mismo poder, tenemos la libertad de elegir el método que más convenga para la aplicación. Sin embargo, supondremos que una máquina de Turing acepta sus cadenas con sólo detenerse, a menos que se especifique lo contrario.

Máquinas de Turing de varias cintas

Ahora consideraremos las máquinas de Turing que tienen más de una cinta (este tipo de máquinas se conoce como máquinas de Turing de k cintas, donde k representa un entero positivo, en aquellos casos donde tenga relevancia el número de cintas). Cada una de estas cintas tiene un extremo izquierdo, se extiende indefinidamente hacia la derecha, y el acceso a cada una se logra a través de una cabeza separada de lectura y escritura. La transición que se ejecutará en un instante determinado dependerá de la colección de símbolos presentes para estas cabezas, junto con el estado actual de la máquina. La acción de una sola transición afecta sólo a una de las cintas de la máquina; esta acción puede ser escribir en la celda actual de esa cinta, mover la cabeza correspondiente una celda a la izquierda o moverla una celda a la derecha.

Para evaluar la aceptación de una cadena de símbolos utilizando una máquina de Turing de varias cintas, comenzamos con la máquina en su estado inicial y con la cadena de entrada registrada en la primera cinta (en el mismo formato que tendría en la cinta única de una máquina convencional), a la vez que las otras cintas están en blanco y todas las cabezas se encuentran en la celda del extremo izquierdo de sus cintas. La cadena es aceptada si, a partir de esta configuración inicial, la máquina se detiene.

Uno podría esperar que las máquinas de Turing de varias cintas tuvieran un potencial de procesamiento de lenguajes mayor que sus homólogos de una sola cinta; no obstante, el siguiente teorema muestra que esto no es cierto. Aunque en ocasiones conviene emplear una máquina con varias cintas, estas máquinas no pueden aceptar más lenguajes que las máquinas de Turing convencionales.

TEOREMA 3.1
Para cada máquina de Turing de varias cintas M, hay una máquina de Turing tradicional (de una cinta) M' tal que $L(M) = L(M')$.

DEMOSTRACIÓN
Suponga que M es una máquina de Turing de k cintas que acepta el lenguaje L. Nuestra estrategia es mostrar la posibilidad de representar el contenido de las k cintas en una sola, de modo que las acciones de M puedan ser simuladas por una máquina M' de cinta única. Visualizamos las cintas de M colocadas paralelas entre sí, con los extremos izquierdos alineados, como se presenta en la figura 3.13a, donde un apuntador bajo cada cinta indica la posición de la cabeza. Con base en esta disposición, es posible representar el contenido de las k cintas y las posiciones de sus cabezas en una tabla que contiene $2k$ filas y un número ilimitado de columnas que se extienden hacia la derecha, como se muestra en la figura 3.13b. Las filas impares de esta tabla representan las cintas; las pares se usan para indicar la posición

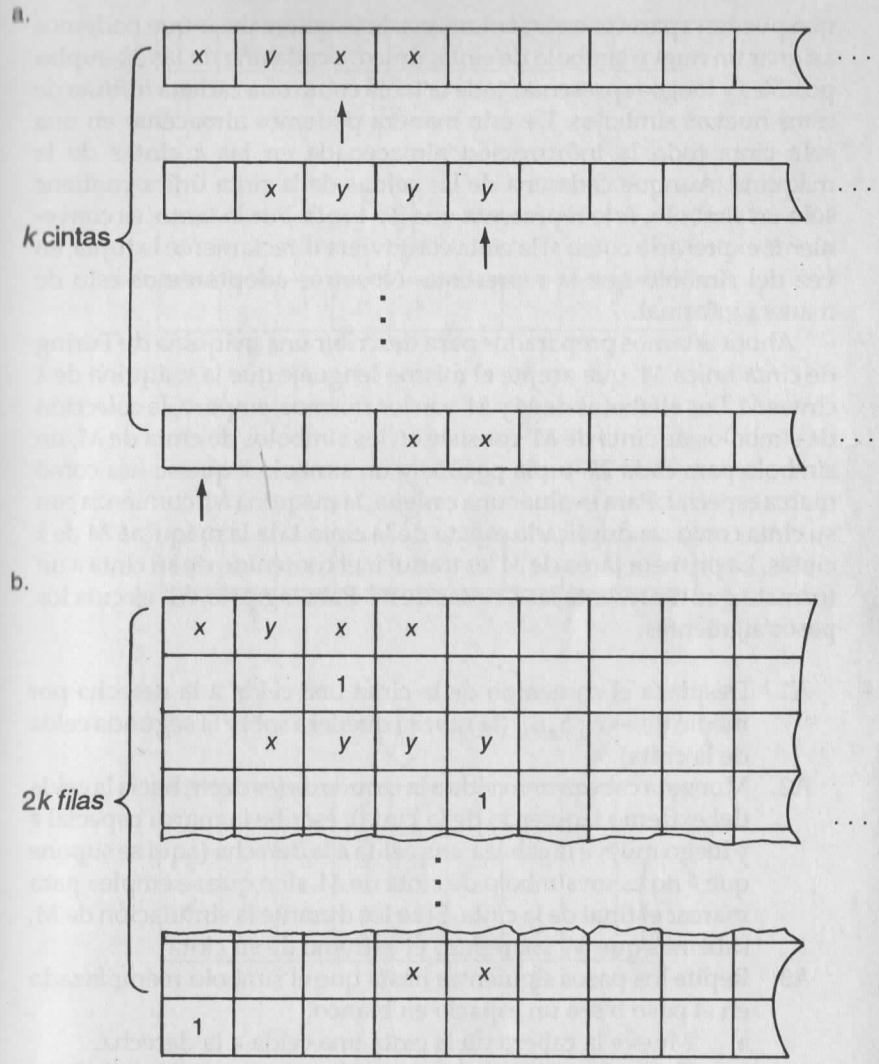


Figura 3.13 Representación de las cintas de una máquina de Turing de k cintas con un solo arreglo

de la cabeza de la cinta de la fila anterior, lo cual se logra colocando un 1 en la columna asociada a la celda actual de la fila superior y dejando en blanco todas las demás celdas.

Observe que cada una de las columnas de la tabla que acabamos de describir es en esencia una $2k$-tupla, siendo los componentes impares símbolos de cinta de M y los pares elementos de $\{\triangle, 1\}$. Por lo tanto, existe sólo un número finito de combinaciones distintas de símbolos

que pueden aparecer en las columnas. Esto quiere decir que podemos asignar un nuevo símbolo de cinta, único, a cada una de las $2k$–tuplas posibles y luego representar toda la tabla como una cadena infinita de estos nuevos símbolos. De esta manera podemos almacenar en una sola cinta toda la información almacenada en las k cintas de la máquina. Aunque cada una de las celdas de la cinta única contiene sólo un símbolo, éste representa una $2k$–tupla. Por lo tanto, es conveniente expresarlo como si la cinta contuviera directamente la tupla, en vez del símbolo que la representa. Nosotros adoptaremos esto de manera informal.

Ahora estamos preparados para describir una máquina de Turing de cinta única M' que acepte el mismo lenguaje que la máquina de k cintas M. Los alfabetos de M y M' son los mismos; empero, la colección de símbolos de cinta de M' consiste en los símbolos de cinta de M, un símbolo para cada $2k$–tupla posible y un símbolo # que se usa como marca especial. Para evaluar una cadena, la máquina M' comienza con su cinta como un duplicado exacto de la cinta 1 de la máquina M de k cintas. La primera tarea de M' es traducir el contenido de su cinta a un formato que represente las k cintas de M. Para lograrlo, M' ejecuta los pasos siguientes:

A1. Desplaza el contenido de la cinta una celda a la derecha por medio de $\rightarrow R_{\triangle} S_R L_{\triangle}$ (la cabeza quedará sobre la segunda celda de la cinta).

A2. Mueve la cabeza una celda a la izquierda (es decir, hacia la celda del extremo izquierdo de la cinta), escribe la marca especial # y luego mueve la cabeza una celda a la derecha (aquí se supone que # no es un símbolo de cinta de M, sino que se emplea para marcar el final de la cinta. Si se lee durante la simulación de M, sabemos que M sobrepasó el extremo de su cinta).

A3. Repite los pasos siguientes hasta que el símbolo reemplazado en el paso b sea un espacio en blanco.
   a. Mueve la cabeza de la cinta una celda a la derecha.
   b. Reemplaza el símbolo actual, digamos x, con el símbolo de cinta que representa a la tupla $(x, \triangle, \triangle, \triangle, \cdots, \triangle, \triangle)$.

A4. Ejecuta $L_{\triangle}$, lo cual moverá la cabeza de regreso a la segunda celda de la cinta. Sustituye el espacio en blanco de esta celda con el símbolo de cinta que representa a la tupla $(\triangle, 1, \triangle, 1, \cdots, \triangle, 1)$.

Después de ejecutar estos pasos, M' habrá traducido su cinta a un formato de cintas múltiples, marcado el extremo izquierdo de la cinta con el símbolo # y devuelto su cabeza a la tupla que representa las celdas del extremo izquierdo de las k cintas de M (véase Fig 3.14).

La tarea de M' sería ahora simular las acciones de M hasta que M se detenga o termine anormalmente. Por supuesto, para simular una

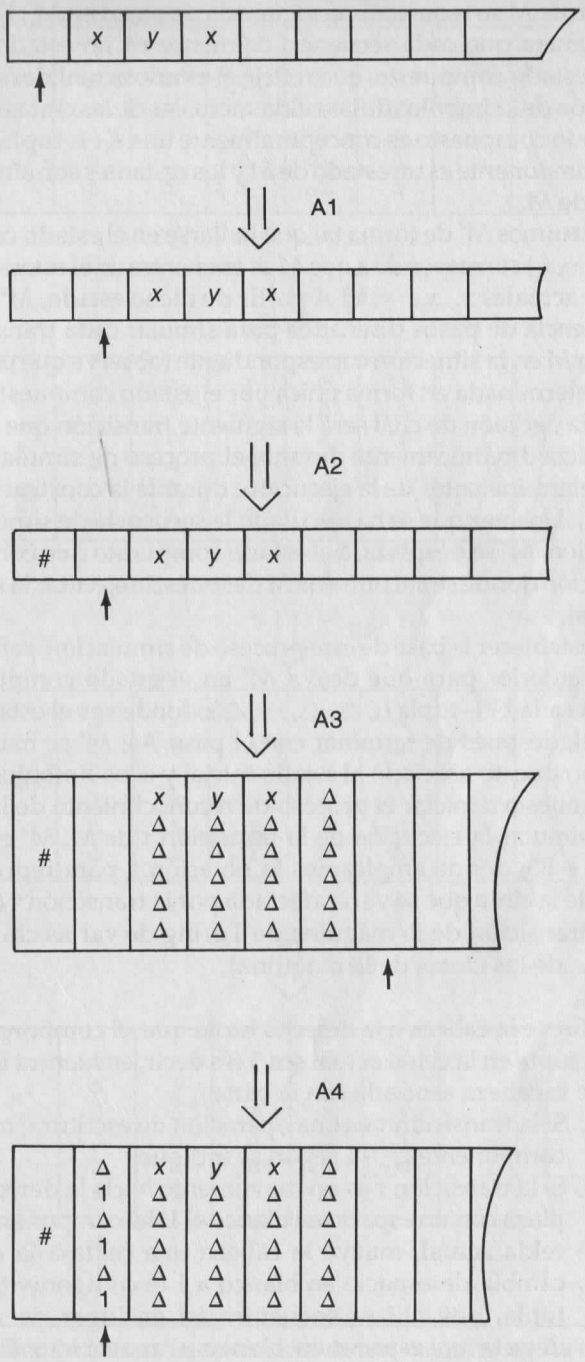


Figura 3.14 Conversión de una cinta única a un formato de tres cintas

transición de M se requiere una secuencia de pasos en M'. Diseñamos M' de manera que cada secuencia comience en un estado especial, llamado estado compuesto, que refleje el estado actual de M así como la colección de k símbolos de las celdas actuales de las cintas de M. Así, cada estado compuesto es conceptualmente una K+1–tupla, donde el primer componente es un estado de M y los restantes son símbolos del alfabeto de M.

Construimos M' de forma tal que hallarse en el estado compuesto $(p, x_1, x_2, \cdots, x_k)$ corresponda a que M se encuentre en el estado p con los símbolos actuales $x_1, x_2, \cdots, x_k$. A partir de dicho estado, M' ejecutará una secuencia de pasos diseñados para simular cada transición que ejecutaría M en la situación correspondiente (observe que esta transición es determinada en forma única por el estado compuesto. De esta manera, la decisión de cuál será la siguiente transición que se ejecute no se realiza dinámicamente durante el proceso de simulación, sino que se determina antes de la ejecución, durante la construcción de la máquina). Una vez que se ha ejecutado la secuencia de simulación de la transición, M' se desplazará al estado compuesto que corresponda a la situación donde se encontraría M después de ejecutar la transición individual.

Para establecer la base de este proceso de simulación, refinamos el paso A4 anterior para que deje a M' en el estado compuesto que representa a la k+1–tupla $(\iota, \triangle, \triangle, \cdots, \triangle)$, donde ι es el estado inicial de M. Así, después de terminar con el paso A4, M' se hallará en el estado compuesto asociado al estado inicial y a los símbolos actuales de M, dispuesta a iniciar el proceso de reconocimiento de la cadena.

Para simular la ejecución de la transición τ de M, M' ejecuta los pasos B1 a B3, donde empleamos la notación j_τ para representar el número de la cinta que se vería afectada por la transición τ (recuerde que una transición de la máquina de Turing de varias cintas afecta sólo a una de las cintas de la máquina).

B1. Mueve la cabeza a la derecha hasta que el componente $2j_\tau$ de la tupla en la celda actual sea 1 (es decir, encuentra la posición de la cabeza asociada con la cinta j_τ).

B2. a. Si la transición τ es una operación de escritura, modifica el componente $2j_\tau - 1$ según se indique.

b. Si la transición τ es un movimiento hacia la derecha, reemplaza con un espacio en blanco el 1 del componente $2j_\tau$ de la celda actual, mueve la cabeza una celda a la derecha y cambia de espacio en blanco a 1 el componente $2j_\tau$ de la tupla, que allí se encuentra (si en lugar de una tupla detecta un espacio en blanco al moverse a la derecha, reemplaza éste con la $2K$–tupla $(\triangle, \triangle, \triangle, \triangle, \cdots, \triangle, \triangle)$ antes de escribir el 1).

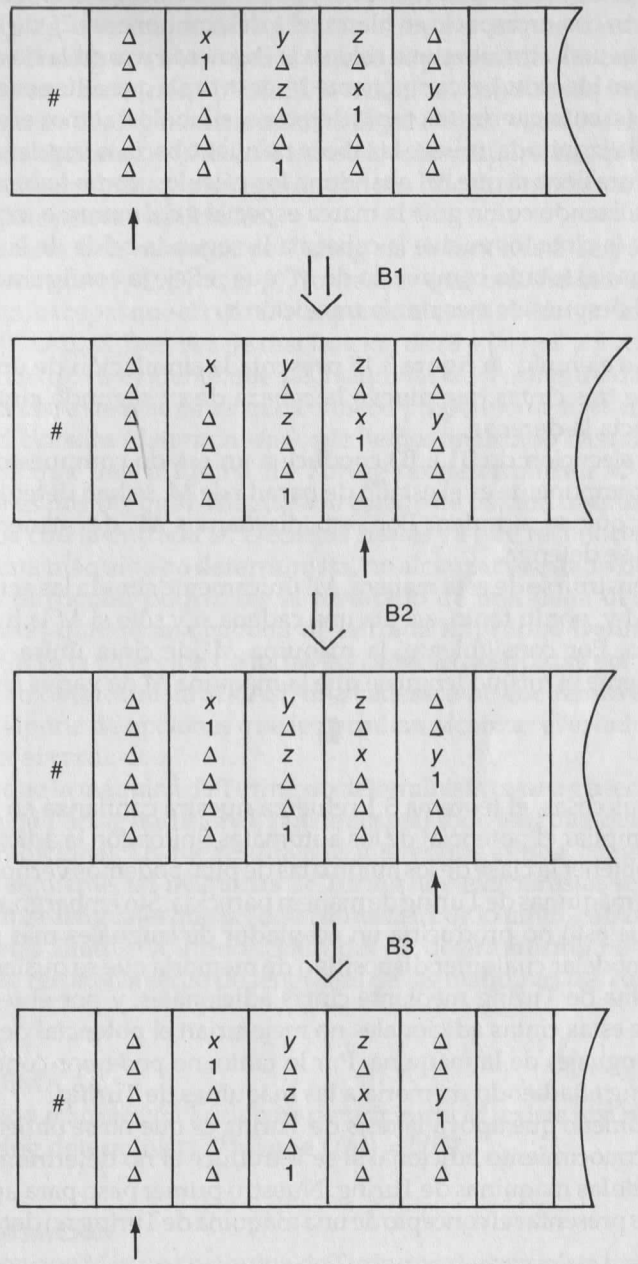


Figura 3.15 Simulación con una sola cinta de una máquina de Turing de tres cintas que ejecuta una operación de movimiento hacia la derecha en su segunda cinta

c. Si la transición τ es un movimiento hacia la izquierda, reemplaza con un espacio en blanco el 1 del componente $2j_\tau$ de la celda actual, se mueve una celda a la izquierda y cambia de espacio en blanco a 1 el componente $2j_\tau$ de la tupla que allí se encuentra (si en lugar de una tupla detecta el símbolo # al moverse hacia la izquierda, mueve la cabeza de nuevo hacia la izquierda; esto ocasionará que M' abandone los cálculos, como lo haría M).

B3. Utilizando como guía la marca especial # del extremo izquierdo de la cinta, devuelve la cabeza a la segunda celda de la cinta y pasa al estado compuesto de M' que refleje la configuración de M después de ejecutar la transición τ.

Como ejemplo, la figura 3.15 presenta la simulación de una máquina de tres cintas que mueve la cabeza de su segunda cinta una celda hacia la derecha.

Si la ejecución de B1 a B3 conduce a un estado compuesto cuyo primer componente es el estado de parada de M, se han detenido los cálculos que se simulan. Por esto diseñamos M' de manera que también se detenga.

Al construirse de esta manera, M' únicamente simula las actividades de M y, por lo tanto, acepta una cadena si y sólo si M la hubiera aceptado. Por consiguiente, la máquina M' de cinta única acepta exactamente el mismo lenguaje que la máquina M de varias cintas.

∎

Entre otras cosas, el teorema 3.1 refuerza nuestra confianza en la tesis de Turing. Al ampliar el potencial de los autómatas finitos con la adición de memoria para obtener la clase de los autómatas de pila, podemos vernos tentados a ampliar las máquinas de Turing de manera parecida. Sin embargo, el teorema 3.1 indica que esto no produciría un aceptador de lenguajes más poderoso. Podríamos modelar cualquier dispositivo de memoria que se qusiera agregar a una máquina de Turing mediante cintas adicionales, y por el teorema 3.1 sabemos que estas cintas adicionales no mejorarían el potencial de reconocimiento de lenguajes de la máquina. Por lo tanto, no podemos contradecir la tesis de Turing añadiendo memoria a las máquinas de Turing.

Otro fenómeno que apoya la tesis de Turing es que no se obtiene ningún poder de reconocimiento adicional si se introduce el no determinismo en el tratamiento de las máquinas de Turing. Nuestro primer paso para apoyar esta afirmación es presentar el concepto de una máquina de Turing no determinista.

Máquinas de Turing no deterministas

Una **máquina de Turing no determinista** es similar a una máquina de Turing tradicional; la diferencia es que quizá una máquina no determinista no se encuentre completamente definida o, lo que es más importante, puede que ofrezca más de una transición aplicable a un par estado–símbolo. Si una máquina de Turing no determinista llegara al par estado–símbolo actual sin que existiera una transición aplicable, la máquina abandonaría los cálculos. Si una máquina de Turing no determinista llegara al par estado–símbolo actual donde puede aplicarse más de una transición, la máquina llevaría a cabo una elección no determinista y proseguiría con sus cálculos mediante la ejecución de una de las opciones aplicables.

En resumen, una máquina de Turing no determinista se puede definir como una sexteta $(S, \Sigma, \Gamma, \pi, \iota, h)$, lo mismo que una máquina de Turing determinista, excepto que el cuarto componente es un subconjunto de $((S - \{h\}) \times \Gamma) \times (S \times (\Gamma \cup \{L, R\}))$ en vez de una función de $(S - \{h\}) \times \Gamma$ a $S \times (\Gamma \cup \{L, R\})$. En este contexto, es evidente que las máquinas de Turing no deterministas forman una clase de máquinas que contiene propiamente a las máquinas de Turing tradicionales (deterministas) que hemos analizado hasta ahora.

Decimos que una máquina de Turing no determinista M acepta una cadena w si es posible que M llegue a su estado de parada después de iniciar sus cálculos con la entrada w. Decimos *posible* ya que reconocemos que en el caso de una máquina no determinista, no alcanzar el estado de parada en un intento particular podría ser el resultado de una mala decisión de la máquina, más que de una cadena de entrada impropia. Definimos como lenguaje $L(M)$ a la colección de todas las cadenas aceptadas por la máquina de Turing no determinista M. Así, una cadena w se encuentra en $L(M)$ si y sólo si M dispone de opciones que le permitan alcanzar el estado de parada al recibir la entrada w.

Puesto que la máquina de Turing no determinista es una generalización de la máquina de Turing tradicional, puede aceptar todo lenguaje que acepta una máquina tradicional. Lo que resulta más interesante es que, como lo muestra el teorema siguiente, las máquinas de Turing no deterministas son incapaces de aceptar más lenguajes que las deterministas. Por lo tanto, como sucede con los autómatas finitos, la introducción del no determinismo no aumenta el potencial de reconocimiento de lenguajes de las máquinas de Turing.

TEOREMA 3.2

Para cada máquina de Turing no determinista M, existe una máquina de Turing determinista D tal que $L(M) = L(D)$.

DEMOSTRACIÓN

Suponga que M es una máquina de Turing no determinista que acepta el lenguaje $L(M)$. Debemos mostrar la existencia de una máquina de Turing determinista que acepte el mismo lenguaje. Esto se logra de manera indirecta, mostrando que existe una máquina de Turing determinista de tres cintas M' tal que $L(M) = L(M')$ y, por el teorema

3.1, debe existir una máquina de Turing tradicional (determinista, de cinta única) que acepte $L(M)$.

Diseñamos M' para que pruebe de manera sistemática todas las opciones posibles para la máquina determinista M, con el objetivo de encontrar una combinación que lleve a la aceptación de la cadena de entrada. Se emplean las tres cintas de M' de la manera siguiente: la primera cinta contiene la cadena de entrada que se evalúa; la segunda cinta se emplea como "cinta de trabajo" en donde repetidamente M' copia la versión intacta de la cadena de entrada y luego, usando esta copia, simula una secuencia de transiciones de M; la tercera cinta sirve para llevar el control de la secuencia de transiciones (de M) que se aplica, así como las secuencias que ya se han simulado.

El proceso de copiado de la cadena de entrada de la cinta 1 en la cinta 2 implica dos aspectos sutiles, pero importantes. En primer lugar, M' debe desplazar la cadena una celda hacia la derecha durante el proceso de copiado. Es decir, hay que colocar el contenido de la celda uno de la cinta 1 en la celda dos de la cinta 2, la celda dos de la cinta 1 va a la celda tres de la cinta 2, etcétera. Esto permite que M' coloque un símbolo especial en la celda del extremo izquierdo de su segunda cinta, lo que a su vez permite que M' detecte una terminación anormal de M sin abortar sus propios cálculos. Si la secuencia de transiciones que se simula ocasiona que M rebase el extremo de la cinta, M' detectará esta marca especial, notará que la secuencia actual no es productiva y comenzará el proceso de evaluación de otra secuencia.

El segundo aspecto que está presente en el proceso de copiado es que M' debe colocar una marca especial en el extremo derecho de la cadena de su segunda cinta. Si esta marca se detecta al simular las transiciones de M, se desplaza hacia la derecha. Así, cuando M' necesita borrar esta cinta antes de comenzar otra simulación, sólo tiene que borrar hasta esta marca especial.

Por último, debemos indicar cómo M' lleva el control de la simulación de las secuencias de transiciones de M. La idea es bastante sencilla. En primer lugar, rotule cada uno de los arcos del diagrama de transiciones de M con un símbolo único. Si existieran sólo cinco arcos en el diagrama, se podrían emplear los dígitos 1, 2, 3, 4 y 5. Luego, construya un componente en M' que genere todas las cadenas de estos símbolos en forma sistemática utilizando la cinta 3 (como modelo, véase Fig 3.10). Cada una de estas cadenas representa una secuencia de transiciones y, a final de cuentas, estará representada cada una de las transiciones posibles.

Al utilizar este generador de secuencias interno, las actividades de M' se llevan a cabo de la siguiente manera:

1. Copia la cadena de entrada de la cinta 1 a la cinta 2, en la forma antes descrita.
2. Genera la siguiente secuencia de transiciones en la cinta 3.
3. Simula esta secuencia con la cinta 2.
4. Si esta simulación conduce a un estado de parada de M, se detiene. De lo contrario, borra la cinta 2 y regresa al paso 1.

∎

En resumen, no podemos ampliar el potencial de reconocimiento de lenguajes de las máquinas de Turing añadiendo cintas o introduciendo un comportamiento no determinista, observación que apoya la tesis de Turing. Esto nos puede llevar a la conjetura de que la clase de lenguajes aceptados por máquinas de Turing representa el final de nuestra jerarquía de lenguajes que las máquinas pueden reconocer y, por lo tanto, son de importancia para nuestro estudio. Por esto, en las secciones siguientes nos dedicaremos a aprender más acerca de esta clase de lenguajes.

Ejercicios

1. a. Diseñe una máquina de Turing que acepte el lenguaje Σ^*, donde $\Sigma = \{x, y\}$.
   b. Diseñe una máquina de Turing que acepte el lenguaje $\varnothing$.

2. Muestre que una máquina de Turing se puede modificar para que evite una terminación anormal pero a la vez acepte las mismas cadenas que antes.

3. Muestre que si se permite en una máquina de Turing de varias cintas que las transiciones individuales afecten a más de una cinta, entonces no aumenta el potencial de la máquina. En otras palabras, muestre que cualquier transición que opera sobre más de una cinta se puede simular con una secuencia de transiciones que operan cada una en una sola cinta.

4. Muestre que cualquier cálculo de una "máquina de Turing" cuya cinta se extiende infinitamente hacia la izquierda y hacia la derecha se puede simular con una máquina de Turing de dos cintas y, por lo tanto, con una máquina de Turing tradicional.

3.4 LENGUAJES ACEPTADOS POR MÁQUINAS DE TURING

En la sección anterior analizamos los rudimentos de las máquinas de Turing como aceptadores de lenguajes, y denominamos lenguajes aceptados por máquinas de Turing a los lenguajes que dos aceptan. En esta sección describiremos con mayor detenimiento esta clase de lenguajes.

Comparación entre lenguajes aceptados por máquinas de Turing y lenguajes estructurados por frases

En el capítulo 1 presentamos las gramáticas estructuradas por frases y analizamos cómo una gramática de este tipo define un lenguaje que consiste en las cadenas de terminales generados por la gramática. Al restringir las formas de las reglas de reescritura disponibles, hemos podido identificar clases de gramáticas que generan lenguajes regulares e independientes del contexto. Ahora queremos considerar las gramáticas estructuradas por frases que no tienen restricción alguna en cuanto a sus reglas de reescritura. Así, tanto el lado izquierdo como el derecho de las reglas de reescritura pueden consistir en cualquier cadena finita de terminales y no terminales, siempre y cuando exista por lo menos un no terminal en el lado izquierdo.

Los lenguajes que generan estas gramáticas se conocen como **lenguajes estructurados por frases.** Éstos son los lenguajes que pueden definirse "gramaticalmente", en el sentido de que sus estructuras de cadenas se pueden analizar empleando una jerarquía de estructuras de frases. Puesto que las gramáticas regulares y las independientes del contexto son casos especiales de las gramáticas sin restricciones, los lenguajes que las primeras generan están contenidos en la clase de los lenguajes estructurados por frases. Además, la figura 3.16 muestra una gramática sin restricciones que genera el lenguaje $\{x^n y^n z^n : n \in \mathbb{N}\}$, el cual, como ya sabemos, no es independiente del contexto. Por ello, los lenguajes estructurados por frases constituyen una clase de lenguajes más extensa que la de los independientes del contexto.

En esta sección, nuestro objetivo es caracterizar a los lenguajes estructurados por frases como aquellos que las máquinas de Turing pueden aceptar. Es decir, *los lenguajes estructurados por frases son precisamente los lenguajes aceptados por máquinas de Turing*. Esto se demostrará en dos etapas. Primero, en el teorema 3.3 mostramos que todo lenguaje aceptados por máquinas de Turing es un lenguaje estructurado por frases; luego, en el teorema 3.4 mostramos que todo lenguaje estructurado por frases es aceptados por máquinas de Turing.

La demostración que daremos para el teorema 3.3 se basa en la construcción de una gramática que genera el mismo lenguaje que el aceptado por una máquina de Turing determinada. Esta construcción requiere un sistema de notación para representar la configuración total de una máquina de Turing en cualquier etapa de sus cálculos. Al emplear este sistema, el contenido de la

$$S \to xyNSz$$
$$S \to \lambda$$
$$yNx \to xyN$$
$$yNz \to yz$$
$$yNy \to yyN$$

Figura 3.16 Gramática que genera el lenguaje $\{x^n y^n z^n : n \in \mathbb{N}\}$

cinta de la máquina se representa como una cadena de símbolos de cinta encerrados entre corchetes. Primero representamos con [el extremo izquierdo de la cinta, luego presentamos la cadena de símbolos que se encuentran en la cinta, comenzando por la celda del extremo izquierdo e incluyendo por lo menos un espacio en blanco después del último símbolo distinto de un espacio, y por último cerramos la representación con]. De esta manera, una cinta que contiene $\triangle xxyx\triangle\triangle\triangle\cdots$ se podría representar como $[\triangle xxyx\triangle]$ o quizás como $[\triangle xxyx\triangle\triangle\triangle\triangle\triangle]$, y a una cinta que contiene $\triangle\triangle\triangle xx\triangle yx\triangle x\triangle\triangle\cdots$ como $[\triangle\triangle\triangle xx\triangle yx\triangle x\triangle\triangle]$.

Para completar la representación de la configuración de la máquina, insertamos en nuestra representación de la cinta el estado actual justo a la izquierda del símbolo actual. Así, si p fuera un estado de la máquina, entonces $[\triangle xpxyxx\triangle]$ representaría la configuración en la cual p es el estado actual y la configuración de la cinta es $\triangle x\underline{x}yxx\triangle\triangle\triangle\cdots$. De manera parecida, $[p\triangle xy\triangle]$ representaría a la máquina en el estado p con la configuración $\underline{\triangle}xy\triangle\triangle\cdots$. (Podemos suponer que los símbolos empleados para representar los estados de la máquina son distintos de los símbolos de cinta de la máquina.)

La importancia que para nosotros tiene esta notación es que nos ofrece un medio para expresar los cálculos de una máquina de Turing como secuencia de cadenas de símbolos, donde cada cadena representa la configuración de la máquina en un instante determinado de los cálculos. Así, si la máquina acepta cadenas deteniéndose con la cinta configurada como $\underline{\triangle}Y\triangle\triangle\triangle\cdots$, entonces el

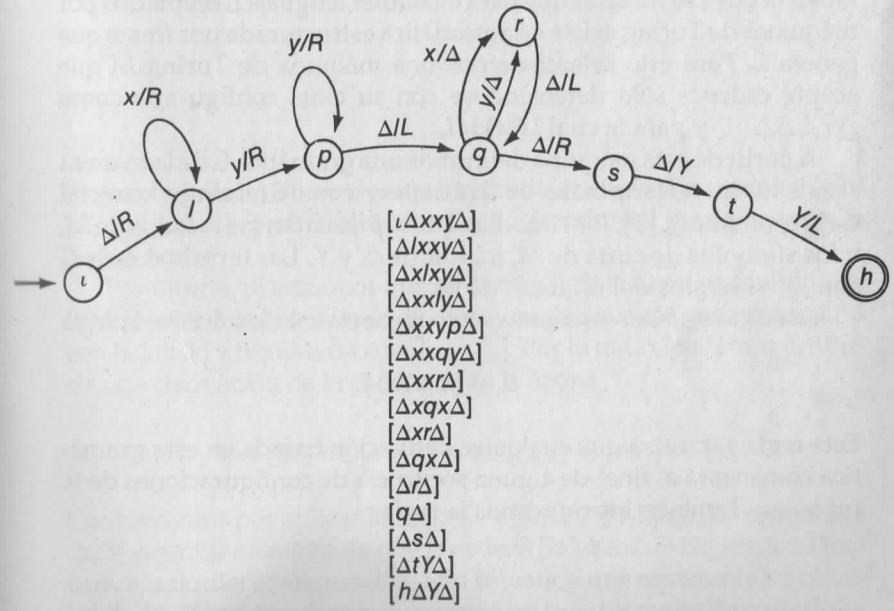


Figura 3.17 Diagrama de transiciones parcial para una máquina de Turing y la secuencia de configuraciones que produce al procesar la cadena *xxy*

proceso de aceptación de la cadena w se podría resumir como la secuencia de configuraciones de la máquina que comienza con $[\iota\triangle w\triangle]$ y termina con $[h\triangle Y\triangle]$, donde ι y h representan los estados de inicio y parada de la máquina, respectivamente.

La figura 3.17 muestra una máquina de Turing que acepta el lenguaje $\{x^m y^n: m \in \mathbb{N}, n \in \mathbb{N}^+\}$ y la secuencia de configuraciones que produce al procesar la cadena xxy. Esta secuencia comienza con una configuración que contiene la cadena de entrada y termina con la configuración $[h\triangle Y\triangle]$. Así, al leerla de atrás hacia adelante obtenemos la secuencia que conduce de la configuración $[h\triangle Y\triangle]$ a una configuración que contiene la cadena de entrada; esta secuencia es bastante similar a una derivación de la cadena. De hecho, si construyéramos una gramática en la cual toda derivación tuviera que comenzar con el patrón $[h\triangle Y\triangle]$ y cuyas reglas de reescritura simularon la acción inversa de una transición de la máquina, entonces estaríamos en camino de construir una gramática que generara las cadenas aceptadas por la máquina. Éste es el enfoque que usaremos para demostrar el teorema 3.3.

TEOREMA 3.3
Todo lenguaje aceptados por máquinas de Turing es un lenguaje estructurado por frases.

DEMOSTRACIÓN
Nuestra tarea es mostrar que para cualquier lenguaje L aceptados por máquinas de Turing existe una gramática estructurada por frases que genera L. Para esto seleccionamos una máquina de Turing M que acepte cadenas sólo deteniéndose con su cinta configurada como $\underline{\triangle}Y\triangle\triangle\triangle\cdots$ y para la cual $L(M) = L$.

A partir de esta máquina definimos una gramática G de la manera siguiente: los no terminales de G se definen como S (el símbolo inicial de la gramática), [,], los símbolos que representan los estados de M, y los símbolos de cinta de M, incluidos $\triangle$ y Y. Los terminales de G son los símbolos del alfabeto de M.

La única regla de reescritura que contiene el símbolo inicial es la regla

$$S \to [h\triangle Y\triangle]$$

Esta regla garantiza que cualquier derivación basada en esta gramática comenzará al final de alguna secuencia de configuraciones de la máquina. También introducimos la regla

$$\triangle] \to \triangle\triangle]$$

que permite que una derivación amplíe la cadena $[h\triangle Y\triangle]$ a cualquier longitud que se desee.

A continuación, introducimos reglas de reescritura que simulan transiciones a la inversa. Para cada transición de la forma $\delta(p, x) = (q, y)$ introducimos la regla de reescritura

$$qy \to px$$

(De esta manera, $[\triangle zqy\triangle]$ puede reescribirse como $[\triangle zpx\triangle]$, lo cual refleja que si la configuración de M fuera $[\triangle zpx\triangle]$, la aplicación de $\delta(p, x) = (q, y)$ desplazaría M hacia $[\triangle zqy\triangle]$.)

Para cada transición de la forma $\delta(p, x) = (q, R)$, introducimos la regla

$$xq \to px$$

(P. ej., $[\triangle xqyz\triangle]$ se podría reescribir como $[\triangle\ pxyz\ \triangle]$.)

Para cada transición de la forma $\delta(p, x) = (q, L)$ y cada símbolo de cinta y de M, introducimos la regla

$$qyx \to ypx$$

(Así, $[\triangle qyx\triangle\triangle]$ se podría reescribir como $[\triangle ypx\triangle\triangle]$.)

La lista de reglas de reescritura de G se completa introduciendo tres reglas que permiten que una derivación elimine los no terminales [, ι, $\triangle$ y] en ciertas circunstancias. Estas reglas son

$$[\iota\triangle \to \lambda$$
$$\triangle\triangle] \to \triangle]$$

y

$$\triangle] \to \lambda$$

(Por consiguiente, si una derivación produjera la configuración inicial $[\iota\triangle xyx\triangle\triangle\triangle]$, podría eliminar los no terminales para producir la cadena xyx.)

Por último, planteamos que $L(M) = L(G)$. Si w fuera una cadena de $L(M)$, existiría una secuencia de configuraciones de M que comenzaría con $[\iota\triangle w\triangle]$ y terminaría con $[h\triangle Y\triangle]$. Por lo tanto, podemos producir una derivación de la cadena w de la forma

$$S \Rightarrow [h\triangle Y\triangle] \Rightarrow \cdots \Rightarrow [\iota\triangle w\triangle] \Rightarrow w\triangle] \Rightarrow w$$

Comenzamos por aplicar la regla $S \to [h\triangle Y\triangle]$ y luego la regla $\triangle] \to \triangle\triangle]$ repetidamente hasta que la cadena $[h\triangle Y\triangle\triangle\cdots\triangle]$ sea tan larga como cualquier configuración de la secuencia que represente los cálculos de M. Después aplicamos en orden inverso las reglas de reescritura correspondientes a las transiciones de la secuencia de configuraciones original. Esto producirá el patrón $[\iota\triangle w\triangle\triangle\cdots\triangle]$, el cual podemos

reducir a w aplicando las reglas $\triangle\triangle] \to \triangle]$, $\triangle] \to \lambda$, y $[\iota\triangle \to \lambda$. Como resultado, w estaría en $L(G)$ (como ejemplo, la figura 3.18 presenta la derivación que corresponde a la secuencia de configuraciones proporcionadas en la figura 3.17).

A la inversa, si tuviéramos una cadena en $L(G)$, su derivación daría origen a una secuencia de configuraciones que mostrarían a su vez cómo podría aceptar M la cadena. Así, cualquier cadena de $L(G)$ también se encuentra en $L(M)$.

∎

El siguiente teorema es lo único que nos falta para justificar nuestra afirmación de que los lenguajes estructurados por frases son exactamente los lenguajes aceptados por máquinas de Turing.

TEOREMA 3.4

Todo lenguaje estructurado por frases es un lenguaje aceptados por máquinas de Turing.

DEMOSTRACIÓN

Comenzamos por observar que al aplicar la demostración del teorema 3.1 a una máquina de Turing no determinista de varias cintas se produciría una máquina no determinista de cinta única que aceptaría el mismo lenguaje que la máquina con varias cintas (es posible que la transición por simular de un estado compuesto de la forma $(p, x_1, x_2, \cdots, x_k)$ ya no esté determinada en forma única, pero entonces se conocerán todas las opciones antes de la ejecución y, por tanto, podrán incorporarse al autómata permitiendo el no determinismo). Así, para cualquier máquina de Turing no determinista M de varias cintas existe una máquina de Turing M' de una sola cinta tal que $L(M) = L(M')$. Tal observación nos permite demostrar este teorema mostrando que para cada gramática G existe una máquina de Turing no determinista N de dos cintas tal que $L(G) = L(N)$. Después, N se podría simular con una máquina de Turing no determinista de cinta única (por la observación anterior), la cual podría ser simulada a su vez por una máquina de Turing convencional (según el Teorema 3.2).

A continuación observamos que una máquina de Turing puede realizar cualquier regla de reescritura de la gramática. Es decir, si una cadena de símbolos v aparece en algún lugar de la cinta de la máquina y la gramática contiene la regla $v \to w$, donde w representa una cadena (posiblemente vacía) de terminales y no terminales, entonces la máquina puede reemplazar la cadena v por la cadena w aplicando operaciones de escritura y de desplazamiento hacia la izquierda y hacia la derecha.

$$
\begin{aligned}
S &\Rightarrow [h\Delta Y\Delta] \\
&\Rightarrow [h\Delta Y\Delta\Delta] \\
&\Rightarrow [h\Delta Y\Delta\Delta\Delta] \\
&\Rightarrow [\Delta tY\Delta\Delta\Delta] \\
&\Rightarrow [\Delta s\Delta\Delta\Delta\Delta] \\
&\Rightarrow [q\Delta\Delta\Delta\Delta\Delta] \\
&\Rightarrow [\Delta r\Delta\Delta\Delta\Delta] \\
&\Rightarrow [\Delta qx\Delta\Delta\Delta] \\
&\Rightarrow [\Delta xr\Delta\Delta\Delta] \\
&\Rightarrow [\Delta xqx\Delta\Delta] \\
&\Rightarrow [\Delta xxr\Delta\Delta] \\
&\Rightarrow [\Delta xxqy\Delta] \\
&\Rightarrow [\Delta xxyp\Delta] \\
&\Rightarrow [\Delta xxly\Delta] \\
&\Rightarrow [\Delta xlxy\Delta] \\
&\Rightarrow [\Delta lxxy\Delta] \\
&\Rightarrow [\iota\Delta xxy\Delta] \\
&\Rightarrow xxy\Delta] \\
&\Rightarrow xxy
\end{aligned}
$$

Figura 3.18 Derivación de la cadena *xxy* correspondiente a la secuencia de configuraciones de la figura 3.17

Construimos ahora una máquina no determinista de dos cintas que opera de la siguiente manera: utiliza la cinta 1 para almacenar la cadena de entrada por evaluar. Escribe el símbolo inicial de la gramática en la cinta 2. Luego aplica repetidamente y de manera no determinista las reglas de reescritura a la cadena de la cinta 2 (decimos "de manera no determinista" ya que puede existir más de una regla aplicable en un momento determinado). Si el contenido de la cinta 2 se convierte en una cadena con sólo terminales, compara esta cadena con la cadena de entrada que está almacenada en la cinta 1. Si ambas cadenas son idénticas, la máquina se detiene; de lo contrario, mueve una de las cabezas hacia la izquierda hasta que ocurra una terminación anormal.

En resumen, este proceso únicamente emplea la cinta 2 para calcular una derivación con base en las reglas de la gramática. Si la cadena de entrada se puede derivar de la gramática, entonces es posible que esta cadena sea la producida en la cinta 2. En este caso, la máquina aceptará la entrada, deteniéndose. Sin embargo, si la entrada no puede derivarse de la gramática, la cadena producida en la cinta 2 nunca podrá ser igual a la de entrada, y sería imposible que la máquina aceptara la cadena. Por consiguiente, el lenguaje aceptado por la máquina es el lenguaje generado por la gramática.

∎

Como resumen, los lenguajes y las máquinas estudiados hasta ahora forman la jerarquía que se presenta en la figura 3.19.

Alcance de los lenguajes estructurados por frases

La equivalencia entre los lenguajes estructurados por frases y los lenguajes aceptados por máquinas de Turing significa que las máquinas de Turing se pueden utilizar para estudiar el alcance de los lenguajes estructurados por frases. Esto es la esencia del teorema siguiente.

TEOREMA 3.5
Para cada alfabeto Σ existe al menos un lenguaje sobre Σ que no es un lenguaje estructurado por frases.

DEMOSTRACIÓN
Sea L un lenguaje estructurado por frases del alfabeto Σ. Entonces, por el teorema 3.4, existe una máquina de Turing M tal que $L(M) = L$. Comenzamos nuestra demostración señalando que existe una M cuyos símbolos de cinta se eligen del conjunto $\Sigma \cup \{\triangle\}$.

Si M es la máquina de Turing tal que $L(M) = L$ y los símbolos de cinta de M incluyen más que $\Sigma \cup \{\triangle\}$, podemos construir otra máquina de Turing M' con símbolos de cinta de $\Sigma \cup \{\triangle\}$, de modo que $L(M') = L(M) = L$. De hecho, sea x cualquier símbolo (distinto de espacio en blanco) de Σ. Entonces, disponga en una lista los símbolos distintos de espacios en blanco de M y represente cada entrada de la lista con una cadena de x de longitud igual a la posición del símbolo

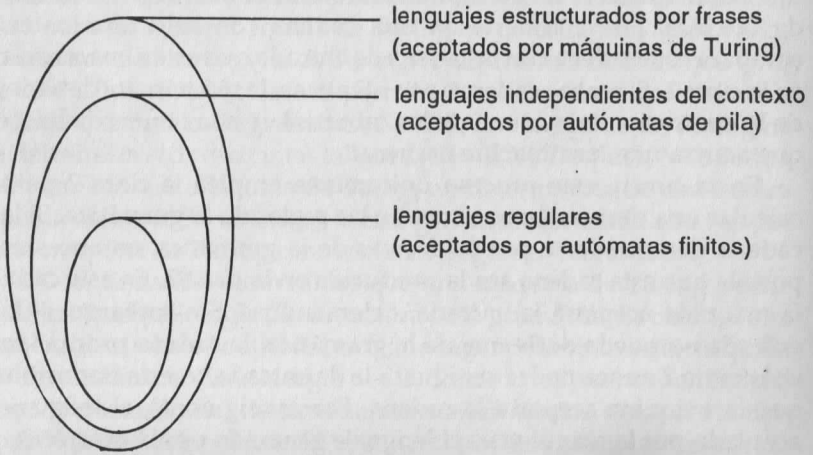


Figura 3.19 Jerarquía de máquinas y lenguajes

en esta lista (el primer símbolo se representa con x, el segundo con xx, etc.). Así, cada uno de los símbolos de la cinta de M que no sea un espacio en blanco, y por consiguiente también cada símbolo del alfabeto de M', se representa con una cadena de x única. De esta manera, siempre es posible representar el contenido de la cinta de M por medio de cortas cadenas de x separadas por espacios en blanco (un espacio en blanco de la cinta de M se codificaría como dos espacios consecutivos, es decir, una cadena de x vacía).

Ahora construya M' para que traduzca su entrada a esta forma codificada, simule las acciones de M y se detenga únicamente si M se detiene. Así, M' aceptará exactamente las mismas cadenas que acepta M, sin emplear más que los símbolos de cinta $\Sigma \cup \{\triangle\}$.

Concluimos que dado un lenguaje estructurado por frases de un alfabeto Σ, existe una máquina de Turing con símbolos de cinta $\Sigma \cup \{\triangle\}$ tal $L(M) = L$.

Lo que queda de nuestra demostración es, en esencia, lo mismo que la demostración del teorema 1.1. Podemos generar de manera sistemática una lista de todas las máquinas de Turing con símbolos de cinta $\Sigma \cup \{\triangle\}$, generando primero aquellas máquinas con sólo dos estados (recuerde que una máquina de Turing tiene cuando menos dos estados), seguido por las que tienen tres estados, etcétera. Por lo tanto, existe una cantidad contable de tales máquinas de Turing. Por otra parte, existe una cantidad infinita de cadenas en Σ^* y, debido a ello, es incontable el número de lenguajes que se pueden formar a partir de Σ. Por consiguiente, existen más lenguajes basados en Σ que máquinas de Turing con símbolos de cinta en $\Sigma \cup \{\triangle\}$. Así mismo, deben existir lenguajes basados en Σ que no son lenguajes estructurados por frases.

∎

Aquí es donde comienza a complicarse la trama. Los teoremas 3.3, 3.4 y 3.5 nos dicen que existen lenguajes que no tienen bases gramaticales y además que las máquinas de Turing sólo pueden reconocer aquellos lenguajes que sí tienen bases gramaticales. Si aceptamos la tesis de Turing de que las máquinas de Turing engloban la esencia de cualquier proceso computacional, debemos concluir que ningún proceso algorítmico puede reconocer los lenguajes que no tienen bases gramaticales.

Esta conclusión no sólo implica que existen lenguajes que no podemos analizar sintácticamente con un computador, sino además tiene ramificaciones relacionadas con la búsqueda de sistemas que comprendan los lenguajes naturales, un área de gran actividad en las investigaciones recientes. De hecho, nos indica que en el desarrollo de un sistema de procesamiento de lenguajes naturales existe el requisito de que el lenguaje que se procesa tenga una estructura gramatical bien definida; si el lenguaje natural no cuenta con esta estructura, no seremos capaces de procesarlo algorítmicamente.

Este requisito tiene además un estrecho vínculo con el tema de la inteligencia natural comparada con la inteligencia artificial. Si, como muchos creen, la mente humana opera ejecutando algoritmos, entonces la tesis de Turing indica que existen lenguajes cuya sintaxis no puede analizar la mente humana. Por otra parte, si la inteligencia natural se basa en un nivel más poderoso que la ejecución de algoritmos, entonces la tesis de Turing sugiere que el desarrollo de máquinas verdaderamente inteligentes seguirá por siempre siendo un sueño.

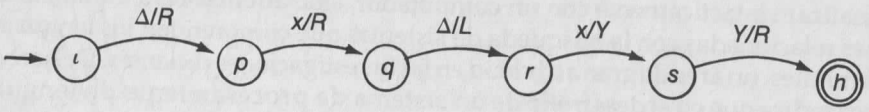
Vemos entonces que la teoría presentada hasta ahora tiene consecuencias significativas. El único punto débil en la cadena es la tesis de Turing, la *conjetura* de que el concepto de computación con una máquina de Turing es tan poderoso (aunque quizás no tan conveniente) como cualquier otro modelo computacional. Como mencionamos antes, la mayoría de los científicos de la computación acepta actualmente esta tesis. Gran parte de esta aceptación proviene de hecho de que la tesis de Turing es congruente con otros resultados y conjeturas que han surgido al estudiar la computación desde otras perspectivas; en el capítulo siguiente consideraremos algunas de estas estrategias.

Ejercicios

1. Dibuje un diagrama de transiciones para todas las máquinas de Turing con símbolos de cinta x y $\triangle$ que sólo tengan dos estados (uno inicial y uno de parada. ¿Cómo se relacionan su capacidad para llevar a cabo esta tarea y el hecho de que sea contable la colección de todas las máquinas de Turing cuyo conjunto de símbolos de cinta es $\{x, \triangle\}$?

2. Muestre que al agregar una segunda pila a un autómata de pila obtenemos una clase de máquinas con el mismo poder de aceptación de lenguajes que la clase de máquinas de Turing.

3. Utilice el proceso de construcción descrito en la demostración del teorema 3.3, para desarrollar una gramática estructurada por frases que acepte el mismo lenguaje aceptado por la máquina de Turing cuyo diagrama de transiciones aparece a continuación (la máquina acepta cadenas deteniéndose con la cinta configurada como $\underline{\triangle}Y\triangle\triangle\triangle\cdots$.) Con la gramática que obtenga, muestre una derivación de la cadena x.



4. Muestre que el lenguaje $\{x^n y^{2n} z^{4n} : n \in \mathbb{N}\}$ es aceptados por máquinas de Turing.

3.5 MÁS ALLÁ DE LOS LENGUAJES ESTRUCTURADOS POR FRASES

Hemos demostrado la existencia de lenguajes que no están estructurados por frases (o, lo que es lo mismo, no son aceptados por máquinas de Turing), pero aún no identificamos explícitamente un lenguaje de éstos. Un objetivo fundamental de esta sección es llenar este hueco; la presentación de las máquinas de Turing universales y la distinción entre lenguajes aceptables y decidibles representan importantes extensiones de nuestro análisis.

Sistema de codificación de máquinas de Turing

Para identificar un lenguaje que no es aceptados por máquinas de Turing (y por lo tanto no es estructurado por frases) necesitaremos un sistema de codificación con el cual podamos representar las máquinas de Turing con alfabeto Σ y símbolos de cinta $\Sigma \cup \{\triangle\}$ como cadenas que únicamente contienen ceros y unos. Por ello, hacemos una pausa para presentar dicho sistema antes de proseguir con el tema principal de la sección.

Dada una máquina M por representar, nuestro sistema de codificación necesita que acomodemos los estados de M en una lista cuyo primer elemento corresponda al estado inicial y el segundo al estado de parada. Con base en el orden de esta lista, podemos hablar del primer estado de M, del segundo estado de M y, en general, del estado j de M. Establecemos que el estado j de M se representará con una cadena de ceros de longitud j. Así, el estado inicial estará representado por 0, el estado de parada por 00 y el siguiente estado (si existe) por 000.

Luego, representamos los símbolos L y R, así como los símbolos en Σ (los símbolos de cinta de M distintos de espacio en blanco) como cadenas de ceros. Esto se hace acomodando en una lista los símbolos de Σ y representando L con 0, R con 00, el primer símbolo de la lista con 000, el segundo símbolo con 0000 y, en general, el símbolo j con una cadena de ceros de longitud $j + 2$.

Si ahora establecemos que el símbolo del espacio en blanco se representará con la cadena vacía, obtenemos un sistema en el cual podemos representar los símbolos L y R, los estados de M y los símbolos de la cinta de M por medio de cadenas de ceros. A su vez, esto nos permite representar cualquier transición de M como una cadena de ceros y unos. A fin de cuentas, se puede identificar cualquier transición (que debe tener la forma $\delta(p, x) = (q, y)$ con una cuádrupla (p, x, q, y) donde p es el estado actual, x es el símbolo actual, q es el nuevo estado y y es o bien un símbolo de cinta (si la transición es una operación de escritura) o bien L o R (si la transición es una operación de movimiento de la cabeza). De esta manera, es posible representar toda la transición como cuatro cadenas de ceros separados por unos. Por ejemplo, la cadena 01000100100 representaría la transición $\delta(\iota, x), = (h, R)$, donde x es el símbolo representado por 000 y h es el estado de parada de la máquina. Puesto que un espacio en blanco se representa

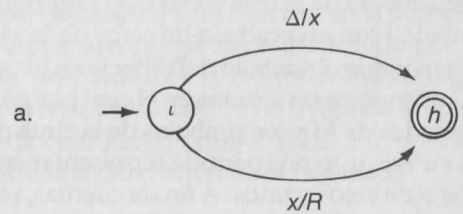
con la ausencia de ceros, la cadena 011001 representa la transición $\delta(\iota, \triangle) = (h, \triangle)$, donde h es una vez más el estado de parada de la máquina.

Observe que una lista de todas las transiciones disponibles para una máquina de Turing con símbolos de cinta $\Sigma \cup \{\triangle\}$ constituye una descripción completa de la máquina. Por lo tanto, cualquier máquina de este tipo se puede representar como una lista de transiciones codificada. Adoptaremos la regla convencional de añadir un 1 al inicio y al final de la lista, y un solo 1 para separar las transiciones de la lista. Así, la cadena 1011001000101000100 1001 (un 1 introductorio seguido por el código de transición 011001000, un 1 separador, el código 01000100100 y un 1 final) representa la máquina con dos transiciones, $\delta(\iota, \triangle) = (h, x)$ y $\delta(\iota, x) = (h, R)$, que se muestra en la figura 3.20.

Al representar de esta forma una máquina de Turing, establecemos que las transiciones por incluir en la lista tendrán el orden siguiente: primero presentamos todas las transiciones que surgen del estado 0 (el inicial), luego las que se originan del estado 000, seguidas por las que tienen su origen en el estado 0000, etc., hasta incluir todos los estados donde pueden originarse transiciones. Las transiciones correspondientes a un estado se acomodan de acuerdo con el símbolo que se requiere en la celda actual de la cinta, comenzando por la transición que requiere un espacio en blanco como símbolo actual, luego la transición que requiere el símbolo con código 000, después la transición cuyo símbolo actual es 0000, etcétera. Esta uniformidad hace más fácil evaluar una cadena de ceros y unos para ver si constituye una representación válida de alguna máquina de Turing (determinista y, por lo tanto, completamente definida).

Un lenguaje no estructurado por frases

Al emplear esta forma de codificación, encontramos que cada máquina de Turing con alfabeto Σ y símbolos de cinta $\Sigma \cup \{\triangle\}$ se puede representar como una cadena de ceros y unos, la cual a su vez puede interpretarse como un entero



b. 1011001000101000100 1001

Figura 3.20 a. Máquina de Turing sencilla
b. La misma máquina en forma codificada

no negativo escrito en binario. Es más, si contáramos en binario llegaríamos en algún momento al patrón que representa una máquina de Turing específica cualquiera con símbolos de cinta $\Sigma \cup \{\triangle\}$. Por supuesto, las representaciones binarias de muchos enteros no negativos no corresponden a representaciones de máquinas válidas. Establezcamos que cada uno de estos enteros se asociará con la sencilla máquina que se muestra en la figura 3.21. Así, tenemos entonces una función de $\aleph$ *sobre* las máquinas de Turing con alfabeto Σ y símbolos de cinta $\Sigma \cup \{\triangle\}$. Empleamos la notación M_i para representar la máquina que esta función relaciona con el entero i.

Con base en esta función podemos construir otra, esta vez de Σ^* sobre la colección de máquinas de Turing con alfabeto Σ y símbolos de cinta $\Sigma \cup \{\triangle\}$. Basta con asociar una cadena w de Σ^* con la máquina $M_{|w|}$, donde $|w|$ representa la longitud de w. A fin de hacer más sencilla esta notación, utilizamos M_w para representar la máquina asociada a w por medio de esta función.

Observe que los símbolos de w también se encuentran en el alfabeto de M_w, por lo que tiene sentido aplicar M_w a la cadena de entrada w. Definimos que el lenguaje L_0 es el subconjunto $\{w: M_w$ no acepta $w\}$ de Σ^*; una cadena w de Σ^* se halla en L_0 si y sólo si ésta no es aceptada por su máquina M_w correspondiente.

Ahora nuestra tarea es mostrar que L_0 no es aceptado por máquinas de Turing. Para esto, mostramos que se llega a una contradicción si se supone lo contrario. Si L_0 fuera aceptado por alguna máquina de Turing, utilizando un argumento similar al de la demostración del teorema 3.5 podemos suponer que el alfabeto de la máquina es Σ y su conjunto de símbolos de cinta es $\Sigma \cup \{\triangle\}$. Así mismo, L_0 debe ser aceptado por M_{w_0} para alguna cadena w_0 en Σ^* (toda máquina de Turing con alfabeto Σ y símbolos de cinta $\Sigma \cup \{\triangle\}$ es M_w para una w de Σ^*). Por lo tanto, $L_0 = L(M_{w_0})$.

Ahora nos preguntamos si la cadena w_0 existe o no en $L(M_{w_0})$ (tiene que estar o no estar), pero, como veremos en un momento, ambas opciones nos llevan a contradicciones. Con base en la definición de L_0, sabemos que $w_0 \in L(M_{w_0})$ implica que $w_0 \notin L_0$, y que $w_0 \notin L(M_{w_0})$ implica que $w_0 \in L_0$. Sin embargo, como $L_0 = L(M_{w_0})$, ambos enunciados son contradictorios. En vista de esta paradoja, debemos concluir que nuestra conjetura con respecto a la aceptación de L_0 es falsa; en otras palabras, el lenguaje L_0 no es aceptado por su máquina de Turing.

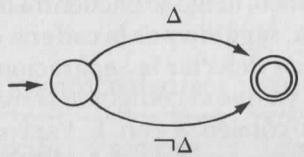


Figura 3.21 Máquina de Turing sencilla

Máquinas de Turing universales

Nuestro siguiente objetivo es mostrar que el complemento de L_0, el conjunto $\{w: M_w \text{ acepta } w\}$, es aceptado por máquinas de Turing. Esto demostrará que la capacidad de una máquina de Turing para aceptar un lenguaje no es simétrica con la capacidad para rechazar el complemento del lenguaje. En otras palabras, existen casos donde se puede construir una máquina de Turing que identifique las cadenas de un lenguaje, pero no se puede construir una máquina de Turing que identifique las cadenas que no están en el lenguaje.

Sin embargo, para mostrar que el complemento de L_0 es aceptados por máquinas de Turing, necesitamos presentar el concepto de **máquina de Turing universal.** Ésta no es más que una máquina de Turing "programable" que, dependiendo de su programa, puede simular cualquier otra máquina de Turing. De esta forma, las máquinas de Turing universales son los antepasados abstractos de nuestros modernos computadores programables que leen y ejecutan los programas almacenados en sus memorias. Es más, las máquinas de Turing universales están diseñadas para ejecutar programas que se almacenan en sus cintas.

Un programa para una máquina de Turing universal no es más que una versión codificada de una máquina de Turing que lleva a cabo la tarea que se desea ejecute la máquina universal. Suponga que queremos programar una máquina de Turing universal para que desempeñe una actividad específica. Primero diseñaríamos una máquina de Turing tradicional que realizara la tarea; luego codificaríamos esta máquina como una cadena de ceros y unos, como ya lo hemos hecho. La cadena de código sería el programa de la máquina universal.

El siguiente paso sería codificar los datos que se usarían como entrada para los cálculos deseados. Para esto recordamos que a cada símbolo de la cinta distinto de espacio en blanco, correspondiente a la máquina que acabamos de codificar, se le asigna un código de tres o más ceros. Por consiguiente, cualquier cadena de estos símbolos se puede representar con la secuencia de códigos adecuada. En esta secuencia separamos con un 1 los códigos adyacentes y encerramos entre corchetes toda la secuencia, con un 1 en cada extremo (Fig. 3.22). Observe que la cadena vacía estaría representada por 11, una secuencia vacía de códigos.

Después de codificar la máquina que se simulará y la cadena que servirá como entrada, colocamos estos códigos en la cinta de entrada de la máquina de Turing universal, como se describe a continuación. La celda del extremo izquierdo permanece en blanco, luego se encuentra la versión codificada de la máquina que se simulará, seguida por la cadena de entrada codifcada (la máquina universal puede detectar la separación entre la codificación de la máquina y los datos, ya que el código de la máquina termina con un 1 y el código de los datos comienza con 1. Para ser más precisos, toda transición codificada debe comenzar con el código de un estado, lo que requiere por lo menos un 0. Así, es posible detectar el fin de la lista de transiciones con la presencia de un código de estado incorrecto. Véase Fig. 3.23).

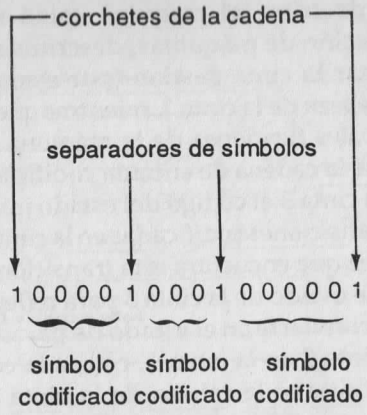


Figura 3.22 Cadena de tres símbolos en forma codificada

Una vez que esta información se ha colocado en la cinta de la máquina universal, ubicamos la cabeza de la máquina sobre la celda del extremo izquierdo de la cinta y ponemos en marcha la máquina a partir de su estado inicial. Partiendo de esta configuración, la máquina de Turing universal extrae y simula las transiciones que encuentra en la primera porción de la cin-ta conforme manipula la cadena existente en la segunda porción de la cinta.

Para ser más precisos, podríamos visualizar una máquina de Turing universal como una máquina de tres cintas. La primera cinta se usa para almacenar el programa de entrada y los datos, así como para contener cualquier salida; la segunda cinta sirve como área de trabajo, en la cual se manipulan los datos; y la tercera cinta se emplea para contener una representación del estado actual de la máquina simulada.

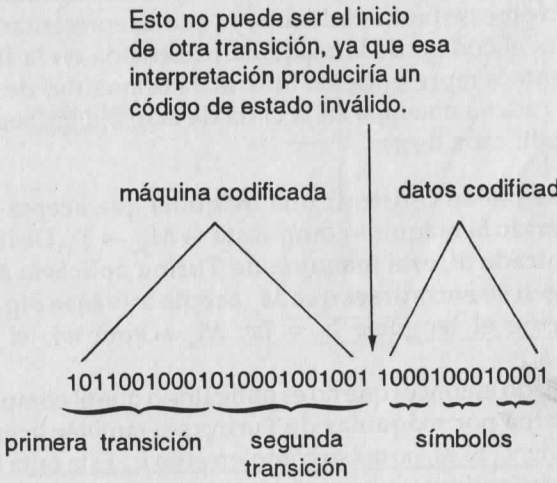


Figura 3.23 Máquina de Turing y sus datos en forma codificada

La figura 3.24 describe tal máquina como una composición de los bloques de construcción de máquinas, descritos antes. Los exponentes se emplean para indicar la cinta destino (por ejemplo, R^1 mueve una celda hacia la derecha la cabeza de la cinta 1, mientras que R^2 mueve la cabeza de la cinta 2). Las principales funciones de la máquina compuesta son: primero encuentra el inicio de la cadena de entrada codificada, copia esta cadena a la cinta 2 y coloca en la cinta 3 el código del estado inicial. Luego, prosigue con la búsqueda de las transiciones codificadas en la cinta hasta encontrar una que sea aplicable. Una vez que encuentra esta transición, la simula en la cinta 2 y actualiza el código de estado de la cinta 3 para reflejar el nuevo estado. Si el estado simulado se convierte en el estado de parada, borra la cinta 1, copia el contenido de la cinta 2 en la cinta 1, coloca la cabeza de la cinta 1 en el lugar donde se encontraba la cabeza de la cinta 2 al llegar al estado de parada, y se detiene.

Esta presentación de una máquina de Turing universal se ha hecho en el contexto de una máquina de tres cintas, pero sabemos que es posible simular cualquier máquina de Turing de tres cintas con una máquina de Turing que sólo tenga una. Consideramos entonces que una máquina de Turing universal es una máquina de una cinta denotada por T_u.

Comparación entre lenguajes aceptables y decidibles

Con el apoyo de una máquina de Turing universal, ahora podemos construir una máquina de Turing que acepte el complemento del lenguaje L_0. Comenzamos por construir una máquina de Turing M_{pre} que procese una cadena de entrada w de Σ^* de la siguiente manera:

1. Genera una cadena de ceros y unos que represente a la máquina M_w (se trata de un proceso bastante directo, ya que la representación deseada para M_w es el código de la máquina presentada en la figura 3.21 o simplemente la representación binaria de la longitud de w).
2. Coloca la cadena obtenida en la cinta de la máquina, seguida por la versión codificada de w.

Por lo tanto, se puede construir una máquina que acepta el complemento de L_0 formando la máquina compuesta $\rightarrow M_{pre} \rightarrow T_u$. De hecho, dada una cadena de entrada w, esta máquina de Turing aplicaría M_w a w y se detendría si y sólo si se encontrara que M_w acepta w (véase Fig. 3.25). Así, acepta precisamente el lenguaje $L_1 = \{w\colon M_w \text{ acepta } w\}$, el cual es el complemento de L_0.

Hemos confirmado entonces que no es obligatorio que el complemento de un lenguaje aceptado por máquinas de Turing sea también aceptado por máquinas de Turing: L_1 lo es, no así su complemento L_0. Esta falta de simetría no es un simple hecho curioso, sino que tiene repercusiones considerables. En nuestro caso, quiere decir que la capacidad para aceptar un lenguaje no es

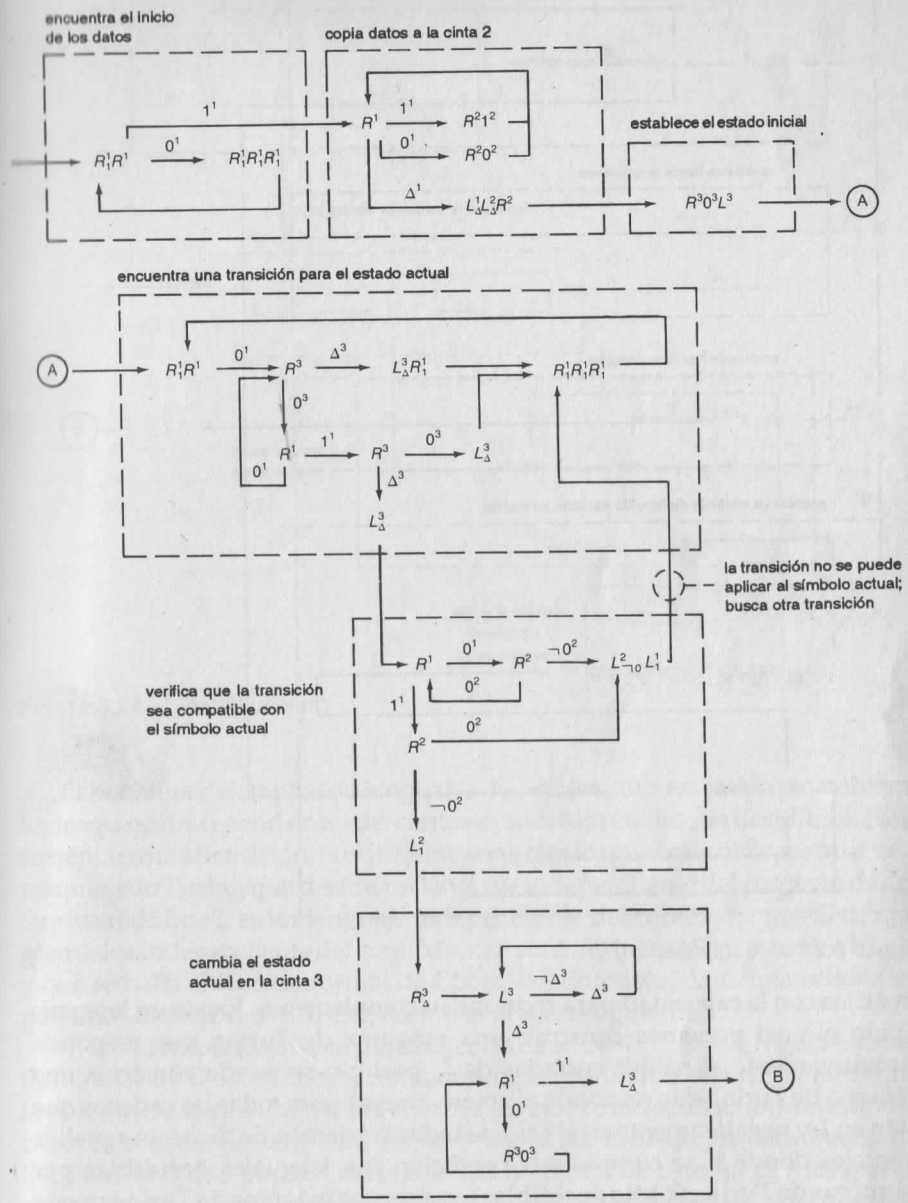


Figura 3.24 Máquina de Turing universal (continúa en la página siguiente)

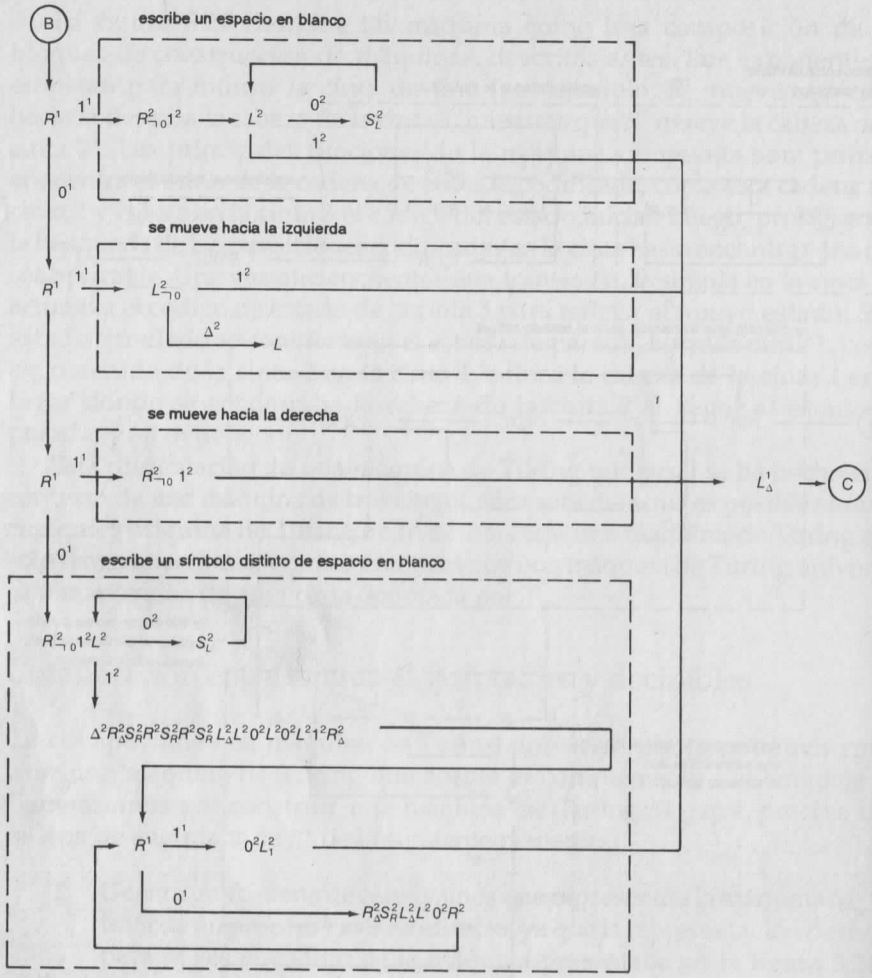


Figura 3.24 (continuación)

simétrica con la capacidad para rechazar su complemento. Existe un lenguaje L para el cual podemos construir una máquina de Turing que responda afirmativamente al recibir entradas de L, pero no se puede construir una máquina de Turing que responda afirmativamente para todas las cadenas que están en L y negativamente en el caso de todas las demás. Se dice que aquellos lenguajes donde sí se cumple esta condición son lenguajes **decidibles por máquinas de Turing** (o sólo **decidibles**), ya que una máquina de Turing puede decidir si una cadena se encuentra o no en el lenguaje, en vez de sólo aceptar las cadenas que pertenecen al mismo (empleando términos menos formales, decimos que la máquina de Turing decide el lenguaje en vez de únicamente aceptarlo). En resumen, el lenguaje L_1 que antes definimos es aceptable, pero no decidible.

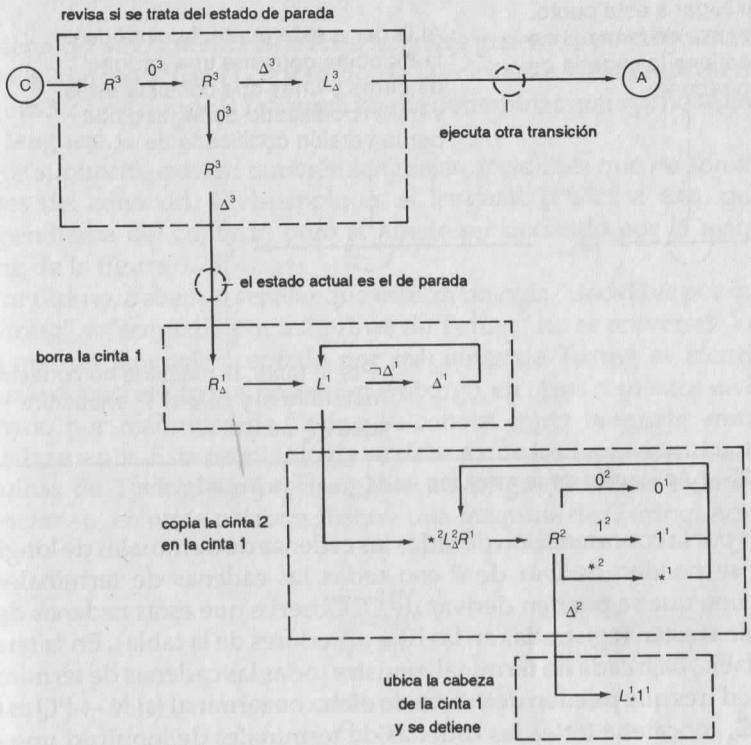


Figura 3.24 (continuación)

Para obtener ejemplos de lenguajes decidibles sólo tenemos que volver a los lenguajes independientes del contexto, todos los cuales son decidibles. Para sustentar esta afirmación, bosquejemos una técnica general para construir una máquina de Turing que decida un lenguaje independiente del contexto dado. Suponemos que L es un lenguaje independiente del contexto y que G es una gramática independiente del contexto, con símbolo inicial S, que genera $L - \{\lambda\}$ y que se halla en forma normal de Chomsky. Entonces, dada una cadena w, podemos decidir si $w \in L$ ejecutando el proceso siguiente.

Si $w = \lambda$, responda sí o no dependiendo de si $\lambda \in L$. Si $w \neq \lambda$, registre los no terminales de G como encabezamientos de las columnas de una tabla y llene la porción de la tabla bajo estos encabezamietos como se describe a continuación: bajo cada no terminal, registre todas las cadenas de terminales de longitud uno que pueden derivarse del no terminal; esto no es más que la colección de terminales que aparecen en el lado derecho de una regla de reescritura cuyo lado izquierdo es el no terminal en cuestión (si las reglas $N \to x$ y $N \to y$ están en G, entonces registre la cadenas x y y bajo el no terminal N). Para construir la segunda fila de la tabla, registre bajo cada no terminal las cadenas de terminales de longitud dos que pueden derivarse de dicho no terminal (si $N \to PQ$ es una regla de G, registre en la columna N las cadenas

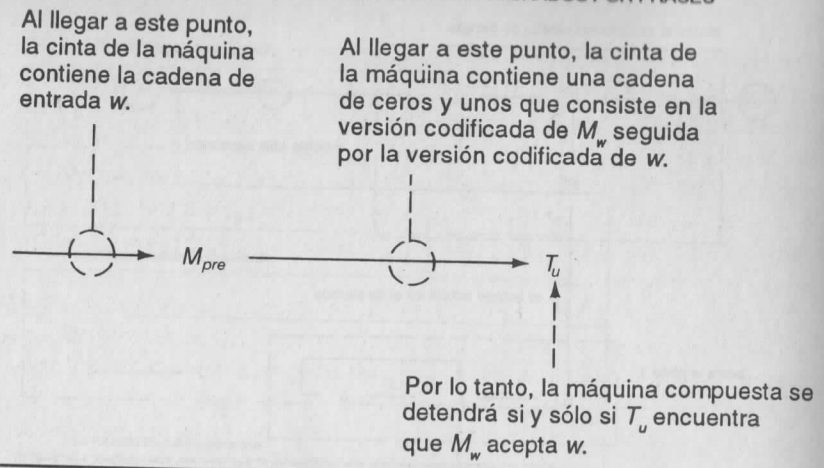


Figura 3.25 Aplicación de la máquina $\rightarrow M_{pre} \rightarrow T_u$ a la cadena w

obtenidas por la concatenación de todas las cadenas de terminales de longitud uno que se pueden derivar de P con todas las cadenas de terminales de longitud uno que se pueden derivar de Q. Observe que estas cadenas de un solo terminal están registradas en las filas anteriores de la tabla). En la tercera fila de la tabla, bajo cada no terminal, registre todas las cadenas de terminales de longitud tres que pueden derivarse de dicho no terminal (si $N \rightarrow PQ$ es una regla de G, concatene todas las cadenas de terminales de longitud uno que pueden derivarse de P con cada una de las cadenas de terminales de longitud dos que se pueden derivar de Q. Luego, concatene todas las cadenas de terminales de longitud dos que pueden derivarse de P con las de longitud uno que se pueden derivar de Q. Una vez más, todas las cadenas que se deben concatenar se encuentran en las casillas de la tabla correspondientes a las filas por encima de la que estamos llenando.)

En general, registramos en la fila n bajo cada no terminal N todas las cadenas de terminales de longitud n que se pueden derivar de dicho no terminal. Estas cadenas se pueden encontrar localizando primero cada regla de la forma $N \rightarrow PQ$ y luego, para cada i en $\{1, 2, \cdots, n-1\}$, concatenando cada cadena de terminales de longitud i derivable de P con cada cadena de terminales de longitud $n-i$ derivable de Q (véase Fig. 3.26). Continuamos con la construcción de esta tabla hasta completar la fila número $|w|$ Al llegar a este punto verificamos si w se ha registrados bajo el símbolo inicial de G. Si es así, entonces $w \in L$; de lo contrario, $w \notin L$. Por lo tanto, concluye el proceso de decisión.

Para realizar este procedimiento con una máquina de Turing, podemos usar una máquina con cintas múltiples que tenga una cinta más que el número de no terminales en G. La primera cinta se puede usar para almacenar la cadena de entrada y la respuesta afirmativa o negativa. Cada una de las cintas restantes se puede emplear para representar una columna de la tabla, separando con diagonales los elementos de distintas filas. Así, el proceso de decidir si

la cadena xyx se encuentra o no en el lenguaje generado por la gramática de la figura 3.26 produciría el contenido de las cintas que está representado en la figura 3.27, de donde la máquina puede determinar que xyx no se encuentra en el lenguaje.

Por supuesto, existen también lenguajes decidibles que no son independientes del contexto. Un ejemplo es el lenguaje $\{x^n y^n z^n : n \in \mathbb{N}\}$, que no es independiente del contexto pero sí puede ser decidido por la máquina de Turing de la figura 3.28.

Por último, debemos señalar que la terminología "decidible por máquinas de Turing" y "aceptado por máquinas de Turing" no es universal. Ya hemos visto que un lenguaje aceptado por máquinas de Turing es idéntico a un lenguaje estructurado por frases. Sin embargo, en otros contextos un lenguaje aceptado por máquinas de Turing se conoce como **lenguaje enumerable recursivamente.** Esta terminología se debe a que los lenguajes aceptado por máquinas de Turing son precisamente los lenguajes cuyas cadenas puede enumerar –o, en otras palabras, listar– una máquina de Turing. Además, en

$S \rightarrow MN$
$M \rightarrow MP$
$N \rightarrow PN$
$M \rightarrow x$
$N \rightarrow y$
$P \rightarrow x$
$P \rightarrow y$

	S	M	N	P
1		x	y	x y
2	xy	xx xy	xy yy	
3	xxy xyy	xxx xxy xyx xyy	xxy yxy xyy yyy	
4	xxxy xyxy xxyy xyyy	xxxx xxyx xyxx xyyx xxxy xxyy xyxy xyyy	xxxy xyxy xxyy xyyy yxxy yyxy yxyy yyyy	

Figura 3.26 Gramática independiente del contexto y primeras cuatro filas de la tabla construida a partir de ella

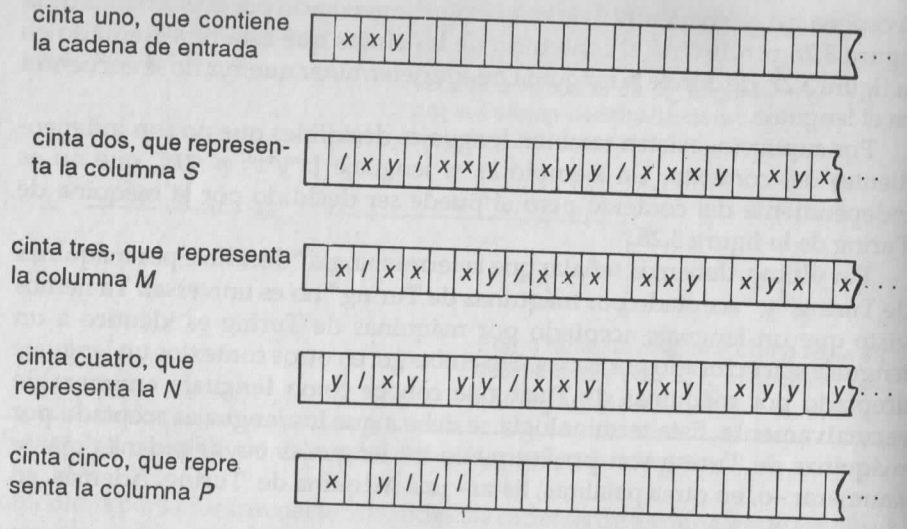


Figura 3.27 Realización con cinco cintas del proceso de construcción de la tabla de la figura 3.26

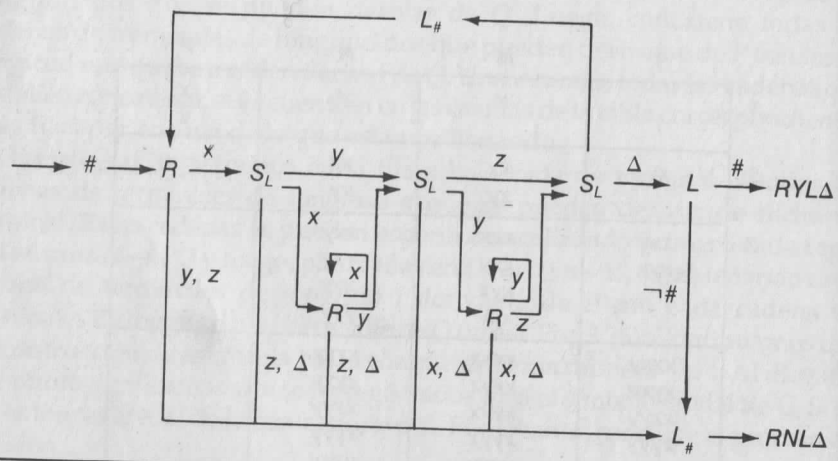


Figura 3.28 Máquina de Turing que decide el lenguaje $\{x^n y^n z^n : n \in \mathbb{N}\}$

en aquellas situaciones en las cuales se habla de lenguajes enumerables recursivamente, por lo general se hace referencia a los lenguajes decidibles por máquinas de Turing como **lenguajes recursivos.**

El problema de la parada

Damos fin a esta sección con un ejemplo de otro lenguaje que no es decidible por una máquina de Turing, no sólo porque precisemos otra muestra sino porque el caso que se presentará, conocido como problema de la parada, es un ejemplo clásico en la teoría de la computación.

Recuerde que cualquier máquina de Turing se puede codificar como una cadena de ceros y unos. Representemos con $\rho(M)$ esta versión codificada de una máquina de Turing M. Si ahora centramos nuestra atención en las máquinas con alfabeto $\{0, 1\}$ y símbolos de cinta $\{0, 1, \triangle)$, entonces la cadena de código $\rho(M)$ se podría usar como entrada para la misma máquina M. No nos interesa si esta forma de usar M tiene relación con el fin para el cual se diseñó originalmente M. Lo único que nos importa es si M se detiene o no cuando inicia su operación con $\rho(M)$ como entrada. Si se detiene, decimos que M es autoterminante. De esta manera, cualquier máquina de Turing con alfabeto $\{0, 1\}$ y símbolos de cinta $\{0, 1, \triangle\}$ es o bien autoterminante o bien no autoterminante.

Definamos ahora el lenguaje L_h de $\{0, 1\}^*$ como $\{\rho(M): M$ es autoterminante$\}$ y preguntemos si L_h es decidible por máquinas de Turing. Para efectuar una decisión con respecto a L_h, se requiere la capacidad para detectar si una cadena de $\{0, 1\}^*$ es la versión codificada de una máquina autoterminante, lo que en esen-cia es el problema de decidir si una máquina se para cuando se le aplica una entrada específica; por consiguiente, el problema de decisión de L_h se conoce como **problema de la parada.**

Por desgracia, el lenguaje L_h no es decidible según Turing. Para convencernos de esto, supongamos lo contrario, que existe una máquina de Turing M_h que decide L_h. Entonces, modificaríamos M_h para obtener otra máquina M'_h que responda con el mensaje 1 en caso de que M_h hubiera respondido Y y 0 cuando la respuesta de M_h fuera N. Como en la demostración del teorema 3.6, podemos establecer que los símbolos de cinta que necesita M'_h consisten sólo en $\{0, 1, \triangle\}$. Así, se podría emplear M'_h para construir la máquina compuesta

$$\rightarrow M'_h \rightarrow R \xrightarrow{1} R \circlearrowleft$$

con símbolos de cinta $\{0, 1, \triangle\}$, que se detiene si y sólo si M'_h llega a su estado de parada con salida 0. Representemos con M_0 a esta máquina compuesta.

A partir de aquí, nuestro argumento gira alrededor de la pregunta ¿es M_0 autoterminante o no autoterminante? Puesto que se trata de una máquina de Turing con símbolos de cinta $\{0, 1, \triangle\}$, debe ser una cosa o la otra. Supongamos que es autoterminante. Entonces, al recibir la entrada $\rho(M_0)$, M'_h se detendrá con salida 1. Esto quiere decir que M_0 no se detendría si iniciara con la entrada $\rho(M_0)$. (La ejecución de M_0 recorrería el arco $R \xrightarrow{1} R$ y quedaría atrapada en el proceso infinito de movimiento de la cabeza a la derecha. Véase Fig. 3.29.) Sin embargo, ésta es la característica que define a una máquina que no es autoterminante. Entonces, si suponemos que M_0 es autoterminante, llegamos a la contradicción de que no es autoterminante.

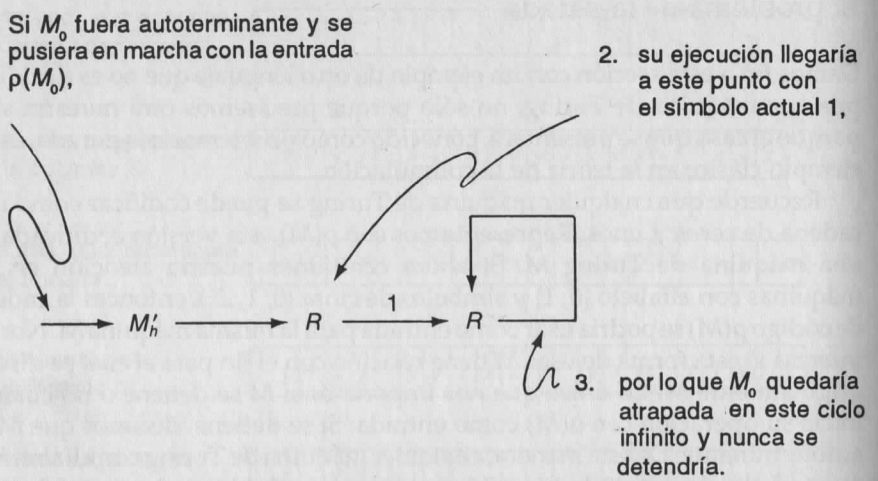


Figura 3.29 Ejecución de la máquina M_0 con entrada $\rho(M_0)$ bajo el supuesto de que M_0 es autoterminante

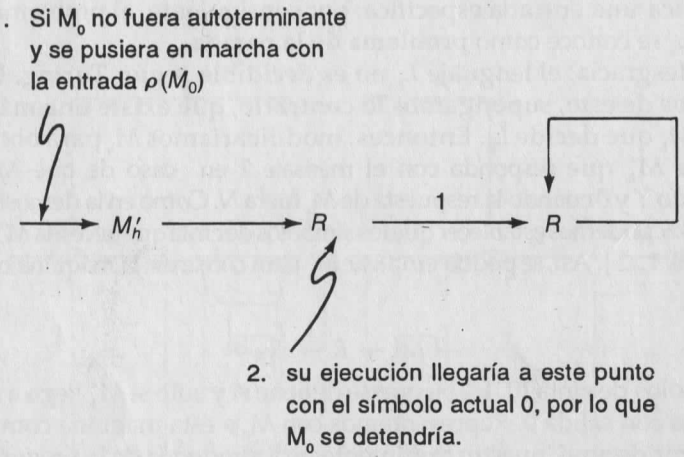


Figura 3.30 Ejecución de la máquina M_0 con entrada $\rho(M_0)$ bajo el supuesto de que no es autoterminante

Puesto que M_0 debe ser autoterminante o no autoterminante, al llegar a este punto debemos concluir que no es autoterminante. Empero, si M_0 no fuera autoterminante, entonces, al recibir la entrada $\rho(M_0)$, M'_h se detendría con salida 0. Esto quiere decir que M_0 se detendría si iniciara con la entrada $\rho(M_0)$.

(La ejecución de M_0 no podría recorrer el arco $R \xrightarrow{1} R$. Véase Fig. 3.30.) Sin embargo, ésta es la característica que define a una máquina autoterminante. Por lo tanto, también debe ser falsa nuestra suposición de que M_0 no es autoterminante.

Hemos llegado a la inconsistencia de una máquina M_0 con símbolos de cinta $\{0, 1, \triangle\}$ que no es autoterminante ni no autoterminante, cuando la máquina debe ser una u otra cosa. Por consiguiente, nos vemos obligados a concluir que nuestra suposición inicial es falsa; es decir, el lenguaje L_h no es decidible por máquinas de Turing.

Ejercicios

1. Diseñe una máquina de Turing que acepte exactamente aquellas cadenas de ceros y unos que sean representaciones válidas de máquinas de Turing deterministas, de acuerdo con el sistema de codificación descrito en esta sección.

2. Identifique la paradoja que existe en el siguiente enunciado:

   El cocinero de una comunidad aislada cocina para aquellas personas, y sólo aquellas personas, que no cocinan para sí mismas.

   (Sugerencia: ¿Quién cocina para el cocinero?) ¿Cómo se relaciona esta pregunta con esta sección?

3. Diseñe una máquina de Turing que acepte las cadenas del lenguaje $L = \{w^n: w = xy\}$ al detenerse con su cinta configurada como $\underline{\triangle}Y\triangle\triangle\triangle\cdots$ y rechace las cadenas que no se hallan en L al detenerse con la configuración $\underline{\triangle}N\triangle\triangle\triangle\cdots$.

   Explique por qué no puede construirse esta máquina en el caso de todos los lenguajes aceptados por máquinas de Turing.

4. Presente un argumento de que cualquier lenguaje que contenga un número finito de cadenas es decible por máquinas de Turing.

5. Amplíe la tabla de la figura 3.26 para solucionar la pregunta de si la cadena $xxyyyxy$ es generada por la gramática.

3.6 COMENTARIOS FINALES

En este capítulo y los anteriores presentamos una jerarquía de lenguajes junto con los autómatas que se requieren para aceptarlos. Se trata de una variante de la jerarquía de lenguajes conocida como jerarquía de Chomsky (nombrada así

en honor de N. Chomsky, pionero del desarrollo de la teoría de lenguajes formales durante la década de los cincuenta). En la figura 3.31 se presenta un resumen de la jerarquía que hemos presentado. Mostramos así mismo que cada nivel de esta jerarquía contiene propiamente el siguiente nivel inferior. De hecho, en cada nivel proporcionamos un ejemplo explícito de un lenguaje que reside en ese nivel pero no en el inferior. En este momento, un excelente ejercicio de repaso sería recopilar estos ejemplos e incluirlos en el diagrama de la figura 3.31.

Uno de los principales resultados de este capítulo ha sido la presentación de la tesis de Turing, la cual, traducida al contexto de la figura 3.31, nos dice que un sistema computacional nunca podrá efectuar un análisis sintáctico de aquellos lenguajes que se encuentran más allá de los lenguajes estructurados por frases. Como se señaló al final de la sección 3.4, esta tesis tiene grandes repercusiones. Por esto, no es ninguna sorpresa que la tesis de Turing haya llamado la atención de muchos investigadores, quienes han tratado de apoyar sus afirmaciones o de desafiar su validez.

En el próximo capítulo veremos algunos de los resultados de las actividades de estos investigadores. Por ahora sólo señalaremos que la amplia gama de terminología que hemos empleado (como "lenguajes aceptable por máquinas de Turing", "lenguajes estructurados por frases" y "lenguajes recursivamente enumerables") es el resultado del alcance de la tesis de Turing. En varias disciplinas se han observado y estudiado los mismos límites aparentes del poder de los procesos computacionales que surgen de la hipótesis de Turing, incluso cuando cada quien ha atacado el problema de la computabilidad desde una perspectiva distinta y con diferente terminología.

Problemas de repaso del capítulo

1. Con los bloques de construcción presentados en la sección 3.2, construya una máquina de Turing compuesta que convierta su cinta de la configuración $\underline{\triangle}w\triangle\triangle\triangle\cdots$, donde w es cualquier cadena de x y y, en la configuración $\underline{\triangle}w\triangle v\triangle\triangle\triangle\cdots$, donde v es la cadena w escrita a la inversa.

2. Con los bloques de construcción presentados en la sección 3.2, construya una máquina de Turing compuesta que forme la concatenación de dos cadenas v y w en $\{x, y\}^*$ convirtiendo su cinta de la configuración $\underline{\triangle}v\triangle w\triangle\triangle\triangle\cdots$ en $\underline{\triangle}vw\triangle\triangle\triangle\cdots$.

3. ¿Cómo puede el resultado de la ejecución de $\rightarrow RL$ diferir del de $\rightarrow LR$?

4. Dibuje un diagrama de transiciones para la máquina de Turing compuesta $\rightarrow R_x\triangle L$.

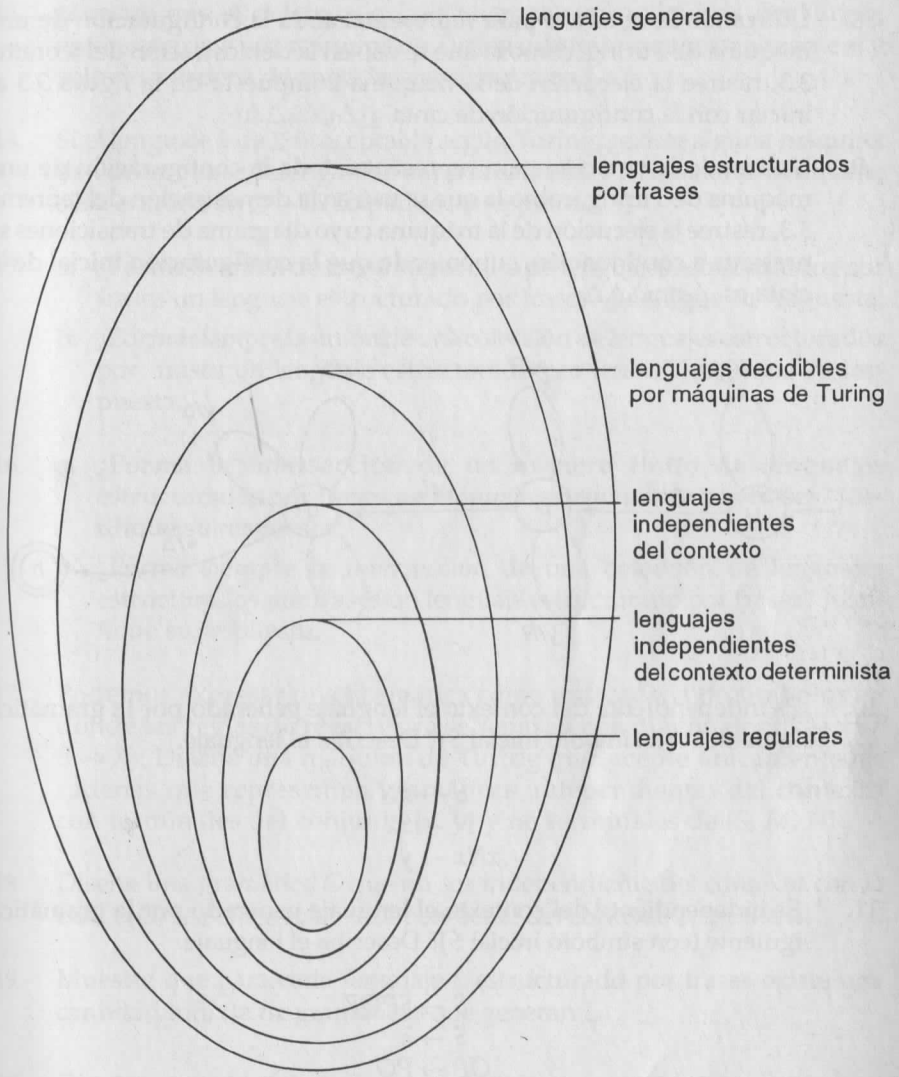


Figura 3.31 Variante de la jerarquía de lenguajes de Chomsky

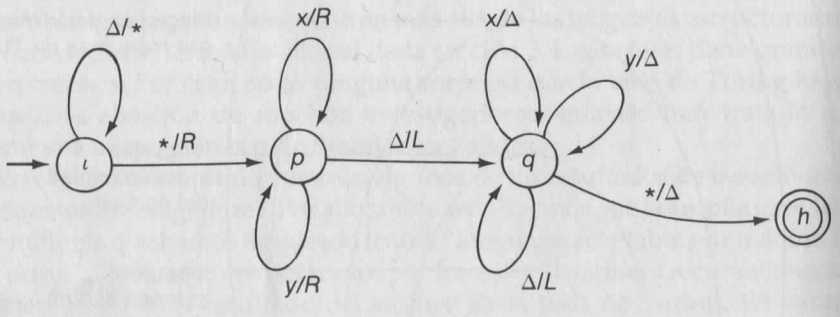
5. Diseñe una máquina de Turing M tal que $L(M) = \{x^n y^{2n} z^n : n \in \mathbb{N}\}$.

6. Diseñe una gramática G estructurada por frases tal que $L(G) = \{x^m y^n x^m y^n : m, n \in \mathbb{N}\}$.

7. Diseñe una gramática G estructurada por frases tal que $L(G) = \{x^n y^{2n} z^n : n \in \mathbb{N}\} \cup \{x^n y^n z^n : n \in \mathbb{N}\}$.

8. Utilizando la notación para representar toda la configuración de una máquina de Turing, como la que se usó en la demostración del teorema 3.3, rastree la ejecución de la máquina compuesta de la figura 3.3 al iniciar con la configuración de cinta $\underline{\triangle}\triangle yx\triangle\triangle\triangle\cdots$.

9. Utilizando la notación para representar toda la configuración de una máquina de Turing, como la que se usó en la demostración del teorema 3.3, rastree la ejecución de la máquina cuyo diagrama de transiciones se presenta a continuación, suponiendo que la configuración inicial de la cinta es $\underline{\triangle}xyx\triangle\triangle\triangle\cdots$.



10. ¿Es independiente del contexto el lenguaje generado por la gramática siguiente (con símbolo inicial S)? Describa el lenguaje.

$$S \to xN$$
$$N \to Sx$$
$$xNx \to y$$

11. ¿Es independiente del contexto el lenguaje generado por la gramática siguiente (con símbolo inicial S)? Describa el lenguaje.

$$S \to SPQR$$
$$S \to \lambda$$
$$QP \to PQ$$
$$PQ \to QP$$
$$PR \to RP$$
$$RP \to PR$$
$$QR \to RQ$$
$$RQ \to QR$$
$$P \to x$$
$$Q \to y$$
$$R \to z$$

12. Diseñe una máquina de Turing M tal que $L(M) = \{x, y, z\}^* - \{x^n y^n z^n : n \in \mathbb{N}\}$.

13. Muestre que si el lenguaje L es aceptado por máquinas de Turing, entonces existe una máquina de Turing que termina anormalmente si y sólo si su cadena de entrada se encuentra en L.

14. Si el lenguaje L de Σ es aceptable según Turing, ¿existe alguna máquina de Turing que se detenga si su entrada está en L y termine anormalmente si su entrada en $\Sigma^* - L$? Explique su respuesta.

15. a. ¿Forma la unión de un número finito de lenguajes estructurados por frases un lenguaje estructurado por frases? Justifique su respuesta.
    b. ¿Forma siempre la unión de una colección de lenguajes estructurados por frases un lenguaje estructurado por frases? Justifique su respuesta.

16. a. ¿Forma la intersección de un número finito de lenguajes estructurados por frases un lenguaje estructurado por frases? Justifique su respuesta.
    b. ¿Forma siempre la intersección de una colección de lenguajes estructurados por frases un lenguaje estructurado por frases? Justifique su respuesta.

17. Podemos expresar una gramática como una cadena de símbolos en donde las reglas de reescritura se separan con diagonales ($S \to xS$/ $S \to \lambda$). Diseñe una máquina de Turing que acepte únicamente las cadenas que representen gramáticas independientes del contexto con terminales del conjunto $\{x, y\}$ y no terminales de $\{S, M, N\}$.

18. Diseñe una gramática G que no sea independiente del contexto con la cual $L(G)$ sea el lenguaje independiente del contexto $\{x^n y^n : n \in \mathbb{N}\}$.

19. Muestre que para cada lenguaje L estructurado por frases existe una cantidad infinita de gramáticas que generan L.

20. Diseñe una máquina de Turing (quizás con varias cintas) que ordene alfabéticamente una lista de cadenas de $\{x, y, Z\}^*$. Suponga que las cadenas de la lista de entrada están separadas por un espacio en blanco.

21. Muestre que la colección de lenguajes no estructurados por frases tiene mayor cardinalidad que la colección de los lenguajes estructurados por frases (por lo tanto, existen más lenguajes no aceptados por máquinas de Turing que lenguajes que sí lo son).

22. Muestre que para cualquier alfabeto Σ existe un lenguaje L de Σ tal que ni L ni $\Sigma^* - L$ son aceptados por máquinas de Turing.

23. Muestre que la colección de lenguajes decidibles por máquinas de Turing para un alfabeto cualquiera es infinita, pero contable.

24. Muestre que si una máquina de Turing M acepta cada cadena w en $L(M)$ sin tener que ejecutar más de $|w| + 1$ pasos, entonces $L(M)$ debe ser regular.

25. ¿Es un subconjunto de un lenguaje estructurado por frases siempre un lenguaje estructurado por frases? Explique su respuesta.

26. Muestre que la estrella de Kleene de un lenguaje aceptado por máquinas de Turing es también aceptable según Turing.

27. Muestre que el lenguaje $\{x^n: n$ es un entero primo positivo$\}$ es aceptado por máquinas de Turing. ¿Es decidible? ¿Por qué?

28. Muestre que el lenguaje $\{x^{n^2}: n \in \mathbb{N}\}$ es aceptado por máquinas de Turing. ¿Es decidible? ¿Por qué?

29. Dibuje un diagrama de transiciones para una máquina de Turing que decida el lenguaje $\{x^m y^n: m, n \in \mathbb{N} \text{ y } m \geq n\}$.

30. Construya, con los bloques de construcción presentados en la sección 3.2, una máquina de Turing que acepte el lenguaje $\{x^m y^n x^m y^n: m, n \in \mathbb{N}\}$.

31. Aplique el algoritmo de construcción de tablas descrito en la sección 3.5 para decidir si la cadena $xxyyxxxyyy$ se encuentra en el lenguaje generado por la gramática siguiente, cuyo símbolo inicial es S.

$$\begin{aligned} S &\to SS \\ S &\to MN \\ N &\to SP \\ M &\to x \\ N &\to y \\ P &\to y \end{aligned}$$

¿Se encuentran también las cadenas $xxxyyxyy$, $xyxyxyxy$ y $xyxxxyyy$?

32. Amplíe la tabla de la figura 3.26 para determinar si la cadena $xyyxyyy$ es generada por la gramática de dicha figura.

33. Suponga que $p(n)$ es una expresión polinomial en n y que M es una máquina de Turing que acepta toda cadena w de $L(M)$ sin tener que ejecutar más de $p(|w|)$ pasos. Muestre que el lenguaje $L(M)$ es decidible.

34. Muestre que todo lenguaje aceptado por máquinas de Turing puede generarse a partir de una gramática estructurada por frases en la que ninguna regla de reescritura tiene una terminal como parte de su lado izquierdo.

35. Muestre que para cualquier lenguaje L recursivamente enumerable, existe una máquina de Turing que, al ponerse en marcha con una cinta en blanco, comenzará a generar en su cinta una lista de las cadenas en L, de manera que cada una de ellas aparecerá tarde o temprano en la cinta.

36. Muestre que todo lenguaje enumerable recursivamente infinito contiene un lenguaje recursivo infinito.

Problemas de programación

1. Desarrolle un simulador de máquinas de Turing. Diseñe este simulador de manera que se proporcione en forma tabular la descripción de la máquina que se simulará, para que pueda reemplazarse fácilmente con la descripción de otra máquina.

2. Escriba un programa que simule la máquina de Turing universal de tres cintas descrita en este capítulo.

3. Escriba un programa para decidir si la cadena que se proporcione como entrada se halla o no en el lenguaje $\{x^m y^n z^m: m, n \in \mathbb{N}\}$, aplicando el algoritmo de construcción de tablas descrito en la sección 3.5.

CAPÍTULO 4

Computabilidad

4.1 Fundamentos de la teoría de funciones recursivas
Funciones parciales
Funciones iniciales
Funciones recursivas primitivas
4.2 Alcance de las funciones recursivas primitivas
Ejemplos de funciones recursivas primitivas
Más allá de las funciones recursivas primitivas
4.3 Funciones recursivas parciales
Definición de las funciones recursivas parciales
Funciones Computables por máquinas de Turing
Computabilidad según Turing de funciones recursivas parciales
Naturaleza recursiva parcial de las máquinas de Turing
4.4 Poder de los lenguajes de programación
Un lenguaje de programación esencial
Recursiva parcial implica programable con lo esencial
Programable con lo esencial implica recursiva parcial
4.5 Comentarios finales

Nuestro estudio de los autómatas como aceptadores de lenguajes nos ha llevado a un límite aparente para el poder de los procesos computacionales, una marca que la tesis de Turing presenta en forma explícita. Uno de los principales factores que apoya esta tesis es que en diversas áreas de investigación ha surgido el mismo límite aparente; por ejemplo, vimos que el poder de las máquinas de Turing como aceptadores de lenguajes corresponde a los poderes generativos de las gramáticas. En este capítulo investigaremos otros casos en los cuales el poder computacional de las máquinas de Turing corresponde a las capacidades de otros sistemas computacionales.

Comenzamos nuestra investigación considerando de nuevo la pregunta que se planteó en el capítulo introductorio, con respecto al poder expresivo de

nuestro lenguaje de programación preferido. Recuerde que deben existir funciones que no pueden calcularse con ningún programa escrito en ese lenguaje. Así mismo, hicimos la pregunta con respecto al grado en que esta limitación se debía al diseño del lenguaje o, al contrario, era un reflejo de las limitaciones de los procesos algorítmicos en general. Nos preguntamos si la imposibilidad de calcular una función de un programa en determinado lenguaje significaba que el lenguaje era incapaz de expresar un algoritmo o más bien que no existía un algoritmo que pudiera calcular la función.

Para responder esta pregunta, primero identificaremos una clase de funciones que al parecer contiene la totalidad de las funciones computables: todas aquellas funciones que pueden calcularse por medios algorítmicos, sin importar cómo pueda expresarse o implantarse ese algoritmo. Luego mostraremos que las funciones de esta clase pueden calcularse por medio de las máquinas de Turing (lo que apoya la tesis de Turing), así como mediante algoritmos expresados en un subconjunto muy sencillo de la mayoría de los lenguajes de programación. A partir de esto llegamos a la conclusión de que los límites detectados en las máquinas de Turing y la mayor parte de los lenguajes de programación son el reflejo de las limitaciones de los procesos computacionales, no del diseño de la máquina o del lenguaje que se emplee.

4.1 FUNDAMENTOS DE LA TEORÍA DE FUNCIONES RECURSIVAS

Hasta ahora, nuestro estudio de los procesos computacionales y sus capacidades se ha basado en un enfoque operativo, no funcional. Nos hemos centrado en cómo se lleva a cabo un cálculo, no en lo que este cálculo logra. Sin embargo, nuestro nuevo objetivo es identificar las funciones que pueden calcularse con al menos un sistema computacional, así que debemos alejarnos de cualquier método específico para expresar o ejecutar los cálculos. En otras palabras, deseamos utilizar el término *computable* o *computable* para decir que una función puede calcularse por algún algoritmo, más que por un algoritmo determinado que pueda implantarse en un sistema específico.

Para obtener esta generalidad, adoptaremos un enfoque funcional para estudiar la computabilidad, con el cual nos centraremos en las funciones que se calcularán y no en la forma de calcularlas. Específicamente, utilizaremos el método que han seguido los matemáticos para el estudio de la teoría de funciones recursivas. El tema básico es comenzar con una colección de funciones, llamadas funciones iniciales, cuya sencillez es tal que no hay dudas acerca de su computabilidad, para luego mostrar que estas funciones se pueden combinar para formar otras cuya computabilidad se desprenda de la de las funciones originales. De esta forma obtendremos una colección de funciones que, a juicio de los investigadores, contiene todas las funciones computables. Así, si un

sistema computacional, como un lenguaje de programación o una máquina computadora, abarca todas estas funciones, concluimos que el sistema tiene todo el poder posible; de lo contrario, establecemos que dicho sistema es innecesariamente restrictivo.

Funciones parciales

Reiteramos que nuestro objetivo actual es identificar aquellas funciones que, en términos generales, son computables (sin importar el sistema computacional específico). Por supuesto, ello abarca una clase de funciones muy amplia, que incluye una diversidad de dominios y contradominios. Por lo tanto, nuestro primer paso es establecer una manera de lidiar con esta diversidad, para lo cual observamos que es posible codificar cualquier dato como una cadena de ceros y unos y, en el contexto de un sistema de codificación adecuado, es posible considerar entonces cualquier función computable como una función cuyas entradas sean tuplas de enteros no negativos. Sin embargo, esto no quiere decir que toda función computable tenga la forma

$$f: \mathbb{N}^m \to \mathbb{N}^n$$

donde m y n son enteros de $\mathbb{N}$. De hecho, la función *div* definida por

$$div(x, y) = \text{la parte entera de } x/y, \quad \text{donde } x, y \in \mathbb{N} \text{ y } y \neq 0$$

cuyo dominio contiene pares de enteros, debería estar en nuestra colección de funciones computables. Sin embargo, *div* no es una función de la forma

$$f: \mathbb{N}^2 \to \mathbb{N}$$

De hecho, no está definida para los pares donde el segundo componente es cero.

Para tomar en cuenta las funciones cuyos dominios no incluyen todo $\mathbb{N}^m$ para un m dado, presentamos el concepto de **función parcial.** Una función parcial de un conjunto X es aquella cuyo dominio constituye un subconjunto de X. De esta manera, podemos hablar de una función parcial de X sin dar a entender que el dominio de la función es todo el conjunto X. Como caso específico, la función *div* que antes definimos, aunque no es una función de $\mathbb{N}^2$, constituye una función parcial de $\mathbb{N}^2$.

Observe que una referencia a una función parcial de X no indica que el dominio tenga que ser un subconjunto propio de X. Para indicar que una función parcial de X efectivamente se encuentra indefinida por lo menos para un elemento de X, nos referimos a ella como **estrictamente parcial** o parcial en el sentido estricto de la palabra. Por otra parte, se llamará **función total** de X a una función parcial de X cuyo dominio es todo el conjunto X. Así, tanto la función *div* (antes descrita) como *más*, definida por

$$más(x, y) = x + y$$

son funciones parciales de $\mathbb{N}^2$. Para mayor precisión, *más* es una función total de $\mathbb{N}^2$, mientras que *div* es estrictamente parcial en $\mathbb{N}^2$ (Fig. 4.1).

Concluimos que al aplicar los sistemas de codificación adecuados, es posible identificar cualquier función computable como una función parcial de la forma $f: \mathbb{N}^m \to \mathbb{N}^n$, para m y n en $\mathbb{N}$. Así, nuestra búsqueda de todas las funciones computables se puede restringir a las funciones parciales de $\mathbb{N}^m$ a $\mathbb{N}^n$, donde m y n están en $\mathbb{N}$.

Por último, puesto que con frecuencia haremos referencia a n-tuplas arbitrarias es conveniente adoptar la sencilla notación $\bar{x}$ para representar una tupla de la forma $(x_1, x_2, \cdots; x_n)$ en aquellos casos donde no se requieren los detalles de la forma expandida.

Funciones iniciales

La base de la jerarquía de las funciones computables que veremos consiste en el conjunto de las **funciones iniciales.** Una de estas funciones es la **función cero**, representada por ζ. Establece una correspondencia entre la cero-tupla (la tupla vacía) y 0, y se escribe $\zeta() = 0$. Así, ζ corresponde al proceso de escribir

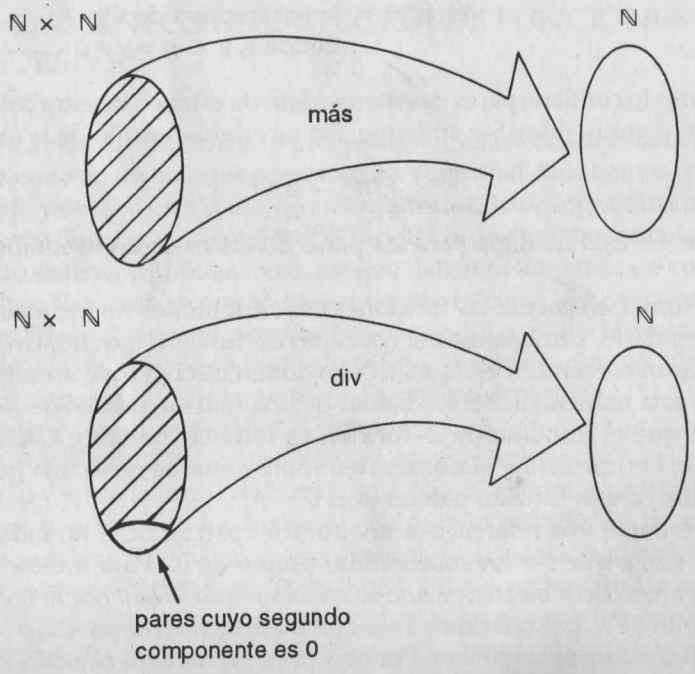


Figura 4.1 La función total *más* y la función estrictamente parcial *div*

un cero en un pedazo de papel en blanco o registrar un cero en una celda de memoria vacía de algún computador digital moderno. Como ambas actividades están de acuerdo con nuestro concepto intuitivo de un proceso computacional, aceptamos con facilidad que ζ es una función que debe clasificarse como computable.

Otra función inicial, representada por σ, establece una correspondencia entre tuplas de un componente, de manera que $\sigma(x) = x + 1$, para cada entero no negativo x. En otras palabras, σ produce el sucesor de su valor de entrada y por esto se conoce como **función sucesor.** Otra manera de considerar σ es que suma uno a su valor de entrada. Viéndolo así, también debemos considerar a σ como una función computable, pues hace tiempo que conocemos un proceso computacional para la suma de enteros.

Para completar la clase de funciones iniciales, incluimos la colección de funciones conocidas como **proyecciones.** Cada una de estas funciones extrae como salida un componente específico de su tupla de entrada. Para representar una función de proyección, utilizamos el símbolo π junto con un supraíndice para indicar el tamaño de su entrada y un supraíndice para indicar el componente que se extrae. Por ejemplo, la función π_2^3 establece una correspondencia entre $\mathbb{N}^3$ y $\mathbb{N}$ asociando cada tupla de tres componentes con el segundo componente de dicha tupla. De esta forma, $\pi_2^3(7, 6, 4) = 6$, $\pi_1^2(5, 17) = 5$, $\pi_2^2(5, 17) = 17$, y $\pi_1^1(8) = 8$ (como caso especial, consideramos que π_0^n establece una correspondencia entre, n-tuplas y tuplas vacías, por lo que $\pi_0^2(6, 5) = ()$).

Al igual que con las demás funciones iniciales, es fácil establecer que las proyecciones deben estar en la clase de las funciones computables. Por ejemplo, podríamos calcular la función π_m^n aplicando el procedimiento de rastrear la n-tupla de entrada hasta encontrar el componente m-ésimo y luego extraer el valor entero que allí se encuentra.

Las funciones iniciales forman la base de la jerarquía que existe en la teoría de funciones recursivas. Por supuesto, se trata de una base muy sencilla cuyas funciones, por sí solas, no pueden lograr mucho. Por lo tanto, nuestra siguiente tarea es investigar cómo pueden emplearse estas funciones para construir otras más complejas.

Funciones recursivas primitivas

Una forma de construir funciones más complejas a partir de las iniciales es la denominada **combinación.** La combinación de dos funciones $f: \mathbb{N}^k \to \mathbb{N}^m$ y $g: \mathbb{N}^k \to \mathbb{N}^n$ es la función $f \times g: \mathbb{N}^k \to \mathbb{N}^{m+n}$ definida por $f \times g(\bar{x}) = (f(\bar{x}), g(\bar{x}))$, donde $\bar{x}$ es una k-tupla. Es decir, la función $f \times g$ toma entradas en forma de k-tuplas y produce salidas en forma de $m + n$ - tuplas cuyos primeros m componentes consisten en la salida de f y los otros n son la salida de g. Así, $\pi_1^3 \times \pi_3^3(4, 6, 8) = (4, 8)$.

Suponiendo que hay forma de calcular las funciones f y g, podemos calcular $f \times g$ calculando primero por separado f y g y luego combinando sus salidas para formar la salida de $f \times g$. Llegamos entonces a la conclusión de que la combinación de funciones computables también es computable.

La **composición** representa otro método para formar funciones más complejas. La composición de dos funciones f: $\mathbb{N}^k \rightarrow \mathbb{N}^m$ y g: $\mathbb{N}^m \rightarrow \mathbb{N}^n$ es la función $g \circ f$: $\mathbb{N}^k \rightarrow \mathbb{N}^n$ definida por $g \circ f(\bar{x}) = g(f(\bar{x}))$, donde $\bar{x}$ es una k-tupla. Así, para encontrar la salida de $g \circ f$, primero aplicamos f a la entrada y después g a la salida de f. Por ejemplo, $\sigma \circ \zeta(\,) = 1$ ya que $\zeta(\,) = 0$ y $\sigma(0) = 1$.

Como hicimos con la combinación, planteamos que la composición de dos funciones computables también debe clasificarse como computable. A fin de cuentas, si hay formas de calcular f y g, podríamos calcular $g \circ f$ calculando primero f y luego usando su salida como entrada para el cálculo de g.

La última técnica de construcción de funciones que veremos en esta etapa es lo que se denomina **recursividad primitiva**. Supongamos que queremos definir una función f:$\mathbb{N}^2 \rightarrow \mathbb{N}$ tal que $f(x, y)$ sea el número de nodos en un árbol completo y equilibrado en el cual cada nodo que no es una hoja tiene exactamente x hijos y cada ruta de la raíz a un nodo contiene y arcos (se dice que este árbol tiene una profundidad y). Nuestro primer paso sería observar que cada nivel del árbol tiene exactamente x^n nodos, donde n es el número de nivel (Fig. 4.2). De esta forma, con un valor fijo de x, para determinar el número de nodos de un árbol con profundidad $y + 1$, nos basta con sumar $x^{y+1} = x^y x$ al número de nodos del árbol de profundidad y. Al combinar esto con el hecho de que un árbol que consiste en sólo un nodo raíz contiene x^0 nodos, podemos definir f recursivamente con el par de fórmulas

$$f(x, 0) = x^0 \qquad (1)$$
$$f(x, y + 1) = f(x, y) + x^y x \qquad (2)$$

Con base en esta definición, podemos calcular $f(3, 2)$ como sigue:

$$\begin{aligned} f(3,2) &= f(3,1) + 3^1 3 && \text{(por la fórmula 2)} \\ &= f(3,1) + 9 \\ &= f(3,0) + 3^0 3 + 9 && \text{(por la fórmula 2)} \\ &= f(3,0) + 3 + 9 \\ &= f(3,0) + 12 \\ &= 3^0 + 12 && \text{(por la fórmula 1)} \\ &= 13 \end{aligned}$$

En un contexto más general, lo que hemos hecho es definir f en términos de otras dos funciones. Una de ellas es g: $\mathbb{N} \rightarrow \mathbb{N}$ tal que $g(x) = 1$ para cada $x \in \mathbb{N}$; la otra es h: $\mathbb{N}^3 \rightarrow \mathbb{N}$ tal que $h(x, y, z) = z + x^y x$. Al emplear estas funciones, f está definida recursivamente por las fórmulas

$$f(x, 0) = g(x)$$
$$f(x, y + 1) = h(x, y, f(x, y))$$

En este caso decimos que f está construida a partir de g y h por medio de recursividad primitiva. De manera general, la recursividad primitiva es una técnica que nos permite construir una función f que establece la correspondencia entre $\mathbb{N}^{k+1}$ y $\mathbb{N}^m$ a partir de otras dos funciones g y h que relacionan

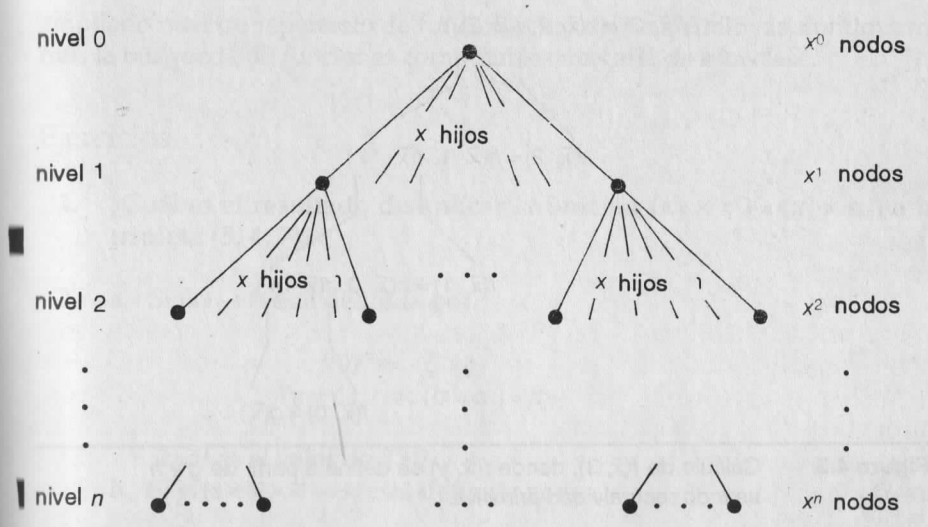


Figura 4.2 Descripción por niveles de un árbol equilibrado en el cual cada nodo que no es una hoja tiene x hijos

$\mathbb{N}^k$ con $\mathbb{N}^m$ y $\mathbb{N}^{k+m+1}$ con $\mathbb{N}^m$, respectivamente, como lo describen las ecuaciones

$$f(\bar{x}, 0) = g(\bar{x})$$
$$f(\bar{x}, y + 1) = h(\bar{x}, y, f(\bar{x}, y))$$

donde $\bar{x}$ representa una k-tupla. Es decir, si el último componente de la tupla de entrada es 0, entonces la salida de f se obtiene eliminando este último componente y aplicando g a la k-tupla restante. Si el último componente de la entrada de f no es 0, entonces la salida de f está determinada por la aplicación de h a la $k + m + 1$ -tupla formada al combinar los primeros k elementos de la tupla original, el predecesor del último componente de la entrada original y el resultado de la aplicación de f a la $k + 1$ -tupla que se obtiene al reducir en uno el último componente de la entrada original. Así, el cálculo de $f(\bar{x}, 3)$ comprende el cálculo de $f(\bar{x}, 2)$, que a su vez requiere el cálculo de $f(\bar{x}, 1)$, que requiere el cálculo de $f(\bar{x}, 0)$, como se representa en la figura 4.3.

Si empleamos la recursividad primitiva junto con otras funciones y operaciones que hemos presentado, la función *más*: $\mathbb{N}^2 \rightarrow \mathbb{N}$ (cuya salida es la suma de sus componentes de entrada) se puede definir de la siguiente manera:

$$\textit{más}(x, 0) = \pi_1^1(x)$$
$$\textit{más}(x, y + 1) = \sigma \circ \pi_3^3(x, y, \textit{más}(x, y))$$

De manera informal, esto quiere decir que $x + 0$ es x, mientras que $x + (y + 1)$ se obtiene (recursivamente) encontrando al sucesor de $x + y$.

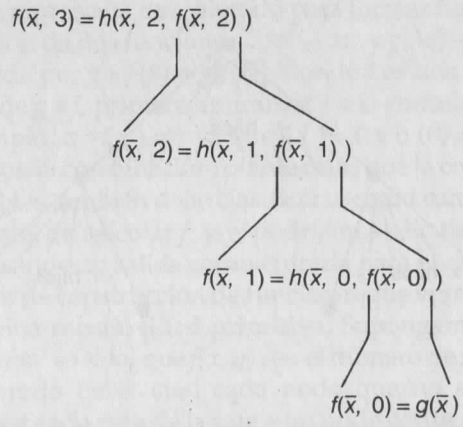


Figura 4.3 Cálculo de $f(\bar{x}, 3)$, donde $f(\bar{x}, y)$ se define a partir de g y h usando recursividad primitiva

Para concluir nuestra introducción a la recursividad primitiva, observamos que una función construida por recursividad primitiva a partir de funciones computables debe considerarse computable. Específicamente, si f se define por recursividad primitiva con base en las funciones computables g y h, podemos calcular $f(\bar{x}, y)$ calculando primero $f(\bar{x}, 0)$, luego $f(\bar{x}, 1)$, después $f(\bar{x}, 2)$, hasta llegar a $f(\bar{x}, y)$.

Ahora estamos preparados para definir la clase de funciones que se encuentra encima de las funciones iniciales en la jerarquía de la teoría de funciones recursivas. Se trata de las **funciones recursivas primitivas**, consistentes en todas aquellas funciones que pueden construirse a partir de las funciones iniciales aplicando un número finito de combinaciones, composiciones y recursividades primitivas. Es evidente que esta clase es una extensión propia de las funciones iniciales, ya que contiene las funciones iniciales y la función *más* que construimos a partir de σ y algunas proyecciones. De hecho, se sabe que la clase de las funciones recursivas primitivas es muy extensa; incluye la mayoría, si no todas, de las funciones totales que se requieren en las aplicaciones de computador tradicionales. En la siguiente sección daremos bases para esta afirmación, considerando diversas funciones recursivas primitivas.

Por último, debemos señalar que si $f: \mathbb{N}^m \to \mathbb{N}^n$ es una función recursiva primitiva, entonces f debe ser total. De hecho, las funciones iniciales son totales, y las técnicas de construcción (combinación, composición y recursividad primitiva) producen funciones totales cuando se aplican a funciones totales. Por esto, la extensa naturaleza de las funciones recursivas primitivas no significa que esta clase complete nuestra jerarquía de funciones computables. Por ejemplo, *div* es computable pero no recursiva primitiva, pues no es total (de hecho, veremos que existen funciones totales computables que no son recursivas primitivas). Por consiguiente, una vez que en la próxima sección hayamos

ampliado nuestro repertorio de funciones recursivas primitivas, continuaremos la búsqueda de funciones computables más allá de esta clase.

Ejercicios

1. ¿Cuál es el resultado de aplicar la función $(\pi_2^2 \times \pi_1^2) \circ (\pi_1^3 \times \pi_3^3)$ a la tripleta (5, 4, 7)?

2. a. Si $f: \mathbb{N} \to \mathbb{N}$ está definida por

$$\begin{aligned} f(0) &= \zeta\,() \\ f(y+1) &= (\sigma \circ \sigma) \circ f(y) \end{aligned}$$

¿cuál es el valor de $f(3)$?

b. Si $g: \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ está definida por

$$\begin{aligned} g(x, y, 0) &= \pi_1^2(x, y) \\ g(x, y, z+1) &= (\sigma \circ g)(x, y, z) \end{aligned}$$

¿cuál es el valor de $g(5, 7, 2)$?

3. Explique cómo puede calcularse el valor de *más*(5, 2) con base en la definición de *más* proporcionada en esta sección.

4. Muestre que cualquier permutación de una n-tupla es una función recursiva primitiva.

4.2 ALCANCE DE LAS FUNCIONES RECURSIVAS PRIMITIVAS

El primer objetivo de esta sección es ampliar el repertorio de funciones que sabemos son recursivas primitivas. Esto no sólo apoyará nuestra afirmación de que la clase de las funciones recursivas primitivas incluye todas las funciones totales que pueden encontrarse en las aplicaciones típicas de un computador, sino que además nos proporcionará ejemplos específicos que serán de utilidad en las secciones subsecuentes. El segundo objetivo de esta sección es esclarecer la diferencia entre la clase de las funciones recursivas primitivas y la clase de las funciones totales computables.

Antes de iniciar, debemos establecer algunas formas de notación convenientes. Suponga que queremos representar una función $h: \mathbb{N}^3 \to \mathbb{N}$ que da como resultado la suma del primer y el tercer componentes de su entrada. Esta función se podría representar con más $\circ\, (\pi_1^3 \times \pi_3^3)$. Es decir, para obtener $h(x, y, z)$ extraemos el primer y el tercer componentes de la entrada por medio de las

proyecciones π_1^3 y π_3^3, luego combinamos los resultados en un par $(\pi_1^3 \times \pi_3^3)$ y por último aplicamos *más* a este par. Así, la notación *más* $\circ$ $(\pi_1^3 \times \pi_3^3)$ describe a h de manera bastante adecuada. Sin embargo, esta notación puede ser muy complicada y difícil de descifrar, sobre todo si se compara con las expresiones $h(x, y, z) = más(x, z)$ o $h(x, y, z) = x + z$. Utilizaremos con frecuencia las expresiones menos formales, para que sea más legible.

Ejemplos de funciones recursivas primitivas

Comencemos ahora nuestra investigación de las funciones recursivas primitivas considerando la colección de las **funciones constantes**, las cuales producen una salida fija, predeterminada, sin importar cuál sea la entrada. Las funciones constantes cuyas salidas son tuplas de un elemento están representadas por K_m^n, donde n es un entero no negativo que indica la dimensión del dominio de la función y m es un entero no negativo que indica el valor de salida de la función. Así, K_5^3 establece la correspondencia entre cualquier tupla de tres elementos y el valor 5, mientras que K_3^5 lo hace entre cualquier tupla de cinco elementos y el valor 3.

Es fácil producir las funciones constantes de la forma K_m^0 (correspondientes a escribir el valor m en un pedazo de papel en blanco). Basta comenzar con ζ y luego, por medio de la composición, aplicar σ el número de veces suficiente para incrementar la salida hasta llegar al valor m. Por ejemplo, K_2^0 sería $\sigma \circ \sigma \circ \zeta$. Llegamos a la conclusión de que cualquier función constante de la forma K_m^0 es recursiva primitiva.

Para mostrar que las funciones de la forma K_m^n, donde n es mayor que cero, son recursivas primitivas, aplicamos inducción en n. En general, si sabemos que K_m^i, es recursiva primitiva para todo i menor que n, entonces podemos definir K_m^n como

$$K_m^n(\bar{x}, 0) = K_m^{n-1}(\bar{x})$$
$$K_m^n(\bar{x}, y + 1) = \pi_{n+2}^{n+2}(\bar{x}, y, K_m^n(\bar{x}, y))$$

También son recursivas primitivas las funciones constantes cuyas salidas tienen más de un componente, ya que no son más que combinaciones de funciones de la forma K_m^n. Por ejemplo, la función que establece la correspondencia entre cualquier tripleta y el par (2, 5) es $K_2^3 \times K_5^3$.

(Para evitar notaciones complicadas, en aquellos casos en que la dimensión del dominio sea evidente o no tenga importancia para el análisis, podemos representar una función constante con el valor de su salida. De esta forma, en ocasiones escribiremos 6 en vez de K_6^3 o únicamente (2, 5) en vez de $K_2^3 \times K_5^3$).

Con base en las funciones constantes y la función *más* de la sección anterior, podemos mostrar de la manera siguiente que la multiplicación es recursiva primitiva:

$$mult(x, 0) = K_0^1(x)$$
$$mult(x, y + 1) = h(x, y, mult(x, y))$$

donde $h(x, y, z) = más(x, z)$; o, de manera más legible

$$mult(x, 0) = 0$$
$$mult(x, y + 1) = más(x, mult(x, y))$$

En forma parecida, podemos mostrar que la exponenciación es recursiva primitiva, definiendo $exp(x, y)$ como

$$exp(x, 0) = K_1^1(x)$$
$$exp(x, y + 1) = h(x, y, exp(x, y))$$

donde $h(x, y, z) = mult(x, z)$. Es decir,

$$exp(x, 0) = 1$$
$$exp(x, y + 1) = mult(x, exp(x, y))$$

Ahora consideremos la **función predecesor**, representada por *pred*. Esta función establece la correspondencia entre tuplas de un componente en forma tal que se establece la correspondencia de 0 a 0, pero los números mayores que 0 se hacen corresponder con su predecesor en el orden natural de $\mathbb{N}$ (de esta forma, $pred(1) = 0$, $pred(2) = 1$, $pred(3) = 2$, etc.). Para ver que *pred* es primitiva recursiva, observamos que puede definirse como

$$pred(0) = \zeta()$$
$$pred(y + 1) = \pi_1^2(y, pred(y))$$

En cierta forma, *pred* es la inversa de σ y, de hecho, puede usarse para definir la resta de la misma manera que usamos σ para desarrollar la suma. Específicamente, podemos definir la función *monus* (como menos, pero diferente) como

$$monus(x, 0) = \pi_1^1(x)$$
$$monus(x, y + 1) = h(x, y, monus(x, y))$$

donde $h(x, y, z) = pred(z)$. Es decir,

$$monus(x, 0) = x$$
$$monus(x, y + 1) = pred(monus(x, y))$$

Así, $monus(x, y)$ es $x - y$ si $x \geq y$, o de lo contrario es 0. Por las características semejantes entre la función primitiva recursiva *monus* y el concepto de la resta, con frecuencia *monus* (x, y) se representará como $x \dot{-} y$.

Una tarea computacional común es determinar si dos valores son iguales. Este proceso está modelado por la función $eq: \mathbb{N}^2 \to \mathbb{N}$, donde

$$eq(x, y) = \begin{cases} 1 \text{ si } x = y \\ 0 \text{ si } x \neq y \end{cases}$$

Se observa que la función *eq* es recursiva primitiva si se reconoce que

$$eq(x, y) = 1 \dot{-} ((y \dot{-} x) + (x \dot{-} y))$$

o, si se emplea una notación más formal, la función *eq* es

$$monus \circ (K_1^2 \times (más \circ ((monus \circ (\pi_2^2 \times \pi_1^2)) \times monus \circ (\pi_1^2 \times \pi_2^2))))$$

Por ejemplo,

$$\begin{aligned} eq(5, 3) &= 1 \dot{-} ((3 \dot{-} 5) + (5 \dot{-} 3)) \\ &= 1 \dot{-} (0 + 2) \\ &= 1 \dot{-} 2 \\ &= 0 \end{aligned}$$

y

$$\begin{aligned} eq(5, 5) &= 1 \dot{-} ((5 \dot{-} 5) + (5 \dot{-} 5)) \\ &= 1 \dot{-} (0 \dot{-} 0) \\ &= 1 \dot{-} 0 \\ &= 1 \end{aligned}$$

También podemos "negar" cualquier función $f: \mathbb{N}^n \to \mathbb{N}$ para producir otra función $\neg f: \mathbb{N}^n \to \mathbb{N}$ que sea 1 cuando f es 0 y 0 cuando f es 1, definiendo $\neg f$ como $monus \circ (K_1^n \times f)$, es decir, $\neg f(x) = 1 \dot{-} f(x)$. Por ejemplo,

$$\neg eq(x, y) = \begin{cases} 1 \text{ si } x \neq y \\ 0 \text{ si } x = y \end{cases}$$

es la función recursiva primitiva $monus \circ (K_1^2 \times eq)$.

Otra colección de funciones recursivas primitivas es aquella que comprende las que pueden definirse en una tabla en la cual se presenta explícitamente en una lista un número finito de entradas posibles junto con sus valores de salida correspondientes, y todas las demás entradas se asocian a un solo valor común. A estas funciones las llamaremos funciones tabulares. Un ejemplo sería la función f definida por

$$f(x) = \begin{cases} 3 \text{ cuando } x = 0 \\ 5 \text{ cuando } x = 4 \\ 2 \text{ en los demás casos} \end{cases}$$

que puede resumirse en la tabla de la figura 4.4. Para convencernos de que estas funciones son recursivas primitivas, primero reconocemos que la función característica κ_i de un valor i es recursiva primitiva. A fin de cuentas, la

$$\kappa_i(x) = \begin{cases} 1 \text{ si } x = i \\ 0 \text{ en los demás casos} \end{cases}$$

función puede expresarse como $monus(I_i, I_{i-1})$, donde

$$I_i = eq \circ ((monus \circ (\pi_1^1 \times K_i^1)) \times K_0^1)$$

o sea, $I_i(x) = eq(x \dot{-} i, 0)$ (véase Fig. 4.5). A continuación vemos que cualquier función tabular no es más que la suma finita del producto de funciones

entrada	salida
0	3
4	5
otros	2

Figura 4.4 Descripción tabular de una función

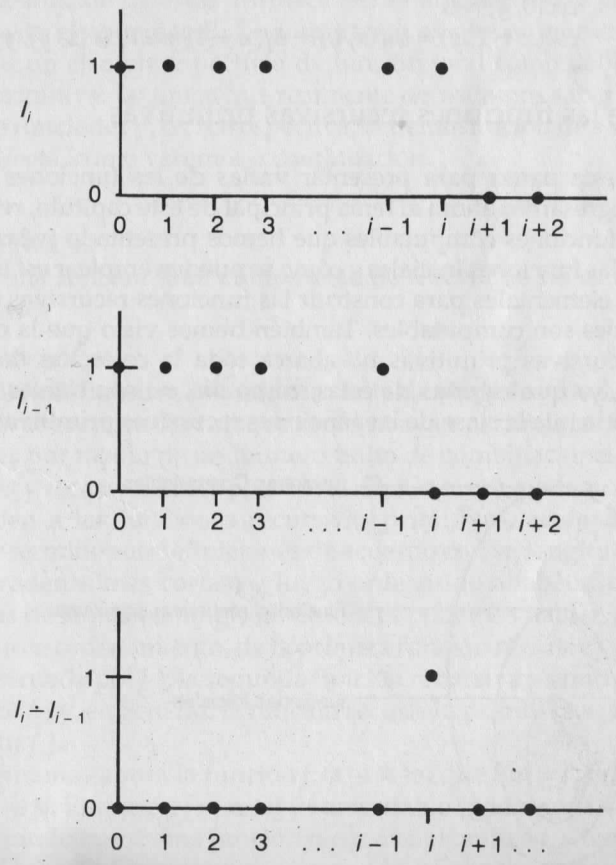


Figura 4.5 Funciones I_i, I_{i-1}, e $I_i \dot{-} I_{i-1}$

características, constantes y características negadas. Por ejemplo, la función f definida por la tabla de la figura 4.4 equivale a

$$mult(3, \kappa_0) + mult(5, \kappa_4) + mult(2, mult(\neg\kappa_0, \neg\kappa_4))$$

Como último ejemplo, mostramos que la función $coc:\mathbb{N}^2 \rightarrow \mathbb{N}$ definida por

$$coc(x, y) = \begin{cases} \text{la parte entera de } x \div y \text{ si } y \neq 0 \\ 0 \text{ si } y = 0 \end{cases}$$

es primitiva recursiva (la notación *coc* representa el cociente). Para esto, primero observamos que aunque la recursividad primitiva se define formalmente empleando como índice el último componente de la tupla de entrada, el uso de proyecciones y combinaciones nos permite aplicar la recursividad primitiva con base en otros índices y permanecer en la clase de funciones recursivas primitivas. Así, *coc* es recursiva primitiva ya que puede definirse con

$$coc(0, y) = 0$$
$$coc(x + 1, y) = coc(x, y) + eq(x + 1, mult(coc(x, y), y) + y)$$

Más allá de las funciones recursivas primitivas

Después de esta pausa para presentar varias de las funciones recursivas primitivas, regresamos ahora al tema principal de este capítulo, repasando la jerarquía de funciones computables que hemos presentado (véase Fig. 4.6). Hemos visto las funciones iniciales y cómo se pueden emplear estas funciones computables elementales para construir las funciones recursivas primitivas, todas las cuales son computables. También hemos visto que la clase de las funciones recursivas primitivas no abarca toda la colección de funciones computables, ya que algunas de éstas, como *div*, no son totales. De hecho, hemos señalado que la clase de las funciones recursivas primitivas no abarca siquiera

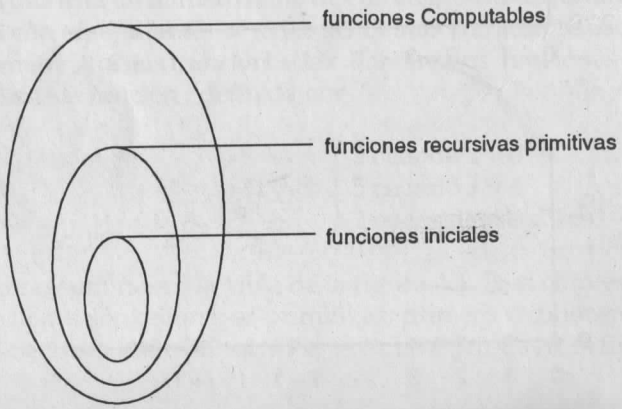


Figura 4.6 Jerarquía de las funciones presentadas hasta ahora

todas las funciones totales computables. En lo que resta de la presente sección trataremos de aclarar esta diferencia entre las funciones recursivas primitivas y las funciones totales computables.

En una etapa del desarrollo de la teoría de funciones recursivas, algunas personas pensaron que la clase de las funciones recursivas primitivas podría contener todas las funciones totales computables. Sin embargo, estas conjeturas fueron acalladas por el descubrimiento de funciones totales computables que no eran recursivas primitivas. En 1928, W. Ackermann presentó un ejemplo, la función $A:\mathbb{N}^2 \rightarrow \mathbb{N}$ (que ahora se conoce como función de Ackermann), definida por las ecuaciones

$$A(0, y) = y + 1$$
$$A(x + 1, 0) = A(x, 1)$$
$$A(x + 1, y + 1) = A(x, A(x + 1, y))$$

la cual demostró ser computable y total, pero no recursiva primitiva.

La demostración de estas afirmaciones es algo tediosa y por lo tanto se pospone hasta el apéndice B. Lo importante es que en nuestro estudio no necesitamos un ejemplo específico de función total computable que no sea recursiva primitiva. Lo único que realmente necesitamos saber es que existe este tipo de funciones y, en retrospectiva, la demostración de su existencia es bastante directa, como veremos a continuación.

TEOREMA 4.1
Existe una función total computable de $\mathbb{N}$ a $\mathbb{N}$ que no es recursiva primitiva.

DEMOSTRACIÓN
Es posible definir con una cadena finita de símbolos a toda función primitiva recursiva de $\mathbb{N}$ a $\mathbb{N}$ que se construya a partir de las funciones iniciales por medio de un número finito de combinaciones, composiciones y recursividades primitivas. Por lo tanto, podemos asignar un orden a las funciones recursivas primitivas, acomodando en primer término sus definiciones de acuerdo con su longitud (primero las cadenas más cortas) y luego ordenando alfabéticamente las cadenas de la misma longitud. Entonces podemos hablar, en términos de este ordenamiento, de la primera función recursiva primitiva (representada por f_1), la segunda función recursiva primitiva (denotada por f_2) y, en general, la función recursiva primitiva n (representada por f_n).

Definamos ahora la función $f: \mathbb{N} \rightarrow \mathbb{N}$ tal que $f(n) = f_n(n) + 1$ para todo $n \in \mathbb{N}$. Entonces, f es total y computable (podemos calcular $f(n)$ encontrando la enésima función recursiva primitiva f_n y luego calculando $f_n(n) + 1$). Sin embargo, f no puede ser recursiva primitiva (si lo fuera, tendría que ser f_m para algún $m \in \mathbb{N}^+$, pero entonces $f(m)$

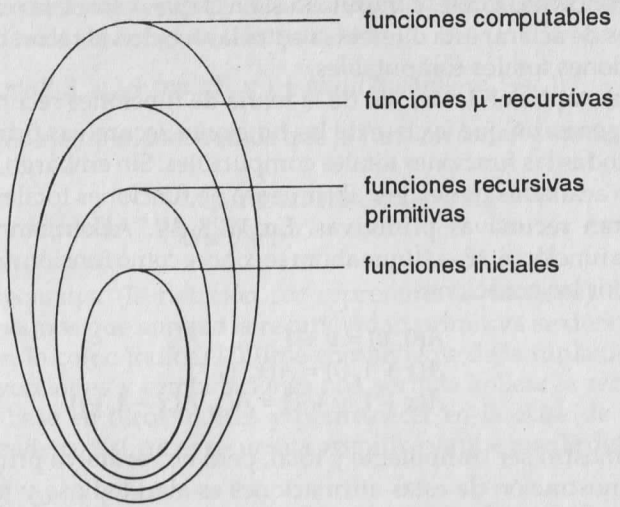


Figura 4.7 Figura 4.6 revisada

sería igual a $f_m(m)$, lo cual no puede ser verdad ya que se definió que $f(m)$ es $f_m(m) + 1$).

Llegamos entonces a la conclusión de que f tiene las características requeridas por el teorema.

∎

La clase de las funciones totales computables se conoce como clase de las **funciones μ-recursivas**. Con base en esto, el teorema 4.1 indica que la figura 4.6 se puede refinar para obtener la figura 4.7. En la próxima sección examinaremos la porción de la figura 4.7 que yace fuera de las funciones μ recursivas.

Ejercicios

1. Muestre que la función característica de cualquier subconjunto finito de ℕ es recursiva primitiva.

2. Muestre que la función *impar*: ℕ → ℕ definida por

$$impar(x) = \begin{cases} 1 \text{ si } x \text{ es impar} \\ 0 \text{ si } x \text{ es par} \end{cases}$$

es recursiva primitiva.

3. Muestre que la función *dif*: $\mathbb{N}^2 \to \mathbb{N}$ definida por

$$dif(x, y) = \text{valor absoluto de } x - y$$

es recursiva primitiva.

4. Muestre que la función factorial $f(x) = x!$ es recursiva primitiva.

5. Encuentre $A(3, 1)$, donde A es la función de Ackermann.

4.3 FUNCIONES RECURSIVAS PARCIALES

Nuestro estudio de las funciones computables nos ha llevado a las clases de las funciones iniciales, las funciones recursivas primitivas y las funciones totales computables, todas las cuales son totales. Ahora ampliaremos nuestro estudio para incluir las funciones parciales computables.

Definición de las funciones recursivas parciales

Para ampliar nuestro estudio de las funciones computables más allá de las funciones totales computables, aplicamos la técnica de construcción conocida como **minimalización**. Esta técnica nos permite construir una función $f: \mathbb{N}^n \to \mathbb{N}$ a partir de otra función $g: \mathbb{N}^{n+1} \to \mathbb{N}$ declarando a $f(\bar{x})$ como la menor y en ℕ tal que $g(\bar{x}\ y) = 0$ y $g(\bar{x}, z)$ esté definida para todos los enteros no negativos z menores a y. Esta construcción se representa con la notación $f(\bar{x}) = \mu y[g(\bar{x}, y) = 0]$, que se lee "$f(\bar{x})$ es igual a la menor y para la cual $g(\bar{x}, y)$ es cero y $g(\bar{x}, z)$ está definida para todos los enteros no negativos z menores que y".

Como ejemplo, suponga que $g(x, y)$ se define de acuerdo con la tabla de la figura 4.8 y que $f(x)$ se define como $\mu y[g(x, y) = 0]$. Entonces, $f(0) = 4$, $f(1) = 2$ y $f(2)$ no está definida (aunque 4 es el menor valor de y para el cual $g(2, y) = 0$, $g(2, z)$ no está definida para todos los valores de z menores que 4, por lo cual $f(2)$ no está definida). En lo que se refiere al valor de $f(3)$, la tabla no nos proporciona suficiente información para determinar si se encuentra definida o no.

Queremos recalcar que, como sucede en el ejemplo anterior, la minimalización puede producir funciones que no están definidas para ciertas entradas. Otro ejemplo es la función $f: \mathbb{N} \to \mathbb{N}$ definida por $f(x) = \mu y[más(x, y) = 0]$. En este caso f es 0 con 0, pero no está definida para las demás entradas (para $x > 0$ no hay ningún y en ℕ tal que $x + y = 0$). Otro ejemplo es la función de cociente entero *div*: $\mathbb{N}^2 \to \mathbb{N}$ definida por

$$div(x, y) = \begin{cases} \text{parte entera de } x/y \text{ si } y \neq 0 \\ \text{no definida si } y = 0 \end{cases}$$

$g(0,0) = 2$	$g(1,0) = 3$	$g(2,0) = 8$	$g(3,0) = 2$ ...
$g(0,1) = 3$	$g(1,1) = 4$	$g(2,1) = 3$	$g(3,1) = 6$ ...
$g(0,2) = 1$	$g(1,2) = 0$	$g(2,2) =$ no definido	$g(3,2) = 7$ ...
$g(0,3) = 5$	$g(1,3) = 2$	$g(2,3) = 6$	$g(3,3) = 2$ ...
$g(0,4) = 0$	$g(1,4) = 0$	$g(2,4) = 0$	$g(3,4) = 8$ ...
$g(0,5) = 1$	$g(1,5) = 0$	$g(2,5) = 1$	$g(3,5) = 4$ ...
⋮	⋮	⋮	⋮

Figura 4.8 Valor de $g(x, y)$ para varias x y y

que con la minimalización puede construirse como

$$div(x, y) = \mu t[((x + 1) \dot{-} (mult(t, y) + y)) = 0]$$

Por otra parte, en algunos casos la minimalización produce funciones totales como $f(x) = \mu y[monus(x, y) = 0]$, que no es más que la función identidad: el menor y tal que $x - y = 0$ es el x mismo.

Consideremos ahora la computabilidad de una función definida mediante la minimalización. Si la función parcial g es computable, entonces se puede calcular $f(\bar{x}) = \mu y[g(\bar{x}, y) = 0]$ calculando los valores $g(\bar{x}, 0)$, $g(\bar{x}, 1)$, $g(\bar{x}, 2)$, etc., hasta obtener el valor 0 o llegar a un valor z para el cual $g(\bar{x}, z)$ no esté definida. En el primer caso, el valor de $f(\bar{x})$ es el valor de y para el cual se encontró que $g(\bar{x}, y)$ era 0; en el segundo caso, $f(\bar{x})$ no está definida. Por ende, el proceso de minimalización aplicado a una función parcial computable produce una función parcial computable.

Esta observación nos permite ampliar nuestro repertorio de funciones computables más allá de los límites de las funciones recursivas primitivas, hasta la clase conocida como **funciones recursivas parciales.** Para ser más precisos, la clase de las funciones recursivas parciales es la clase de las funciones parciales que pueden construirse a partir de las funciones iniciales aplicando un número finito de combinaciones, composiciones, recursividades primitivas y minimalizaciones. En este contexto, la función *div* previamente definida es recursiva parcial pero no recursiva primitiva.

Observe que la construcción de una función parcial recursiva puede requerir la aplicación de combinaciones, composiciones o recursividades parciales a funciones estrictamente parciales. En estos casos, las definiciones de estas operaciones se extienden en dos formas evidentes: la combinación de dos funciones parciales f y g está definida en $\bar{x}$ si y sólo si f y g están definidas en $\bar{x}$, y la composición de f y g está definida en $\bar{x}$ si y sólo si g está definida en $\bar{x}$ y f está definida en $g(\bar{x})$. En forma similar, f está definida en $(\bar{x}, y)$ por recursividad primitiva a partir de g y h si y sólo si g está definida en $\bar{x}$ y h está definida en $(\bar{x}, z, f(\bar{x}, z))$, para todos los enteros no negativos z menores que y.

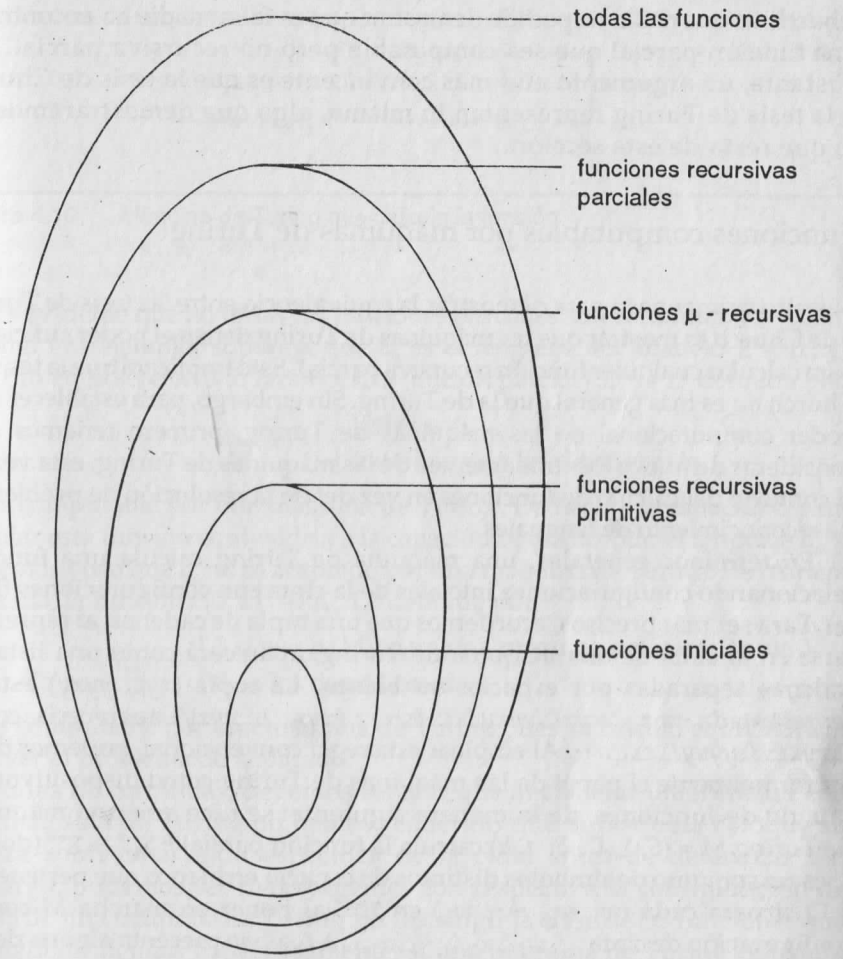


Figura 4.9 Jerarquía de las clases de funciones en la teoría de funciones recursivas

Una vez más subrayamos que el término parcial no quiere decir que todas las funciones de la clase de las funciones recursivas parciales sean parciales en el sentido estricto. De hecho, la clase contiene las funciones μ recursivas, todas las cuales son totales (véase Fig. 4.9).

Con la presentación de las funciones recursivas parciales llegamos al final de nuestra jerarquía de funciones computables. Así como la tesis de Turing propone que la clase de las máquinas de Turing posee el poder computacional de cualquier sistema de computación, se plantea que la clase de las funciones recursivas parciales contiene todas las funciones parciales computables. Esta tesis se conoce como **tesis de Church**, aunque Church la propuso originalmente en un contexto un tanto distinto. Un hecho que apoya la tesis de

Church es que nadie ha podido demostrar que es falsa: nadie ha encontrado una función parcial que sea computable pero no recursiva parcial. No obstante, un argumento aún más convincente es que la tesis de Church y la tesis de Turing representan lo mismo, algo que demostraremos en lo que resta de esta sección.

Funciones computables por máquinas de Turing

Nuestro primer paso para demostrar la equivalencia entre las tesis de Turing y de Church es mostrar que las máquinas de Turing tienen el poder suficiente para calcular cualquier función recursiva parcial. Esto implicará que la tesis de Church no es más general que la de Turing. Sin embargo, para establecer este poder computacional de las máquinas de Turing, primero tenemos que considerar de nuevo los fundamentos de las máquinas de Turing, esta vez en el contexto del cálculo de funciones en vez del de la resolución de problemas de reconocimiento de lenguajes.

En términos generales, una máquina de Turing calcula una función relacionando configuraciones iniciales de la cinta con configuraciones finales. Para ser más precisos, acordemos que una tupla de cadenas, al representarse en la cinta de una máquina de Turing, aparecerá como una lista de cadenas separadas por espacios en blanco. La tupla $(xyz, yyxy)$ estaría representada por $\triangle xyx\triangle yyxy\triangle\triangle\triangle\cdots$, $(xyx, \lambda, yyxy)$ aparecería como $\triangle xyx\triangle\triangle yyxy\triangle\triangle\triangle\cdots$. Al emplear esta regla convencional, podemos definir formalmente el papel de las máquinas de Turing como dispositivos de cálculo de funciones, de la manera siguiente: se dice que una máquina de Turing $M = (S, \Sigma, \Gamma, \delta, \iota, h)$ calcula la función parcial $f: \Sigma^{*m} \to \Sigma_1^{*n}$ (donde Σ_1 es un conjunto de símbolos distintos de espacio en blanco que pertenecen a Γ) si para cada $(w_1, w_2, \cdots, w_m)$ en Σ^{*m}, al poner en marcha M con la configuración de cinta $\triangle w_1\triangle w_2\triangle\cdots\triangle w_m\triangle\triangle\triangle\cdots$ se presenta alguna de las siguientes consecuencias.

1. En aquellos casos en que $f(w_1, w_2, \cdots, w_m)$, está definida, M se detiene con su cinta conteniendo $\triangle v_1\triangle v_2\triangle\cdots\triangle v_n\triangle\triangle\triangle\cdots$, donde $(v_1, v_2, \cdots, v_n) = f(w_1, w_2, \cdots, w_m)$.
2. En aquellos casos en que $f(w_1, w_2, \cdots, w_m)$ no está definida, M nunca se detiene (quizás por una terminación anormal).

Se dice que una función parcial que puede calcularse con alguna máquina de Turing es **computable por una máquina de Turing.**

La figura 4.10 muestra una máquina de Turing para calcular la función que establece una correspondencia entre cada tripleta de cadenas (w_1, w_2, w_3) y el par (w_1, w_3). Otro ejemplo es la máquina $\to L$ que calcula la función parcial $f: \Sigma^* \to \Sigma^*$, la cual no está definida para ninguna cadena en Σ^*.

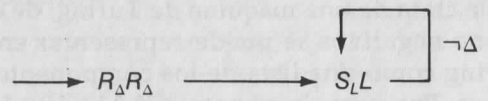


Figura 4.10 Máquina de Turing que calcula la función $f(w_1, w_2, w_3) = (w_1, w_3)$

Es evidente que no todas las funciones parciales son computables según Turing. Por ejemplo, suponga que L_0 es el lenguaje del alfabeto $\Sigma = \{x, y\}$ descrito en la sección 3.5. Entonces, la función parcial $f: \Sigma^* \to \Sigma^*$ definida por

$$f(w) = \begin{cases} y \text{ si } w \in L_0 \\ \text{indefinida en los demás casos} \end{cases}$$

no es computable por una máquina de Turing. De hecho, la capacidad para calcular esta función equivaldría a la capacidad para aceptar el lenguaje L_0, y ya hemos visto que L_0 no es aceptado por una máquina de Turing. De manera parecida, la función $f: \{0, 1\}^* \to \{0, 1\}^*$ definida por

$$f(w) = \begin{cases} 1 \text{ si } w = \rho(M) \text{ para una máquina autoterminante } M \\ 0 \text{ en los demás casos} \end{cases}$$

no es computable por una máquina de Turing pues su cálculo equivaldría a resolver el problema de la parada.

Antes de continuar, debemos señalar que al calcular una función con una máquina de Turing no hemos requerido que la máquina coloque su cabeza sobre una celda específica de la cinta antes de detenerse. Sin embargo, podemos ser más exigentes con respecto a la configuración de salida de una máquina de Turing sin restringir la clase de las funciones que se pueden calcular. Específicamente, si una máquina de Turing calculara una función parcial de acuerdo con la definición anterior, podríamos modificar la máquina para que antes de detenerse devolviera la cabeza a la celda del extremo izquierdo de la cinta (en esencia, se trata de marcar la celda del extremo izquierdo, simular la máquina original y devolver la cabeza a la celda marcada si la simulación llega al estado de parada de la máquina original). Con frecuencia es más conveniente trabajar con esta máquina modificada que con una menos restringida, por lo que acudiremos a estas máquinas cuando resulte conveniente.

Computabilidad con una máquinas de Turing de funciones recursivas parciales

Ahora el objetivo es mostrar que las funciones recursivas parciales están incluidas en la clase de funciones parciales computables por una máquinas de Turing.

Establezcamos que se representarán con notación binaria los enteros no negativos en la cinta de una máquina de Turing; de esta manera, una n-tupla enteros no negativos se puede representar en la cinta de una máquina de Turing como una lista de los componentes separados por espacios en blanco. Por ejemplo, el patrón △11△10△100△ representaría la tupla de tres componentes (3, 2, 4), y △10△0△11△1△ representaría la tupla de cuatro componentes (2, 0, 3, 1).

Basándonos en esta notación, podemos utilizar las máquinas de Turing con alfabeto {0, 1} para calcular funciones parciales de la forma $f: \mathbb{N}^m \to \mathbb{N}^n$, donde m y n son enteros no negativos. Así mismo, las máquinas de Turing se pueden considerar como posibles instrumentos para calcular las funciones recursivas parciales. De hecho, el siguiente teorema muestra que estas máquinas pueden calcular todas las funciones recursivas parciales.

TEOREMA 4.2
Toda función recursiva parcial es computable por una máquina de Turing.

DEMOSTRACIÓN
Primero mostramos, en la figura 4.11, que las funciones iniciales son computables por una máquinas de Turing. Esta figura describe máquinas explícitas que calculan las funciones cero, sucesor y de proyección.

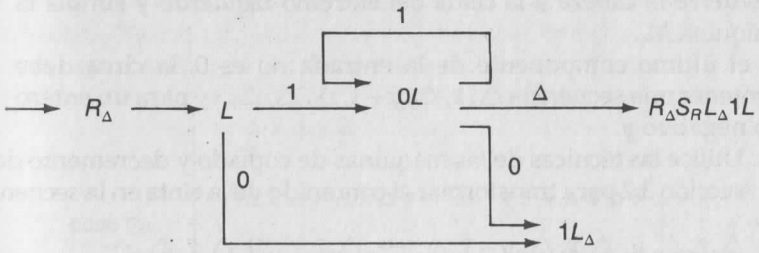
Luego mostramos que también son computables por una máquinas de Turing las funciones parciales construidas por combinación, composición, recursividad primitiva y minimalización a partir de funciones parciales computables por una máquina de Turing. Comencemos por la combinación. Si tenemos máquinas de Turing M_1 y M_2 que calculan las funciones parciales g_1 y g_2, respectivamente, podemos construir una máquina de Turing M que calcule $g_1 \times g_2$ de la manera siguiente: diseñe M como una máquina de tres cintas que comience por duplicar su entrada en cada una de las demás cintas. Luego, M simula M_1 usando la cinta 2 y M_2 con la cinta 3. Si cada una de estas simulaciones lleva a una terminación válida, M borra su primera cinta, copia el contenido de las otras cintas a la cinta 1 para que ésta contenga las salidas de g_1 y g_2 separadas por espacios en blanco, y se detiene.

Es muy fácil construir una máquina de Turing que calcule la composición de dos funciones parciales, cada una computable por una máquina de Turing. Si la máquina M_1 calcula g (devolviendo la cabeza a la celda del extremo izquierdo antes de detenerse) y M_2 calcula f, entonces $\to M_1M_2$ calcula $f \circ g$.

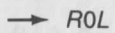
Consideremos ahora la recursividad primitiva. Para esto, suponga que f está definida por

$$f(\bar{x}, 0) = g(\bar{x})$$
$$f(\bar{x}, y + 1) = h(\bar{x}, y, f(\bar{x}, y))$$

Máquina para calcular σ :



Máquina para calcular ζ:



Máquina para calcular π_j^i :

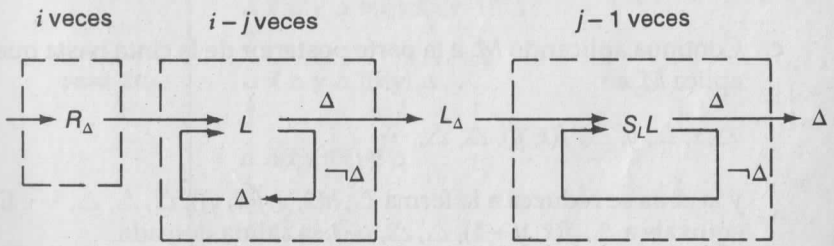


Figura 4.11 Cálculo de las funciones iniciales

donde g es una función parcial calculada por la máquina de Turing M_1 y h es una función parcial calculada por M_2 (podemos suponer que M_1 y M_2 devuelven su cabeza a la celda del extremo izquierdo de la cinta antes de detenerse). Entonces, f se puede calcular con una máquina de Turing que opera de la siguiente forma (véase Fig. 4.12):

1. Si el último componente de la entrada es 0, borra este componente, devuelve la cabeza a la celda del extremo izquierdo y simula la máquina M_1.
2. Si el último componente de la entrada no es 0, la cinta debe contener una secuencia $\triangle, \bar{x}, \triangle, y+1, \triangle, \triangle, \triangle, \cdots$ para un entero no negativo y.
   a. Utilice las técnicas de las máquinas de copiado y decremento de la sección 3.2 para transformar el contenido de la cinta en la secuencia

   $$\triangle, \bar{x}, \triangle, y, \triangle, \bar{x}, \triangle, y-1, \triangle, \cdots, \triangle, \bar{x}, \triangle, 0, \triangle, \bar{x}, \triangle, \cdots$$

   Luego ubica la cabeza sobre la celda que sigue a 0 y simula M_1.
   b. La cinta contendrá ahora la secuencia

   $$\triangle, \bar{x}, \triangle, y, \triangle, \bar{x}, \triangle, y-1, \triangle, \cdots, \triangle, \bar{x}\triangle, 0, \triangle, g(\bar{x}), \triangle, \cdots$$

   que equivale a

   $$\triangle, \bar{x}, \triangle, y, \triangle, \bar{x}, \triangle, y-1, \triangle, \cdots, \triangle, \bar{x}, \triangle, 0, \triangle, f(\bar{x}, 0), \triangle, \cdots$$

   Ubica la cabeza sobre la celda antes de la última $\bar{x}$ y simula M_2. Esto producirá una cinta que contiene

   $$\triangle, \bar{x}, \triangle, y, \triangle, \bar{x}, \triangle, y-1, \triangle, \cdots, \triangle, \bar{x}, \triangle, 1, \triangle, h(\bar{x}, 0, f(\bar{x}, 0)), \triangle, \cdots$$

   que equivale a

   $$\triangle, \bar{x}, \triangle, y, \triangle, \bar{x}, \triangle, y-1, \triangle, \cdots, \triangle, \bar{x}, \triangle, 1, \triangle, f(\bar{x}, 1), \triangle, \cdots$$

   c. Continúa aplicando M_2 a la parte posterior de la cinta hasta que se aplica M_2 a

   $$\triangle, \bar{x}, \triangle, y, \triangle, f(\bar{x}, y), \triangle, \triangle, \cdots$$

   y la cinta se reduzca a la forma $\triangle, h(\bar{x}, y, f(\bar{x}, y)), \triangle, \triangle, \triangle, \cdots$. Esto equivale a $\triangle, f(\bar{x}, y+1), \triangle, \triangle, \cdots$, la salida deseada.

Finalmente, debemos establecer que son computables por máquinas de Turing las funciones parciales construidas con la minimalización a partir de funciones parciales computables por máquinas de Turing. Esto es bastante sencillo; para calcular $\mu y[g(\bar{x}, y) = 0]$, donde g es calculada por la máquina de Turing M, podríamos utilizar una máquina de tres cintas que operase de la siguiente manera:

1. Escribe 0 en la cinta 2.

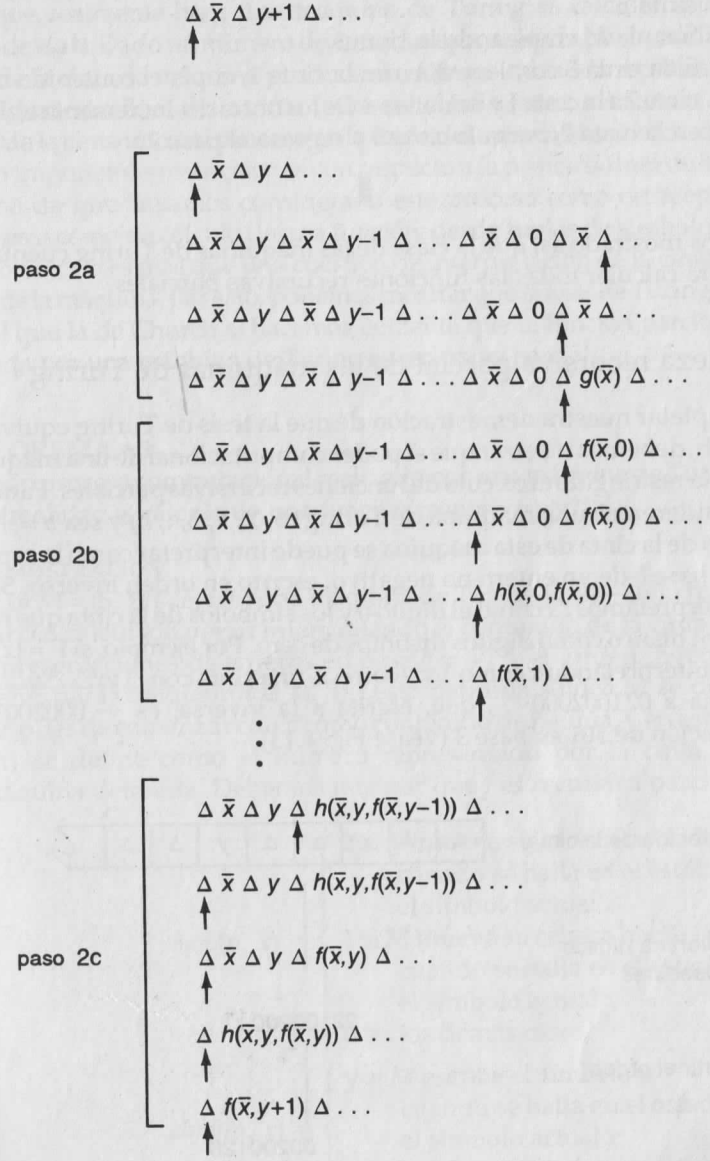


Figura 4.12 Representación del contenido de la cinta de una máquina de Turing durante los pasos del cálculo de una función recursiva primitiva

2. Copia $\bar{x}$ de la cinta 1 a la cinta 3, seguido por el contenido de la cinta 2.
3. Simula M empleando la cinta 3.
4. Si la cinta 3 contiene 0, borra la cinta 1, copia el contenido de la cinta 2 a la cinta 1 y se detiene. De lo contrario, incrementa el valor en la cinta 2, borra la cinta 3 y regresa al paso 2.

∎

Hemos mostrado así que la clase de las máquinas de Turing cuenta con el poder para calcular todas las funciones recursivas parciales.

Naturaleza recursiva parcial de las máquinas de Turing

Para completar nuestra demostración de que la tesis de Turing equivale a la de Church, debemos mostrar que el poder computacional de una máquina de Turing está restringido al cálculo de funciones recursivas parciales. Para lograr esto, consideremos una máquina de Turing $(S, \Sigma, \Gamma, \delta, \iota, h)$ y sea $b = |\Gamma|$. El contenido de la cinta de esta máquina se puede interpretar como la representación en base b de un entero no negativo, escrito en orden inverso. Simplemente interpretamos $\triangle$ como el dígito 0 y los símbolos de la cinta que no sean espacios en blanco como dígitos distintos de cero. Por ejemplo, si $\Gamma = \{x, y, \triangle\}$, entonces, interpretando x como 1 y y como 2, una cinta con $\triangle yx\triangle\triangle y\triangle\triangle\triangle\cdots$ equivaldría a $021002000\cdots_{\triangle}$ que, escrito a la inversa, es $\cdots$ 000200120, la representación de 501 en base 3 (véase Fig. 4.13).

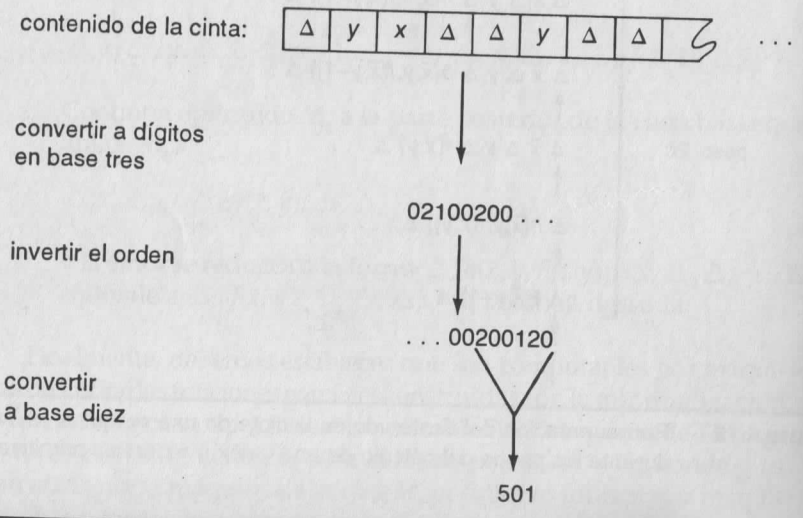


Figura 4.13 Interpretación del contenido de la cinta de una máquina de Turing como valor numérico

Si interpretamos de esta manera las cintas de las máquinas, vemos que lo único que realmente hace una máquina de Turing es calcular una función parcial de $\mathbb{N}$ a $\mathbb{N}$. Dado un número de entrada representado por la configuración inicial de la cinta, la máquina produce un número de salida representado por la configuración final de la cinta (para ser consistentes, insistimos en que la cabeza de la cinta inicie el proceso en la celda del extremo izquierdo de la cinta, pero no imponemos restricciones con respecto a la posición final de la cabeza). El hecho de que hayamos considerado este proceso como de aceptación de lenguajes o como de cálculo de una función, desde tuplas de símbolos a tuplas de símbolos, no es más que una cuestión de interpretación de la cinta y no una acción de la máquina. Por esto, podemos mostrar que la tesis de Turing no es más general que la de Church si hacemos evidente que la función parcial de $\mathbb{N}$ a $\mathbb{N}$ calculada por una máquina de Turing es recursiva parcial.

TEOREMA 4.3
Todo proceso computacional realizado por una máquina de Turing es en realidad el cálculo de una función recursiva parcial.

DEMOSTRACIÓN
Sean $M = (S, \Sigma, \Gamma, \delta, \iota, h)$ una máquina de Turing y $f: \mathbb{N} \to \mathbb{N}$ la función parcial calculada por M interpretando el contenido de la cinta como representaciones de enteros en base $b = |\Gamma|$ en orden inverso. Es decir, para cualquier $n \in \mathbb{N}$, $f(n)$ está definida sólo si M se detiene después de comenzar con n como contenido de la cinta, y en este caso $f(n)$ se define como el número representado por la cinta de la máquina detenida. Debemos mostrar que f es recursiva parcial.

$$mov(p, x) = \begin{cases} 2 \text{ si } M \text{ mueve su cabeza hacia la redecha cuando se halla en el estado } p \text{ con el símbolo actual } x \\ 1 \text{ si } M \text{ mueve su cabeza hacia la izquierda cuando se halla en el estado } p \text{ con el símbolo actual } x \\ 0 \text{ en los demás casos} \end{cases}$$

$$símb(p, x) = \begin{cases} y \text{ si } M \text{ escribe el símbolo } y \text{ cuando se halla en el estado } p \text{ con el símbolo actual } x \\ x \text{ en los demás casos} \end{cases}$$

y

$$estado(p, x) = \begin{cases} q \text{ si } M \text{ cambia al estado } q \text{ cuando se halla en el estado } p \text{ el símbolo actual } x \\ k \text{ si } p = 0 \text{ o } (p, x) \text{ no es un par estado-símbolo válido} \end{cases}$$

Para esto, asociamos 0 al estado de parada de M, 1 al estado inicial y los demás enteros no negativos menores que k a los demás estados, donde k es el número de estados en M. Así, los estados de M están representados por los enteros $0, 1, \cdots, k-1$.

Al llegar a este punto se han asignado valores numéricos tanto a los estados de M como a los símbolos del alfabeto de M.

Ahora defina las siguientes funciones, las cuales resumen la información que contendría el diagrama de transiciones de M.

Es posible representar cada una de estas funciones con una tabla finita, como se mencionó en la sección 4.2, por lo que cada una de ellas es recursiva primitiva. En resumen, $mov(p, x)$ indica si la acción realizada por M al enfrentarse al par estado-símbolo (p, x) es un movimiento a la derecha, un movimiento a la izquierda o una operación de escritura; $símb(p, x)$ indica el símbolo que dejaría M en la celda actual si el par estado-símbolo actual fuera (p, x); y $estado(p, x)$ indica el estado a donde pasaría M si el par estado-símbolo actual fuera (p, x). Observe que $estado(p, x)$ devuelve el número de estado k, inválido (los estados de M se numeraron $0, 1, \cdots, k-1$), si el estado de entrada es el de parada o el par estado-símbolo de entrada no es válido. Esta característica nos garantiza que la función recursiva parcial no estará definida para aquellas entradas que produzcan una terminación anormal de M.

Después observamos que en cualquier momento de los cálculos la configuración de M se puede representar con lo que llamamos una tripleta de configuración, lo cual no es más que una tupla de tres componentes (w, p, n) donde w representa el contenido de la cinta (como un número entero escrito en base $b = |\Gamma|$), p es el entero que representa el estado actual de M, y n es un entero no negativo que representa la posición de la celda actual, donde la celda del extremo izquierdo de la cinta es 1, la primera celda a la derecha es 2, la siguiente celda es 3, etcétera.

Otros aspectos de los cálculos se pueden determinar a partir de la información en una tripleta de configuración. Por ejemplo, a partir de la tripleta (w, p, n) podemos determinar el símbolo actual como $coc(w, b^{n-1}) + mult(b, coc(w, b^n))$ (véase Fig. 4.14). Además, este cálculo sólo implica funciones que, como ya hemos mostrado, pertenecen a la clase de las funciones recursivas primitivas. Por lo tanto, la función $símbact: \mathbb{N}^3 \to \mathbb{N}$ definida por

$$símbact(w, p, n) = coc(w, b^{n-1}) + mult(b, coc(w, b^n))$$

calcula el símbolo actual a partir de la información en un tripleta de configuración, por lo que también es recursiva parcial.

Las funciones *cabezasig*, *estadosig* y *cintasig* son modelos de otros cálculos basados en la información de una tripleta de configuración. En particular, *cabezasig* está definida por

$$cabezasig(w, p, n) = n + eq(mov(p, simbact(w, p, n)), 1) + eq(mov(p, simbact(w, p, n)), 2))$$

Esta función produce el entero que representa la siguiente posición de la cabeza de la cinta. Si la salida de *cabezasig* es 0, indica una terminación anormal ocasionada cuando la cabeza sobrepasa el final de la cinta (recuerde que se asignó el número 1 a la celda del extremo izquierdo de la cinta).

La función *estadosig* está definida por

$$estadosig(w, p, n) = estado(p, simbact(w, p, n)) + mult(k, \neg cabezasig(w, p, n))$$

Normalmente, el segundo término en esta suma es 0, por lo que *estadosig* devuelve un número de estado válido que representa el estado al cual pasaría M si se encontrara en la configuración (w, p, n). Sin embargo, si este desplazamiento está acompañado por una terminación anormal, entonces el segundo término produciría el valor k, por lo que *estadosig* devolvería un número de estado inválido.

La función *cintasig* está definida por

$$cintasig(w, p, n) = (w + mult(b^n, símbact(w, p, n))) + mult(b^n, símb(p, símbact(w, p, n)))$$

Esta función produce el entero que representa el contenido de la cinta de M después de ejecutar una transición de la configuración (w, p, n).

Las funciones *cabezasig*, *estadosig* y *cintasig* se construyen por medio de la combinación y la composición a partir de funciones recursivas parciales, por lo que también son recursivas parciales. Además, al combinar estas funciones obtenemos otra función recursiva parcial $paso: \mathbb{N}^3 \to \mathbb{N}^3$ que simula un paso de un cálculo de M. Esta función relaciona la tripleta de configuración de su entrada con la tripleta de salida que representa a la siguiente configuración de M. De manera más precisa,

$$paso = cintasig \times estadosig \times cabezasig$$

Ahora, empleando la recursividad primitiva, definimos una función $ejec: \mathbb{N}^4 \to \mathbb{N}^3$ tal que $ejec(w, p, n, t)$ produzca la tripleta de configuración que representa a la situación de M después de ejecutar t transiciones desde la configuración de arranque (w, p, n). En particular, tras 0 transiciones M estaría todavía en la configuración (w, p, n), por lo que tenemos

$$ejec(w, p, n, 0) = (w, p, n)$$

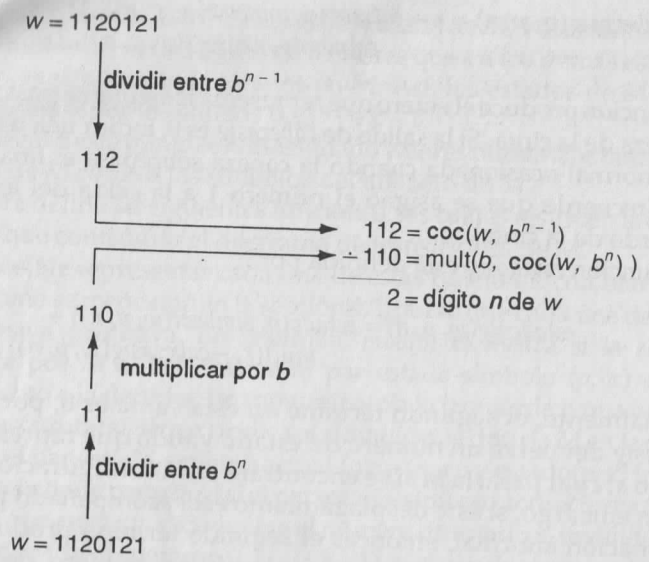


Figura 4.14 Localización del dígito n de w, donde $w = 1120121$, $n = 5$ y $b = 3$

y después de $t + 1$ transiciones, M habría ejecutado una transición a partir de la configuración $ejec(w, p, n, t)$, por lo que

$$ejec(w, p, n, t + 1) = paso(ejec(w, p, n, t))$$

Finalmente, la salida de la función calculada por M (dada una entrada w) es el valor de la cinta al llegar al estado de parada (estado 0). El número de pasos requeridos para llegar a este estado es $\mu t[\pi_2^3(ejec(w, 1, 1, t) = 0]$, o sea, la menor t para la cual el segundo componente de $ejec(w, 1, 1, t)$ es 0 (observe que la función *ejec* es total y, por lo tanto, no hay problemas para calcular $\mu t[\pi_2^3(ejec(w, 1, 1, t) = 0]$ cuando existe dicha t). Con base en esto, definimos la función recursiva parcial *momentoparada*: $\mathbb{N} \to \mathbb{N}$ como

$$momentoparada(w) = \mu t[\pi_2^3(ejec(w, 1, 1, t) = 0]$$

Así, si f:$\mathbb{N} \to \mathbb{N}$ es la función parcial calculada por M, entonces

$$f(w) = \pi_1^3(ejec(w, 1, 1, momentoparada(w)))$$

Por consiguiente, f es una función recursiva parcial.

∎

En resumen, hemos mostrado que las funciones recursivas parciales son computables por máquinas de Turing y que cualquier máquina de Turing no hace más que calcular una función recursiva parcial. Esto significa que las tesis de Turing y de Church son equivalentes aunque se presenten en contextos distintos. Con frecuencia, el concepto común en estas tesis se conoce como tesis de Church-Turing en vez de tesis de Turing o tesis de Church. Sin embargo, la equivalencia de estas hipótesis es más importante que la mezcla de la terminología, pues implica que hemos llegado al mismo límite aparente para el poder de los procesos computacionales siguiendo dos enfoques distintos, un enfoque operacional y uno funcional, lo cual refuerza nuestra confianza en estas conjeturas.

Ejercicios

1. ¿Cuál es el valor de $f(4)$ si f está definida por $f(x) = \mu y[monus(x, pred(y)) = 0]$? ¿Es f total o estrictamente parcial?

2. Si $g(x, y)$ es recursiva primitiva y m es un entero positivo, muestre que f: $\mathbb{N} \to \mathbb{N}$ definida por $f(x) = \mu y[g(x, y) = m]$ es recursiva parcial (por lo tanto, se puede emplear la minimalización para encontrar valores distintos de cero sin salir de los límites de las funciones recursivas parciales).

3. Diseñe una máquina de Turing que calcule la función f:$\Sigma^* \to \Sigma^*$ definida por $f(w) = w^R$, donde $\Sigma = \{x, y\}$ y w^R es la cadena w escrita en orden inverso.

4. ¿Para cuáles cadenas está definida la función parcial f:$\{x, y\}^*$, calculada por la máquina de Turing compuesta $\to R \xrightarrow{x} yR_{\triangle}L_xLL$? Describa la función.

4.4 PODER DE LOS LENGUAJES DE PROGRAMACIÓN

Como aplicación de la teoría presentada en las secciones anteriores, regresemos a nuestra pregunta con respecto al poder expresivo de los lenguajes de programación. Lo que nos concierne es la cuestión de qué aspectos deben incluirse para garantizar que, una vez diseñado e implantado un lenguaje de programación, no descubramos que existen problemas cuyas soluciones no pueden especificarse con el lenguaje, y que sí podrían haberlo sido si hubiéramos implantado una versión ampliada del lenguaje.

Nuestra estrategia es desarrollar un sencillo lenguaje de programación esencial con el cual pueda expresarse un programa para calcular cualquier función recursiva parcial. Esto asegurará (suponiendo que la tesis de Church-Turing es verdadera) que mientras un lenguaje de programación cuente con

las características de nuestro sencillo lenguaje, permitirá expresar una solución para cualquier problema que pueda resolverse de manera algorítmica.

Un lenguaje de programación esencial

Puesto que nuestro lenguaje de programación esencial se usará para calcular funciones recursivas parciales, el único tipo de datos que se requiere es el entero no negativo (como ya hemos señalado, en un computador digital moderno todo elemento de datos se representa como un entero no negativo, aunque el lenguaje de alto nivel pueda disfrazar esta realidad). Así mismo, nuestro sencillo lenguaje de programación no requiere enunciados de declaración de tipo, sino que los identificadores, que consisten en letras y dígitos (comenzando por una letra) se declaran automáticamente como de tipo entero no negativo con sólo aparecer por primera vez en un programa (por conveniencia, en ocasiones emplearemos identificadores con subíndices a sabiendas de que podemos eliminar esta informalidad a costa de perder un poco de claridad).

Nuestro lenguaje contiene los dos enunciados de asignación siguientes:

```
incr nombre;
```

y

```
decr nombre;
```

El primero incrementa en uno el valor asignado al identificador nombre, mientras que el segundo lo decrementa en uno (a menos que el valor por decrementar sea cero, en cuyo caso permanece con dicho valor).

El único otro enunciado de nuestro lenguaje es el par de enunciados de control

```
while nombre ≠ 0 do;
        :
end;
```

el cual indica que es necesario repetir los enunciados que se encuentran entre los enunciados while y end mientras el valor asignado al identificador nombre no sea cero.

Éste es, entonces, nuestro lenguaje de programación esencial. Es muy sencillo; tanto que nuestra primera meta es incorporar algunos enunciados más poderosos que pueden simularse con secuencias de enunciados esenciales. Entonces podremos utilizar estos enunciados adicionales para hacer más sencillos los programas esenciales, de la misma forma que las macroinstrucciones se utilizan para simplificar los programas en lenguaje ensamblador. Específicamente, adoptaremos la sintaxis

```
clear nombre;
```



```
clear aux;
clear nombre1;
while nombre2 ≠ 0 do;
   incr aux;
   decr nombre2;
end;
while aux ≠ 0 do;
   incr nombre1;
   incr nombre2;
   decr aux;
end;
```

Figura 4.15 Segmento de programa representado por ***nombre1 ← nombre2***

como versión abreviada de la secuencia

```
while nombre ≠ 0 do;
      decr nombre;
end;
```

cuyo efecto es asignar el valor cero al identificador nombre. Además utilizaremos

```
nombre1 ← nombre2;
```

para representar el segmento de programa de la figura 4.15, que asigna el valor de nombre2 al identificador nombre1 (primero se asigna el valor a la variable auxiliar aux y luego se reasigna tanto a nombre1 como a nombre2; El esfuerzo adicional que implica aux evita el efecto secundario de destruir la asignación original de nombre2).

Recursiva parcial implica programable con lo esencial

Ahora nuestra tarea es mostrar que para cualquier función recursiva parcial existe un algoritmo para calcular la función, el cual puede expresarse con nuestro sencillo lenguaje de programación esencial Para esto, estableceremos la convención de que para calcular una función de $\mathbb{N}^m$ a $\mathbb{N}^n$ escribiremos un programa con los identificadores $x_1, x_2, \cdots, x_m$ para contener los valores de entrada, y con $z_1, z_2, \cdots, z_n$ para almacenar los valores de salida.

Con nuestro lenguaje es fácil expresar los programas para calcular las funciones iniciales. La función ζ es manejada por

σ por

```
clear z1;
z1 ← x1;
incr z1;
```

y π_j^m por

```
z1 ← xj;
```

Ahora centramos nuestra atención en las funciones recursivas parciales. Si F y G son programas que calculan las funciones parciales $f: \mathbb{N}^k \to \mathbb{N}^m$ y $g: \mathbb{N}^k \to \mathbb{N}^n$, respectivamente, entonces la función $f \times g$ se puede calcular concatenando el programa G al final del programa F, modificando F y G para que asignen sus salidas a los identificadores apropiados (F debe asignar su salida a $z_1, \cdots, z_m$ a la vez que G asigna su salida a $z_{m+1}, \cdots, z_{m+n}$) y ajustando F para que no destruya los valores de entrada antes de que G pueda utilizarlos.

(Aquí, como en los casos que se presentan más adelante, suponemos que los programas en cuestión no tienen nombres de variables auxiliares comunes que pudieran ocasionar efectos secundarios desastrosos. Si acaso los tuvieran, cambiaríamos los nombres; por ejemplo, todos los identificadores auxiliares del programa F estarían precedidos por la letra F, mientras que los de G estarían precedidos por G.)

Si F y G calculan las funciones parciales $f: \mathbb{N}^k \to \mathbb{N}^l$ y $g: \mathbb{N}^l \to \mathbb{N}^m$, respectivamente, entonces $g \circ f$ se puede calcular concatenando G al final de F y ajustando los identificadores de salida de F para que vayan de acuerdo con los identificadores de entrada de G.

```
G
aux ← x_{k+1};
clear x_{k+1};
while aux ≠ 0 do;
    x_{k+2} ← z1;
    x_{k+3} ← z2;
    ⋮
    x_{k+m+1} ← zm;
    H
    incr x_{k+1};
    decr aux;
end;
```

Figura 4.16 Programa para calcular una función definida por recursividad primitiva

Ahora suponga que el programa G calcula la función parcial $g: \mathbb{N}^k \to \mathbb{N}^m$, H calcula $h: \mathbb{N}^{k+m+1} \to \mathbb{N}^m$, y, usando recursividad primitiva, $f: \mathbb{N}^{k+1} \to \mathbb{N}^m$ se define como

$$f(\bar{x}, 0) = g(\bar{x})$$
$$f(\bar{x}, y + 1) = h(\bar{x}, y, f(\bar{x}, y))$$

Entonces se puede calcular f con el programa de la figura 4.16, donde suponemos (sin perder generalidad) que G y H no tienen efectos secundarios indeseables.

Mostramos ahora que si G es un programa en nuestro lenguaje esencial que calcula la función recursiva parcial $g: \mathbb{N}^{n+1} \to \mathbb{N}$, entonces podemos producir un programa que calcule $\mu y[g(\bar{x}, y) = 0]$. El programa de la figura 4.17 lo lleva a cabo calculando $g(\bar{x}, 0), g(\bar{x}, 1), \cdots$ hasta producir una salida cero (observe que G se puede diseñar para que no altere las asignaciones originales de sus variables de entrada).

En este momento hemos alcanzado nuestro objetivo de mostrar que cualquier función recursiva parcial puede calcularse por medio de un programa escrito en nuestro lenguaje de programación esencial. Así, de acuerdo con la tesis de Church-Turing, sabemos que cualquier lenguaje que proporcione el tipo entero no negativo y la capacidad para incrementar un valor, decrementarlo y ejecutar un ciclo while, tendrá el poder expresivo suficiente para plantear la solución de cualquier problema que tenga una solución algorítmica. Todas las características adicionales de un lenguaje representan una conveniencia, pero no poder adicional.

Programable con lo esencial implica recursiva parcial

Después de haber descubierto el sorprendente poder de nuestro sencillo lenguaje, se podría pensar que ofrece un medio para calcular más que las funciones recursivas parciales. Por supuesto, si esta conjetura fuera verdadera, contradiría la tesis de Church-Turing ya que obtendríamos un método para calcular una clase de funciones mayor que la de las funciones recursivas parciales (computables por máquinas de Turing). Por esto, no es ninguna

```
clear x_{n+1};
G
while z1 ≠ 0 do;
    incr x_{n+1};
    G
end;
z1 ← x_{n+1}
```

Figura 4.17 Programa para calcular $\mu y[g(\bar{x}, y) = 0]$

sorpresa saber que cualquier cálculo expresado en nuestro sencillo lenguaje puede mode-larse por medio de una función recursiva parcial.

Para demostrar la realidad de este límite del poder de nuestro lenguaje, primero observamos que cualquier programa en nuestro sencillo lenguaje debe comprender cuando menos un identificador, ya que debe contener cuando menos una de las tres formas de enunciados (incr, decr, while), y cada uno de estos enunciados implica una variable. Si un programa contiene k variables y presentamos colectivamente estas variables como una k-tupla, entonces el cálculo expresado por el programa en realidad representa una función de $\mathbb{N}^k$ a $\mathbb{N}^k$, donde la k-tupla de entrada comprende los valores asignados a las k variables al iniciar el programa, y la salida es la k-tupla de los valores asignados a estas k variables al llegar al final del programa (si el programa nunca termina, entonces la función no está definida para esa tupla de entrada). Procedamos ahora a mostrar que esta función debe ser recursiva parcial. Nuestra estrategia será inducir en el número de enunciados en el programa.

Si únicamente existe un enunciado en el programa, hay tres posibilidades: puede ser un enunciado incr, decr o while. Los primeros dos casos calculan las funciones recursivas primitivas σ y *pred*, respectivamente, mientras que el tercer caso

```
while nombre ≠ 0 do;
end;
```

calcula la función

$$f(nombre) = \begin{cases} 0 \text{ si } nombre = 0 \\ \text{no definida en los demás casos} \end{cases}$$

que es idéntica a la función recursiva parcial

$$f(nombre) = \mu\, y[más(nombre, y) = 0]$$

Por lo tanto, todos los programas que contienen un solo enunciado de nuestro sencillo lenguaje calculan funciones recursivas parciales.

Ahora consideremos los programas con n enunciados, donde $n > 1$, suponiendo que todo programa con menos de n enunciados debe calcular una función recursiva parcial. Si el programa en cuestión no tiene la estructura de un enunciado while de gran tamaño, entonces es la concatenación de dos programas más pequeños. Por nuestra hipótesis de inducción, cada uno de estos programas más pequeños calcula una función recursiva parcial, pero el programa global calcula la composición de estas funciones. Por consiguiente, el programa completo calcula una función recursiva parcial.

Para concluir nuestro argumento, suponemos que el programa en cuestión consiste en una gran estructura while, que representamos como

```
while X ≠ 0 do;
  B
end;
```

Puesto que B, el cuerpo de este ciclo, contiene menos de n enunciados, nuestra hipótesis de inducción nos indica que calcula una función recursiva parcial $h: \mathbb{N}^k \to \mathbb{N}^k$. Además, podemos suponer que la variable X identificada en el enunciado while es uno de los componentes, digamos el j, de la k-tupla que manipula B (si B no manipulara esta variable durante el proceso cíclico, una vez iniciado nunca terminaría); por lo tanto, toda la estructura while calcularía la función recursiva parcial que coincide con la función identidad cuando X es cero, y no está definida para todas las demás entradas).

Aplicando la recursividad primitiva, definimos la función $f: \mathbb{N}^{k+1} \to \mathbb{N}^k$ como

$$f(\bar{x}, 0) = ident(\bar{x})$$
$$f(\bar{x}, y + 1) = h(f(\bar{x}, y))$$

donde *ident* es la función identidad (la función identidad se puede construir como la combinación de proyecciones, por lo que es recursiva primitiva). Observe que el valor de $f(\bar{x}, y)$ es la k-tupla producida por la asignación de valores iniciales a $\bar{x}$ durante el cuerpo del ciclo, B, y la posterior ejecución el ciclo y veces. El número de veces que se ejecutará realmente el cuerpo de la estructura while es $\mu y[\pi_j^k \circ f(\bar{x}, y) = 0]$. Entonces, la función $g: \mathbb{N}^k \to \mathbb{N}^k$ calculada por toda la estructura while está definida por

$$g(\bar{x}) = f(\bar{x}, \mu y[\pi_j^k \circ f(\bar{x}, y) = 0])$$

Por consiguiente, la función calculada por toda la estructura while es recursiva primitiva.

Para concluir, debemos señalar que el estudio de la computabilidad por medio de los lenguajes de programación es otra área de investigación cuyos resultados apoyan la tesis de Church-Turing. De hecho, no se ha diseñado ningún lenguaje de programación que tenga mayor poder expresivo que nuestro sencillo lenguaje (aunque los lenguajes más elaborados que se emplean en la actualidad son obviamente superiores en cuanto a legibilidad).

Ejercicios

1. Muestre que el lenguaje de programación que consiste en enunciados de la forma

```
clear nombre;        incr nombre;        loop nombre veces;
                                              .
                                              .
                                              end;
```

tiene el poder expresivo equivalente a las funciones recursivas primitivas.

2. Escriba un programa en el sencillo lenguaje while de esta sección que calcule la función $f: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ definida por

$$f(x, y) = \begin{cases} 1 \text{ si } x > y \\ 0 \text{ en los demás casos} \end{cases}$$

3. Muestre cómo puede simularse la estructura de programación

$$\text{if } x = 0 \text{ then } S_1 \text{ else } S_2$$

donde S_1 y S_2 representan segmentos de programa, con el sencillo lenguaje while de esta sección.

4. Muestre cómo puede simularse la estructura de programación

$$\text{repeat } S \text{ until } x = 0$$

donde S representa un segmento de programa, con el sencillo lenguaje while de esta sección.

4.5 COMENTARIOS FINALES

En este capítulo hemos ampliado nuestro estudio, partiendo del procesamiento de lenguajes hasta incluir el cálculo de funciones. Nuestro objetivo principal era apoyar la tesis de Turing mostrando que está de acuerdo con las conjeturas que han surgido en otras áreas de investigación.

Comenzamos por identificar las limitaciones generales de los procesos computacionales. Para liberar a nuestro estudio de un sistema computacional específico, decidimos considerar un acercamiento funcional a la computabilidad, en vez de uno operativo. Es decir, nos centramos en lo que logran los procesos computacionales en vez de cómo se realizan. Esto nos llevó a la clase de las funciones recursivas parciales que, de acuerdo con la tesis de Church, es la clase de las funciones parciales que cualquier sistema computacional puede calcular. Luego mostramos que este límite aparente para los procesos computacionales coincide con la tesis de Turing.

Después consideramos el poder expresivo de los lenguajes de programación tradicionales. Lo que buscábamos era determinar si un lenguaje dado

impone restricciones innecesarias con respecto a la clase de problemas que pueden resolver los programas en ese lenguaje (¿puede cualquier problema con solución algorítmica resolverse por medio de un programa escrito en ese lenguaje?). Nuestra estrategia fue definir un sencillo lenguaje de programación esencial y mostrar que un programa escrito en ese lenguaje podía calcular cualquier función recursiva parcial. Por esto, si aceptamos la tesis de Church-Turing, podemos llegar a la conclusión de que cualquier lenguaje de programación que contenga las características de nuestro lenguaje esencial tendrá la generalidad suficiente para permitir que cualquier problema con solución algorítmica se resuelva utilizando un programa escrito en dicho lenguaje.

Por último, consideremos de nuevo todo nuestro estudio. Atacamos el problema de la computabilidad desde varias direcciones: máquinas computacionales, gramáticas generativas, teoría de funciones recursivas y lenguajes de programación. En cada caso descubrimos un límite aparente para las capacidades computacionales de cada enfoque, y mostramos que estos límites coinciden con los demás. Los lenguajes estructurados por frases son iguales que los lenguajes aceptados por máquinas de Turing, las funciones computables por máquinas de Turing son iguales que las funciones recursivas parciales, y las funciones recursivas parciales son iguales que las funciones computables con el lenguaje de programación esencial. Por lo tanto, parece que hemos identificado los confines de los procesos computacionales y, específicamente, de los computadores. Es decir, hemos encontrado un firme apoyo para la tesis de Church-Turing: si una máquina de Turing no puede resolver un problema, entonces ningún computador puede hacerlo pues simplemente no existe un algoritmo para obtener su solución. En otras palabras, las limitaciones que hemos detectado corresponden a los procesos computacionales, no a la tecnología.

Con base en estas observaciones, es común considerar un problema como **problema soluble** (o **problema soluble con una máquina de Turing**) si y sólo si puede resolverse con los cálculos de una máquina de Turing. Así, el problema de aceptación del lenguaje L_0, definido en la sección 3.5, y el problema de la parada son ejemplos de problemas insolubles. En el apéndice C se analizan otros problemas insolubles.

Problemas de repaso del capítulo

1. Encuentre el valor de
   a. $(\sigma \circ \zeta) \times \zeta)(\,)$ b. $\pi_2^3 \times \pi_3^3 \times \pi_2^3(5, 6, 7)$
   c. $(\sigma \times \sigma) \circ \pi_2^2(4, 7)$ d. $\zeta \circ \pi_0^3(4, 5, 6)$

2. Encuentre el valor de $f(5, 4)$ si la función f está definida por

$$f(x, 0) = \sigma(x)$$
$$f(x, y + 1) = (\pi_1^1 \circ f)(x, y)$$

3. Muestre que la función f definida por

$$f(x, y, z) = \begin{cases} x \text{ si } z \text{ es par} \\ y \text{ si } z \text{ es impar} \end{cases}$$

es recursiva primitiva.

4. Muestre que la función *mástripleta*:$\mathbb{N}^3 \to \mathbb{N}$ definida por *mástripleta*$(x, y, z) = x + y + z$ es recursiva primitiva.

5. Muestre que es recursiva primitiva la función que asigna a cada tripleta de la forma (x, y, z) el par ordenado que resulta de intercambiar x y y un total de z veces.

6. Encuentre $f(0), f(1), f(2)$ y $f(3)$ si f está definida por $f(x) = \mu y[eq(x, y) = 0]$. Encuentre $g(0), g(1), g(2)$ y $g(3)$ si g está definida por $g(x) = \mu y[\neg eq(x, y) = 0]$.

7. Calcule $A(2, 2)$, donde A es la función de Ackermann.

8. Resuma la importancia de la función de Ackermann en la jerarquía de funciones iniciales, recursivas primitivas y recursivas parciales.

9. Proporcione un ejemplo de una función recursiva primitiva que no sea función inicial y un ejemplo de una función recursiva parcial que no sea recursiva primitiva.

10. Muestre que la función $f: \mathbb{N} \to \mathbb{N}$ definida por

$$f(x) = \begin{cases} x + 1 \text{ si } x \text{ es par} \\ x \text{ si } x \text{ es impar} \end{cases}$$

es recursiva primitiva.

11. Muestre que si $f: \mathbb{N} \to \mathbb{N}$ es una función recursiva primitiva uno a uno, entonces la función inversa $g: \mathbb{N} \to \mathbb{N}$ definida por $g(f(x)) = x$ es recursiva parcial.

12. Los computadores digitales son capaces de evaluar ciertas relaciones entre enteros, como "igual a", "menor que", "mayor que", etcétera. Muestre que, no obstante, existen relaciones entre enteros que no pueden resolverse con un computador. (Sugerencia: considere un argumento de cardinalidad.)

13. Muestre que el lenguaje de programación Pascal ofrece una forma para calcular cada una de las funciones recursivas parciales. ¿De qué manera se restringe potencialmente este poder al implantarlo en una máquina real?

14. Utilizando el lenguaje de programación esencial de la sección 4.4, escriba un programa que calcule la función *más* de la sección 4.1.

15. Utilizando el lenguaje de programación esencial de la sección 4.4, escriba un programa que calcule la función factorial.

16. Utilice un argumento de cardinalidad para mostrar que existen funciones de $\mathbb{N}$ a $\mathbb{N}$ que no pueden calcularse con el lenguaje de programación esencial de la sección 4.4.

17. Utilizando el lenguaje de programación esencial de la sección 4.4, escriba un programa que calcule la función $f: \mathbb{N} \to \mathbb{N}$ definida por $f(x) = \mu y[mult(x, y) > más(x, y)]$. ¿Para qué valores no está definida f?

18. Muestre que el poder computacional del lenguaje de programación esencial de la sección 4.4 no se reduce si el enunciado while se sustituye por una estructura if/then y la capacidad para expresar procedimientos recursivos.

19. ¿Para qué cadenas está definida la función parcial $f: \{x, y\}^* \to \{x, y\}^*$ calculada por la máquina de Turing compuesta $\to R_x R_y L_\triangle$? Describa la función.

20. Muestre que si podemos colocar etiquetas a los enunciados y usar el enunciado goto en el lenguaje de programación del ejercicio 1 de la sección 4.4, obtenemos un lenguaje con el mismo poder expresivo que nuestro lenguaje de programación esencial de la sección 4.4.

21. Proporcione un ejemplo de una función que no sea recursiva parcial.

22. Suponga que $g: \mathbb{N} \to \mathbb{N}$ y $h: \mathbb{N}^3 \to \mathbb{N}$ son recursivas primitivas. Muestre que también lo es la función $f: \mathbb{N} \to \mathbb{N}$ definida por

$$f(0, y) = g(y)$$
$$f(x + 1, y) = h(f(x, y), y, x)$$

23. Muestre que si $f: \mathbb{N} \to \mathbb{N}$ es una función y M es una máquina de Turing que al recibir la entrada n calcula el valor $f(n)$ en no más de 2^n pasos, entonces f es recursiva primitiva.

24. Diseñe una máquina de Turing que calcule la función $f: \{x, y\}^* \times \{x, y\}^* \to \{x, y\}^* \times \{x, y\}^*$ tal que $f(w_1, w_2) = (w_2, w_1)$.

25. Suponga que $g:\mathbb{N}^2 \to \mathbb{N}$ es recursiva primitiva y que $f:\mathbb{N}^2 \to \mathbb{N}$ está definida de manera que $f(x, y)$ sea el y más pequeño, menor que z donde $g(x, y) = 0$, o cero si no existe tal y. Muestre que f es recursiva primitiva.

26. Muestre que la función $mcd:\mathbb{N}^2 \to \mathbb{N}$, donde $mcd(x, y)$ es el máximo común divisor de x y y, es recursiva parcial.

27. Muestre que, aplicando un número finito de combinaciones, composiciones, recursividades primitivas y cuando mucho una minimalización, cada función recursiva parcial se puede construir a partir de las funciones iniciales.

28. Defina $f:\mathbb{N} \to \mathbb{N}$ para que $f(0) = 0, f(1) = 1$ y $f(n+2) = f(n) + f(n+1)$. Muestre que f es recursiva primitiva.

29. Defina $f:\mathbb{N} \to \mathbb{N}$ para que $f(x)$ sea la suma de los divisores de x. Por ejemplo, $f(5) = 1 + 5 = 6$ y $f(6) = 1 + 2 + 3 + 6 = 12$. Muestre que f es recursiva primitiva.

30. Proporcione un argumento con respecto a que los lenguajes de programación Pascal, Modula-2, Ada, FORTRAN y COBOL tienen, a fin de cuentas, idéntico poder computacional (observe que esto no quiere decir que sean equivalentes en cuanto a su apoyo a ciertos objetivos de diseño, como son la abstracción de datos, la ocultación de información, el diseño modular y la implantación de objetos).

31. Muestre que si la función $f: \mathbb{N}^2 \to \mathbb{N}$ es recursiva parcial, también lo es la función $g: \mathbb{N} \to \mathbb{N}$, definida por $g(x) = f(x, 5)$.

32. Suponga que $f: \mathbb{N} \to \mathbb{N}$ y $g: \mathbb{N} \to \mathbb{N}$ son funciones recursivas parciales. Muestre que existe una función recursiva parcial $h: \mathbb{N} \to \mathbb{N}$ tal que $h(x)$ este definida para exactamente aquellos valores de x para los cuales $f(x)$ está definida $\circ$ $g(x)$ está definida.

33. Suponga que $f:\mathbb{N} \to \mathbb{N}$ y $g:\mathbb{N} \to \mathbb{N}$ son funciones recursivas parciales. Diga si existe una función recursiva parcial $h:\mathbb{N} \to \mathbb{N}$ tal que

$$h(x) = \begin{cases} f(x) \text{ si } f(x) \text{ está definida} \\ g(x) \text{ si } g(x) \text{ está definida pero } f(x) \text{ no lo está} \\ \text{no definida si ni } f(x) \text{ ni } g(x) \text{ está definida} \end{cases}$$

Problemas de programación

1. Escriba un programa para calcular $g(x) = \mu y[\neg eq(f(x, y), 1) = 0]$ para varias funciones $f: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. ¿Qué sucede durante la ejecución de su programa si la función f es *div* (véase Sec. 4.4) y trate de calcular $g(2)$?

2. Escriba un intérprete para el lenguaje de programación esencial de este capítulo.

3. Escriba un programa para calcular la función de Ackermann. ¿Qué problemas surgen al tratar de ejecutar su programa?

CAPÍTULO 5

Complejidad

5.1 Complejidad de los cálculos
Medición de la complejidad
Complejidad de los cálculos de máquinas de Turing
5.2 Complejidad de los algoritmos
Complejidad temporal de las máquinas de Turing
Rendimiento medio
Análisis informal de algoritmos
5.3 Complejidad de los problemas
El problema de comparación de cadenas
Tasas de crecimiento
Limitaciones de la escala de tasas de crecimiento
5.4 Complejidad temporal de los problemas de reconocimiento de lenguajes
Cálculos de tiempo polinómico
La clase *P*
Lenguajes decidibles en tiempo polinómico
Problemas de decisión
5.5 Complejidad temporal de máquinas no deterministas
La clase *NP*
Reducciones polinómicas
Teorema de Cook
5.6 Comentarios finales

Nuestro estudio de los procesos computacionales nos ha llevado a clasificar los problemas en dos grandes categorías: los problemas con solución y los problemas sin solución. En este capítulo nos centraremos en la clase de los problemas solubles; nuestro objetivo es estudiar las soluciones de estos problemas desde una perspectiva práctica más que teórica. En resumen, veremos que muchos de estos problemas, aunque en teoría sean solubles, requieren tal cantidad de recursos (como tiempo o espacio de almacenamiento) que desde el punto de vista práctico permanecen sin solución.

5.1 COMPLEJIDAD DE LOS CÁLCULOS

Uno de los principales objetivos de este capítulo es establecer una escala para clasificar los problemas de acuerdo con su complejidad. Consideraremos que un problema es complejo si su resolución requiere la ejecución de un algoritmo complejo. A su vez, se considera que un algoritmo es complejo si su aplicación requiere la ejecución de un cálculo complicado (consideramos que un cálculo es el proceso que se lleva a cabo cuando se aplica un algoritmo en una situación determinada. Por ejemplo, el algoritmo de multiplicación tradicional conduce a diferentes cálculos, quizá con complejidades distintas, al aplicarse a diversos valores). Para alcanzar nuestro objetivo de medir la complejidad de los problemas, debemos aprender a medir la complejidad de los cálculos individuales, a partir de lo cual podemos determinar la complejidad de los algoritmos y por último la complejidad de los problemas. Así, comenzamos nuestro estudio de la complejidad formalizando el concepto intuitivo de la complejidad de un solo cálculo.

Medición de la complejidad

Consideremos primero el concepto de complejidad. Aunque es posible que contemos con una idea intuitiva de lo que ésta es, necesitamos una definición precisa para llevar a cabo una investigación científica. De manera intuitiva, un cálculo es complejo si es difícil de realizar. Sin embargo, ¿cómo medimos la dificultad? Un enfoque común, el cual seguiremos, es medir indirectamente la dificultad de un cálculo, midiendo los recursos necesarios para ejecutarlo; esto se basa en que un cálculo difícil requerirá más recursos que uno de menor dificultad. Por lo tanto, definimos que la complejidad de un cálculo es la cantidad de recursos necesarios para efectuarlo.

El tiempo es uno de los recursos que con frecuencia se emplea en este contexto. Consideramos que un cálculo es más complejo que otro si la ejecución del primero requiere más tiempo, y llamamos **complejidad temporal** a la cantidad de tiempo necesaria para efectuar el cálculo. La búsqueda de la reducción de la complejidad temporal es lo que nos lleva a aplicar una estrategia de búsqueda binaria para una lista de gran tamaño, en vez de emplear una búsqueda secuencial. Durante la búsqueda secuencial de una lista de 1000 elementos, se evaluará un promedio de 500 elementos; una búsqueda binaria requiere 10, cuando mucho. Por esto, la complejidad temporal estimada de una búsqueda binaria es menor que la del enfoque en secuencia.

Otro recurso que se emplea con frecuencia para medir la complejidad de un cálculo es la cantidad de espacio de almacenamiento requerido. Esto se basa en el supuesto de que, conforme aumenta la complejidad de un cálculo, más espacio de almacenamiento se necesita para su ejecución. La cantidad de espacio de almacenamiento que requiere un cálculo se conoce como **complejidad espacial**. La figura 5.1, donde los cuadros representan los espacios de almacenamiento

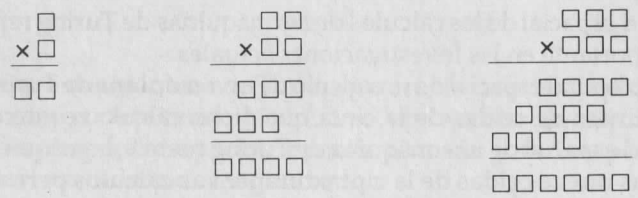


Figura 5.1 Requisitos de espacio para cálculos de multiplicaciones típicas

para dígitos, compara gráficamente la complejidad espacial de los cálculos obtenidos al aplicar el algoritmo de multiplicación tradicional a enteros de uno, dos y tres dígitos.

Es fácil ver que la complejidad espacial o temporal de un cálculo puede variar dependiendo del sistema donde se ejecute el cálculo. El tiempo que se requiere para efectuar un cálculo en un computador moderno es mucho menor que el requerido para realizar el mismo cálculo con las máquinas de la década de los cincuenta. Además, la cantidad de espacio necesario para el almacenamiento de datos depende del sistema de codificación que se emplee. Cualquier valor entero mayor que 2 necesita más espacio cuando se escribe en notación binaria que cuando se emplea la notación decimal.

Para eliminar estas variaciones del análisis, es común estudiar la complejidad en el contexto de un sistema computacional fijo. Para nuestros fines, nos centraremos en la complejidad de los cálculos de las máquinas de Turing. Esto no sólo nos proporciona un ambiente bien definido donde trabajar, sino que además nos lleva a conclusiones que son fáciles de transferir a otros sistemas computacionales, ya que las máquinas de Turing tienen muchas características semejantes a las de los computadores digitales modernos.

Complejidad de los cálculos de máquinas de Turing

Consideremos la ejecución de una transición en una máquina de Turing como un paso en los cálculos de la máquina y definamos la complejidad temporal de una máquina de Turing como el número de pasos que se ejecutan durante los cálculos. Por ejemplo, la complejidad temporal de los cálculos efectuados por la máquina de Turing de la figura 5.2, al iniciar con la configuración de la cinta $\underline{\triangle}xxx\triangle\triangle\triangle\cdots$, sería 9 (se requieren cuatro pasos para mover la cabeza al primer espacio en blanco después de las x, un paso para escribir una x en este lugar y cuatro pasos más para colocar la cabeza en su posición original). O, si la configuración original de la cinta fuera $\underline{\triangle}\triangle\triangle\triangle\cdots$, los cálculos efectuados tendrían una complejidad temporal 3.

La complejidad temporal en el contexto de las máquinas de Turing será uno de los principales ejemplos en este capítulo, aunque debemos reconocer la importancia de otras mediciones de la complejidad. Por ejemplo, la

complejidad espacial de los cálculos de las máquinas de Turing representa un aspecto importante en las investigaciones actuales.

La complejidad espacial de un cálculo de una máquina de Turing se define como el número de celdas de la cinta que dicho cálculo requiere. Así, si la complejidad espacial de una máquina de Turing fuera 9, la máquina utilizaría las primeras nueve celdas de la cinta durante sus cálculos pero no requeriría que estuviera presente el resto de la cinta. La complejidad espacial del cálculo que realiza la máquina de la figura 5.2, al iniciar con una configuración de cinta $\underline{\triangle}xxx\triangle\triangle\triangle\cdots$, es 5 (la cabeza se moverá hasta la quinta celda de la cinta antes de regresar a la primera y detenerse). De modo similar al iniciar con la configuración de cinta $\underline{\triangle}\triangle\triangle\triangle\cdots$, la máquina ejecutará un cálculo con complejidad espacial 2 ya que sólo se utilizarán dos celdas de la cinta.

Observe que las complejidades espacial y temporal son diferentes y, por lo tanto, es posible que difieran para el mismo cálculo. Por ejemplo, una máquina que nunca mueve la cabeza de la cinta pero escribe 100 veces un espacio en blanco en la celda actual antes de detenerse, ejecutaría un cálculo con complejidad temporal 100 pero complejidad espacial 1. De hecho, es fácil ver cómo puede extenderse este ejemplo para producir cálculos en las cuales las complejidades espacial y temporal difieren en cualquier cantidad deseada.

Por otra parte, las complejidades espacial y temporal no son totalmente independientes: en n pasos, una máquina de Turing tiene acceso a un máximo de $n + 1$ celdas de la cinta. Por consiguiente, *si la complejidad temporal de una máquina de Turing es* n, *entonces la complejidad espacial del cálculo no será mayor que* $n + 1$.

Concluimos con la observación de una consecuencia algo peculiar de nuestras definiciones. Una máquina de Turing puede aceptar una cadena a través de un cálculo con complejidad espacial menor que la requerida para contener la cadena. Esto se debe a que la máquina de Turing no tiene que leer una cadena en orden para poder aceptarla. Por ejemplo, la máquina que simplemente escribe un espacio en blanco en la celda actual y pasa al estado de parada aceptará cualquier cadena con un cálculo que sólo tiene acceso a una celda y que, por lo tanto, tiene una complejidad espacial de 1. Sin embargo, si requerimos que además acepte cadenas deteniéndose con la cinta configurada como $\underline{\triangle}Y\triangle\triangle\triangle\cdots$, entonces la máquina deberá borrar la cadena de entrada antes de detenerse. En estos casos el cálculo de aceptación tendrá una complejidad espacial por lo menos equivalente a la longitud de su entrada.

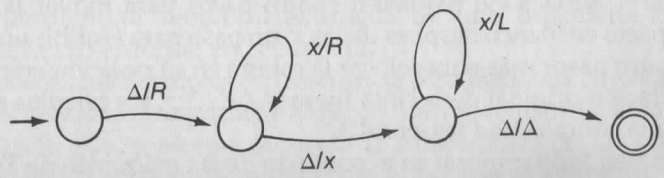


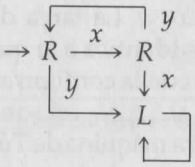
Figura 5.2 Máquina de Turing sencilla

Ejercicios

1. Muestre que una máquina de Turing que acepta cadenas deteniéndose con su cinta configurada como $\underline{\triangle}Y\triangle\triangle\triangle\cdots$ debe ejecutar un cálculo con complejidades temporal y espacial de por lo menos $3n$ y $n + 1$ respectivamente, al aceptar una cadena de longitud n.

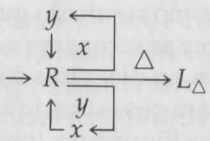
2. Diseñe una máquina de Turing que acepte cualquier cadena cuyo comienzo sea x mediante un cálculo con complejidades espacial y temporal de 2.

3. Determine las complejidades espacial y temporal del cálculo realizado por la máquina de Turing



al procesar la cadena xyx. ¿Qué pasa con la cadena $xyxx$?

4. Determine las complejidades espacial y temporal del cálculo realizado por la máquina de Turing



al ponerse en marcha con la configuración de cinta $\underline{\triangle}xxyyx\triangle\triangle\triangle\cdots$. ¿Qué pasa con la configuración inicial $\underline{\triangle}yyx\triangle xxy\triangle\triangle yyxx\triangle\triangle\triangle\cdots$?

5.2 COMPLEJIDAD DE LOS ALGORITMOS

En general, distintas aplicaciones del mismo algoritmo producen cálculos diferentes. Por ejemplo, el cálculo que efectúe el algoritmo de búsqueda binaria dependerá del contenido de la lista y del valor que se busca o, como ya vimos, el algoritmo de multiplicación tradicional producirá cálculos diferentes con distintos valores de entrada. Intuitivamente, para considerar si un algoritmo es complejo hay que basarse en la cuestión de si los cálculos son complejos o no. Empero, ¿qué sucede si algunas aplicaciones del algoritmo nos llevan a cálculos sencillos mientras que otras generan cálculos complejos?

¿Debemos considerar que el algoritmo es sencillo o complejo? La presente sección trata estas preguntas en el contexto de la complejidad temporal.

Complejidad temporal de las máquinas de Turing

Comenzamos nuestro estudio de la complejidad de algoritmos considerando los algoritmos en el contexto de las máquinas de Turing. Cada máquina de Turing no es más que la implantación de un algoritmo, representado en la forma del diagrama de transiciones de la máquina.

Consideremos una máquina de Turing para comparar dos cadenas de igual longitud en $\{x, y, z\}^*$. Suponemos que estas cadenas están escritas en la cinta de la máquina, una tras la otra y separadas por un asterisco (para comparar las cadenas $yxxz$ y $yxzx$, iniciaríamos la máquina con la configuración de cinta $\underline{\triangle}yxxz*yxzx\triangle\triangle\triangle\cdots$.) La tarea de la máquina es decidir si la cadena anterior al asterisco es idéntica a la que viene después del asterisco. La máquina deberá detenerse con la configuración de cinta $\underline{\triangle}Y\triangle\triangle\triangle\cdots$ si las cadenas son iguales, y con $\underline{\triangle}N\triangle\triangle\triangle\cdots$ si son diferentes.

La técnica que utiliza nuestra máquina de Turing es comparar repetidamente los elementos correspondientes de las cadenas hasta haber considerado todos los pares o detectar una discrepancia. Esto se hace leyendo el primer símbolo de la primera cadena y luego moviendo la cabeza hasta el primer símbolo de la segunda, para confirmar si son iguales. Si lo son, la máquina regresa a la primera cadena para observar el segundo símbolo antes de pasar a la segunda cadena y verificar que su segundo elemento sea igual. Así, el proceso de comparación da como resultado que la cabeza de la cinta alterne entre las dos cadenas conforme se comparan sus elementos.

En la figura 5.3 se muestra un diagrama compuesto para la máquina. Al parecer el tiempo requerido para que esta máquina complete su tarea depende de las cadenas de entrada. Específicamente, tomará más tiempo reconocer que dos cadenas largas son idénticas que llegar a la misma conclusión con dos más cortas. De hecho, se trata de una característica común de los algoritmos: el tiempo requerido para ejecutar un algoritmo tiende a ser una función de la longitud de la entrada. En nuestro caso, la siguiente fórmula indica el tiempo necesario (medido en pasos ejecutados) para confirmar que dos cadenas de longitud n son idénticas:

$$2n^2 + 10n + 9$$

Esto incluye $2n^2 + 5n + 1$ pasos para completar el proceso de comparación, $5n + 4$ pasos para mover la cabeza hacia el extremo derecho de la entrada y borrar la cinta de derecha a izquierda, tres pasos para escribir el símbolo Y en la cinta y, por último, un paso para trasladarse al estado de parada (este último paso se debe a la forma en que construimos las máquinas compuestas en el capítulo 3). Entonces, para confirmar que dos cadenas de longitud cuatro son

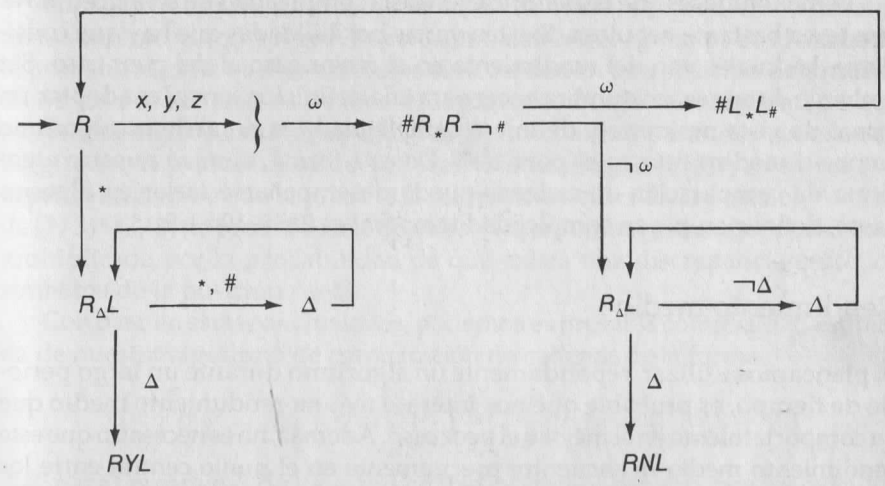


Figura 5.3 Máquina de Turing para comparar cadenas

idénticas, la complejidad temporal sería 81 (se requieren 81 pasos), mientras que el proceso de comparación de dos cadenas idénticas de longitud 10 tendría una complejidad temporal de 309.

Sin embargo, aunque las cadenas que se comparan tengan longitud n, esto no quiere decir que los cálculos realizados por la máquina tendrán una complejidad temporal de exactamente $2n^2 + 10n + 9$. Si las cadenas no son iguales, la máquina dejará de comparar elementos tan pronto como detecte una discrepancia. Para ser más precisos, si sólo los primeros r elementos son idénticos, donde $r < n$, la máquina llevará a cabo un cálculo que consiste en sólo $r(2n + 4) + 6n + 10$ pasos. Por lo tanto, al comparar cadenas de longitud cuatro, donde los primeros dos elementos son iguales pero los terceros difieren, la máquina sólo ejecutará $2(8 + 4) + 24 + 10 = 58$ pasos, en vez de 81. Además, si las cadenas difieren en la primera posición, sólo se ejecutarán 34 pasos.

Al evaluar la complejidad de un algoritmo, generalmente se manejan estas variaciones en el rendimiento de un algoritmo identificando las situaciones de mejor caso y peor caso. Así, se garantiza que cualquier aplicación del algoritmo caiga en este intervalo. En nuestro ejemplo es fácil ver que, al comparar cadenas de longitud n, el cálculo más largo ocurrirá cuando las cadenas sean idénticas, mientras que el más breve se presentará cuando las cadenas difieran en la primera posición. Concluimos que la complejidad temporal de cualquier cálculo se hallará en el intervalo de $6n + 10$ (mejor caso) a $2n^2 + 10n + 9$ (peor caso). (En realidad, la máquina ejecutará cálculos más breves en ciertos casos de entradas inválidas, como una cinta completamente en blanco, pero sólo nos interesa el desempeño de la máquina con entradas válidas.)

Vemos entonces que la identificación de la complejidad de un algoritmo es una tarea bastante nebulosa. Existen varias posibilidades que hay que considerar, las cuales van del rendimiento en el mejor caso al del peor caso. Sin embargo, lo que se acostumbra hacer para una definición formal es adoptar un punto de vista pesimista y definir la complejidad temporal de un algoritmo como su rendimiento en el peor caso. De esta forma, aunque nuestro algoritmo de comparación de cadenas pueda desempeñarse mejor en algunos casos, definimos que su complejidad temporal es $2n^2 + 10n + 9$.

Rendimiento medio

Si planeamos utilizar repetidamente un algoritmo durante un largo periodo de tiempo, es probable que nos interese más su rendimiento medio que su comportamiento en el mejor o el peor caso. Además, no es necesario que este rendimiento medio se encuentre precisamente en el punto central entre los extremos. Por ejemplo, si las situaciones que producen el mejor funcionamiento de un algoritmo son muy raras, entonces su rendimiento real durante un periodo puede tender hacia el peor caso, y viceversa. Evaluemos entonces la complejidad temporal media (o estimada) de nuestra máquina de Turing para comparar cadenas, suponiendo que cada disposición de símbolos de longitud n tiene la misma probabilidad de ocurrencia que las demás.

Estos valores estimados se calculan multiplicando primero la complejidad de cada cálculo posible por la probabilidad de que ocurra dicho cálculo, y luego sumando estos productos. Si la aplicación de un algoritmo puede producir m cálculos distintos con complejidades $c_1, c_2, \cdots, c_m$ y probabilidades de ocurrencia $p_1, p_2, \cdots, p_m$, entonces la complejidad estimada (la complejidad media después de varias aplicaciones del algoritmo) sería

$$\sum_{i=1}^{m} p_i c_i$$

En nuestro ejemplo de comparación de cadenas, cualquier aplicación del algoritmo con cadenas de longitud n nos dará como resultado una de las $n + 1$ posibilidades siguientes: no encontrar discrepancias entre las cadenas, encontrar una discrepancia entre los primeros elementos, encontrar una discrepancia entre los segundos elementos, $\cdots$, o encontrar una discrepancia entre los últimos elementos. Ya evaluamos la complejidad de cada uno de estos casos y encontramos que cuando no hay discrepancias se requieren

$$2n^2 + 10n + 9$$

pasos, a la vez que se ejecutarán

$$r(2n + 4) + 6n + 10$$

pasos si la primera discrepancia se presenta en la posición $r + 1$.

Para encontrar la probabilidad de ocurrencia de cada caso, nuestro razonamiento es como sigue. Puesto que existen tres símbolos en el alfabeto, la probabilidad de que los símbolos concuerden en una posición determinada de las dos cadenas es de sólo 1/3, mientras que la probabilidad de que exista una discrepancia es de 2/3. Por lo tanto, al comparar dos cadenas de longitud n, la probabilidad de que sean idénticas es de $(1/3)^n$, mientras que la probabilidad de que la primera discrepancia ocurra en la posición $r + 1$ es de $(1/3)^r (2/3)$ (la probabilidad de que concuerden las primeras r posiciones multiplicada por la probabilidad de que exista una discrepancia entre los símbolos de la posición $r + 1$).

Con base en estas conclusiones, podemos expresar la complejidad estimada de nuestro algoritmo de comparación de cadenas de la forma

$$\left(\frac{1}{3}\right)^n (2n^2 + 10n + 9) + \sum_{r=1}^{n-1} \left(\frac{1}{3}\right)^r \left(\frac{2}{3}\right) [r(2n + 4) + 6n + 10]$$

Así, para las cadenas de longitud cuatro, podemos esperar que los cálculos producidos por el algoritmo contengan un promedio de $39\,{}^{74}/_{81}$ pasos (véase Fig. 5.4), comparado con los 34 pasos del mejor caso y los 81 del peor. Entonces, en este ejemplo, el rendimiento estimado se aproxima al funcionamiento del algoritmo en el mejor caso, una situación que debe ser del agrado del usuario potencial del algoritmo.

Análisis informal de algoritmos

Ahora nos dirigimos a los problemas de la evaluación de la complejidad temporal de los algoritmos en aquellos ambientes que son mucho menos precisos que los de una máquina de Turing. En estos ambientes,

Posición en la cual se detecta la primera discrepancia	Longitud de cálculo en ese caso	Probabilidad de ocurrencia	Producto de la complejidad y la probabilidad
1	34	$^2/_3$	$22\ ^2/_3$
2	46	$^2/_9$	$10\ ^2/_9$
3	58	$^2/_{27}$	$4\ ^8/_{27}$
4	70	$^2/_{81}$	$1\ ^{59}/_{81}$
no hay discrepancias	81	$^2/_{81}$	1
			Complejidad estimada = $39\ ^{74}/_{81}$

Figura 5.4 Cálculo del rendimiento promedio del proceso de comparación de cadenas

determinar la complejidad temporal no es tan sencillo como calcular el número de pasos que se ejecutarán. De hecho, la capacidad para predecir la complejidad temporal con este método se basa en la suposición, conocida como **suposición de costo uniforme**, de que cada uno de los pasos requerirá la misma cantidad de tiempo. Ésta es una suposición razonable en el caso de una máquina de Turing teórica, pero es posible que la suposición de costo uniforme no sea válida al tratar con otros sistemas. Por ejemplo, en un computador moderno generalmente se requiere más tiempo para multiplicar dos valores que para sumarlos, y una operación de entrada y salida consume mucho más tiempo que una operación efectuada en la unidad central de procesamiento.

Por fortuna, en estas situaciones lo único que se requiere es una estimación de la complejidad temporal del algoritmo que sea lo suficientemente precisa para comparar las opciones disponibles. Por ejemplo, al elegir entre dos técnicas de búsqueda en una lista, no es necesario conocer con exactitud las complejidades temporales que implican, sino basta contar con información suficiente para evaluar las ventajas relativas de las técnicas propuestas. En estos casos normalmente se identifican los pasos dominantes de cada técnica, se estima el número de veces que se ejecutarán estos pasos y se basa la comparación en estas estimaciones. Esto fue lo que hicimos hace poco, cuando comparamos la técnica secuencial de búsqueda en una lista con el método binario. Específicamente, estimamos el número de entradas de la lista que cada estrategia evaluaría y basamos nuestra comparación en estos valores.

Suponga entonces que queremos evaluar la complejidad temporal del algoritmo de clasificación por inserción expresado por el segmento de programa tipo ALGOL/Pascal/Ada de la figura 5.5. Recuerde que un algoritmo de clasificación por inserción ordena una lista designando elementos sucesivos como el elemento pivote (véase el ciclo repeat de la figura 5.5) y, conforme se designa cada pivote, se coloca en el lugar correcto entre sus predecesores (véase el ciclo while de la figura 5.5). Entonces, la ejecución de la rutina de la figura 5.5 estaría dominada por el proceso de comparación y movimiento de los elementos de la lista, siguiendo las indicaciones del ciclo while. Así mismo, debemos esperar que el tiempo requerido para ejecutar el algoritmo sea aproximadamente equivalente al número de veces que se ejecuta este ciclo.

Determinemos cuántas veces se ejecutará el cuerpo de este ciclo en el peor caso. No es difícil ver que este caso se presentará si la lista se halla en orden inverso, ya que el ciclo while se ejecutará para cada predecesor de cada uno de los elementos pivote. Así, cuando el segundo elemento es el pivote, el cuerpo del ciclo se ejecutará una vez; si el pivote es el tercer elemento, se ejecutará dos veces; y, en términos generales, cuando el pivote sea el elemento m, el cuerpo del ciclo se ejecutará $m - 1$ veces. En resumen, si la lista contuviera n elementos, el cuerpo del ciclo while se ejecutaría un total de

$$1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2} \; \frac{1}{2}(n^2 - n)$$


```
program ClasificaciónPorInserción (inout: Lista;in: LongitudLista)
var
    PosiciónPivote, I: integer;
    Pivote: TipoElementoLista;
begin
    if (LongitudLista ≥ 2) then
      begin
        PosiciónPivote : = 2;
        repeat
          Pivote : = Lista[PosiciónPivote];
          I : = PosiciónPivote;
          while (I > 1 and Lista[I – 1] > Pivote) do
            begin
              Lista[I] = Lista[I – 1];
              I : = I – 1
            end;
          Lista[I] : = Pivote;
          PosiciónPivote : = PosiciónPivote + 1
        until (PosiciónPivote > LongitudLista)
    end.
```

Figura 5.5 Clasificación por inserción

veces. Por lo tanto, estimamos que el funcionamiento del algoritmo en el peor caso tendrá una complejidad temporal proporcional a $n^2 - n$, donde n es la longitud de la lista de entrada.

Por supuesto, se trata de una aproximación y sólo la debemos emplear para conocer los aspectos generales de la eficiencia del algoritmo, no para presentar afirmaciones precisas acerca de los requisitos temporales del mismo. Por ejemplo, no podemos afirmar que el tiempo exacto requerido por el algoritmo para clasificar listas de n elementos siempre estará limitado por una constante específica múltiplo de $n^2 - n$, pero sí podemos afirmar que si el algoritmo se aplica a listas cada vez más largas, entonces su rendimiento en el peor caso tenderá a aumentar en proporción a una expresión cuadrática en la longitud de la lista. Es decir, si duplicamos la longitud de la lista, debemos esperar que los requisitos temporales del algoritmo aumenten en un factor de aproximadamente cuatro.

Estas aproximaciones son de gran utilidad cuando hay que elegir entre varias estrategias para resolver un problema en ciertos ambientes. Si se encuentra que una solución tiene una complejidad temporal proporcional a n^2 mientras que otra tiene una complejidad proporcional a n^3, tenderíamos a elegir la primera puesto que, potencialmente, la segunda opción requerirá mucho más tiempo con entradas de gran magnitud.

Sin embargo, hay que advertir al lector que no debe tomar estas decisiones con demasiada rapidez. La aplicación específica puede ser el mejor caso para

el algoritmo n^3 pero estar cerca del peor caso para el algoritmo n^2. Considere además un algoritmo que resuelve un problema en 2^n pasos, comparado con otro que requiere n^3 pasos para resolver el mismo problema. Con entradas grandes, la segunda opción será mucho más eficiente que la primera, pero ésta se desempeñará mejor si las entradas se encuentran en el intervalo de dos a diez.

Así, llegamos a la conclusión de que las consideraciones necesarias para juzgar el rendimiento estimado de un algoritmo varían de acuerdo con la situación. En un ambiente de aplicación, muchas veces no bastan las evaluaciones del peor caso, del mejor caso e incluso del caso promedio; en cambio, hay que considerar el ambiente donde se aplicará el algoritmo.

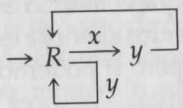
Por último, el lector debe tener presente que hemos hablado de la complejidad temporal de los algoritmos sin contar con un conocimiento preciso del mecanismo empleado para su ejecución. Por esto, nuestras afirmaciones son válidas sin importar si usamos las lentas máquinas de la década de 1950, las veloces máquinas actuales o las supermáquinas del mañana. Este hecho tendrá importancia cuando consideremos algoritmos con complejidades temporales tan grandes que su ejecución no es práctica, pues veremos que esta impracticabilidad se debe al algoritmo y no a la tecnología: ejecutar el algoritmo con mayor velocidad tendrá un efecto mínimo sobre su practicabilidad.

Ejercicios

1. Modifique el diagrama de la figura 5.3 para que responda N ante cualquier entrada inválida.

2. Evalúe la complejidad temporal de las máquinas de desplazamiento S_R y S_L definidas en la sección 3.2.

3. ¿Cuál es la complejidad temporal del caso promedio de la máquina de Turing



al recibir cadenas de entrada de $\{x, y\}^*$?

4. Lleve a cabo un análisis de caso promedio del tiempo que requiere la clasificación por inserción de la figura 5.5.

5.3 COMPLEJIDAD DE LOS PROBLEMAS

Veamos ahora la tarea de medir la complejidad de un problema. Intuitivamente, esta medición debe relacionarse con la complejidad de las soluciones del

problema: un problema es difícil porque no es fácil resolverlo. Sin embargo, no siempre hay que declarar que un problema es complejo simplemente porque es difícil llegar a su solución; casi siempre existe una forma difícil de resolver un problema. Un problema debe declararse como complejo únicamente cuando no tenga una solución sencilla. Para ser más precisos, nos gustaría decir que la complejidad de un problema es la complejidad de su solución más sencilla. Por desgracia, veremos que la abundancia de soluciones para un problema hace muy difícil la identificación de una solución más sencilla. De hecho, se ha mostrado que varios problemas no tienen una solución más sencilla.

El problema de comparación de cadenas

Uno de los principales obstáculos al tratar de establecer la complejidad de un problema por medio de la búsqueda de una solución óptima es que, al parecer, se puede mejorar cualquier solución que encontremos. Por consiguiente, la búsqueda de una solución óptima se convierte en una tarea sin fin de descubrimiento de mejores soluciones. Investiguemos este fenómeno con mayor detenimiento regresando al ambiente de las máquinas de Turing y al problema de comparar dos cadenas de la misma longitud en $\{x, y, z\}^*$.

En la figura 5.3 se presentó una solución para este problema. Allí, la estrategia fue comparar cada uno de los símbolos de la primera cadena con el símbolo correspondiente de la segunda hasta encontrar una discrepancia o el fin de las cadenas. Recordará el lector que la complejidad temporal de este algoritmo (su funcionamiento, en el peor caso) era $2n^2 + 10n + 9$, donde n es la longitud de las cadenas de entrada.

Sin embargo, la anterior no es una solución óptima para el problema. Podemos producir una más rápida si diseñamos una máquina de Turing que compare dos símbolos a la vez, una estrategia representada por el diagrama compuesto de la figura 5.6. La máquina basada en este diagrama comparará las cadenas examinando dos elementos de la primera cadena antes de mover la cabeza de la cinta a la otra cadena para investigar sus posiciones correspondientes.

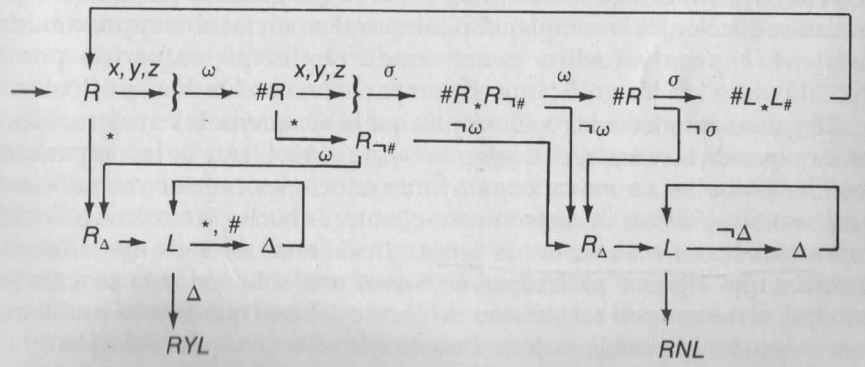


Figura 5.6 Otra máquina de Turing para comparar cadenas de igual longitud

Si lo comparamos con nuestra solución original, este enfoque alternativo reduce aproximadamente a la mitad el número de veces que hay que mover la cabeza de una cadena a la otra, por lo que esperaríamos encontrar una reducción correspondiente en el tiempo requerido para el cálculo. Esta sospecha se refuerza al observar que el peor caso de nuestra solución alternativa sólo requiere

$$n^2 + 7n + 8$$

pasos para confirmar que dos cadenas de longitud n son idénticas, en comparación con los

$$2n^2 + 10n + 9$$

pasos necesarios en nuestra solución original. La figura 5.7 presenta un resumen de estas expresiones para diversos valores de n.

Al reconocer que podemos mejorar nuestra solución para el problema de comparación de cadenas, comparando dos símbolos a la vez, se abre la puerta para una plétora de posibles mejoras. Si comparamos k símbolos a la vez, donde k es cualquier entero mayor que 1, podríamos esperar una solución aproximadamente $1/k$ veces mejor que nuestra solución original.

Además, se trata sólo de una técnica, entre un gran número de métodos posibles para reducir la complejidad temporal. Una técnica más general es extender el alfabeto que se usa, para que cada patrón de k símbolos del alfabeto original se pueda representar con un solo símbolo del sistema extendido. Así, producimos un nuevo algoritmo para resolver el problema original, que primero recodifica la entrada a este alfabeto extendido y luego simula el viejo algoritmo con esta forma recodificada. Observe que una sola operación de este sistema recodificado puede simular varios pasos del algoritmo original. Una sola operación de escritura puede simular los $2k - 1$ pasos (k pasos de escritura y $k - 1$ pasos de movimiento) requeridos por el algoritmo original para escribir símbolos en k celdas consecutivas de la cinta. Por supuesto, el beneficio real de esta técnica varía de acuerdo con la situación, pero como regla general podemos esperar una reducción de la complejidad temporal en un factor de aproximadamente $1/k$ para aquellos casos donde el tiempo requerido por la recodificación sea insignificante en comparación con los demás cálculos.

Llegamos entonces a la conclusión de que la búsqueda de la mejor solución (en términos de la complejidad temporal) para el problema de la comparación de cadenas nos lleva a una cadena sin fin de soluciones, cada una más eficiente que la anterior. Se trata de un fenómeno común; de hecho, el teorema conocido como teorema de aceleración de Blum, el cual analizaremos más adelante, establece que algunos problemas no tienen una solución más sencilla (en realidad, el teorema de aceleración de Blum establece que existen problemas para los cuales es posible mejorar *considerablemente* cualquier solución).

Por lo anterior, es muy difícil clasificar los problemas de acuerdo con su complejidad; de hecho, se trata de una tarea que muchas veces no llegamos

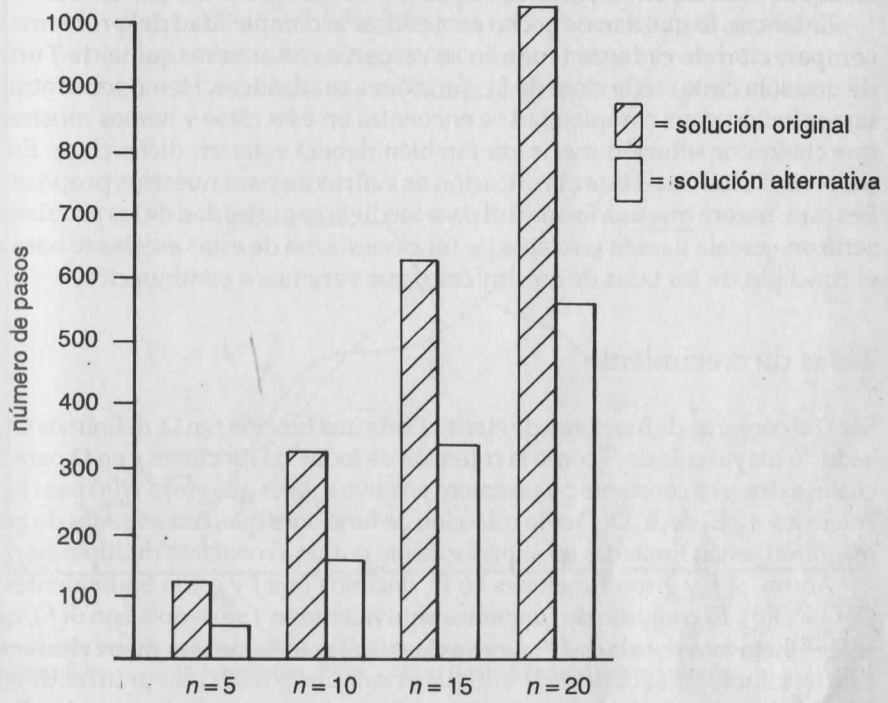


Figura 5.7 Comparación de nuestras soluciones original y alternativa para el problema de comparación de cadenas

a concluir. En algunas situaciones basta con establecer un límite superior para la complejidad de un problema, como hemos hecho en el caso del problema de comparación de cadenas. Hemos encontrado una solución cuya complejidad temporal es $2n^2 + 10n + 9$; así, sabemos que la complejidad del problema no puede ser mayor que esto. Además, si esta solución es suficiente para nuestros fines, entonces no necesitamos tratar de clasificar con mayor precisión el problema.

Sin embargo, en otros casos las soluciones conocidas para un problema pueden ser demasiado complejas. En esta situación, nuestra tarea es encontrar una solución mucho mejor o mostrar que no existe dicha solución. En el apéndice D demostraremos este segundo enfoque, al mostrar que cualquier solución para el problema de comparación de cadenas con una máquina de Turing (de una sola cinta) debe tener una complejidad temporal que sea por lo menos una expresión cuadrática en la longitud de las listas que se comparan. En otras palabras, conforme aumenta la longitud de las cadenas, el tiempo requerido para cualquier solución mejorada aumentará en proporción a las soluciones ya obtenidas. Lo anterior quiere decir que si estas soluciones

conocidas son demasiado complejas para una aplicación, es probable que cualquier otra solución para el problema presente el mismo panorama.

Entonces, lo que hemos hecho es clasificar la complejidad del problema de comparación de cadenas (cuando se resuelve con una máquina de Turing de una sola cinta) en la clase de las funciones cuadráticas. Hemos encontrado una solución cuya complejidad se encuentra en esta clase y hemos mostrado que cualquier solución mejorada también deberá estar en dicha clase. En la mayoría de los casos esta clasificación es suficiente para nuestros propósitos. Por esto, parece que una forma útil para medir la complejidad de los problemas sería una escala basada en clases de funciones. Una de estas escalas se basa en el concepto de las tasas de crecimiento, que veremos a continuación.

Tasas de crecimiento

Sea Ω el conjunto de funciones de $\mathbb{N}$ en $\mathbb{N}$. Dada una función f en Ω, definimos $O(f)$ leído "o mayúscula de f" como la colección de todas las funciones g en Ω para las cuales existe una constante c y un entero positivo n_0 tales que $g(n) \leq cf(n)$ para todo entero $n > n_0$. Es decir, $O(f)$ es la colección de funciones que, con entradas de gran magnitud, están limitadas superiormente por una constante múltiplo de f.

Ahora, si f y g son funciones en Ω, decimos que f y g son equivalentes si $O(f) = O(g)$. El conjunto de funciones equivalentes a f se denota con $\theta(f)$, que se lee "theta mayúscula de f". A cada clase $\theta(f)$ se le llama **tasa de crecimiento**. Esta terminología se obtiene de que, para cualquier $g \in \theta(f)$, las gráficas de $g(n)$ y $f(n)$, si se extienden lo suficiente, debe encontrarse en el mismo corredor. Para ser más precisos, existen constantes c_1 y c_2 tales que $g(n) \leq c_1 f(n)$ y $f(n) \leq c_2\, g(n)$ para todos los enteros n mayores que un entero positivo fijo n_0. Así, para $n \geq n_0$,

$$\frac{1}{c_2} f(n) \leqslant g(n) \leqslant c_1 f(n)$$

Es decir, la gráfica de $g(n)$ debe yacer finalmente en el corredor entre $\frac{1}{c_2} f(n)$ y $c_1 f(n)$, como se muestra en la figura 5.8. Por lo tanto, aunque la gráfica de $g(n)$ pueda oscilar en este corredor, desde una perspectiva global las funciones, f y g deben crecer con la misma tasa para grandes valores de n (observe que c_1 y c_2 se pueden elegir de modo que $f(n)$ también caiga en el corredor).

Mostramos ahora que el valor absoluto de cualquier polinomio de grado d es $\theta(n^d)$. Dado el polinomio $\sum_{i=0}^{d} a_i n^i$, donde $a_d \neq 0$, entonces, para cada n en $\mathbb{N}^+$,

$$\frac{n^d}{\left|\sum_{i=0}^{d} a_i n^i\right|}$$

se puede reescribir como

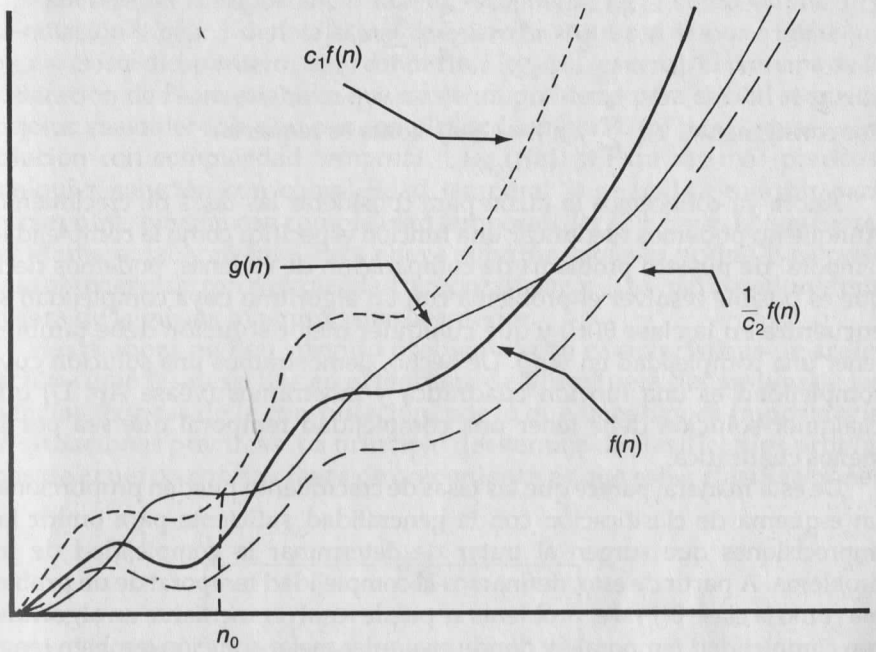


Figura 5.8 Dos funciones, f y g, con la misma tasa de crecimiento

$$\frac{1}{\left|a_d + a_{d-1}\frac{1}{n} + \cdots + a_0 \frac{1}{n^d}\right|}$$

lo que se acerca arbitrariamente a $\frac{1}{|a_d|}$ para grandes valores de n. Entonces, existe un entero positivo n_0 tal que

$$\frac{1}{2|a_d|} \leqslant \frac{n^d}{\left|\sum_{i=0}^{d} a_i n^i\right|} \leqslant \frac{3}{2|a_d|}$$

para todo $n \geq n_0$. La desigualdad del lado izquierdo implica que

$$\left|\sum_{i=0}^{d} a_i n^i\right| \leqslant 2n^d |a_d|$$

mientras que la del lado derecho indica

$$n^d \leqslant \frac{3}{2|a_d|}\left|\sum_{i=0}^{d} a_i n^i\right|$$

Por consiguiente, $O(\sum_{i=0}^{d} a_i n^i) = O(n^d)$ como se requería.

Ahora ya conocemos la razón para considerar las tasas de crecimiento. Aunque no podemos identificar una función específica como la complejidad temporal de nuestro problema de comparación de cadenas, podemos decir que es posible resolver el problema con un algoritmo cuya complejidad se encuentra en la clase $\theta(n^2)$ y que cualquier mejor solución debe también tener una complejidad en $\theta(n^2)$. De hecho, demostramos una solución cuya complejidad es una función cuadrática y mostramos (véase Ap. D) que cualquier solución debe tener una complejidad temporal que sea por lo menos cuadrática.

De esta manera, parece que las tasas de crecimiento pueden proporcionar un esquema de clasificación con la generalidad suficiente para omitir las imprecisiones que surgen al tratar de determinar la complejidad de un problema. A partir de esto, definamos la complejidad temporal de un problema como la clase $\theta(f)$ si el problema se puede resolver mediante un algoritmo con complejidad temporal f y donde cualquier mejor solución también tenga una complejidad temporal en $\theta(f)$. Entonces, establecemos un orden entre las tasas, definiendo $\theta(f) \leq \theta(g)$ para decir $O(f) \subseteq O(g)$. Así mismo, escribimos $\theta(f) < \theta(g)$ para representar $\theta(f) \leq \theta(g)$ y $\theta(f) \neq \theta(g)$. Por ejemplo, $\theta(n^2) < \theta(n^3)$; se considerará más complejo un problema con complejidad $\theta(n^3)$que uno con complejidad $\theta(n^2)$.

Limitaciones de la escala de tasas de crecimiento

La escala de tasas de crecimiento se ha utilizado con éxito para clasificar la complejidad de varios problemas en distintas situaciones. Sin embargo, no tiene la generalidad suficiente para proporcionar un sistema en el cual puedan clasificarse todos los problemas. Una de las razones es que existen problemas para los cuales cualquier solución puede mejorarse indefinidamente, de manera que cada nueva solución corresponda a una distinta tasa de crecimiento. Este resultado se conoce como teorema de la aceleración de Blum, descubierto por M. Blum en 1967. En términos generales, este teorema establece que, en el contexto de las máquinas de Turing (y por ende en muchos otros sistemas computacionales), para cualquier función μ- recursiva $g: \mathbb{N} \rightarrow \mathbb{N}$ existe un problema para el cual cualquier solución, con complejidad temporal $t(n)$, se puede mejorar para obtener una nueva solución con complejidad temporal $g(t(n))$ para todos excepto un número finito de valores en $\mathbb{N}$ (a su vez, esta nueva solución se puede mejorar para obtener una complejidad temporal $g(g(t(n)))$, etc.).

Para resaltar la importancia de esto, escojamos $\lfloor log_2 \rfloor$ como la función g (la notación $\lfloor log_2 x \rfloor$ denota el mayor entero en $\mathbb{N}$ que sea menor o igual que $log_2 x$ si existe dicho entero; de lo contrario, $\lfloor log_2 x \rfloor$ es cero). El teorema de la aceleración de Blum establece que existe un problema para el cual se puede mejorar cualquier solución con complejidad temporal $t(n)$ para obtener otra solución con complejidad temporal $\lfloor log_2(t(n)) \rfloor$. Para ser más precisos, cualquier solución con complejidad temporal 2^n se podría mejorar para formar otra solución con complejidad temporal $\lfloor log_2(2^n) \rfloor = n$, lo que representa una notable mejora. Esta nueva solución también podría mejorarse para obtener otra, con complejidad temporal $\lfloor log_2 n \rfloor$, lo cual constituye una mejora de la misma magnitud que la anterior.

Hasta ahora no han surgido problemas con características de aceleración como las descritas en el teorema de Blum fuera del ámbito de las ciencias teóricas de la computación, por lo que tienen poca importancia en situaciones prácticas. La principal desventaja de clasificar los problemas de acuerdo con sus tasas de crecimiento es que estas clasificaciones

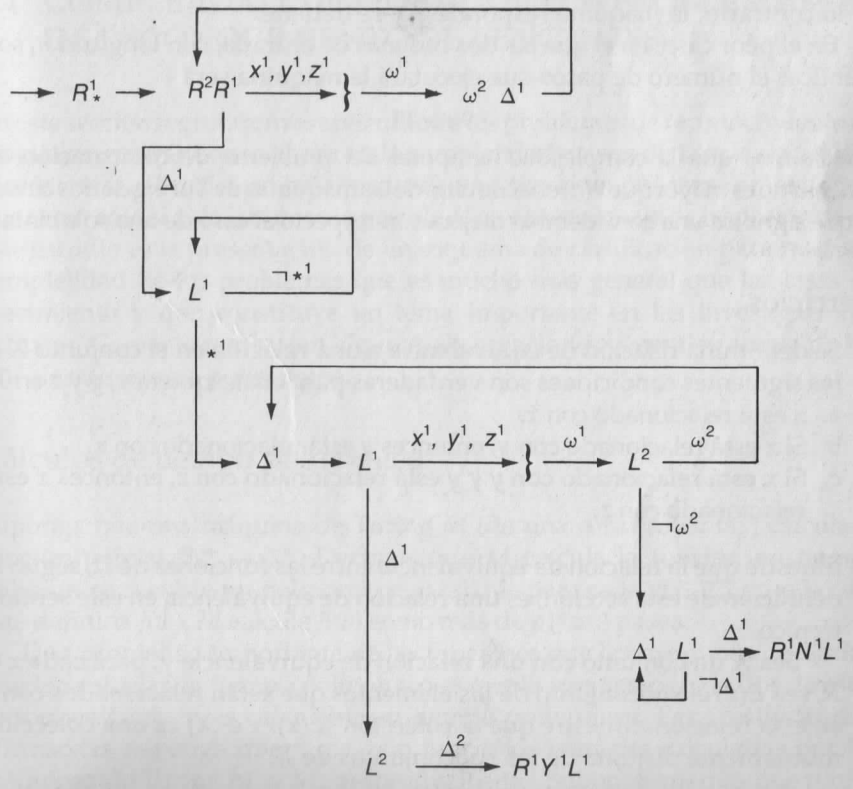


Figura 5.9 Máquina de Turing de dos cintas para resolver el problema de comparación de cadenas

se ven alteradas por los cambios en el sistema computacional subyacente: si cambiamos de un sistema computacional a otro, es probable que la complejidad temporal de un problema corresponda a otra tasa de crecimiento. Por esto, la complejidad de un problema, medida como una tasa de crecimiento, no constituye una propiedad exclusiva del problema, sino además del sistema computacional subyacente.

Este fenómeno se hace evidente en el caso de nuestro problema de comparación de cadenas, el cual, como hemos visto, tiene una complejidad temporal $\theta(n^2)$ en el contexto de las máquinas de Turing de una sola cinta. No obstante, podemos encontrar una mejor solución si cambiamos a una máquina con dos cintas, como la máquina de Turing representada en la figura 5.9. Esta máquina rastrea su cinta de entrada de izquierda a derecha hasta encontrar el asterisco que separa las dos cadenas de entrada. Después copia la segunda cadena a la segunda cinta, a la vez que la borra de la primera. A continuación, la máquina rastrea de izquierda a derecha las cadenas de ambas cintas, comparando los elementos en posiciones correspondientes y borrándolas de la cinta 1 conforme las compara. Si detecta entradas desiguales, la máquina borra el resto de la cinta 1 y responde N antes de detenerse; de lo contrario, la máquina responde Y y se detiene.

En el peor caso, en el que las dos cadenas de entrada, con longitud n, son idénticas el número de pasos que ejecutará la máquina será

$$9n + 11$$

Concluimos que la complejidad temporal del problema de comparación de cadenas no es mayor que $\theta(n)$ en el ámbito de las máquinas de Turing de dos cintas, lo que significa una considerable mejora con respecto al caso de una sola cinta.

Ejercicios

1. Se denomina relación de equivalencia a una relación en el conjunto X si las siguientes condiciones son verdaderas para cualesquiera x, y y z en X:
   a. x está relacionado con x.
   b. Si x está relacionado con y, entonces y está relacionado con x.
   c. Si x está relacionado con y y y está relacionado con z, entonces x está relacionado con z.

   Muestre que la relación de equivalencia entre las funciones de Ω, según la definición de esta sección, es una relación de equivalencia en este sentido técnico.

   Sea X un conjunto con una relación de equivalencia y, para cada $x \in X$, sea $E(x)$ el subconjunto de los elementos que están relacionados con x en esta relación. Muestre que la colección $\{E(x): x \in X\}$ es una colección mutuamente disjunta de los subconjuntos de X.

2. Se dice que un conjunto ordenado es aquel cuyos elementos tienen cierto orden. Así mismo se dice que tiene orden lineal si dos elementos cualesquiera están relacionados por el orden. Por lo tanto, el conjunto de los enteros está ordenado linealmente por el orden $\leq$ ya que, dados dos elementos x y y, entonces $x \leq y$ o $y \leq x$. Empero, el orden "padre de" no es un orden lineal del conjunto de las personas en el mundo, ya que existen individuos x y y para los cuales x no es padre de y ni y es padre de x. Muestre que el orden $\leq$ en la colección de clases de equivalencia de la forma $\theta(f)$ no es lineal, presentando dos funciones f y g tales que ni $\theta(f) \leq \theta(g)$ ni $\theta(g) \leq \theta(f)$.

3. En el texto mostramos que la complejidad temporal de nuestro problema de comparación de cadenas, en el contexto de una máquina de Turing de dos cintas, no era mayor que $\theta(n)$, donde n es la longitud de las cadenas. Complete la clasificación mostrando que la complejidad no es menor que $\theta(n)$.

4. Muestre que $\theta(log_2 n) < \theta(n)$.

5.4 COMPLEJIDAD TEMPORAL DE LOS PROBLEMAS DE RECONOCIMIENTO DE LENGUAJES

En esta sección regresaremos al estudio de los problemas de reconocimiento de lenguajes, centrándonos ahora en la complejidad en vez de la calculabilidad. Nos interesa saber si el procesamiento de un lenguaje es una tarea práctica más que saber sólo si es teóricamente posible. Una consecuencia de importancia de este estudio es la presentación de un esquema de clasificación para medir la complejidad de los problemas que es mucho más general que las tasas de crecimiento y que constituye un tema importante en las investigaciones actuales. Comenzaremos con algunas observaciones generales acerca de los cálculos de tiempo polinómico.

Cálculos de tiempo polinómico

Suponga que una máquina de Turing M (de una o varias cintas) calcula la función parcial $f: \Sigma_1^* \rightarrow \Sigma_2^*$. Decimos que M calcula la función en tiempo polinómico si existe un polinomio $p(x)$ tal que para cada $w \in \Sigma_1^*$ para la cual esté definida $f(w)$, M calcule $f(w)$ en no más de $p(|w|)$ pasos.

Una propiedad importante de las funciones que las máquinas de Turing pueden calcular en tiempo polinómico es que la composición de dos de estas funciones también es calculable en tiempo polinómico. Para justificar esta afirmación, suponga que f_1 y f_2 son funciones parciales calculadas por las máquinas de Turing M_1 y M_2, respectivamente. Suponga además que $p_1(x)$ y $p_2(x)$ son expresiones polinómicas tales que para cada entrada v y w, M_1 calcula $f_1(v)$ en no más de $p_1(|v|)$ pasos y M_2 calcula $f_2(w)$ en un máximo de $p_2(|w|)$

pasos. Ahora considere el tiempo requerido para que la máquina compuesta $\rightarrow M_1M_2$ calcule $f_2 \circ f_1$. Dada la entrada v, M_1 pasará el control a M_2 después de ejecutar no más de $p_1(|v|)$ pasos. Entonces, la cadena producida por M_1, que se proporciona a M_2 como entrada, no puede tener una longitud mayor que $p_1(|v|) + 1$ (podemos suponer que $|v| < p_1(|v|)$ para todas las entradas v de f_1). A su vez, M_2 se detendrá después de ejecutar un máximo de $p_2(p_1(|v|) + 1)$ pasos. Por lo tanto, toda la operación realizada por $\rightarrow M_1M_2$ no requerirá más de $(p_1(|v|) + p_2(p_1(|v|) + 1)$ pasos, lo cual constituye una expresión polinómica en $|v|$.

También es cierto que la clase de las funciones que pueden calcularse en tiempo polinómico con máquinas de Turing de varias cintas es igual que la de las máquinas tradicionales (de una sola cinta). Esto se obtiene si consideramos nuevamente la demostración del teorema 3.1, en la cual mostramos cómo una máquina de Turing de cinta única, puede simular los cálculos de cualquier máquina de varias cintas. Sólo hay que hacer dos observaciones adicionales. En primer lugar, el proceso de simulación descrito en la demostración del teorema 3.1 puede ampliarse para que la máquina de cinta única traduzca su cinta a un formato de una sola cinta de manera que antes de detenerse esté configurada como la cinta 1 de una máquina de cintas múltiples. Esto quiere decir que la máquina de cinta única produce la misma salida que la versión de múltiples cintas. En segundo lugar, si la máquina de varias cintas realiza su tarea en tiempo polinómico, también se efectuará en tiempo polinómico la simulación en una sola cinta.

Una vez establecidos estos antecedentes, pasamos al tema de esta sección: la complejidad de los problemas de reconocimiento de lenguajes.

La clase P

Si M es una máquina de Turing, decimos que M acepta el lenguaje L en tiempo polinómico si $L = L(M)$ y existe un polinomio $p(n)$ tal que el número de pasos necesarios para aceptar cualquier $w \in L(M)$ no sea mayor que $p(|w|)$. Definimos que P es la clase de los lenguajes que las máquinas de Turing pueden aceptar en tiempo polinómico.

Nuestro interés por la clase P surge de la noción intuitiva de que contiene aquellos lenguajes que pueden ser aceptados en un tiempo razonable. Considere, por ejemplo, una máquina de Turing M que acepta cualquier cadena $w \in L(M)$ en una cantidad de tiempo proporcional al polinomio $|w|^2$, en comparación con otra máquina M' que acepta cada cadena $w \in L(M')$ en un tiempo proporcional a la potencia $2^{|w|}$. Si duplicamos la longitud de la entrada de M, digamos de 10 a 20, el tiempo requerido para los cálculos correspondientes aumentaría a lo sumo en un factor de cuatro, mientras que un cambio similar en la entrada de M' daría como resultado un aumento en un factor de $2^{10} = 1024$. Así, conforme aumenta la longitud de las cadenas evaluadas, esperamos que M consuma mucho menos tiempo que M'. De hecho, si la

ejecución de cada paso requiriera un microsegundo, entonces M podría procesar una cadena de longitud 50 en menos de un segundo, mientras que M' necesitaría más de 35 años para procesar la misma cadena (véase Fig. 5.10).

Otra característica importante de P es que permanece estable en un amplio rango de sistemas computacionales. Si cambiamos el sistema computacional, la clase de los lenguajes que pueden aceptarse en tiempo polinómico tiende a permanecer sin cambios. Esto no debe causar sorpresa alguna; después de todo, la clase P consiste en todos los lenguajes que pueden aceptarse en un tiempo $O(n^d)$ para algún $d \in \mathbb{N}^+$, por lo que una clasificación basada en complejidad temporal polinómica contra no polinomial es más general que una que hace distinciones entre las tasas de crecimiento.

El teorema siguiente brinda un testimonio adicional con respecto a la solidez de P. Muestra que la clase de los lenguajes que pueden aceptarse en tiempo polinómico no cambia si pasamos de las máquinas que indican la aceptación con sólo detenerse a aquellas que la indican al detenerse con su cinta configurada como $\underline{\triangle}Y\triangle\triangle\triangle\cdots$.

TEOREMA 5.1

Si una máquina de Turing acepta el lenguaje L en tiempo polinómico, entonces existe otra máquina de Turing que también acepta L en tiempo polinómico pero que indica su aceptación deteniéndose con la configuración de cinta $\underline{\triangle}Y\triangle\triangle\triangle\cdots$.

DEMOSTRACIÓN

En la sección 3.3 planteamos que, dada una máquina de Turing M que acepta el lenguaje L, podíamos construir otra máquina de la forma

$$\rightarrow R_\triangle S_R R * L_\triangle L \# R \rightarrow M_0 \rightarrow R_* \rightarrow \begin{array}{l} \triangle L \; \neg\# \\ \downarrow\# \\ \triangle RYL \end{array}$$

n = longitud de la entrada	Complejidad temporal n^2	Complejidad temporal 2^n
10	.0001 segundos	.0001 segundos
20	.0004 segundos	1.05 segundos
30	.0009 segundos	17.92 minutos
40	.0016 segundos	12.74 días
50	.0025 segundos	35.75 años
60	.0036 segundos	36.6 siglos
70	.0049 segundos	374.8 millones de años

Figura 5.10 La complejidad temporal polinómica n^2 comparada con la complejidad temporal exponencial 2^n

que también acepta L pero lo hace deteniéndose con su cinta configurada como $\underline{\triangle}Y\triangle\triangle\triangle\cdots$. (recuerde que M_0 es en esencia una copia de M, excepto por unos cuantos pasos adicionales en aquellos casos en que el símbolo actual es # o *). Nuestra tarea es mostrar que si M acepta L en tiempo polinómico, entonces esta nueva máquina también debe hacerlo.

Suponga entonces que p es un polinomio y que para cada $w \in L$, M acepta w en un tiempo $p(|w|)$. En bastante sencillo mostrar que el componente de nuestra máquina compuesta que precede a M_0 completará su tarea en un máximo de $q(|w|)$ pasos, donde q es un polinomio. Además, la salida de esta etapa será la configuración de cinta $\#\underline{\triangle}w*\triangle\triangle\triangle\cdots$, por lo que M_0 simulará las acciones de M como si M recibiera la entrada w. La mayoría de las veces, M_0 ejecutará un paso por cada paso de M que se simula, excepto en aquellos casos en que M_0 encuentra el símbolo # o *, pero incluso en esta situación M_0 no ejecuta más de cuatro pasos adicionales. Así, si M hubiera concluido su tarea en el tiempo $p(|w|)$, entonces M_0 la completaría en el tiempo $4p(|w|)$.

Para concluir, una vez que M_0 transfiere el control al componente restante de nuestra máquina compuesta, la porción con datos de la cinta no será mayor que $k = |w| + 3 + 4p(|w|)$, es decir, la longitud de la porción con datos donde comenzó M_0 más el número máximo de celdas donde M_0 pudo haber escrito datos. Sin embargo, esto quiere decir que la última etapa de nuestra máquina compuesta no requerirá más de $3k + 4$ pasos para borrar la cinta, escribir una Y y regresar a la celda del extremo izquierdo. Así, toda la máquina compuesta aceptará el lenguaje L en tiempo polinómico.

∎

Observe que es fácil extender el teorema 5.1 para que incluya máquinas de varias cintas. Si una máquina de Turing puede aceptar un lenguaje L en tiempo polinómico, existe una máquina de Turing de varias cintas que calcula la función parcial definida por

$$f(w) = \begin{cases} Y \text{ si } w \in L \\ \text{indefinida si } w \notin L \end{cases}$$

en tiempo polinómico. Sin embargo, ya mostramos que debe existir entonces una máquina de Turing de una sola cinta que calcule esta misma función en tiempo polinómico. Por consiguiente, *el cambio de las máquinas de Turing tradicionales (de una sola cinta) a máquinas de varias cintas no aumenta la clase de los lenguajes que pueden aceptarse en tiempo polinómico.* Éste es un resultado más que apoya la solidez de la clase P.

Lenguajes decidibles en tiempo polinómico

Decimos que un lenguaje L del alfabeto Σ puede decidirse en tiempo polinómico si existe una máquina de Turing M que decida el lenguaje L y un polinomio $p(x)$ tal que para cada $w \in \Sigma^*$, poniendo en marcha M con la entrada w, el resultado sea un cálculo de no más de $p(|w|)$ pasos. Puesto que el proceso de decisión con respecto a un lenguaje de un alfabeto Σ es calcular una función específica de Σ^* en $\{Y, N\}$, nuestro conocimiento de los cálculos de tiempo polinómico nos permite concluir que la clase de los lenguajes que pueden decidirse en tiempo polinómico con una máquina de Turing de varias cintas es idéntica a la clase que puede ser decidida en tiempo polinómico por las máquinas de una sola cinta.

Ahora nuestro objetivo es mostrar que esta clase de lenguajes es igual que P. Es decir, si restringimos nuestra consideración a los cálculos en tiempo polinómico, la capacidad de una máquina de Turing para aceptar un lenguaje es equivalente a la capacidad de una máquina de Turing para decidir el lenguaje. Comenzamos nuestra demostración con el lema siguiente, el cual dice esencialmente que una función polinómica puede calcularse en tiempo polinómico.

LEMA

Para cualquier polinomio $p(x)$ con coeficientes en $\mathbb{N}$, existen un polinomio $q(x)$ y una máquina de Turing que, al ponerse en marcha con una cadena de entrada de longitud n, se detendrá después de efectuar un máximo de $q(n)$ pasos con la cinta configurada como $\underline{\triangle}w\triangle\triangle\triangle\cdots$, donde w es una cadena de unos de longitud $p(n)$.

DEMOSTRACIÓN

Aplicamos la inducción al grado de $p(x)$. Si este grado es cero, $p(x) = a_0$ para alguna constante a_0. En este caso, la máquina de Turing sólo tiene que marcar su celda del extremo izquierdo con una marca especial (un paso), borrar la cadena de entrada de izquierda a derecha ($2n + 1$ pasos), mover la cabeza de regreso a la marca especial ($n + 1$ pasos), escribir una cadena de a_0 unos a la derecha de la marca especial ($2a_0$ pasos), colocar de nuevo la cabeza en la marca especial (a_0 pasos) y reemplazar la marca especial con un espacio en blanco (un paso). Esto requiere un total de $3n + (3a_0 + 4)$ pasos, un polinomio de n, según lo requerido.

Supongamos ahora que el lema es verdadero para los polinomios de grado menor que r y que $p(x) = \sum_{i=0}^{r} a_i x^i$. Reacomodando la expresión para $p(x)$, obtenemos

$$p(x) = \left(\sum_{i=1}^{r} a_i x^{i-1}\right)x + a_0$$

Sin embargo $\sum_{i=1}^{r} a_i x^{i-1}$ (que representamos con $p_1(x)$) es un polinomio con coeficientes en $\mathbb{N}$ de grado menor que r, por lo que debe existir una máquina de Turing M que, dada una entrada de longitud n, produzca una salida de $p_1(n)$ unos en no más de $q(n)$ pasos, donde $q(x)$ es una expresión polinómica. Construimos la máquina de Turing para calcular $p(n)$ efectuando las siguientes modificaciones a M.

Primero, agregue a M una segunda y una tercera cintas luego, altere M para que copie su entrada a la segunda cinta antes de calcular $p_1(n)$. (Este proceso de copiado requerirá $2n + 2$ pasos para rastrear la entrada de la primera cinta hasta detectar un espacio en blanco y luego devolver la cabeza a la celda del extremo izquierdo, otros $2n$ pasos para mover hacia la derecha la cabeza de la segunda cinta durante la escritura de la cadena de entrada, y otros n pasos para devolver esta cabeza a la celda del extremo izquierdo. Así, esta modificación añade otros $5n + 2$ pasos al cálculo total efectuado por M.)

Segundo, modifique M para que, después de calcular $p_1(n)$, la máquina prosiga con la copia de la cadena de $p_1(n)$ unos de su primera cinta a la tercera, devuelva la cabeza de la primera cinta al extremo izquierdo borrando las celdas, produzca una cadena de $np_1(n)$ unos en la primera cinta concatenando copias de la tercera cinta conforme se rastrean los símbolos de la segunda, y por último agregue a_0 unos a la cadena de la primera cinta antes de devolver la cabeza a la celda del extremo izquierdo de la primera cinta (véase Fig. 5.11). La ejecución de este proceso requerirá

$$4np_1(n) + 6p_1(n) + 2n + 3a_0 + 3$$

pasos.

En resumen, la versión modificada de M, al recibir una entrada de longitud n, producirá en su primera cinta una cadena de salida de unos de longitud

$$\left(\sum_{i=1}^{r} a_i n^{i-1}\right) n + a_0$$

sin necesitar más de

$$[5n + 2] + q(n) + [4np_1(n) + 6p_1(n) + 2n + 3a_0 + 3]$$

pasos. Sin embargo, como $q(n)$ y $p_1(n)$ son polinomios de n, esta expresión polinómica también es un polinomio de n. Por lo tanto, la máquina modificada satisface las condiciones del teorema.

∎

Ahora usaremos este lema para mostrar que la capacidad de aceptar un lenguaje en tiempo polinómico con una máquina de Turing equivale a la capacidad para decidir el lenguaje en tiempo polinómico.

TEOREMA 5.2

Si una máquina de Turing puede aceptar un lenguaje L en tiempo polinómico, entonces existe una máquina de Turing que decide L en tiempo polinómico.

DEMOSTRACIÓN

Podemos suponer que la máquina que acepta L en tiempo polinómico nunca sufre una terminación anormal, ya que de ser así podríamos

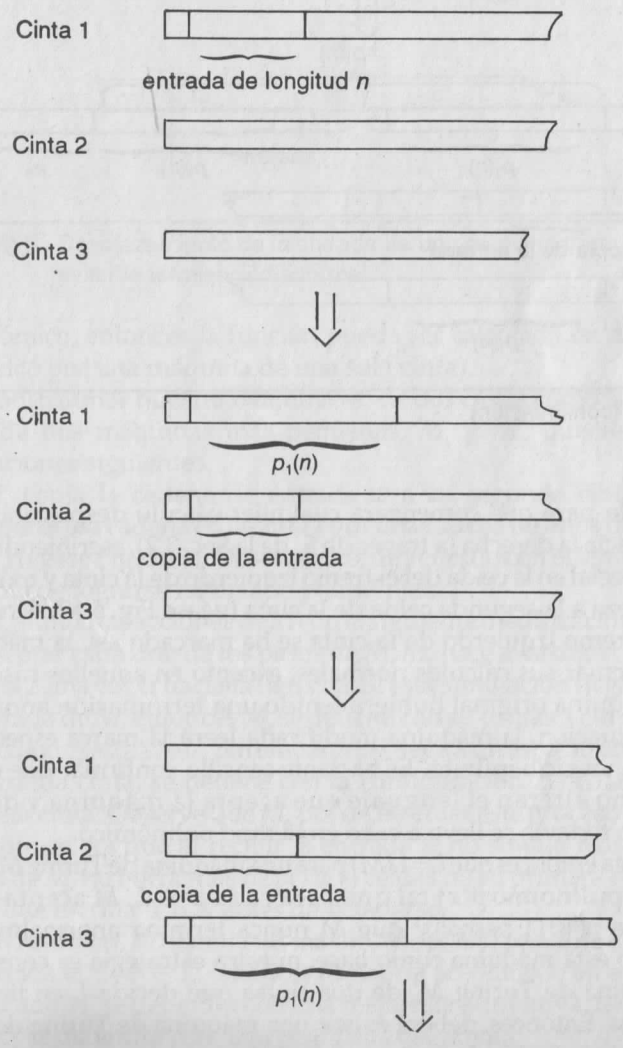


Figura 5.11 Cálculo del valor de un polinomio (continúa en la página siguiente)

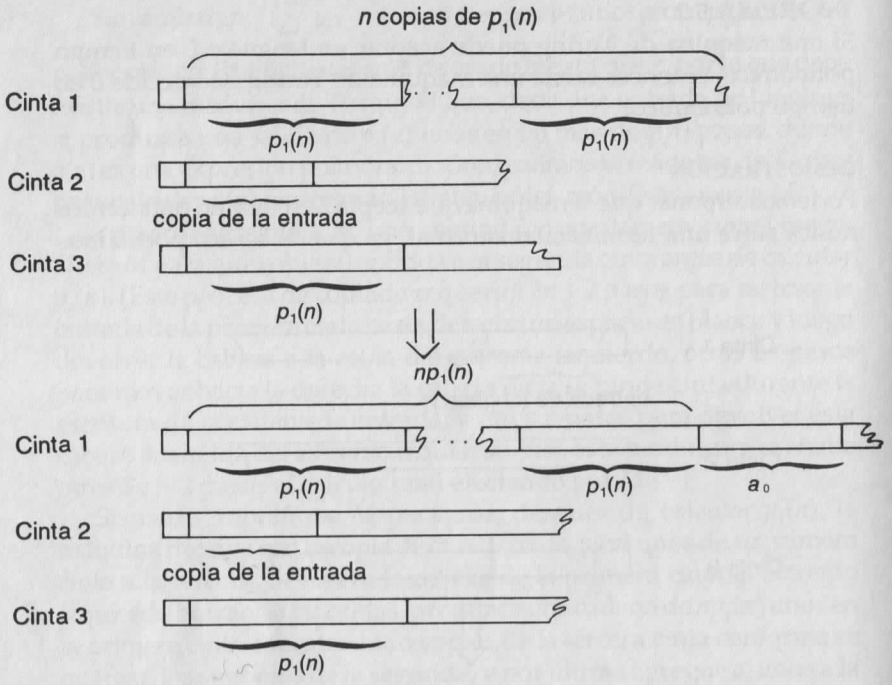


Figura 5.11 (continuación)

modificarla para que comenzara cualquier cálculo desplazando su entrada hacia la derecha (a través de S_R de la Sec. 3.2), escribiendo una marca especial en la celda del extremo izquierdo de la cinta y moviendo su cabeza a la segunda celda de la cinta (véase Fig. 5.12). Una vez que el extremo izquierdo de la cinta se ha marcado así, la máquina puede efectuar sus cálculos normales, excepto en aquellos casos en que la máquina original hubiera tenido una terminación anormal. En esta situación, la máquina modificada leerá la marca especial y entrará en un ciclo infinito. Es bastante sencillo confirmar que estos cambios no alteran el lenguaje que acepta la máquina y que la aceptación todavía se lleva a cabo en tiempo polinómico.

Suponga entonces que $L = L(M)$ para una máquina de Turing M; que existe un polinomio $p(x)$ tal que para cada $w \in L$, M acepta w en no más de $p(|w|)$ pasos; y que M nunca termina anormalmente. Utilizando esta máquina como base, nuestra estrategia es construir una máquina de Turing M' de dos cintas que decida L en tiempo polinómico. Entonces, deberá existir una máquina de Turing de una sola cinta que decida L en tiempo polinómico (si una máquina de varias cintas puede calcular una función de Σ^* en $\{Y, N\}$ en tiempo

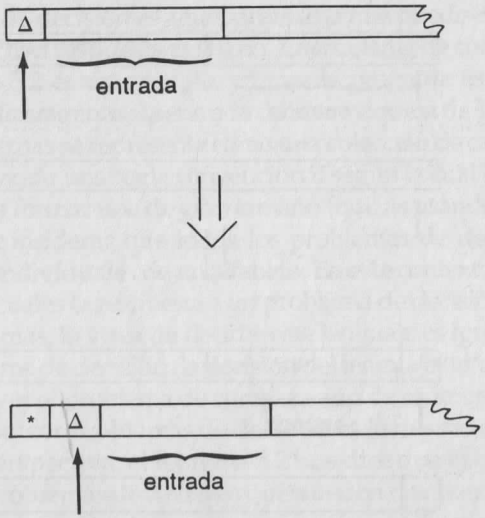


Figura 5.12 Desplazamiento de la entrada de una máquina para evitar la terminación anormal

polinómico, entonces la función puede ser calculada en tiempo polinómico por una máquina de una sola cinta).

Construimos nuestra máquina M' de dos cintas como la composición de dos máquinas más pequeñas, M_1 y M_2, que realizan las operaciones siguientes.

M_1 copia la cadena de entrada w a su segunda cinta y luego reemplaza esa cadena en la cinta 2 por una cadena de unos de longitud $p(|w|)$ (véase Fig. 5.13). Observe que, de acuerdo con el lema anterior, esto puede lograrse en tiempo polinómico.

M_2 simula las acciones de M con la siguiente modificación: después de ejecutar cada uno de los pasos de M, M_2 hace avanzar la cabeza de la cinta 2 una celda hacia la derecha. Si esta simulación llega al estado de parada de M, entonces M_2 se detiene con su primera cinta configurada como $\underline{\Delta}N\Delta\Delta\Delta\cdots$. Sin embargo, si llega a un espacio en blanco en la segunda cinta, se detiene con la configuración $\underline{\Delta}Y\Delta\Delta\Delta\cdots$. en la primera cinta. Observe que M_2 puede efectuar este proceso en tiempo polinómico, ya que al recibir la entrada w no simula más de $p(|w|)$ pasos de M, no borra más de $p(|w|)$ celdas de su primera cinta y sólo tiene que escribir Y o N antes de detenerse.

En resumen, M' simula un máximo de $p(|w|)$ pasos de M al recibir la entrada w. Si $w \in L$, entonces M habría aceptado w en ese tiempo, por lo que M' se detendrá con una respuesta afirmativa; de lo contrario, M' se detendrá con una respuesta negativa.

∎

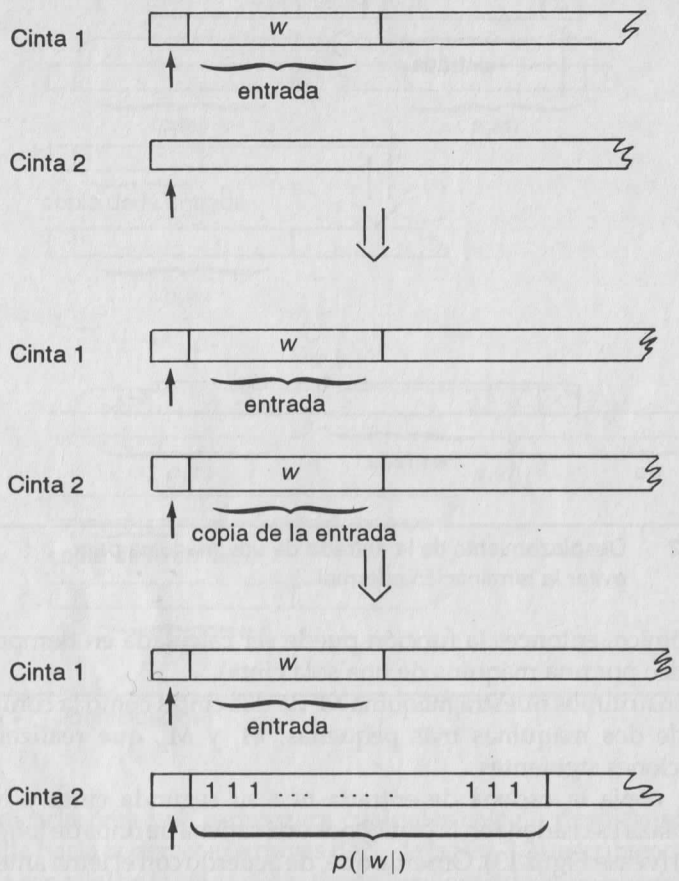


Figura 5.13 Cálculos realizados por M_1

Una consecuencia inmediata del teorema 5.2 es que el lenguaje L_1 de la sección 3.5 no se halla en P. Recuerde que se mostró que este lenguaje era aceptado por máquinas de Turing, pero no decidible por máquinas de Turing. Por esto, el teorema 5.2 dice que una máquina de Turing no puede aceptar L_1 en tiempo polinómico.

Problemas de decisión

Otra consecuencia del teorema 5.2 es que nos permite ampliar nuestro concepto de la clase P. Según la definición, P es una colección de lenguajes pero, por el teorema 5.2, se le considera como una clase de problemas de decisión. Expliquemos esto.

Un problema de decisión es aquel problema que puede expresarse en forma de una pregunta cuya respuesta es sí o no. El problema de comparación de cadenas de la sección 5.2 es un ejemplo, ya que la pregunta es si las cadenas son idénticas. Al codificarse como la entrada de una máquina de Turing, cada caso de este tipo de problemas se representa como una colección de cadenas de símbolos. Así mismo, aceptando una regla convencional según la cual las cadenas puedan concatenarse para formar una de gran tamaño (quizás usando marcas de separación), podemos considerar que todos los problemas de decisión tienen como entrada cadenas individuales de un alfabeto. En este contexto, la colección de las cadenas para las cuales la respuesta a un problema de decisión es "sí" constituye un lenguaje. Además, la tarea de decidir este lenguaje es igual que aquella para resolver el problema de decisión (la decisión del lenguaje $\{w^*w: w \in \{x, y, z\}^*\}$ es lo mismo que resolver el problema de comparación de cadenas).

Entonces, resolver problemas de decisión es lo mismo que decidir lenguajes; desde esta perspectiva, el teorema 5.2 nos dice que la clase P corresponde a la clase de los problemas de decisión que pueden resolverse con máquinas de Turing en tiempo polinómico. Es más, esta correspondencia es tan estrecha que con frecuencia pasamos, sin miramientos, de considerar a P como una clase de lenguajes a considerarla como una clase de problemas de decisión.

En resumen, hemos establecido las siguientes caracterizaciones equivalentes de la clase P:

1. La clase de los lenguajes que pueden aceptar las máquinas de Turing en tiempo polinómico.
2. La clase de los lenguajes que pueden decidir las máquinas de Turing en tiempo polinómico.
3. La clase de los problemas de decisión que pueden resolver las máquinas de Turing en tiempo polinómico.

Así mismo, cada una de estas caracterizaciones puede establecerse en términos de máquinas de Turing de varias cintas.

Ejercicios

1. Suponga que A es un problema que se puede resolver con un algoritmo de complejidad temporal n^2 y que el problema B requiere un algoritmo con complejidad temporal 2^n. Suponga además que la tecnología actual permite resolver en una hora los casos del problema A con entradas de longitud 100. ¿Cuál sería el tamaño de los casos del problema A que podrían resolverse en una hora si la tecnología produjera una máquina que fuese 100 veces más rápida que las actuales? Responda a las mismas preguntas para el problema B. ¿Qué indican sus respuestas con respecto a la viabilidad de resolver problemas con soluciones en tiempo polinómico, en comparación con los que sólo tienen soluciones en tiempo exponencial?

2. Muestre que la complejidad temporal de la aceptación de cualquier lenguaje regular con una máquina de Turing es $\theta(n)$, donde n es la longitud de la cadena de entrada, y que por lo tanto todo lenguaje regular está en P.

3. Muestre que no se encuentra en P un lenguaje aceptado por máquinas de Turing que no puede ser aceptado por una máquina de Turing en menos de 2^n pasos, donde n es la longitud de la cadena de entrada.

4. Presente los siguientes problemas de decisión como problemas de reconocimiento de lenguajes.
   a. Dadas dos entradas $m, n \in \mathbb{N}$, decidir si existe un número primo entre m y n.
   b. Dadas dos cadenas de símbolos de entrada, decidir si una es permutación de la otra.
   c. Dados un entero positivo y una lista finita de enteros positivos, decidir si existe un subconjunto de la lista cuya suma sea el primer entero.

5.5 COMPLEJIDAD TEMPORAL DE MÁQUINAS NO DETERMINISTAS

En el capítulo 3 vimos que las máquinas de Turing tradicionales (de una sola cinta), las de varias cintas y las no deterministas poseen el mismo poder de reconocimiento de lenguajes en lo que se refiere a que un lenguaje aceptado por una máquina de una de las clases puede ser aceptado por una máquina de cualquier otra clase. Vimos además que esta igualdad de poder se conserva entre las máquinas de Turing de una y varias cintas cuando se restringen a cálculos en tiempo polinómico. Por esto, es natural preguntarnos cuál es el efecto que tiene la restricción de tiempo polinómico sobre las máquinas de Turing no deterministas; esta pregunta es la que ahora nos concierne.

La clase *NP*

Decimos que una máquina de Turing no determinista M acepta el lenguaje L en tiempo polinómico si $L = L(M)$ y existe un polinomio $p(x)$ tal que para cualquier $w \in L$, M acepta w con una serie de cálculos que no excede de $p(|w|)$ pasos. Así mismo, definimos NP como la clase de los lenguajes que pueden aceptar las máquinas de Turing no deterministas en tiempo polinómico.

Puesto que toda máquina de Turing determinista está contenida en la clase de las máquinas de Turing no deterministas, podemos afirmar de inmediato que $P \subseteq NP$. Sin embargo, la cuestión de si $P = NP$ aún no se ha resuelto de hecho, quizás se trate del problema de investigación más importante en las ciencias de la computación actualmente. Existen numerosos problemas de

decisión (algunos de los cuales se presentan en el apéndice E) que pueden plantearse en función del reconocimiento de lenguajes que se sabe están en NP pero cuya pertenencia (o no pertenencia a P) no se ha podido determinar todavía. Entonces, si $P = NP$, estos problemas tendrían soluciones algorítmicas prácticas, pero si $P \neq NP$, la posibilidad de encontrar soluciones algorítmicas eficientes para estos problemas se reduciría en forma considerable.

No obstante, debemos admitir que la relación entre la resolución de problemas de decisión y la aceptación de lenguajes en NP no se ha definido tan bién como para P. Recuerde que en P, la estrecha relación entre la resolución de problemas de decisión y la aceptación de lenguajes surgió como consecuencia de que la capacidad para aceptar un lenguaje en tiempo polinómico equivale a la capacidad para decidir el lenguaje en tiempo polinómico. Los investigadores todavía no han resuelto la relación entre la aceptación y la decisión de lenguajes en tiempo polinómico en el contexto de las máquinas de Turing no deterministas (Fig. 5.14). Por esto, el hecho de que un lenguaje en NP se encuentra asociado a un problema de decisión específico no significa que sea posible resolver completamente el problema en tiempo polinómico con una máquina no determinista.

Resumamos lo anterior con un ejemplo. Considere el siguiente problema, conocido como problema de decisión del viajante, que se presenta en numerosas situaciones aunque con diferentes facetas.

> Dados un conjunto de ciudades, la distancia que separa cada par de ciudades y una distancia de viaje total permitida d, ¿existe una forma de viajar entre las ciudades de manera que se visite cada ciudad, que la ruta finalice en la ciudad de inicio y que la distancia total no exceda d?

Este problema de decisión está asociado con el problema de decidir el lenguaje L_S, el cual consiste en aquellas cadenas que representan casos con respuesta afirmativa del problema del viajante. Este lenguaje es aceptado por

Cálculos deterministas en tiempo polinómico

problemas de decisión solubles	=	lenguajes decidibles	=	lenguajes aceptables

Cálculos no deterministas en tiempo polinómico

problemas de decisión solubles	=	lenguajes decidibles	? =	lenguajes aceptables

Figura 5.14 Equivalencias entre los cálculos en tiempo polinómico

la máquina de Turing no determinista que, al recibir un caso del problema del viajante, genera una ruta entre las ciudades de manera no determinista, donde cada ruta es una salida potencial, y luego evalúa la ruta generada. Si encuentra que la ruta es suficientemente corta, la máquina se detiene; de lo contrario, entra en un ciclo infinito.

Además, esta máquina aceptaría L_S en tiempo polinómico, ya que cualquier ruta se puede generar y evaluar en un intervalo limitado por alguna expresión polinómica del número de ciudades (si existen n ciudades, entonces para generar una ruta se unen n rutas entre ciudades, y la evaluación de la ruta consiste en sumar las distancias correspondientes entre las n ciudades). Llegamos así a la conclusión de que L_S está en NP.

Sin embargo, el hecho de que L_S se halle en NP sólo significa que L_S puede ser *aceptado* en tiempo polinómico por una máquina de Turing no determinista. Esto quizá no implique que pueda ser *decidido* en tiempo polinómico. Este requisito, mucho más restrictivo, es el que se necesita para resolver el problema de decisión del viajante. Así, si sabemos que L_S se encuentra en NP, obtenemos algo de información acerca de la complejidad del problema correspondiente, pero esto no nos dice todo. Existe la posibilidad de que ninguna variante de una máquina de Turing pueda resolver por completo el problema del viajante en tiempo polinómico.

Por otra parte, si pudiéramos demostrar que $P = NP$, entonces L_S pertenecería a P; entonces, con una máquina de Turing determinista podríamos decidirlo en tiempo polinómico y así resolveríamos en la misma cantidad de tiempo el problema de decisión del viajante.

Reducciones polinómicas

El apéndice E contiene unos cuantos entre los muchos problemas que, según se ha determinado, se ubican en la clase NP pero cuya pertenencia a P aún no se ha resuelto. Por ello, mostrar que uno de estos problemas realmente pertenece a P sería un paso muy pequeño hacia la resolución de la relación entre las clases P y NP. Existe, sin embargo, una forma de atacar al mismo tiempo la clasificación de muchos problemas. Este enfoque se basa en el concepto de NP-compleción, que a su vez se apoya en el concepto de las reducciones polinómicas, que definimos a continuación.

Una **reducción polinómica** (o transformación polinómica) de un lenguaje L_1 del alfabeto Σ_1 a otro lenguaje L_2 del alfabeto Σ_2, es una función $f{:}\Sigma_1^* \to \Sigma_2^*$ que puede ser calculada por una máquina de Turing en tiempo polinómico, y para la cual $w \in L_1$ si y sólo si $f(w) \in L_2$ para toda $w \in \Sigma_1^*$. Si existe una reducción polinómica del lenguaje L_1 al lenguaje L_2, decimos que L_1 se reduce a L_2 y lo escribimos como $L_1 \propto L_2$.

Estas reducciones permiten convertir las preguntas relacionadas con la pertenencia de una cadena a un lenguaje a preguntas con respecto a la pertenencia de una cadena a otro lenguaje. Específicamente, si $f{:}\Sigma_1^* \to \Sigma_2^*$ es una reducción polinómica de L_1 a L_2 calculada por una máquina de Turing M_f, y si

M_2 es una máquina de Turing donde $L(M_2) = L_2$, entonces el lenguaje que acepta la máquina compuesta $\to M_f M_2$ es L_1. Además, el hecho de que M_f calcule f en tiempo polinómico significa que la complejidad temporal de la máquina compuesta $\to M_f M_2$ es comparable con la de M_2, afirmación que justifica el siguiente teorema.

TEOREMA 5.3

Si $L_1 \propto L_2$ y L_2 está en P, entonces L_1 está en P.

DEMOSTRACIÓN

Puesto que $L_1 \propto L_2$, existe una reducción polinómica f de L_1 a L_2 que una máquina de Turing puede calcular M_f puede calcular en tiempo polinómico. Específicamente, existe un polinomio $p(x)$ tal que M_f calcula $f(w)$ en el tiempo $p(|w|)$ para cada $w \in L_1$. Así mismo, la longitud de $f(|w|)$ no puede ser mayor que $p(|w|) + |w|$ (M_f no puede escribir más de $p(|w|)$ símbolos en $p(|w|)$ pasos; entonces la salida de M_f, al recibir la entrada w, no puede exceder de $p(|w|)$ más la longitud de la entrada original, $|w|$).

Sea ahora M_2 una máquina de Turing que acepta L_2 en tiempo polinómico. Entonces, existe un polinomio $q(x)$ tal que M_2 acepta toda v en L_2 en un máximo de $q(|v|)$ pasos. Sin embargo, la máquina compuesta $\to M_f M_2$ acepta entonces cada $w \in L_1$ en no más de $p(|w|) + q(p(|w|) + |w|)$ pasos, un polinomio en $|w|$, de acuerdo con lo requerido. Es decir, $\to M_f M_2$ acepta L_1 en tiempo polinómico, por lo que L_1 está en P.

∎

Como ejemplo, la función $f{:}\{x, y\}^* \to \{x, y, z\}^*$ definida por $f(v) = vzv$ es una reducción polinómica del lenguaje

$$L_1 = \{w{:}\ w \text{ es un palíndromo en } \{x, y\}^*\}$$

al lenguaje

$$L_2 = \{wzw^R{:}\ w \in \{x, y\}^* \text{ y } w^R \text{ es la cadena } w \text{ escrita a la inversa}\}$$

Es fácil confirmar que $v \in L_1$ si y sólo si $f(v) \in L_2$. Además, f se puede calcular en tiempo polinómico aplicando la rutina de copiado de la figura 3.8 seguida por la escritura de una z en el espacio en blanco entre las dos partes. Por consiguiente, si tuviéramos una máquina de Turing que aceptara L_2 en tiempo polinómico, la podríamos usar en conjunción con esta reducción polinómica para construir una máquina que aceptara L_1 en tiempo polinómico.

Una de las principales aplicaciones del teorema 5.3 es como herramienta para mostrar que ciertos lenguajes se encuentran en P. Si podemos mostrar la

existencia de una reducción polinómica de un lenguaje L_1 a otro lenguaje L_2, entonces podemos demostrar que ambos lenguajes están en P con sólo mostrar que L_2 se halla en P.

Además, si replanteamos el teorema 5.3 como "si $L_1 \propto L_2$ y L_1 no está en P, entonces L_2 no está en P", obtenemos una herramienta para mostrar que ciertos lenguajes no se encuentran en P. O sea, si demostramos la existencia de una reducción polinómica de L_1 a L_2, entonces se puede mostrar que ambos lenguajes están fuera de P si mostramos que L_1 no se halla en P.

Concluimos entonces que las reducciones polinómicas ofrecen un medio para atacar la clasificación de más de un problema a la vez.

Teorema de Cook

El teorema que estamos a punto de demostrar, conocido como teorema de Cook en honor de su descubridor, S. A. Cook, permite obtener el mayor partido posible del teorema 5.3. Identifica un lenguaje (un problema de decisión) específico en la clase NP al que cualquier otro lenguaje en NP se puede reducir por reducción polinómica. Así, si alguna vez se muestra que este lenguaje se halla en P, todos los lenguajes en NP deben pertenecer a P. Sin embargo, antes de pasar al enunciado y a la demostración del teorema de Cook, debemos presentar cierta terminología adicional.

Sea $V = \{v_1, v_2, \cdots, v_n\}$ un conjunto finito de variables booleanas (llamadas proposiciones por los especialistas en lógica), y definamos una **asignación de verdad** a V como una función de V en el conjunto {verdadero, falso}. Así mismo, representemos con $\bar{v}_i$ a la negación de la variable v_i. De este modo, si una asignación de verdad asigna a v_i el valor verdadero, entonces $\bar{v}_i$ será falso, y viceversa. Las variables y las negaciones de variables se denominan **literales**.

Definimos que una **cláusula** de V es un conjunto no vacío de literales asociados con V. Si una cláusula contiene dos o más literales, los visualizamos como si estuvieran separados por la palabra "o", como "v_1 o $\bar{v}_3$ o v_4". Se dice que una cláusula es **satisfecha** por una asignación de verdad si por lo menos uno de sus literales es verdadero bajo dicha asignación de verdad (la asignación de verdad que asigna v_1 como verdadero y v_2 como falso satisface las cláusulas "v_1 o v_2" y "$\bar{v}_4$ o v_2 o $\bar{v}_2$," pero no la cláusula "$\bar{v}_i$ o v_2").

Ahora podemos presentar el problema de decisión asociado con el teorema de Cook, de la manera siguiente:

Dado un conjunto finito de variables V y una colección de cláusulas con respecto a V, ¿existe una asignación de verdad que satisfaga las cláusulas?

Este problema se conoce como problema de la **satisfactibilidad** o **SAT**, en forma breve.

Cualquier caso de SAT se puede codificar como una sola cadena, de la forma siguiente. Si $\{v_1, v_2, \cdots, v_m\}$ es el conjunto de variables, denote cada literal

con una cadena de longitud m de acuerdo con el siguiente esquema: cada literal v_i está representado por una cadena que sólo contiene ceros, excepto por una p (forma abreviada de positivo) en la posición i; cada literal $\bar{v}_i$ está representado por una cadena que sólo contiene ceros excepto por una n (forma abreviada de negativo) en la posición i. Luego, cada cláusula se representa con una lista de sus literales, separados por diagonales. Por último, todo el caso de SAT se representa con una lista de estas cláusulas, en la cual cada cláusula se encierra entre paréntesis. El caso de SAT que contiene las variables v_1, v_2 y v_3 y las cláusulas "v_1 o $\bar{v}_2$", "v_2 o v_3" y "$\bar{v}_1$ o $\bar{v}_3$ o v_2" se representaría con la cadena

$$(p00/0n0)(0p0/00p)(n00/00n/0p0)$$

Denotamos con L_{SAT} al lenguaje consistente en aquellas cadenas que representan casos de SAT que satisfacen alguna asignación de verdad. Así, $(p00/0n0)(0p0/00p)(n00/00n/0p0)$ se encuentra en L_{SAT} porque el caso correspondiente de SAT se puede satisfacer con la asignación de verdad que asigna falso a v_1 y v_2, y verdadero a v_3. La cadena $(0p0/00p)(00n/0p0/n00)(p00/0n0)$ también se halla en L_{SAT}, pues representa el mismo caso de SAT que la cadena anterior. Sin embargo, la cadena $(p00/0p0)(n00/0p0)(p00/0n0)(n00/0n0)$ no está en L_{SAT} pues no existe una asignación de verdad que satisfaga "v_1 o v_2", "$\bar{v}_1$ o v_2" y "$\bar{v}_1$ o $\bar{v}_2$" simultáneamente.

Representemos ahora las asignaciones de verdad con cadenas de varias p y varias n donde una p en la posición i indica que la variable v_i es verdadera y una n en dicha posición indica que v_i es falsa. Así, es bastante sencillo evaluar si una asignación de verdad satisface un caso codificado de SAT: lo único que tenemos que hacer es rastrear de izquierda a derecha la cadena que representa al caso del problema a la vez que revisamos cada cláusula para ver si contiene un literal donde la p o la n esté en una posición relativa igual a la que tiene en la asignación de verdad (véase Fig. 5.15).

Con base en esta técnica, podemos construir una máquina de Turing no determinista que acepte L_{SAT} en tiempo polinómico. Observe que una variante del teorema 3.1 nos indica que basta con una máquina de varias cintas. Específicamente, podríamos construir una máquina de Turing no determinista con dos cintas que 1) iniciara cada cálculo comprobando que su entrada fuera una cadena que representa una colección de cláusulas; 2) escribiera en la segunda cinta, de manera no determinista, una cadena de p y n de longitud igual al número de variables en las cláusulas, y 3) recorriera la cabeza de la cinta por la cadena de entrada a la vez que evaluara si la asignación de verdad de la segunda cinta satisface las cláusulas que se rastrean (Fig. 5.16).

Es fácil implantar este proceso de manera que la complejidad temporal de los cálculos de aceptación esté limitada por un polinomio de la longitud de la cadena de entrada. Concluimos que L_{SAT} está en NP.

Ahora estamos listos para enunciar y demostrar el teorema de Cook.

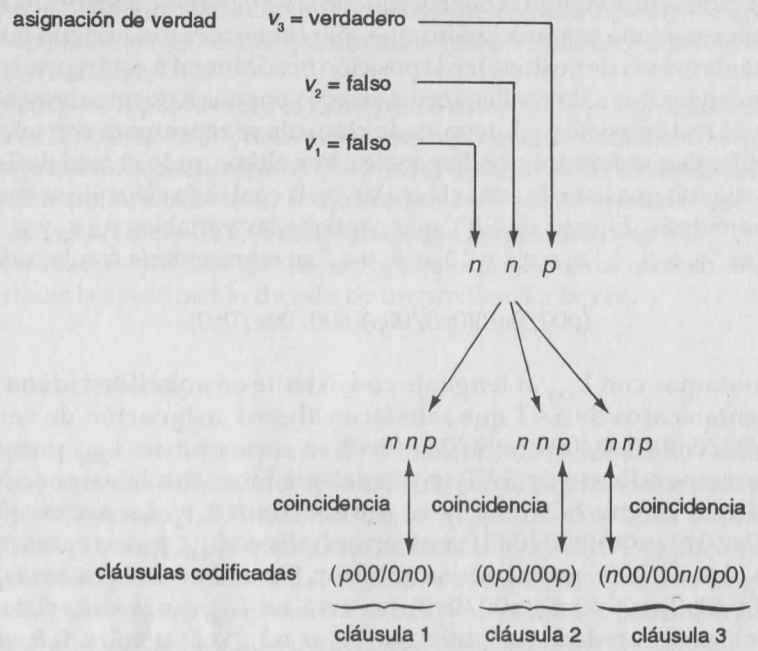


Figura 5.15 Confirmación de que una asignación de verdad satisface una colección de cláusulas

TEOREMA 5.4
Si L es cualquier lenguaje en NP, entonces $L \propto L_{SAT}$.

DEMOSTRACIÓN
Puesto que L está en NP, entonces existen una máquina de Turing M no determinista y un polinomio $p(x)$ tales que para cada $w \in L$, M acepta w en un máximo de $p(|w|)$ pasos. Además, podemos suponer que $|w| < p(|w|)$ para toda $w \in L$ y, como en el teorema 5.2, suponer que M nunca termina anormalmente.

Construimos una reducción polinómica f de L a L_{SAT} basada en la estructura de M. De modo más preciso, para cada $w \in L$, $f(w)$ será una colección de cláusulas que pueden satisfacerse simultáneamente si y sólo si existe un cálculo por medio del cual M acepte w. Se diseñarán las cláusulas en $f(w)$ para que representen los cuatro enunciados siguientes:

1. En cualquier instante de los cálculos, la máquina M se halla en uno y sólo uno de los estados, la cabeza de la cinta se encuentra sobre una y sólo una de las celdas y cada celda contiene uno y sólo un símbolo.

Configuración inicial	Cinta 1: $\Delta(p00/0n0)(0p0/00p)(n00/00n/0p0)\Delta\Delta\ldots$
	Cinta 2: $\Delta\Delta\Delta\ldots$
Asignación de verdad se ha escrito en la cinta 2	Cinta 1: $\Delta(p00/0n0)(0p0/00p)(n00/00n/0p0)\Delta\Delta\ldots$
	Cinta 2: $\Delta nnp\Delta\Delta\ldots$
Se encuentra que se satisface la primera cláusula	Cinta 1: $\Delta(p00/0n0)(0p0/00p)(n00/00n/0p0)\Delta\Delta\ldots$
	Cinta 2: $\Delta nnp\Delta\Delta\ldots$
Se encuentra que se satisface la segunda cláusula	Cinta 1: $\Delta(p00/0n0)(0p0/00p)(n00/00n/0p0)\Delta\Delta\ldots$
	Cinta 2: $\Delta nnp\Delta\Delta\ldots$
Se encuentra que se satisface la tercera cláusula	Cinta 1: $\Delta(p00/0n0)(0p0/00p)(n00/00n/0p0)\Delta\Delta$
	Cinta 2: $\Delta nnp\Delta\Delta\ldots$
La máquina se detiene al encontrar que se satisfacen todas las cláusulas	Cinta 1: $\Delta(p00/0n0)(0p0/00p)(n00/00n/0p0)\Delta\Delta\ldots$
	Cinta 2: $\Delta nnp\Delta\Delta\ldots$

Figura 5.16 Evaluación de la satisfactibilidad mediante una máquina de Turing no determinista con dos cintas

2. La máquina comienza sus cálculos a partir de su estado inicial, con la cabeza sobre la celda del extremo izquierdo de la cinta y con la cinta conteniendo un espacio en blanco, seguido por la cadena de entrada w y después espacios en blancos.
3. En cada paso de los cálculos, la posición de la cabeza, el estado actual y el contenido de la cinta cambian de acuerdo con el programa de la máquina.
4. Los cálculos llegan al estado de parada de la máquina.

Es evidente que un conjunto de cláusulas que representan estos enunciados se puede satisfacer simultáneamente si y sólo si w es una cadena en $L(M)$. Así, para definir $f(w)$, sólo tenemos que traducir estos enunciados a cláusulas.

Comenzamos este proceso de traducción representando los estados de M como $q_1, q_2, \cdots, q_r$, donde q_1 y q_2 son los estados de inicio y parada respectivamente. También establecemos que se usará x_1 para

representar el símbolo del espacio en blanco y $x_2, x_3, \cdots, x_m$ para los símbolos de la cinta de M que no sean espacios en blanco.

En la figura 5.17 se identifican las variables que comprende $f(w)$. Como se indica allí, cada variable representa un enunciado acerca de los cálculos que ejecuta M al recibir la entrada w. Estos enunciados se presentan en el contexto de un reloj imaginario que comienza a contar en cero y contabiliza las unidades de tiempo al mismo ritmo que la ejecución de pasos durante los cálculos de M. Después de que M realiza su primer paso, el reloj indicará 1; después del segundo paso, indicará 2, etcétera. Así, lo que en esencia hace el reloj es llevar a cabo un recuento de los pasos ejecutados por M. Decimos "en esencia" porque el reloj seguirá su marcha aunque M se haya detenido, en cuyo caso la indicación del reloj ya no corresponderá al número de pasos ejecutados. De hecho, ésta es la razón por la cual introducimos el reloj: si un cálculo se detiene después de 10 pasos, no tiene sentido referirnos a su configuración después de la ejecución de 12 pasos, pero sí podemos hablar de su configuración tras 12 movimientos del reloj (una vez que M se detiene, permanece en su configuración de detención todo el tiempo que avance el reloj).

Como M puede aceptar cualquier $w \in L(M)$ en un máximo de $p(|w|)$ pasos, la parte más importante de los cálculos de M concluirá en no más de $p(|w|)$ pasos. Así, los enunciados representados por las variables de $f(w)$ se refieren a las lecturas de reloj en el intervalo de 0 a $p(|w|)$. Así mismo, en $p(|w|)$ pasos la máquina no podrá mover su cabeza más allá de la celda $p(|w|) + 1$, donde consideramos que las celdas se nume-ran de izquierda a derecha a partir de uno. Por consiguiente, las variables en $f(w)$ sólo se refieren a las celdas 1 a $p(|w|) + 1$ de la cinta.

Ahora estamos preparados para traducir a cláusulas los enunciados 1 a 4. El enunciado 4 es el más sencillo ya que puede representarse con la cláusula de una sola variable

$$St_{p(n),2}$$

Variable	Intervalo de los subíndices	Enunciado correspondiente	Número de variables de este tipo
$Hd_{i,j}$	$0 \le i \le p(n)$ $1 \le j \le p(n)+1$	La cabeza de l a cinta de M está sobre la celda j en el instante i	$[p(n)+1]^2$
$St_{i,j}$	$0 \le i \le p(n)$ $1 \le j \le r$	M se halla en el estado q_j en el instante i	$r[p(n)+1]$
$Cont_{i,j,k}$	$0 \le i \le p(n)$ $1 \le j \le p(n)+1$ $1 \le k \le m$	(La celda j contiene el símbolo x_k en el instante i	$m[p(n)+1]^2$

Figura 5.17 Variables empleadas en el caso de SAT representado por $f(w)$

Los enunciados 1 y 2 son un poco más rebuscados pero todavía son sencillos; se traducen en las figuras 5.18 y 5.19 respectivamente. Las cláusulas que representan al enunciado 1 se basan en la observación de que si x, y y z son variables, entonces la cláusula

$$x \text{ o } y \text{ o } z$$

refleja el requisito de que por lo menos una de las variables sea verdadera, y las cláusulas

$$\bar{x} \text{ or } \bar{y}$$
$$\bar{x} \text{ or } \bar{z}$$
$$\bar{y} \text{ or } \bar{z}$$

Restricción	Cláusulas	Número de cláusulas de este tipo
La cabeza de la cinta de M debe encontrarse sobre un mínimo de una celda en cualquier momento.	$Hd_{i,1}$ o $Hd_{i,2}$ o ... o $Hd_{i,p(n)+1}$ para cada i en $\{0, 1, ..., p(n)\}$	$p(n)+1$
La cabeza de la cinta de M sólo puede estar sobre una celda en cada momento.	$\overline{Hd_{i,j}}$ o $\overline{Hd_{i,k}}$ para cada i en $\{0, 1, ..., p(n)\}$ y cada j y k en $\{1, 2, ..., p(n)+1\}$ donde $j < k$	$\frac{(p(n)+1)^2\, p(n)}{2}$
M debe estar por lo menos en un estado en cualquier momento.	$ST_{i,1}$ o $ST_{i,2}$ o ... o $ST_{i,r}$ para cada i en $\{0, 1, ..., p(n)\}$	$p(n)+1$
M sólo puede hallarse en un estado en cada momento.	$\overline{St_{i,j}}$ o $\overline{St_{i,k}}$ para cada i en $\{0, 1, ..., p(n)\}$ y cada j y k en $\{1, 2, ..., r\}$ donde $j < k$	$(p(n)+1)\,\frac{r(r-1)}{2}$
Cada celda de la cinta debe contener cuando menos un símbolo en cualquier momento.	$Cont_{i,j,1}$ o $Cont_{i,j,2}$ o ... o $Cont_{i,j,m}$ para cada i en $\{0, 1, ..., p(n)\}$ y cada j en $\{1, 2, ..., p(n)+1\}$	$(p(n)+1)^2$
Cada celda de la cinta puede contener cuando mucho un símbolo en cada momento.	$\overline{Cont_{i,j,k}}$ o $\overline{Cont_{i,j,l}}$ para cada i en $\{0, 1, ..., p(n)\}$, cada j en $\{1, 2, ..., p(n)+1\}$ y cada k y l en $\{1, 2, ..., m\}$ donde $k < l$	$(p(n)+1)^2\,\frac{m(m-1)}{2}$

Figura 5.18 Cláusulas que representan al enunciado 1

$$St_{0,1}$$
$$Hd_{0,1}$$
$$Cont_{0,1,1}$$
$$Cont_{0,2,k_1}$$
$$Cont_{0,3,k_2}$$
$$\vdots$$
$$Cont_{0,n+1,k_n}$$
$$Cont_{0,n+2,1}$$
$$\vdots$$
$$Cont_{0,p(n)+1,1}$$

Figura 5.19 Cláusulas que representan al enunciado 2

contienen el requisito de que sólo una de las variables puede ser verdadera.

Consideremos ahora el enunciado 3. Lo traducimos de acuerdo con las transiciones que puede efectuar M. Recuerde que estas transiciones se pueden clasificar en tres categorías: las que mueven la cabeza hacia la derecha, las que la mueven hacia la izquierda y las que cambian el contenido de la celda actual.

Cada transición de la primera categoría requiere que la máquina se halla en un estado q_s determinado, con un símbolo x_k en su celda actual, y da como resultado un desplazamiento al nuevo estado q_t moviendo la cabeza hacia la derecha. Así, para cada una de estas transiciones, cada i en $\{0, 1, \cdots, p(n) - 1\}$ y cada j en $\{1, 2, \cdots, p(n)\}$, introducimos las tres cláusulas

$$\overline{St}_{i,s} \text{ o } \overline{Hd}_{i,j} \text{ o } \overline{Cont}_{i,j,k} \text{ o } Hd_{i+1,j+1}$$
$$\overline{St}_{i,s} \text{ o } \overline{Hd}_{i,j} \text{ o } \overline{Cont}_{i,j,k} \text{ o } St_{i+1,t}$$
$$\overline{St}_{i,s} \text{ o } \overline{Hd}_{i,j} \text{ o } \overline{Cont}_{i,j,k} \text{ o } Cont_{i+1,j,k}$$

En conjunto, estas tres cláusulas corresponden al enunciado "Si en un instante i, M se encuentra en el estado q_s con su cabeza sobre la celda j que contiene el símbolo x_k, entonces en el instante $i + 1$ la cabeza estará sobre la celda $j + 1$, la máquina se hallará en el estado q_t y el contenido de la celda j seguirá siendo x_k".

Así mismo, cada transición de la segunda categoría requiere que M se encuentre en un estado q_s con x_k como contenido de su celda actual, y produce un desplazamiento al estado q_t moviendo la cabeza hacia la izquierda. Entonces, para cada una de estas transiciones, cada i en $\{0, 1, \cdots, p(n) - 1\}$ y cada j en $\{2, 3, \cdots, p(n) + 1\}$, introducimos las tres cláusulas

$$\overline{St}_{i,s} \text{ o } \overline{Hd}_{i,j} \text{ o } \overline{Cont}_{i,j,k} \text{ o } Hd_{i+1,j-1}$$
$$\overline{St}_{i,s} \text{ o } \overline{Hd}_{i,j} \text{ o } \overline{Cont}_{i,j,k} \text{ o } St_{i+1,t}$$
$$\overline{St}_{i,s} \text{ o } \overline{Hd}_{i,j} \text{ o } \overline{Cont}_{i,j,k} \text{ o } Cont_{i+1,j,k}$$

En conjunto, estas cláusulas corresponden al enunciado "Si en un instante i, M se halla en el estado q_s con su cabeza sobre la celda j que contiene el símbolo x_k, entonces en el instante $i + 1$ la cabeza estará sobre la celda $j - 1$, la máquina se encontrará en el estado q_t y el contenido de la celda j seguirá siendo x_k".

Cada transición de la tercera categoría requiere que la máquina se encuentre en un estado q_s con el símbolo x_k como contenido de la celda actual, y da como resultado un desplazamiento hacia el estado q_t escribiendo un símbolo x_l en la celda actual. Así, para cada una de estas transiciones, cada i en $\{0, 1, \cdots, p(n) - 1\}$ y cada j en $\{1, 2, \cdots, p(n)\}$, introducimos las tres cláusulas

$$\overline{St}_{i,s} \text{ o } \overline{Hd}_{i,j} \text{ o } \overline{Cont}_{i,j,k} \text{ o } Hd_{i+1,j}$$
$$\overline{St}_{i,s} \text{ o } \overline{Hd}_{i,j} \text{ o } \overline{Cont}_{i,j,k} \text{ o } St_{i+1,t}$$
$$\overline{St}_{i,s} \text{ o } \overline{Hd}_{i,j} \text{ o } \overline{Cont}_{i,j,k} \text{ o } Cont_{i+1,j,l}$$

Estas cláusulas corresponden al enunciado "Si en un instante i, M se halla en el estado q_s con su cabeza sobre la celda j que contiene el símbolo x_k, entonces en el instante $i + 1$ la cabeza estará sobre la celda j, que contendrá el símbolo x_l, y la máquina estará en el estado q_t".

Vemos entonces que cada transición de M no contribuye con más de $3(p(n) + 1)^2$ al número total de cláusulas.

Para asegurar que el contenido de la cinta sólo cambie de acuerdo con estas reglas, agregamos la colección de cláusulas de la forma

$$Hd_{i,j} \text{ o } \overline{Cont}_{i,j,k} \text{ o } Cont_{i+1,j,k}$$

para cada i en $\{0, 1, \cdots, p(n) - 1\}$, cada j en $\{1, 2, \cdots, p(n) + 1\}$ y cada k en $\{1, 2, \cdots, m\}$. Estas cláusulas representan el enunciado. "Si la cabeza de la cinta no está sobre la celda j en el instante i, entonces la celda j permanecerá sin cambios en el instante $i + 1$". Observe que sólo hay $mp(n)(p(n) + 1)$ cláusulas en esta colección.

Por último, añadimos las cláusulas de la forma

$$\overline{St}_{i,2} \text{ o } St_{i+1,2}$$

para i en $\{0, 1, \cdots, p(n) - 1\}$. Estas cláusulas nos indican que una vez que la máquina se encuentre en su estado de parada, allí permanecerá.

Si agrupamos todas las cláusulas descritas, obtenemos un conjunto de cláusulas satisfechas únicamente por aquellas asignaciones de

verdad que corresponden a los cálculos de M que aceptan la cadena w en tiempo $p(|w|)$. De esta forma hemos obtenido un caso de SAT que puede satisfacerse si y sólo si $w \in L$. Así mismo, la versión codificada de este caso es la cadena $f(w)$ que deseamos, ya que, de acuerdo con esta definición, $w \in L$ si y sólo si $f(w) \in L_{SAT}$.

Falta mostrar que $f(w)$ se puede calcular en tiempo polinómico. Para esto señalamos que el cálculo de $f(w)$ es esencialmente el proceso de generación de una lista de las cláusulas representadas por $f(w)$. Sin embargo, el número de cláusulas que aparecerán en la lista, el número de literales en cada cláusula y las longitudes de las cadenas que representa cada literal se encuentran limitados por un polinomio de $|w|$. Por lo tanto, la cadena $f(w)$ se puede calcular en tiempo polinómico.

∎

Desde el descubrimiento del teorema de Cook, se han encontrado muchos otros lenguajes en NP que poseen las mismas propiedades fundamentales que L_{SAT}. En otras palabras, existe actualmente un gran número de lenguajes en NP (incluyendo aquellos identificados en el apéndice E) que son reducciones polinómicas de cualquier otro lenguaje en NP. Estos lenguajes se conocen como **NP - completos.**

La clasificación de los lenguajes NP-completos representa un importante aspecto de las investigaciones actuales. Si alguno de estos lenguajes pudiera ser aceptado por una máquina de Turing determinista en tiempo polinómico, entonces NP debería ser igual a P y se habría mostrado que muchos problemas que al parecer consumen tanto tiempo que han de considerarse como intratables, pueden ser tratables, después de todo. Al contrario, si se llegase a demostrar que algún lenguaje cualquiera de NP yace fuera de P, entonces todos los lenguajes NP-completos también deberían estar fuera de P, y sabríamos que sería inútil seguir buscando soluciones eficientes para estos problemas. Por desgracia, los investigadores aún no han podido determinar cuál de estas opciones es verdadera.

Ejercicios

1. Muestre que el siguiente problema de decisión está en NP.

   Dado un entero de prueba y un conjunto finito de enteros de $\mathbb{N}$, ¿existe un subconjunto cuya suma sea igual al valor de prueba?

2. Encuentre una reducción polinómica del lenguaje de $\{x, y\}$ que consiste en las cadenas que contienen por lo menos dos x, al lenguaje de $\{y, z\}$ que consiste en las cadenas que contienen por lo menos tres z.

3. Con base en las definiciones de esta sección, identifique cuáles de las siguientes cadenas están en L_{SAT}.
   a. $(p00/0n0/00p)(0p0/00n)(n00/0p0/00n)$
   b. $(0n00/00n0)(n000/00n0)(n000/000n)(0n00/000n)$
   c. $(p00)(0p0)(00p)(n00/0n0)(0n0/00n)(n00/00n)$
   d. $(p0/0p)(n0/0p)(p0/0n)(n0/0n)$

4. Es común utilizar el nombre co-NP para representar la clase de los lenguajes cuyos complementos están en NP. Muestre que si co-$NP \neq NP$ entonces $P \neq NP$.

5.6 COMENTARIOS FINALES

Es conveniente dar fin a este último capítulo con un repaso de todo el texto. Comenzamos presentando los autómatas finitos y los lenguajes regulares en el contexto del análisis léxico. A partir de allí, nuestro objetivo de desarrollar técnicas más poderosas para el reconocimiento de lenguajes nos llevó al estudio de los autómatas de pila y los lenguajes independientes del contexto. En la búsqueda de técnicas aun más poderosas, pasamos a las máquinas de Turing y los lenguajes generales estructurados por frases, donde nos enfrentamos a las fronteras aparentes de los procesos algorítmicos; llegamos al límite identificado por la tesis de Church-Turing.

Una vez conocida esta tesis, nos centramos en el poder de los procesos computacionales, tratando de aprender más acerca de las limitaciones que habíamos detectado. Aquí ampliamos nuestro uso de los autómatas (en particular las máquinas de Turing), que dejaron de ser simples dispositivos de reconocimiento de lenguajes para convertirse en máquinas de propósito más general que calculaban funciones. Identificamos a la clase de las funciones que pueden calcular estas máquinas como funciones calculables según Turing y encontramos que esta clase era idéntica a la clase de las funciones recursivas parciales y también a las funciones calculables con nuestro sencillo lenguaje de programación esencial. Esta identificación de clases dio considerable credibilidad a la tesis de Church-Turing pues mostró que las limitaciones descubiertas en otras disciplinas son equivalentes.

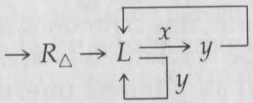
Después de investigar la frontera entre la calculababilidad y la no calculabilidad, restringimos nuestra atención a los problemas que en teoría pueden resolverse por medio de algoritmos, con el objetivo de clasificar la viabilidad de los cálculos implicados. Aquí detectamos dificultades considerables, porque el gran número de posibles soluciones para un problema puede dificultar en gran medida la identificación de la solución más eficiente. De hecho, existen problemas cuyas diversas soluciones forman una cadena infinita de algoritmos cada vez más eficientes; por consiguiente,

cualquier esquema general para clasificar problemas de acuerdo con la complejidad de su "mejor" solución no puede ser muy detallado.

Esto nos llevó a considerar la clase P, consistente en aquellos lenguajes que pueden ser aceptados (o aquellos problemas de decisión que pueden resolverse) en tiempo polinómico por una máquina de Turing. Intuitivamente, consideramos que se trataba de la clase de los problemas que tienen soluciones computacionales prácticas (con entradas de gran tamaño, la solución para los problemas que caen fuera de esta clase requiere mucho más tiempo que para aquellos que pertenecen a P). Por desgracia, los esfuerzos de los investigadores para clasificar los problemas han sido vanos, incluso en términos tan generales. Uno de los principales obstáculos es la clase NP, consistente en los lenguajes que pueden aceptar las máquinas de Turing no deterministas en tiempo polinomial. Una de las preguntas más relevantes que se plantean actualmente en las ciencias de la computación es si las clases P y NP son idénticas, ya que al parecer su solución desempeña un papel importante para determinar cuáles son las aplicaciones donde pueden emplearse los computadores.

Problemas de repaso del capítulo

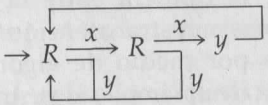
1. Determine las complejidades espaciales y temporales del proceso computacional efectuado por la máquina de Turing



al procesar las cadenas $xyxxy$, xxx y yyy.

2. Diseñe una máquina de Turing que acepte únicamente las cadenas que contienen una o más x seguidas por una o más y. ¿Cuál es la complejidad temporal de su solución? ¿Con qué precisión puede determinarse la complejidad temporal de este problema de reconocimiento de lenguajes?

3. Encuentre la complejidad temporal (peor caso) de la máquina de Turing descrita a continuación, suponiendo que las entradas se toman de $\{x, y\}^*$.



4. Encuentre la complejidad temporal del caso promedio de la máquina de Turing del problema anterior, suponiendo que las entradas se toman de $\{x, y\}^*$.

5. Muestre que una máquina de Turing no determinista de varias cintas con complejidad temporal $\theta(n)$ puede aceptar el lenguaje del alfabeto Σ que consiste en las cadenas de la forma ww^R, donde $w \in \Sigma^*$ y w^R es la cadena w escrita a la inversa.

6. Muestre que $\theta(n^d) < \theta(n!)$ para cualquier $d \in \mathbb{N}$.

7. Muestre que $\theta(log_a n) = \theta(log_b n)$.

8. Muestre que $\bigcup_{d \in \mathbb{N}^+} O(n^d) \neq \bigcup_{d \in \mathbb{N}^+} \theta(n^d)$.

9. ¿Es el lenguaje L_{SAT}, descrito en la sección 5.5, independiente del contexto? Proporcione bases para su respuesta.

10. Encuentre una reducción polinómica del lenguaje de $\{z, y\}$ que consiste en aquellas cadenas con por lo menos tres z, al lenguaje de $\{x, y\}$ que consiste en las cadenas con por lo menos dos x.

11. Muestre que si L_1 y L_2 son lenguajes en P (distintos de $\varnothing$ y Σ^*), entonces $L_1 \propto L_2$ y $L_2 \propto L_1$.

12. Muestre que el lenguaje $\varnothing$ está en P y que, para cualquier alfabeto Σ, el lenguaje Σ^* también lo está.

13. Suponga que x es un símbolo del alfabeto Σ. Muestre que $\varnothing \propto \Sigma^* - \{x\}$ pero que no existe reducción polinómica de $\Sigma^* - \{x\}$ a $\varnothing$.

14. Muestre que la complejidad temporal de la aceptación del lenguaje que consiste en cadenas de la forma ww^R, donde $w \in \{x, y\}^*$ y w^R es w en orden inverso, es $\theta(n^2)$ al usar una máquina de Turing de una sola cinta.

15. Suponga que $T: \mathbb{N} \to \mathbb{N}$ es una función total computable. Muestre que existe un lenguaje aceptable según Turing que no puede ser aceptado por ninguna máquina de Turing con una complejidad temporal en $O(T)$.

16. Suponga que $T: \mathbb{N} \to \mathbb{N}$ es una función total computable. Muestre que existe un lenguaje aceptable según Turing que no puede ser aceptado por ninguna máquina de Turing con una complejidad espacial en $O(T)$.

17. ¿Cuáles de las siguientes funciones están en $O(n^3)$? ¿Cuáles están en $\theta(n^3)$?

    a. $3n^2 + 2n + 1$ b. $n!$

    c. $\lfloor log_2 n \rfloor$ d. $5n^3 + 2n^2$

18. Encuentre enteros c_0 y n_0 tales que $4n^2 + 3n < c_0(2n^2 + n)$ para todos los enteros n mayores que n_0.

19. Muestre que $\theta(2^n) < \theta(2^{2n})$.

20. Muestre que si los lenguajes L_1 y L_2 están en P, entonces $L_1 \cup L_2$, $L_1 \circ L_2$, y L_1^* también se encuentran en P.

21. Muestre que la intersección de dos lenguajes cualesquiera en NP siempre es un lenguaje en NP.

22. Encuentre una reducción polinómica $f:\{x, y, z\}^* \rightarrow \{x, y\}^*$ del lenguaje $\{wzw^R: w \in \{x, y\}^*$ y w^R es la cadena w en orden inverso$\}$ al lenguaje $\{w:w$ es un palíndromo en $\{x, y\}^*\}$.

23. Encuentre una reducción polinómica del lenguaje $\{x, y\}^*$ al lenguaje consistente en las cadenas que contienen un número par de x.

24. Encuentre una reducción polinómica $f:\{x, y\}^* \rightarrow \{x, y\}^*$ del lenguaje de las cadenas que contienen un número par de y al lenguaje de las cadenas que contienen un número impar de x.

25 Muestre que el lenguaje consistente en todos los palíndromos en $\{x, y\}^*$ está en P.

26. Plantee los siguiente problemas de decisión como problemas de reconocimiento de lenguajes.
    a. Decidir si el nombre Carol se encuentra en una lista determinada.
    b. Decidir si una colección de enteros cuya suma sea 100 puede escogerse de una lista dada de enteros.
    c. Decidir si se puede construir un sistema de computación completo a partir de los elementos de una lista de componentes de computador.

27. ¿Existe un problema de reconocimiento de lenguaje cuya solución requiera tiempo exponencial al emplear una máquina de Turing con una sola cinta, pero que puede resolverse en tiempo polinómico con una máquina de varias cintas? Explique su respuesta.

28. Proporcione un ejemplo de un lenguaje que
    a. esté en P
    b. no esté en P
    c. esté en NP y posiblemente no esté en P

29. Diseñe una máquina de Turing no determinista (de una cinta) que acepte L_{SAT} en tiempo polinómico.

30. Muestre que si existe un lenguaje NP-completo que se encuentre en co-NP, entonces NP = co-NP.

31. Muestre que si L se halla en NP, entonces L es decidible por una máquina de Turing.

32. ¿Están en P todos los lenguajes independientes del contexto? Justifique su respuesta.

Problemas de programación

1. Escriba un programa para aceptar el lenguaje L_{SAT}. ¿Por qué esperaría que su programa consumiera mucho tiempo con entradas de gran magnitud?

2. Escriba un programa para resolver el problema de decisión del viajante.

APÉNDICE A

Más acerca de la construcción de tablas de análisis sintáctico *LR*(1)

En el capítulo 2 vimos cómo se basa una tabla de análisis sintáctico *LR*(1) en un autómata finito construido a partir de una gramática independiente del contexto, pero no analizamos los detalles de esta construcción. Nuestro objetivo actual es proporcionar una explicación de este proceso, y la estrategia será mostrar cómo se construyó el diagrama de transiciones de la figura 2.37 a partir de la gramática

$$
\begin{aligned}
S &\rightarrow zMNz \\
M &\rightarrow aMa \\
M &\rightarrow z \\
N &\rightarrow bNb \\
N &\rightarrow z
\end{aligned}
$$

con símbolo inicial S. Si se aplican los mismos pasos, es posible construir autómatas adecuados a partir de otras gramáticas independientes del contexto (que no sean ambiguas) donde el lado derecho de las reglas no consista en la cadena vacía.

El primer paso es introducir un nuevo símbolo de inicio, que representamos con S', y la nueva regla de reescritura

$$S' \rightarrow S$$

Observe que estos cambios no afectan al lenguaje generado por la gramática; sin embargo, sí aseguran que el símbolo de inicio aparezca en una sola y muy sencilla regla de reescritura.

Después introducimos la marca ^ para indicar el estado del proceso de análisis sintáctico. Por ejemplo, con esta marca escribiríamos

$$S \rightarrow z \wedge MNz$$

para indicar el estado de haber encontrado la z inicial en el patrón S y así estar listos para buscar el patrón MNz restante. La utilización de esta marca nos permite resumir las etapas del reconocimiento del patrón S de la manera siguiente:

$$S \rightarrow \wedge zMNz$$
$$S \rightarrow z \wedge MNz$$
$$S \rightarrow zM \wedge Nz$$
$$S \rightarrow zMN \wedge z$$
$$S \rightarrow zMNz \wedge$$

Establezcamos que una regla de reescritura marcada se encuentra en su forma inicial si la marca se ubica en el extremo izquierdo del lado derecho de la regla. Así mismo, una regla marcada está en forma terminal si su marca se encuentra en el extremo derecho del lado derecho de la regla de reescritura. De acuerdo con esto, la primera regla de la lista anterior se halla en forma inicial y la última en forma terminal.

Después tenemos que definir qué queremos decir con el cierre de un conjunto de reglas de reescritura marcadas. Formamos este cierre encontrando primero todos los no terminales que aparecen inmediatamente a la derecha de una marca de alguna de las reglas del conjunto. Luego añadimos al conjunto las formas iniciales de todas las reglas de reescritura de la gramática cuyo lado izquierdo consista en dichos no terminales. Si alguna de las reglas agregadas tiene no terminales que aparezcan inmediatamente a la derecha de una marca, añadimos también las formas iniciales de todas las reglas de reescritura para estos no terminales. Continuamos con este proceso hasta que no aparezcan nuevos no terminales inmediatamente a la derecha de alguna marca (observe que este proceso tiene que detenerse ya que existe un número finito de no terminales en la gramática). Se dice que el resultado es el cierre del conjunto original. Por ejemplo, el cierre del conjunto que contiene las reglas

$$S \rightarrow zM \wedge Nz$$

y

$$M \rightarrow a \wedge Ma$$

basadas en nuestro ejemplo de una gramática, es la colección

$$S \rightarrow zM \wedge Nz$$
$$N \rightarrow \wedge bNb$$
$$N \rightarrow \wedge z$$
$$M \rightarrow a \wedge Ma$$
$$M \rightarrow \wedge aMa$$
$$M \rightarrow \wedge z$$

Se añadieron la segunda y la tercera reglas porque N aparecía inmediatamente a la derecha de una marca, y se agregaron las dos últimas ya que M aparecería inmediatamente a la derecha de una marca.

Una vez establecidas estas definiciones, estamos listos para describir el proceso de construcción del diagrama de transiciones de la figura 2.37. El proceso es el que sigue:

1. Forme el cierre del conjunto que contiene la regla marcada del nuevo símbolo inicial $S' \rightarrow \wedge S$. Establezca este conjunto como el estado inicial del diagrama de transiciones.

2. Lleve a cabo los pasos siguientes mientras sea posible, sin ser redundante:

   a. Seleccione un símbolo s (terminal o no terminal) que aparezca inmediatamente a la derecha de la marca en una regla de algún estado A establecido.
   b. Sea X la colección de todas las reglas marcadas en A que tengan s inmediatamente a la derecha de sus marcas.
   c. Sea Y el conjunto de todas las reglas marcadas que se obtienen al mover la marca de cada regla de X a la derecha del símbolo s.
   d. Si después de calcular el cierre de Y, este cierre no es un estado entre los construidos hasta el momento, inclúyalo ahora.
   e. Dibuje un arco con etiqueta s del estado A al cierre de Y.

3. Señale como estado de aceptación del autómata cada estado que contenga al menos una regla marcada en forma terminal.

Hagamos más claro este proceso siguiendo las primeras etapas de la construcción de la figura 2.37. Comenzamos por establecer el conjunto que contiene las dos reglas marcadas

$$S' \rightarrow \wedge S \qquad \text{y} \qquad S \rightarrow \wedge zMNz$$

como el estado inicial del diagrama de transiciones (se trata del cierre del conjunto que consiste en la regla marcada $S' \rightarrow \wedge S$, como lo establece el primer paso de nuestro proceso de construcción).

De acuerdo con el paso 2, existirán dos arcos que saldrán de tal estado: uno con etiqueta S y otro con etiqueta z, ya que estos símbolos aparecen inmediatamente a la derecha de una marca en este estado. El arco z llevará al estado establecido mediante la formación del cierre del conjunto que consiste en

$$S \rightarrow z \wedge MNz$$

que es el conjunto

$$S \rightarrow z \wedge MNz$$
$$M \rightarrow \wedge aMa$$
$$M \rightarrow \wedge z$$

El arco con etiqueta S llevará al estado establecido al formar el cierre del conjunto que consiste en

$$S' \rightarrow S \wedge$$

(Observe que este estado será de aceptación en el diagrama de transiciones terminado.)

En la figura A.1 se resume nuestra construcción hasta llegar a este punto. Si continuáramos con la construcción, el estado representado por

$$S \rightarrow z \wedge MNz$$
$$M \rightarrow \wedge aMa$$
$$M \rightarrow \wedge z$$

tendría tres arcos: uno con etiqueta M, que conduce al cierre del conjunto consistente en $S \rightarrow zM \wedge Nz$; otro con etiqueta a, que lleva al cierre del conjunto consistente en $M \rightarrow a \wedge Ma$; y el tercero, con etiqueta z, que va al cierre del conjunto consistente en $M \rightarrow z \wedge$.

El diagrama terminado sería como el que aparece en la figura A.2. Compare esto con el diagrama de la figura 2.37; son iguales, excepto que en la figura 2.37 los estados se encuentran rotulados con números en vez de conjuntos de reglas de reescritura marcadas.

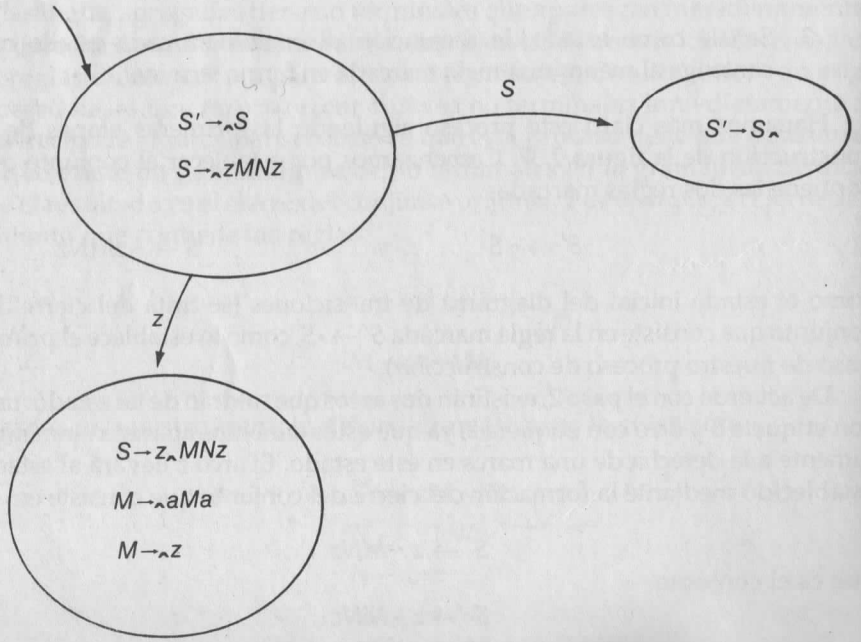


Figura A.1 Primeras etapas de la construcción de la figura 2.34

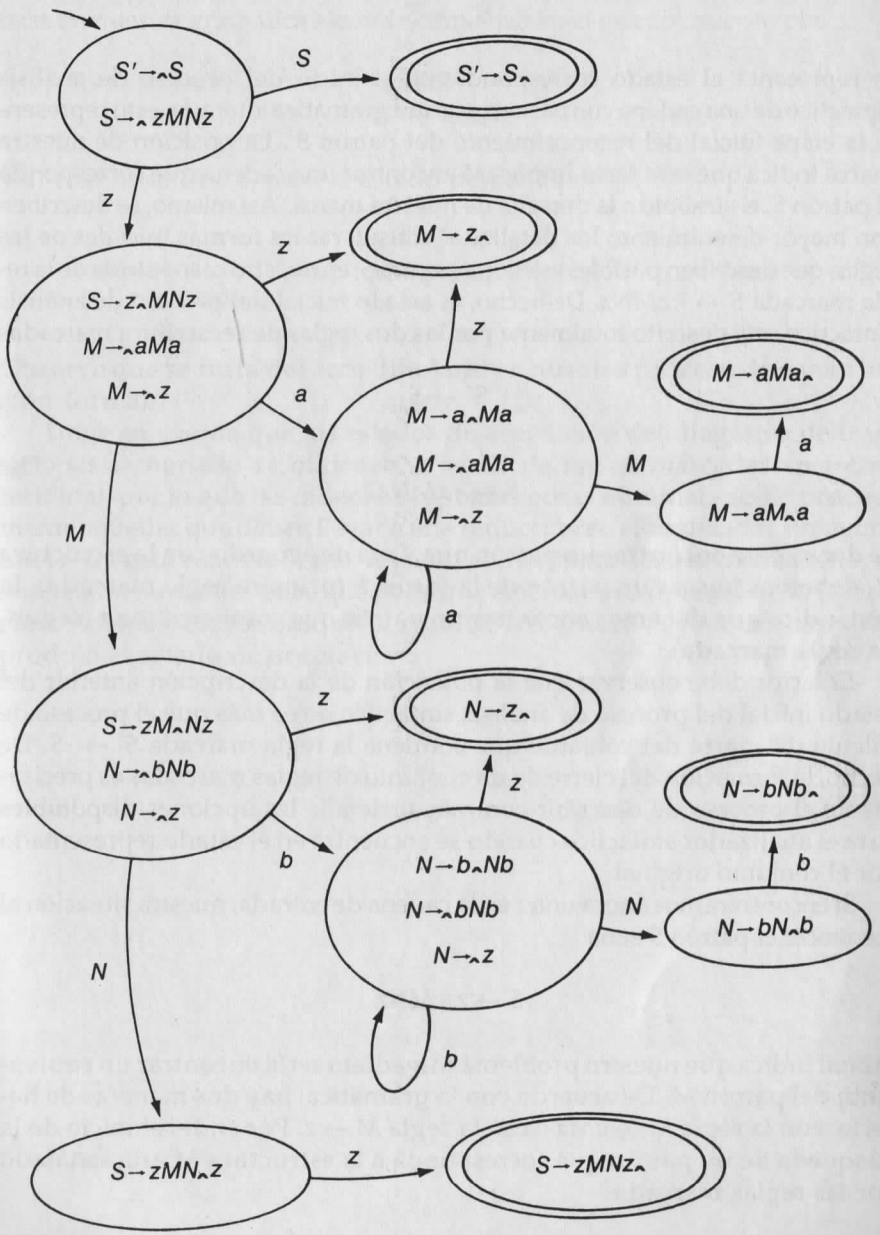


Figura A.2 Diagrama completo

Debemos concluir presentando los conceptos intuitivos que sustentan este proceso de construcción. Con

$$S' \rightarrow \wedge S$$

se representa el estado correspondiente al inicio del proceso de análisis sintáctico de una cadena con base en nuestra gramática alterada; esto representa la etapa inicial del reconocimiento del patrón S'. La posición de nuestra marca indica que esta tarea implicará encontrar una cadena que corresponda al patrón *S*, el símbolo a la derecha de nuestra marca. Así mismo, se describen con mayor detenimiento los detalles al considerar las formas iniciales de las reglas que describen posibles estructuras para *S*; en nuestro caso se trata de la regla marcada $S \rightarrow \wedge zMNz$. De hecho, el estado inicial del proceso de análisis sintáctico está descrito totalmente por las dos reglas de reescritura marcadas

$$S' \rightarrow \wedge S$$

y

$$S \rightarrow \wedge zMNz$$

Es decir, para encontrar un patrón que vaya de acuerdo con la estructura S', debemos buscar un patrón de la forma *S* (primera regla marcada), lo que indica que debemos encontrar un patrón que comience con z (segunda regla marcada).

El lector debe observar que la obtención de la descripción anterior del estado inicial del proceso de análisis sintáctico no es más que el proceso de cálculo del cierre del conjunto que contiene la regla marcada $S' \rightarrow \wedge S$. De hecho, la formación del cierre de un conjunto de reglas marcadas es precisamente el proceso de describir con mayor detalle las opciones disponibles para el analizador sintáctico cuando se encuentra en el estado representado por el conjunto original.

Si encontráramos ahora una z en la cadena de entrada, nuestra situación al reconocer el patrón *S* sería

$$S \rightarrow z \wedge MNz$$

lo cual indica que nuestro problema inmediato sería encontrar un equivalente del patrón *M*. De acuerdo con la gramática, hay dos maneras de hacerlo: con la regla $M \rightarrow aMa$ o con la regla $M \rightarrow z$. Por ende, el inicio de la búsqueda de un patrón que corresponda a la estructura *M* está señalado por las reglas marcadas

$$M \rightarrow \wedge aMa$$

y

$$M \rightarrow \wedge z$$

Una vez más, se obtiene una descripción de estado completa formando el cierre de su descripción preliminar.

Vemos entonces que el proceso de análisis sintáctico de una cadena con base en nuestra gramática alterada comienza en el estado descrito por

$$S' \rightarrow \wedge S$$
$$S \rightarrow \wedge zMNz$$

y al encontrar una z, pasa al estado descrito por

$$S \rightarrow z \wedge MNz$$
$$M \rightarrow \wedge aMa$$
$$M \rightarrow \wedge z$$

Observe que se trata del arco dibujado en nuestro proceso de construcción formal.

Tome en cuenta que los estados de aceptación del diagrama de transiciones terminado se obtienen a partir de reglas marcadas en forma terminal, por lo que las cadenas aceptadas por el autómata serán precisamente aquellas que deben llevar a una reducción en el analizador sintáctico *LR*(1). De esta manera, cada vez que el autómata acepta una cadena, el analizador sintáctico sabe que tiene que efectuar una operación de reducción. Además, ésta se basa en la regla de reescritura cuya forma terminal produjo el estado de aceptación.

APÉNDICE B

Más acerca de la función de Ackermann

El objetivo de este apéndice es demostrar que la función de Ackermann, definida por las ecuaciones

$$A(0, y) = y + 1 \qquad \text{(A1)}$$
$$A(x + 1, 0) = A(x, 1) \qquad \text{(A2)}$$
$$A(x + 1, y + 1) = A(x, A(x + 1, y)) \qquad \text{(A3)}$$

es total y calculable, pero no recursiva primitiva. El hecho de que sea total se obtiene aplicando la inducción sobre los pares de entrada en orden lexicográfico, como veremos a continuación.

Para todos los pares de la forma $(0, y)$, $A(0, y)$ está definida por la ecuación A1. Suponga ahora que $A(x, y)$ está definida para toda (x, y) menor que $(m + 1, 0)$ en orden lexicográfico. Entonces, $A(m + 1, 0) = A(m, 1)$ por la ecuación A2, de modo que $A(m + 1, 0)$ está definida. Por último, suponga que $A(x, y)$ está definida para toda (x, y) menor que $(m + 1, n + 1)$ en orden lexicográfico. Entonces, $A(m + 1, n + 1) = A(m, A(m + 1, n))$ por la ecuación A3 y, por inducción, A está definida para $(m + 1, n)$ y $(m, A(m + 1, n))$. Por consiguiente, $A(m + 1, n + 1)$ también está definida.

La computabilidad de A se desprende de nuestra demostración de que A es total. De hecho, la definición de $A(x, y)$ es la descripción de un proceso para calcular el valor de salida correspondiente.

La siguiente demostración (de que la función de Ackermann no es recursiva primitiva) es extensa, pero en realidad bastante sencilla. Mostramos que para cualquier función recursiva primitiva f que establece una correspondencia n-tuplas y tuplas de un solo componente existe un entero positivo N tal, que para cada $\bar{x}$ en $\mathbb{N}^n$,

$$f(\bar{x}) < A\,(N, \Sigma(\bar{x}))$$

donde $\Sigma(\bar{x})$ representa la suma de los componentes de $\bar{x}$ (la suma de los componentes de la tupla vacía es cero). De esto se obtiene, en forma casi inmediata, que la función de Ackermann no es recursiva primitiva. Si A fuera

recursiva primitiva, entonces la función $f = A \circ (\pi_1^1 \times \pi_1^1)$ también sería recursiva primitiva. Entonces, existiría un entero positivo N tal que $f(x) < A(N, x)$ para todo x en $\mathbb{N}$. Específicamente, al elegir $x = N$, tendríamos $f(N) < A(N, N)$; pero $f(N) = A \circ (\pi_1^1 \times \pi_1^1)(N) = A(N, N)$, de modo que llegaríamos al enunciado contradictorio $A(N, N) < A(N, N)$. Por lo tanto, debemos llegar a la conclusión de que la función de Ackermann no es recursiva primitiva.

Así, vemos que nuestro objetivo se alcanzará si demostramos la existencia de un entero positivo N, que tenga las propiedades antes descritas, para cada función recursiva primitiva. Para esto, establecemos algunas propiedades de la función de Ackermann, descritas en los lemas siguientes.

LEMA 1
Para todo z en $\mathbb{N}$,

a. $A(1, z) = z + 2$
b. $A(2, z) = 2z + 3$

DEMOSTRACIÓN
a. Aplicamos la inducción a z. Cuando $z = 0$, tenemos

$$\begin{aligned} A(1, 0) &= A(0, 1) && \text{(por la ecuación A2)} \\ &= 1 + 1 && \text{(por la ecuación A1)} \\ &= 0 + 2 \end{aligned}$$

Si la ecuación es verdadera para $z \leq n$, entonces

$$\begin{aligned} A(1, n + 1) &= A(0, A(1, n)) && \text{(por la ecuación A3)} \\ &= A(0, n + 2) && \text{(por la hipótesis de inducción)} \\ &= (n + 2) + 1 && \text{(por la ecuación A1)} \\ &= (n + 1) + 2 \end{aligned}$$

b. Una vez más, aplicamos inducción a z. Si $z = 0$, tenemos

$$\begin{aligned} A(2, 0) &= A(1, 1) && \text{(por la ecuación A2)} \\ &= A(0, A(1, 0)) && \text{(por la ecuación A3)} \\ &= A(0, A(0, 1)) && \text{(por la ecuación A2)} \\ &= A(0, 2) && \text{(por la ecuación A1)} \\ &= 3 && \text{(por la ecuación A1)} \end{aligned}$$

Suponga ahora que la igualdad es verdadera para $z \leq n$. Entonces,

$$\begin{aligned} A(2, n + 1) &= A(1, A(2, n)) && \text{(por la ecuación A3)} \\ &= A(1, 2n + 3) && \text{(por la hipótesis de inducción)} \\ &= (2n + 3) + 2 && \text{(por el lema 1a)} \\ &= 2(n + 1) + 3 \end{aligned}$$

∎

LEMA 2
Para todos x y y en $\mathbb{N}$, $y + 1 \leq A(x, y)$

DEMOSTRACIÓN
Aplicamos la inducción. Si $x = 0$, la desigualdad es verdadera para todo y, por la ecuación A1.

Suponga ahora que la desigualdad es verdadera para $x = n$. A fin de demostrar la desigualdad para $x = n + 1$, inducimos en y, comenzando por $y = 0$.

$$\begin{aligned} 0 + 1 &\leq 1 + 1 \\ &\leq A(n, 1) && \text{(por la hipótesis de inducción)} \\ &= A(n + 1, 0) && \text{(por la ecuación A2)} \end{aligned}$$

de manera que la desigualdad es verdadera para $x = n + 1$ y $y = 0$. Además, si la desigualdad es verdadera para $x = n + 1$ y $y = m$, entonces

$$\begin{aligned} (m + 1) + 1 &\leq A(n + 1, m) + 1 && \text{(por la hipótesis de inducción)} \\ &\leq A(n, A(n + 1, m)) && \text{(por la hipótesis de inducción)} \\ &= A(n + 1, m + 1) && \text{(por la ecuación A3)} \end{aligned}$$

de modo que la desigualdad es verdadera para $x = n + 1$ y $y = m + 1$.

∎

LEMA 3
La función de Ackermann satisface las siguientes desigualdades para todos x y y en $\mathbb{N}$.

a. $A(x, y) < A(x, y + 1)$
b. $A(x, y + 1) \leq A(x + 1, y)$
c. $A(x, y) < A(x + 1, y)$

DEMOSTRACIÓN
a. Aplicamos la inducción a x. Para $x = 0$, la desigualdad se desprende de A1. Si $0 < x$, entonces

$$\begin{aligned} A(x, y) &< A(x, y) + 1 \\ &\leq A(x - 1, A(x, y)) && \text{(por el lema 2)} \\ &= A(x, y + 1) && \text{(por la ecuación A3)} \end{aligned}$$

b. Aplicamos la inducción a y. Para $y = 0$, la desigualdad se desprende de la ecuación A2. Si $0 < y$ y la desigualdad es verdadera para los valores menores que y, entonces

$A(x, y + 1) \leq A(x, A(x, y))$ ($y + 1 \leq A(x, y)$ por el lema 2 y A es monotónica en su segundo argumento por el lema 3a)

$\leq A(x, A(x+1, y-1)$ (por la hipótesis de inducción y A es monotónica en su segundo argumento por el lema 3a)

$= A(x+1, y)$ (por la ecuación A3)

c. Esto es un resultado de las desigualdades precedentes, obtenido de la manera siguiente:

$A(x, y) < A(x, y+1)$ (por el lema 3a)

$\leq A(x+1, y)$ (por el lema 3b)

∎

LEMA 4

Para todos x_1, x_2, y y en $\mathbb{N}$, $A(x_1, y) + A(x_2, y) < A(máx(x_1, x_2) + 4, y)$.

DEMOSTRACIÓN

Por conveniencia, sea $m = máx(x_1, x_2)$. Entonces,

$A(x_1, y) + A(x_2, y) \leq A(m, y) + A(m, y)$ (por el lema 3c)

$= 2A(m, y)$

$< 2A(m, y) + 3$

$= A(2, A(m, y))$ (por el lema 1b)

$< A(2, A(m+3, y))$ (A es monotónica en sus dos argumentos por los lemas 3a y 3c)

$\leq A(m+2, A(m+3, y))$ (por el lema 3c)

$= A(m+3, y+1)$ (por la ecuación A3)

$\leq A(m+4, y)$ (por el lema 3b)

$= A(máx(x_1, x_2) + 4, y)$

∎

LEMA 5

Para todos x y y, $A(x, y) + y < A(x+4, y)$.

DEMOSTRACIÓN

$A(x, y) + y < A(x, y) + y + 1$

$= A(x, y) + A(0, y)$ (por la ecuación A1)

$< A(x+4, y)$ (por el lema 4)

∎

Ahora estamos listos para demostrar la proposición siguiente, la cual, como hemos visto, indica que la función de Ackermann no es recursiva primitiva.

PROPOSICIÓN

Para cualquier función recursiva primitiva f que establece una correspondencia entre tuplas de n componentes y agrupaciones de un solo componente, existe un entero positivo N tal que, para cada $\bar{x}$ en $\mathbb{N}^n$,

$$f(\bar{x}) < A(N, \Sigma(\bar{x}))$$

donde $\Sigma(\bar{x})$ representa la suma de los componentes de $\bar{x}$ (la suma de los componentes de la tupla de cero componentes es 0).

DEMOSTRACIÓN

Puesto que f es una función recursiva primitiva, debe consistir en un número finito de combinaciones, composiciones y recursividades primitivas de funciones iniciales. Podemos inducir en el número de estas operaciones. Si no se requieren dichas operaciones para obtener f, entonces f debe ser una función inicial. Si $f = \zeta$, al tomar $N = 0$ se satisfacen nuestros requisitos ya que $f(\ ) = \zeta(\ ) = 0 < 1 = A(0, 0) = A(N, 0)$. Si f es una proyección, también bastará $N = 0$ pues $f(\bar{x}) = \pi_j^n(\bar{x}) = x_j \leq \Sigma(\bar{x}) < \Sigma(\bar{x}) + 1 = A(0, \Sigma(\bar{x})) = A(N, \Sigma(\bar{x}))$. Si f es la función sucesor, entonces $N = 1$ será suficiente, ya que $f(x) = \sigma(x) = x + 1 = A(0, x) < A(1, x) = A(N, x)$, donde la desigualdad se obtiene del lema 3c.

Ahora suponga que f se construye a partir de funciones iniciales por medio de k aplicaciones de combinaciones, composiciones y recursividades primitivas (donde $0 < k$), y que la proposición es verdadera para cualquier función recursiva primitiva construida a partir de menos de k aplicaciones. Puesto que la salida de f es una tupla de un componente, la última operación de la construcción de f debe ser una composición o una recursividad primitiva. Entonces, f se puede expresar como

$$f = h \circ g$$

o como

$$f(\bar{x}, 0) = g(\bar{x})$$
$$f(\bar{x}, y+1) = h(\bar{x}, y, f(\bar{x}, y))$$

donde h y g son funciones recursivas primitivas construidas a partir de menos de k aplicaciones de combinaciones, composiciones y recursividades primitivas.

Consideremos el primero de estos casos, donde $f = h \circ g$. En general, la salida de g podría consistir en más de un componente, por lo que debemos considerar que g tiene la forma $g_1 \times g_1 \times \cdots \times g_m$ para un positivo m. Por nuestra hipótesis de inducción, existen enteros $N_1, N_2, \cdots N_m$ tales que $g_j(\bar{x}) < A(N_j, \Sigma(\bar{x}))$ para toda x en $\mathbb{N}^n$, así como un N_0 tal que $h(\bar{x}) < A(N_0, \Sigma(\bar{x}))$ para toda $\bar{x}$ en $\mathbb{N}^m$. Entonces

$$
\begin{aligned}
f(\bar{x}) &= h(g_1(\bar{x}), g_2(\bar{x}), \cdots, g_m(\bar{x}))\\
&< A(N_0, \sum_{i=1}^{m} g_i(\bar{x})) && \text{(por la elección de } N_0)\\
&< A(N_0, \sum_{i=1}^{m} A(N_i, \Sigma(\bar{x}))) && \text{(por la elección de } N_i \text{ y el lema 3a)}\\
&< A(N_0, A(M, \Sigma(\bar{x}))), \text{ donde } M = \text{máx}(N_1, \cdots, N_m) + 4(m-1) && \text{(por el lema 4)}\\
&< A(N_0, A(M+1, \Sigma(\bar{x}))) && \text{(por los lemas 3a y 3c)}\\
&< A(N_0, A(\text{máx}(N_0, M) + 1, \Sigma(\bar{x}))) && \text{(por los lemas 3a y 3c)}\\
&< A(\text{máx}(N_0, M), A(\text{máx}(N_0, M+1, \Sigma(\bar{x}))) && \text{(por el lema 3c)}\\
&< A(\text{máx}(N_0, M) + 1, \Sigma(\bar{x}) + 1) && \text{(por la ecuación A3)}\\
&< A(\text{máx}(N_0, M) + 2, \Sigma(\bar{x})) && \text{(por el lema 3b)}
\end{aligned}
$$

Por consiguiente, si escogemos N para que sea $(\text{máx}(N_0, M) + 2$, sabemos que $f(\bar{x}) < A(N, \Sigma(\bar{x}))$, para todas las $\bar{x}$ en $\mathbb{N}^n$, como se desea.

Consideremos ahora el caso en el cual f está definida por recursividad primitiva a partir de h y g, de la manera siguiente:

$$
\begin{aligned}
f(\bar{x}, 0) &= g(\bar{x})\\
f(\bar{x}, y+1) &= h(\bar{x}, y, f(\bar{x}, y))
\end{aligned}
$$

Como h y g se construyen con menos de k aplicaciones de combinaciones, composiciones y recursividades primitivas, existen enteros N_1 y N_2 tales que $g(\bar{z}) < A(N_1, \Sigma(\bar{z}))$ para toda $\bar{z}$ que esté en N^{n-1}, y $h(\bar{z}) < A(N_2, \Sigma(\bar{z}))$ para toda $\bar{z}$ que se encuentre en $\mathbb{N}^{n+1}$ (podemos suponer que $2 < N_2$). Debemos mostrar entonces que existe un entero positivo N tal que $f(x, y) < A(N, \Sigma(x) + y)$, para toda (x, y) en $\mathbb{N}^n$. Para esto, seleccionamos N para que sea cualquier entero mayor que $\text{máx}(N_1, N_2) + 4$ y demostramos la desigualdad resultante aplicando inducción en y. Comenzamos con $y = 0$, en cuyo caso tenemos

$$
\begin{aligned}
f(\bar{x}, 0) &\le f(\bar{x}, 0) + \Sigma(\bar{x})\\
&= g(\bar{x}) + \Sigma(\bar{x}) && \text{(por la definición de } g)\\
&< A(N_1, \Sigma(\bar{x})) + \Sigma(\bar{x}) && \text{(por la elección de } N1)\\
&< A(N_1 + 4, \Sigma(\bar{x})) && \text{(por el lema 5)}
\end{aligned}
$$

$$
\begin{aligned}
&< A(N, \Sigma(\bar{x})) && \text{(por el lema 3c y la elección de } N)\\
&= A(N, \Sigma(\bar{x}) + 0)
\end{aligned}
$$

Después suponemos que $f(\bar{x}, y) < A(N, \Sigma(\bar{x}) + y)$ para todo $y \le k$ y mostramos que esto implica que se cumpla la desigualdad para $y = k + 1$. De hecho,

$$
\begin{aligned}
f(\bar{x}, k+1) &= h(\bar{x}, k, f(\bar{x}, k))\\
&< A(N_2, \Sigma(\bar{x}) + k + f(\bar{x}, k)) && \text{(por la elección de } N_2)\\
&< A(N_2, A(0, \Sigma(\bar{x}) + k) + f(\bar{x}, k)) && \text{(por la ecuación A1)}\\
&< A(N_2, A(0, \Sigma(\bar{x}) + k) + A(N, \Sigma(\bar{x}) + k) && \text{(por la hipótesis de inducción)}\\
&< A(N_2, A(N, \Sigma(\bar{x}) + k) + A(N, \Sigma(\bar{x}) + k)) && (N > 0, \text{ lemas 3c y 3a})\\
&< A(N_2, 2A(N, \Sigma(\bar{x}) + k) + 3)\\
&= A(N_2, A(2, A(N, \Sigma(\bar{x}) + k))) && \text{(por el lema 1b)}\\
&\le A(N_2, A(N_2 + 1\ A(N, \Sigma(\bar{x}) + K))) && \text{(por los lemas 3a y 3c y la elección de } N_2)\\
&= A(N_2 + 1, A(N, \Sigma(\bar{x}) + k) + 1) && \text{(por la ecuación A3)}\\
&< A(N_2 + 2, A(N, \Sigma(\bar{x}) + k)) && \text{(por el lema 3b)}\\
&< A(N - 1, A(N, \Sigma(\bar{x}) + k)) && \text{(por la elección de } N \text{ y el lema 3c)}\\
&= A(N, \Sigma(\bar{x}) + k + 1) && \text{(por la ecuación A3)}
\end{aligned}
$$

que es el resultado que deseamos.

∎

APÉNDICE C

Algunos problemas importantes sin solución

El principal ejemplo de problema sin solución que se presenta en el texto (Cap. 4) es el problema de la parada. Quizá se trate del ejemplo más popular en los libros de texto porque es fácil de presentar en el contexto de las máquinas de Turing, una clase de dispositivos computacionales que por lo general ya se ha mencionado en algún curso previo al análisis de los problemas sin solución. Sin embargo, esta presentación es un tanto abstracta y con frecuencia los estudiantes sienten que es algo irrelevante. Muchos de ellos piensan que los problemas sin solución, aunque existen, nunca se presentan en situaciones normales, y consideran que pueden atacar los problemas cotidianos sin preocuparse por la posibilidad de que el problema en cuestión no tenga una solución algorítmica.

Empero, se trata de una evaluación incorrecta de la importancia de los problemas sin solución, pues éstos surgen en diversas situaciones. En el presente apéndice identificamos dos problemas sin solución que se presentan en situaciones reales y analizamos algunas de sus ramificaciones.

C.1 EVALUACIÓN DE GRAMÁTICAS INDEPENDIENTES DEL CONTEXTO PARA DETERMINAR SI EXISTE AMBIGÜEDAD

En el capítulo 2 presentamos el concepto de árbol de análisis sintáctico. Allí, lo que queríamos era mostrar que cualquier cadena que pudiera derivarse de una gramática independiente del contexto también se podía generar con una derivación por la izquierda. Un principio importante que se empleó en este planteamiento fue que la derivación de una cadena está asociada a un solo árbol de análisis sintáctico. Esto nos permitió mostrar que dada una derivación de una cadena, si no era una derivación por la izquierda, podíamos construir el árbol de análisis sintáctico asociado a esa derivación y después construir, a partir del árbol, una derivación por la izquierda para la cadena.

Puesto que no tenía importancia en el momento y se prestaba a posibles confusiones, en el capítulo 2 no mencionamos que una cadena podía estar asociada a más de un árbol de análisis sintáctico (toda derivación de una cadena se asocia a un solo árbol, pero la cadena puede estar asociada a más de uno). Considere la gramática

< frase > → < frase > o < frase >
< frase > → < frase > y < frase >
< frase > → X
< frase > → Y
< frase > → Z

que puede generar la frase

X o Y y Z

De hecho, puede generar esta frase a través de dos árboles de análisis sintáctico distintos, como se muestra en la figura C.1. Observe que la figura C.1a corresponde a una interpretación en la cual la veracidad de toda la frase depende de la veracidad de Z, mientras que la figura C.1b corresponde a una interpretación en la cual la frase puede ser verdadera sin que lo sea Z.

Otros ejemplos incluyen la estructura de programación común de estructuras if then else anidadas, en la cual una gramática inadecuadamente construida puede permitir dos interpretaciones distintas de la estructura

if *A* then if *B* then *X* else *Y*

¿Se ejecutará *Y* cuando *A* sea verdadero y *B* sea falso, o cuando *A* sea falso, sin importar *B*?

Se dice que las gramáticas que presentan estas anomalías son ambiguas. Está de más decir que sería una tarea frustrante construir un compilador para un lenguaje de programación basado en una gramática ambigua: dada la estructura if-then-else anterior, no se sabría qué código de máquina generar. Por consiguiente, hay que eliminar todas estas ambigüedades antes de intentar desarrollar un compilador para el lenguaje. Por ejemplo, la gramática ambigua que se mencionó antes se podría sustituir por

< frase > → < frase > o < término >
< frase > → < frase > y < término >
< frase > → < término >
< término > → X
< término > → Y
< término > → Z

< frase > → < término > o <frase>
< frase > → < término > y <frase>
< frase > → < término >
< término > → X
< término > → Y
< término > → Z

dependiendo de cual de las dos posibles interpretaciones se desee.

Por desgracia, las gramáticas para los lenguajes de programación actuales son mucho más complicadas que este sencillo ejemplo, por lo que es fácil (y frecuente) que un análisis intuitivo pase por alto ambigüedades sutiles. Lo que se necesita es un procedimiento rutinario que, aplicado a la gramática, detecte e informe acerca de las ambigüedades existentes. En otras palabras, lo que necesitamos es un algoritmo para detectar si determinada gramática independiente del contexto es ambigua. Sin embargo, esto no es posible: se ha mostrado que el problema de determinar si una gramática independiente del contexto es ambigua es un problema sin solución.

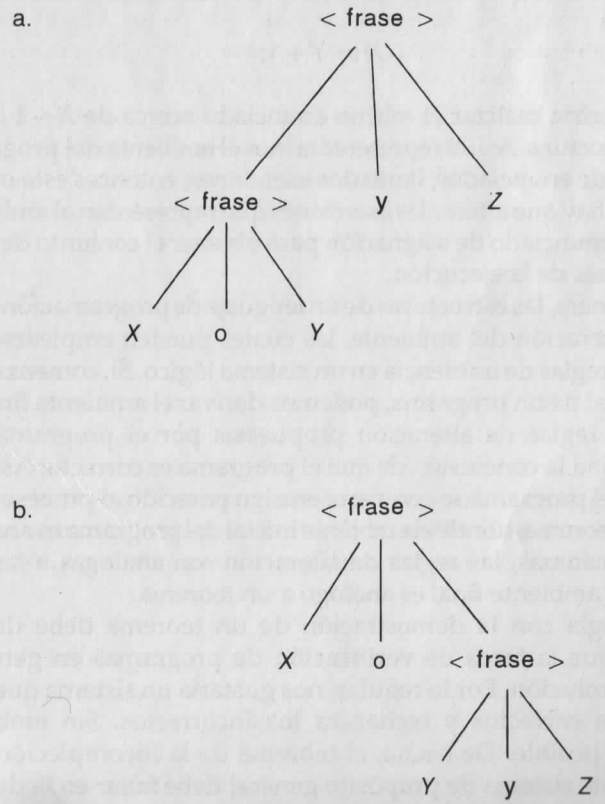


Figura C.1 Dos árboles de análisis sintáctico para la frase "*X* o *Y* y *Z*"

C.2 DEMOSTRACIÓN DE LA CORRECCIÓN DE PROGRAMAS

Conforme el software se hace más complejo y se aplica a situaciones cada vez más críticas, su corrección se convierte en un aspecto dominante. En varios casos se ha mostrado que la verificación con datos de prueba no ofrece la certidumbre requerida; lo que se necesita es una forma para demostrar de manera rigurosa la exactitud de un paquete de software. Puesto que estas demostraciones pueden convertirse en algo tan complejo y tedioso como el propio software, nos gustaría desarrollar un sistema automatizado para llevarlas a cabo.

La elaboración de estas técnicas ha sido el objetivo de muchos investigadores en años recientes, y se han logrado ciertos avances. Uno de los enfoques importantes es identificar formalmente el efecto que cada estructura de un lenguaje de programación tiene en el estado del ambiente del programa durante la ejecución. Por ejemplo, si pudiera efectuarse algún enunciado acerca de la variable Y antes de ejecutar la estructura de Pascal

$$X := Y + 1;$$

entonces se podría realizar el mismo enunciado acerca de $X - 1$ después de ejecutar la estructura. Así, si representáramos el ambiente del programa como una colección de enunciados, llamados aserciones, entonces esta observación nos dice cómo hay que alterar las aserciones que representan al ambiente antes de ejecutar el enunciado de asignación para obtener el conjunto de aserciones correcto después de la ejecución.

De esta manera, las estructuras de un lenguaje de programación dan origen a reglas de alteración del ambiente, las cuales pueden emplearse en forma parecida a las reglas de inferencia en un sistema lógico. Si, comenzando con el ambiente inicial de un programa, podemos derivar el ambiente final deseado aplicando las reglas de alteración propuestas por el programa, entonces podemos llegar a la conclusión de que el programa es correcto. Así mismo, la verificación del programa se convierte en algo parecido al proceso de demostración de un teorema, donde el ambiente inicial del programa es análogo a una colección de axiomas, las reglas de alteración son análogas a las reglas de inferencia y el ambiente final es análogo a un teorema.

Esta analogía con la demostración de un teorema debe despertar la sospecha de que la tarea de verificación de programas en general es un problema sin solución. Por lo regular, nos gustaría un sistema que verificara los programas correctos y rechazara los incorrectos. Sin embargo, este sistema no es posible. De hecho, el teorema de la incompleción de Godel implica que este sistema de propósito general debe fallar en la detección de algunos programas incorrectos.

En caso de que el lector no se encuentre todavía dispuesto a aceptar la carencia de solución del problema general de verificación a partir de esta analogía intuitiva con la demostración de teoremas, veamos la tarea de verificación de programas desde otra perspectiva. Uno de los principales componentes de cualquier proceso de verificación de programas es mostrar que el programa terminará. Sin embargo, puede verse que esto no es más que el problema de la parada disfrazado. Si el lenguaje de programación tiene el poder para calcular todas las funciones recursivas parciales, como sucede con el sencillo lenguaje que se desarrolló en el capítulo 4, entonces no existe ningún algoritmo para detectar si los programas se detendrán en ese lenguaje. Por lo tanto, como no se puede resolver el problema de la parada, esto nos indica que el sueño de los ingenieros de software, en su forma más general, es imposible.

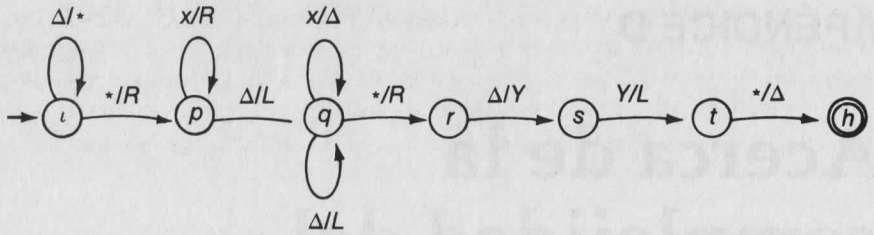
APÉNDICE D

Acerca de la complejidad del problema de comparación de cadenas

En este apéndice nuestro objetivo es mostrar que cualquier solución (empleando una máquina de Turing de cinta única) para el problema de comparación de cadenas presentado en el capítulo 5 tendrá una complejidad temporal que por lo menos es una expresión cuadrática de n, donde n es la longitud de las cadenas de entrada (recuerde que en este problema de comparación se tenía que decidir si dos cadenas de $\{x, y\}^*$ de igual longitud eran idénticas; las cadenas se registran en la cinta de una máquina de Turing, separadas por un asterisco). Para lograr nuestro objetivo nos apoyaremos en el concepto de secuencia de cruce, por lo que explicaremos este concepto antes de proceder.

Dada una máquina de Turing (en la cual imaginamos que las celdas de la cinta están numeradas de izquierda a derecha, comenzando por el número uno) y un cálculo de dicha máquina, la **secuencia de cruce** de la celda i de la cinta es un registro de todas las veces que la cabeza de la cinta ha cruzado la frontera entre las celdas i e $i + 1$ durante los cálculos. Este registro se mantiene como una lista de estados: cada vez que la cabeza cruza la frontera, registramos el estado de la máquina que se presenta inmediatamente después de ejecutar la transición. Así, si sabemos que un cálculo determinado comenzó con la cabeza colocada sobre la celda del extremo izquierdo de la cinta y que produjo la secuencia de cruce $q_1, q_2, \cdots, q_n$ de la celda i, sabemos que la primera vez que la cabeza pasó de la celda i a la celda $i + 1$ la máquina pasó al estado q_1. Más tarde, cuando la cabeza regresó por la derecha a la celda i, la máquina pasó al estado q_2, etcétera. En la figura D.1 se presenta un ejemplo más detallado.

Definimos la longitud de una secuencia de cruce como el número de entradas en la secuencia. Es sencillo ver que la suma de las longitudes de todas las secuencias de cruce para las distintas celdas de la máquina equivale al número de transiciones de movimiento ejecutadas por la máquina (suponiendo que los cálculos no terminaron anormalmente). De manera específica, esta suma proporciona un límite inferior para la complejidad temporal de los cálculos.



Número de celda	Secuencia de cruce (dada la configuración inicial $\underline{\Delta}xx\Delta\Delta\ldots$)
1	*p, q, r, t*
2	*p, q,*
3	*p, q*
4	vacía
5	vacía
⋮	⋮

Figura D.1 Diagrama de transiciones para una máquina de Turing y la secuencia de cruce correspondiente al aceptar la cadena *xx*

Otro punto que hay que señalar es que si w_1, w_2, w_3, w_4 son cadenas tales que $|w_1| = |w_3| = i$, y una máquina de Turing M acepta w_1w_2 y w_3w_4 con cálculos que producen secuencias de cruce idénticas en la celda $i + 1$ (la celda que contiene el último símbolo en w_1 o w_3) entonces M debe aceptar también la cadena w_1w_4. Los cálculos realizados en las celdas 1 a $i + 1$ al procesar w_1w_4 serían los mismos que se efectuarían al procesar w_1w_2, a la vez que los cálculos efectuados en las celdas $i + 2$ en adelante serían idénticos a los que se ejecutarían cuando se procesara la cadena w_3w_4.

Ahora estamos preparados para demostrar que *cualquier solución para nuestro problema de comparación de cadenas, empleando una máquina de Turing (de cinta única), debe tener una complejidad temporal que es una expresión cuadrática de* n, *donde* n *es la longitud de la cadena de entrada.*

Sea M cualquier máquina de Turing tal que $L(M)$ sea el lenguaje que consiste en todas las cadenas de la forma $w*w$, donde w es una cadena en $\{x, y, z\}^*$. Para todo $n \in \mathbb{N}^+$, sea W_n el subconjunto de $L(M)$ que consiste en las cadenas de la forma $w*w$, donde $|w| = n$. Mostraremos ahora que la longitud promedio de todos los cálculos de M con entradas de W_n tiene como límite inferior una expresión cuadrática de n. De aquí se desprende que la complejidad temporal de M es, por lo menos, una expresión cuadrática de n (el funcionamiento de M en el peor caso no puede ser más rápido que el del caso promedio).

El número medio de pasos que ejecuta M al aceptar una cadena de W_n, que representamos como $A_n(M)$, no es menor que la suma de las longitudes promedio de las secuencias de cruce para las celdas 1 a n en los mismos cálculos. Es decir, si $a_n(i)$ es la longitud promedio de las secuencias de cruce de la celda i para todos los cálculos de M con entradas de W_n, entonces

$$A_n(M) \geqslant \sum_{i=1}^{n} a_n(i)$$

Completamos la demostración mostrando que para cada i en $\{1, 2, \cdots, n\}$, existen constantes $c_i > 0$ y b_i tales que $a_n(i) \geq c_i i + b_i$ ya que podemos ampliar la desigualdad anterior de la manera siguiente:

$$\begin{aligned} A_n(M) &\geqslant \sum_{i=1}^{n} a_n(i) \\ &\geqslant \sum_{i=1}^{n} (c_i i + b_i) \\ &\geqslant \text{mín}(c_1, \cdots, c_n) \sum_{i=1}^{n} i + \sum_{i=1}^{n} b_i \\ &= \text{mín}(c_1, \cdots, c_n) \left(\frac{n(n+1)}{2}\right) + \sum_{i=1}^{n} b_i \\ &= \frac{\text{mín}(c_1, \cdots, c_n)}{2}(n^2 + n) + \sum_{i=1}^{n} b_i \end{aligned}$$

lo que muestra que $A_n(M)$ tiene como límite inferior una expresión cuadrática de n, como se planteó.

Pasemos entonces a mostrar que para cada i en $\{1, 2, \cdots, n\}$ existen constantes $c_i > 0$ y b_i tales que $a_n(i) \geq c_i i + b_i$.

Para cada i en $\{1, 2, \cdots, n\}$ observamos que para un mínimo de dos tercios de las 3^n cadenas $w*w$ en W_n, los cálculos de M durante la aceptación de $w*w$ producen una secuencia de cruce en la celda i que no tiene una longitud mayor que $3a_n(i)$ (si no fuera así, existirían 3^{n-1} cadenas en W_n que producirían secuencias de cruce de longitud mayor que $3a_n(i)$ en la celda i y entonces,

$$a_n(i) > \frac{3^{n-1}[3a_n(i)]}{3^n} = a_n(i)$$

lo cual es una contradicción). En consecuencia, por lo menos 3^{n-1} cadenas en W_n llevan a secuencias de cruce de longitud no mayor que $3a_n(i)$.

De aquí se desprende que muchas cadenas de W_n llevan a la misma secuencia de cruce para la celda i. Si k es el número de estados en M (entonces $k \geq 2$), existen cuando mucho k^j secuencias de cruce diferentes con longitud j, y por lo tanto un máximo de

$$\sum_{j=0}^{3a_n(i)} k^j$$

secuencias de cruce diferentes con longitud máxima de $3a_n(i)$. Sin embargo, esta suma no es mayor que $(3a_n(i) + 1)\,(k^{3a_n(i)})$, que tiene como límite superior $(k^{3a_n(i)+1})\,(k^{3a_n(i)}) = k^{6a_n(i)+1}$. Entonces, existen por lo menos

$$\frac{3^{n-1}}{k^{6a_n(i)+1}}$$

cadenas en W_n que producen la misma secuencia de cruce en la celda i (existen por lo menos 3^{n-1} cadenas en W_n con secuencias de cruce que no son más largas que $3a_n(i)$ y no más de $k^{6a_n(i)+1}$ de estas secuencias).

Ahora afirmamos que

$$\frac{3^{n-1}}{k^{6a_n(i)+1}} \leqslant 3^{n-i}$$

(De lo contrario, existirían más cadenas en W_n que llevarían a la misma secuencia de cruce para la celda i que cadenas en $\{x, y, z\}^*$ de longitud $n-i$. Por lo tanto, existirían cadenas diferentes w_1w_3 y w_2w_3 de longitud n, donde $|w_3| = n-i$, para las cuales $w_1w_3{*}w_1w_3$ y $w_2w_3{*}w_2w_3$ producirían la misma secuencia de cruce para i al emplearse como entrada de M. En consecuencia, por la observación que precede a esta demostración, M tendría que aceptar la cadena $w_1w_3{*}w_2w_3$ lo cual es una contradicción.)

Sin embargo, esta última desigualdad implica que

$$log_3\left(\frac{3^{n-1}}{k^{6a_n(i)+1}}\right) \leqslant log_3(3^{n-i})$$

o

$$log_3(3^{n-1}) - log_3(k^{6a_n(i)+1}) \leqslant log_3(3^{n-i})$$

o

$$(n-1) - [6a_n(i) + 1]\,log_3 k \leqslant n - i$$

por lo que

$$a_n(i) \geqslant \frac{i-1}{6\,log_3 k} - \frac{1}{6} = \frac{i}{6\,log_3 k} + \frac{-1 - log_3 k}{6\,log_3 k}$$

como se requiere.

APÉNDICE E

Muestras de problemas *NP*

En este apéndice identificamos algunos de los muchos problemas de decisión cuyos lenguajes asociados se ha mostrado que pertenecen a la clase *NP*, pero no se ha podido determinar si pertenece o no a *P* (de hecho, todos los problemas que aquí se presentan son *NP*-completos).

E.1 LENGUAJES FORMALES Y TEORÍA DE AUTÓMATAS

Existen diversos problemas relacionados con nuestro estudio de los lenguajes formales y la teoría de autómatas cuyos lenguajes están en *NP*. He aquí algunos.

1. Dados dos autómatas finitos no deterministas con el mismo alfabeto, responder *S* si y sólo si aceptan lenguajes distintos.
2. Dadas dos expresiones regulares del mismo alfabeto, responder *S* si y sólo si representan lenguajes diferentes (puede comparar este problema con el anterior).
3. Dada una gramática independiente del contexto y un entero positivo k, responder *S* si y sólo si la gramática no es $LR(k)$.
4. Dada una colección finita de autómatas finitos deterministas, responder *S* si y sólo si existe una cadena aceptada por todos y cada uno de los autómatas de la colección.

E.2 TEORÍA DE NÚMEROS

El tema de la teoría de números es rico en ejemplos de problemas *NP*. A continuación se presentan algunos.

1. Dados tres enteros positivos a, b y c, responder S si y sólo si la ecuación $ax^2 + by = c$ tiene una solución que consista en enteros positivos.
2. Dados tres enteros positivos a, b y c, responder Y si y sólo si existe un entero positivo x que sea menor que c y $x^2 \equiv a(mod\ b)$.
3. Dados una colección finita K de enteros positivos y otro entero positivo n, responder Y si y sólo si existe un subconjunto de K cuya suma sea n.

E.3 LÓGICA

En el texto vimos que el problema de la satisfactibilidad es un importante problema *NP*. He aquí dos problemas relacionados.

1. Dada una colección de cláusulas donde cada una contiene exactamente tres literales, responder Y si y sólo si existe una asignación de verdad que satisfaga toda la colección (es el mismo problema que SAT, excepto que se requiere que las cláusulas tengan un tamaño uniforme de tres. Por esto se le conoce como SAT-3).
2. Dada una expresión que consiste en variables conectadas por los símbolos $\neg$, $\cap$, $\cup$ y $\rightarrow$ (que significan no, y, o e implica, respectivamente), responder Y si y sólo si existe una manera de asignar los valores falso y verdadero a las variables para que la expresión sea falsa.

E.4 PROBLEMAS DE PLANIFICACIÓN

En el contexto de la preparación de planes (scheduling) existen varios problemas *NP* para los cuales no se conoce una solución eficiente; los siguientes son dos de dichos problemas.

1. Dados un conjunto T de tareas, la cantidad de tiempo requerida para completar cada tarea, un entero positivo n y un tiempo límite, responder S si y sólo si existe una manera de asignar las tareas a n procesadores para que todo el conjunto de tareas se cumpla antes del tiempo límite.
2. El siguiente es un interesante problema de planificación que ocurre en el contexto de la administración de bases de datos. Suponga que tenemos una base de datos central que es manipulada por varios usuarios remotos. Cada transacción solicitada por un usuario consiste en una serie de pasos que leen y escriben varios datos. Mientras los pasos de las distintas transacciones no tengan que ver con el mismo dato o no se mezclen, la validez de la base de datos no peligra. Sin embargo, si se mezclan los pasos de transacciones diferentes y llegan a manipular el mismo dato, la base de datos puede llegar a ser inválida.

   Una solución es requerir que cualquier transacción termine antes de permitir a otra el acceso a la base de datos; esto genera lo que se denomina planificación secuencial de las transacciones. Esta estrategia resuelve el problema, pero tiene el efecto secundario indeseable de hacer innecesariamente más lento el funcionamiento del sistema. Existen varias formas de mezclar los pasos de las distintas transacciones para que no produzcan datos inválidos, incluso si los pasos manipulan los mismos datos. Así, resulta innecesariamente restrictivo bloquear el acceso a la base de datos de una transacción hasta que otra termine.

   Lo que se necesita es un paquete de software, conocido como planificador (scheduling), que reciba todas las solicitudes de los usuarios y planifique su acceso a la base de datos de manera que permita su combinación (por cuestiones de eficiencia) pero evite las solicitudes conflictivas (para mantener la validez). Un enfoque bastante sencillo se basa en el concepto de la posibilidad de serializar vistas. Se dice que la planificación de transacciones tiene vistas serializables si cada paso de lectura de las transacciones lee exactamente el mismo valor que habría leído si las transacciones se hubieran ejecutado en algún orden secuencial y sin superposiciones.

   Esta planificación basta para mantener la validez de la base de datos, y se podría conjeturar la posibilidad de desarrollar un planificador basado en esta idea. El planificador permitiría que varias transacciones tuvieran acceso concurrente a la base de datos mientras la combinación tuviera una vista serializable. Así mismo, las tran-sacciones que destruyeran la posibilidad de serializar vistas se retendrían hasta que pudieran ejecutarse sin peligro.

   Por desgracia, no se ha encontrado un algoritmo para evaluar una serie de transacciones y determinar si tiene tiene vistas serializables. Las planificaciones de vista serializable constituyen un lenguaje en *NP*, pero nadie ha encontrado un algoritmo determinista, en tiempo polinómico, para reconocer dicho lenguaje. Por lo tanto, aún no se ha determinado si es posible construir planificadores eficientes basados en la posibilidad de serializar vistas.

E.5 TEORÍA DE GRAFOS

Recuerde que un grafo es una colección de nodos llamados vértices, algunos de los cuales están unidos por líneas denominadas arcos. A continuación aparecen algunos de los muchos problemas *NP* que se presentan en el contexto de la teoría de grafos.

1. Dado un grafo, responder *S* si y sólo si dicho grafo contiene una ruta que visita precisamente una vez cada uno de los vértices y termina en el mismo vértice donde se inició (esta ruta se conoce como **circuito de Hamilton;** el problema se conoce como **problema del circuito de Hamilton**).
2. Dados un grafo y un entero positivo *n*, responder *S* si y sólo si el grafo contiene un clan con *n* vértices (en un grafo, un clan es un conjunto de vértices donde dos cualesquiera están conectados por un arco).
3. Dados un grafo *G* y un entero positivo *n*, responder *S* si y sólo si *G* contiene *n* clanes que juntos contengan todos los vértices de *G*.

E.6 TEORÍA DE CONJUNTOS

Los siguientes son problemas *NP* que tienen que ver con conjuntos.

1. Dados un conjunto *X* y una colección finita de subconjuntos de *X*, responder *S* si y sólo si existe una subcolección de dichos subconjuntos que sea disjunta por pares y cuya unión sea todo *X* (esto se conoce como problema de la cobertura exacta).
2. Dados un conjunto finito *X* en el cual a cada elemento se le asigna un valor entero positivo llamado tamaño, y dos enteros positivos *m* y *n*, responder *S* si y sólo si existen por lo menos *m* subconjuntos de *X* en los cuales la suma de los tamaños no sea mayor que *n*.
3. Dados un conjunto finito *X* en el cual a cada elemento se le asigna un valor entero positivo llamado tamaño, y un entero positivo *n*, responder *S* si y sólo si existe un subconjunto de *X* para el cual el producto de los tamaños de sus miembros sea *n*.

Lecturas adicionales

Existen numerosos libros que analizan con mayor detalle, o quizás desde otra perspectiva, el material cubierto en este texto. A continuación se presenta una lista parcial de estos libros. Aquellos que se incluyen bajo "Cobertura general" tratan una mayor variedad de temas que aquellos que se incluyen bajo encabezamientos específicos.

LENGUAJES FORMALES Y TEORÍA DE AUTÓMATAS

Bavel, Z., *Introduction to the Theory of Automata*, Reston, Virginia, Reston, 1983.

Cohen, D. I. A., *Introduction to Computer Theory*, Nueva York, John Wiley, 1986.

Harrison, M. A., *Introduction to Formal Language Theory*, Menlo Park, California, Addison-Wesley, 1978.

Revesz, G. E., *Introduction to Formal Languages*, Nueva York, McGraw-Hill, 1983.

ANÁLISIS SINTÁCTICO Y CONSTRUCCIÓN DE COMPILADORES

Aho, A. V., Sethi, R. y Ullman, J. D., *Compilers: Principles, Techniques, and Tools*, Menlo Park, California, Addison-Wesley, 1986. (Existe traducción al castellano: Compiladores: principios, técnicas y herramientas, Wilmington, Delaware, Addison-Wesley Iberoamericana, 1990.)

Aho, A. V. y Ullman, J. D., *The Theory of Parsing, Translation, and Compiling*, vols. 1 y 2, Englewood Cliffs, Nueva Jersey, Prentice–Hall, 1972.

Barrett, W. A. y Couch, J. D., *Compiler Construction: Theory and Practice*, Chicago, Illinois, SRA, 1979.

Fischer, C. N. y LeBlanc, Jr., R. J., *Crafting a Compiler*, Menlo Park, California, Benjamin/Cummings, 1988.

Tremblay, J. y Sorenson, P. G., *The Theory and Practice of Compiler Writing*, Nueva York, McGraw-Hill, 1985.
Waite, W. M. y Goos, G., *Compiler Construction*, Nueva York, Springer-Verlag, 1984.

COMPUTABILIDAD Y TEORÍA DE FUNCIONES RECURSIVAS

Cutland, N. J., *Computability: An Introduction to Recursive Function Theory*, Cambridge, Inglaterra, Cambridge University Press, 1980.
Machtey, M. y Young, P., *An Introduction to the General Theory of Algorithms*, Nueva York, North Holland, 1978.
Rogers, H. Jr., *Theory of Recursive Functions and Effective Computability*, Cambridge, Massachusetts, MIT Press, 1987.

COMPLEJIDAD

Garey, M. R. y Johnson, D. S., *Computers and Intractability*, Nueva York, Freeman, 1979.
Wilf, H. S., *Algorithms and Complexity*, Englewood Cliffs, Prentice-Hall, 1986.

COBERTURA GENERAL

Beckman, F. S., *Mathematical Foundations of Programming*, Menlo Park, California, Addison-Wesley, 1981.
Brainerd, W. S. y Landweber, L. H., *Theory of Computation*, Nueva York, John Wiley, 1974.
Davis, M. D. y Weyuker, E. J., *Computability, Complexity, and Languages*, Londres, Academic Press, 1983.
Denning, P. J., Dennis, J. B. y Qualitz, J. E., *Machines, Languages, and Computation*, Englewood Cliffs, Nueva Jersey, Prentice-Hall, 1978.
Hopcroft, J. E. y Ullman, J. D., *Introduction to Automata Theory, Languages, and Computation*, Menlo Park, California, Addison-Wesley, 1979.
Lewis, H. R. y Papadimitriou, C. H., *Elements of the Theory of Computation*, Englewood Cliffs, Nueva Jersey, Prentice–Hall, 1981.
McNaughton, R., *Elementary Computability, Formal Languages, and Automata*, Englewood Cliffs, Nueva Jersey, Prentice–Hall, 1982.
Sudkamp, T. A., *Languages and Machines*, Menlo Park, California, Addison-Wesley, 1988.
Wood, D., *Theory of Computation*, Nueva York, Harper and Row, 1987.

ÍNDICE DE MATERIAS

Ackermann, W., 217
Ackermann, función de, 217, 305-311
alfabeto, 28
algoritmo, 1
  complejidad de un, 251-258
  Véase también proceso computacional
almacenamiento en pila, 76
análisis léxico, 13, 22-28
análisis sintáctico, 13, 131
  árbol de, 83, 314
  tabla de,
    para analizadores sintácticos *LL*, 119-120
    para analizadores sintácicos *LR*, 126-131, 297-303
analizador
  léxico, 13
  sintáctico, 13
    ascendente, 122
    descendente, 122
    LL(*k*), 121-133
    LR(*k*), 114-122,131-133
    predictivo, 118
arco en un diagrama de transiciones, 22
aserción en la demostración de la corriente de un programa, 316
asignación de verdad, 282
autómatas de pila, 16, 75-140
  definición de los, 76, 79
  determinista, 106-11
    definición de los, 106-107
autómatas finitos, 16, 21-72, 128-131, 297-303
  deterministas, 28-42
    definición de los, 29, 32
  no deterministas, 42-49
axioma, 14
  de elección, 9

Blum, M., 264
Blum, teorema de aceleración de, 260-264

cadena(s)
  aceptación de,
    por autómatas de pila, 79
    por autómatas de pila deterministas, 80-81
    por autómatas de pila deterministas con pila vacía,
    por autómatas finitos deterministas, 30,32
    por autómatas finitos no deterministas, 44
    por máquinas de Turing, 158
    por máquinas de Turing con varias cintas, 162
    por máquinas de Turing no deterministas, 169
  generación por una gramática, 51
  vacía (λ), 38
computabilidad, 203-243
cálculos de tiempo polinómico, 267-268
  de máquinas de Turing deterministas, 267-278
  de máquinas de Turing no deterministas, 278-290
cardinalidad de un conjunto, 7-8
caso básico en demostración por inducción, 6

Chomsky, N., 16, 100, 196

Chomsky,
forma normal de, 94-100, 189
definición de la, 100
jerarquía de, 196
Church, A., 221
Church, tesis de, 16, 221, 233
Church-Turing, tesis de, 233, 241, 291
cinta, símbolos de, 143
clase, 3
de equivalencia, 266
clasificación por inserción, 256
cláusula, 282
cláusulas satisfechas, conjuntos de, 282
cobertura exacta, problema de, 326
combinación de funciones, 207
complejidad
espacial, 248
de cálculos de máquinas de Turing, 250
temporal, 248-296
definición de la, 248
teoría de la, 247-296
componente léxico [símbolo], 13
composición de funciones, 207
concatenación de lenguajes, 60
conjuntos (s)
contable, 10
enumerable. *Véase* conjunto contable
incontable, 10
infinito, 6
potencia, 4, 9
regular. *Véase* lenguajes regulares

teoría de, 2-12, 326
complemento, 3
complemento relativo, 3
intersección, 3
producto cartesiano, 3
subconjunto, 3
unión, 3
vacío,
contradominio, de una función, 4
Cook, S. A., 282
Cook, teorema de, 282-290
corrección de un programa,
demostración de la, 236-327
costo uniforme, suposición de, 256

decisión, problemas de, 276-278
DeMorgan, leyes de, 17, 136
demostración por inducción, 4-6
derivación, 52
por la derecha, 83
por la izquierda, 4-6
desplazamiento, operación de,
en un analizador sintáctico *LR(k)*, 122
diagrama(s)
compuesto para una máquina de Turing, 149-157
definición de un, 149-150
de estado. *Véase* transiciones, diagrama de
de sintaxis, 55
de transiciones. *Véase* transiciones, diagrama de
dominio de una función, 4

Einstein, teoría de la relatividad de, 2
elementos de un conjunto, 2
encadenamiento λ, 96, 97
enteros
no negativos. *Véase* números naturales
positivos ($\mathbb{N}^+$), 3
estado,
de aceptación, 33
inicial, 23
expresión regular, 62

flujo de entrada, 29
función (funciones), 4
computables, por máquinas de Turing, 22-227
característica, 11, 214-215
cero, 206
constantes, 211
de proyección, 207
estrictamente parcial, 205
inicial, 206-207
μ-recursivas, 218
parcial, 205-206
predecesor, 213
recursiva, teoría de, 16, 204-234
recursivas parciales, 219-22
definición de las, 220
equivalencias con las funciones computables según Turing, 222-223

recursivas primitivas, 207-212
coc, 216
eq, 213
exp, 213
K^n_m, 212
más, 209
monus, 213
mult, 213
pred, 213
sobre, 4
sucesor, 207
tabular, 215
total, 205
uno a uno, 4

generador de código, 13, 131
Gödel, Kurt, 15
grafos, teoría de, 325-326
gramática
ambigua, 101, 324-325
estructurada por frases, 51, 172
independiente del contexto, 82-102
definición de una, 82-85
regular, 50-57
definición de una, 53
lenguaje generado por una, 52-53
Griebach, forma normal de, 325

Hamilton, circuito de, 325
Hamilton, problema del circuito de, 325
incompletabilidad, teorema de la, 15
inferencia, reglas de, 15

Kleene, estrella de, 60
Kleene, S. C., 60

lema de bombeo
para lenguajes independientes del contexto, 102-103
para lenguajes regulares, 42
lenguaje(s)
aceptado por una máquina de Turing, 158
aceptado por un autómata de pila, 79
aceptado por un autómata finito, 37
de programación esencial, 233-239
definición de un, 37
enumerables recursivamente, 191
estructurados por frases, 141-202
definición de los, 172
equivalencia con los lenguajes aceptados por máquinas de Turing, 172-178
formales, 12
independientes del contexto, 75-140
decidibles, 189-192
definición de los, 85
deterministas, 110, 132
naturales, 12
comparados con los lenguajes formales, 12
no estructurados por frases, 178-179, 182-183
recursivos, 190-192
regulares, 21-73
definición de los, 37
pertenencia a *P* de los , 277
vacío, 38
literales, 282

máquina(s) de Turing, 16, 141-202
autoterminante, 188
construcción de, 148-157
de varias cintas, 158-168
definición de una, 146
no determinista, 168-171
universal, 184-186
marca de fin de cadena, 22
mecanismo de control, 29
método axiomático, 14
miembros, de un conjunto. *Véase* elementos
minimalización, 219
modus ponens, 15

Newton, Isaac, 2
no terminal en una gramática, 50
NP, 278-290
definición de, 278
NP-completo, problemas, ejemplos, 313-316
número(s)
cardinal, 8
naturales ($\mathbb{N}$), 3
teoría de, 323-324

O, notación, 262
orden lineal, 266

P, 268-278
definición de, 268
palíndromo, 69, 138, 281
parada,
estado de, 143
problema de la, 193-196, 313,137
paso de un cálculo de una máquina de Turing, 249
pila, 76
símbolos de, 77
Post, Emil L., 142
preanálisis, principio de, 11-113
en analizadores sintácticos *LL(k)*, 116-119
en analizadores sintácticos *LR(k)*, 125-127
problema,
de decisión del viajante, 279
soluble. *Véase* Turing, problema soluble según
proceso computacional, 1
limitaciones de un, 11-12, 179-180, 203-204, 240-241
Véase también algoritmo
producto cartesiano, 3
programa
fuente, 13
objeto, 14

razonamiento deductivo, 15
recursividad primitiva, 208
red de transición. *Véase* transiciones, diagrama de
reducción operación de, en un analizador sintáctico *LR(k)*, 122-123
polinómica, 280-282
regla
de reescritura, 50-51
lineal por la derecha, 70
lineal por la izquierda, 70
regla λ,
definición de la, 51-52
necesidad de la, en gramáticas independientes del contexto, 96-98
relación de equivalencia, 266
SAT, 282-290
definición de, 282
SAT-3, 324
satisfactibilidad. *Véase* SAT
Schröder-Bernstein, teorema de, 8
secuencia
de configuración de una máquina de Turing, 174
de cruce, 319
símbolo de inicio de una gramática, 50
subconjunto, 3
propio, 3

tasa de crecimiento, 262-266
definición de la, 262
teorema, 15
terminal en una gramática, 50
traducción, proceso de, 12-14
transición (transiciones),
de estado, 29
diagrama de,
completamente definido, 32
generalizado, 63-64
para autómatas de pila, 78
para autómatas finitos, 22
para máquinas de Turing, 144-145
función de,
para autómatas finitos, 32
para máquinas de Turing, 144
λ, 70
tabla de, 24
Turing, Alan M., 142
Turing,
función computables según, 122
lenguajes aceptados por máquinas de, 158-180
definición de los, 158-159
equivalencia con los lenguajes estructurados por frases, 172-180
lenguages decidibles por un máquina de, 189-195
definición de los, 189
máquina de. *Véase* máquina(s) de Turing,
problema soluble por, 241
tesis de, 16, 142, 147, 179-180

vista serializable, planificación de, 325

VOCABULARIO TÉCNICO BILINGÜE *

aceptación de cadenas
string acceptance
alfabeto
alphabet
algoritmo
algorithm
almacenamiento en pila
pushdown store
análisis léxico
lexical analysis
análisis sintáctico
parsing
analizador léxico
lexical analyzer
analizador sintáctico
parser
analizador sintáctico ascendente
bottom-up parser
analizador sintáctico descendente
top-down parser
analizador sintáctico *LL(k)*
LL(k) parser
analizador sintáctico *LR(k)*
LR(k) parser
analizador sintáctico predictivo
predictive parser
árbol de análisis sintáctico
parse tree
arco, en un diagrama de transiciones
arc, in a transition diagram
aserción en la demostración de la corrección de un programa
assertion in prof of program correctness
asignación de verdad
truth assignment
autómatas de pila
pushdown automata
autómatas de pila deterministas
deterministic pushdown automata
autómatas finitos
finite automata
autómatas finitos deterministas
deterministic finite automata
autómatas finitos no deterministas
nondeterministic finite automata
axioma
axiom
axioma de elección
Axiom of Choice

cadena vacía (λ)
empty string (λ)
cadenas, generación por una gramática
string generation by a grammargs
computabilidad
computability
cálculo de una función por máquinas de Turing
computation of a function by Turing machines

* Ante la falta de un lenguaje estandarizado en castellano para las ciencias de la computación, se ha elaborado el presente vocabulario con la traducción que hemos dado en este libro a los principales términos de la versión original en inglés. Esta labor se verá compensada por el servicio que pueda prestar al lector. *(N. del E.)*

cálculos de tiempo polinómico
polynomial-time computations
cardinalidad de un conjunto
cardinality of a set
caso básico en demostración por inducción
base case in proof by induction
circuito de Hamilton
Hamilton circuit
clase
class
clase de equivalencia
equivalent class
clasificación por inserción
insertion sort
cláusula
clause
combinación de funciones
combination of functions
complejidad de un algoritmo
complexity of an algorithm
complejidad espacial
space complexity
complejidad temporal
time complexity
complemento
complement
complemento relativo
relative complement
componente léxico (símbolo)
token (en contextos de análisis léxico)
composición de funciones
composition of functions
concatenación de lenguajes
concatenation of languages
conjunto
set
conjunto contable
countable set
conjunto de cláusulas satisfechas
satisfied set of clauses
conjunto enumerable
enumerable set
conjunto incontable
uncountable set
conjunto infinito
infinite set
conjunto potencia
power set
conjunto regular
regular set
conjunto vacío
empty set
contradominio de una función
range of a function

decidibilidad de los lenguajes independientes del contexto
decidability of context-free languages
demostración de la corrección de un programa
proof of program correctness
demostración por inducción
proof by induction
derivación
derivation
derivación por la derecha
rightmost derivation
derivación por la izquierda
leftmost derivation
diagrama compuesto para una máquina de Turing
composite diagram for a Turing machine
diagrama de estado
state diagram
diagrama de sintaxis
syntax diagram
diagrama de transiciones
transition diagram
diagrama de transiciones completamente definido
fully defined transition diagram
diagrama de transiciones generalizado
generalized transition diagram
dominio de una función
domain of a function

encadenamiento λ
λ -chain
enteros no negativos
nonnegative integers
enteros positivos ($\mathbb{N}^+$)
positive integers ($\mathbb{N}^+$)
estado
state

estado de aceptación
accept state
estado de parada
halt state
estado inicial
initial state
estrella de Kleene
Kleene star
expresión regular
regular expression

flujo de entrada
input stream
forma normal de Chomsky
Chomsky normal form
forma normal de Greibach
Greibach normal form
función
function
función característica
characteristic function
función cero
zero function
función de Ackermann
Ackermann's function
función de transición
transition function
función estrictamente parcial
strictly partial function
función inicial
initial function
función parcial
partial function
función predecesor
predecessor function
función recursiva primitiva
primitive recursive function
función recursiva primitiva *coc*
quo primitive recursive function
función recursiva primitiva *eq*
eq primitive recursive function
función recursiva primitiva *exp*
exp primitive recursive function
función recursiva primitiva K^n_m
K^n_m primitive recursive function
función recursiva primitiva *más*
plus primitive recursive function
función recursiva primitiva *monus*
monus primitive recursive function
función recursiva primitiva *mult*
mult primitive recursive function
función recursiva primitiva *pred*
pred primitive recursive function
función sobre
onto function
función sucesor
successor function
función tabular
tabular function
función total
total function
función uno a uno
one-to-one function
funciones calculables según Turing
Turing-computable functions
funciones constantes
constant functions
funciones de proyección
projection functions
funciones iniciales
initial functions
funciones μ-recursivas
μ-recursive functions
funciones recursivas parciales
partial recursive functions
funciones recursivas primitivas
primitive recursive functions

generador de código
code generator
gramática ambigua
ambiguous grammar
gramática estructurada por frases
phrase-structure grammar
gramática independiente del contexto
context-free grammar
gramática regular
regular grammar

intersección
intersection

jerarquía de Chomsky
Chomsky's hierarchy
lema de bombeo
pumping lemma
lenguaje
language

lenguaje de programación esencial
bare-bones programming language
lenguaje generado por una gramática regular
language generated by a regular grammar
lenguaje vacío
empty language
lenguajes aceptados por máquinas de Turing
Turing-acceptable languages
lenguajes decidibles por máquinas de Turing
Turing-decidable languages
lenguajes enumerables recusivamente
recursively enumerable lenguages
lenguajes estructurados por frases
phrase-structure languages
lenguajes formales
formal languages
lenguajes independientes del contexto
context-free languages
lenguajes deterministas independientes del contexto
deterministic context-free languages
lenguajes naturales
natural languages
lenguajes no estructurados por frases
non-phrase-structure languages
lenguajes recursivos
recursive languages
lenguajes regulares
regular languages
leyes de DeMorgan
DeMorgan's Laws
literales
literals

máquina de Turing
Turing machine
máquina de Turing autoterminante
self-terminating Turing machine
máquina de Turing de varias cintas
multiple-tape Turing machine
máquina de Turing universal
universal Turing machine
marca de fin de cadena
end-of-string marker
mecanismo de control
control mechanism
método axiomático
axiomatic method
miembros de un conjunto
members of a set
minimalización
minimalization
modus ponens
modus ponens

no terminal en una gramática
nonterminal in a grammar
notación *O*
O notation
NP
NP
***NP*-completo**
NP-complete
número cardinal
cardinal number
números naturales (ℕ)
natural numbers (ℕ)

operación de desplazamiento en un analizador sintáctico *LR(k)*
shift operation in a *LR(k)* parser
operación de reducción en un analizador sintáctico *LR(k)*
reduce operation in a *LR(k)* parser
orden lineal
linear order

palíndromo
palindrome
paso de un cómputo de una máquina de Turing
step of a Turing machine computation
pila
stack
planificación de vista serializables
view serializable schedule
principio de preanálisis
lookahead principle
problema de decisión del viajante
traveling salesman decision problem
problema de la cobertura exacta
exact covering problem

problema de decisión del vendedor ambulante
traveling salesman decision problem
problema de la parada
halting problem
problema del circuito de Hamilton
Hamilton circuit problem
problema soluble
solvable problem
problema soluble por máquinas de Turing
Turing-solvable problem
problemas de decisión
decision problems
proceso computacional
computational process
proceso de traducción
translation process
producto cartesiano
cartesian product
programa fuente
source program
programa objeto
object program

razonamiento deductivo
deductive reasoning
recursividad primitiva
primitive recursion
red de transiciones
transition network
reducción polinómica
polynomial reduction
regla λ
λ-rule
regla de reescritura
rewrite rule
regla de reescritura lineal por la derecha
right linear rewrite rule
regla de reescritura lineal por la izquierda
left linear rewrite rule
reglas de inferencia
inference rules
relación de equivalencia
equivalence relation

SAT
SAT
SAT-3
3-SAT
satisfactibilidad
satisfiability
secuencia de configuración de una máquina de Turing
configuration sequence of a Turing machine
secuencia de cruce
crossing sequence
símbolo de inicio de una gramática
start symbol, of a grammar
símbolo (componente léxico) especial
special token
símbolos de cinta
tape symbols
símbolos de pila
stack symbols
subconjunto
subset
subconjunto propio
proper subset
suposición de costo uniforme
uniform cost assumption

tabla de análisis sintáctico
parse table
tabla de transiciones
transition table
tasa de crecimiento
rate of growth
teorema
theorem
teorema de Cook
Cook's theorem
teorema de la aceleración de Blum
Blum's speedup theorem
teorema de la incompleción
incompleteness theorem
teorema de Schröder-Bernstein
Schröder-Bernstein theorem
teoría de conjuntos
set theory
teoría de funciones recursivas
recursive function theory
teoría de grafos
graph theory

teoría de la complejidad
complexity theory
teoría de la relatividad de Einstein
Einstein's theory of relativity
teoría de números
number theory
terminación anormal de una máquina de Turing
abnormal termination of a Turing machine
terminal en una gramática
terminal in a grammar

tesis de Church
Church's thesis
tesis de Church-Turing
Church-Turing thesis
tesis de Turing
Turing's thesis
transición λ
λ - transition
transición de estado
state transition

unión
union

En esta obra se presentan los fundamentos de la teoría de la computación en un formato accesible para los estudiantes universitarios, La intención del autor ha sido presentar estas ideas como la base para la resolución de problemas reales, en vez de concebirlas como abstracciones de difícil aplicación.

Hay dos características que hacen de esta obra un texto único: el énfasis en la relación de la teoría con la práctica y la cobertura concisa (se tocan sólo los temas indispensables para un curso introductorio). El estudiante comprende en su real magnitud cada aspecto teórico porque se presenta aplicado a un uso real y, por otro lado, no se extiende en detalles que pueden desviarlo de la comprensión general del tema.

Los resúmenes de capítulo, problemas, ejercicios y figuras son ayudas didácticas de gran utilidad. Por todo esto, la obra del doctor Brookshear es una gran alternativa para los cursos introductorios de teoría de la computación, lenguajes formales, autómatas y complejidad, en los que tradicionalmente se han usado textos áridos, de escasa cobertura o poca claridad.

OTRAS OBRAS DE INTERÉS PUBLICADAS
POR **ADDISON-WESLEY IBEROAMERICANA:**

AHO, HOPCROFT y ULLMAN: *Estructuras de datos y algoritmos* (64024)
AHO, SETHI y ULLMAN: *Compiladores. Principios, técnicas y herramientas* (62903)
FOLEY: *Introducción a la programación con Turbo Pascal* (60123)
SALMON: *Estructuras y abstracciones. Introducción a la computación y a la programación con Turbo Pascal* (60134)
SETHI: *Lenguajes de programación* (51858)
STROUSTRUP: *El lenguaje de programación C++, segunda edición* (60104)

ADDISON-WESLEY IBEROAMERICANA
Billinghurst 897 PB-A, Buenos Aires 1174, Argentina
Ave. Brigadeiro Luis Antonio 2344, Conjunto 114, São Paulo 01402, São Paulo, Brasil
Casilla 70060, Santiago 7, Chile
Apartado Aéreo 241-943, Santa Fé de Bogotá, Colombia
Espalter 3 bajo, Madrid 28014, España
7 Jacob Way, Reading, Massachusetts 01867, E.U.A.
Apartado Postal 22-012, México D.F. 14000, México
Apartado Postal 29853, Río Piedras, Puerto Rico 00929
Apartado Postal 51454, Caracas 1050-A, Venezuela