Fundamentos de informática

Gregorio Fernández y
Fernando Sáez Vacas

Fundamentos de informática

Alianza
Editorial

Calle Milán, 38; 28043 Madrid; teléf. 2000045
ISBN: 84-206-8604-2
Depósito legal: M. 18.709-1987
Fotocomposición EFCA, S. A., Dr. Federico Rubio y Galí, 16; 28039 Madrid
Impreso en Hijos de E. Minuesa, S. L., Ronda de Toledo, 24; 28005 Madrid
Printed in Spain

INDICE GENERAL

PROLOGO

Este libro trata exactamente de lo que dice su título.

Como primera impresión, a algunas personas puede parecerles innecesario publicar un texto sobre fundamentos en una época en que la informática ha llegado en su difusión casi a formar parte material del mobiliario hogareño. Además de este efecto distributivo sobre la sociedad, los espectaculares progresos tecnológicos han producido un crecimiento desbordante de la especialización. Los sistemas operativos, las bases de datos, los lenguajes concurrentes, la programación lógica, la inteligencia artificial, la arquitectura de ordenadores, las redes, las herramientas de ayuda para ingeniería de software y tantas otras más específicas e instrumentales son áreas de trabajo o técnicas que por sí solas requieren esfuerzos considerables por parte de quienes pretenden estudiar y seguir su evolución.

Nadie discute que la especialización es inevitable. También es verdad que los inconvenientes que genera su abuso dentro de muchos de los dominios de aplicación de la informática son importantes y cada día somos más los que pensamos que aquéllos tienden a crecer en la misma proporción en que el especialista ignora la perspectiva global donde se integra su particular disciplina. Con mayor razón, si sucede que además desconoce las raíces de sus técnicas profesionales. Es decir, la proliferación de especialistas demasiado pragmáticos contribuye a crear una situación que comienza a ser ya preocupante y contra la que se han alzado voces diversas. Esas voces se muestran unánimes en apelar a la necesidad de construir o reconstruir una formación más sólida y básica en informática, dirigida como mínimo a aquéllos que se sienten deseosos de comprender profundamente su profesión o de implicarse en un papel técnico significativo con independencia de la especialidad escogida (por no hablar de la lucha contra la obsolescencia técnica que acecha a todos cuantos trabajamos en este campo).

El libro que presentamos responde a esta llamada, puesto que se ocupa de algunas

de las raíces o fundamentos de la informática. Lo hemos escrito pensando en los estudiantes universitarios de las ramas de informática, así como en los profesionales antes mencionados. Estos últimos encontrarán un texto autocontenido, desprovisto en lo posible del aparato teórico habitual y preocupado permanentemente en la tarea de desarrollar aperturas a cuestiones de la más viva actualidad, como los sistemas borrosos o la complejidad del software, y a cuestiones en las que parece vislumbrarse un futuro. En cuanto a los estudiantes, nuestra experiencia nos dice que, por un cúmulo de circunstancias que no hacen al caso, se ven obligados con frecuencia a estudiar las materias objeto de nuestro libro, tal vez, sí, con mayor extensión y formalismo matemático, pero no siempre bajo condiciones óptimas: apuntes improvisados, textos en lenguas extranjeras, dispersión de estas mismas materias en distintas asignaturas y por tanto fragmentación de su sentido radical (de raíces), o desapego del sentido de su aplicación. Sin poner en tela de juicio la necesidad científica del mejor formalismo posible, está constatado que dosis excesivas y exclusivas de esa medicina conducen en el plano educativo a un estéril desánimo de los estudiantes.

Consta el libro de cuatro partes, centradas en cuatro de los pilares básicos sobre los que se sustenta el edificio teórico y práctico de la informática. Estos son la lógica, la teoría de autómatas, la teoría de la computación y la teoría de los lenguajes formales. Por razón del enfoque adoptado, mixto entre teoría y práctica, y por razones de concisión, aquí los hemos enmarcado bajo los escuetos rótulos de «Lógica», «Autómatas», «Algoritmos» y «Lenguajes». Para las referencias cruzadas entre partes del texto se decidió utilizar el vocablo *tema* en lugar de *parte*. Así, mencionaremos el apartado tal del capítulo cual del tema de «Lógica», y no de la *Primera Parte* (por ejemplo).

A muy grandes rasgos, podemos esbozar el orígen o motivación inicial de cada área: filosófico (naturaleza del razonamiento humano) para la lógica, tecnológico (formalización del diseño de circuitos lógicos) para los autómatas, matemático (determinación de lo que puede o no computarse, y del grado de complejidad de las computaciones) para la teoría de la computación, lingüísticos (estudio científico del lenguaje natural) para la teoría de lenguajes. El paso del tiempo ha ido haciendo emerger nuevas y múltiples relaciones entre dichas áreas y de ellas con la gran mayoría de los desarrollos técnicos de la informática, por lo que su cabal conocimiento añade al dividendo reconocido de los buenos fundamentos la promesa de su permanente rentabilidad.

Hemos intentado dotar al libro de determinadas características, que a continuación resumiremos. En primer lugar, dos propiedades dignas de mención son por un lado su carácter básico y su carácter señalizador. El adjetivo *básico* debe tomarse en su acepción de *fundamental* y no de *elemental,* ya que estamos presuponiendo en poder del lector los conocimientos equivalentes a una formación introductoria sólida sobre generalidades y estructura de ordenadores y sobre programación, más un lenguaje de alto nivel.

Por el término *señalizador* queremos aludir a la incorporación de numerosos apartados que introducen (a menudo en forma pormenorizada) a cuestiones directa o indirectamente relacionadas con técnicas informáticas muy actuales o con caminos científicos que se están abriendo. En pocas palabras, buscamos comprometer —en la medida de lo factible— el interés del lector, y lo que queremos significar con este

objetivo se comprenderá mejor cuando entremos dentro de unos momentos a relacionar algunos aspectos de los contenidos del libro.

Nuestra esperanza, sin embargo, reside en que su propiedad más notoria sea la pedagógica, pues no en balde este libro ha tenido una vida anterior en la que a lo largo de tres ediciones consecutivas y varias reimpresiones ha servido de texto durante ocho o nueve años en la Escuela Técnica Superior de Ingenieros de Telecomunicación de Madrid, donde ambos autores somos profesores. Era el segundo de una obra en dos volúmenes titulada genéricamente Fundamentos de los Ordenadores y editada por dicha Escuela.

Sobre ese material el presente libro contiene aproximadamente un cuarenta por ciento de cuestiones nuevas, lo que ha llevado a eliminar bastantes de las anteriores y a remodelar el sesenta por ciento restante. Aunque sus cualidades didácticas tendrán que ser juzgadas por los lectores, en lo que a nosotros concierne hemos puesto el máximo cuidado en estructurar y redactar los textos con la mayor claridad de que hemos sido capaces, y utilizando tanto cuanto hemos creído conveniente la ayuda de los ejemplos. Al formato también le hemos dado importancia. Y es así que todos los temas se organizan idénticamente por capítulos, y todos los capítulos, menos el primero, poseen la misma estructura, pues cualquiera que sea su desarrollo terminan con un apartado de Resumen, un apartado llamado de Notas Histórica y Bibliográfica y (siempre que procede) un apartado de Ejercicios. El primer capítulo de cada tema es especial, ostenta el nombre de Ideas Generales y la misión de exponer un planteamiento panorámico del conjunto de los capítulos y de sus relaciones con los otros tres temas. Al final del libro una bibliografía ordenada alfabéticamente recoge exclusivamente las referencias citadas en las Notas de todos los capítulos, por lo que huelga decir que, dado el enfoque primordialmente orientador de las Notas, la bibliografía no pretende alcanzar ninguna meta de exhaustividad científica.

Por lo general, de los contenidos de un libro poco cabe decir en un prólogo, ya que su índice normalmente excusa de tal esfuerzo, salvo que se entienda que no ha de resultar ocioso captar la atención del lector acerca de diversas singularidades. Haremos de éstas un bosquejo rápido.

Refiriéndonos a la primera parte, es del común saber la naturaleza formativa de la lógica formal para el razonamiento, pero en el campo de la informática destaca además su naturaleza fundamentadora para numerosas aplicaciones de inteligencia artificial, de programación y de diseño de circuitos. Este segundo aspecto es el que hemos enfatizado en el tema de «Lógica», buscando, por ejemplo, culminar la exposición de las lógicas proposicional y de predicados de primer orden en el desarrollo de sistemas inferenciales, con la vista puesta en los sistemas expertos (dominio aplicado y en auge de la inteligencia artificial), a los que se dedica asimismo un amplio capítulo. Otro nos habla de Otras Lógicas que se prefiguran en el horizonte como prometedoras bases para los mencionados dominios aplicativos: ampliaciones de la lógica de predicados (lógicas modal y temporal), lógicas multivaloradas y, en especial, un extenso desarrollo de la lógica borrosa. Una cuestión clásica como los circuitos combinacionales tiene aquí su capítulo, lo mismo que en el tema de «Autómatas» lo tienen los circuitos lógicos secuenciales.

Con respecto a la parte llamada «Autómatas», que versa sobre un área de expresión típicamente matemática, hemos suavizado mucho su formalismo descripti-

vo (a fin de cuentas, los autómatas de los que nos ocupamos son construcciones formales). No solamente bastantes teoremas se enuncian y se explican sin demostrarlos, también se usan ejemplos prácticos sencillos, entre los que, por ilustrar, citamos el detector de paridad, el sumador, el problema del castillo encantado, el contador, el reconocedor de una cadena de símbolos, etc. Como una muestra, entre otras, de nuestro declarado objetivo de interconectar los cuatro temas del libro está el capítulo dedicado a Autómatas Reconocedores y Lenguajes Regulares, tradicional por otro lado en el área de estudio de los autómatas. Menos tradicional en un libro sobre principios básicos es aquel otro capítulo destinado a introducir Otros Autómatas, a saber: las redes de Petri, que han adquirido considerable predicamento en el ámbito del diseño de sistemas concurrentes, los autómatas estocásticos, los autómatas de aprendizaje y los autómatas borrosos.

En «Algoritmos» se establecen con cuidado definiciones de los conceptos de algoritmo, programa y máquina y sus profundas interrelaciones. Se ha tomado un esmero especial para que los lectores comprendan bien el análisis y diseño de máquinas de Turing, así como el significado de piedra angular de esta máquina en la teoría informática, en conexión con las cuestiones de la computabilidad. Por último, un capítulo sobre Complejidad intenta exponer las nociones esenciales que, en nuestra opinión, todo informático debe poseer sobre un campo teórico y práctico de gran desarrollo en los últimos años. La complejidad algorítmica se aplica en dominios tan diversos como la robótica, el diseño de circuitos integrados a muy grande escala y el diseño de estructuras de datos eficientes.

Los lenguajes formales en general, y con un mayor detalle los lenguajes libres de contexto y las relaciones entre los diversos tipos de lenguaje y las estructuras de máquina capaces de reconocerlos, constituyen el contenido de los cuatro primeros capítulos de la última parte del libro. Es ésta un área obligada si se quiere comprender algo sobre la esencia de los lenguajes para ordenador. Precisamente, el extenso capítulo 5, *Aplicaciones a los lenguajes de programación,* intenta salvar un poco la brecha explicativa con la que nos encontramos cotidianamente entre la teoría de los lenguajes formales y su materialización en las herramientas que todo el mundo utiliza para construir el software. De las diversas cuestiones tratadas en ese capítulo, tal vez puedan destacarse por su interés sistemático las referidas a sintaxis, semántica y procesadores de lenguaje. Como sección incitadora de novedades y de futuro es de subrayar el apartado acerca de los lenguajes de la programación declarativa (programación funcional, programación lógica, etc.).

Nuestros alumnos de la rama de Informática de la Escuela se encontrarán a partir de este momento con un texto completamente nuevo. Comoquiera que hace dos años no se reimprimía aquél del que éste es una reencarnación, ellos se van a llevar una alegría, acaso un tanto atemperada por causa de los gajes del oficio de tener que estudiárselo. Al mismo tiempo, estamos razonablemente seguros de que también se alegrarán (con nosotros) de saber que desde ahora podrán compartir esta fuente de conocimiento con otros estudiantes y con la vasta comunidad de profesionales informáticos en el ámbito de la lengua castellana. Que así sea.

Los autores
Febrero, 1987

Primera parte
LOGICA

Capítulo 1
IDEAS GENERALES

1. LÓGICA E INFORMÁTICA

Alfredo Deaño define la Lógica como «la ciencia de los principios de la inferencia formalmente válida». Como tal, constituye una herramienta para el análisis de los razonamientos o argumentaciones generados por la mente humana. Pero, si es así, ¿qué relación puede haber entre la lógica y las técnicas de la informática? Pues bien, podemos encontrar, al menos, los siguientes puntos de cóntacto:

(a) *En aplicaciones «inteligentes»*. Los ordenadores son máquinas diseñadas para mecanizar trabajos intelectuales. Normalmente, esos trabajos son los relacionados con tareas sencillas y rutinarias: cálculos basados en operaciones aritméticas (que el hombre aprende de memoria y aplica sin necesidad de razonar), búsqueda de datos (por simple comparación o emparejamiento con una clave dada), clasificación (ordenación de datos basada en un criterio elemental), etc. Para estas aplicaciones no hace falta aplicar la lógica (en el sentido de ciencia; naturalmente que todo informático, como todo ser racional, utiliza razonamientos lógicos de manera informal). Si pretendemos mecanizar tareas más complejas (inducción, deducción, etc.) entramos en el campo de la informática, llamado «inteligencia artificial». (En realidad, se trata de un campo interdisciplinar, que interesa también a la psicología y a la epistemología; aquí hablamos de «aplicaciones inteligentes de los ordenadores», y, por tanto, nos referimos a la vertiente más ingenieril). Ahora se trata de conseguir que la máquina sea capaz de hacer, precisamente, esos razonamientos que, de manera informal, realiza el hombre. Y para ello es preciso analizarlos, definirlos con precisión, en una palabra, formalizarlos. Y eso es precisamente lo que hace la lógica (formal).

(b) *En programación*. Hace ya años que preocupa la llamada «crisis del software»: los programas son cada vez más complejos, menos fiables y más difíciles de mantener. Se han propuesto y se utilizan diversas metodologías para la construcción de progra-

mas, basadas en principios teóricos de los que se dará una idea en el tema «Algoritmos». Pero un enfoque complementario de tales metodologías es el de buscar una programación «declarativa» en lugar de «imperativa». Esto quiere decir que se trataría de buscar lenguajes de programación tales que los programas no sean una secuencia de instrucciones que le digan al ordenador, paso a paso, cómo hay que resolver el problema, sino una especificación de qué es lo que se pretende resolver, y que sea el propio ordenador quien determine las acciones necesarias para ello. Pues bien, la lógica puede verse, precisamente, como un lenguaje de especificación mediante el cual podemos plantear los problemas de manera rigurosa. En el tema «Lenguajes» comentaremos los principios de la programación lógica. Por otra parte, la tarea de programación se complica cuando se hace preciso considerar procesos concurrentes e intercomunicantes (por ejemplo, en los sistemas operativos de multiprogramación, en las aplicaciones de «tiempo real», o en los sistemas distribuidos); ciertas extensiones de la «lógica clásica», como las lógicas modal y temporal que veremos en el capítulo 5, son herramientas matemáticas adecuadas para estos casos.

2. LÓGICA Y CIRCUITOS DE CONMUTACIÓN

Desde el punto de vista estrictamente electrónico, el soporte tecnológico principal de los ordenadores lo constituyen los circuitos de conmutación, o «circuitos lógicos». Esta última denominación es, desde luego, discutible. Porque los circuitos electrónicos básicos, con la tecnología actual, no permiten realizar directamente las operaciones inferenciales que caracterizan a la lógica. (Aunque indirectamente sí, ya que podemos, por ejemplo, programar a un ordenador para que las haga, e, incluso, podemos pensar en nuevas estructuras de ordenador cuyas operaciones elementales no sean ya las instrucciones de máquina de los ordenadores actuales, sino inferencias lógicas. Este es un campo de la arquitectura de ordenadores sometido a intensa investigación). Hay, no obstante, motivos para llamar «lógicos» a tales circuitos. Motivos que pueden reducirse a una consideración matemática: el hecho de que las formas elementales de la lógica y los circuitos de conmutación tienen un modelo matemático común: el álgebra de Boole binaria. Dada la importancia de los circuitos lógicos como componentes, y aunque no tengan una relación demasiado estrecha con el resto de las aplicaciones informáticas de la lógica, dedicaremos un capítulo de este tema a su estudio.

3. LÓGICA, LENGUAJES Y AUTÓMATAS

Un lenguaje es un sistema de símbolos y de convenios que se utiliza para la comunicación, sea ésta entre personas, entre personas y máquinas o entre máquinas. El estudio matemático de los lenguajes es uno de los pilares de la informática, y a él vamos a dedicar parte del tema «Autómatas» y todo el último tema del libro. Pero la lógica formal también puede considerarse como un lenguaje (aunque éste es un tema debatido filosóficamente), «el mejor hecho de los lenguajes», como dice Ferrater Mora.

Por otra parte, existe una correspondencia muy estrecha, como veremos en los temas correspondientes, entre los autómatas y los lenguajes: a cada tipo de autómata corresponde un tipo de lenguaje, y viceversa. La máquina de Turing es una máquina lógica, un autómata de un tipo especial, que ha permitido formalizar el concepto de algoritmo y ha proporcionado una de las aproximaciones teóricas al campo de la computabilidad, como se verá en el tema «Algoritmos».

Puesto que la terminología asociada a la teoría de lenguajes va a utilizarse en todas las partes de este libro, nos parece conveniente presentar ya las definiciones y los conceptos básicos.

4. LENGUAJES FORMALES: DEFINICIONES Y CONCEPTOS BÁSICOS

4.1. Alfabeto, cadenas y lenguaje universal

Llamaremos *alfabeto* a un conjunto finito, no vacío, de símbolos.

Una *cadena* (o palabra, o expresión, o secuencia finita) es una secuencia ordenada, finita, con o sin repetición, de los símbolos de un alfabeto. De una manera general, utilizaremos las letras x, y, z para representar cadenas construidas con cualquier alfabeto (que, por supuesto, no puede contener esos símbolos). A veces interesa definir la *cadena vacía* como aquella que no contiene ningún símbolo. Para designarla utilizaremos el símbolo especial «λ» (en el supuesto, naturalmente, de que este símbolo no forme parte del alfabeto con el que estamos construyendo las cadenas).

Sobre el conjunto de cadenas que pueden construirse con un determinado alfabeto puede definirse una función total, llamada *longitud,* y abreviada «lg», que asigna a cada cadena el número de símbolos de que consta. Por ejemplo, si el alfabeto es $\{0, 1\}$, entonces $\lg(0) = 1$, $\lg(1) = 1$, $\lg(00) = 2$, $\lg(01) = 2$, etc. Por definición, $\lg(\lambda) = 0$.

Si A es un alfabeto, llamaremos A^n al conjunto de todas las cadenas de longitud n. Por ejemplo, si $A = \{a, b\}$, entonces,

$$A^0 = \{\lambda\};\ A^1 = \{a, b\};\ A^2 = \{aa, ab, ba, bb\},\ \text{etc.}$$

Llamaremos *lenguaje universal* sobre un alfabeto A, y lo representaremos por A^*, al conjunto infinito

$$A^* = A^0 \cup A^1 \cup A^2 \cup \ldots = \bigcup_{i=0}^{\infty} A^i$$

Es decir, A^* es el conjunto de todas las cadenas (incluida λ) que pueden formarse a partir de A.

4.2. Concatenación

La *concatenación* de dos cadenas, $x, y \in A^*$, es una ley de composición interna sobre A^*, es decir, una aplicación $A^* \times A^* \to A^*$, que consiste en formar la cadena xy poniendo x delante de y.

Por ejemplo, si $A = \{a, b, c\}$ y consideramos las cadenas $x = ab$, $y = bba$, entonces $xy = abbba$ y $yx = bbaab$.

La representación de la concatenación de una cadena consigo misma puede abreviarse mediante notación exponencial: $x^0 = \lambda$; $x^1 = x$; $x^2 = xx$; $x^3 = xxx$, etc.

(Obsérvese, aunque sea a título anecdótico, que la longitud cumple respecto a la concatenación las mismas propiedades formales que el logaritmo respecto a la multiplicación: $\lg(xy) = \lg(x) + \lg(y)$; $\lg(x^n) = n \cdot \lg(x)$).

La operación de concatenación satisface las siguientes propiedades:

a) Es asociativa: $x(yz) = (xy)z = xyz$.
b) En general, no es conmutativa: $xy \neq yx$. (La conmutatividad sólo se da en el caso particular de que A conste de un solo elemento).
c) Tiene un elemento neutro, λ: $\lambda x = x\lambda = x$.
d) Ninguna cadena (salvo λ) tiene inverso: dada $x \in A$ $(x \neq \lambda)$, no existe ningún $y \in A$ tal que $xy = \lambda$.

Por consiguiente, $\langle A^*, \rangle$, es decir, A^* con la operación de concatenación, es una estructura algebraica de tipo monoide, a la que se llama *monoide libre generado por* A.

También es posible prescindir de la cadena vacía, y, en lugar de con A^*, trabajar con $A^+ = A^* - \{\lambda\}$. Algebraicamente, la diferencia estaría en que, al no existir elemento neutro, en lugar de hablar de monoide habríamos de hablar de semigrupo.

4.3. Lenguaje y metalenguaje

Dado un alfabeto A, todo subconjunto de A^* se llama *lenguaje* sobre A. Esta definición es rigurosamente válida, pero de poca utilidad. Porque en la aplicación del concepto de lenguaje (ya sea a asuntos prácticos, como en los lenguajes de programación, o teóricos, como en la lógica formal) lo que interesa es poder resolver dos problemas duales: dado un lenguaje, *generar* cadenas que pertenezcan a ese lenguaje, y dada una cadena, *reconocer* si pertenece o no a un determinado lenguaje. Si el lenguaje en cuestión es finito (es decir, consta de un número finito de cadenas), la solución de ambos problemas es inmediata, puesto que el lenguaje se describe mediante la simple enumeración de sus cadenas. Pero lo normal es que el lenguaje sea infinito, y entonces tendremos que dar una *descripción finita* para poder resolverlos.

Y esto nos lleva al concepto de metalenguaje, porque dicha descripción finita tendrá que hacerse utilizando un sistema de símbolos que no pueden ser los mismos del lenguaje que trata de describir. Por tanto, tal descripción habrá de hacerse en otro lenguaje, que, como sirve para describir al primero (*lenguaje objeto*), se llama *metalenguaje*. Por ejemplo, en el lenguaje del álgebra elemental, escribimos expresiones como «$x + 7 = 17$»; cuando le explicamos a alguien lo que significan expresiones

de ese tipo, estamos utilizando el español como metalenguaje para describir el lenguaje del álgebra. Cuando hablamos de propiedades del lenguaje natural, español en nuestro caso, solemos utilizarlo indistintamente como lenguaje objeto o como metalenguaje. Por ejemplo, si decimos «el ordenador no es más que una máquina», hemos construido una frase del lenguaje en la que se *usa*, entre otras, la palabra «ordenador». Y si decimos «'ordenador' es un vocablo de cuatro sílabas» también hacemos una frase, pero en ella, la palabra «ordenador» no se «usa», se *menciona*. En realidad, estamos acudiendo al español como metalenguaje para describir una cierta propiedad del propio lenguaje. Y puede haber una jerarquía de lenguajes, es decir, varios niveles de descripción. Así, una frase del «metametalenguaje» sería: "«'ordenador' es un vocablo de cuatro sílabas» es un enunciado verdadero".

La descripción de un lenguaje artificial suele hacerse mediante unas reglas que permiten generar cadenas pertenecientes al lenguaje («cadenas o expresiones válidas», o «sentencias», o «frases»). Son las «reglas de formación» que veremos en este tema, o las «reglas de escritura» que definiremos en el tema «Lenguajes». También, para ciertos tipos de lenguajes es posible dar descripciones algebraicas, como veremos en el tema «Autómatas» para los lenguajes llamados regulares.

4.4. Sintaxis, semántica y pragmática

En la lingüística hay tres campos: la sintaxis, la semántica y la pragmática. La sintaxis se ocupa estrictamente de los símbolos y de la manera de combinarlos para obtener sentencias del lenguaje. La semántica estudia los símbolos y las sentencias en relación con los objetos que designan. La pragmática, finalmente, trataría (en realidad, es el nivel menos desarrollado, y roza ya con los campos de la psicología y la sociología) de las relaciones entre los símbolos y los sujetos que los usan.

Un ejemplo de enunciado perteneciente a la sintaxis es:

> *'algoritmo' es un sustantivo.*

Un ejemplo de enunciado perteneciente a la semántica es:

> *no es cierto que 'algoritmo' derive del griego 'αλγοζ' ('dolor') y 'αριθμοζ' ('número') y signifique 'dolor producido por los números'.*

Finalmente,

> *en otro tema veremos la etimología correcta de la palabra 'algoritmo'.*

es un enunciado perteneciente a la pragmática.

En el primer caso se ha usado 'algoritmo' para relacionarlo con otra expresión ('sustantivo'); en el segundo, para relacionarlo con el objeto que designa (en este caso es más bien el objeto que no designa), y en el tercero, para relacionarlo con quienes lo usan (nosotros).

5. RESUMEN

La lógica formal es uno de los fundamentos teóricos de la informática, que, además, tiene importancia práctica en las aplicaciones llamadas de «inteligencia artificial», en los lenguajes de programación declarativos y en la programación concurrente.

Los conceptos de alfabeto, cadenas, concatenación, lenguaje, sintaxis y semántica se utilizarán en éste y en otros temas del libro.

Los circuitos lógicos, sin otra relación con la lógica formal que la existencia de un modelo matemático común, son los componentes tecnológicos básicos de los ordenadores actuales.

Capítulo 2
LOGICA DE PROPOSICIONES

1. INTRODUCCIÓN

1.1. Variables proposicionales y sentencias

Los lógicos clásicos distinguían entre *juicio* (el acto mental mediante el cual se piensa o se concibe algún hecho elemental), *proposición* (lo pensado o concebido en ese acto mental) y *razonamiento* (combinación, con una forma determinada, de proposiciones enlazadas por construcciones primitivas, como «o», «y», «si... entonces», etc.).

El concepto de proposición se formaliza mediante el de *variable proposicional*. Las variables proposicionales representan *enunciados declarativos*, es decir, frases expresadas en el modo gramatical indicativo. (La formalización de modos subjuntivo, condicional e imperativo exige otros tipos de lógica). Para escribir las variables proposicionales utilizaremos las letras *p, q, r, s, t...*, eventualmente con subíndices.

Una *sentencia* representa a un enunciado compuesto por enunciados elementales y esas construcciones primitivas («y», «o», etc.), que se formalizan mediante las llamadas *conectivas*. Este enunciado compuesto puede ser un «razonamiento» en el sentido clásico, o no.

1.2. Conectivas

La conectiva unaria (o monádica) de *negación*, «no», cuyo efecto es negar lo que dice el enunciado que le sigue, se representa mediante el símbolo «¬» antepuesto a la variable proposicional o a la sentencia correspondiente. Así, si con «*p*» estamos formalizando «la nieve es blanca», «¬ *p*» representaría «la nieve no es blanca». (En otros libros se utilizan otros símbolos: «~ *p*», o «$\bar{p}$», o «*p'*»).

Las conectivas binarias (o diádicas) son las que enlazan entre sí a dos variables proposicionales o a dos sentencias. Las más utilizadas son:

- La *conjunción*, con símbolo «$\wedge$» (en otros libros, «&» o «$\cdot$»), que representa el «y» del lenguaje natural.
- La *disyunción*, con símbolo «$\vee$» (en otros libros, «|» o «+»), que representa el «o» (en sentido inclusivo: «$p \vee q$» significa «o bien p, o bien q, o bien ambos»).
- El *condicional*, con símbolo «$\rightarrow$» (en otros libros, «$\Rightarrow$» o «$\supset$»), que representa el «si ... entonces». De las dos variables enlazadas por el condicional, la primera (la de la izquierda) se llama *antecedente* y la segunda, *consecuente*.
- El *bicondicional*, con símbolo «$\leftrightarrow$» (en otros libros, «=» o «$\equiv$»), que representa el «si y sólo si».

Combinando de forma adecuada variables y conectivas se forman sentencias, o cadenas correctas en el lenguaje de la lógica de proposiciones.

1.3. Interpretación binaria de variables proposicionales y de sentencias

Aunque en el apartado dedicado a semántica definiremos con mayor rigor el concepto de interpretación, vamos a introducirlo ya, restringido al caso particular (y más frecuente) de la *interpretación binaria*. Una interpretación binaria consiste en asignar a cada una de las variables proposicionales uno de entre dos valores: «verdadero» (o «cierto») o «falso». En lo sucesivo, utilizaremos el símbolo «1» para la representación de «verdadero» y «0» para la de «falso». Dada una interpretación de las variables proposicionales que intervienen en una sentencia, para poder dar una interpretación a la sentencia, es decir, para poder decir si la sentencia es verdadera (valor 1) o falsa (valor 0), tenemos que dar un significado a las conectivas. El significado universalmente admitido es el que se resume en la siguiente tabla:

	p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
i_0:	0	0	1	0	0	1	1
i_1:	0	1	1	0	1	1	0
i_2:	1	0	0	0	1	0	0
i_3:	1	1	0	1	1	1	1

Cada una de las filas de la tabla corresponde a una interpretación de las variables proposicionales p y q. En el caso binario sólo hay cuatro interpretaciones distintas de dos variables proposicionales: ambas falsas ($i_0(p) = 0$, $i_0(q) = 0$), una verdadera y otra falsa ($i_1(p) = 0$, $i_1(q) = 1$; $i_2(p) = 1$, $i_2(q) = 0$), o ambas verdaderas ($i_3(p) = 1$, $i_3(q) = 1$). Para cada una de estas interpretaciones, la tabla muestra la interpretación de la sentencia formada mediante la negación de una variable proposicional (columna de $\neg p$) o mediante la unión de dos variables proposicionales con una conectiva binaria (columnas siguientes). Vamos a comentar estas interpretaciones:

• *Negación*. Parece natural que si una variable proposicional representa un hecho verdadero, su negación sea un hecho falso, y viceversa. Por ejemplo, si p representa «la nieve es blanca», y la interpretamos como verdadera, $\neg p$ representará «la nieve no es blanca», y se interpretará como falsa.

• *Conjunción*. También parece natural que la conjunción de dos hechos sólo sea un hecho verdadero si lo es cada uno de ellos separadamente: si $p \wedge q$ representa «la nieve es blanca y me apetece esquiar», $p \wedge q$ sólo es verdadero si, además de serlo p, verdaderamente me apetece esquiar.

• *Disyunción*. Como ya se ha dicho más arriba, la interpretación que se da a la disyunción de dos variables proposicionales es la inclusiva: basta con que una de ellas sea verdadera para que su disyunción lo sea. Es decir, lo que representa es «o bien p, o bien q, o ambos». Por tanto, no sirve para representar enunciados como «o viajo en avión o viajo en tren (pero no ambos)». (La representación de este enunciado, en el que se usa la disyunción exclusiva, sería, con las conectivas que tenemos: $(p \wedge \neg q) \vee (\neg p \wedge q)$, es decir, traduciendo al lenguaje natural: «una de dos: o bien viajo en avión (p) y entonces no viajo en tren, o bien viajo en tren (q), y entonces no viajo en avión»).

• *Condicional*. Como puede verse en la tabla, la interpretación de dos variables enlazadas por el condicional es tal que la sentencia siempre es verdadera salvo en el caso de que el antecedente sea verdadero y el consecuente falso. Este sentido del condicional requiere una explicación algo más detallada. Para ello, analicemos cada uno de los casos con algunos ejemplos:

a) $p = 1$, $q = 1$ (antecedente y consecuente ciertos). Parece evidente que en tal caso el condicional («si p, entonces q») deberá ser cierto. Así,

> 'Si como mucho, entonces engordo'

es un condicional cierto en el caso de que tanto el antecedente («como mucho») como el consecuente («engordo») lo sean. Pero piénsese que también pueden ser ciertos condicionales en los que no haya una relación causa-efecto entre antecedente y consecuente; así:

> 'Si Madrid es la capital de España, entonces París es la capital de Francia'

es un condicional verdadero. Por ello, aunque a veces, en lugar de decir «condicional» se diga «implicación» (o «implicación material»), no hay que confundirlo con la llamada «implicación estricta»:

> 'Madrid es la capital de España' implica que 'París es la capital de Francia'

es una implicación falsa. La implicación estricta se estudia en la lógica modal, una ampliación de la lógica «clásica» que comentaremos en el capítulo 5 (apartado 2.4).

b) $p = 1$, $q = 0$. En este caso parece natural decir que el condicional es falso. En efecto, decir «si p, entonces q» viene a ser equivalente a decir «si p es cierto, entonces

q es cierto», o, lo que es lo mismo, «p es condición suficiente (aunque no necesaria) de q»; por tanto, el hecho de que p sea cierto y q falso viene a refutar la expresión «$p \rightarrow q$», es decir, a hacerla falsa.

c) $p = 0$, $q = 1$. El sentido común nos dicta que un condicional de este tipo no es ni verdadero ni falso: ¿qué sentido tiene preguntarse por la verdad o falsedad de un condicional cuando el antecedente es falso? Pero esta respuesta del sentido común no nos satisface, porque estamos trabajando con una lógica binaria, y, establecida una interpretación ($p = 0$, $q = 1$ en este caso), toda sentencia deberá tener un valor, 0 ó 1, en esa interpretación. Si una sentencia no es falsa, será cierta y viceversa. Ahora bien, en el caso que nos ocupa, el condicional no es falso. Y no es falso porque, como hemos dicho antes, «$p \rightarrow q$» es como decir que p es condición suficiente *pero no necesaria* de q, es decir, que no es la única condición, por lo que perfectamente puede ser q cierto, siendo p falso. Es decir, la falsedad del antecedente no hace falso al condicional, y si no lo hace falso, lo hace verdadero. Por ejemplo: «si corro, entonces me canso». ¿Qué ocurre si se dan los hechos de que no corro y, sin embargo, me canso? Ello no invalida la sentencia, porque no se dice que no pueda haber otras causas que me producen cansancio.

d) $p = 0$, $q = 0$. Pasa algo parecido al caso anterior: la condición no se da, por lo que q puede ser tan verdadero como falso, y el condicional, al no ser falso, será verdadero. Obsérvese, además, que este empleo del condicional se encuentra en el lenguaje coloquial para señalar que, ante un dislate, cualquier otro está justificado: «si Fulano de Tal es demócrata, yo soy el Emperador de Asiria».

Este significado del condicional, aunque hoy está universalmente admitido, ha ocasionado numerosos problemas en lógica, y ha sido una fuente de paradojas. Por ejemplo, la sentencia «$p \rightarrow (q \rightarrow p)$» tiene siempre la interpretación «verdadera», sea cual sea la interpretación de las variables p y q. En efecto, podemos construir la siguiente tabla para ver, en dos pasos, las cuatro interpretaciones posibles:

p	q	$q \rightarrow p$	$p \rightarrow (q \rightarrow p)$
0	0	1	1
0	1	0	1
1	0	1	1
1	1	1	1

Esto es lo que se llama una *tautología*, es decir, una sentencia que siempre es verdadera sean cuales sean las interpretaciones de las variables proposicionales que intervienen en ella. (De hecho, en algunos de los sistemas axiomáticos que se han propuesto para la lógica proposicional, esta sentencia es un axioma). Ahora bien, algunos consideran este resultado paradójico, porque, considerado como la formalización de un razonamiento en el que «p» fuese la premisa y «$q \rightarrow p$» la conclusión, viene a decir que si un enunciado (p) es verdadero, todo condicional en el que ese enunciado sea el consecuente es verdadero, independientemente de que el antecedente sea verdadero o falso. Por ejemplo, de un enunciado como «tengo frío» se deduce que «si

salgo al campo, entonces tengo frío» siempre es verdadero, independientemente de que «salgo al campo» sea verdadero o falso. Pero, teniendo en cuenta el sentido material del condicional, no es que «tengo frío» *implique* «si salgo al campo entonces tengo frío», sino que la segunda cosa *se sigue* de la primera (es verdadera cuando la primera lo es), y entonces la interpretación parece perfectamente razonable: si tengo frío, tengo frío, pase lo que pase.

- *Bicondicional.* Se interpreta como verdadero si las dos variables son verdaderas o las dos son falsas. En caso contrario (una verdadera y otra falsa) se interpreta como falso. Pueden hacerse consideraciones similares a las del condicional (e invitamos al lector a reflexionar sobre ello). De la misma manera que al condicional se le llama a veces «implicación» (material), al bicondicional se le conoce también como «equivalencia» (material, no estricta).

Una observación final: La tabla se puede escribir también utilizando dos sentencias cualesquiera, S_1 y S_2, en lugar de las variables proposicionales p y q. Para una determinada interpretación de las variables que forman parte de ellas, S_1 y S_2 tendrán unas interpretaciones «0» o «1» que se calcularán de acuerdo con la misma tabla. Y todo lo que hemos explicado sobre el significado de las distintas conectivas aplicadas a variables es extensible al caso en que apliquen a sentencias.

1.4. Ejemplos de formalización e interpretación

Aunque más adelante presentaremos con mayor rigor algunas de las ideas expuestas, lo que hemos visto es ya suficiente para poder formalizar y analizar muchas frases, razonamientos y argumentaciones expresados en lenguaje natural. Para ello, una vez construida la sentencia correspondiente, escribimos todas las interpretaciones de sus variables proposicionales; si hay n variables proposicionales, habrá 2^n interpretaciones binarias, que escribiremos en 2^n líneas. Después calculamos la interpretación de la sentencia para cada una de esas interpretaciones de variables proposicionales (lo cual puede exigir, para evitar errores, cálculos intermedios de las partes de la sentencia). El resultado es una tabla con 2^n filas, a la que llamaremos *tabla de verdad*. (Ya hemos hecho antes una, cuando comentábamos el significado del condicional). Veamos algunos ejemplos concretos:

Ejemplo 1.4.1. «Si corro, entonces me canso». Ya hemos comentado antes este enunciado, que podemos formalizar, simplemente, como «$p \rightarrow q$». Si lo repetimos aquí es para invitar al lector a reflexionar sobre el hecho de que hay formas alternativas de decir lo mismo. Por ejemplo: «$\neg p \vee q$» («o bien no corro, o bien me canso, o ambos»), «$\neg (p \wedge \neg q)$» («no es el caso que corro y no me canso»), etc.:

p	q	$\neg p$	$\neg q$	$\neg p \vee q$	$p \wedge \neg q$	$\neg (p \wedge \neg q)$
0	0	1	1	1	0	1
0	1	1	0	1	0	1
1	0	0	1	0	1	0
1	1	0	0	1	0	1

La primera de esas formas alternativas ($\neg p \vee q$) es una forma disyuntiva que, como veremos en los apartados 5.7 y 5.8, es importante para ciertos procedimientos de inferencia automática.

Ejemplo 1.4.2. «Si tengo hambre o tengo sed, entonces voy al bar». Definiendo las variables proposicionales p, q, r para representar respectivamente «tengo hambre», «tengo sed» y «voy al bar», podemos formalizar la frase en cuestión mediante la sentencia:

$$p \vee q \rightarrow r$$

Como intervienen tres variables, hay $2^3 = 8$ interpretaciones binarias para el conjunto de las tres, y para cada una de ellas la sentencia tendrá una interpretación. Las ocho interpretaciones, que se calculan de acuerdo con lo dicho anteriormente, se resumen en esta tabla de verdad:

p	q	r	$p \vee q$	$p \vee q \rightarrow r$
0	0	0	0	1
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

Como puede observarse, la sentencia es falsa cuando ocurre que «tengo hambre» o «tengo sed» (o ambos) son verdaderos y, sin embargo, «voy al bar» es falso. En cualquier otro caso, incluso cuando p y q son las dos falsas, la sentencia es verdadera (lo cual es razonable: la sentencia no dice que no se pueda ir al bar por otros motivos; lo diría si en lugar de un condicional se utilizase un bicondicional).

Ejemplo 1.4.3. Muchos razonamientos consisten en la obtención de una conclusión a partir de dos o más premisas: «si Premisa 1 y Premisa 2 y ... entonces Conclusión». Si las premisas y la conclusión pueden expresarse con el formulismo de la lógica de proposiciones, el razonamiento es válido siempre y cuando la sentencia global $[(P1 \wedge P2 \wedge \ldots) \rightarrow C]$ sea una tautología. En efecto, lo que queremos expresar con el razonamiento es justamente lo que expresa un condicional de esa forma: que en el caso de que las premisas sean ciertas la conclusión también lo es. Y si hay algún caso (interpretación de las variables proposicionales que intervienen en las premisas y en la conclusión) en el cual todas las premisas son verdaderas y la conclusión falsa, entonces el razonamiento falla; en tal caso, la sentencia condicional dada se interpreta como falsa. Si nunca ocurre tal cosa, la sentencia es una tautología y el razonamiento es correcto.

Consideremos ahora el siguiente razonamiento:

*P*1: 'Si Bernardo se casa, entonces Florinda se suicida'.
*P*2: 'Florinda se suicida si y sólo si Bernardo no se hace monje'.
C : 'Si Bernardo se casa, entonces no se hace monje'.

Para formalizar este razonamiento, vamos a definir las variables proposicionales:

c = 'Bernardo se casa'.
s = 'Florinda se suicida'.
m = 'Bernardo se hace monje'.

Los elementos del razonamiento quedarán así:

$P1 = c \rightarrow s$
$P2 = s \leftrightarrow \neg m$
$C = c \rightarrow \neg m$

y la expresión formal del razonamiento vendrá dada por la sentencia:

$$(c \rightarrow s) \wedge (s \leftrightarrow \neg m) \rightarrow (c \rightarrow \neg m)$$

Veamos si es una sentencia demostrable; para ello, construyamos su tabla de verdad:

c	s	m	$c \rightarrow s$	$s \leftrightarrow \neg m$	$(c \rightarrow s) \wedge$ $\wedge (s \leftrightarrow \neg m)$	$c \rightarrow \neg m$	$(c \rightarrow s) \wedge$ $\wedge (s \leftrightarrow \neg m) \rightarrow$ $\rightarrow (c \rightarrow \neg m)$
0	0	0	1	0	0	1	1
0	0	1	1	1	1	1	1
0	1	0	1	1	1	1	1
0	1	1	1	0	0	1	1
1	0	0	0	0	0	1	1
1	0	1	0	1	0	0	1
1	1	0	1	1	1	1	1
1	1	1	1	0	0	0	1

Comprobamos así que se trata de una tautología, y, por consiguiente, el razonamiento es correcto. Como ejercicio, sugerimos al lector que analice los resultados en los casos de que la premisa 2 fuese:

a) «Si Bernardo no se hace monje, entonces Florinda se suicida», y
b) «Si Florinda se suicida, entonces Bernardo no se hace monje».

Ejemplo 1.4.4. Veamos ahora un razonamiento con tres premisas:

*P*1: No hay judíos en la cocina.
*P*2: Ningún gentil dice «sphoonj».
*P*3: Todos mis sirvientes están en la cocina.
C : Mis sirvientes no dicen nunca «sphoonj».

Aunque el modo más natural de formalizar este razonamiento sería en lógica de predicados, y así lo veremos en el capítulo 4, también podemos hacerlo en lógica de proposiciones. En efecto, definiendo las variables proposicionales:

j : Alguien es judío.
c : Alguien está en la cocina.
d: Alguien dice «sphoonj».
s : Alguien es sirviente.

podemos escribir así el razonamiento:

$P1: j \rightarrow \neg c$
(«si alguien es judío, entonces no está en la cocina»)
$P2: \neg j \rightarrow \neg d$
(«si alguien no es judío, entonces no dice 'sphoonj'»)
$P3: s \rightarrow c$
(«si alguien es sirviente, entonces está en la cocina»)
$C : s \rightarrow \neg d$
(«si alguien es sirviente, entonces no dice 'sphoonj'»)

Enlazando las tres premisas entre sí mediante conjunciones y a la conclusión mediante un condicional, obtenemos la siguiente sentencia, expresión formal del razonamiento (obsérvese que, gracias a la asociatividad de la conjunción, podemos escribir $[(P1 \wedge P2) \wedge P3]$ como $[P1 \wedge P2 \wedge P3]$):

$$[(j \rightarrow \neg c) \wedge (\neg j \rightarrow \neg d) \wedge (s \rightarrow c)] \rightarrow (s \rightarrow \neg d)$$

Veamos ahora las posibles interpretaciones de la sentencia. En la siguiente tabla de verdad se indican los pasos intermedios y el resultado:

j	c	d	s	$P1$	$P2$	$P3$	$P1 \wedge P2 \wedge P3$	C	S
0	0	0	0	1	1	1	1	1	1
0	0	0	1	1	1	0	0	1	1
0	0	1	0	1	0	1	0	1	1
0	0	1	1	1	0	0	0	0	1
0	1	0	0	1	1	1	1	1	1
0	1	0	1	1	1	1	1	1	1
0	1	1	0	1	0	1	0	1	1
0	1	1	1	1	0	1	0	0	1
1	0	0	0	1	1	1	1	1	1
1	0	0	1	1	1	0	0	1	1
1	0	1	0	1	1	1	1	1	1
1	0	1	1	1	1	0	0	0	1
1	1	0	0	0	1	1	0	1	1
1	1	0	1	0	1	1	0	1	1
1	1	1	0	0	1	1	0	1	1
1	1	1	1	0	1	1	0	0	1

Vemos que la sentencia es una tautología, y, por tanto, el razonamiento es válido. No obstante, hay que insistir en que, para este ejemplo y otros similares, la lógica de proposiciones empieza a ser un lenguaje demasiado limitado para representar todos los matices del lenguaje natural. Concretamente, aquí hemos hecho una pequeña «trampa»: tal como se han definido las variables proposicionales, lo representado por «$j \rightarrow \neg\, c$» sería: «si alguien es júdío, entonces no es cierto que alguien esté en la cocina», lo cual no es exactamente lo que queremos decir, y lo mismo con las otras tres sentencias. Lo que ocurre es que con la palabra «alguien» del lenguaje natural se representa a un miembro cualquiera de un colectivo, es decir, formalmente, a una variable (pero no una variable proposicional). Nuestras proposiciones hacen referencia a propiedades de tales variables que determinan subconjuntos del colectivo, y esto es algo no previsto en la lógica de proposiciones. Si aquí hemos podido llegar a una formalización ha sido «fijando» ese «alguien», es decir, considerando que en todas las premisas y en la conclusión se trata del mismo individuo (como si, en vez de «alguien», hubiésemos dicho, por ejemplo, «Juan») y suponiendo implícitamente que si el razonamiento es válido para un individuo cualquiera, lo es para todos.

Ejemplo 1.4.5. Consideremos un político que declara en la prensa:

$P1$: 'Si los impuestos suben, la inflación bajará si y sólo si la peseta no se devalúa';

para, posteriormente, decir en televisión:

$P2$: 'Si la inflación baja o si la peseta no se devalúa, los impuestos no subirán';

y afirmar en el Congreso:

$P3$: 'O bien baja la inflación y se devalúa la peseta, o bien los impuestos deben subir'.

Nuestro político publica luego un informe en el que, tras analizar tales aseveraciones, saca de ellas la conclusión:

C: 'Los impuestos deben subir, pero la inflación bajará y la peseta no se devaluará'.

Nos preguntamos si tal conclusión es consistente con las premisas, sin entrar en la validez de éstas. Es decir, $P1$, $P2$ y $P3$ pueden ser verdaderas o falsas; si alguna es falsa, la conclusión no tiene por qué ser cierta, pero si las tres son verdaderas, C debe serlo también. En definitiva, para que la inferencia de C a partir de $P1$, $P2$ y $P3$ sea correcta $(P1 \wedge P2 \wedge P3) \rightarrow C$, deberá ser una tautología.

Si definimos las variables proposicionales,

p = 'los impuestos suben';
q = 'la inflación baja';
r = 'la peseta no se devalúa',

la sentencia a comprobar será:

$$[(p \rightarrow (q \leftrightarrow r)) \wedge ((q \vee r) \rightarrow \neg\, p) \wedge ((q \wedge \neg\, r) \vee p)] \rightarrow (p \wedge q \wedge r)$$

Dejamos al lector la construcción de la tabla de verdad de esta sentencia. Podrá comprobar que no es una tautología (por lo que la conclusión no es correcta), y estudiar los dos casos que hacen que el razonamiento no sea válido.

1.5. Cálculo, sistema axiomático y sistema inferencial

Veremos en el siguiente apartado que la lógica de predicados se define como un lenguaje formal, en el sentido definido en el capítulo 1, complementado con un conjunto de axiomas y de reglas de transformación o inferencia.

Un *cálculo* es la estructura formal de un lenguaje, abstrayendo el significado, y, en rigor, sólo se transforma en lenguaje cuando se interpretan sus símbolos y sus construcciones (es decir, se les atribuye un significado, se les pone en relación con los objetos que designan). Según esta definición, un cálculo no entra en los aspectos semánticos. No obstante, muchas veces se habla de «cálculo de proposiciones» y «cálculo de predicados» como sinónimos de «lógica de proposiciones» y «lógica de predicados». ¿Es esto correcto? La duda proviene de que en lógica hablamos de interpretación de variables proposicionales y sentencias como verdaderas o falsas, con lo que estamos haciendo semántica. Ahora bien, ésta es una semántica «aséptica», desligada de lo que realmente representan esas variables y sentencias. En los ejemplos vistos puede verse claramente: cuando construimos las tablas de verdad estamos en la semántica de la lógica, pero la «verdadera» semántica la hacemos después: cuando trasladamos esos resultados a la interpretación de la frase que estamos representando en lógica.

El cálculo es, además, un *sistema axiomático* cuando se construye sobre la base de unos *axiomas*, o construcciones que se admiten como verdaderas en el lenguaje o lenguajes de los que procede el cálculo (o, lo que es lo mismo, en todos los lenguajes que se pueden formalizar con él).

Para definir un sistema axiomático hay que especificar:

* el *alfabeto*, o conjunto de símbolos válidos en el cálculo;
* las *reglas de formación*, que permiten derivar sentencias (cadenas de símbolos válidos o correctas);
* los *axiomas*;
* las *reglas de transformación*, que permiten obtener o *demostrar* ciertas sentencias a partir de los axiomas (*sentencias demostrables, teoremas* o *leyes*).

Los axiomas y los teoremas constituyen las *tesis* del sistema axiomático, y pertenecen al lenguaje del cálculo; las reglas de formación y de transformación pertenecen al metalenguaje del cálculo.

La formulación de los axiomas no es arbitraria ni fácil. Tiene que satisfacer dos exigencias metalógicas imprescindibles:

a) El cálculo resultante debe ser *completo*, es decir, todas las sentencias *verdaderas* (o sea, todas las tautologías: la semántica es ineludible) relativas a las teorías (o lenguajes) que el cálculo pretende formalizar tienen que poder demostrarse a partir de los axiomas.

b) El cálculo debe ser *consistente*, es decir, que no se puedan demostrar sentencias no verdaderas.

Conviene, además, que los axiomas sean independientes, es decir, que ninguno de ellos pueda demostrarse a partir de los restantes. Esta condición no es fácil de comprobar. Prueba de ello es que en el sistema PM, que veremos en el apartado 2.5, se proponían inicialmente 5 axiomas, y sólo años más tarde se comprobó que uno era redundante (se podía demostrar a partir de los otros cuatro).

El lector más interesado por las aplicaciones informáticas de la lógica que por ésta en sí misma puede pensar que todo esto es de poca utilidad. Vamos a ver que no es así. En las aplicaciones, efectivamente, no tiene un interés especial la demostración de teoremas a partir de unos axiomas. Lo realmente interesante es mecanizar los procesos de inferencia. Es decir, dadas unas premisas o hechos comprobados como ciertos en una determinada situación, poder deducir conclusiones para esa situación. Esto es lo que llamaremos un *sistema inferencial*: un conjunto de reglas (pertenecientes al metalenguaje de la lógica) que permiten ejecutar inferencias y unas metarreglas (pertenecientes al metametalenguaje) que dicen cómo aplicarlas. Ahora bien, veremos en el apartado 5 que los teoremas del sistema axiomático son precisamente la base de las reglas que permiten realizar inferencias. Por otra parte, para acercarnos al ideal de «poder asegurar» el correcto funcionamiento de los programas es preciso trabajar sobre una base teóricamente sólida y rigurosa. (La pragmática de la informática dice que nunca se puede asegurar que un programa funciona correctamente. Gran parte de las investigaciones actuales sobre programación se dirigen, precisamente, a mejorar la fiabilidad del software, y en ellas juegan un papel fundamental las construcciones teóricas).

2. SINTAXIS

2.1. Alfabeto

Entramos ya a definir formalmente la lógica de proposiciones como un lenguaje. El alfabeto que utilizaremos será el formado por los siguientes símbolos:

- *Variables proposicionales: p, q, r, s, t...* (En caso necesario, con subíndices numéricos, y, en los ejemplos de aplicación, cualquiera otra minúscula, si conviene por motivos nemotécnicos).
- *Conectivas:* $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$. (Estas serán las más utilizadas, aunque pueden definirse otras, como veremos).
- *Símbolos de puntuación:* (,), [,], {, }. (Lo mismo que en el lenguaje del álgebra, los utilizaremos para evitar ambigüedades).
- *Metasímbolos*. Como indica su nombre, no son, propiamente, símbolos del lenguaje. Los utilizaremos para abreviar ciertas expresiones, y serán los siguientes:
  - * *A, B, C...* para representar cualquier sentencia del lenguaje (también, en caso necesario, con subíndices).

* k para representar cualquier conectiva binaria.
* l para representar una variable proposicional sola o con el símbolo de negación delante. (Esto es lo que se llama un *literal:* literal *positivo* si corresponde a una variable proposicional sin negación, y *negativo* en caso contrario).

2.2. Expresiones y sentencias

Definición 2.2.1. Llamaremos *expresión* o *cadena* a toda secuencia finita de símbolos del alfabeto. Por ejemplo:

$$(p \vee \neg q) \rightarrow r$$
$$r \vee) \rightarrow p\ q \neg ($$
$$(p \vee q \neg) \rightarrow r$$

son tres expresiones. Intuitivamente, se aprecia que sólo la primera «significa» algo; diremos que, de las tres, sólo ella es una sentencia. Vamos a definir formalmente este concepto.

Definición 2.2.2. Llamaremos *secuencia de formación* a toda secuencia finita de expresiones, $A_1, A_2, \ldots, A_n$, en la que cada A_i satisface al menos una de las tres condiciones siguientes:

a) A_i es una variable proposicional;
b) existe un j menor que i tal que A_i es el resultado de anteponer el símbolo «$\neg$» a A_j;
c) existen j y h, ambos menores que i, tales que A_i es el resultado de enlazar A_j y A_h con k (cualquier conectiva binaria).

Definición 2.2.3. Llamaremos *sentencia* a toda expresión A_n tal que existe una secuencia de formación $A_1, A_2, \ldots, A_n$. Y ésta será *una secuencia de formación de* A_n (puede no ser la única). Obsérvese que, según esta definición, todas las expresiones que componen una secuencia de formación son sentencias.

Una forma equivalente de definir el concepto de sentencia es enunciando tres *reglas de formación* (correspondientes a las tres condiciones anteriores):

RF1: Una variable proposicional es una sentencia.
RF2: Si A es una sentencia, $\neg A$ también lo es.
RF3: Si A y B son sentencias, AkB también lo es.

Estas tres reglas constituyen una *definición recursiva* de sentencia. (Recursiva, porque en ella se alude al concepto definido).

2.3. Sentencias equivalentes

En realidad, el concepto de equivalencia entre sentencias es semántico, y lo definiremos en el apartado 3.5. De momento, digamos que dos sentencias son

equivalentes si «significan» lo mismo, y, aunque esta definición no sea muy precisa, haremos ya uso de algunas equivalencias sencillas, que nos permitirán sustituir unas sentencias por otras. Por ejemplo, dos equivalencias triviales son: $A \vee B$ es equivalente a $B \vee A$, y $A \wedge B$ es equivalente a $B \wedge A$.

2.4. Conectivas primitivas y definidas

De las cuatro conectivas binarias del alfabeto definido, basta con una. Si, por ejemplo, nos quedamos con las conectivas primitivas «$\neg$» y «$\vee$», podemos definir las tres restantes del siguiente modo:

a) Definición de «$\wedge$»: Una sentencia de la forma

$$A \wedge B$$

es equivalente a una sentencia de la forma

$$\neg (\neg A \vee \neg B)$$

b) Definición de «$\rightarrow$»: Una sentencia de la forma

$$A \rightarrow B$$

es equivalente a una sentencia de la forma

$$\neg A \vee B$$

c) Definición de «$\leftrightarrow$»: Una sentencia de la forma

$$A \leftrightarrow B$$

es equivalente a una sentencia de la forma

$$\neg (\neg (\neg A \vee B) \vee \neg (\neg B \vee A))$$

O bien, si se permite utilizar la conectiva previamente definida «$\wedge$»,

$$(\neg A \vee B) \wedge (\neg B \vee A)$$

2.5. Axiomas, demostraciones y teoremas

2.5.1. *Axiomas*

El sistema axiomático más conocido es el llamado «PM», nombre derivado del título de una obra clásica: «Principia Mathematica», de Whitehead y Russell. Utiliza cuatro axiomas:

A1. $(p \vee p) \rightarrow p$
A2. $q \rightarrow (p \vee q)$
A3. $(p \vee q) \rightarrow (q \vee p)$
A4. $(p \rightarrow q) \rightarrow [(r \vee p) \rightarrow (r \vee q)]$

Como hemos dicho en el apartado 1.5, la idea es que con estos axiomas y unas reglas de transformación se demuestran teoremas. Vamos a definir las reglas de transformación y el proceso de demostración, pero antes hemos de introducir una operación previa: la sustitución.

2.5.2. *Sustitución*

Definición 2.5.2.1. Dadas unas variables proposicionales, $p_1, p_2, \ldots, p_n$ y unas sentencias cualesquiera, $B_1, B_2, \ldots, B_n$, llamaremos *sustitución* a un conjunto de pares ordenados:

$$s = \{B_1/p_1, B_2/p_2, \ldots, B_n/p_n\}$$

Definición 2.5.2.2. Dadas, una sentencia A que contiene (quizas, entre otras) las variables proposicionales $p_1, p_2, \ldots, p_n$, y una sustitución s, la *operación de sustitución* en A de esas variables proposicionales por las sentencias $B_1, B_2, \ldots, B_n$ consiste en poner en A, en todos los lugares donde aparezca la variable p_i, la sentencia B_i que le corresponde según s, y esto para todo $i = 1, \ldots, n$. El resultado es una expresión que representaremos como As.

Teorema 2.5.2.3. Si A es una sentencia y s una sustitución, As es una sentencia. La demostración, que dejamos al lector, es sencilla: basta con ver que si A y todas las B_i tienen secuencias de formación, entonces As tiene también una secuencia de formación (al menos). Es interesante observar que este teorema, como otros que iremos dando, es, en realidad, un «metateorema» para la lógica de proposiciones.

2.5.3. *Demostraciones y teoremas*

Definición 2.5.3.1. Llamaremos *demostración* (o *prueba formal*) a toda secuencia finita de sentencias $A_1, A_2, \ldots, A_n$, en la que cada A_i satisface, al menos, una de las cuatro condiciones siguientes:

a) A_i es un axioma.
b) Existe algún j menor que i y alguna sustitución s tal que A_i es el resultado de la sustitución s en A_j, es decir, A_i es lo mismo que $A_j s$.
c) Existen h y j menores que i tales que A_i es lo mismo que $A_h \wedge A_j$.
d) Existen h y j menores que i tales que A_h es lo mismo que $A_j \rightarrow A_i$.

Definición 2.5.3.2. Llamaremos *teorema* a toda sentencia A_n que no es un axioma y que es tal que existe una demostración $A_1, A_2, \ldots, A_n$, y diremos que ésta es *una*

demostración de A_n (puede no ser única). Obsérvese que todas las sentencias que componen una demostración son o bien axiomas o bien teoremas.

Definición 2.5.3.3. Llamaremos *tesis* (o *ley*) del sistema axiomático a cualquier sentencia que sea o bien un axioma o bien un teorema. Utilizaremos la notación «$\vdash A$» para indicar que «A es una tesis».

De modo similar a lo que hacíamos al definir las sentencias, podemos dar una definición recursiva de tesis mediante cuatro *reglas de transformación* (correspondientes a las cuatro condiciones anteriores).

2.5.4. *Reglas de transformación*

RT1. Si A es un axioma, entonces A es una tesis.
RT2. (*Regla de sustitución*): Si A es una tesis en la que aparecen $p_1, p_2, \ldots, p_n$ y $B_1, B_2, \ldots, B_n$ son sentencias, entonces $A\{B_1/p_1, \ldots, B_n/p_n\}$ es una tesis.
RT3. (*Regla de unión*): Si A y B son tesis, entonces $A \wedge B$ es una tesis.
RT4. (*Regla de separación*): Si A y $A \rightarrow B$ son tesis, entonces B es una tesis.

2.5.5. *Ejemplos de demostraciones en el sistema PM*

Teorema 1: $(p \rightarrow q) \rightarrow [(r \rightarrow p) \rightarrow (r \rightarrow q)]$.

Demostración:

1. $(p \rightarrow q) \rightarrow [(\neg r \vee p) \rightarrow (\neg r \vee q)]$
   (Por sustitución A4$\{\neg r/r\}$).
2. $(p \rightarrow q) \rightarrow [(r \rightarrow p) \rightarrow (r \rightarrow q)]$
   (Por definición de la conectiva $\rightarrow$).

Teorema 2: $p \rightarrow (p \vee p)$.

Demostración:

Simplemente, por sustitución A2$\{p/q\}$.

Teorema 3: $p \rightarrow p$.

Demostración:

1. $[(p \vee p) \rightarrow p] \rightarrow [(p \rightarrow (p \vee p)) \rightarrow (p \rightarrow p)]$
   (Por sustitución en el Teorema 1: T1$\{p \vee p)/p, p/q, p/r\}$.
2. $[p \rightarrow (p \vee p)] \rightarrow (p \rightarrow p)$
   (Por separación de A1 en la sentencia anterior).
3. $p \rightarrow p$
   (Por separación del Teorema 2 en la sentencia anterior).

Teorema 4: $\neg p \vee p$ (Principio del tercio excluso).

Demostración:

Por definición de la conectiva $\rightarrow$ en el Teorema 3.

(*Nota:* para no recargar la notación, no hemos escrito el símbolo «$\vdash$», lo que podríamos haber hecho delante de cada una de las sentencias que han ido apareciendo en las demostraciones).

2.5.6. *Algunos teoremas útiles*

Entre la infinidad de teoremas que pueden demostrarse están, por ejemplo:

* La *ley de modus ponendo ponens* (o *modus ponens*):

$$[p \wedge (p \rightarrow q)] \rightarrow q$$

(Corresponde al tipo de razonamiento según el cual, dado un condicional y afirmando («ponendo») el antecedente, se puede afirmar («ponens») el consecuente, pero esto ya es una «regla de inferencia», de la que hablaremos más adelante).

* La *ley de modus tollendo tollens* (o *modus tollens*):

$$[\neg q \wedge (p \rightarrow q)] \rightarrow \neg p$$

(Corresponde al tipo de razonamiento según el cual, dado un condicional y negando («tollendo») el consecuente, se puede negar («tollens») el antecedente, otra regla de inferencia).

* Las *leyes de transitividad*:

$$[(p \rightarrow q) \wedge (q \rightarrow r)] \rightarrow (p \rightarrow r)$$
$$[(p \leftrightarrow q) \wedge (q \leftrightarrow r)] \rightarrow (p \leftrightarrow r)$$

(Corresponden a los «silogismos hipotéticos» de la lógica clásica).

* Las *leyes de inferencia de la alternativa,* o de los *silogismos disyuntivos*:

$$[\neg p \wedge (p \vee q)] \rightarrow q$$
$$[p \wedge (\neg p \vee \neg q)] \rightarrow \neg q$$

* La *ley del dilema constructivo*:

$$[(p \rightarrow q) \wedge (r \rightarrow s) \wedge (p \vee r)] \rightarrow (q \vee s)$$

* Las *leyes de De Morgan*:

$$\neg (p \wedge q) \leftrightarrow (\neg p \vee \neg q)$$
$$\neg (p \vee q) \leftrightarrow (\neg p \wedge \neg q)$$

Teoremas como estos dos últimos, de la forma $A \leftrightarrow B$, establecen una equivalencia entre las sentencias A y B, como veremos en el apartado 3.5. Otras equivalencias que utilizaremos más adelante son:

* La *ley de doble negación*:

$$\neg \neg p \leftrightarrow p$$

* La *ley de reducción al absurdo*:

$$[\neg p \rightarrow (q \wedge \neg q)] \leftrightarrow p$$

* Las *leyes de distributividad*:

$$[(p \wedge q) \vee r] \leftrightarrow [(p \vee r) \wedge (p \vee q)];$$
$$[(p \vee q) \wedge r] \leftrightarrow [(p \wedge r) \vee (q \wedge r)];$$
$$[(p \rightarrow q) \vee r] \leftrightarrow [(p \vee r) \rightarrow (q \vee r)];$$
$$[p \rightarrow (q \vee r)] \leftrightarrow [(p \rightarrow q) \vee (p \rightarrow r)];$$
$$[p \rightarrow (q \wedge r)] \leftrightarrow [(p \rightarrow q) \wedge (p \rightarrow r)];$$
$$[p \rightarrow (q \rightarrow r)] \leftrightarrow [(p \rightarrow q) \rightarrow (p \rightarrow r)]$$

3. SEMÁNTICA

3.1. Interpretaciones

Representaremos por P al conjunto (finito) de variables proposicionales, por S al conjunto (infinito) de sentencias que pueden construirse con P, por P_n y S_n a los mismos conjuntos cuando queramos indicar que el número de variables proposicionales es n, y por S^n al subconjunto de S (finito si n es finito) formado por las sentencias que tengan un número de conectivas igual o inferior a n.

Definición 3.1.2. Un *conjunto de valores semánticos* es un conjunto de símbolos, V, de cardinalidad igual o superior a 2, cerrado bajo ciertas operaciones definidas en él. A cada conectiva definida en el alfabeto debe corresponder una operación en V. A los elementos de V les llamaremos *valores de verdad*.

Obsérvese que, en principio, no imponemos más restricción a V que la de tener dos o más elementos (puede, incluso, tener infinitos), y de tener definidas en él (sin fijar cómo) una operación por cada conectiva.

Definición 3.1.3. Una *interpretación* es un par ordenado $\langle i, V \rangle$, donde V es un conjunto de valores semánticos e i es una aplicación cualquiera de P en V.

De este modo, cuando establecemos una determinada interpretación, lo que hacemos es asignar a cada variable proposicional un elemento de V. Para interpretar sentencias, tendremos que establecer una extensión de i para que su dominio sea S en lugar de P.

Teorema 3.1.4. Dada una interpretación $\langle i, V \rangle$, existe una (y sólo una) extensión del dominio de i a S, a la que llamaremos I, tal que satisface estas dos condiciones:

a) $\forall A \in S, I(\neg A) = \neg I(A)$.
b) $\forall A, B \in S, I(AkB) = I(A)kI(B)$.

(Nótese que, aunque los símbolos utilizados sean los mismos, $\neg$ y k representan distintas cosas, en el primero y en el segundo miembro de esas igualdades: en el primero, son las conectivas utilizadas en el alfabeto, que se aplican a variables o a sentencias para obtener otras sentencias; en el segundo, son las operaciones definidas en el conjunto V).

Demostración:

a) Existencia. La demostración de que existe al menos una I que cumple esas condiciones se hace por inducción sobre n (número de conectivas de las sentencias) y teniendo en cuenta que toda sentencia tiene que estar construida según las reglas de formación dadas en 2.2. Si $A \in S^n$, denotaremos $I(A)$ por $I^n(A)$:

* Para $n = 0$, $I^0(A) = i(A)$;
* Supuesto que existe I^n, definimos I^{n+1}, así:
  si $A \in S^n$, $I^{n+1}(\neg A) = \neg I^n(A)$;
  si $A, B \in S^n$, $I^{n+1}(AkB) = I^n(A)kI^n(B)$.

De este modo, hemos construido una aplicación I^{n+1} que cumple las condiciones impuestas, y que está definida para toda sentencia de S^{n+1}.

b) Unicidad. I^0 es única, puesto que es igual a i. Y si I^n es única, I^{n+1} también lo será, puesto que se ha construido de manera única.

Corolario: I está unívocamente determinada por i. Esto quiere decir que, dada una interpretación de variables proposicionales, puede calcularse la interpretación de cualquier sentencia construida con esas variables proposicionales.

3.2. Interpretaciones binarias

En el apartado anterior hemos definido el concepto de interpretación de una manera general, lo cual permite aplicarlo a lógicas en las que las variables proposicionales no se interpretan de manera dicotómica, sino que pueden tomar valores intermedios entre «verdadero» y «falso». Veremos algunos de estos tipos de lógica en el capítulo 5. De momento, y hasta entonces, nos limitaremos a la interpretación clásica.

Definición 3.2.1. Una *interpretación binaria* es una interpretación $\langle i, V\rangle$ (o su extensión, $\langle I, V\rangle$) en la que V tiene dos elementos, que llamaremos «0» y «1» y (además de otras que luego veremos), cinco operaciones definidas en él de la siguiente forma:

$$\begin{gathered}\neg(0) = 1;\ \neg(1) = 0\\ 0 \wedge 0 = 0;\ 0 \wedge 1 = 0;\ 1 \wedge 0 = 0;\ 1 \wedge 1 = 1\\ 0 \vee 0 = 0;\ 0 \vee 1 = 1;\ 1 \vee 0 = 1;\ 1 \vee 1 = 1\\ 0 \to 0 = 1;\ 0 \to 1 = 1;\ 1 \to 0 = 0;\ 1 \to 1 = 1\\ 0 \leftrightarrow 0 = 1;\ 0 \leftrightarrow 1 = 0;\ 1 \leftrightarrow 0 = 0;\ 1 \leftrightarrow 1 = 1\end{gathered}$$

Esta definición de las operaciones es la misma que ya habíamos visto y comentado en el apartado 1.3. Lo que hemos hecho ahora ha sido formalizar (y generalizar al caso no binario). Por ejemplo, si $P = \{p, q\}$, tendremos cuatro interpretaciones binarias posibles, las que dimos en el apartado 1.3. Pero si V es un conjunto con más de dos elementos, entonces tendremos más interpretaciones, como veremos en el capítulo 5 (apartado 3).

En el caso de interpretación binaria, si consideramos, por ejemplo, la sentencia $p \vee q$, la extensión de i_1 aplicada a esta sentencia, de acuerdo con la construcción definida en la demostración del teorema 3.1.4 será:

$$I_1(p \vee q) = I_1(p) \vee I_1(q) = i_1(p) \vee i_1(q) = 0 \vee 1 = 1$$

e igualmente podrían calcularse todas las interpretaciones dadas en la tabla del apartado 1.3.

Si $\text{card}(P) = n$, entonces habrá 2^n interpretaciones binarias de las variables proposicionales, $i_j: P \to \{0, 1\}$ $(j = 0, \ldots, 2^{n-1})$, y, para cada una de ellas, cualquier sentencia A tendrá una interpretación, $I_j(A)$.

3.3. Tautologías y contradicciones

Definición 3.3.1. Diremos que una sentencia A es una *tautología* si $(\forall i_j)$ $(I_j(A) = 1)$. Utilizaremos la notación «$\models A$» para indicar que «A es una tautología».

Definición 3.3.2. Diremos que una sentencia A es una *contradicción* si $(\forall i_j)$ $(I_j(A) = 0)$.

Con las interpretaciones binarias que estamos considerando, y por definición de la operación $\neg$ en el conjunto $V = \{0, 1\}$, si $\models A$, entonces $\neg A$ es una contradicción, y viceversa: si A es una contradicción, entonces $\models (\neg A)$. Por ejemplo, como puede comprobarse fácilmente $(\neg p \vee p)$ es una tautología, y $\neg(\neg p \vee p)$ es una contradicción. Del mismo modo $(\neg p \wedge p)$ es una contradicción, y $\neg(\neg p \wedge p)$ una tautología.

3.4. Completitud y consistencia de un sistema axiomático

Ahora podemos expresar con mayor rigor algo que definíamos en el apartado 1.5:

Definición 3.4.1. Diremos que un sistema axiomático es *completo* si toda sentencia A que sea una tautología es también una tesis. Es decir, para toda A, si $\models A$, entonces $\vdash A$.

Definición 3.4.2. Diremos que un sistema axiomático es *consistente* si toda sentencia A que sea una tesis es también una tautología. Es decir, para toda A, si $\vdash A$, entonces $\models A$.

En particular, puede demostrarse que el sistema PM, con la interpretación binaria que hemos definido, es completo y consistente. El lector puede comprobar, por ejemplo, que tanto los axiomas como los teoremas que hemos demostrado o citado son tautologías.

3.5. Equivalencia entre sentencias

Definición 3.5.1. Dados un conjunto de variables proposicionales P y una sentencia A construida con ellas, diremos que una determinada interpretación i_j *satisface* A si (y sólo si) $I_j(A) = 1$.

Definición 3.5.2. Dadas dos sentencias A y B construidas con las mismas variables proposicionales, P, diremos que A es *equivalente* a B, y escribiremos $A \equiv B$, si (y sólo si) $(\forall i_j)\ (I_j(A) = I_j(B))$. Es inmediato comprobar que la relación es, en efecto, una relación de equivalencia, es decir, que es reflexiva, simétrica y transitiva.

Como ejercicio, el lector puede comprobar que las equivalencias indicadas en los apartados 2.3 y 2.4 cumplen los requisitos de esta definición.

Como toda relación de equivalencia, $\equiv$ particiona al conjunto (infinito) de sentencias construidas con n variables en un conjunto de clases de equivalencia, $E_n = S_n/\equiv$. Si el número de interpretaciones diferentes de las variables proposicionales es finito (lo cual ocurre si V es finito), entonces E_n es también finito. Por ejemplo, con $n = 2$ (dos variables proposicionales, p y q) y con $V = \{0, 1\}$ resultan 16 clases de equivalencia, $E_2 = \{C_0, C_1, \ldots, C_{15}\}$ cuyas interpretaciones son las siguientes:

p	q	$C0$	$C1$	$C2$	$C3$	$C4$	$C5$	$C6$	$C7$	$C8$	$C9$	$C10$	$C11$	$C12$	$C13$	$C14$	$C15$
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Así, todas las sentencias cuyas cuatro interpretaciones sean las de la clase C_1 ($p \wedge q$, $\neg(\neg p \vee \neg q)$, $p \rightarrow \neg q$, etc.) son equivalentes entre sí.

Teorema 3.5.3. $A \equiv B$ si y sólo si $\vdash (A \leftrightarrow B)$. Es decir, las sentencias A y B son equivalentes si y sólo si la sentencia $A \leftrightarrow B$ es una tesis.

Demostración:

a) Si $\vdash (A \leftrightarrow B)$ y el sistema axiomático es consistente, entonces $\models (A \leftrightarrow B)$, es decir, $A \leftrightarrow B$ es una tautología. Por definición del bicondicional, $A \leftrightarrow B$ es verdadera (y sólo es verdadera) si A y B son ambas verdaderas o ambas falsas. Por tanto, si $A \leftrightarrow B$ es una tautología (verdadera para todas las interpretaciones), entonces A y B han de tener las mismas interpretaciones para todas y cada una de las interpretaciones de las variables proposicionales. Y ésta es precisamente la definición de equivalencia entre A y B. Concluimos así que si $\vdash (A \leftrightarrow B)$, entonces $A \equiv B$.

b) Si $A \equiv B$, por la misma definición 3.5.2. y por la definición del bicondicional, $\models (A \leftrightarrow B)$. Y si el sistema axiomático es completo ello implica que $\vdash (A \leftrightarrow B)$. Concluimos así que si $A \equiv B$, entonces $\vdash (A \leftrightarrow B)$.

3.6. Las conectivas binarias

Hasta ahora venimos utilizando cuatro conectivas diádicas o binarias (binarias, en el sentido de que enlazan dos variables proposicionales; además, e independientemente de ello, estamos considerando interpretaciones binarias, en el sentido de que $V = \{0, 1\}$). Hemos visto que, gracias a algunas equivalencias (apartado 2.4), podemos limitarnos a usar una sola de ellas, junto con la negación. Pero también podemos hacer lo contrario: definir nuevas conectivas. Veamos cuántas. Si comparamos la tabla del apartado 1.3, en la que se definía el significado de las conectivas utilizadas, con la tabla del apartado anterior, en la que se muestran las interpretaciones de las 16 clases de equivalencia de las sentencias construidas con dos variables proposicionales, vemos inmediatamente que las conectivas «$\wedge$», «$\vee$», «$\rightarrow$» y «$\leftrightarrow$» se corresponden con las clases de equivalencia C_1, C_7, C_{13} y C_9, respectivamente. Y para cada una de las otras doce clases podemos definir una nueva conectiva binaria. Por ejemplo, C_6 correspondería a la disyunción exclusiva, para la que a veces se utiliza el símbolo «$\oplus$».

Las equivalencias dadas en el apartado 2.4 nos permitían representar las clases de equivalencia C_1, C_{13} y C_9 mediante sentencias que sólo utilizan las conectivas «$\neg$» y «$\vee$». Lo mismo puede hacerse para las otras clases. Por ejemplo, la clase C_0 se representaría mediante $\neg (p \vee \neg p)$, la C_2 mediante $\neg (\neg p \vee q)$, etc. Igualmente, podríamos trabajar con la pareja «$\neg$», «$\wedge$», o con la «$\neg$», «$\leftrightarrow$», etc. Pero también es posible representar cualquier clase de equivalencia utilizando solamente una conectiva. Ello puede hacerse con dos de las dieciséis conectivas binarias:

a) La «incompatibilidad», o «negación alternativa», o «NAND», que corresponde a la clase C_{14}, y que representaremos con el símbolo «$|$».
b) La «negación conjunta», o «NOR», que corresponde a la clase C_8, y que representaremos con el símbolo «$\downarrow$».

Por ejemplo, las equivalencias que permiten sustituir a las conectivas más utilizadas por una cualquiera de éstas son:

Para C_{12}: $\neg p \equiv p \mid p \equiv p \downarrow p$
Para C_1 : $p \wedge q \equiv (p \mid q) \mid (p \mid q) \equiv (p \downarrow p) \downarrow (q \downarrow q)$
Para C_7 : $p \vee q \equiv (p \mid p) \mid (q \mid q) \equiv (p \downarrow q) \downarrow (p \downarrow q)$
Para C_{13}: $p \rightarrow q \equiv p \mid (q \mid q) \equiv ((p \downarrow p) \downarrow q) \downarrow ((p \downarrow p) \downarrow q)$
Para C_9 : $p \leftrightarrow q \equiv ((p \mid p) \mid (q \mid q)) \mid (p \mid q) \equiv (p \downarrow (q \downarrow q)) \downarrow ((p \downarrow p) \downarrow q)$

Estas equivalencias, que en lógica tienen un interés puramente teórico, son de utilidad en las aplicaciones a circuitos de conmutación, como veremos en el capítulo 3.

4. MODELO ALGEBRAICO DE LA LÓGICA DE PROPOSICIONES

4.1. Algebra de Boole de las interpretaciones binarias

Recordemos que un álgebra de Boole es una estructura

$$\langle B, +, \cdot, \bar{} \rangle$$

formada por un conjunto de base, B, con tres operaciones definidas en él, suma (+), producto ($\cdot$) y complementación ($\bar{}$), tales que se cumplen los siguientes axiomas:

A1: B es cerrado para las tres operaciones (es decir, si $a, b \in B$, entonces $\bar{a} \in B$, $a + b \in B$ y $a \cdot b \in B$).

A2: Existen dos elementos neutros:

$$\exists \mathbf{0} \in B \text{ tal que } \forall a \in B,\ a + \mathbf{0} = a$$
$$\exists \mathbf{1} \in B \text{ tal que } \forall a \in B,\ a \cdot \mathbf{1} = a$$

A3: Todo elemento a tiene un simétrico $\bar{a}$ tal que:

$$a + \bar{a} = \mathbf{1} \text{ y } a \cdot \bar{a} = \mathbf{0}$$

A4: Las operaciones suma y producto son conmutativas:

$\forall a, b \in B$,

$$a + b = b + a$$
$$a \cdot b = b \cdot a$$

A5: Las operaciones suma y producto son asociativas:

$\forall a, b, c \in B$,

$$a + (b + c) = (a + b) + c$$
$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

A6: La suma es distributiva con respecto al producto, y viceversa:

$\forall a, b, c \in B,$

$$a + (b \cdot c) = (a + b) \cdot (a + c)$$
$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

Teniendo en cuenta esta definición, se pueden demostrar teoremas o leyes (utilizando la sustitución como regla de transformación), como:

Idempotencia:

$\forall a \in B,$

$$a + a = a$$
$$a \cdot a = a$$

Absorción:

$\forall a, b \in B,$

$$a + \mathbb{1} = \mathbb{1}$$
$$a \cdot \mathbb{0} = \mathbb{0}$$
$$a + a \cdot b = a$$
$$a \cdot (a + b) = a$$

De Morgan:

$\forall a, b \in B,$

$$\overline{a + b} = \overline{a} \cdot \overline{b}$$
$$\overline{a \cdot b} = \overline{a} + \overline{b}$$

etcétera.

Las operaciones «$\neg$», «$\vee$» y «$\wedge$» definidas para la interpretación binaria (Definición 3.2.1) cumplen, como es fácil comprobar, los axiomas del álgebra de Boole para el conjunto $V = \{0, 1\}$, con $\mathbb{0} = 0$ y $\mathbb{1} = 1$. Por tanto,

$$\langle \{0, 1\}, \vee, \wedge, \neg \rangle$$

tiene estructura de álgebra de Boole.

Pero hay algo más interesante: la estructura de álgebra de Boole de las sentencias, o, con mayor rigor, de las clases de equivalencia entre sentencias.

4.2. Algebra de Boole de las clases de equivalencia entre sentencias

Veamos, en primer lugar, que si $A \equiv A'$, entonces $\neg A \equiv \neg A'$. En efecto, lo primero, por definición de equivalencia, es lo mismo que decir que $(\forall i_j)(I_j(A) = I_j(A'))$ y, por tanto $(\forall i_j)(\neg I_j(A) = \neg I_j(A'))$, de donde (por definición de I):

$(\forall i_j)(I_j(\neg A) = I_j(\neg A')$, es decir, $\neg A \equiv \neg A'$. Esto nos permite decir que si A_1, A_2, ... están en la clase de equivalencia C_i, entonces $\neg A_1$, $\neg A_2$, ... están todas en otra clase de equivalencia, a la que llamaremos $\neg C_i$.

Del mismo modo, se demuestra que si $A \equiv A'$ y $B \equiv B'$, entonces $(A\ k\ B) \equiv (A'\ k\ B')$. Ello nos permite decir que si A_1, A_2, ... están todas en la clase de equivalencia C_i y B_1, B_2, ... están todas en la clase de equivalencia C_j, entonces todas las sentencias formadas por una sentencia cualquiera del primer grupo y otra del segundo unidas por una conectiva binaria, $A_1\ k\ B_m$, estarán en una misma clase de equivalencia, a la que llamaremos $C_i\ k\ C_j$.

De este modo, hemos definido las operaciones «$\neg$», «$\vee$», «$\wedge$», etc., en el conjunto $E_n = S_n/\equiv$ de las clases de equivalencia de las sentencias con n variables proposicionales. (Obsérvese que ahora utilizamos los mismos símbolos con un tercer significado: operaciones en el conjunto E_n; los otros dos significados son los que recordábamos en la nota que acompañaba al Teorema 3.1.4).

Ahora es fácil comprobar que E_n junto con «$\neg$», «$\vee$» y «$\wedge$» satisface los axiomas del álgebra de Boole con los elementos neutros $\mathbb{0} = C_0$ (clase de equivalencia correspondiente a las contradicciones) y $\mathbb{1} = C_N$ (clase de equivalencia correspondiente a las tautologías, en la que el subíndice N depende del número de clases de equivalencia que se formen en la partición, que, a su vez, depende del conjunto de valores semánticos, V; en el caso de que $V = \{0, 1\}$, sabemos que $N = 2^{2^n}$). Comprobémoslo para uno de los axiomas, por ejemplo, el que se refiere a la existencia de elementos simétricos (A3):

Tendremos que demostrar que, para todo C_i, se verifica que $C_i \vee \neg C_i = C_N$ y $C_i \wedge \neg C_i = C_0$. Sabemos que dada una sentencia cualquiera de un C_i cualquiera, $A \in C_i$, se cumple que $\neg A \in \neg C_i$, que $A \vee \neg A$ es una tautología, y, por tanto $(A \vee \neg A) \in C_N$, y que $A \wedge \neg A$ es una contradicción y, por tanto $(A \wedge \neg A) \in C_0$. Pero, por otra parte, tal como se han definido las operaciones en E_n, también sabemos que $(A \vee \neg A) \in (C_i \vee \neg C_i)$ y que $(A \wedge \neg A) \in (C_i \wedge \neg C_i)$. Ello nos permite concluir que $C_i \vee \neg C_i = C_N$ y $C_i \wedge \neg C_i = C_0$.

Como consecuencia de la estructura de álgebra de Boole de la lógica de proposiciones, las transformaciones que estudiaremos en el capítulo 3 y que aplicaremos a los circuitos (formas canónicas, formas mínimas, etc.), son también aplicables a las sentencias de la lógica de proposiciones.

4.3. Dos teoremas del álgebra de Boole

Los dos teoremas que vamos a enunciar (omitiendo su demostración, que exige el recurso a ciertos conceptos, como relaciones de orden parcial, que ya desbordan el alcance de este libro) nos serán de utilidad en el capítulo 3.

Definición 4.3.1. Dada un álgebra de Boole $\langle B, +, \cdot, \bar{} \rangle$, se llama *elemento atómico* a cualquier $b \in B$ $(b \neq 0)$ tal que para todo $a \in B$ se cumple que o bien, $a \cdot b = 0$ o bien, $a \cdot b = a$.

Teorema 4.3.2. Dada un álgebra de Boole $\langle B, +, \cdot, \bar{} \rangle$ cuyos elementos atómicos

son $b_1, b_2, \ldots, b_n$, todo $a \in B$ $(a \neq 0)$ puede expresarse de manera única como suma de elementos atómicos:

$$a = b_\alpha + b_\beta + \ldots + b_\gamma \quad (1 \leq \alpha, \beta, \ldots, \gamma \leq n)$$

Por ejemplo, supongamos $n = 2$, es decir, consideremos las sentencias que pueden formarse con dos variables proposicionales, p y q, y centrémonos en la interpretación binaria ($V = \{0, 1\}$). Sabemos que en este caso hay 16 clases de equivalencia, cuyas interpretaciones son las dadas en el apartado 3.5. Los elementos atómicos serán aquellas clases cuyas interpretaciones sean siempre «0» excepto para una sola interpretación de p y q, es decir (véase la tabla del apartado 3.5), C_1, C_2, C_4 y C_8. Puede comprobarse, en efecto, que el resultado del producto (conjunción) de cualquier otra clase por una de éstas es siempre ésta misma o C_0. Y también que cualquier otra clase (salvo C_0) puede expresarse como suma (disyunción) de varios elementos atómicos: $C_{11} = C_1 + C_2 + C_8$, etc.

Definición 4.3.3. Dos álgebras de Boole, $\langle B_1, +_1, \cdot_1, \bar{}^1 \rangle$ y $\langle B_2, +_2, \cdot_2, \bar{}^2 \rangle$ son *isomorfas* si existe una aplicación biyectiva h: $B_1 \to B_2$ tal que se cumple:

$$(\forall a \in B_1)(h(\bar{a}^1) = \overline{h(a)}^2)$$
$$(\forall a, b \in B_1)(h(a +_1 b) = h(a) +_2 h(b)$$
$$(\forall a, b \in B_1)(h(a \cdot_1 b) = h(a) \cdot_2 h(b)$$

Teorema 4.3.4. Dos álgebras de Boole que tengan el mismo número de elementos ($\text{card}(B_1) = \text{card}(B_2)$) son isomorfas.

5. SISTEMAS INFERENCIALES

5.1. Análisis y generación de razonamientos

La lógica es, entre otras cosas, una herramienta para analizar los procesos de razonamiento que habitualmente se expresan en el lenguaje ordinario. Pero dada la diversidad de matices del lenguaje, no puede pensarse en una traducción automática al lenguaje lógico-formal (al menos, no todavía; esto está íntimamente relacionado con uno de los campos de investigación en inteligencia artificial, el del procesamiento del lenguaje natural). Ahora bien, una vez obtenida la traducción formalizada de un determinado proceso de razonamiento, puede analizarse éste, y puede completarse con nuevas conclusiones de modo automático.

En la presentación del Ejemplo 1.4.2 veíamos que los razonamientos pueden formalizarse como sentencias condicionales cuyo antecedente es la conjunción de las premisas y cuyo consecuente es la conclusión, y que la condición necesaria y suficiente para que el razonamiento sea válido es que la sentencia así formada sea una tautología (o, equivalentemente, una tesis). Esto nos permite analizar razonamientos, pero no obtener conclusiones de modo automático. Lo que necesitamos ahora no es un

sistema axiomático con el que podemos demostrar, por ejemplo, que $\vdash A$, sino un procedimiento para que, dadas $P_1, P_2, \ldots$, podamos obtener $C_1, C_2, \ldots$, tales que

$$\vdash (P_1 \wedge P_2 \wedge \ldots \rightarrow C_1)$$
$$\vdash (P_1 \wedge P_2 \wedge \ldots \rightarrow C_2)$$
$$\ldots$$

Es decir, expresado en otros términos, lo que queremos no es poder demostrar teoremas a partir de unos axiomas, lo cual tiene un interés puramente teórico, sino poder derivar conclusiones a partir de unas premisas que no son tautologías, pero que sabemos (o suponemos) que son verdaderas en una determinada situación.

Pues bien, si disponemos de un repertorio de teoremas que tengan la forma $A_1 \wedge A_2 \wedge \ldots \rightarrow B$, podemos pensar en el siguiente procedimiento: elegir una tesis tal que su antecedente se ajuste exactamente a una premisa o a la conjunción de dos o más de ellas; aplicándolo, obtenemos como conclusión el consecuente de la tesis, que se añade al conjunto de premisas, y repetir el proceso hasta que ya no puedan obtenerse más conclusiones.

Llamaremos *inferencias* a los procesos mediante los cuales obtenemos una conclusión a partir de unas premisas de modo tal que el razonamiento es válido. Una *regla de inferencia* será la declaración de las condiciones bajo las cuales puede hacerse una inferencia, así como del resultado de la misma.

Por ejemplo, consideremos un razonamiento que se formaliza así:

$$[(p \rightarrow \neg q) \wedge (\neg q \rightarrow r)] \rightarrow (p \rightarrow r);$$

Para analizar tal razonamiento y averiguar si es correcto, hemos de ver que la sentencia es un teorema, o, equivalentemente, que es una tautología. Es decir, podemos proceder de dos maneras:

a) Tratando de demostrar la sentencia en el sistema axiomático. (En este caso, basta aplicar la sustitución $(\neg q/q)$ en la primera de las leyes de transitividad dadas en el apartado 2.5.6).
b) Construyendo la tabla de verdad, para ver si la sentencia es una tautología, que es lo que hacíamos en los ejemplos del apartado 1.4. Esta forma podría parecer, en principio, más fácil, pero obsérvese que, por ejemplo, el teorema

$$[(p_1 \rightarrow p_2) \wedge (p_2 \rightarrow p_3) \wedge \ldots \wedge (p_9 \rightarrow p_{10})] \rightarrow (p_1 \rightarrow p_{10})$$

requeriría una tabla de $2^{10} = 1.024$ líneas (una por cada interpretación del conjunto de variables proposicionales).

Pero lo que más nos interesa no es el análisis del razonamiento, sino su generación. En este caso, dadas las premisas

P_1: $p \rightarrow \neg q$
P_2: $\neg q \rightarrow r$

¿qué conclusión (o conclusiones) podemos obtener? Pues bien, dado que la sentencia anteriormente escrita es un teorema en forma de condicional que tiene como antecedente la conjunción de P_1 y P_2, podemos afirmar como conclusión el consecuente:

C: $p \rightarrow r$

5.2. Leyes y reglas de inferencia

Por lo que acabamos de ver, a toda tesis (o ley) del cálculo proposicional que tenga la forma $P_1 \wedge P_2 \wedge \ldots \rightarrow C$ puede hacérsele corresponder una regla de inferencia. (Y también, desde luego, a cada axioma, aunque ninguno de los del sistema PM tienen esa forma).

Pero *ley* y *regla* no son la misma cosa. La diferencia es lingüística: una ley, o teorema, pertenece al *lenguaje* del cálculo, y *representa* a una regla o esquema válido de inferencia; la regla, entonces, pertenece al *metalenguaje* del cálculo.

Por ejemplo, la ley de modus ponens es:

$$[p \wedge (p \rightarrow q)] \rightarrow q$$

o bien, si A y B son sentencias, utilizando la sustitución $\{A/p, B/q\}$,

$$[A \wedge (A \rightarrow B)] \rightarrow B$$

La correspondiente regla de inferencia se expresaría así:

'De A y de $A \rightarrow B$ puede inferirse B'.

'Puede inferirse' pertenece al metalenguaje del cálculo y establece una relación de deducibilidad que sería abusivo representar por «→», porque este último símbolo pertenece al lenguaje. Por ello, la forma habitual de simbolizar esa regla de inferencia es la siguiente:

$$\frac{\begin{array}{c} A \\ A \rightarrow B \end{array}}{B}$$

Nótese que, aun guardando cierta relación, esta regla de inferencia no es exactamente igual que la regla de transformación RT3 (regla de separación). En efecto, esta última dice: 'si A y $A \rightarrow B$ son tesis (o sea, *siempre* son verdaderas), B también lo es', mientras que la regla de inferencia correspondiente al modus ponens dice: '*en el caso* de que tanto A como $A \rightarrow B$ sean verdaderas, B también lo es'.

La generalización para escribir cualquier regla de inferencia es evidente: cada condición se escribe en una línea, y la conclusión en una final, bajo una raya. El lector no tendrá dificultades para escribir en esta forma, por ejemplo, las reglas correspon-

dientes a las leyes de modus tollens, de transitividad, de inferencia de la alternativa, del silogismo disyuntivo y del dilema constructivo, dadas en el apartado 2.5.6.

En los tratados de lógica se presentan conjuntos seleccionados de reglas de inferencia bajo el nombre de «sistemas de deducción natural». El problema para su aplicación en un sistema automático es que, para cada problema, es preciso tener una cierta «habilidad» o «conocimiento» (que se expresaría en forma de metarreglas) para saber en qué orden aplicar las reglas y a qué premisas. Más adelante veremos una formulación que permite trabajar con una sola regla de inferencia. En cualquier caso, el conjunto de reglas que se utilicen debe ser *consistente*, y, a ser posible, *completo*. (Por razones prácticas, en los sistemas pensados para programarse en ordenador a veces se sacrifica la completitud en aras de la eficacia). Estos conceptos son paralelos a los definidos para un sistema axiomático: Así como un sistema axiomático es completo si toda tautología es una tesis, un sistema inferencial lo es si toda conclusión de un razonamiento correcto puede inferirse; y así como un sistema axiomático es consistente si toda tesis es una tautología, un sistema inferencial lo es si toda inferencia corresponde a un razonamiento correcto. Vamos a precisarlo con algo más de detalle.

5.3. Inferencia y deducción

Haremos aquí una distinción entre dos términos que en el lenguaje normal se consideran sinónimos. Ya hemos definido el concepto de *inferencia* (apartado 5.1): un proceso consistente en aplicar ciertas reglas a unas premisas para obtener una conclusión. También hemos dicho (Definición 3.5.1) que una interpretación i_j *satisface* a una sentencia A si (y sólo si) $I_j(A) = 1$.

Definición 5.3.1. Una sentencia C (conclusión) *se deduce de* un conjunto de sentencias $P_1, P_2, \ldots, P_n$ (premisas) si (y sólo si) toda interpretación que satisface $P_1 \wedge P_2 \wedge \ldots \wedge P_n$ también satisface C.

La condición (necesaria y suficiente) que establece esta definición para que C sea deducible equivale a decir que la sentencia $[(P_1 \wedge P_2 \wedge \ldots \wedge P_n) \rightarrow C]$ ha de ser una tautología, criterio que era, precisamente, el que utilizábamos en el apartado 1.4 para analizar razonamientos.

Ilustremos esta diferencia entre inferencia y deducción reconsiderando el Ejemplo 1.4.3. Teníamos las premisas:

$P1$: $c \rightarrow s$
$P2$: $s \leftrightarrow \neg m$

$P2$ puede descomponerse en:

$P2a$: $s \rightarrow \neg m$
$P2b$: $\neg m \rightarrow s$

(Porque $(s \leftrightarrow \neg m) \leftrightarrow (s \rightarrow \neg m) \wedge (\neg m \rightarrow s)$ es un teorema, y, por tanto, *P2* y *P2a* $\wedge$ *P2b* son equivalentes).

De acuerdo con las leyes de transitividad (apartado 2.5.6), una regla de inferencia es:

$$\frac{\begin{array}{l} A \rightarrow B \\ B \rightarrow C \end{array}}{A \rightarrow C}$$

Aplicándola a las premisas *P1* y *P2a* se infiere la conclusión:

$$C\text{: } c \rightarrow \neg m$$

Lo que hacíamos en la exposición del Ejemplo 1.4.3 era ver que la sentencia $(P1 \wedge P2) \rightarrow C$ es una tautología, es decir, comprobar que *C* se deduce de *P1* y *P2*.

¿Es *C* la única conclusión que puede deducirse? No, porque, según nuestra definición, basta con que una sentencia se satisfaga para todas las interpretaciones que satisfacen a las premisas para que tal sentencia sea una conclusión, siendo indiferente lo que ocurra con las otras interpretaciones. Rehaciendo la tabla de verdad, podemos poner en ella diversas conclusiones:

c	s	m	$P1$	$P2$	$P1 \wedge P2$	C	C'	C''	C'''
0	0	0	1	0	0	1	1	0	1
0	0	1	1	1	1	1	1	1	1
0	1	0	1	1	1	1	1	1	1
0	1	1	1	0	0	1	1	1	1
1	0	0	0	0	0	1	0	0	1
1	0	1	0	1	0	0	0	1	0
1	1	0	1	1	1	1	1	1	1
1	1	1	1	0	0	0	0	1	0

$C : c \rightarrow \neg m$
$C' : c \rightarrow s \wedge \neg m$
$C'' : \neg s \rightarrow m$
C''': $m \rightarrow \neg c$

etcétera.

Y, en general, se deduce cualquier sentencia que se satisfaga para i_1, i_2 e i_6. (Entre ellas habrá muchas conclusiones triviales, como las propias premisas, la conjunción de las mismas, etc., y, desde luego, cualquier tautología: según la definición de «deducción», toda tautología «se deduce» de cualquier conjunto de premisas).

Pero C', C'', C''', etc., también pueden obtenerse por inferencia: C' se infiere de *P1* y de *C* (por la regla de inferencia que corresponde al teorema $(p \rightarrow q) \wedge (p \rightarrow r) \rightarrow (p \rightarrow q \wedge r)$); C'', de *P2a* (teorema: $(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$); C''', de *C*, etc.

5.4. Completitud y consistencia de un sistema inferencial

Definición 5.4.1. Un sistema inferencial es un conjunto de reglas de inferencia junto con unas metarreglas que especifican cómo se aplican las reglas. Estas metarreglas constituyen la *estrategia* del sistema: orden de aplicación de las reglas, orden de elección de las premisas, orden en que se añaden las conclusiones a las premisas para obtener nuevas conclusiones, etc.

Definición 5.4.1. Diremos que un sistema inferencial es *completo* si, para cualquier conjunto de premisas, el sistema infiere toda conclusión que pueda deducirse de las premisas.

Definición 5.4.2. Diremos que un sistema inferencial es *consistente* si, para cualquier conjunto de premisas, toda conclusión que infiera el sistema también se deduce de las premisas.

Teorema 5.4.3. Todo sistema inferencial cuyas reglas de inferencia se puedan formalizar como tesis de un sistema axiomático consistente es un sistema inferencial consistente.

Demostración:

Sean P_1, P_2, ..., P_n las premisas y C la conclusión de una regla de inferencia cualquiera. La hipótesis del teorema establece que:

$$\vdash [(P_1 \wedge P_2 \wedge \ldots \wedge P_n) \rightarrow C]$$

Y como el sistema axiomático es consistente, esto implica que:

$$\models [(P_1 \wedge P_2 \wedge \ldots \wedge P_n) \rightarrow C]$$

Ello significa que toda interpretación que satisfaga $(P_1 \wedge P_2 \ldots \wedge P_n)$ también satisface C (porque si en algún caso no fuese así, el condicional se interpretaría como falso y la sentencia no sería una tautología), y, por definición de deducción, que C se deduce de las premisas. Como todas las inferencias se obtienen por aplicación de las reglas de inferencia, y lo anterior es válido para cualquier regla de inferencia, concluimos que toda sentencia que pueda inferir el sistema se deduce de las premisas.

5.5. Forma clausulada de la lógica de proposiciones

La regla de inferencia que veremos en el apartado 5.7 se aplica únicamente a una forma especial de sentencias, pero vamos a ver que toda sentencia de la lógica de proposiciones puede expresarse de modo equivalente en esa forma.

Definición 5.5.1. Una *cláusula* es una sentencia de la forma:

$$l_1 \vee l_2 \vee \ldots \vee l_n$$

Es decir, una cláusula es una disyunción de literales (recuérdese que un «literal» es una variable proposicional sola o con la negación). Por ejemplo: $\neg p \vee q, p \vee q \vee r$, $p \vee \neg q \vee r \vee \neg s$, etc.

Definición 5.5.2. Diremos que una sentencia está en *forma clausulada* si tiene la forma:

$$(l_{11} \vee l_{12} \vee \ldots) \wedge (l_{21} \vee l_{22} \vee \ldots) \wedge \ldots$$

Es decir, una sentencia en forma clausulada es una conjunción de cláusulas.

Dado que las operaciones de conjunción y disyunción son asociativas y conmutativas, podemos decir que:

a) una cláusula es una colección de literales (implícitamente unidos por disyunciones); y
b) una sentencia en forma clausulada es una colección de cláusulas (implícitamente unidas por conjunciones).

Teorema 5.5.3. Para toda sentencia de la lógica proposicional existe una sentencia equivalente en forma clausulada.

Demostración:

La demostración será constructiva, es decir, vamos a ver un procedimiento para pasar cualquier sentencia a forma clausulada. Supondremos que en la sentencia original no se utilizan más que las cuatro conectivas binarias más conocidas. (Ya sabemos que cualquiera otra puede expresarse en función de una de ellas y de la negación). El procedimiento, que, naturalmente, se basa en teoremas de tipo equivalencia, consta de tres pasos:

1. Eliminación de condicionales y bicondicionales mediante las equivalencias derivadas de la definición de ambos:

$$(A \rightarrow B) \leftrightarrow (\neg A \vee B)$$
$$(A \leftrightarrow B) \leftrightarrow (\neg A \vee B) \wedge (A \vee \neg B)$$

2. Introducción de negaciones de modo que queden afectando sólo a variables proposicionales, mediante aplicación sucesiva de las equivalencias derivadas de las leyes de de Morgan:

$$\neg (A \wedge B) \leftrightarrow \neg A \vee \neg B$$
$$\neg (A \vee B) \leftrightarrow \neg A \wedge \neg B$$

así como de:

$$\neg \neg A \leftrightarrow A$$

Con ello, se habrá llegado a una sentencia formada por literales unidos por las conectivas «∨» y «∧».

3. Paso a forma clausulada, distribuyendo «∧» sobre «∨» mediante la equivalencia:

$$[(A_1 \wedge A_2) \vee A_3] \leftrightarrow [(A_1 \vee A_3) \wedge (A_2 \vee A_3)]$$

Con frecuencia, la forma obtenida puede simplificarse. Así, si dentro de una cláusula aparece dos o más veces el mismo literal, se escribe una sola vez ($l \vee l \vee l \vee \ldots \leftrightarrow l$); si aparece un literal y su complementario ($p \vee \neg p$), la cláusula es una tautología, y puede eliminarse del conjunto de cláusulas; si hay dos o más cláusulas idénticas, se escribe una sola de ellas.

Por ejemplo, pasemos a forma clausulada la sentencia:

$$\neg \{[(p \wedge q) \rightarrow p] \rightarrow [(q \vee r) \wedge (\neg q \wedge \neg r)]\}$$

1. $\neg \{\neg [\neg (p \wedge q) \vee p] \vee [(q \vee r) \wedge (\neg q \wedge \neg r)]\}$
2. $[\neg (p \wedge q) \vee p] \wedge \neg [(q \vee r) \wedge (\neg q \wedge \neg r)]$
   $[(\neg p \vee \neg q) \vee p] \wedge [\neg (q \vee r) \vee \neg (\neg q \wedge \neg r)]$
   $[\neg p \vee \neg q \vee p] \wedge [(\neg q \wedge \neg r) \vee (q \vee r)]$
3. $[\neg p \vee \neg q \vee p] \wedge [(\neg q \vee q \vee r)] \wedge [\neg r \vee q \vee r]$

En este ejemplo, cada una de las tres cláusulas obtenidas es una tautología, indicando así que la sentencia original era un teorema. De acuerdo con las reglas de simplificación, las tres cláusulas pueden hacerse desaparecer, pero, para no confundir con la cláusula vacía, que, como veremos luego, corresponde a todo lo contrario (una contradicción), representaríamos el resultado final como la única cláusula $\neg p \vee p$.

Definición 5.5.4. Diremos que trabajamos en la *forma clausulada de la lógica* cuando expresamos todas las sentencias en forma clausulada.

La forma clausulada es más concisa (aunque menos natural) que la forma «estándar» de la lógica. Por ejemplo, las seis sentencias:

$$p \wedge q \rightarrow r;\ p \wedge \neg r \rightarrow \neg q;\ q \wedge \neg r \rightarrow \neg p;$$
$$p \rightarrow \neg q \vee r;\ q \rightarrow \neg p \vee r;\ \neg r \rightarrow \neg p \vee \neg q;$$

se escriben en forma clausulada de la misma manera:

$$\neg p \vee \neg q \vee r$$

Sin embargo, aunque este ejemplo pueda inducir a pensar lo contrario, la forma

clausulada no es única: pueden existir sentencias que estén en formas clausuladas diferentes y que sean equivalentes. Para ilustrarlo, consideremos esta sentencia:

$$(p \leftrightarrow q) \wedge \neg (p \wedge q \wedge r)$$

La aplicación del método expuesto nos conduce a la forma clausulada:

$$(\neg p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee \neg q \vee \neg r)$$

Y, como puede comprobarse fácilmente, estas otras dos formas clausuladas son equivalentes a ella:

$$(\neg p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee \neg r)$$
$$(\neg p \vee q) \wedge (p \vee \neg q) \wedge (\neg q \vee \neg r)$$

5.6. Las cláusulas como sentencias condicionales

Dada una cláusula cualquiera, $l_1 \vee l_2 \vee \ldots$, podemos hacer otras transformaciones, basadas también en equivalencias, que nos permiten escribirla como una sentencia condicional. Para ello, escribimos primero los literales negativos y luego los positivos:

$$\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k \vee q_1 \vee q_2 \vee \ldots \vee q_m$$

Por la generalización de una de las leyes de de Morgan, esta sentencia es equivalente a:

$$\neg (p_1 \wedge p_2 \wedge \ldots \wedge p_k) \vee q_1 \vee q_2 \vee \ldots \vee q_m$$

Y por la ley $(\neg A \vee B) \leftrightarrow (A \rightarrow B)$, esta otra sentencia también es equivalente:

$$p_1 \wedge p_2 \wedge \ldots \wedge p_k \rightarrow q_1 \vee q_2 \vee \ldots \vee q_m$$

Con esta nueva escritura vemos que la interpretación metalógica de una cláusula es: «si se dan como antecedentes todos los literales negativos, entonces se siguen como consecuentes uno o varios de los literales positivos».

Hay algunos casos particulares de cláusulas que tienen un interés especial en las aplicaciones:

a) *Cláusulas de Horn con cabeza.* Son las que sólo tienen un literal positivo:

$$(\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k \vee q) \equiv (p_1 \wedge p_2 \wedge \ldots \wedge p_k \rightarrow q)$$

b) *Cláusulas de Horn sin cabeza.* Son las que no tienen ningún literal positivo:

$$(\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k) \equiv \neg (p_1 \wedge p_2 \wedge \ldots \wedge p_k)$$

En este caso, las variables proposicionales $p_1, p_2, \ldots, p_k$ son *incompatibles*, es decir, no es posible que sean todas verdaderas.

c) Las que no tienen ningún literal negativo:

$$(q_1 \vee q_2 \vee \ldots \vee q_m)$$

En este caso, al menos una de las variables es verdadera.

d) La *cláusula vacía*, λ, en la que han desaparecido todos los literales, y que, como veremos, aparece en la inferencia cuando las premisas son incompatibles.

5.7. La regla de resolución

Definición 5.7.1. La regla de inferencia llamada *resolución* se aplica a dos premisas en forma de cláusulas, tales que tengan en común un literal positivo en una y negativo en otra, y a las que llamaremos *generatrices*. La inferencia consiste en construir otra cláusula, llamada *resolvente*, formada por la disyunción de todos los literales de las generatrices salvo el común.

Por ejemplo:

$$\frac{p \vee \neg q \qquad \neg p \vee r \vee s}{\neg q \vee r \vee s}$$

Como toda regla de inferencia, la resolución se fundamenta en una tesis, concretamente, en

$$\vdash [(\neg p \vee A) \wedge (p \vee B) \rightarrow (A \vee B)]$$

que es una generalización de las leyes de inferencia de la alternativa. Por consiguiente, de acuerdo con el Teorema 5.4.3, todo sistema inferencial basado en la resolución será consistente.

Las reglas de inferencia «clásicas» pueden expresarse como resoluciones, si previamente se escriben las premisas en forma clausulada. Por ejemplo:

* Modus ponens: $\dfrac{A \qquad A \rightarrow B}{B}$

Resolución: $\dfrac{A \qquad \neg A \vee B}{B}$

* Modus tollens: $\dfrac{\neg B \qquad A \rightarrow B}{\neg A}$

Resolución: $\dfrac{\neg B \qquad \neg A \vee B}{\neg A}$

* Transitividad : $\dfrac{A \rightarrow B \qquad B \rightarrow C}{A \rightarrow C}$

Resolución: $\dfrac{\neg A \vee B \qquad \neg B \vee C}{\neg A \vee C}$

Para completar el sistema inferencial tenemos qué dar unas metarreglas, o un procedimiento de aplicación de la resolución. El más inmediato es el de la búsqueda exhaustiva:

Definición 5.7.2. Dadas *n* premisas en forma de cláusulas, la *búsqueda exhaustiva* consiste en aplicar la resolución a todas las parejas posibles de cláusulas, añadir las resolventes al conjunto de cláusulas, aplicar la resolución a todas las nuevas parejas, y así sucesivamente, hasta que en la aplicación de la resolución no se obtengan nuevas resolventes.

Por lo dicho anteriormente (Teorema 5.4.3), el sistema es consistente, pero ¿será completo? En principio, no lo es, porque, como sabemos, de dos premisas puede deducirse más de una conclusión, mientras que la resolución sólo produce una (la resolvente). Ahora bien, si aplicamos repetidamente la resolución (añadiendo toda conclusión al conjunto de premisas) y considerando como inferencias no solamente las resolventes que se van obteniendo, sino también todas las conjunciones en el conjunto de premisas y resolventes, todas las cláusulas que se obtienen por disyunción de dos o más cláusulas en ese conjunto y todas las conclusiones «triviales» (entendiendo como tales las disyunciones de cualquier premisa o conclusión con cualquier literal; por ejemplo, de la premisa «p» son conclusiones triviales «$p \vee q$» y «$p \vee \neg q$»), entonces puede verse que se infieren todas las deducciones posibles. Veámoslo con algunos ejemplos.

Ejemplo 5.7.3. Consideremos de nuevo la sentencia estudiada al final del apartado 5.5, y supongamos que corresponde a dos premisas:

$P1$: $(p \leftrightarrow q)$
$P2$: $\neg (p \wedge q \wedge r)$

En forma clausulada:

$P1a$: $(\neg p \vee q)$
$P1b$: $(p \vee \neg q)$
$P2$: $(\neg p \vee \neg q \vee \neg r)$

Si aplicamos la resolución a $P1a$ y a $P2$ obtenemos la resolvente:

$C1$: $(\neg p \vee \neg r)$

Y aplicándola a $P1b$ y $P2$,

$C2$: $(\neg q \vee \neg r)$

Resolviendo con todas las parejas que pueden formarse con $C1$, $C2$ y las premisas, se puede comprobar que no se obtienen más conclusiones. (Por ejemplo, la resolución de $C1$ con $P1b$ conduce de nuevo a $C2$). Si se analizan todas las deducciones posibles (construyendo la tabla de verdad) se comprobará que toda deducción corresponde a algún elemento del conjunto formado por estas cinco cláusulas ($P1a$, $P1b$, $P2$, $C1$, $C2$) más las conclusiones «triviales» (por ejemplo, de $P1a$ son conclusiones triviales $\neg p \vee q \vee r$ y $\neg p \vee q \vee \neg r$) o a una cláusula formada por la disyunción de dos o más de ellas, o a una conjunción de dos o más de ellas.

Ejemplo 5.7.4. En el apartado 5.3 veíamos algunas conclusiones posibles de unas premisas enunciadas en el Ejemplo 1.4.3. Las premisas en forma clausulada son:

$P1$: $\neg c \vee s$
$P2a$: $\neg s \vee \neg m$
$P2b$: $m \vee s$

Resolviendo $P1$ y $P2a$,

C: $\neg c \vee \neg m$

De $P2a$ y $P2b$ se obtienen $\neg s \vee s$, o $\neg m \vee m$, que son tautologías, y de $P2b$ y C, $\neg c \vee s$, que es $P1$. Y, como puede comprobarse, todas las deducciones posibles equivalen a conjunciones o disyunciones en el conjunto $\{P1, P2a, P2b, C\}$. Así, las señaladas en el apartado 5.3:

C' : $c \rightarrow s \wedge \neg m$ es en forma clausulada $(\neg c \vee s) \wedge (\neg c \vee \neg m)$, conjunción de $P1$ y C.
C'' : $\neg s \rightarrow m$ es en forma clausulada igual que $P2a$.
C''': $m \rightarrow \neg c$ es en forma clausulada igual que C.

Ejemplo 5.7.5. Apliquemos ahora la resolución al ejemplo 1.4.4. Las premisas son:

$P1$: $\neg j \vee \neg c$
$P2$: $j \vee \neg d$
$P3$: $\neg s \vee c$

De $P1$ y $P2$, $C1$: $\neg c \vee \neg d$
De $P1$ y $P3$, $C2$: $\neg j \vee \neg s$
De $P2$ y $C2$, $C3$: $\neg d \vee \neg s$

El lector puede comprobar que no hay más conclusiones (salvo disyunciones y conjunciones de las premisas y las tres conclusiones obtenidas). Puede, asimismo, analizar, las interpretaciones en forma de condicional de estas conclusiones. La que se veía en el Ejemplo 1.4.4, $s \rightarrow \neg d$, es, puesta en forma clausulada, $C3$.

5.8. Refutación

La *refutación* es un procedimiento útil cuando lo que se pretende no es generar cuantas conclusiones sean posibles, sino comprobar si una determinada conclusión es válida o no.

Definición 5.8.1. La *refutación* consiste en comprobar que el conjunto de cláusulas formado por las correspondientes a las premisas y la que procede de la conclusión negada es una contradicción, lo cual demuestra que la conclusión se infiere de las premisas.

El fundamento de la refutación es la «ley de reducción al absurdo» (apartado 2.5.6). Si en ese teorema hacemos la sustitución $\{p/(P \rightarrow C)\}$, donde $P = P1 \wedge P2 \wedge \ldots \wedge Pn$ es la conjunción de premisas y C la conclusión a comprobar, resulta:

$$\vdash \{[\neg (P \rightarrow C) \rightarrow (q \wedge \neg q)] \leftrightarrow (P \rightarrow C)\}$$

o, lo que es lo mismo,

$$\vdash \{[(P \wedge \neg C) \rightarrow (q \wedge \neg q)] \leftrightarrow (P \rightarrow C)\}$$

Es decir, «$(P \rightarrow C)$ es verdadera (y, por tanto, C es una conclusión) si (y sólo si) de la conjunción de P y $\neg C$ resulta una contradicción».

Comprobemos que en el último ejemplo que vimos puede inferirse $s \rightarrow \neg d$. Pasemos primero su negación a forma clausulada:

$$\neg (s \rightarrow \neg d) \equiv \neg (\neg s \vee \neg d) \equiv s \wedge d$$

(es decir, resultan dos cláusulas: s y d).

Apliquemos repetidamente la resolución a $\{P1, P2, P3, s, d\}$:

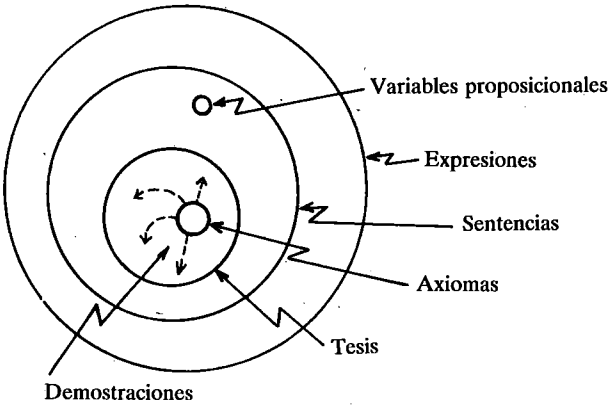
De $P2$ y d resulta como resolvente j;
De $P1$ y j resulta como resolvente $\neg c$;
De $P3$ y $\neg c$ resulta como resolvente $\neg s$;
Finalmente, de $\neg s$ y s resulta una contradicción (cláusula vacía, λ).

Una observación final: si $P1 \wedge P2 \wedge \ldots \wedge Pn$ son ya de por sí una contradicción, la resolución con cualquier $\neg C$ generará siempre la cláusula vacía, es decir, cualquier conclusión es válida, lo cual está de acuerdo con el significado del condicional, y es consistente con la definición de deducción.

6. RESUMEN

En un *sistema axiomático* se establecen unos *axiomas* («verdades indemostrables») y unas reglas de transformación que permiten demostrar *teoremas* o *leyès*. Los

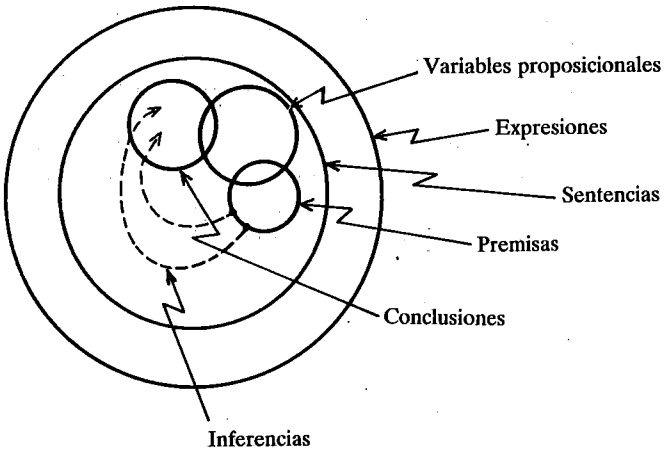
axiomas y los teoremas son las *tesis* del sistema. Podemos ilustrar estas definiciones con un diagrama de Venn:



El concepto de *tautología* es semántico: una sentencia es una tautología si su interpretación es verdadera para todas las interpretaciones posibles de las variables proposicionales que la forman.

El sistema axiomático es *completo* si toda tautología es una tesis ($\models A \rightarrow \vdash A$), y es *consistente* si toda tesis es una tautología ($\vdash A \rightarrow \models A$).

En un *sistema inferencial* no se establecen axiomas, sino *premisas*, que son sentencias supuestamente verdaderas en un determinado contexto, pero que no tienen por qué serlo siempre. Una premisa puede ser, por ejemplo, una simple variable proposicional. La idea es que, *supuesto* que las premisas sean verdaderas, el sistema pueda inferir de ello *conclusiones*. El diagrama ahora podría ser este otro:



La conexión entre el sistema axiomático y el sistema inferencial se produce porque si de las premisas *P*1, *P*2, ..., *Pn* se infiere la conclusión *C*, entonces la sentencia (*P*1 $\wedge$ *P*2 $\wedge$ ... $\wedge$ *Pn* $\rightarrow$ *C*) es una tesis, y viceversa. Por ello, toda tesis que tenga esa forma da lugar a una *regla de inferencia*.

Decimos que una conclusión *se deduce* (a diferencia de «se infiere») de unas premisas si su interpretación es verdadera para toda interpretación que *satisfaga* a las premisas (o sea, que haga verdadera a su conjunción).

Un sistema inferencial es *completo* si toda conclusión que se deduce puede inferirse, y es *consistente* si toda conclusión que se infiere puede deducirse.

En la *forma clausulada de la lógica* todas las sentencias se expresan como colecciones (conjunciones) de *cláusulas*, siendo éstas colecciones (disyunciones) de *literales*. La *resolución* es una regla de inferencia que se aplica a dos cláusulas (*generatrices*) y, si tienen una pareja de literales complementarios, produce como conclusión otra cláusula (*resolvente*).

7. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

Es bien sabido que la lógica, como ciencia del análisis del comportamiento racional, tiene una historia milenaria, y que fue, sobre todo, Aristóteles quien sentó las bases de los desarrollos posteriores. De la lógica aristotélica se puede decir que era formal (atendía a la forma de los razonamientos), pero no formalizada simbólicamente. Algunos filósofos medievales, entre los que cabe destacar al mallorquín Ramón Llull, hicieron algunos avances hacia la formalización de la lógica, pero los trabajos más importantes en este sentido no se realizaron hasta finales del siglo pasado, cuando Boole elaboró su modelo algebraico de la lógica de proposiciones y Fregge formalizó la de predicados, y principios de éste, cuando aparece, como obra culminante, el libro de Whitehead y Russell (1910-1913).

Las conectivas de «negación alternativa» y «negación conjunta» fueron propuestas por Sheffer en 1913, y en lógica tienen un interés puramente teórico, y casi anecdótico. Sin embargo, como veremos en el próximo capítulo, corresponden a funciones de conmutación tecnológicamente importantes (que, respectivamente, se suelen llamar «NAND» y «NOR» en la literatura técnica).

En los años 60 se desarrollaron diversos métodos orientados al procesamiento automático de las inferencias, de los que el más conocido es el de «resolución», debido a Robinson (1965). Lo que aquí hemos visto es la particularización a la lógica de proposiciones de ese método, que en el capítulo 4 expondremos ya en su integridad.

Hay libros de introducción a la lógica cuya lectura, además de recomendable, es muy amena. Destacamos el de Ferrater y Leblanc (1962) (donde se llama «lógica sentencial» a lo que aquí hemos denominado «lógica de proposiciones») y el de Deaño (1974) (donde se le llama «lógica de enunciados»). El Ejemplo 1.4.3 está tomado del primero, y el 1.4.4 del segundo (Deaño hace uso de este ejemplo, procedente de Lewis Carroll, para ilustrar la necesidad de la lógica de predicados; aquí hemos visto que también se puede formalizar en lógica de proposiciones, aunque abusando un poco del lenguaje natural). En el libro de Deaño nos hemos inspirado también para la explicación sobre el significado del condicional y la diferencia entre «leyes» y «reglas

de inferencia». El Ejemplo 1.4.5 es una adaptación de otro similar de Gilbert (1976), un libro sobre álgebra aplicada cuyo capítulo 2 se dedica a la lógica y los circuitos lógicos.

8. EJERCICIOS

8.1. Formalizar las siguientes frases como sentencias proposicionales, y analizar sus tablas de verdad:

«La verdad es una brújula loca que no funciona en este caos de cosas desconocidas» (Baroja).

«Se puede conocer la utilidad de una idea y, sin embargo, no acertar a comprender el modo de utilizarla» (Goethe).

«Ese lapso de tiempo, corto si se le mide por el calendario, es interminablemente largo cuando, como yo, se ha galopado a través de él» (Kafka).

«El mismo diablo citará a la Sagrada Escritura si viene bien a sus propósitos» (Shakespeare).

8.2. Analizar los siguientes razonamientos:

P1: 'Si no llueve, salgo al campo'.
P2: 'Si salgo al campo, respiro'.
C: 'Respiro sí y sólo si no llueve'.

P1: 'Si un monte se quema, algo suyo se quema'.
P2: 'Algo suyo se quema sí y sólo si es usted descuidado'.
P3: 'Si usted no es descuidado, es acreedor a una felicitación'.
C: 'Si usted no es acreedor a una felicitación, entonces es que un monte se quema'.

P1: 'Si un país es una democracia, el Presidente del Gobierno se elige por sufragio universal'.
P2: 'En España, el Presidente no se elige por sufragio universal'.
C: 'Luego España no es una democracia'.

El mismo anterior, modificando así la primera premisa:

P1: 'Si el Presidente de un país es elegido por sufragio universal, ese país es una democracia'.

8.3. Aplicar la resolución y la refutación a los anteriores razonamientos.

8.4. Aplicar la resolución para inferir cuantas conclusiones sean posibles en los siguientes casos, y comprobar los resultados con las deducciones que se obtienen de las tablas de verdad:

á) P1: $p \leftrightarrow q$
P2: $p \rightarrow \neg r$

b) P1: $p \leftrightarrow q$
P2: $r \rightarrow \neg p$

(Contrastar estos dos casos con el del Ejemplo 5.7.4).

c) P1: $p \rightarrow q$
P2: $r \rightarrow p$
P3: $\neg r \rightarrow \neg t$
P4: $\neg (s \wedge \neg r)$
P5: $\neg t \rightarrow s$

d) P1: $p \rightarrow q \vee r$
P2: $q \rightarrow p$

Capítulo 3
CIRCUITOS LOGICOS COMBINACIONALES

1. INTRODUCCIÓN

El hardware puede describirse y estudiarse en varios niveles de abstracción. En el más bajo («nivel físico-electrónico») se consideran los fenómenos físicos básicos y las propiedades de los materiales (semiconductores, metales, dieléctricos) que explican el funcionamiento de los componentes electrónicos (resistores, diodos, transistores, etc.). En el siguiente («nivel electrónico-circuital») se hace abstracción de tales fenómenos: se da por supuesta la existencia de los componentes, con comportamiento funcional conocido, y se estudian los circuitos que resultan de su interconexión. Aquí vamos a movernos en el nivel de abstracción inmediatamente superior (el «nivel lógico»): supuesto que disponemos de unos circuitos básicos, las «puertas lógicas», estudiaremos cómo pueden interconectarse para conseguir sistemas con determinado comportamiento. En este nivel, hacemos abstracción tanto de los detalles del nivel «físico-electrónico» como de los del nivel «electrónico-circuital». Por ejemplo, una de las cosas de las que haremos abstracción será el valor real de los potenciales eléctricos. En los circuitos digitales se opera siempre con sólo dos valores diferentes de tensión. Según sea la tecnología utilizada, así serán esos dos valores, que, en el nivel lógico, los representaremos por los símbolos «0» y «1». El «0» podría corresponder, en el circuito electrónico, a una tensión de 0 voltios (o, con mayor realismo, a un margen, por ejemplo, de 0 a 1 voltios), y el «1» a 5 voltios (o al margen 3-6). Pero también podría ser al revés (corresponder el «0» a 5 voltios y el «1» a 0 voltios), o podríamos tener otros valores u otros márgenes de tensión totalmente diferentes. Nosotros consideraremos perfectamente definido el comportamiento de los componentes básicos («puertas») mediante relaciones de entrada-salida que se refieren sólo a los valores lógicos.

Hay dos ideas esenciales que conviene tener muy claras desde el principio para la buena comprensión de este capítulo:

a) Cualquier punto de un circuito digital, en un instante determinado, sólo puede estar en uno de dos estados (tensiones o márgenes de tensión) determinados: no hay estados intermedios.
b) El valor lógico de un punto de un circuito es la simbolización del estado en que se encuentra ese punto. Este valor lógico puede ser constante (0 ó 1: el punto siempre se encuentra en ese estado) o variable (el punto puede estar en un estado u otro, pero en cada instante tiene que estar en alguno de ellos).

En este capítulo veremos los circuitos digitales más sencillos, los llamados «circuitos combinacionales». En ellos, el valor de la salida en cada momento sólo depende de los valores que en ese momento tengan las entradas, y no de los que hubieran podido tener anteriormente. En el tema «Autómatas» veremos los «circuitos secuenciales», en los que la salida depende de la «historia», es decir, de la sucesión de valores anteriores de las entradas.

Veremos enseguida que el álgebra de Boole, inicialmente propuesta como modelo matemático para la lógica de proposiciones (capítulo 2, apartado 4), es también la herramienta básica para el estudio y el diseño de los circuitos digitales, o circuitos de conmutación, que, por esta misma razón, recordémoslo, se llaman «circuitos lógicos».

2. LAS PUERTAS BÁSICAS

Los bloques elementales para la construcción de circuitos lógicos son las «puertas». Este nombre responde al hecho de que pueden tener una o varias entradas pero una sola salida, y esta salida puede tomar el valor lógico «0» («puerta cerrada») o «1» («puerta abierta»), dependiendo de los valores lógicos que tengan las entradas. Las tres puertas básicas (más adelante veremos otras) son las llamadas «NOT», «OR» y «AND», y los símbolos más utilizados para representarlas en los diagramas son los indicados en la figura 3.1. Su función se corresponde exactamente con la que simbólicamente realizan las conectivas «$\neg$», «$\vee$» y «$\wedge$» en la lógica de proposiciones. Es decir, si siguiésemos el convenio de representar por «p» y «q» las entradas de una puerta «OR», por ejemplo, la salida estaría representada por la sentencia «$p \vee q$», lo que quiere decir que el nivel lógico de esa salida será «0» si los niveles lógicos de las dos entradas son «0», y «1» si una de las entradas, o ambas, tienen el nivel lógico «1».

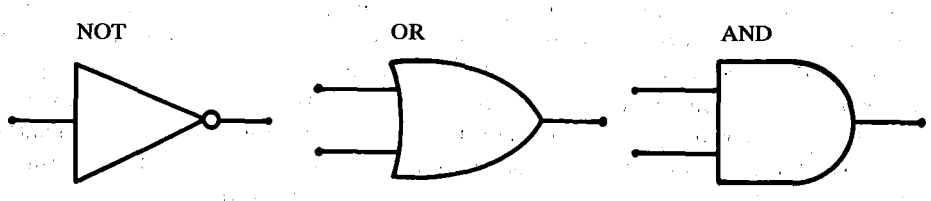


FIGURA 3.1.

En este capítulo vamos, sin embargo, a adoptar otra notación, más habitual cuando se trabaja con el álgebra de Boole: en lugar de «¬», «∨» y «∧» utilizaremos «$\bar{\ }$», «+» y «·», respectivamente. Y para las variables que representan valores lógicos (lo que luego llamaremos «variables booleanas») emplearemos las letras x, y, z, eventualmente con subíndices.

Teniendo en cuenta lo dicho, la figura 3.2, que especifica el funcionamiento de las tres puertas mediante tablas de verdad, se explica por sí sola.

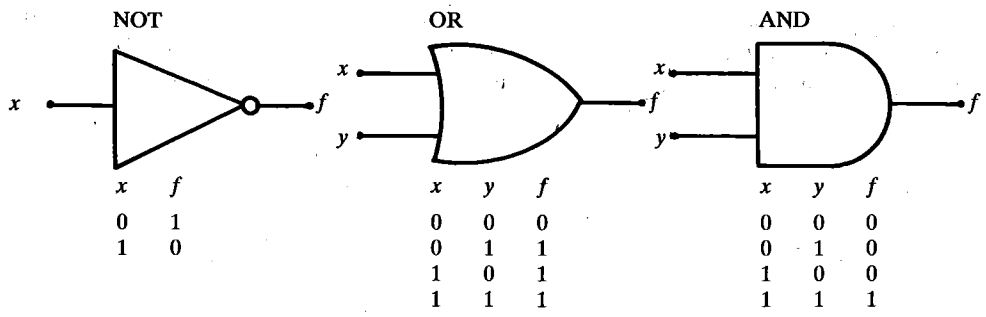


FIGURA 3.2.

Las puertas «OR» y «AND» pueden tener más de dos entradas. Las operaciones que realizan son, formalmente, las mismas operaciones conocidas del álgebra de Boole, y, por tanto, tienen las mismas propiedades, y, concretamente, la propiedad asociativa. Esto nos permite decir, por ejemplo, que el funcionamiento de una puerta «AND» de tres entradas (figura 3.3., izquierda) es el mismo del circuito formado por dos puertas «AND» de dos entradas conectadas según indica la figura 3.3 (derecha). Y lo mismo podríamos decir para las puertas «OR».

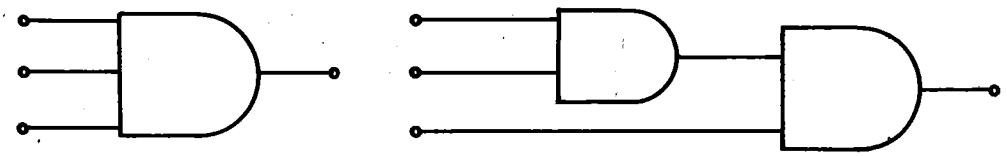


FIGURA 3.3.

3. CIRCUITOS

Las puertas se pueden interconectar teniendo en cuenta la regla de que la salida de una puerta cualquiera puede servir de entrada a una o varias puertas, pero nunca

pueden conectarse juntas dos o más salidas*. Los sistemas así construidos serán circuitos lógicos con una o varias entradas y una o varias salidas. Como existe una correspondencia biunívoca entre las operaciones que realizan las puertas y los operadores del álgebra de Boole, si representamos por variables (x_i, y_i, etc.) los valores lógicos de las entradas del circuito, podremos escribir una fórmula para representar cada salida, en la que intervendrán esas variables y los símbolos « $\bar{}$ », «+» y «·». A cada salida de un circuito lógico corresponderá así una fórmula. Por otra parte, el comportamiento del circuito puede especificarse mediante una tabla de verdad para cada salida que nos dé los valores lógicos que toma esa salida para todas y cada una de las posibles combinaciones de valores lógicos de las entradas. La mejor manera de comprender todo esto es a través de ejemplos. En cada uno de los ejemplos que siguen sólo se considera una salida, cuyo valor lógico se simboliza por «*f*».

Ejemplo 3.1

Fórmula:

$$f = x + y \cdot z$$

Circuito:

El de la figura 3.4.

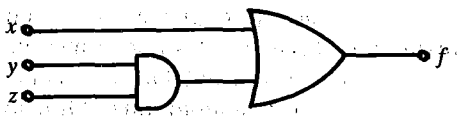


FIGURA 3.4.

Tabla de verdad:

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

* En realidad, esta regla tiene excepciones: por una parte, el número de entradas que puede alimentar la salida de una puerta tiene una limitación tecnológica; por otra, a veces pueden conectarse directamente la salidas de dos o más puertas, consiguiendo una función «OR» o «AND» «implícita». Pero estas excepciones entran ya en el nivel de descripción «electrónico», más que en el «lógico», por lo que no las consideraremos aquí.

Ejemplo 3.2

Fórmula:

$$f = x \cdot \overline{y} + x \cdot z + x \cdot y \cdot \overline{z} + \overline{x} \cdot y \cdot z$$

Circuito:

El de la figura 3.5.

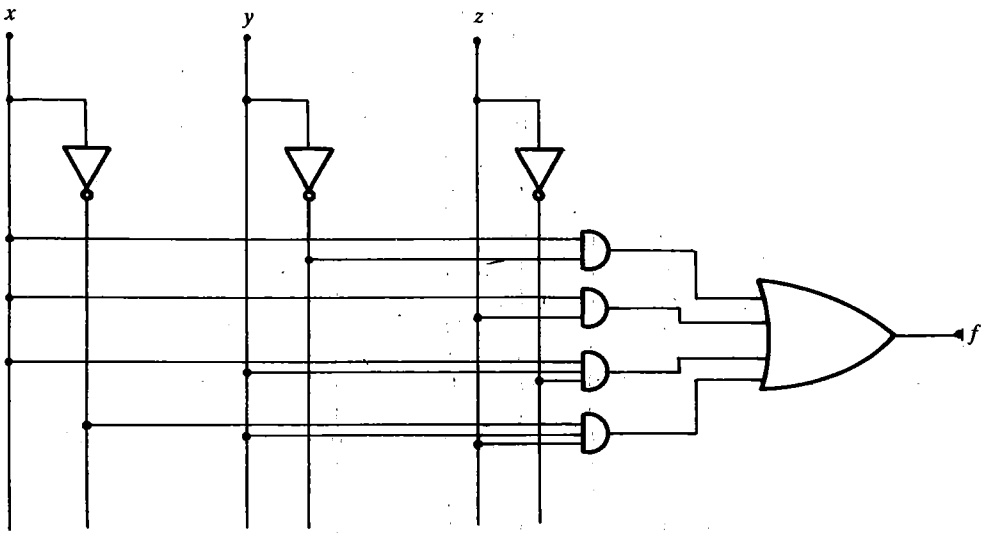


FIGURA 3.5.

Tabla de verdad:

La misma del ejemplo anterior.

Ejemplo 3.3

Fórmula:

$$f = x_1 \cdot (\overline{x}_2 + x_4) + x_3$$

Circuito:

El de la figura 3.6.

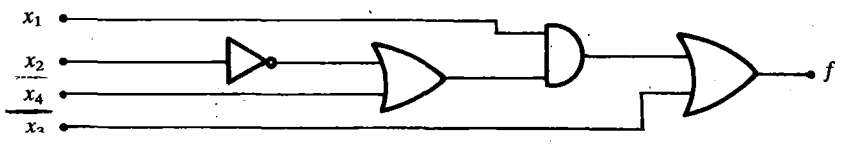


FIGURA 3.6.

Tabla de verdad:

$x1$	$x2$	$x3$	$x4$	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Ejemplo 3.4

Fórmula:

$$f = x_1 \cdot \overline{x}_2 \cdot \overline{x}_3 \cdot \overline{x}_4 + x_1 \cdot \overline{x}_3 \cdot x_4 + x_3$$

Circuito:

El de la figura 3.7.

Tabla de verdad:

La misma del ejemplo anterior.

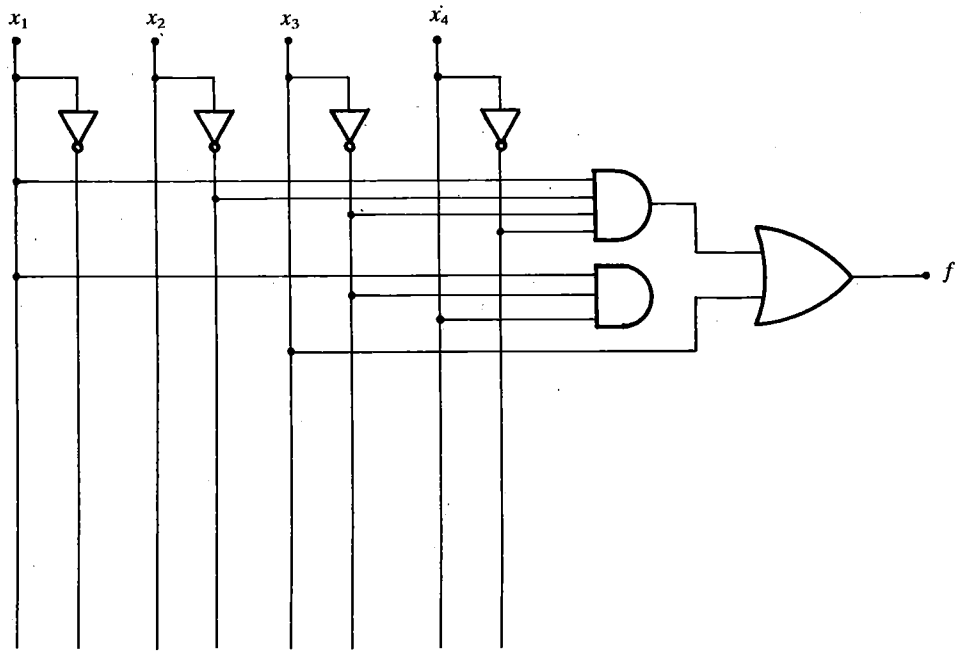


FIGURA 3.7.

Ejemplo 3.5

Fórmula:

$$f = x \cdot y + (x + y) \cdot z$$

Circuito:

El de la figura 3.8.

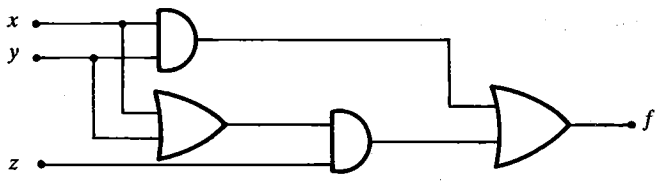


FIGURA 3.8.

Tabla de verdad:

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Ejemplo 3.6

Fórmula:

$$f = x \cdot y \cdot \overline{z} + (x + y) \cdot z$$

Circuito:

El de la figura 3.9.

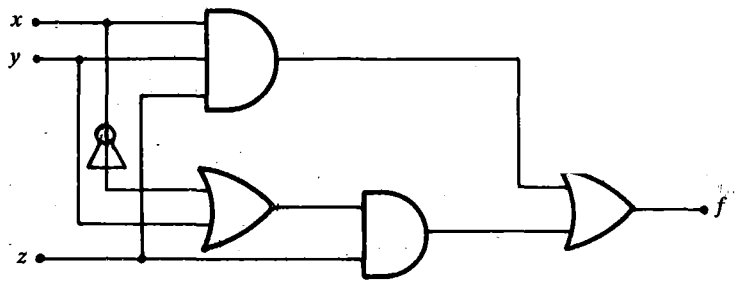


FIGURA 3.9.

Tabla de verdad:

La misma del ejemplo anterior.

Ejemplo 3.7

Fórmula:

$$f = \overline{x} \cdot y \cdot z + x \cdot \overline{y} \cdot z + x \cdot y \cdot \overline{z} + x \cdot y \cdot z$$

Circuito:

El de la figura 3.10.

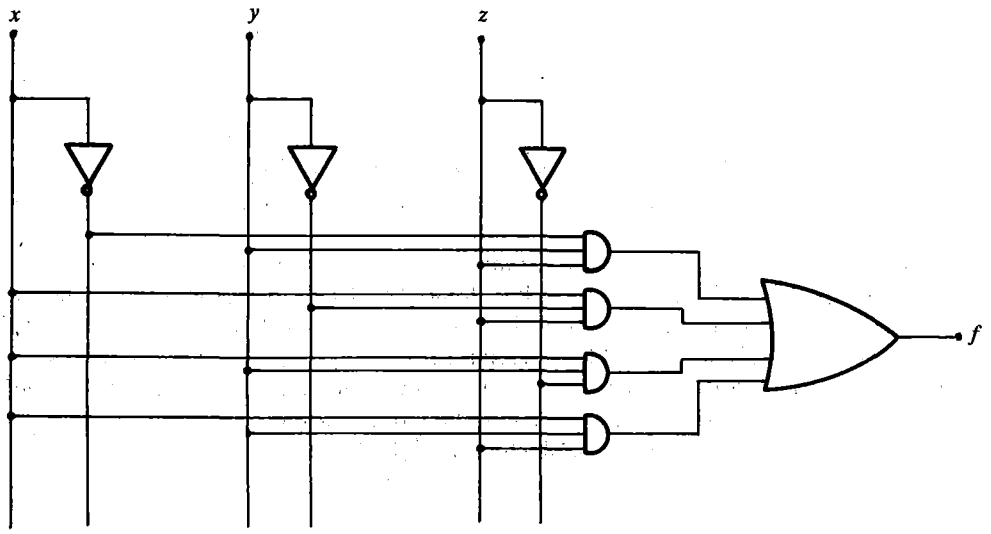


FIGURA 3.10.

Tabla de verdad:

La misma de los dos ejemplos anteriores.

4. MODELOS MATEMÁTICOS DE LOS CIRCUITOS

4.1. Utilidad de los modelos matemáticos

Cuando se aborda la tarea de diseñar un circuito lógico para realizar una determinada función, lo primero es especificar claramente esa función. Con frecuencia, esa especificación, que inicialmente puede ser verbal, conduce a una tabla de verdad para cada una de las salidas del circuito. Ahora bien, en los ejemplos anteriores se ve claramente que la misma tabla de verdad puede realizarse mediante circuitos diferentes. Es claro que por muchas razones (coste, tamaño, fiabilidad, etc.) interesará encontrar el circuito que, respetando las especificaciones, tenga el menor número posible de puertas. Este es el problema de la *minimización*, que trataremos en el siguiente apartado. Para poder abordarlo, necesitamos formalizar algunos conceptos; los modelos matemáticos de los circuitos nos permitirán diseñar éstos de forma óptima.

En los ejemplos hemos visto que la función de cada salida de un circuito puede expresarse mediante una tabla de verdad, pero que esta relación no es biunívoca:

puede haber varios circuitos con la misma tabla; diremos que todos ellos realizan la misma *función de conmutación*. Por otra parte, también se ve en los mismos ejemplos que a cada circuito podemos asociar de manera biunívoca una «fórmula», construida por variables lógicas y operaciones « $\bar{}$ », «+» y «·»; estas fórmulas corresponden a lo que vamos a definir como *formas booleanas*.

4.2. Funciones de conmutación

Definición 4.2.1. Llamaremos *función de conmutación de orden n* a cualquier aplicación $\{0, 1\}^n \to \{0, 1\}$, donde $\{0, 1\}^n$ es el producto cartesiano de orden n de $\{0, 1\}$ consigo mismo.

Así, el dominio de una función de conmutación de orden n está formado por todas las n-tuplas que pueden formarse con los elementos 0 y 1, mientras que el rango es solamente $\{0, 1\}$. La forma más natural de representar una determinada función de conmutación es mediante una tabla de verdad. En los ejemplos 3.1 y 3.2 tenemos una función de conmutación de orden 3, en 3.3 y 3.4 una función de conmutación de orden 4, y en 3.5, 3.6 y 3.7 otra función de conmutación de orden 3.

Consideremos el álgebra de Boole binaria $\langle \{0, 1\}, +, \cdot, \bar{} \rangle$, y llamemos F_n al conjunto de funciones de conmutación de orden n:

$$F_n = \{f\colon \{0, 1\}^n \to \{0, 1\}\}$$

Vamos a definir las operaciones «suma», «producto» y «complementación» en el conjunto F_n. Utilizaremos para ellas los mismos símbolos del álgebra de Boole binaria.

Definición 4.2.2. Si f_i y f_j son dos funciones de conmutación de orden n,

$$f_i\colon X \to f_i(X);\ f_j\colon X \to f_j(X),$$

con $X \in \{0, 1\}^n$; $f_i(X), f_j(X) \in \{0, 1\}$,

se definen:

a) La función complementaria de f_i, $\overline{(f_i)}$, como aquella que hace corresponder a X el elemento complementario del que le corresponde según f_i:

$$\overline{(f_i)}\colon X \to \overline{f_i(X)}$$

b) La función suma de f_i y f_j $(f_i + f_j)$, como aquella que hace corresponder a X la suma de los elementos que le corresponden según f_i y f_j:

$$(f_i + f_j)\colon X \to f_i(X) + f_j(X)$$

c) La función producto de f_i y f_j $(f_i \cdot f_j)$, como aquella que hace corresponder a X el producto de los elementos que le corresponden según f_i y f_j:

$$(f_i \cdot f_j) : X \rightarrow f_i(X) \cdot f_j(X)$$

Ejemplo 4.2.3. Llamando f_1 y f_2 a las funciones de conmutación de orden 3 que corresponden a los ejemplos 3.1 (y 3.2) y 3.5 (y 3.6, 3.7), en la siguiente tabla podemos ver sus correspondientes funciones complementarias, su suma y su producto:

x	y	z	f_1	f_2	$\overline{f_1}$	$\overline{f_2}$	$f_1 + f_2$	$f_1 \cdot f_2$
0	0	0	0	0	1	1	0	0
0	0	1	0	0	1	1	0	0
0	1	0	0	0	1	1	0	0
0	1	1	1	1	0	0	1	1
1	0	0	1	0	0	1	1	0
1	0	1	1	1	0	0	1	1
1	1	0	1	1	0	0	1	1
1	1	1	1	1	0	0	1	1

Teorema 4.2.4. $\langle F_n, +, \cdot, \bar{} \rangle$ es un álgebra de Boole cuyos elementos neutros son las funciones que hacen corresponder todas las n-tuplas a 0 o a 1:

$$\mathbb{0} = f_0, \text{ tal que } f_0(X) = 0 \quad \forall X \in \{0, 1\}^n$$
$$\mathbb{1} = f_N, \text{ tal que } f_N(X) = 1 \quad \forall X \in \{0, 1\}^n$$

y cuyos elementos atómicos son las 2^n funciones que hacen corresponder todas las n-tuplas a 0, excepto una.

Demostración:

Las operaciones sobre funciones de orden n dan como resultado otras funciones de orden n, por lo que F_n es cerrado bajo las tres operaciones. Por otra parte, tal como se han definido los elementos neutros, es inmediato que, para cualquier f_i, resulta que $(f_i + f_0) = f_i$ y que $(f_i \cdot f_N) = f_i$. El resto de los axiomas del álgebra de Boole (capítulo 2, apartado 4.1) se cumplen también por el hecho de que las operaciones entre funciones se han definido a partir de las operaciones en el álgebra de Boole $\langle \{0, 1\}, +, \cdot, \bar{} \rangle$. Finalmente, es fácil comprobar que las funciones que hacen corresponder todas las n-tuplas, salvo una, a 0, cumplen la definición de elemento atómico (capítulo 2, definición 4.3.1): su producto por cualquier otra función da, o bien f_0, o bien el mismo elemento atómico.

4.3. Formas booleanas

4.3.1. *El lenguaje de las formas booleanas*

Definición 4.3.1.1. Dada un álgebra de Boole binaria, $\langle \{0, 1\}, +, \cdot, \bar{} \rangle$, llamaremos *variables booleanas* a unos símbolos, $x_1, y_1, z_1, x_2, y_2, z_2, \ldots$, que representan a los elementos del conjunto $\{0, 1\}$. Es decir, una variable booleana puede tomar uno de dos *valores*: 0 ó 1. Cuando 0 y 1 representan los valores lógicos de un circuito, las variables booleanas corresponden a las variables lógicas de las que hablábamos en los apartados 1, 2 y 3.

Partiendo del alfabeto formado por el conjunto de variables booleanas y los símbolos «+», «·» y « $\bar{}$ » podemos definir un lenguaje siguiendo exactamente los mismos pasos de la lógica de proposiciones (capítulo 2, apartado 2), que ahora nos limitamos a dejar indicados:

a) Definición de expresión o cadena como secuencia finita de símbolos del alfabeto.
b) Definición de secuencia de formación.
c) Definición de sentencia, a la que ahora llamaremos *forma booleana.*

Las seis «fórmulas» de los ejemplos del apartado 3 son ejemplos de formas booleanas. Como los símbolos «+», «·» y « $\bar{}$ » se corresponden, respectivamente, con las puertas «OR», «AND» y «NOT», a cada circuito lógico (o, mejor, a cada salida del mismo) le corresponde una forma booleana, y viceversa.

En realidad, el lenguaje de las formas booleanas es el mismo de la lógica de proposiciones, cambiando las conectivas «$\vee$», «$\wedge$» y «$\neg$» por los símbolos «+», «·» y « $\bar{}$ », respectivamente. El hecho de que las variables y las formas booleanas puedan tomar los valores «0» y «1» de un álgebra de Boole binaria se corresponde con la interpretación binaria de variables proposicionales y de sentencias. Por todo ello, lo que sigue viene a ser una repetición de lo ya visto en el capítulo 2, y lo expondremos de forma resumida.

4.3.2. *Valuación de formas booleanas*

Definición 4.3.2.1. Dado un conjunto de variables booleanas, A, llamaremos *función de asignación* (o, simplemente, *asignación*) a una aplicación del conjunto A en el conjunto $\{0, 1\}$:

$$v: A \to \{0, 1\}$$

El concepto es exactamente el mismo de «interpretación binaria» de la lógica de proposiciones (capítulo 2, apartado 3). Aquí lo hemos restringido de entrada al caso binario porque lo aplicaremos exclusivamente a circuitos con dos estados, pero podríamos haberlo planteado de un modo más general, de modo que el modelo matemático pudiera servir para circuitos con tres o más estados.

Definición 4.3.2.2. Dados un conjunto de formas booleanas construidas a partir de un conjunto de variables A y una asignación v_i, llamaremos *función de valuación*, V_i (o, simplemente, *valuación*), a la extensión del dominio de la asignación al conjunto de formas booleanas. Este otro concepto es exactamente el mismo de «interpretación de sentencias» (capítulo 2, teorema 3.1.4), y, lo mismo que allí, se demuestra que esta extensión es única. Las tablas de verdad de los ejemplos del apartado 3 representan todas las funciones de valuación posibles de las correspondientes formas booleanas.

4.3.3. *Equivalencia de formas booleanas*

Definición 4.3.3.1. Llamemos B_n al conjunto (infinito) de formas booleanas construidas con n variables booleanas diferentes, y sean $A, B \in B_n$. Diremos que A y B son equivalentes ($A \equiv B$) si y sólo si sus valuaciones son iguales para todas las asignaciones:

$$(\forall v_i)(V_i(A) = V_i(B))$$

El concepto es exactamente el mismo de la «equivalencia entre sentencias» (capítulo 2, apartado 3.5). Llamaremos ahora $C_n = B_n/\equiv$ al conjunto de clases de equivalencia que resultan de la partición de B_n por la relación de equivalencia. El número de clases de equivalencia es $\text{card}(C_n) = 2^{2^n}$, pero dentro de cada una hay infinitas formas booleanas diferentes (en efecto, x es equivalente a $x + x$, y a $x \cdot x$, y a $x + x + x$, ...).

Definición 4.3.3.2. Las valuaciones de una clase de equivalencia, $V_k(c_i)(c_i \in C_n)$, son las valuaciones de una cualquiera de las formas booleanas que pertenecen a esa clase:

$$V_k(c_i) = V_k(A)(A \in c_i)$$

La forma booleana del ejemplo 3.1 es equivalente a la del ejemplo 3.2, la del 3.3 es equivalente a la del 3.4, y las tres formas booleanas de los ejemplos 3.5, 3.6 y 3.7 son equivalentes entre sí.

4.3.4. *Algebra de Boole de las clases de equivalencia entre formas booleanas*

Definamos primero las operaciones «suma», «producto» y «complementación» en el conjunto C_n. Utilizaremos los mismos símbolos del álgebra de Boole binaria.

Definición 4.3.4.1. Si $c_i, c_j \in C_n$ son dos clases de equivalencia de formas booleanas con n variables, se definen las clases complementaria, suma y producto del siguiente modo:

a) $\overline{c_i}$ es aquella clase tal que:

$$(\forall v_k)[V_k(\overline{c_i}) = \overline{V_k(c_i)}]$$

b) $c_i + c_j$ es aquella clase tal que:

$$(\forall v_k)[V_k(c_i + c_j) = V_k(c_i) + V_k(c_j)]$$

c) $c_i \cdot c_j$ es aquella clase tal que:

$$(\forall v_k)[V_k(c_i \cdot c_j) = V_k(c_i) \cdot V_k(c_j)]$$

Teorema 4.3.4.2. $\langle C_n, +, \cdot, \overline{\ } \rangle$ es un álgebra de Boole cuyos elementos neutros son c_0 (clase de equivalencia cuyas valuaciones son 0 para todas las asignaciones) y c_N (clase de equivalencia cuyas valuaciones son 1 para todas las asignaciones) y cuyos elementos atómicos son las 2^n clases de equivalencia tales que sus valuaciones son 0 para todas las asignaciones salvo una.

Este teorema es, en realidad, el mismo que habíamos visto para las clases de equivalencia entre sentencias (capítulo 2, apartado 4.2), y su demostración es la misma que allí apuntábamos (como también para el teorema 4.2.4 de este capítulo): basta con ver que se satisfacen los axiomas del álgebra de Boole. En cuanto a los elementos atómicos, basta con ver que se cumplen las condiciones de su definición (definición 4.3.1 del capítulo 2): el producto de cualquier c_i por un elemento atómico da c_0 o c_i.

4.3.5. *Formas canónicas*

Consideremos el álgebra de Boole $\langle C_n, +, \cdot, \overline{\ } \rangle$ de las formas booleanas generadas por las variables $x_1, x_2, \ldots, x_n$. Denominaremos l_i a una «metavariable», que puede valer x_i o $\overline{x}_i$ (es lo que en el capítulo 2 llamábamos un «literal»).

Definición 4.3.5.1. Llamaremos *producto canónico* a toda forma booleana compuesta por el producto de todas las variables complementadas o no:

$$P_i = l_1 \cdot l_2 \cdot \ldots \cdot l_n = \prod_{j=1}^{n} l_j$$

(donde «Π» representa el producto booleano).

Un cálculo combinatorio elemental nos da que el número de productos canónicos diferentes con n variables booleanas es 2^n.

Definición 4.3.5.2. Llamaremos *forma canónica* a toda forma booleana compuesta por una suma de productos canónicos diferentes entre sí.

Teorema 4.3.5.3. Toda clase de equivalencia en C_n puede representarse mediante su forma canónica, que es única para esa clase de equivalencia.

Demostración:

a) Como en un producto canónico intervienen todas las variables, sus valuaciones serán siempre «0», excepto para una determinada asignación: aquella que asigne el valor «1» a las variables sin complementar y el valor «0» a las variables complementadas. Por tanto, cada uno de los 2^n productos canónicos sirve para representar a uno de los 2^n elementos atómicos de $\langle C_n, +, \cdot, \bar{} \rangle$ (teorema 4.3.4.2).

b) Según el teorema 4.3.2 del capítulo 2, todo elemento de un álgebra de Boole distinto de 0 puede expresarse de manera única como suma de elementos atómicos. Luego cualquier clase de equivalencia distinta de c_0 puede expresarse de manera única como suma de elementos atómicos, y, por tanto, podrá representarse de manera única como suma de productos canónicos, es decir, en la primera forma canónica.

Existe una notación abreviada para escribir formas canónicas, que se basa en asociar un número decimal a cada producto canónico. Este número es el que resulta de considerar como un número binario la combinación de «ceros» y «unos» de las variables para la cual la valuación del producto es «1». Por ejemplo, $\bar{x} \cdot y \cdot z$ es «1» para $x = 0$, $y = 1$, $z = 1$, y «011» en binario es «3» en decimal. Del mismo modo, $x \cdot \bar{y} \cdot x$ es «1» para $x = 1$, $y = 0$, $z = 1$, y «101» en binario es «5» en decimal. La notación abreviada de la forma canónica $\bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot z$ es: $\Sigma(3, 5)$ (suma de los productos canónicos «3» y «5»).

Ejemplo 4.3.5.4. Si consideramos las formas booleanas que pueden formarse con dos variables, x e y, el número de clases de equivalencia es 16 (lo mismo que entre las sentencias construidas con dos variables proposicionales y con interpretaciones binarias, véase el apartado 3.5 del capítulo 2). Los productos canónicos son cuatro: $x \cdot y$, $x \cdot \bar{y}$, $\bar{x} \cdot y$, $\bar{x} \cdot \bar{y}$ (en notación abreviada, «3», «2», «1» y «0», respectivamente), que corresponden a los cuatro elementos atómicos c_1 (clase de equivalencia de las formas cuyas valuaciones son «0», excepto para $x = 1$, $y = 1$), c_2 (idem, excepto para $x = 1$, $y = 0$), c_4 (idem, excepto para $x = 0$, $y = 1$) y c_8 (idem, excepto para $x = 0$, $y = 0$). Cualquier otra clase de equivalencia puede representarse como suma de dos o más productos canónicos. Por ejemplo, la c_9 tiene valuación «1» para dos asignaciones: para $x = 0$, $y = 0$, y para $x = 1$, $y = 1$; su forma canónica será la suma de los dos productos canónicos correspondientes: $\bar{x} \cdot \bar{y} + x \cdot y$ (o, en notación abreviada, $\Sigma(0, 3)$).

Ejemplo 4.3.5.5. Con tres variables booleanas tendremos $2^3 = 8$ elementos atómicos (representados por los productos canónicos: $x \cdot y \cdot z$, $x \cdot y \cdot \bar{z}$, ..., $\bar{x} \cdot \bar{y} \cdot \bar{z}$) y $2^8 = 256$ clases de equivalencia. Las dos formas booleanas de los ejemplos 3.1 y 3.2 pertenecen a la misma clase. Para ver cuál es su forma canónica basta con mirar en la tabla de verdad, donde están indicadas todas las valuaciones posibles, qué productos

canónicos son los responsables de las cinco valuaciones «1» que aparecen en dicha tabla, y sumarlos. El resultado es:

$$\bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot \bar{z} + x \cdot \bar{y} \cdot z + x \cdot y \cdot \bar{z} + x \cdot y \cdot z$$

(en notación abreviada: $\Sigma(3, 4, 5, 6, 7)$). Del mismo modo, el lector puede comprobar que la forma canónica que corresponde a los ejemplos 3.5, 3.6 y 3.7 es la forma booleana del último de ellos (en notación abreviada: $\Sigma(3, 5, 6, 7)$).

Ejemplo 4.3.5.6. Con cuatro variables booleanas habrá $2^4 = 16$ elementos atómicos (productos canónicos $x_1 \cdot x_2 \cdot x_3 \cdot x_4$, $x_1 \cdot x_2 \cdot x_3 \cdot \bar{x}_4$, ..., $\bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 \cdot \bar{x}_4$) y $2^{16} = 65.536$ clases de equivalencia. El procedimiento para escribir la primera forma canónica de una clase cualquiera es el mismo de antes: sumar los productos canónicos que corresponden a las valuaciones «1». A la vista de la tabla de verdad de los ejemplos 3.3 y 3.4, obtenemos la correspondiente forma canónica, que, escrita en notación abreviada, es: $\Sigma(2, 3, 6, 7, 8, 9, 10, 11, 13, 14, 15)$. Obsérvese que el paso de la tabla de verdad a la notación abreviada es inmediato: basta con numerar sus filas de 0 a $2^n - 1$, e incluir todos los números de las filas que correspondan a una valuación de «1».

4.4. Relación entre los dos modelos matemáticos

En los últimos ejemplos hemos utilizado las tablas de verdad como representaciones de las valuaciones de una forma booleana (y de todas las que están en su misma clase de equivalencia). Por otra parte, en el apartado 4.2 hacíamos uso de las mismas tablas de verdad como ejemplos de funciones de conmutación. Esta dualidad tiene un fundamento matemático. En efecto, el álgebra de Boole de las funciones de conmutación de orden n, $\langle F_n, +, \cdot, \bar{} \rangle$, tiene 2^{2^n} elementos, ya que el número de n-tuplas diferentes en $\{0, 1\}^n$ es $k = 2^n$, y hace 2^k combinaciones distintas para aplicar cada una de esas k n-tuplas en $\{0, 1\}$. Por otra parte, el número de clases de equivalencia en el álgebra de Boole de las formas booleanas con n variables, $\langle C_n, +, \cdot, \bar{} \rangle$ es también 2^{2^n}: $k = 2^n$ productos canónicos con los que pueden escribirse 2^k primeras formas canónicas diferentes. Por tanto, y de acuerdo con el teorema 4.3.4 enunciado en el capítulo 2, ambas álgebras de Boole son isomorfas, es decir, existe una correspondencia biunívoca entre cada función de conmutación de orden n y cada clase de equivalencia de formas booleanas de n variables, y esta correspondencia es tal (definición 4.3.3 del capítulo 2) que si f_i y f_j están en correspondencia con c_i y c_j, respectivamente, entonces f_i está en correspondencia con c_i y $(f_i + f_j)$ y $(f_i \cdot f_j)$ están en correspondencia con $(c_i + c_j)$ y $(c_i \cdot c_j)$, respectivamente.

La importancia de esta base teórica para la aplicación práctica al diseño de circuitos lógicos es la que insinuábamos en el apartado 4.1: Normalmente, tendremos la especificación de cada salida del circuito como una función de conmutación (o, equivalentemente, como una primera forma canónica). Sabemos que a esa función de conmutación le corresponde toda una clase de equivalencia de formas booleanas con n variables, dentro de la cual hay infinitas formas booleanas diferentes; todas ellas

corresponden a la misma función de conmutación, pero a cada una le corresponde un circuito lógico diferente. Todos estos circuitos lógicos realizarán la misma función, y lo que nos interesa es elegir el más sencillo. Entramos así en el problema de la minimización.

5. MINIMIZACIÓN

5.1. Principios

Para encontrar formas booleanas equivalentes a una dada y más sencillas que ella, podemos utilizar dos equivalencias:

a) Reducción de términos adyacentes: Si A es una forma booleana cualquiera y x una variable booleana,

$$A \cdot x + A \cdot \overline{x} \equiv A.$$

Las formas booleanas que sólo difieren en una variable, que aparece complementada en una y sin complementar en la otra, se llaman «términos adyacentes».

b) Idempotencia: Si A es una forma booleana cualquiera,

$$A + A \equiv A.$$

Por ejemplo, consideremos la forma booleana de tres variables:

$$\overline{x} \cdot y \cdot z + x \cdot \overline{y} \cdot z + x \cdot y \cdot \overline{z} + x \cdot y \cdot z$$

Observamos que el último producto es adyacente a cada uno de los otros tres (pero éstos no lo son entre sí). Podemos reducirlo con uno cualquiera de ellos; si lo hacemos con el primero, resulta:

$$y \cdot z + x \cdot \overline{y} \cdot z + x \cdot y \cdot \overline{z}$$

con el segundo:

$$\overline{x} \cdot y \cdot z + x \cdot z + x \cdot y \cdot \overline{z}$$

y con el tercero:

$$\overline{x} \cdot y \cdot z + x \cdot \overline{y} \cdot z + x \cdot y$$

Estas tres formas booleanas son equivalentes entre sí, y equivalentes a la primera. Pero se puede conseguir otra con menos operaciones si previamente «complicamos» la forma de partida teniendo en cuenta la propiedad de idempotencia: se puede sumar

dos veces el producto $x \cdot y \cdot z$, y cada uno de estos tres productos se podrá reducir con uno de los tres primeros. El resultado será:

$$x \cdot y + x \cdot z + y \cdot z$$

que es la «forma mínima en suma de productos». Su realización como circuito requiere una puerta OR de tres entradas y tres puertas AND de dos entradas. Una pequeña reducción se consigue teniendo en cuenta la propiedad distributiva, que nos permite escribir esta otra forma booleana equivalente:

$$x \cdot (y + z) + y \cdot z$$

5.2. Método de Karnaugh

Es evidente que en casos más complicados que el del ejemplo que acabamos de considerar no es fácil ver qué elementos conviene «desdoblar» para poder aplicar luego reducciones. Se han propuesto diversos métodos para minimizar de modo sistemático. Algunos son numéricos, y conducen a un algoritmo que puede programarse. Aquí veremos un método gráfico que es el más sencillo y también el más conocido (aunque prácticamente deja de tener utilidad para formas booleanas con más de seis variables): el método de las tablas de Karnaugh.

Una tabla de Karnaugh no es otra cosa que una presentación de la tabla de verdad con dos entradas y con las asignaciones colocadas de tal modo que las que corresponden a productos canónicos adyacentes están físicamente contiguas. En la figura 3.11 pueden verse las disposiciones de las tablas de Karnaugh para los casos de tres, cuatro y cinco variables booleanas. Cada casilla corresponde a una línea de la tabla de verdad, y en cada caso se pondrá en ella un «0» o un «1». En la figura 3.11 hemos numerado las casillas con los números de los productos canónicos que corresponden a un «1» en esa casilla, lo cual es muy útil cuando se representa una forma canónica expresada en notación abreviada.

Veamos, mediante algunos ejemplos, cómo se procede para simplificar utilizando las tablas.

Ejemplo 5.2.1. La tabla de Karnaugh para el ejemplo considerado más arriba:

$$x \cdot \overline{y} \cdot \overline{z} + \overline{x} \cdot y \cdot \overline{z} + \overline{x} \cdot \overline{y} \cdot z + \overline{x} \cdot \overline{y} \cdot \overline{z} = \Sigma(3, 5, 6, 7)$$

es la que puede verse en la figura 3.12. Como se indica en ella, se agrupa uno de los «1» (el que corresponde a $\overline{x} \cdot \overline{y} \cdot \overline{z}$) con los tres que le son adyacentes. Cuando se agrupan dos «1» desaparece una variable: la que tiene asignación «0» en un caso y «1» en otro. El resultado es: $x \cdot y + x \cdot z + y \cdot z$.

Ejemplo 5.2.2. Si consideramos la forma canónica $\Sigma(3, 4, 5, 6, 7)$ tenemos la tabla de la figura 3.13. Vemos en ella que hay dos productos, $x \cdot y$, $x \cdot \overline{y}$, resultado de reducir dos parejas de productos adyacentes, y que, a su vez, son adyacentes y se

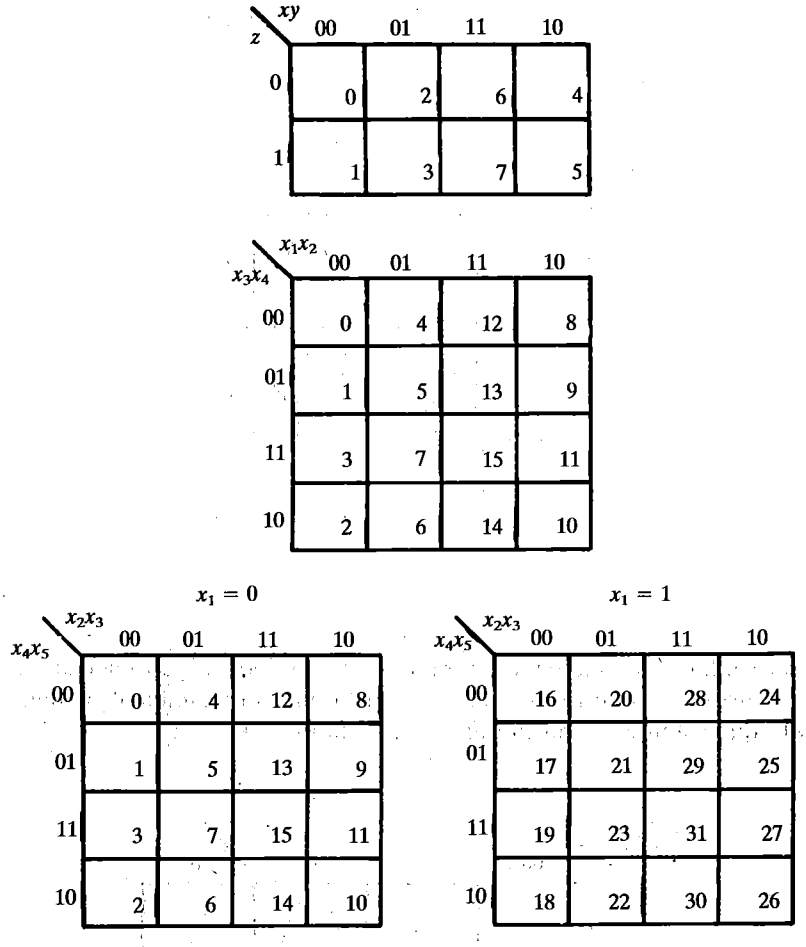


FIGURA 3.11.

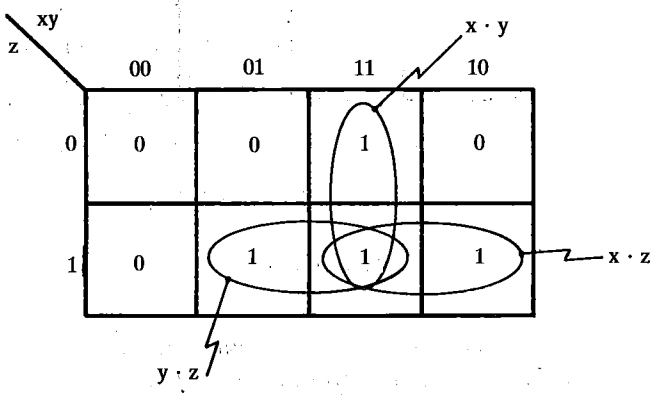


FIGURA 3.12.

reducen a x. A efectos prácticos, cuando vemos cuatro «1» formando un cuadrado, eliminamos dos variables: las que tienen valor «0» en unos casos y «1» en otro. El resultado final para este ejemplo es: $x + y \cdot z$.

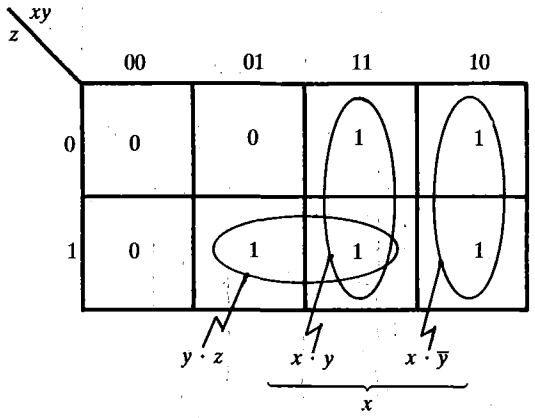


FIGURA 3.13.

Ejemplo 5.2.3. La tabla de la figura 3.14 corresponde a la forma canónica $\Sigma(1, 3, 5, 6, 7)$. Vemos que ocurre algo parecido a lo anterior: también pueden reducirse cuatro «1» que estén en la misma línea. Resultado: $z + x \cdot y$.

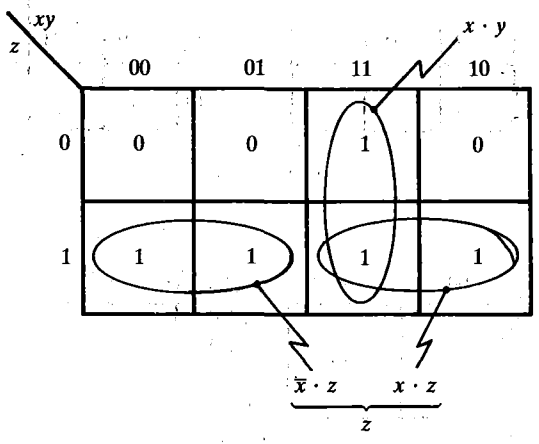


FIGURA 3.14.

Ejemplo 5.2.4. En la tabla de la figura 3.15, correspondiente a la forma canónica $\Sigma(0, 1, 4, 5, 7)$ tenemos otro hecho importante: las casillas de los bordes son también «adyacentes» a las del borde contrario. Resultado de este ejemplo: $\bar{y} + x \cdot z$.

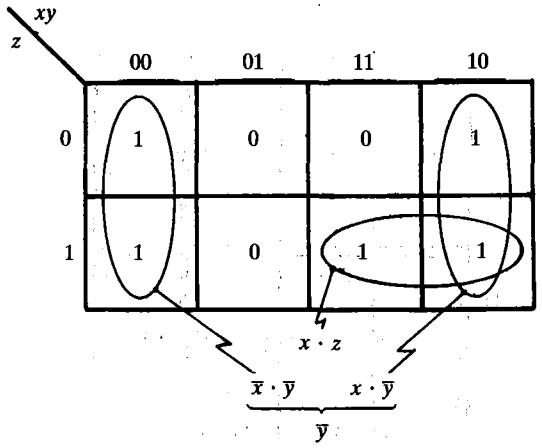
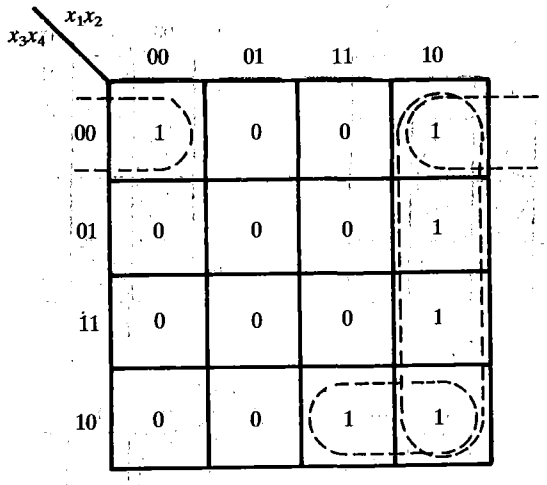


FIGURA 3.15.

Ejemplo 5.2.5. En la figura 3.16 podemos ver un ejemplo de minimización de una forma booleana de cuatro variables: $\Sigma(0, 8, 9, 10, 11, 14)$.



$$f = \bar{x}_2 \cdot \bar{x}_3 \cdot \bar{x}_4 + x_1 \cdot \bar{x}_2 + x_1 \cdot x_3 \cdot \bar{x}_4$$

FIGURA 3.16.

Ejemplo 5.2.6. Si recordamos los ejemplos 3.3 y 3.4 y construimos su correspondiente tabla de Karnaugh (figura 3.17) vemos que la forma mínima (en producto de sumas) es: $x_1 \cdot \bar{x}_2 + x_1 \cdot x_4 + x_3$.

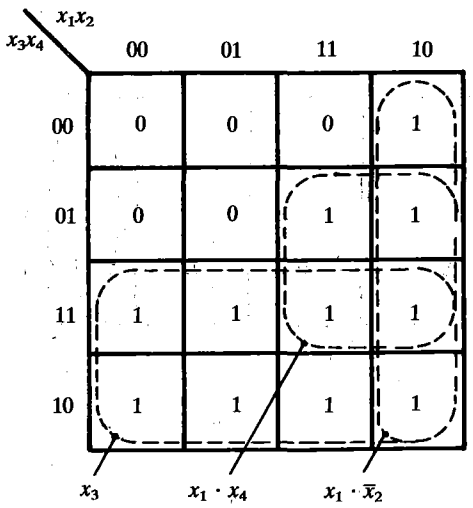


FIGURA 3.17.

Ejemplo 5.2.7. En la figura 3.18 podemos ver otras posibles agrupaciones en tablas de Karnaugh de cuatro variables.

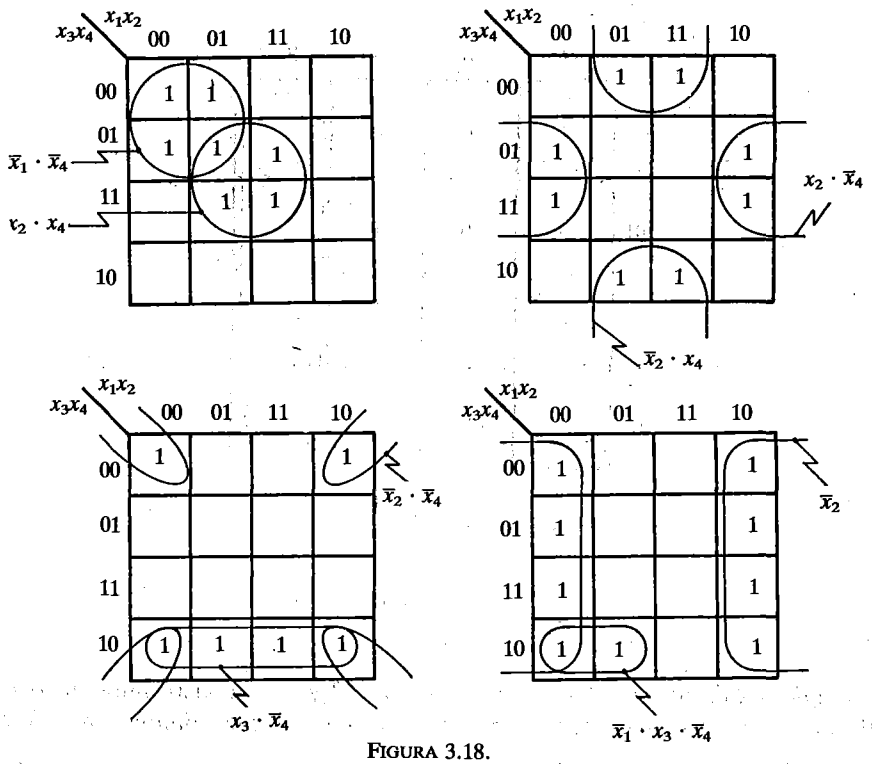


FIGURA 3.18.

Ejemplo 5.2.8. La figura 3.19 es un ejemplo de cómo se utiliza la tabla de Karnaugh de cinco variables. Basta considerar que las casillas (o los grupos) que ocupan la misma posición en las dos tablas son adyacentes.

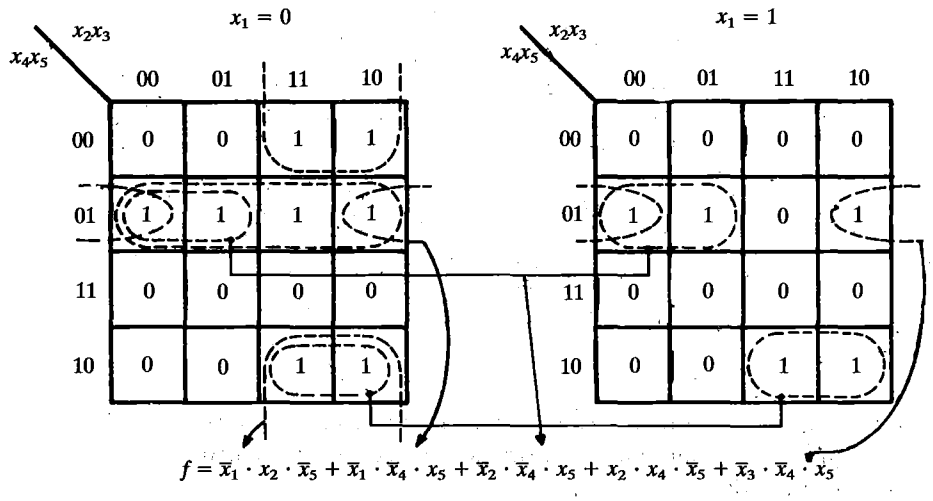


FIGURA 3.19.

En resumen, el método de Karnaugh consiste en *recubrir* todos los «1» que aparecen en la tabla con el menor número posible de grupos (menor número total de sumandos) y de modo que cada grupo sea lo más grande posible (menor número de variables en cada uno de los productos que forman los sumandos).

6. EJEMPLOS DE APLICACIÓN

6.1. Máquina de escrutinio

Supongamos que hay un comité formado por cuatro miembros, de los que uno es presidente, y que las decisiones se toman por mayoría simple, decidiendo el voto del presidente cuando existe empate. Se trata de diseñar una máquina con cuatro entradas (un pulsador para cada miembro) cuya salida dé el resultado de la votación. Llamando A, B, C y D a las variables lógicas que representan el estado del pulsador de cada miembro (donde A es el que corresponde al presidente) y S a la que representa la salida del circuito, la especificación viene dada por la siguiente tabla de verdad:

A	B	C	D	S
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

En la figura 3.20 (a) tenemos esta misma información representada en una tabla de Karnaugh. Vemos que pueden formarse tres grupos de cuatro «1» y un grupo de dos, con el resultado:

$$S = A \cdot B + A \cdot C + A \cdot D + B \cdot C \cdot D = A \cdot (B + C + D) + B \cdot C \cdot D$$

El circuito correspondiente es el dibujado en la figura 3.20 (b), si bien para este caso resultaría más económico basarse en un simple circuito con interruptores como

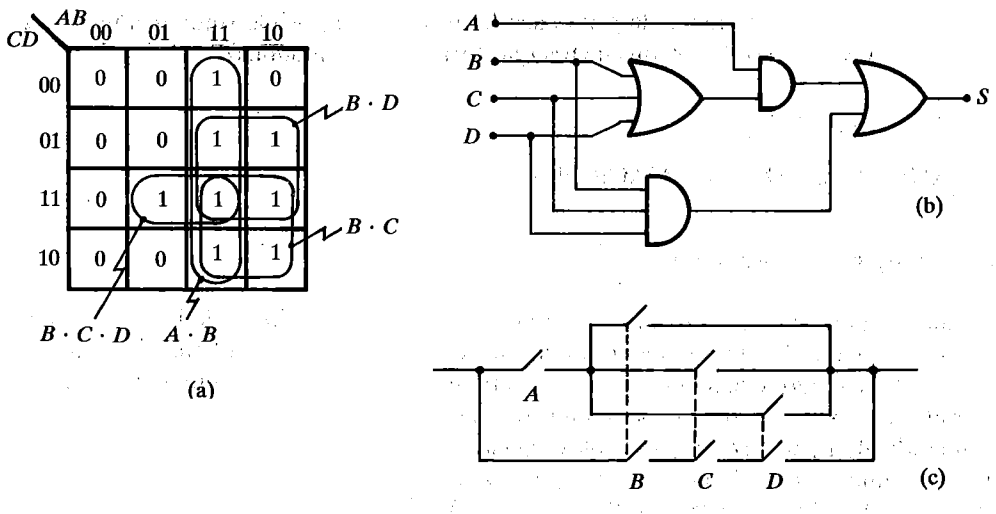


FIGURA 3.20.

muestra la figura 3.20 (c). (Obsérvese que la conexión en serie de interruptores realiza una función de conmutación correspondiente a un producto lógico, y la combinación en paralelo realiza la de una suma).

6.2. Alarma para incendios

Tenemos un detector de llamas, un detector de humos y dos detectores de temperatura distribuidos por una sala. Las salidas de esos detectores las simbolizamos, respectivamente, por las variables lógicas A, B, C y D, con valor «0» en caso de normalidad y «1» en caso de detección positiva. Suponemos que el detector de llamas (A) no da «falsos positivos» pero sí «falsos negativos» (es decir, si $A = 1$ la alarma debe dispararse, pero es posible que la alarma también deba dispararse con $A = 0$). Los otros tres detectores pueden fallar tanto en el caso positivo (salida «1» sin incendio) como en el negativo (salida «0» con incendio); consideramos que para que se confirme la alarma es necesario y suficiente que den detección positiva el de humos (B) y uno al menos de los de temperatura (C o D). De acuerdo con todo esto, podemos especificar el circuito mediante esta tabla:

A	B	C	D	S
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

En la figura 3.21 (a) podemos ver los agrupamientos, que conducen a la forma booleana minimizada:

$$S = A + B \cdot C + B \cdot D = A + B \cdot (C + D)$$

y al circuito de la figura 3.21 (b).

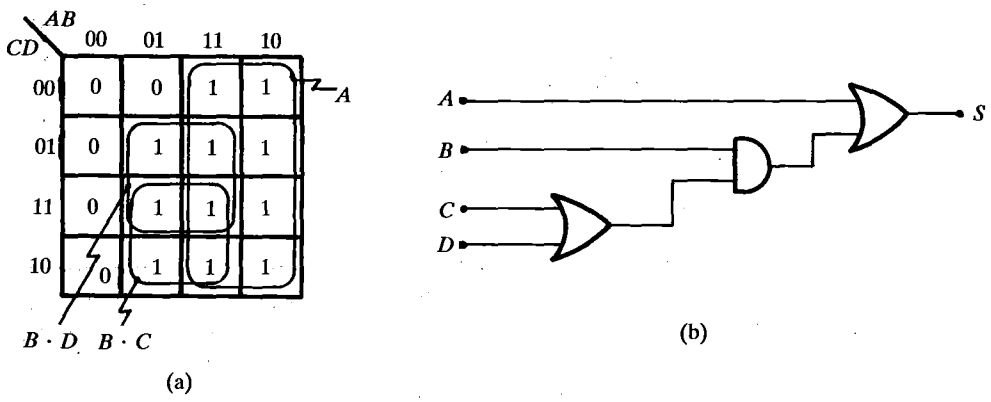


FIGURA 3.21.

6.3. Etapa de sumador binario

Este ejemplo nos va a permitir ilustrar un caso interesante en la minimización de formas booleanas: el de aquellas que corresponden a *funciones de conmutación incompletamente especificadas*. Este caso aparece cuando se da la circunstancia de que hay combinaciones de valores de las variables de entrada que o bien sabemos que nunca se van a producir o bien, cuando se producen, nos es indiferente el valor que pueda tomar la salida.

Una etapa de sumador binario es un circuito que suma dos dígitos binarios, teniendo en cuenta el posible acarreo o arrastre de la suma de los dos dígitos de peso inmediatamente inferior (que pueden haberse sumado en otra etapa). Por tanto, tendrá tres entradas binarias: x, y, r', correspondientes, respectivamente, a los dos dígitos a sumar y al arrastre de la etapa anterior, y dos salidas: s (suma) y r (arrastre producido). Un sumador binario paralelo de n bits constará de n etapas (de las que la primera no necesita la entrada r'), en las que la salida r de cada una se conecta a la entrada r' de la siguiente.

Si analizamos las posibilidades de x, y, r', y para cada una anotamos los valores que deben tomar s y r, llegamos a las siguientes tablas de verdad:

x	y	r'	s	r
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

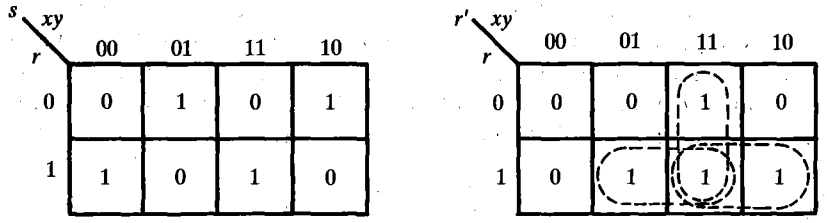


FIGURA 3.22.

De las tablas de Karnaugh de la figura 3.22 obtenemos las formas booleanas:

$$s = \overline{x} \cdot \overline{y} \cdot r' + \overline{x} \cdot y \cdot \overline{r}' + x \cdot y \cdot r' + x \cdot \overline{y} \cdot \overline{r}'$$
$$r = x \cdot y + r' \cdot (x + y)$$

(La forma correspondiente a r es la que ya habíamos considerado como ejemplo en el apartado 5.1. $x \cdot y$ es el «arrastre generado» en esta etapa y $r' \cdot (x + y)$ es el «arrastre propagado» desde la anterior a la siguiente).

Se observará que s no se ha podido simplificar, y se ha expresado por su forma canónica. Sin embargo, podemos reducir el número de puertas del circuito global si tenemos en cuenta que, en realidad, estamos diseñando dos circuitos: uno para s y otro para r; cabe entonces pensar en utilizar r como entrada adicional para s, que será así una función de cuatro entradas (x, y, r', r), como muestra la figura 3.23 (a). Si, siguiendo esa idea, tratamos a s como una función de cuatro variables, x, y, r', r, lo primero que encontramos es que existen cuádruplas para las que s no está definida, por ejemplo, para $x = 0, y = 0, r' = 0, r = 1$; y s no está definida en este caso (como en otros similares) porque esa combinación es imposible, ya que si x, y, r' valen las tres «0», entonces necesariamente $r = 0$. Para estas combinaciones de entrada, *que*

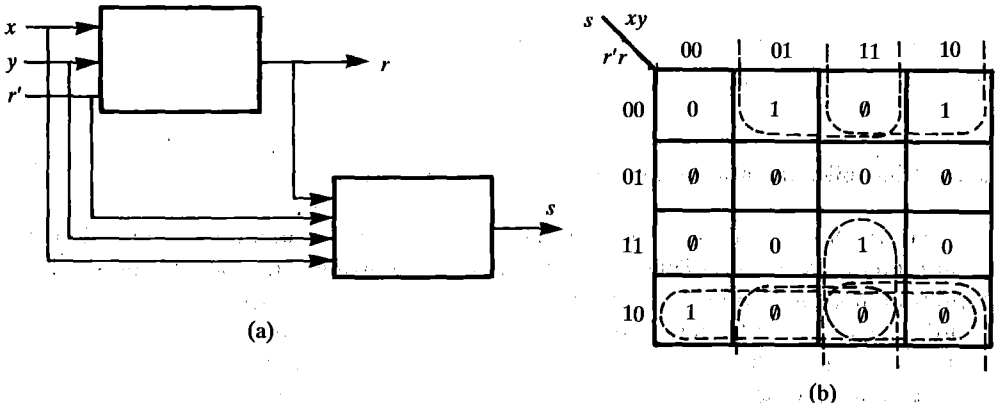


FIGURA 3.23.

nunca se van a presentar a la entrada del circuito de s, podemos tomar para s el valor «0» o «1» indiferentemente, y esto lo representaremos en la tabla de Karnaugh con el símbolo «∅». Agruparemos los «∅» con los «1» o no, según convenga mejor a efectos de formar el menor número de grupos lo más grandes posibles. Con los grupos que pueden verse en la figura 3.23 (b) llegamos a la forma booleana:

$$s = r' \cdot \overline{r} + y \cdot \overline{r} + x \cdot \overline{r} + x \cdot y \cdot r' = \overline{r} \cdot (x + y + r') + x \cdot y \cdot r'$$

El circuito global será el dibujado en la figura 3.24.

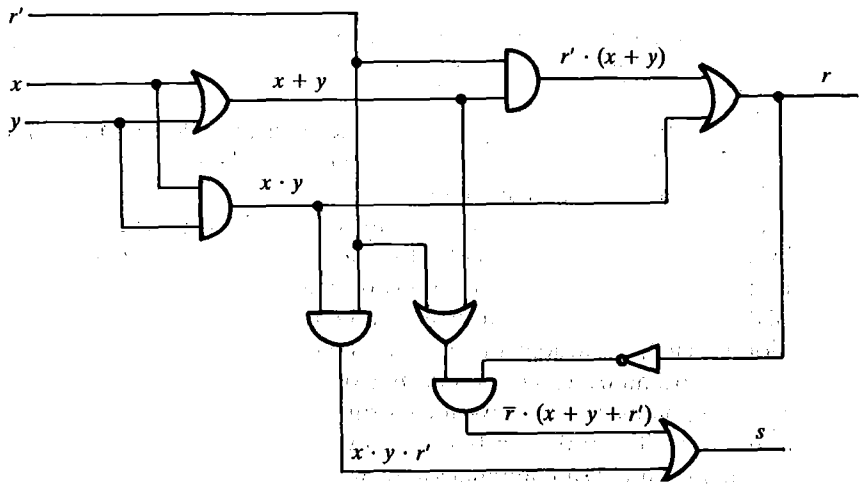


FIGURA 3.24.

6.4. Multiplicador binario de dos bits

Se trata ahora de diseñar un circuito para multiplicar dos números enteros de valor comprendido entre 0 y 3. Tendrá, por tanto, cuatro entradas (dos para los dos bits que representan a cada uno de los multiplicandos) y cuatro salidas (puesto que el máximo resultado es 3 * 3 = 9, que en binario puro es 1001). Llamemos x_0, x_1 a las entradas correspondientes a los bits de peso 0 y 1 de uno de los multiplicandos, y_0, y_1 a los del otro, y z_0 a z_3 a las salidas correspondientes a los bits de peso 0 a 3 del resultado. Analizando todas las posibilidades, llegamos a esta tabla de verdad (a la izquierda de cada fila figura la correspondiente operación en decimal):

$x * y = z$	x_1	x_0	y_1	y_0	z_3	z_2	z_1	z_0
$0 * 0 = 0$	0	0	0	0	0	0	0	0
$0 * 1 = 0$	0	0	0	1	0	0	0	0
$0 * 2 = 0$	0	0	1	0	0	0	0	0
$0 * 3 = 0$	0	0	1	1	0	0	0	0
$1 * 0 = 0$	0	1	0	0	0	0	0	0
$1 * 1 = 1$	0	1	0	1	0	0	0	1
$1 * 2 = 2$	0	1	1	0	0	0	1	0
$1 * 3 = 3$	0	1	1	1	0	0	1	1
$2 * 0 = 0$	1	0	0	0	0	0	0	0
$2 * 1 = 2$	1	0	0	1	0	0	1	0
$2 * 2 = 4$	1	0	1	0	0	1	0	0
$2 * 3 = 6$	1	0	1	1	0	1	1	0
$3 * 0 = 0$	1	1	0	0	0	0	0	0
$3 * 1 = 3$	1	1	0	1	0	0	1	1
$3 * 2 = 6$	1	1	1	0	0	1	1	0
$3 * 3 = 9$	1	1	1	1	1	0	0	1

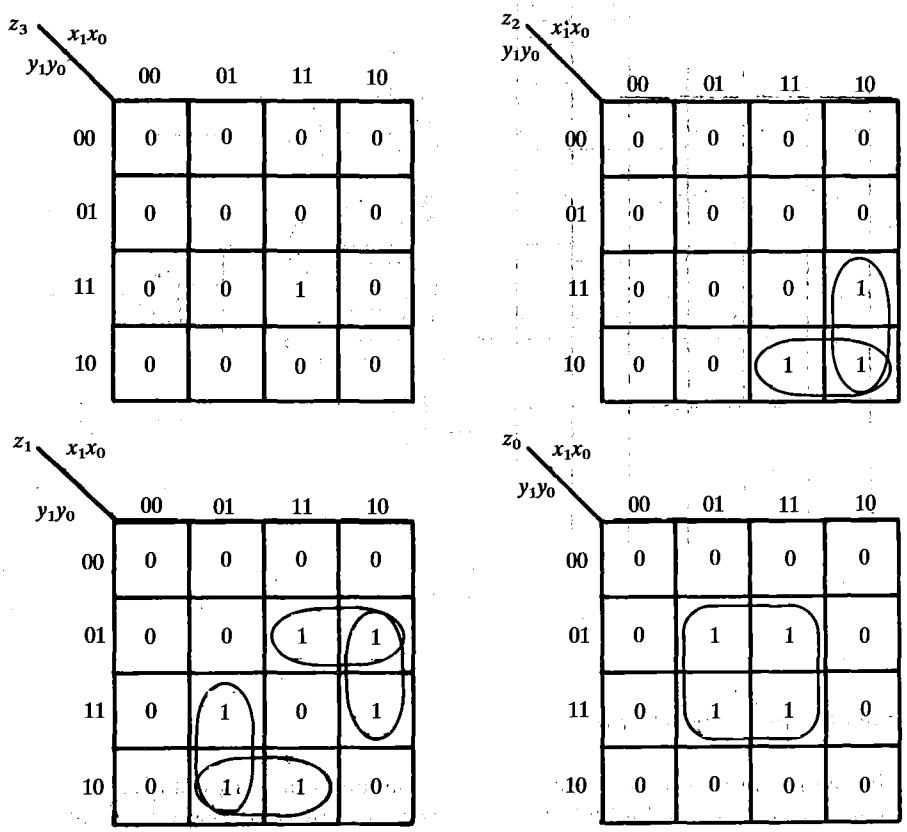


FIGURA 3.25.

De las tablas de Karnaugh (figura 3.25) obtenemos las formas booleanas:

$$z_3 = x_1 \cdot x_0 \cdot y_1 \cdot y_0$$
$$z_2 = x_1 \cdot \overline{x}_0 \cdot y_1 + x_1 \cdot y_1 \cdot \overline{y}_0 = x_1 \cdot y_1 \cdot (\overline{x}_0 + \overline{y}_0)$$
$$z_1 = \overline{x}_1 \cdot x_0 \cdot y_1 + x_0 \cdot y_1 \cdot \overline{y}_0 + x_1 \cdot \overline{x}_0 \cdot y_0 + x_1 \cdot \overline{y}_1 \cdot y_0 =$$
$$= x_0 \cdot y_1 \cdot (\overline{x}_1 + \overline{y}_0) + x_1 \cdot y_0 \cdot (\overline{x}_0 + \overline{y}_1)$$
$$z_0 = x_0 \cdot y_0$$

y, de ellas, el circuito de la figura 3.26. Como ejercicio, sugerimos al lector que trate de encontrar otras formas booleanas más sencillas para z_1 y z_2 considerando como entradas adicionales las otras dos salidas.

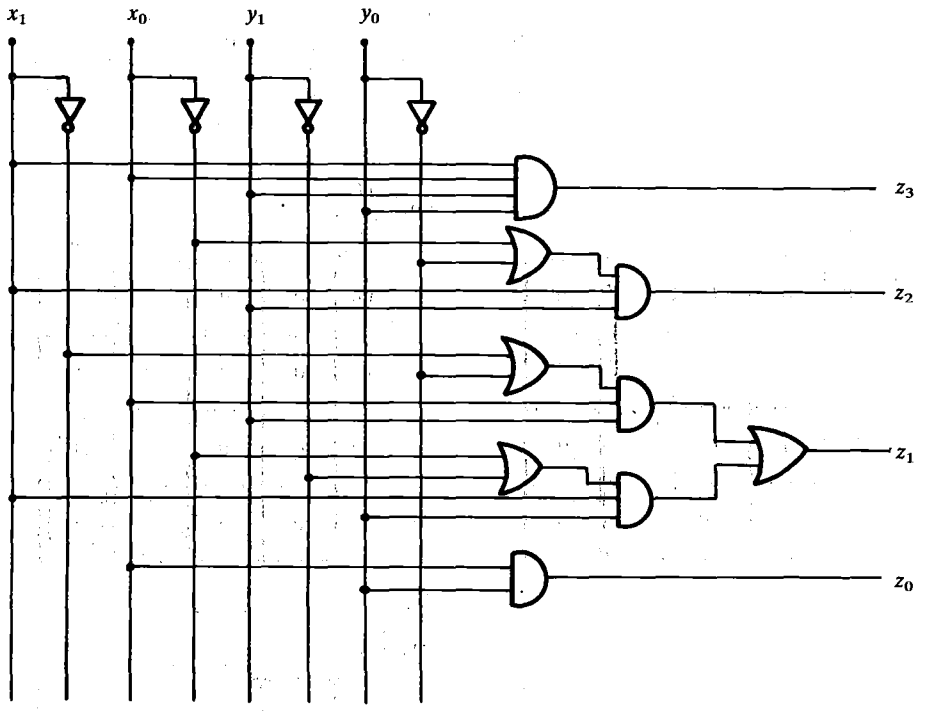


FIGURA 3.26.

6.5. Regulación de una piscina

Se desea diseñar un sistema de regulación de la temperatura y del nivel de agua en una piscina. El sistema recibirá la información de tres sensores colocados dentro de la piscina: un termómetro y dos detectores de nivel, y deberá actuar sobre una

resistencia para calentar el agua y sobre dos válvulas: una que permite desalojar agua, y otra que permite añadir agua fría. Las especificaciones de su funcionamiento vienen dadas por la siguiente tabla:

Información recibida		*Acción a tomar*
Temperatura	*Nivel*	
Normal	Normal	Ninguna
Normal	Bajo	Añadir agua
Normal	Alto	Sacar agua
Baja	Normal	Calentar
Alta	Normal	Añadir agua
Baja	Bajo	Añadir agua y calentar
Baja	Alto	Sacar agua y calentar
Alta	Bajo	Añadir agua
Alta	Alto	Añadir y sacar agua al mismo tiempo

Se supondrá que el termómetro tiene dos terminales de salida, en cada uno de los cuales pueden tenerse dos niveles de tensión (a los que daremos los niveles lógicos 0 y 1). Si la temperatura es demasiado alta, el primer terminal, T_1, estará en el nivel 1 (y el segundo, T_2, en el 0); si es demasiado baja ocurrirá lo contrario, y si está entre los límites prefijados («temperatura normal»), ambos serán 0.

Los detectores del nivel de agua dan también señales 0 y 1: uno de ellos, N_1, está en 1 si el nivel es demasiado alto, y el otro, N_2, está en 1 si es demasiado bajo (de modo que ambos en 0 indicarán «nivel normal»).

En la salida, para que una válvula (V_1: sacar, V_2: añadir) esté cerrada se deberá dar el nivel lógico 0, y para que esté abierta el 1. Análogamente, para que la resistencia (R) caliente se dará un 1 y para que no actúe un 0.

La parte lógica del sistema será un circuito con cuatro entradas (T_1, T_2, N_1, N_2) y tres salidas (V_1, V_2, R). Las tablas de verdad se obtienen inmediatamente de las especificaciones. Se observará que hay 7 combinaciones de las variables de entrada que son imposibles (las que corresponden a $T_1 = T_2 = 1$ y $N_1 = N_2 = 1$), por lo que las funciones V_1, V_2 y R son incompletamente especificadas. Aprovecharemos este hecho, del mismo modo que en el Ejemplo 6.3 para minimizar tales funciones. Si el lector sigue los pasos necesarios llegará al siguiente resultado:

$$V_1 = N_1;\ V_2 = T_1 + N_2;\ R = T_2$$

Y el circuito (a nivel lógico) sólo necesita de una puerta «OR», como indica la figura 3.27.

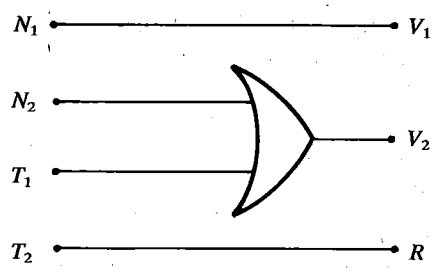


FIGURA 3.27.

7. LA SEGUNDA FORMA CANÓNICA Y LA FORMA MÍNIMA EN PRODUCTO DE SUMAS

7.1. La segunda forma canónica

En el apartado 4.3.5 hemos visto cómo cada clase de equivalencia en el conjunto de formas booleanas que pueden construirse con n variables puede representarse por una forma canónica formada por sumas de productos canónicos. Existen, como vamos a ver, otras posibles formas canónicas para representar de manera única a las clases de equivalencia, por lo que, para diferenciarlas, llamaremos a la que ya conocemos *primera forma canónica* o *forma canónica en suma de productos*. A los productos canónicos se les llama también *minitérminos*.

Definición 7.1.1. Llamaremos *suma canónica* o *maxitérmino* a toda forma booleana compuesta por la suma de todas las variables complementadas o no:

$$S_i = l_1 + l_2 + \ldots + l_n = \sum_{j=1}^{n} l_j$$

(donde «Σ» representa la suma booleana).

Definición 7.1.2. Llamaremos *segunda forma canónica* (o *forma canónica en producto de sumas*) a toda forma booleana compuesta por un producto de maxitérminos diferentes entre sí.

Teorema 7.1.3. Toda clase de equivalencia en C_n puede representarse mediante su segunda forma canónica, que es única para esa clase de equivalencia.

Demostración:

Sea $c_i \in C_n$ una clase de equivalencia cualquiera, y consideremos su complementa-

ria, $\overline{c}_i$. Por el Teorema 4.3.5.3 sabemos que $\overline{c}_i$ tiene una representación única en la primera forma canónica:

$$\overline{c}_i = \Sigma\left(\prod_{j=1}^{n} l_j\right)$$

Aplicando los teoremas de de Morgan, obtenemos:

$$c_i = \overline{\left[\Sigma\left(\prod_{j=1}^{n} l_j\right)\right]} = \Pi\left[\overline{\left(\prod_{j=1}^{n} l_j\right)}\right] = \Pi\left(\sum_{j=1}^{n} \overline{l}_j\right)$$

que será la representación de c_i en la segunda forma canónica. Y como $\overline{c}_i$ es única para c_i, esta representación también será única.

Una notación abreviada para la segunda forma canónica se obtiene asignando a cada maxitérmino un número decimal. Por ejemplo, $x + \overline{y} + \overline{z}$ sólo vale «0» para la combinación 011 (3 en decimal), $\overline{x} + y + \overline{z}$ sólo vale «0» para 101 (5), y la representación de $(x + \overline{y} + \overline{z}) \cdot (\overline{x} + y + \overline{z})$ será: $\Pi(3, 5)$.

Ejemplo 7.1.4. La función de conmutación correspondiente a la disyunción exclusiva («OR» exclusivo) viene dada por esta tabla de verdad (en la que hemos incluido también los valores de $\overline{f}$):

x	y	f	$\overline{f}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Considerando los productos canónicos que corresponden a los «1» de f obtenemos la primera forma canónica:

$$f = \overline{x} \cdot y + x \cdot \overline{y}$$

Y considerando los que corresponden a los «1» de $\overline{f}$ («0» de f) obtenemos la primera forma canónica de $\overline{f}$:

$$\overline{f} = \overline{x} \cdot \overline{y} + x \cdot y$$

De aquí,

$$= \overline{\overline{x} \cdot \overline{y} + x \cdot y} = \overline{(\overline{x} \cdot \overline{y})} \cdot \overline{(x \cdot y)} = (x + y) \cdot (\overline{x} + \overline{y})$$

que será la segunda forma canónica de f (en notación abreviada, $\Pi(0, 3)$).

Ejemplo 7.1.5. Para los ejemplos 3.1 y 3.2, fijándonos en los «0» de la tabla de verdad, resulta:

$$\overline{f} = \overline{x} \cdot \overline{y} \cdot \overline{z} + \overline{x} \cdot \overline{y} \cdot z + \overline{x} \cdot y \cdot \overline{z}$$

y, de aquí, la segunda forma canónica:

$$f = (x + y + z) \cdot (x + y + \overline{z}) \cdot (x + \overline{y} + z) = \Pi(0, 1, 2)$$

Ejemplo 7.1.6. Análogamente, para los ejemplos 3.5, 3.6 y 3.7:

$$\overline{f} = \overline{x} \cdot \overline{y} \cdot \overline{z} + \overline{x} \cdot \overline{y} \cdot z + \overline{x} \cdot y \cdot \overline{z} + x \cdot \overline{y} \cdot \overline{z}$$
$$f = (x + y + z) \cdot (x + y + \overline{z}) \cdot (x + \overline{y} + z) \cdot (\overline{x} + y + z) = \Pi(0, 1, 2, 4)$$

Ejemplo 7.1.7. Para los ejemplos 3.3 y 3.4,

$$\overline{f} = \overline{x}_1 \cdot \overline{x}_2 \cdot \overline{x}_3 \cdot \overline{x}_4 + \overline{x}_1 \cdot \overline{x}_2 \cdot \overline{x}_3 \cdot x_4 + \overline{x}_1 \cdot x_2 \cdot \overline{x}_3 \cdot \overline{x}_4 + \overline{x}_1 \cdot x_2 \cdot \overline{x}_3 \cdot x_4 + x_1 \cdot x_2 \cdot \overline{x}_3 \cdot \overline{x}_4$$
$$f = (x_1 + x_2 + x_3 + x_4) \cdot (x_1 + x_2 + x_3 + \overline{x}_4) \cdot (x_1 + \overline{x}_2 + x_3 + x_4) \cdot (x_1 + \overline{x}_2 + x_3 + \overline{x}_4) \cdot (\overline{x}_1 + \overline{x}_2 + x_3 + x_4) = \Pi(0, 1, 4, 5, 12)$$

7.2. La forma mínima en producto de sumas

Los dos principios que permiten la simplificación de formas booleanas expresadas en producto de sumas son duales de los que veíamos en el apartado 5.1:

a) Reducción de términos adyacentes:

$$(A + x) \cdot (A + \overline{x}) \equiv A$$

(Obsérvese que esta «reducción» se corresponde exactamente con la «regla de resolución» de la lógica (capítulo 2, apartado 5.7).

b) Idempotencia:

$$A \cdot A \equiv A$$

Para el mismo ejemplo que analizábamos en el apartado 5.1, la segunda forma canónica es exactamente la que hemos visto en el Ejemplo 7.1.6. Podemos observar que el primer maxitérmino es adyacente a los otros tres. Si lo descomponemos en el producto de tres iguales y reducimos cada uno de ellos con cada uno de los otros tres, resulta:

$$f = (x + y) \cdot (x + z) \cdot (y + z)$$

que es la forma mínima en producto de sumas. Aplicando la propiedad distributiva, podemos escribir otras formas equivalentes:

$$(x + y) \cdot (z + x \cdot y) \equiv (y + z) \cdot (x + y \cdot z) \equiv (x + z) \cdot (y + x \cdot z)$$

A efectos prácticos, y utilizando el método de las tablas de Karnaugh, lo más cómodo es minimizar $\overline{f}$ (agrupando los «0» de la tabla en lugar de los «1») y luego aplicar los teoremas de de Morgan. El circuito resultante mediante este procedimiento es, generalmente, el mismo obtenido a partir de la forma mínima en suma de productos, o su dual (cambiando las puertas «OR» por «AND» y viceversa). Pero a veces, cuando la función está incompletamente especificada, el circuito puede ser más sencillo (o más complicado). Como ejercicio, el lector puede probar a hacer las minimizaciones correspondientes a los ejemplos 5.2.1 a 5.2.8 y comparar los circuitos que se obtienen de una manera y de la otra. Veamos cuáles serían los circuitos alternativos para algunos de los «ejemplos de aplicación» del apartado 6.

Ejemplo 7.2.1. Si agrupamos los «0» en la tabla de Karnaugh del Ejemplo 6.1 (figura 3.28 (a)) resulta:

$$\overline{S} = \overline{A} \cdot \overline{D} + \overline{A} \cdot \overline{C} + \overline{A} \cdot \overline{B} + \overline{B} \cdot \overline{C} \cdot \overline{D}$$
$$S = (A + D) \cdot (A + C) \cdot (A + B) \cdot (B + C + D)$$

O, aplicando la propiedad distributiva:

$$S = (A + B \cdot C \cdot D) \cdot (B + C + D)$$

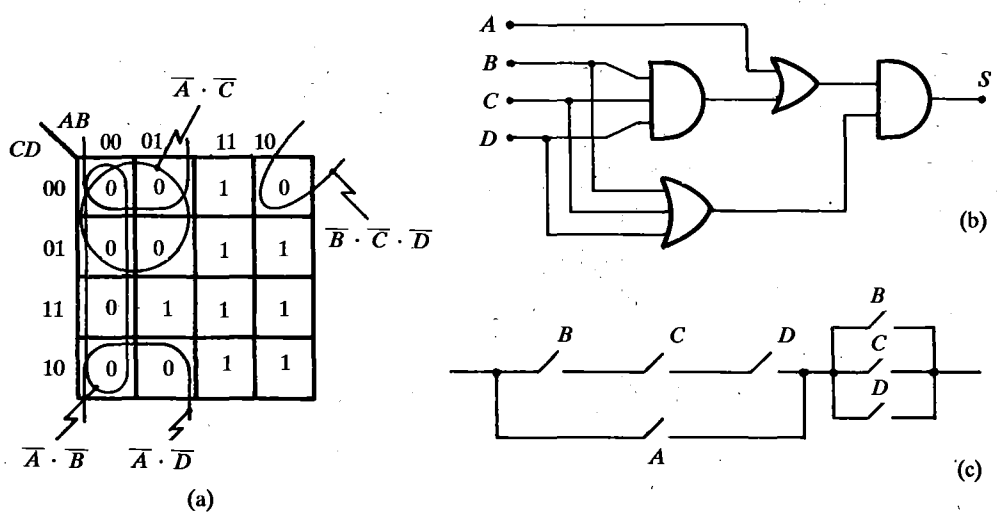


FIGURA 3.28.

El circuito (figura 3.28 (b)) es igual que el de la figura 3.20 (b), cambiando los tipos de puertas.

Ejemplo 7.2.2. Para el caso del Ejemplo 6.2, obtenemos ahora (figura 3.29):

$$\overline{S} = \overline{A} \cdot \overline{B} + \overline{A} \cdot \overline{C} \cdot \overline{D}$$
$$S = (A + B) \cdot (A + C + D) = A + B \cdot (C + D)$$

Forma que es la misma a la que habíamos llegado por el otro procedimiento, por lo que el circuito sería el mismo (figura 3.21 (b)).

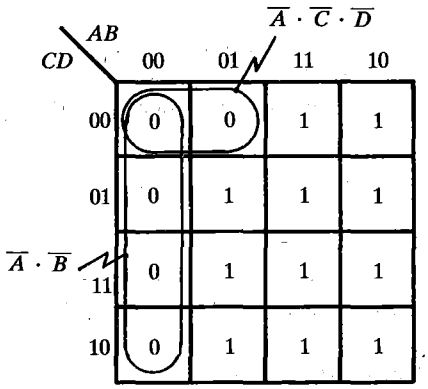


FIGURA 3.29.

Ejemplo 7.2.3. La función de conmutación de la salida r del sumador binario es exactamente el ejemplo que hemos considerado al principio de este apartado (llamando z a r'). Una de las formas minimizadas era:

$$r = (x + y) \cdot (r' + x \cdot y)$$

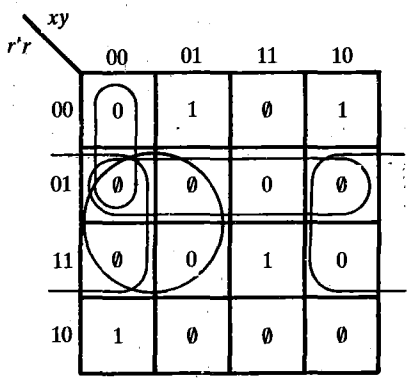


FIGURA 3.30.

En cuanto a s, si tomamos r como entrada adicional, resulta (figura 3.30):

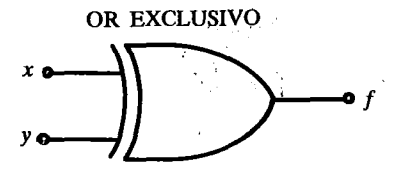
$$\overline{s} = \overline{r}' \cdot r + \overline{x} \cdot r + \overline{y} \cdot r + \overline{x} \cdot \overline{y} \cdot \overline{r}'$$

$$s = (r' + \overline{r}) \cdot (x + \overline{r}) \cdot (y + \overline{r}) \cdot (x + y + r') =$$
$$= (\overline{r} + x \cdot y \cdot r') \cdot (x + y + r')$$

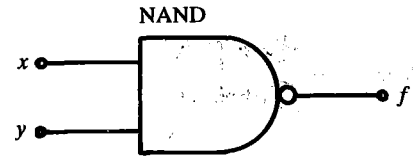
Y el circuito es el dual del de la figura 3.24.

8. OTRAS PUERTAS

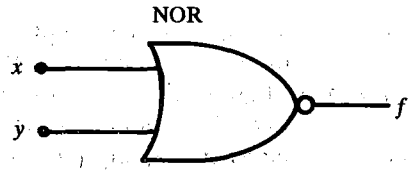
Igual que en lógica de proposiciones, donde teníamos 16 conectivas binarias (entre dos variables proposicionales), también podemos definir 16 operaciones distintas entre dos variables lógicas, a las que corresponderían otras tantas puertas de dos entradas, aunque cuatro de ellas no tienen sentido: las que dan siempre «0» o «1» a la salida (las tautologías y contradicciones de la lógica) y las que dan el valor de una de las entradas, independientemente del que tome la otra. Ahora bien, las más utilizadas, aparte de las ya vistas («NOT», «OR» y «AND») son «ORX» («OR» exclusivo), «NAND» y «NOR», cuyas tablas de verdad son las de la figura 3.31. Para simbolizar las operaciones que realizan estas puertas utilizaremos los símbolos «$\oplus$», «$|$» y «$\downarrow$», respectivamente (que son los mismos que ya habíamos visto en el capítulo 2 para las correspondientes conectivas).



x	y	f
0	0	0
0	1	1
1	0	1
1	1	0



x	y	f
0	0	1
0	1	1
1	0	1
1	1	0



x	y	f
0	0	1
0	1	0
1	0	0
1	1	0

FIGURA 3.31.

Y del mismo modo que, como estudiábamos en el capítulo 2 (apartado 3.6), unas conectivas pueden expresarse en función de otras, también la función que realiza una puerta puede realizarse con un circuito formado por otras. Por ejemplo, la suma lógica con «AND» y «NOT» (figura 3.32 (a)), el producto lógico con «OR» y «NOT» (figura 3.32 (b)), «NAND» y «NOR» con «AND» y «NOT» o «OR» y «NOT» (figura 3.32 (c) y (d)), etc. Obsérvense, en la figura, las representaciones alternativas para «NAND» y «NOR»: la misma función se realiza complementando la salida de una puerta «AND» que complementando las entradas de una «OR», y viceversa.

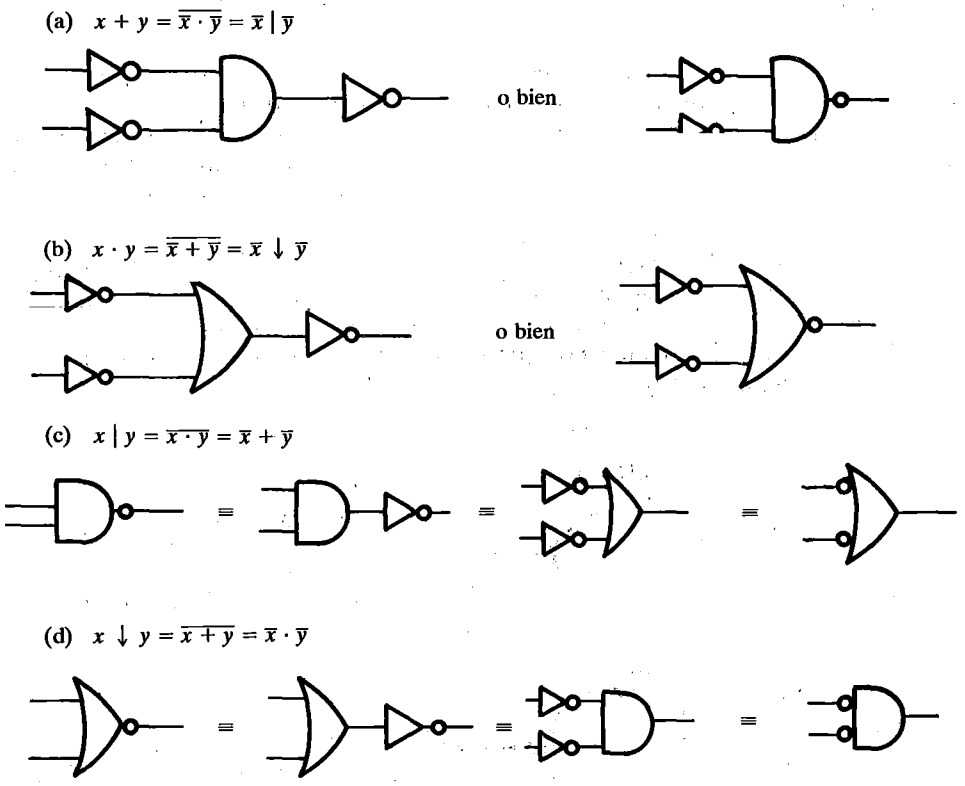
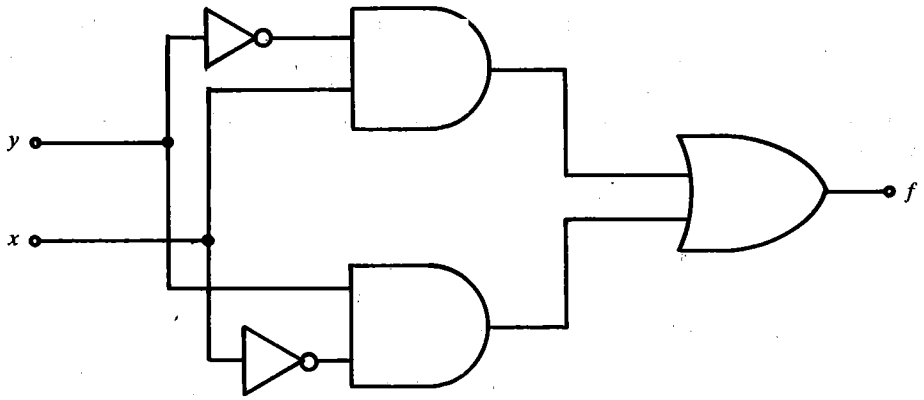


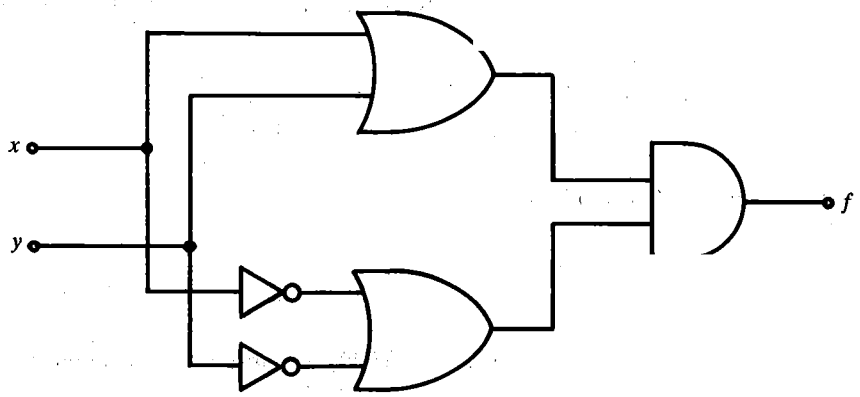
FIGURA 3.32.

De acuerdo con las dos formas canónicas vistas en el Ejemplo 7.1.4 (que no pueden reducirse), la puerta «ORX» podría realizarse con puertas «NOT», «OR» y «AND» indistintamente con el circuito de la figura 3.33 (a) o con el de la figura 3.33 (b). La disponibilidad directa de puertas «ORX» permite realizar de forma más simple algunos circuitos. Por ejemplo, la salida z_1 del Ejemplo 6.4 puede también escribirse:

$$z_1 = (x_0 \cdot y_1) \oplus (x_1 \cdot y_0)$$



(a) $f = x_1 \cdot \overline{x}_2 + \overline{x}_1 \cdot x_2$



(b) $f = (x + y) \cdot (\overline{x} + \overline{y})$

FIGURA 3.33.

Por tanto, si puede utilizarse una puerta «ORX», el circuito se simplifica algo con relación al de la figura 3.26.

9. CIRCUITOS CON PUERTAS «NAND» Y «NOR»

9.1. «NAND» y «NOR» como operaciones completas

Por razones tecnológicas, las puertas «NAND» y «NOR» se utilizan mucho en el diseño de circuitos. Estas puertas pueden tener más de dos entradas:

$$x \mid y \mid z \mid \ldots = \overline{x \cdot y \cdot z \cdot \ldots} = \overline{x} + \overline{y} + \overline{z} + \ldots$$
$$x \downarrow y \downarrow z \downarrow \ldots = \overline{x + y + z + \ldots} = \overline{x} \cdot \overline{y} \cdot \overline{z} \cdot \ldots$$

Tanto NAND como NOR tienen la propiedad de ser *operaciones completas*; esto quiere decir que cualquier función de conmutación puede representarse utilizando sólo la operación NAND o utilizando sólo la operación NOR. Para demostrarlo basta con ver que las tres operaciones básicas, complementación, producto y suma, pueden realizarse con ellas:

a) $\overline{x} = \overline{x \cdot x} = x \mid x$
$\overline{x} = \overline{x + x} = x \downarrow x$

b) $x \cdot y = \overline{\overline{x \cdot y}} = \overline{x} \mid \overline{y} = (x \mid y) \mid (x \mid y)$
$x \cdot y = \overline{\overline{x} + \overline{y}} = \overline{x} \downarrow \overline{y} = (x \downarrow x) \downarrow (y \downarrow y)$

c) $x + y = \overline{\overline{x} \cdot \overline{y}} = \overline{x} \mid \overline{y} = (x \mid x) \mid (y \mid y)$
$x + y = \overline{\overline{x + y}} = \overline{(x \downarrow y)} = (x \downarrow y) \downarrow (x \downarrow y)$

Es preciso prestar atención a los paréntesis, ya que, a diferencia de la suma y el producto, NAND y NOR *no* son asociativas:

$$(x \mid y) \mid z \neq x \mid (y \mid z) \neq x \mid y \mid z$$
$$(x \downarrow y) \downarrow z \neq x \downarrow (y \downarrow z) \neq x \downarrow y \downarrow z$$

(Compruébese por medio de las respectivas tablas de verdad).

9.2. La tercera forma canónica

La tercera forma canónica (sólo con NAND) se obtiene directamente de la primera mediante aplicación del

Teorema 9.2.1:

$$(x_1 \cdot x_2 \cdot \ldots \cdot x_n) + (y_1 \cdot y_2 \cdot \ldots \cdot y_m) + \ldots + (z_1 \cdot z_2 \cdot \ldots \cdot z_h) =$$
$$= (x_1 \mid x_2 \mid \ldots \mid x_n) \mid (y_1 \mid y_2 \mid \ldots \mid y_m) \mid \ldots \mid (z_1 \mid z_2 \mid \ldots \mid z_h)$$

(Para demostrarlo basta con aplicar dos veces las leyes de de Morgan).

9.3. La forma mínima sólo con NAND

Se obtiene aplicando el Teorema 9.2.1 a la forma mínima en suma de productos

Ejemplo (corresponde al Ejemplo 5.2.6):

$$f = x_3 + x_1 \cdot x_4 + x_1 \cdot \overline{x}_2 = (x_3 \mid x_3) \mid (x_1 \mid x_4) \mid (x_1 \mid \overline{x}_2) =$$
$$= (x_3 \mid x_3) \mid (x_1 \mid x_4) \mid (x_1 \mid (x_2 \mid x_2))$$

(Obsérvese que cuando uno de los productos consta de una sola variable, en este caso x_3, este producto puede representarse como $x_3 \cdot x_3$, y de ahí que al aplicar el teorema pongamos $x_3 \mid x_3$ (es decir, $\overline{x}_3$), y no x_3).

9.4. La cuarta forma canónica

Se obtiene de la segunda aplicando el

Teorema 9.4.1:

$$(x_1 + x_2 + \ldots + x_n) \cdot (y_1 + y_2 + \ldots + y_m) \cdot \ldots \cdot (z_1 + z_2 + \ldots + z_h) =$$
$$= (x_1 \downarrow x_2 \downarrow \ldots \downarrow x_n) \downarrow (y_1 \downarrow y_2 \downarrow \ldots \downarrow y_m) \downarrow \ldots \downarrow (z_1 \downarrow z_2 \downarrow \ldots \downarrow z_h)$$

(Se demuestra igualmente con las leyes de de Morgan).

9.5. La forma mínima sólo con NOR

Si aplicamos el Teorema 9.4.1 a la forma mínima en producto de sumas, resultará una forma mínima con operaciones NOR solamente.

Ejemplo:

$$f = x \cdot (\overline{x} + y) \cdot (y + \overline{z}) = (x \downarrow x) \downarrow (\overline{x} \downarrow y) \downarrow (y \downarrow \overline{z}) =$$
$$= (x \downarrow x) \downarrow ((x \downarrow x) \downarrow y) \downarrow (y \downarrow (z \downarrow z))$$

(Aquí podemos hacer una observación similar a la del ejemplo anterior: si una de las sumas sólo tiene un sumando, x en este caso, como $x = x + x$, al aplicar el teorema ponemos $x \downarrow x = \overline{x}$, en lugar de x).

10. RESUMEN

Las funciones de conmutación y las formas booleanas son dos tipos completamente distintos de modelos para los circuitos lógicos combinacionales. Una función de conmutación es un modelo funcional: indica «lo que hace» el circuito, desde el punto de vista de cuáles son sus «respuestas» (valores lógicos de la salida) para los posibles «estímulos» (valores lógicos de las entradas). Por el contrario, una forma booleana es un modelo estructural: indica cómo se conectan los componentes (puertas) del circuito. A cada circuito que sólo tenga una salida le corresponde una forma booleana, y viceversa. Pero a una misma función de conmutación le corresponden infinidad de circuitos (y, por tanto, de formas booleanas).

Frecuentemente, de las especificaciones verbales se llega inmediatamente a las funciones de conmutación (una para cada salida), expresadas como tablas de verdad.

Y el problema que se plantea es el de elegir de entre todos los circuitos (formas booleanas) posibles el más sencillo (forma booleana con menos operaciones)*. A la solución de este problema general ha ido encaminado todo el desarrollo de este capítulo, que podemos resumir así:

a) El conjunto de funciones de conmutación de orden n, junto con unas operaciones de suma, producto y complementación adecuadamente definidas, constituye un álgebra de Boole, $\langle F_n, +, \cdot, \bar{} \rangle$.

b) Dentro del conjunto de formas booleanas de n variables, B_n, hemos definido una relación de equivalencia (que también es una relación de equivalencia entre los circuitos con n entradas y una salida). Esta relación establece una partición de B_n en un conjunto de clases de equivalencia, C_n.

c) El conjunto de clases de equivalencia entre formas booleanas de n variables, junto con las operaciones de suma, producto y complementación definidas en él, constituye un álgebra de Boole, $\langle C_n, +, \cdot, \bar{} \rangle$.

d) Las álgebras de Boole $\langle F_n, +, \cdot, \bar{} \rangle$ y $\langle C_n, +, \cdot, \bar{} \rangle$ son isomorfas, porque tienen el mismo número de elementos ($\text{card}(F_n) = \text{card}(C_n) = 2^{2^n}$). Por tanto, a cada función de conmutación de orden n le corresponde biunívocamente una clase de equivalencia entre formas booleanas de n variables, y, por tanto, un conjunto (infinito) de circuitos equivalentes. Esta doble correspondencia se ilustra en la figura 3.33.

e) Hemos definido las formas canónicas; formas booleanas que permiten representar de manera única a las clases de equivalencia, y que se obtienen inmediatamente a la vista de la función de conmutación.

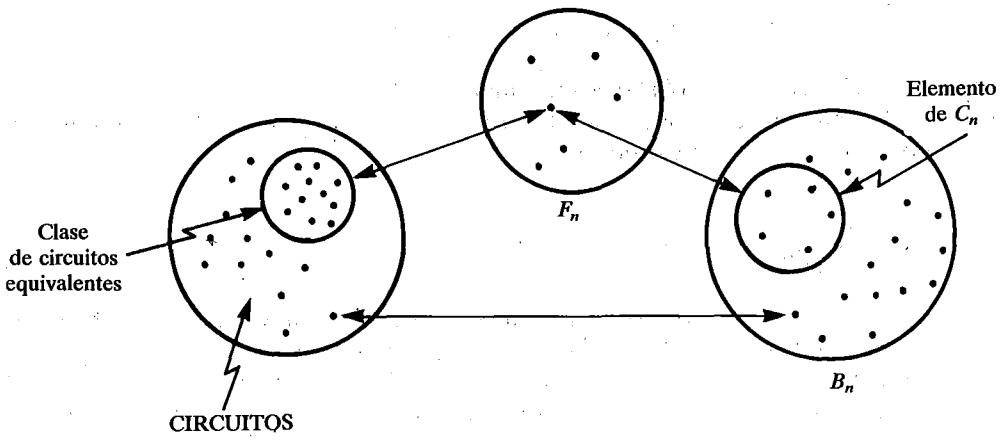


FIGURA 3.34.

* Esta es, desde luego, una simplificación del problema general. No siempre el circuito con menos puertas es el de menor coste de realización, y esto es especialmente cierto desde la aparición de los circuitos integrados.

f) Desde un punto de vista práctico, lo que interesa es, dada una función de conmutación, o una forma canónica, o una forma booleana cualquiera, encontrar la forma booleana que esté en la misma clase de equivalencia y que tenga el menor número posible de operaciones (que corresponderá a un circuito con el menor número posible de puertas). Es el problema de la minimización, para cuya solución hemos estudiado el método más sencillo: el de las tablas de Karnaugh.

g) También hemos analizado algunos casos en los que la función de conmutación está incompletamente especificada, lo que se aprovecha asimismo para minimizar el circuito.

h) Finalmente, hemos visto cómo a través de unas sencillas transformaciones cualquier forma booleana puede expresarse mediante otra equivalencia que sólo contiene operaciones «NAND» o sólo operaciones «NOR», lo que permite diseñar circuitos con uno u otro de esos tipos de puertas exclusivamente.

11. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

Fue Shannon (1938) el primero en aplicar el álgebra de Boole como modelo matemático para los circuitos de conmutación (cuyos componentes básicos, en esa época, eran siempre de tecnología electromecánica). En los años siguientes, diversos autores, entre ellos el propio Shannon (1949), desarrollaron la teoría. El método gráfico para minimización lo introdujo Veitch (1952) y lo perfeccionó Karnaugh (1953).

La tecnología de realización de circuitos digitales ha evolucionado muy rápidamente en los últimos años, pero las técnicas básicas de diseño, independientes de esa tecnología, son prácticamente las mismas de los años 60. Por eso, libros de esa época, como los de Bartee (1960), Bartee *et al.* (1962), Harrison (1965), Kohavi (1970) y Oberman (1970) son aún referencias válidas. Algunos de ellos se han ido actualizando para recoger los avances tecnológicos. Por ejemplo, el primero, en su sexta edición, de 1985, incluye descripciones de circuitos integrados y de microprocesadores de 16 y 32 bits, además de otros temas propios de las estructuras de los ordenadores: tecnologías de memorias, de buses, de dispositivos periféricos, etc.

Otros textos recomendables son los de Nagle *et al.* (1975), Clements (1985) (éste es también, como el de Bartee, un libro sobre estructura de ordenadores), o, en español, Mandado (1977), Hill y Peterson (1978) y Muñoz (1983). En el campo muy especializado, del diseño de circuitos lógicos con tecnología de integración de muy alta densidad (VLSI) hay que citar, inevitablemente, el texto ya clásico de Mead y Conway (1980).

12. EJERCICIOS

12.1. Considérese el Ejemplo 5.7.4 del capítulo 2. Cada cláusula se corresponde, en el lenguaje de las formas booleanas, con una suma de variables. (Concretamente, P2 es una suma canónica). Representar en una tabla de Karnaugh las interpretaciones de la conjunción de las tres cláusulas, y ver cómo las dos aplicaciones de la regla de resolución

que hacíamos allí se corresponden con reducciones de términos adyacentes, y tienen su representación gráfica como agrupaciones de «0».

12.2. Diseñar un circuito con cuatro entradas, x_0 a x_3, que representan los bits componentes de los números 0 a 15 expresados en binario (0000 a 1111), y una salida, f, que sólo debe tomar el valor «1» en el caso de que el número presentado a la entrada sea múltiplo de 3. En éste, como en los siguientes ejercicios, se diseñarán cuatro circuitos minimizados: utilizando puertas «NOT», «AND» y «OR», por los dos métodos (forma mínima en suma de productos y forma mínima en producto de sumas), utilizando sólo puertas «AND» y utilizando sólo puertas «OR».

12.3. El mismo ejercicio anterior, considerando que los números que se presentan a la entrada están siempre comprendidos entre 0 y 9.

12.4. Los dígitos decimales pueden codificarse en binario como se hace en el anterior ejercicio: asignando a cada uno el conjunto de 4 bits que corresponde a su escritura en el sistema de numeración de base 2. Pero éste, que se llama código BCD «natural», no es el único código BCD (Binary Coded Decimal) posible: como sólo necesitamos 10 de los 16 símbolos que pueden codificarse con 4 bits, hay, realmente, $V_{16,10} = 16!/6! = 2{,}9 * 10^{10}$ códigos BCD diferentes. Tres de los más utilizados (incluyendo el «8-4-2-1», que es el también llamado «natural»), son:

	8-4-2-1	2-4-2-1	Exceso de 3
0	0000	0000	0011
1	0001	0001	0100
2	0010	0010	0101
3	0011	0011	0110
4	0100	0100	0111
5	0101	0101	1000
6	0110	0110	1001
7	0111	0111	1010
8	1000	1110	1011
9	1001	1111	1100

Diseñar un circuito para pasar del «8-4-2-1» a uno de los otros dos. El circuito tendrá cinco entradas: las cuatro correspondientes a los bits del código fuente y una entrada de control, C. Si $C = 0$, entonces sus cuatro salidas deben dar los bits del código «2-4-2-1» que correspondan, y si $C = 1$ en la salida se deberá obtener el código «exceso de 3».

12.5. Diseñar un circuito que permita obtener el cuadrado de cualquier número comprendido entre 0 y 7.

12.6. Un codificador es un circuito con 2^n entradas y n salidas que genera a la salida el código correspondiente a la entrada cuyo valor es «1» (se supone que en cada momento sólo una de las entradas puede tomar el valor «1»). Diseñar un codificador con cuatro entradas (y dos salidas). Generalizar el diseño para ocho entradas (tres salidas).

12.7. Un multiplexor es un circuito con 2^n entradas de información binaria y n entradas de selección y una sola salida. Esta salida toma el valor de la entrada de información que haya sido seleccionada por las entradas de selección. Diseñar un multiplexor con $n = 3$.

Capítulo 4
LOGICA DE PREDICADOS DE PRIMER ORDEN

1. INTRODUCCIÓN

1.1. Variables y constantes

En el capítulo anterior vimos un ejemplo (el 1.4.4) en el que parecía natural introducir la idea de variable para referirnos a un miembro de un colectivo. Hay razonamientos imposibles de formalizar si no es formalizando esa idea. Por ejemplo, el clásico

todos los hombres son mortales
Sócrates es un hombre
luego Sócrates es mortal

Si tratamos de formalizar este razonamiento en lógica de proposiciones, para la primera premisa podríamos escribir la sentencia «$h \rightarrow m$». La segunda premisa, al ser un simple enunciado declarativo, se formalizaría como una simple variable proposicional, «s». Y de tales dos premisas es imposible deducir ninguna conclusión (salvo, según nuestra definición de «deducción», las triviales: las propias premisas, su conjunción y disyunción, y tautologías: $h \vee \neg h$, etc.).

Lo que ocurre en este ejemplo es que en la primera premisa estamos diciendo algo de todos los miembros de un colectivo, que, por tanto, es válido para uno *cualquiera* de ellos, que representaremos como una *variable*. En la segunda, hablamos de un valor determinado de esa variable, de una *constante*. Y la relación de deducibilidad que aplica el razonamiento es un paso de lo general a lo particular: lo que se dice de todos los miembros de un colectivo es válido para uno cualquiera de ellos. Para expresarla es preciso entrar en la composición de los enunciados, cosa que no puede

hacerse en lógica de proposiciones, en la que todo enunciado declarativo simple se representa como una variable proposicional.

1.2. Propiedades y relaciones

En el ejemplo anterior hablamos de *propiedades* de individuos: la propiedad de ser hombre y la propiedad de ser mortal. Y estas propiedades las aplicábamos a un individuo cualquiera (una variable) o a uno concreto (una constante). Otros enunciados se refieren a *relaciones* entre individuos. Por ejemplo, en «Juan ama a María» establecemos una relación unidireccional («ama a») entre dos individuos determinados (dos constantes), en «algunos hombres aman en secreto a Ana Belén» la relación es entre una variable (hombre) y una constante (Ana Belén), y en «todos los hombres aman a alguien» es entre dos variables.

Pueden establecerse relaciones entre un número cualquiera de individuos. Por ejemplo, cuando decimos «Juan regala flores a María» establecemos una relación ternaria («regala») entre tres constantes.

1.3. Predicados y fórmulas atómicas

Un predicado es la formalización de una propiedad o de una relación. Para propiedades, tendremos *predicados monádicos*, que tienen un solo argumento (constante o variable). Seguiremos la notación de expresar los predicados con letras mayúsculas, y las constantes con las minúsculas *a, b, c, ...*, reservando *x, y, z* para las variables. El argumento se pondrá entre paréntesis a continuación del predicado. Por ejemplo:

«Sócrates es mortal»: $M(s)$
«alguien es mortal» : $M(x)$

Para relaciones entre dos individuos tendremos los *predicados diádicos*:

«Juan ama a María» : $A(j, m)$
«alguien ama a Ana»: $A(x, a)$
«alguien ama a otro»: $A(x, y)$

Y, en general, tendremos predicados poliádicos:

«Juan regala flores a María»: $R(j, m, f)$

(Gramaticalmente, esta frase tendría un sujeto, «Juan», y un predicado, formado por los verbos y los complementos. Pero en la definición que se da en lógica el predicado es sólo el verbo, y los complementos, junto con el sujeto, son los argumentos de ese predicado).

La construcción formada por un predicado seguido de sus argumentos se llama

fórmula atómica. Obsérvese que las fórmulas atómicas representan a enunciados declarativos; corresponden, pues, a las variables proposicionales, en cuya composición ahora estamos entrando.

1.4. Sentencias abiertas y cerradas

Igual que en lógica de proposiciones construíamos sentencias enlazando variables proposicionales con las conectivas, aquí lo haremos enlazando fórmulas atómicas con las mismas conectivas. (Una simple fórmula atómica también será una sentencia).

Así como en lógica de proposiciones podíamos interpretar las variables proposicionales como verdaderas o falsas, y, a partir de una determinada interpretación, calcular la interpretación de la sentencia, ahora no siempre podemos hacerlo. Porque si tenemos, por ejemplo, un predicado monádico, su valor de verdad o falsedad normalmente dependerá del valor que tome la variable: si digo «x es español», o, formalmente, $E(x)$, dependiendo de quién sea x, el resultado será verdadero o no.

En general, tendremos *sentencias abiertas*, que son aquellas que, al depender de variables, no se les puede calcular un valor de verdad o falsedad, y *sentencias cerradas*, en las que tal cálculo es posible.

Un modo obvio de *cerrar* una sentencia abierta consiste en fijar valores para las variables que intervienen en ellas. En el último ejemplo, si hacemos x = Picasso, la sentencia (en este caso, fórmula atómica) $E(x)$ es verdadera, y si x = Matisse, $E(x)$ es falsa.

Pero las sentencias pueden ser también verdaderas o falsas sin que tengan que referirse exclusivamente a constantes: «todos los hombres son mortales» es una sentencia verdadera, y, por tanto, cerrada. Ello es así porque lo que se predica es algo sobre una variable, pero que es válido para *todos* los valores de la variable. Por tanto, a la sentencia $H(x) \rightarrow M(x)$ (si alguien, x, es hombre entonces es mortal) hay que añadirle algo que exprese su validez para cualquier valor de x, cerrando de ese modo la sentencia. Ese algo se llama «cuantificador universal».

1.5. Cuantificadores

Representaremos el *cuantificador universal* con el símbolo clásico «$\forall$» seguido de la variable que se cuantifica. Para delimitar el *alcance* de la cuantificación, pondremos la sentencia cuya variable se cuantifica entre paréntesis. Por ejemplo,

$$(\forall x)(H(x) \rightarrow M(x))$$

es una sentencia cerrada, que formaliza la frase «para todo x, si x tiene la propiedad H, entonces tiene la propiedad M», mientras que

$$(\forall x)(H(x)) \rightarrow M(x)$$

sería una sentencia abierta («si todos los x tienen la propiedad H, entonces x, cualquiera, tiene la propiedad M»), lo mismo que

$$(\forall x)(H(x) \rightarrow M(y))$$

(«para todos los x, si x tiene la propiedad H, entonces y tiene la propiedad M», que viene a ser lo mismo de antes, supuesto que x e y tienen un rango común).

El otro cuantificador, también conocido del lenguaje matemático, es el *cuantificador existencial*, que representaremos con el símbolo «$\exists$», y que se lee «existe un ... tal que ...».

En realidad, bastaría con uno solo de los dos cuantificadores, y si usamos los dos es por comodidad (por acercar el lenguaje lógico al lenguaje natural). En efecto, decir que todos los individuos en consideración tienen cierta propiedad es lo mismo que decir que no es cierto que exista algún individuo que no tenga esa propiedad:

$$(\forall x)(P(x)) \equiv \neg\, (\exists x)(\neg\, P(x))$$

Y decir que existe un x (al menos) que tenga cierta propiedad es lo mismo que decir que no es cierto que para todos los individuos la propiedad no se da:

$$(\exists x)(P(x)) \equiv \neg\, (\forall)(\neg\, P(x))$$

1.6. Interpretación binaria

Como ya hemos dicho, una fórmula atómica sólo puede interpretarse como verdadera o falsa si sus variables toman valores concretos. Por ejemplo, si $H(x)$ significa «x es hombre», la fórmula será verdadera para x = Sócrates, pero falsa para x = Rocinante, y, en general, no podremos decir que $H(x)$ sea verdadera ni falsa.

De igual modo, una sentencia abierta no puede interpretarse si no se cierra. Pero hemos de considerar varias posibilidades:

a) Si una sentencia es cerrada porque a todas las variables que intervienen en ella se les ha asignado un valor constante, entonces las fórmulas atómicas juegan el mismo papel que las variables proposicionales, y las distintas interpretaciones de la sentencia pueden escribirse en una tabla de verdad, con una línea para cada una de las posibles interpretaciones del conjunto de fórmulas atómicas.

b) Si la sentencia es cerrada porque todas sus variables están cuantificadas, entonces cabe considerar dos casos:

   b1) Si el «universo del discurso» (conjunto en el que toman valores las variables) es finito, entonces el cuantificador universal puede sustituirse por un número finito de conjunciones, y el cuantificador existencial, por un número finito de disyunciones. En efecto, si, por ejemplo, los valores posibles de x son $\{a, b, c, d\}$, la sentencia cerrada

$$(\forall x)(P(x))$$

es equivalente a

$$P(a) \wedge P(b) \wedge P(c) \wedge P(d)$$

y la sentencia cerrada

$$(\exists x)(P(x))$$

es equivalente a

$$P(a) \vee P(b) \vee P(c) \vee P(d)$$

Estas equivalencias proceden de las mismas definiciones de los cuantificadores, que, realmente, son generalizaciones de las operaciones de conjunción y disyunción para sentencias construidas con variables de cualquier rango. En tal caso, podemos escribir la tabla de verdad de cualquier sentencia en la que intervenga $P(x)$ con x cuantificada, considerando todas las interpretaciones posibles del conjunto $\{P(a), P(b), P(c), P(d)\}$ (que ahora ya es como si fuera un conjunto de variables proposicionales).

b2) Si el universo del discurso es infinito (o inabordable) es claro que no podemos construir ninguna tabla de verdad. Pese a ello, hay sentencias que siempre son verdaderas. Por ejemplo, nadie dudará de que

$$(\forall x)(P(x)) \rightarrow (\exists x)(P(x))$$

es siempre verdadera, sea finito o infinito el universo del discurso.

A las sentencias que son siempre verdaderas se les llama, en lógica de proposiciones, tautologías. En lógica de predicados se les suele llamar *sentencias válidas*.

1.7. Ejemplos de formalización e interpretación

Ejemplo 1.7.1. Consideremos el razonamiento

todos los hombres son mortales
Sócrates es hombre
---
luego Sócrates es mortal

Su formalización sería:

$P1$: $(\forall x)(H(x) \rightarrow M(x))$
$P2$: $H(s)$

C : $M(s)$

Si el razonamiento es correcto, la sentencia

$$[(\forall x)(H(x) \rightarrow M(x))] \wedge H(s) \rightarrow M(s)$$

tiene que ser válida, o, lo que es lo mismo, todas las interpretaciones que satisfacen a la conjunción de las dos premisas han de satisfacer también a la conclusión para que ésta se deduzca de aquélla. Ahora bien, como el universo del discurso es inabordable, no podemos comprobar tal cosa construyendo una tabla de verdad. Pero hagámoslo considerando que el universo se restringiese a dos hombres, Sócrates (s) y Platón (p), y a una entidad desconocida (d). La premisa $P1$ se escribiría entonces como $P1s \wedge P1p \wedge P1d$. con

$$P1s\colon H(s) \rightarrow M(s)$$
$$P1p\colon H(p) \rightarrow M(p)$$
$$P1d\colon H(d) \rightarrow M(d)$$

Y la tabla de verdad sería:

$H(s)$	$M(s)$	$H(p)$	$M(p)$	$H(d)$	$M(d)$	$P1s$	$P1p$	$P1d$	$P1$	$P2$	C	$P1 \wedge P2 \rightarrow C$
0	0	0	0	0	0	1	1	1	1	0	0	1
0	0	0	0	0	1	1	1	1	1	0	0	1
0	0	0	0	1	0	1	1	0	0	0	0	1
0	0	0	0	1	1	1	1	1	1	0	0	1
0	0	0	1	0	0	1	1	1	1	0	0	1
0	0	0	1	0	1	1	1	1	1	0	0	1
0	0	0	1	1	0	1	1	0	0	0	0	1
0	0	0	1	1	1	1	1	1	1	0	0	1
0	0	1	0	0	0	1	0	1	0	0	0	1
0	0	1	0	0	1	1	0	1	0	0	0	1
0	0	1	0	1	0	1	0	0	0	0	0	1
0	0	1	0	1	1	1	0	1	0	0	0	1
0	0	1	1	0	0	1	1	1	1	0	0	1
0	0	1	1	0	1	1	1	1	1	0	0	1
0	0	1	1	1	0	1	1	0	0	0	0	1
0	0	1	1	1	1	1	1	1	1	0	0	1
0	1	0	0	0	0	1	1	1	1	0	1	1
0	1	0	0	0	1	1	1	1	1	0	1	1
0	1	0	0	1	0	1	1	0	0	0	1	1
0	1	0	0	1	1	1	1	1	1	0	1	1
0	1	0	1	0	0	1	1	1	1	0	1	1
0	1	0	1	0	1	1	1	1	1	0	1	1
0	1	0	1	1	0	1	1	0	0	0	1	1
0	1	0	1	1	1	1	1	1	0	0	1	1
0	1	1	0	0	0	1	0	1	0	0	1	1
0	1	1	0	0	1	1	0	1	0	0	1	1
0	1	1	0	1	0	1	0	0	0	0	1	1
0	1	1	0	1	1	1	0	1	0	0	1	1

$H(s)$	$M(s)$	$H(p)$	$M(p)$	$H(d)$	$M(d)$	$P1s$	$P1p$	$P1d$	$P1$	$P2$	C	$P1 \wedge P2 \rightarrow C$
0	1	1	1	0	0	1	1	1	1	0	1	1
0	1	1	1	0	1	1	1	1	1	0	1	1
0	1	1	1	1	0	1	1	0	0	0	1	1
0	1	1	1	1	1	1	1	1	1	0	1	1
1	0	0	0	0	0	0	1	1	0	1	0	1
1	0	0	0	0	1	0	1	1	0	1	0	1
1	0	0	0	1	0	0	1	0	0	1	0	1
1	0	0	0	1	1	0	1	1	0	1	0	1
1	0	0	1	0	0	0	1	1	0	1	0	1
1	0	0	1	0	1	0	1	1	0	1	0	1
1	0	0	1	1	0	0	1	0	0	1	0	1
1	0	0	1	1	1	0	1	1	0	1	0	1
1	0	1	0	0	0	0	0	1	0	1	0	1
1	0	1	0	0	1	0	0	1	0	1	0	1
1	0	1	0	1	0	0	0	0	0	1	0	1
1	0	1	0	1	1	0	0	1	0	1	0	1
1	0	1	1	0	0	0	1	1	0	1	0	1
1	0	1	1	0	1	0	1	1	0	1	0	1
1	0	1	1	1	0	0	1	0	0	1	0	1
1	0	1	1	1	1	0	1	1	0	1	0	1
1	1	0	0	0	0	1	1	1	1	1	1	1
1	1	0	0	0	1	1	1	1	1	1	1	1
1	1	0	0	1	0	1	1	0	0	1	1	1
1	1	0	0	1	1	1	1	1	1	1	1	1
1	1	0	1	0	0	1	1	1	1	1	1	1
1	1	0	1	0	1	1	1	1	1	1	1	1
1	1	0	1	1	0	1	1	0	0	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1
1	1	1	0	0	0	1	0	1	0	1	1	1
1	1	1	0	0	1	1	0	1	0	1	1	1
1	1	1	0	1	0	1	0	0	0	1	1	1
1	1	1	0	1	1	1	0	1	0	1	1	1
1	1	1	1	0	0	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1	1
1	1	1	1	1	0	1	1	0	0	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1

Tras escribir toda esta tabla, comprobamos que, para este universo finito, la sentencia que formaliza al razonamiento es una tautología, y, por tanto, la conclusión se deduce de las premisas. Pero comprobamos también que, además de tedioso, este procedimiento (que en lógica de proposiciones podría tener algún sentido) es inútil, porque, en principio, no nos garantiza que el razonamiento siga siendo correcto para un universo del discurso mayor.

Ahora bien, se ha demostrado que si hay n predicados monádicos cualquier sentencia válida en un universo que tenga 2^n elementos es válida en todo universo. En este ejemplo, como hay dos predicados monádicos, bastaría comprobar la validez en

un universo de cuatro elementos, lo cual sería factible. Pero para un número mayor de predicados el procedimiento deja de ser práctico, y si hay predicados poliádicos no nos garantiza nada. De hecho, también se ha demostrado que no hay un procedimiento general que permita determinar la validez de cualquier sentencia del cálculo de predicados. Por ello, se dice que el cálculo de predicados es *indecidible*. Sin embargo, sí existe un procedimiento tal que si una sentencia es válida termina dictaminándolo (y si no lo es, no termina), por lo que también se dice que el cálculo de predicados es *semidecidible*.

En cualquier caso, lo que nos interesa, como en la lógica de proposiciones, es disponer de sistemas de inferencia. En el apartado 4.5 generalizaremos el método de resolución. En los ejemplos que siguen nos limitaremos a formalizar los razonamientos, dejando al lector la construcción de tablas de verdad para comprobar su validez con universos finitos.

Ejemplo 1.7.2.

ningún profesional es diletante
todos los buenos informáticos son profesionales
---
ningún buen informático es diletante

Con los predicados P(profesional), D(diletante) e I(buen informático), una formalización de este razonamiento (que es un caso del llamado «modo silogístico Celarent» en la lógica clásica) es:

$$P1: (\forall x)(P(x) \rightarrow \neg D(x))$$
$$P2: (\forall x)(I(x) \rightarrow P(x))$$
$$C : (\forall x)(I(x) \rightarrow \neg D(x))$$

Y la sentencia global sería: $P1 \wedge P2 \rightarrow C$.

Teniendo en cuenta las relaciones que hay entre las conectivas y entre los cuantificadores, la formalización no es única. Por ejemplo, la primera premisa puede también escribirse así:

$$P1: \neg (\exists x)(P(x) \wedge D(x))$$

Ejemplo 1.7.3. Definiendo los predicados J, C, D y S en sustitución de las variables proposicionales j, c, d y s, el Ejemplo 1.4.4 del capítulo 2 se formalizaría así:

$$P1: \neg (\exists x)(J(x) \wedge C(x))$$
$$P2: (\forall x)(\neg J(x) \rightarrow \neg D(x))$$
$$P3: (\forall x)(S(x) \rightarrow C(x))$$
$$C : (\forall x)(S(x) \rightarrow C(x))$$

Ejemplo 1.7.4.

Ana es madre de Luis
José es padre de Ana
Un abuelo de una persona es alguien que es padre del padre o de la madre de esa persona
---
José es abuelo de Luis

Definiendo los predicados diádicos $P(x, y)$ (x es padre de y), $M(x, y)$ (x es madre de y) y $A(x, y)$ (x es abuelo de y), podemos establecer la siguiente formalización:

$P1$: $M(a, l)$
$P2$: $P(j, a)$
$P3$: $(\forall x)(\forall y)(\forall z)(P(x, y) \wedge (P(y, z) \vee M(y, z)) \rightarrow A(x, z))$

C : $A(j, l)$

Obsérvese que la premisa 3 establece una *definición de la relación* «abuelo de» en función de las relaciones «padre de» y «madre de». Las premisas 1 y 2 son *hechos* concretos que junto con esa definición nos permiten deducir otro hecho.

Ejemplo 1.7.5.

«Un antepasado de una persona es alguien que o bien es padre o madre de esa persona o bien es antepasado de su padre o de su madre».

Esta es una *definición recursiva* de la relación «antepasado de». Dados unos hechos (relaciones «padre de» y «madre de» definidas sobre unos individuos concretos), se podrán obtener conclusiones sobre las relaciones «antepasado de» entre esos individuos. En lógica de predicados, podemos formalizar la definición con la siguiente sentencia (ahora, con $A(x, y)$ representamos «x es antepasado de y», y, para no hacer la sentencia excesivamente larga, con $P(x, y)$ representamos «x es padre o madre de y»; si quisiéramos conservar las dos relaciones diferenciadas, tendríamos que escribir $P(x, y) \vee M(x, y)$ en donde aparece $P(x, y)$):

$$(\forall x)(\forall y)(\forall z)[(P(x, y) \vee (A(x, z) \wedge P(z, y))) \rightarrow A(x, y)]$$

Pero, para mayor claridad, es preferible descomponer la sentencia en dos, una no recursiva y la otra sí:

$$(\forall x)(\forall y)(P(x, y) \rightarrow A(x, y))$$
$$(\forall x)(\forall y)(\forall z)(A(x, z) \wedge (P(z, y)) \rightarrow A(x, y))$$

Esta descomposición es posible porque la sentencia $P1 \vee P2 \rightarrow C$ es equivalente a $(P1 \rightarrow C) \wedge (P2 \rightarrow C)$.

1.8. Funciones

Para terminar esta introducción informal, tenemos que hacer referencia a las *funciones*. Se trata de una construcción del cálculo de predicados que no es necesaria en la lógica clásica, pero que, como veremos, es muy útil para poder expresar las sentencias en forma clausulada como paso previo a la aplicación de la resolución.

Las funciones, como los predicados, tienen argumentos que pueden ser constantes o variables. Pero, a diferencia de los predicados, no representan ninguna propiedad o relación entre los argumentos que pueda interpretarse como verdadera o falsa. Una función en cálculo de predicados es una función matemática definida en el universo del discurso. Por ejemplo, podemos definir la función «padre», p. Aplicada a un individuo, nos daría como resultado otro individuo, $p(x)$ (su padre). Si definimos un predicado de igualdad, $I(x, y)$, que es verdadero cuando x es el mismo individuo que y, y falso en otro caso, sería lo mismo escribir $P(x, y)$ que $I(x, p(y))$. Vemos con este ejemplo que los argumentos de los predicados (y de las mismas funciones) pueden ser funciones, lo cual nos lleva a ampliar el concepto de fórmula atómica, cosa que haremos enseguida.

2. SINTAXIS

2.1. Alfabeto

Utilizaremos los siguientes símbolos (en su caso, y si es menester, con subíndices):

- Variables proposicionales: $p, q, r, \ldots$ (normalmente, no serán necesarias; en su lugar tendremos fórmulas atómicas).
- Variables: x, y, z, u, v, w.
- Constantes: $a, b, c, \ldots$
- Símbolos de función: f, g, h.
- Símbolos de predicado: P, Q, R.
- Conectivas (las mismas de la lógica de proposiciones).
- Cuantificadores: $\forall$, $\exists$.
- Símbolos de puntuación: (,), [,], {, }.
- Metasímbolos:
  - $A, B, C, \ldots$ para cualquier sentencia.
  - $A(x)$ para una sentencia con la variable libre x.
  - k para cualquier conectiva binaria.
  - L para cualquier literal (fórmula atómica positiva o negativa).
  - t para cualquier «término», una construcción que enseguida vamos a definir.

Igual que hacíamos en lógica de proposiciones, podemos definir una *expresión* como una secuencia cualquiera de símbolos del alfabeto, y una *sentencia* como una expresión «bien construida» de acuerdo con las reglas propias del lenguaje y que vamos a ver a continuación.

2.2. Términos y fórmulas atómicas

Definición 2.2.1. Un *término* se define recursivamente:

* Las constantes y las variables son términos.
* Si f es un símbolo de función y $t_1, t_2, \ldots, t_n$ son términos, entonces $f(t_1, t_2, \ldots, t_n)$ es un término.

(Podríamos haber dado una definición equivalente, no recursiva, definiendo previamente una «secuencia de formación de términos», lo mismo que hacíamos en lógica de proposiciones para las sentencias).

Definición 2.2.2. Una *fórmula atómica* es una expresión de la forma $P(t_1, t_2, \ldots, t_n)$, donde P es un símbolo de predicado y $t_1, t_2, \ldots, t_n$ son términos.

2.3. Sentencias

Definición 2.3.1. Llamaremos *secuencia de formación* a toda secuencia finita de expresiones, $A_1, A_2, \ldots, A_n$ en la que cada A_i satisface al menos una de las cinco condiciones siguientes:

a) A_i es una fórmula atómica (o una variable proposicional);
b) existe un j menor que i tal que A_i es el resultado de anteponer el símbolo «$\neg$» a A_j;
c) existen j y h menores que i tales que A_i es el resultado de enlazar A_j y A_h con k;
d) existe un j menor que i tal que A_i es lo mismo que $(\forall x)(A_j(x))$, donde x es cualquier variable;
e) existe un j menor que i tal que A_i es lo mismo que $(\exists x)(A_j(x))$, donde x es cualquier variable.

Definición 2.3.2. Llamaremos *sentencia* (o *fórmula molecular*) a toda expresión A_n tal que existe una secuencia de formación $A_1, A_2, \ldots, A_n$. (Todas las expresiones de la secuencia serán sentencias).

Alternativamente, podemos definir la sentencia de modo recursivo, mediante cinco *reglas de formación*:

RF1: Una fórmula atómica (o una variable proposicional) es una sentencia.
RF2: Si A es una sentencia, $\neg A$ también lo es.
RF3: Si A y B son sentencias, $A \; k \; B$ también lo es.
RF4: Si $A(x)$ es una sentencia con la variable libre x $(\forall x)(A(x))$, también lo es.
RF5: Si $A(x)$ es una sentencia con la variable libre x $(\exists x)(A(x))$, también lo es.

Según cualquiera de estas definiciones, una expresión como

$$(\forall x)(\exists y)(P(x, y))$$

no sería una sentencia. En su lugar habría que escribir

$$(\forall x)((\exists y)(P(x, y)))$$

No obstante, por comodidad, utilizaremos preferentemente la primera notación.

2.4. Sentencias abiertas y cerradas

Definición 2.4.1. Llamaremos *alcance* de un cuantificador a la sentencia A que le sigue en las reglas de formación RF4 y RF5. Diremos que la variable que sigue al símbolo de cuantificación está *cuantificada* en la sentencia A.

Definición 2.4.2. Diremos que una sentencia A es *cerrada* si todas las variables que intervienen en las fórmulas atómicas que la componen están cuantificadas. En caso contrario, diremos que la sentencia A es *abierta*.

Aunque aún no hemos entrado en el campo de la semántica, por lo visto en el apartado anterior sabemos que sólo a las sentencias cerradas se les puede dar una interpretación («verdadera» o «falsa», en el caso binario). En lo sucesivo, salvo que digamos lo contrario, supondremos que todas las sentencias son cerradas. De hecho, en otros libros, el nombre de «sentencia» se reserva para lo que aquí llamamos «sentencias cerradas», y lo que llamamos «sentencia» se denomina «fórmula bien formada» (en los libros escritos en inglés, «wff», de «well formed formula»), denominación, cuando menos, discutible: ¿que sería una «fórmula mal formada»?; con nuestra terminología, una expresión que no es sentencia, pero ¿sería entonces lícito llamarle «fórmula»?

2.5. Axiomas, demostraciones y teoremas

2.5.1. *Axiomas*

Los conceptos de sistema axiomático, de completitud y de consistencia ya fueron expuestos de una manera general en el capítulo 2 (apartado 1.5). El sistema axiomático de la lógica de predicados es una extensión del visto para la lógica de proposiciones (o éste es una restricción del primero); por ello, aquí nos iremos refiriendo continuamente al apartado 2.5 del capítulo 2. Y, en particular, los axiomas son los mismos expuestos en 2.5.1, a los que ahora añadimos estos dos:

$$\text{A5: } (\forall x)(P(x)) \rightarrow P(a)$$
$$\text{A6: } (\forall x)(p \rightarrow P(x)) \rightarrow (p \rightarrow (\forall x)(P(x)))$$

Las leyes que interrelacionan a los cuantificadores,

$$(\forall x)(P(x)) \leftrightarrow \neg\, (\exists x)(\neg\, P(x))$$
$$(\exists x)(P(x)) \leftrightarrow \neg\, (\forall x)(\neg\, P(x))$$

pueden considerarse como definición de un símbolo en función de otro, o, si se prefiere que ambos símbolos sean primitivos, una sería un axioma y la otra un teorema.

El axioma A5 (llamado *«ley de especificación»*) significa que si algo (P) puede afirmarse de todos los individuos, entonces puede afirmarse de uno cualquiera de ellos: a es una constante arbitraria definida sobre el mismo universo del discurso que x. Igualmente, en A6, p es una variable proposicional arbitraria.

2.5.2. *Sustitución*

En lógica de proposiciones, la sustitución era de variables proposicionales por sentencias. Ahora, además, podremos sustituir fórmulas atómicas por sentencias en las que figuren los mismos argumentos, y variables por términos en los que no aparezcan otras variables de la sentencia.

Definición 2.5.2.1. Dadas unas variables proposicionales p_i y unas sentencias cerradas B_i, unas fórmulas atómicas $F_j = P_j(x_{j1}, \ldots, x_{jn}, a_1, \ldots, a_m)$ y unas sentencias S_j con las variables libres $x_{j1}, \ldots, x_{jn}$ y las constantes $a_1, \ldots, a_m$, y unas variables x_k y unos términos t_k, llamaremos *sustitución* a un conjunto de pares ordenados

$$s = \{\ldots, B_i/p_i, \ldots, S_j/F_j, \ldots, t_k/x_k, \ldots\}$$

Definición 2.5.2.2. Dada una sentencia A que contiene las variables proposicionales p_i, las fórmulas atómicas F_j y las variables x_k, y una sustitución s, la *operación de sustitución* de unas por otras en A consiste en poner en A:

a) en todos los lugares donde aparezca p_i, B_i;
b) en todos los lugares donde aparezca F_j, S_j;
c) en todos los lugares donde aparezca x_k, t_k, salvo detrás de los cuantificadores, donde se pondrán las variables que aparecen en t_k precedidas del mismo cuantificador que afectaba a x_k. Además, para que la regla de sustitución que veremos luego sea correcta, t_k no puede contener variables que ya figuren en A.

La justificación de esta última restricción puede verse claramente con un sencillo ejemplo: consideremos la sentencia

$$(\forall x)(\exists y)(D(x, y))$$

donde $D(x, y)$ significa «x es diferente de y»; podemos hacer la sustitución $s = \{z/x\}$, con la que obtenemos

$$(\forall z)(\exists y)(D(z, y))$$

pero no $s = \{y/x\}$, con la que resultaría

$$(\forall y)(\exists y)(D(y, y))$$

Teorema 2.5.2.3. Si A es una sentencia y s una sustitución, el resultado de efectuar la operación de sustitución de s en A es otra sentencia que representaremos como As. La demostración, que omitimos, consiste en ver que As tiene una secuencia de formación.

2.5.3. *Demostraciones y teoremas*

Para este apartado es íntegramente aplicable todo lo dicho en el de igual numeración del capítulo 2.

2.5.4. *Ejemplos de demostraciones*

Todos los teoremas de la lógica de proposiciones son aplicables también en lógica de predicados. Veamos algunos otros específicos de ésta.

Teorema 1: $P(a) \rightarrow (\exists x)(P(x))$ *(«Ley de particularización»).*

Demostración:

1. $(\forall x)(\neg P(x)) \rightarrow \neg P(a)$
   (Por sustitución en A5: $\{\neg P/P\}$).
2. $(p \rightarrow \neg q) \rightarrow (q \rightarrow \neg p)$
   (Teorema de la lógica de proposiciones).
3. $((\forall x)(\neg P(x)) \rightarrow \neg P(a)) \rightarrow (P(a) \rightarrow \neg (\forall x)(\neg P(x)))$
   (Por sustitución en 2: $\{(\forall x)(\neg P(x))/p, P(a)/q\}$.
4. $P(a) \rightarrow \neg (\forall x)(\neg P(x))$
   (Por separación de 1 y 3).
5. $P(a) \rightarrow (\exists x)(P(x))$
   (Por definición de «$\exists$»).

Teorema 2: $(\forall x)(P(x)) \rightarrow (\exists x)(P(x))$.

Demostración:

1. $((\forall x)P(x) \rightarrow P(a)) \wedge (P(a) \rightarrow (\exists x)(P(x)))$
   (Por unión de A5 y el teorema 1).
2. $((p \rightarrow q) \wedge (q \rightarrow r)) \rightarrow (p \rightarrow r)$
   (Ley de transitividad de la lógica de proposiciones).
3. $(((\forall x)P(x) \rightarrow P(a)) \wedge (P(a) \rightarrow (\exists x)(Px))) \rightarrow ((\forall x)(P(x)) \rightarrow (\exists x)(P(x)))$
   (Por sustitución en 2: $\{(\forall x)(P(x))/p, P(a)/q, (\exists x)(P(x))/r\}$.
4. $(\forall x)(P(x)) \rightarrow (\exists x)(P(x))$
   (Por separación de 1 y 3).

2.5.5. *Algunos teoremas útiles*

La mayoría de las leyes de la lógica de proposiciones se pueden generalizar. Por ejemplo:

* *Ley del tercio excluso*:

$$(\forall x)(P(x) \vee \neg P(x))$$

* *Ley de distribución del cuantificador universal sobre la conjunción*:

$$(\forall x)(P(x) \wedge Q(x)) \rightarrow [(\forall x)(P(x)) \wedge (\forall y)(P(y))]$$

* *Leyes de modus ponens y modus tollens*:

$$((\forall x)(P(x) \rightarrow Q(x)) \wedge P(a)) \rightarrow Q(a)$$
$$((\forall x)(P(x) \rightarrow Q(x)) \wedge \neg Q(a)) \rightarrow \neg P(a)$$

* *Leyes de inferencia de la alternativa*:

$$(\neg P(a) \wedge (\forall x)(P(x) \vee Q(x))) \rightarrow Q(a)$$
$$(P(a) \wedge (\forall x)(\neg P(x) \vee \neg Q(x))) \rightarrow \neg Q(a)$$

Todas estas leyes, escritas para predicados monádicos, pueden generalizarse para predicados poliádicos. Por ejemplo, la de modus ponens para diádicos es:

$$((\forall x)(\forall y)(P(x, y) \rightarrow Q(x, y)) \wedge P(a, b)) \rightarrow Q(a, b)$$

Para la regla de resolución utilizaremos una generalización de las leyes de inferencia de la alternativa:

$$(\forall x_1) \ldots (\forall x_n)((\neg P(x_1, \ldots, x_n) \vee A) \wedge (P(x_1, \ldots, x_n) \vee B)) \rightarrow A \vee B$$

donde A y B son sentencias cualesquiera.

3. SEMÁNTICA

Definición 3.1. Dadas unas fórmulas atómicas construidas con unas variables, llamaremos *universo del discurso* al conjunto de valores posibles de esas variables.

Todo lo dicho en el apartado 3 del capítulo 2 es extensible a la lógica de predicados, con las siguientes matizaciones:

1) La *función de interpretación i*, que aplicaba el conjunto de variables proposicionales sobre el conjunto de valores semánticos, V, ahora extenderá su dominio para abarcar también a todas las fórmulas atómicas particularizadas sobre todos los elementos del universo del discurso. Es decir, si tenemos, por ejemplo, una fórmula

atómica, $P(x)$ y el universo del discurso es $\{a, b, c\}$, una interpretación será una asignación de valores de V a $P(a)$, $P(b)$ y $P(c)$.

2) La extensión del dominio de i al conjunto de las sentencias cerradas tendrá que tener en cuenta la presencia de cuantificadores, para lo que se añadirán dos condiciones:

$$\text{c) } I((\forall x)(A(x)) = I(A(a)) \wedge I(A(b) \wedge I(A(c)) \wedge \ldots$$
$$\text{d) } I((\exists x)(A(x)) = I(A(a)) \vee I(A(b)) \vee I(A(c)) \vee \ldots$$

donde $a, b, c, \ldots$ son todos los valores posibles de x.

3) Una sentencia cerrada cuyas interpretaciones binarias sean todas igual a 1 no se llama tautología, sino *sentencia válida*. Como hemos dicho en la explicación del Ejemplo 1.7.1, la lógica de predicados es *semidecidible*.

4) La relación de equivalencia en el conjunto de sentencias se define de igual modo, pero el número de clases de equivalencia, si el universo del discurso es infinito, es también infinito. Por lo demás, sigue siendo válido el teorema: $A \equiv B$ si y sólo si $\vdash (A \leftrightarrow B)$.

4. SISTEMAS INFERENCIALES

4.1. Teoremas y reglas de inferencia

Por las mismas razones expuestas para la lógica de proposiciones, a toda ley del cálculo de predicados que tenga la forma $P1 \wedge P2 \wedge \ldots \rightarrow C$ le corresponderá una regla de inferencia, en la que las sentencias $P1, P2, \ldots$ son las premisas y la sentencia C es la conclusión.

Por ejemplo, a la ley de modus ponens dada en el apartado 2.5.5 le corresponde la regla de inferencia

$$\frac{(\forall x)(P(x) \rightarrow Q(x)) \quad P(a)}{Q(a)}$$

(que es la que aplicábamos informalmente en el ejemplo con el que hemos comenzado este capítulo).

Los «sistemas de deducción natural» comprenden un conjunto de reglas de inferencia que, adecuadamente aplicadas, permiten disponer de un sistema inferencial *consistente*. A continuación vamos a ver cómo se generaliza la regla de resolución, explicada en el capítulo 2 para la lógica de proposiciones, a la lógica de predicados. La *completitud*, en general, no se puede demostrar debido a que el cálculo de predicados es indecidible. Sí se puede demostrar para casos particulares, por ejemplo, para la refutación con resolución (apartado 4.5).

4.2. Forma clausulada de la lógica de predicados

Definición 4.2.1. Una *cláusula* es una sentencia de la forma:

$$L_1 \vee L_2 \vee \ldots \vee L_n$$

donde los L_i son literales (fórmulas atómicas con o sin el símbolo «¬») con cualquier número de variables cada uno. Todas las variables se suponen cuantificadas universalmente, aunque no se escriba $(\forall x_1)(\forall x_2)$ ... delante de la cláusula.

Definición 4.2.2. Diremos que una sentencia está en *forma clausulada* si tiene la forma:

$$(L_{11} \vee L_{12} \vee \ldots) \wedge (L_{21} \vee L_{22} \ldots) \wedge \ldots$$

en la que cada cláusula $(L_{i1} \vee L_{i2} \vee \ldots)$ tiene sus propias variables.

De una cláusula se dice también que es una *colección de literales* (implícitamente unidos por disyunciones), y de una sentencia en forma clausulada, que es una *colección de cláusulas* (implícitamente unidas por conjunciones).

Se observará que, a partir de este momento, suponemos que no intervienen variables proposicionales en las sentencias de la lógica de predicados (pero, si existieran, bastaría tratarlas igual que a fórmulas atómicas en las que sólo interviniesen constantes).

Teorema 4.2.3. Para toda sentencia de la lógica de predicados existe una sentencia equivalente en forma clausulada.

La demostración es constructiva: vamos a ver un procedimiento, ampliación del dado para la lógica de proposiciones, que permite pasar a forma clausulada cualquier sentencia. Los pasos, ahora, son siete:

1. Eliminación de todas las conectivas que no sean «∨» o «∧» (normalmente, condicionales y bicondicionales). Se hace igual que en lógica de proposiciones:

$$(A \rightarrow B) \equiv (\neg A \vee B)$$
$$(A \leftrightarrow B) \equiv (\neg A \vee B) \wedge (A \vee \neg B)$$

2. Introducción de negaciones. Se seguirán utilizando las leyes de de Morgan:

$$\neg (A \vee B) \equiv (\neg A \wedge \neg B)$$
$$\neg (A \wedge B) \equiv (\neg A \vee \neg B)$$

y ahora, además, las relaciones ya conocidas entre los cuantificadores:

$$\neg (\exists x)(A) \equiv (\forall x)(\neg A)$$
$$\neg (\forall x)(A) \equiv (\exists x)(\neg A)$$

3. Independización de las variables cuantificadas. Se cambian (aplicando la regla de sustitución) los nombres de las variables que sean necesarios para que cada cuantificador se refiera a su propia variable. Por ejemplo,

$$(\forall x)(\neg P(x) \vee (\exists x)(Q(x)) \equiv (\forall x)(\neg P(x) \vee (\exists y)(Q(y))$$

4. Eliminación de los cuantificadores existenciales. Este es, de todos los pasos, el de más difícil justificación teórica. Consideraremos dos casos:

a) La sentencia abarcada por el cuantificador existencial a eliminar *no* está dentro del alcance de ningún cuantificador universal. En tal caso, nos limitamos a introducir una constante en el lugar de la variable cuantificada. Por ejemplo, si tenemos la sentencia

$$(\exists x)(V(x) \wedge M(e, x))$$

la transformamos en

$$V(a) \wedge M(e, a)$$

Ahora bien, para que esta transformación fuese correcta (de acuerdo con nuestro sistema axiomático) tendría que existir una ley como:

$$(\exists x)(P(x)) \rightarrow P(a)$$

Pero no existe tal ley. Podemos, de hecho, comprobar que esa sentencia no es válida, considerando que el universo del discurso se reduce a $\{a, b\}$. En ese caso, la sentencia equivale a:

$$(P(a) \vee (P(b)) \rightarrow P(a)$$

Y su tabla de verdad

$P(a)$	$P(b)$	$P(a) \vee P(b)$	$(P(a) \vee P(b)) \rightarrow P(a)$
0	0	0	1
0	1	1	0
1	0	1	1
1	1	1	1

nos muestra que no es una sentencia válida (tautología).

La transformación únicamente puede justificarse considerando que la constante que se introduce en lugar de la variable cuantificada por «$\exists$» no es como las demás constantes que puedan figurar en la sentencia: no es un valor concreto de entre los que pueda tomar x, sino un valor arbitrario. Informalmente, el razonamiento es éste: «la

sentencia dice que existe un individuo (al menos) con cierta propiedad; inventemos un nombre para ese individuo, y sigamos adelante». Por ejemplo, la sentencia anterior,

$$((\exists x)(V(x) \wedge M(e, x))$$

podría ser una formalización de «existe una persona, x, tal que x es varón ($V(x)$) y la madre de x es Eva ($M(e, x)$). Al sustituirla por

$$V(a) \wedge M(e, a)$$

hemos creado un nombre, a, para esa persona. Si la sentencia fuese más larga e incluyese otras constantes, este nombre que introducimos no podría coincidir con ninguna de ellas. Se trata de una «falsa constante», porque no es un valor *concreto*, sino un valor *arbitrario*, y la dificultad teórica estriba en que el sistema axiomático que tenemos no contempla este tipo de constantes, que se llaman *«constantes de Skolem»*.

b) Si la sentencia abarcada por el cuantificador existencial a eliminar está dentro del alcance de un cuantificador universal, no se puede, como antes, sustituir la variable cuantificada por una constante de Skolem. Para ver por qué es así, consideremos un sencillo ejemplo:

«para todo x, si x es humano, existe una y tal que y es la madre de x»

La sentencia podría ser:

$$(\forall x)(H(x) \rightarrow (\exists y)(M(y, x)))$$

Si sustituimos y por una constante de Skolem, a, tendríamos:

$$(\forall x)(H(x) \rightarrow M(a, x))$$

que, literalmente, diría: «para todo x, si x es humano, entonces la madre de x es a», lo cual es falso («a» sería la madre de todos los humanos). Lo que ocurre ahora es que el valor de la variable (y) cuantificada por «$\exists$» depende del que tenga x, y, por tanto, en lugar de sustituirla por una constante la sustituiremos por una *«función de Skolem»*, $f(x)$. Este símbolo de función no puede coincidir con ningún otro que figurase ya en la sentencia. En el ejemplo, la sentencia quedaría así:

$$(\forall x)(H(x) \rightarrow M(f(x), x))$$

(todo humano, x, tiene una madre, $f(x)$).

5. Eliminación de los cuantificadores universales. Contra lo que pudiera pensarse, este paso no plantea ningún problema. En este momento, si se han seguido los pasos anteriores, sólo existen cuantificadores universales, y cada uno de ellos afecta a una variable diferente. Por tanto, podemos escribirlos todos al comienzo de la sentencia. Y, puesto que todas las variables están universalmente cuantificadas (como ya hemos dicho, sólo consideramos sentencias cerradas) podemos omitir la escritura de tales

cuantificadores. No se trata, pues, en rigor, de «eliminarlos» sino de suponer que siempre existen.

Para centrar ideas sobre los pasos seguidos hasta ahora, supongamos que partimos de la sentencia

$$[(\forall x)(P(x))] \rightarrow [(\forall x)(\forall y)(\exists z)(P(x, y, z) \rightarrow (\forall u)(R(x, y, z, u)))]$$

Eliminando los condicionales,

$$[\neg\, (\forall x)(P(x))] \vee [(\forall x)(\forall y)(\exists z)(\neg\, P(x, y, z) \vee (\forall u)(R(x, y, z, u)))]$$

Introduciendo negaciones (la única a introducir es la primera),

$$[(\exists x)(\neg\, P(x))] \vee [(\forall x)(\forall y)(\exists z)(\neg\, P(x, y, z) \vee (\forall u)(R(x, y, z, u)))]$$

Independizando las dos variables que tienen el nombre «x»,

$$[(\exists x)(\neg\, P(x))] \vee [(\forall w)(\forall y)(\exists z)(\neg\, P(w, y, z) \vee (\forall u)(R(w, y, z, u)))]$$

La variable x puede sustituirse por una constante de Skolem. Pero la z está dentro del alcance de dos cuantificadores universales: el de w y el de y. Por tanto, la sustituiremos por una función de Skolem, $f(w, y)$:

$$[\neg\, P(a)] \vee [(\forall w)(\forall y)(\neg\, P(w, y, f(w, y)) \vee (\forall u)(R(w, y, f(w, y), u)))]$$

Por último, ponemos en cabeza todos los cuantificadores universales:

$$(\forall w)(\forall y)(\forall u)\{[\neg\, P(a)] \vee [\neg\, P(w, y, f(w, y)) \vee R(w, y, f(w, y), u)]\}$$

Y considerando la asociatividad de la disyunción, y dando por implícitamente cuantificadas universalmente a todas las variables, escribiremos:

$$\neg\, P(a) \vee \neg\, P(w, y, f(w, f)) \vee R(w, y, f(w, y), u)$$

6. Distribución de «$\wedge$» sobre «$\vee$». En el último ejemplo, ya hemos llegado a una disyunción de literales, o sea, a una cláusula. Pero, en general, tras el último paso tendremos una sentencia formada por literales unidos por las conectivas «$\wedge$» y «$\vee$», y aplicaremos, igual que hacíamos en lógica de proposiciones, la propiedad distributiva

$$(L_1 \wedge L_2) \vee L_3 \equiv (L_1 \vee L_3) \wedge (L_2 \vee L_3)$$

para llegar a una conjunción de cláusulas.

7. Redenominación de variables para que cada cláusula tenga las suyas propias. Este paso tiene su justificación en la ley de distribución del cuantificador universal

sobre la conjunción. En efecto, si tras los pasos anteriores se ha llegado a una sentencia como

$$P(x) \wedge Q(x)$$

(dos cláusulas constituidas cada una por un literal), en realidad la sentencia es:

$$(\forall x)(P(x) \wedge Q(x))$$

y lo que hacemos es sustituirla por su equivalente

$$(\forall x)(P(x)) \wedge (\forall y)(P(y))$$

es decir, nos quedamos con las cláusulas $P(x)$ y $P(y)$.

Ejemplo:

Sigamos todos los pasos con la siguiente sentencia:

$$(\forall x)\{[(\exists y)(P(y) \rightarrow Q(x, y) \wedge R(y))] \wedge [\neg (\exists y)(P(f(y)) \rightarrow Q(x, y))]\}$$

1: $(\forall x)\{[(\exists y)(\neg P(y) \vee (Q(x, y) \wedge R(y)))] \wedge [\neg (\exists y)(\neg P(f(y)) \vee Q(x, y))]\}$

2: $(\forall x)\{[(\exists y)(\neg P(y) \vee (Q(x, y) \wedge R(y)))] \wedge [(\forall y)(P(f(y)) \wedge \neg Q(x, y))]\}$

3: $(\forall x)\{[(\exists y)(\neg P(y) \vee (Q(x, y) \wedge R(y)))] \wedge [(\forall z)(P(f(z)) \wedge \neg Q(x, z))]\}$

4: $(\forall x)\{[\neg P(g(x)) \vee (Q(x, g(x)) \wedge R(g(x)))] \wedge [(\forall z)(P(f(z)) \wedge \neg Q(x, z))]\}$

5: $[\neg P(g(x)) \vee (Q(x, g(x)) \wedge R(g(x)))] \wedge [P(f(z)) \wedge \neg Q(x, z)]$

6: $[\neg P(g(x)) \vee Q(x, g(x))] \wedge [\neg P(g(x)) \vee R(g(x))] \wedge [P(f(z))] \wedge [\neg Q(x, z)]$

7: Resultan, finalmente, cuatro cláusulas:

$\neg P(g(x_1)) \vee Q(x_1, g(x_1))$
$\neg P(g(x_2)) \vee R(g(x_2))$
$P(f(z_1))$
$\neg Q(x_3, z_2)$

Las cláusulas se pueden expresar también como sentencias condicionales, igual que veíamos en el capítulo 2: el antecedente es la conjunción de los literales negativos, hechos positivos, y el consecuente es la disyunción de los literales positivos. Y, lo mismo que allí, se llaman cláusulas de Horn a las que sólo tienen un literal positivo («cabeza») o ninguno, y cláusula vacía, λ, a aquella en la que han desaparecido todos los literales.

Con las sentencias en forma clausulada, la idea de la resolución es la misma que en lógica de proposiciones: dadas dos generatrices que comparten un literal positivo en la una y negativo en la otra, obtener la resolvente eliminando ese literal y conservando los demás. Pero ahora hay un detalle nuevo a considerar: como los literales dependen

de argumentos, habrá que igualar los argumentos de los literales que se eliminan. Por ejemplo, consideremos la inferencia de modus ponens:

$$\frac{(\forall x)(H(x) \rightarrow M(x)) \quad H(s)}{M(s)}$$

Poniendo las sentencias en forma clausulada,

$$\frac{\neg H(x) \vee M(x) \quad H(s)}{M(s)}$$

Si eliminamos $H(s)$ es porque podemos sustituir x por s en la primera premisa, y así *«unificar»* los dos literales. El proceso de unificación no es siempre tan fácil como en este ejemplo, por lo que vamos a estudiarlo en general.

4.3. Sustitución y unificación

Ya hemos definido la sustitución en el apartado 2.5.2. Pero ahora, por una parte, no tenemos variables proposicionales, y, por otra, sólo nos interesa sustituir variables por términos, y sólo en los literales, no en las sentencias en general. Nos quedaremos, pues, con una definición restringida de sustitución:

Definición 4.3.1. Dadas unas variables $x_1, x_2, \ldots, x_n$ y unos términos $t_1, t_2, \ldots, t_n$ en los que no figuran esas variables, llamaremos ahora *sustitución* a un conjunto de pares ordenados

$$s = \{t_1/x_1, t_2/x_2, \ldots, t_n/x_n\}$$

Definición 4.3.2. Dados un literal L que contiene las variables $x_1, x_2, \ldots, x_n$, y una sustitución s (cuyos términos no pueden contener símbolos constantes ni de función que ya estén en L), la *operación de sustitución* consiste en poner t_i en todos los lugares de L donde aparezca x_i, y ello para todos los pares t_i/x_i de s. El resultado, que se representa por Ls, es un *caso de sustitución en* L.

Por ejemplo, sean:

$$\begin{aligned} L &= P(a, x, f(y)) \\ s_1 &= \{b/x, c/y\} \\ s_2 &= \{b/x, g(z)/y\} \\ s_3 &= \{z/x, w/y\} \end{aligned}$$

Los tres casos respectivos de sustitución en L serán:

$$Ls_1 = P(a, b, f(c))$$
$$Ls_2 = P(a, b, f(g(z)))$$
$$Ls_3 = P(a, z, f(w))$$

Ls_1 es un *caso terminal de L*: llamaremos así a los que no contienen variables. Ls_3 es una *variante alfabética de L*: sólo se han cambiado los nombres de las variables por otros.

Definición 4.3.3. Dadas dos sustituciones, s_1 y s_2, su *composición*, s_1s_2, es una sustitución tal que $Ls_1s_2 = (Ls_1)s_2$.

La composición de sustituciones es asociativa ($(s_1s_2)s_3 = s_1(s_2s_3)$) pero, en general, no es conmutativa ($s_1s_2 \neq s_2s_1$). La razón es que s_1 y s_2 pueden incluir variables idénticas que se sustituyen por términos diferentes, por lo que el resultado de la composición dependerá de cuál se aplique primero. Por la misma razón, no puede calcularse la composición uniendo simplemente los conjuntos s_1 y s_2. Para el cálculo de s_1s_2 hemos de aplicar primero s_2 a los términos de s_1 y después añadir los pares de s_2 cuyas variables no están entre las de s_1. Por ejemplo, la composición de

$$s_1 = \{f(x, y)/z\}$$

y

$$s_2 = \{a/x, b/y, c/z, d/u\}$$

sería:

$$s_1s_2 = \{f(a, b)/z, a/x, b/y, d/u\}$$

mientras que, si invertimos el orden,

$$s_2s_1 = \{a/x, b/y, c/z, d/u\} = s_2$$

(porque cuando fuésemos a aplicar s_1 ya habríamos aplicado antes s_2, con la que habría desaparecido la z, sustituida por c).

Definición 4.3.4. Diremos que un conjunto de literales $\{L_i\}$ ($i = 1, 2, \ldots, n$) es *unificable* si existe una sustitución s tal que $L_1s = L_2s = \ldots = L_ns$. En tal caso, diremos que s es un *unificador* de $\{L_i\}$ y que los literales L_i *se unifican en* L_is.

Por ejemplo, sea $\{L_i\} = \{P(a, x, f(y)), P(a, x, f(b))\}$. Un unificador sería $s = \{c/x, y/b\}$, con el que ambos literales se unifican en $P(a, c, f(b))$. Pero no es el único: hay otro unificador más pequeño, $\mu = \{y/b\}$, con el que se unifican en $P(a, x, f(b))$.

Teorema 4.3.5. Si $\{L_i\}$ es unificable, entonces existe un *unificador más general*, o *unificador mínimo*, μ, que tiene dos propiedades:

a) Si s es otro unificador de $\{L_i\}$, entonces existe una sustitución s' tal que $s = \mu s'$, es decir, $L_i s$ es un caso de $L_i\mu$. (En el ejemplo anterior, $P(a, c, f(b))$ es un caso de $P(a, x, f(b))$ para $s' = \{c/x\}$).
b) $L_i\mu$ es único salvo por variantes alfabéticas.

No entraremos en la demostración de este teorema, que es algo laboriosa, y que es constructiva: se llega a un algoritmo para encontrar el unificador más general de cualquier conjunto de literales que sea unificable.

4.4. La regla de resolución

Sean dos generatrices,

$$G_1 = L_{11} \vee L_{12} \vee \ldots$$
$$G_2 = L_{21} \vee L_{22} \vee \ldots$$

o, expresadas como conjuntos de literales,

$$G_1 = \{L_{1i}\}$$
$$G_2 = \{L_{2i}\}$$

Y sean $\{l_{1i}\} \subset \{L_{1i}\}$ y $\{l_{2i}\} \subset \{L_{2i}\}$ tales que $\{l_{1i}\} \cup \{\neg\, l_{2i}\}$ es unificable, siendo μ el unificador mínimo. Entonces decimos que $\{L_{1i}\}$ y $\{L_{2i}\}$ *se resuelven en* l_{1i} y que *de ambas generatrices se infiere la resolvente*:

$$[\{L_{1i}\} - \{l_{1i}\}]\mu \cup [\{L_{2i}\} - \{l_{2i}\}]\mu$$

La justificación teórica de esta regla de inferencia es la generalización de las leyes de inferencia de la alternativa que veíamos en el apartado 2.5.5. Para comparar aquella ley con lo que acabamos de decir, obsérvese que, al unificar, $\{l_{1i}\}$ se reduce a un solo literal, que corresponde al que allí llamábamos $P(x_1, \ldots, x_n)$, y $\{l_{2i}\}$ a otro, $\neg\, P(x_1, \ldots, x_n)$.

Ejemplo:

Sean,

$$G_1 = P(a, x, f(a)) \vee \neg\, Q(x)$$
$$G_2 = \neg\, P(a, y, f(a)) \vee \neg\, P(a, y, f(z)) \vee Q(z)$$

Podemos elegir $\{l_{1i}\}$ y $\{l_{2i}\}$ de varios modos, y con cada uno obtendremos una resolvente distinta:

a) $\{l_{1i}\} = \{P(a, x, f(a))\}$; $\{l_{2i}\} = \{\neg\, P(a, y, f(a))\}$
El unificador mínimo es $\mu = \{x/y\}$, y la resolvente:
$R = \neg\, Q(x) \vee \neg\, P(a, x, f(z)) \vee Q(z)$

b) $\{l_{1i}\} = \{P(a, x, f(a))\}$; $\{l_{2i}\} = \{\neg P(a, y, f(z))\}$
$\mu = \{x/y, a/z\}$
$R = \neg Q(x) \vee \neg P(a, x, f(a)) \vee Q(a)$

c) $\{l_{1i}\} = \{P(a, x, f(a))\}$; $\{l_{2i}\} = \{\neg P(a, y, f(a)), \neg P(a, y, f(z))\}$
$\mu = \{x/y, a/z\}$
$R = \neg Q(x) \vee Q(a)$

Las tres anteriores son resoluciones en P. También podemos hacer una resolución en Q:

d) $\{l_{1i}\} = \{\neg Q(x)\}$; $\{l_{2i}\} = \{Q(z)\}$
$\mu = \{x/z\}$
$R = P(a, x, f(a)) \vee \neg P(a, y, f(a)) \vee \neg P(a, y, f(x))$

Puede demostrarse que la resolución es consistente: toda conclusión que se infiera también se deduce (o sea, se satisface para todas las interpretaciones que satisfacen a las premisas).

4.5. Refutación

Aquí es totalmente aplicable todo lo dicho en el apartado 5.8 del capítulo 2, porque la ley de reducción al absurdo se aplica también en lógica de predicados. Es decir, si

$$P = P_1 \wedge P_2 \wedge \ldots \wedge P_n$$

es una sentencia con todas sus variables cuantificadas universalmente, igual que C, $P \rightarrow C$ será verdadera y, por tanto, C será una conclusión de $P_1, P_2, \ldots, P_n$, si (y sólo si) de la conjunción de P y $\neg C$ resulta una contradicción. Con la resolución, la contradicción se manifestará por la obtención de la cláusula vacía.

Se puede demostrar que la refutación con resolución y con búsqueda exhaustiva (Definición 5.7.2 del capítulo 2) es un sistema inferencial consistente (si se obtiene la cláusula vacía, C se deduce de P) y completo (si C se deduce de P siempre se obtiene la cláusula vacía). El problema es que la búsqueda exhaustiva exige ahora comprobar para todas las unificaciones posibles entre todas las cláusulas, y en problemas cuyo tamaño (que depende del número de premisas, de cláusulas en cada premisa y de literales en cada cláusula) empiece a ser elevado, se produce el fenómeno llamado «explosión combinatoria»: el número de combinaciones a explorar se hace tan elevado que las máquinas actuales no pueden llegar a la solución en un tiempo aceptable. Por ello, se utilizan otras estrategias para el sistema inferencial, y no todas ellas son completas.

Ejemplo:

Volvamos al Ejemplo 1.7.4, y, para simplificar, supongamos que con el predicado $P(x, y)$ representamos «x es el padre o la madre de y». Tendremos entonces dos hechos y la definición de «abuelo de»:

$$P1\colon P(a, l)$$
$$P2\colon P(j, a)$$
$$P3\colon (\forall x)(\forall y)(\forall z)(P(x, y) \wedge P(y, z) \rightarrow A(x, z))$$

La conclusión evidente en este caso es $A(j, l)$. Pero normalmente tendremos muchos más hechos (y muchas más definiciones), y podríamos plantearnos la pregunta: «dados los hechos y la definición de abuelo, ¿existen abuelos?». Vamos a ver que la refutación no sólo nos permite responder a la pregunta, sino, si la respuesta es afirmativa, averiguar los individuos que están en la relación.

La pregunta que acabamos de hacer se formalizaría así:

$$C\colon (\exists x)(\exists y)(A(x, y))$$

Si la respuesta es positiva entonces se podrá refutar $\neg C$. Pongamos primero $P3$ y $\neg C$ en forma clausulada:

$$P3 \ : \neg P(x, y) \vee \neg P(y, z) \vee A(x, z)$$
$$\neg C\colon \neg A(u, v)$$

Resolviendo en A estas dos cláusulas mediante $\mu = \{x/u, z/v\}$,

$$C1\colon \neg P(x, y) \vee \neg P(y, z)$$

Resolviendo en P, $P1$ con $C1$ mediante $\mu = \{a/y, l/z\}$,

$$C2\colon \neg P(x, a)$$

Y resolviendo finalmente $P2$ con $C2$ mediante $\mu = \{j/x\}$,

$$C3\colon \lambda$$

Pero no solamente hemos obtenido la refutación, sino también los valores de las variables mediante los cuales se refuta. Efectivamente, las dos variables iniciales de la pregunta, u y v, las sustituimos por x y z, y posteriormente z se sustituyó por l y x por j. Luego l y j son los dos individuos que están en la relación.

Ahora bien, supongamos que nuestro procedimiento es tal que al resolver $P1$ con $C1$ escoge el primer literal de $P1$ en lugar del segundo. En tal caso, haría la unificación con $\mu = \{a/x, l/y\}$ y obtendría

$$C2\colon \neg P(l, z)$$

y si el procedimiento (no de búsqueda exhaustiva) ya no vuelve a considerar $C1$, entonces habrá llegado a un punto muerto, puesto que no puede hacer más resoluciones.

5. RESUMEN

La lógica de predicados nos permite entrar en la composición de los enunciados, que en lógica de proposiciones se representan con simples variables proposicionales. Ahora tendremos, en lugar de variables proposicionales, predicados, que representan propiedades de individuos o relaciones entre individuos, y constantes y variables, que representan a esos individuos.

Hemos seguido un camino paralelo al del capítulo 2, ampliando a la lógica de predicados el sistema axiomático, la semántica y los sistemas inferenciales, deteniéndonos especialmente en la regla de resolución, cuya aplicación es algo más complicada aquí, primero por la existencia de variables cuantificadas, que exige ciertas manipulaciones sobre las sentencias para expresarlas en forma clausulada, y luego porque en el emparejamiento de literales complementarios hay que buscar unificaciones.

El último ejemplo, que se complementa más adelante con el Ejercicio 7.4, permite vislumbrar cómo el método de resolución puede servir para búsquedas en bases de datos construidas sobre la declaración de hechos elementales y de definiciones de relaciones.

6. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

Robinson (1965) propuso el método de resolución junto con un algoritmo de unificación. Anteriormente, Davis y Putnam (1960) habían presentado un procedimiento para la expresión de sentencias en forma clausulada. Y mucho antes, el matemático Thoralf Skolem (1928) había propuesto el uso de las funciones que llevan su nombre.

La que hemos llamado «búsqueda exhaustiva» es la estrategia de control más rudimentaria para sistemas inferenciales basados en la regla de resolución (en la terminología de la inteligencia artificial, es una búsqueda del tipo «breadth first»). Se han propuesto muchas otras estrategias, algunas no completas, en aras de una mayor eficacia. Una buena síntesis puede encontrarse en Nilsson (1982, cap. 5).

Aunque en este capítulo hayamos puesto un énfasis especial en los métodos basados en la regla de resolución, no se debe concluir por ello que éste sea el único sistema inferencial programable. De hecho, hay situaciones en las que la forma clausulada de la lógica no es la más apropiada, y se prefiere un sistema inferencial que opere sobre sentencias expresadas en la forma «estándar».

Aparte de las aplicaciones para el diseño de sistemas basados en conocimiento, de las que hablaremos en el capítulo 6, la lógica formal se utiliza también como base para la definición de lenguajes de especificación de programas. Este es un tema de gran importancia en ingeniería del software: la programación, tradicionalmente, ha tenido siempre más de «arte» que de «ciencia»; ahora se trata de elaborar teorías y métodos

formales que permitan la mejora de la productividad y la fiabilidad del software. Un método que se está utilizando cada vez más en Europa es el VDM (Vienna Development Method), y una referencia aconsejable sobre este tema es el libro de Jones (1986).

Hay muchos libros sobre lógica recomendables para ampliar lo expuesto en este capítulo. Destacamos el de Robinson (1979), el de Kowalski (1979) y el de Cuena (1986).

7. EJERCICIOS

7.1. Expresar como sentencias en lógica de predicados los siguientes enunciados:

a) «Existen individuos que, sin ser completamente idiotas, se comportan como tales».
b) «Todo lo que no es tradición es plagio». (Eugenio D'Ors).
c) «Diremos que una máquina es inteligente para un tipo de tareas si un observador es incapaz de distinguir por el resultado a la máquina de un humano que sabe resolver ese tipo de tareas». (Esto es lo que se llama «test de Turing»).
d) «Un descendiente de una persona es o bien un hijo de esa persona o bien un hijo de un descendiente».

7.2. Considerar la premisa:

P: «En un pueblo hay un barbero que afeita a todas las personas del pueblo que no se afeitan a sí mismas, y sólo a ellas».

¿Se puede inferior la conclusión *C*: «El barbero se afeita a sí mismo»? Si esta conclusión fuese verdadera, la premisa nos dice que tendría que ser falsa (sólo afeita a los que *no* se afeitan a sí mismos), y si fuese falsa, la premisa nos dice que tendría que ser verdadera (afeita a *todos* los que no se afeitan a sí mismos). Esta es la llamada «paradoja del barbero», debida a Bertrand Russell. Descubrir el origen de la paradoja mediante el análisis formal de la premisa.

7.3. Formalizar los siguientes razonamientos y comprobar las conclusiones mediante resolución y refutación:

a) *P*1: Ningún ordenador se equivoca.
*P*2: El que tiene boca se equivoca.
C : Ningún ordenador tiene boca.

b) *P*1: Algunos ordenadores se equivocan.
*P*2: El que tiene boca se equivoca.
*P*3: Algunos ordenadores tienen boca.

c) *P*1: Todos los libros de informática son instructivos.
*P*2: Ninguna novela es un libro de informática.
C : Ninguna novela es instructiva.

d) *P*1: Todos los informáticos saben programar.
*P*2: Los humanistas no saben programar.
*P*3: Algunos humanistas saben leer programas.
C : Algunos que saben leer programas no son informáticos.

e) P : No existe nadie que sea al mismo tiempo maestro de alguien y alumno de ese mismo alguien.
$C1$: Nadie es maestro de sí mismo.
$C2$: Nadie es alumno de sí mismo.

f) $P1$: Todos los animales que no cocean son flemáticos.
$P2$: Los asnos no tienen cuernos.
$P3$: Un búfalo siempre puede lanzarlo a uno contra una puerta.
$P4$: Ningún animal que cocea es fácil de engullir.
$P5$: Ningún animal sin cuernos puede lanzarlo a uno contra una puerta.
$P6$: Todos los animales son excitables, excepto los búfalos.
C : Los asnos no son fáciles de engullir.

g) $P1$: Los animales se irritan siempre mortalmente si no les presto atención.
$P2$: Los únicos animales que me pertenecen están en ese prado.
$P3$: Ningún animal puede adivinar un acertijo a menos que haya sido adecuadamente instruido en un colegio con internado.
$P4$: Ningún animal de los que están en este prado es un tejón.
$P5$: Cuando un animal está mortalmente irritado corre de un lado para otro salvajemente y gruñe.
$P6$: Nunca presto atención a un animal, a no ser que me pertenezca.
$P7$: Ningún animal que haya sido adecuadamente instruido en un colegio con internado corre de un lado para otro salvajemente y gruñe.
C : Ningún tejón puede adivinar un acertijo.

(Estos dos últimos razonamientos proceden de Lewis Carroll. En el libro de Deaño vienen las correspondientes inferencias mediante un «sistema de deducción natural», que pueden compararse con las realizadas mediante resolución; para el c), Deaño utiliza 21 pasos de aplicación de las reglas de inferencia, y para el d), 24).

7.4. Completar la definición recursiva de «antepasado de» que dábamos en el Ejemplo 1.7.5 para que incluya separadamente las relaciones «padre de» y «madre de». Añadir varios hechos (relaciones «padre de» y «madre de» definidas sobre individuos concretos) imaginarios o reales (tomados, por ejemplo, de una familia real, en cualquiera de las acepciones de «real») de manera que existan varios individuos que estén en la relación «antepasado de». Ver entonces cómo mediante refutación podemos encontrar estas parejas de individuos.

Capítulo 5
OTRAS LOGICAS

1. INTRODUCCIÓN

La lógica nos permite modelar los procesos del razonamiento. Pero, como todo modelo, no es más que una aproximación a la realidad que se concentra en determinados aspectos relevantes de esa realidad y que no contempla infinidad de matices y detalles. Las lógicas llamadas «no clásicas» son las que abordan alguno de los aspectos no modelados en la lógica de proposiciones ni en la lógica de predicados. Y son de utilidad en informática cuando tales aspectos juegan un papel importante en el fenómeno o problema a estudiar o en la aplicación a desarrollar.

Dentro de estas «otras lógicas» pueden considerarse dos tipos: uno, el de las que amplían los lenguajes de los cálculos de proposiciones y de predicados y en las que todas las construcciones y teoremas de éstos siguen siendo válidos, y otro, el de las que invalidan algunas leyes. Por ejemplo, en las lógicas multivaloradas o polivalentes, que son aquellas en las que se admiten interpretaciones intermedias entre la «verdad» y la «falsedad», la ley del tercio excluso, $\vdash (A \vee \neg A)$, deja de ser válida.

El campo es demasiado amplio como para pretender darle aquí un tratamiento al mismo nivel formal y con la misma extensión que hemos hecho con las lógicas de proposiciones y de predicados. Por ello, nos limitaremos a una exposición informal y resumida de las lógicas más conocidas, apuntando solamente sus aplicaciones en informática y extendiéndonos finalmente con un poco más de detalle en una de ellas, la lógica borrosa. Remitimos al lector interesado en profundizar sobre alguna de estas lógicas, a la bibliografía comentada en el apartado 6.

2. AMPLIACIONES DE LA LÓGICA DE PREDICADOS

2.1. Lógica de predicados de orden superior

Tras la lectura del capítulo anterior, el lector que haya reparado en su título albergará, seguramente, una duda: ¿por qué «lógica de predicados *de primer orden*»? La respuesta es, naturalmente, que hay lógicas de predicados de segundo orden, de tercero, etc.

En la lógica de predicados de segundo orden los predicados pueden cuantificarse y utilizarse como variables. En el lenguaje cotidiano solemos hacerlo. Por ejemplo, cuando decimos «todos los españoles comparten algún rasgo común», que habríamos de formalizar así:

$$(\exists R)((\forall x)(E(x) \rightarrow R(x)))$$

(«existe al menos un rasgo, R, tal que, para todo x, si x es español, x tiene el rasgo R»). O cuando decimos «el cobre es conductor, propiedad que es muy interesante», que formalizaríamos así:

$$C(c) \wedge I(C)$$

Los predicados que se refieren a propiedades (I, en el último ejemplo) pueden también cuantificarse, y así entramos en la lógica de predicados de tercer orden, y así sucesivamente.

Las reglas de formación dadas en el capítulo 4 pueden ampliarse fácilmente para que permitan obtener sentencias de la lógica de predicados de orden superior. No ocurre lo mismo con el sistema axiomático. El problema es que muchos sistemas que se han propuesto son inconsistentes y conducen a paradojas. (Por ejemplo, la paradoja de la sentencia «esta sentencia es falsa»: si la sentencia es verdadera, hay que concluir que es falsa, y viceversa). Para resolverlo, se han elaborado esquemas teóricos, como las «teorías de los tipos», que en informática encuentran aplicaciones en la definición semántica de lenguajes de programación, aunque aquí no entraremos en ese campo.

2.2. Lógica de clases y lógica de relaciones

En realidad, las lógicas de clases y de relaciones no añaden nuevos aspectos a la lógica de predicados: representan otra manera de expresarla, con construcciones sintácticas más cercanas a la teoría de conjuntos.

Una *clase* es una entidad abstracta que designa a todos los individuos que comparten alguna propiedad. Conceptualmente, «clase» y «conjunto» son dos cosas diferentes: no es lo mismo la clase de los libros de informática, algo abstracto que designa a todos los objetos que comparten las propiedades de ser libros y de tratar sobre informática, del conjunto de todos los libros de informática, que es algo concreto. Pero las mismas operaciones que se definen entre conjuntos (unión,

intersección, etc.) son también válidas entre clases. Y consiguientemente, el álgebra de Boole, que es un modelo para la teoría de conjuntos, también lo es para la lógica de clases.

El alfabeto de la lógica de clases incluye los símbolos para representar a las clases (A, B, C, ...), los símbolos de la teoría de conjuntos ($\cup$, $\cap$, $\subset$, $\in$, etc.) y las conectivas de la lógica clásica. Por ejemplo, el razonamiento «todos los hombres son mortales, Sócrates es un hombre, luego Sócrates es mortal» se formalizaría así en lógica de clases:

$$((H \subset M) \wedge (s \in H)) \rightarrow (s \in M)$$

Puesto que a toda propiedad corresponde una clase, y los predicados monádicos representan propiedades de individuos, todo lo que pueda representarse en lógica de predicados monádicos puede también representarse en lógica de clases y estudiarse con la herramienta del álgebra de Boole. Sugerimos, por ejemplo, al lector, que analice mediante diagramas de Venn los razonamientos utilizados como ejemplos en el capítulo anterior.

La equivalencia entre lógica de clases y lógica de predicados monádicos se ve claramente si consideramos que la clase C de los individuos que comparten la propiedad P puede definirse, utilizando la terminología de conjuntos, así:

$$C = \{x \mid P(x)\}$$

Así como la lógica de clases es equivalente a la lógica de predicados monádicos y su modelo matemático es el álgebra de clases, que es un álgebra de Boole, la lógica de relaciones es equivalente a la lógica de predicados poliádicos y su modelo matemático es el álgebra de relaciones, una extensión del álgebra de Boole elaborada por de Morgan y Pierce. Por ejemplo, la relación R entre padre e hijo será la que existe entre todos los pares ordenados (x, y) tales que el predicado $P(x, y)$ es cierto:

$$R = \{(x, y) \mid P(x, y)\}$$

En el álgebra de relaciones se definen la relación universal y la relación vacía y un conjunto de operaciones: complementación, unión, suma, diferencia, selección, proyección y productos de relaciones. Su interés informático radica en el hecho de constituir un modelo matemático de gran utilidad para el diseño de un tipo de bases de datos llamadas, precisamente, *relacionales*.

2.3. Lógica de predicados con identidad

Muchos enunciados establecen la relación de identidad entre individuos. Por ejemplo, en

«España es el más meridional de los países europeos»

establecemos una identidad entre «España» y «el más meridional de los países europeos». Este uso del verbo «ser» para denotar la identidad es muy distinto de otros

que hemos visto antes, como el «ser» de la predicación («España es un país europeo»), el «ser» de la pertenencia («España pertenece a la clase de los países europeos») o el «ser» de la inclusión («la clase de los españoles está incluida en la clase de los europeos»).

En principio, la identidad no es más que un predicado diádico como cualquier otro. Si los lógicos le dan una importancia especial es porque muchos razonamientos sólo pueden explicarse ampliando el lenguaje de la lógica de predicados para que lo considere como un caso especial. Por ejemplo:

*P*1: El que inventó la palabra «telemática» es abulense
*P*2: El que inventó la palabra «telemática» ha escrito «La vida en un chip»
*P*3: El que ha escrito «La vida en un chip» es Luis Arroyo

C : Luis Arroyo es abulense

Obsérvese que en *P*1 y en *C* se utiliza el «ser» de la predicación (o de la pertenencia, según se adopte la óptica de la lógica de predicados o de la lógica de clases), mientras que en *P*2 y *P*3 es el «ser» de la identidad.

Tradicionalmente, en lógica se definen las *«descripciones»* para tratar con enunciados de ese tipo, y se introducen un nuevo símbolo, =, para la relación de igualdad, y un nuevo cuantificador, ι (iota), para formalizar expresiones del tipo «el x tal que...», de modo que el razonamiento anterior se formaliza así:

$$\begin{array}{l} P1\text{: } A((\iota x)(I(x, t))) \\ P2\text{: } (\iota x)(I(x, t)) = (\iota x)(E(x, v)) \\ \underline{P3\text{: } (\iota x)(E(x, v)) = a} \\ C\text{ : } A(a) \end{array}$$

Y una ampliación del sistema axiomático permite analizar razonamientos de ese tipo.

Pero tenemos otro recurso expresivo para formalizar ese tipo de razonamientos: las funciones. Definiendo las funciones $f_i(y)$ y $f_e(y)$ en sustitución de «el x tal que x inventó y» y «el x tal que x ha escrito y», tendremos:

$$\begin{array}{l} P1\text{: } A(f_i(t)) \\ P2\text{: } f_i(t) = f_e(v) \\ \underline{P3\text{: } f_e(v) = a} \\ C\text{ : } A(a) \end{array}$$

Nuestro sistema inferencial sólo necesita de la adición de una regla mediante la cual pueda sustituirse cualquier término por otro que sea idéntico a él.

2.4. Lógica modal

Una variable proposicional p (o, en su caso, una fórmula atómica) expresa «es el caso que p», o «p es verdadera», mientras que $\neg p$ expresa «no es el caso que p», o «p

es falsa». La lógica modal se introdujo para poder dar cuenta de las llamadas «expresiones modales» o «modalidades», que son las que incluyen declaraciones del tipo «es necesario que», «es posible que», «es imposible que».

En lógica modal se definen dos nuevas conectivas unarias: una para la posibilidad, $\blacklozenge$, y otra para la necesidad, $\blacksquare$. «Es posible que p» se representa por «$\blacklozenge p$», y «es necesario que p», por «$\blacksquare p$». En realidad, bastaría con una sola de ellas para las tres modalidades enunciadas:

«es posible que p»: $\blacklozenge p$, o bien $\neg \blacksquare \neg p$
«es necesario que p»: $\blacksquare p$, o bien $\neg \blacklozenge \neg p$
«es imposible que p»: $\neg \blacklozenge p$, o bien $\blacksquare \neg p$

Mediante estos nuevos conceptos, los lógicos han formalizado de manera más convincente la relación de deducibilidad, que en lógica de proposiciones y de predicados se hace con un simple condicional. A diferencia de la «implicación material» ($A \rightarrow B$, recuérdese la discusión sobre el significado del condicional en el apartado 1.3 del capítulo 2), la «implicación estricta», $A \Rightarrow B$, se define así:

$$\neg \blacklozenge (A \wedge \neg B)$$

donde, por la misma definición de $\blacklozenge$, interviene la idea de posibilidad. El uso de esta noción requiere, a efectos semánticos, el concepto de «mundos posibles»: $\blacklozenge A$ será verdadera si A lo es en alguno de los mundos posibles, y $\blacksquare A$ será verdadera si A lo es en todos los mundos posibles.

En informática, la lógica modal tiene aplicaciones en la teoría de lenguajes de programación y en los sistemas inferenciales llamados «no monotónicos», que son aquellos en los que se obtienen conclusiones provisionales que pueden verse invalidadas a la luz de nuevas evidencias.

2.5. Lógica temporal

Considerada por algunos autores como un tipo de lógica modal, la lógica temporal es aquella en la que la función de interpretación depende del instante. Se introducen en ella dos predicados temporales:

$F(A)$: A será verdadera en algún instante futuro
$P(A)$: A fue verdadera en algún instante pasado

a partir de los cuales pueden definirse otros, por ejemplo:

$\neg F(\neg A)$: A será verdadera en todo instante futuro
$\neg P(\neg A)$: A fue verdadera en todo instante pasado

y un predicado de precedencia temporal, $T(t_1, t_2)$, que será verdadero si $t_1 < t_2$ y falso

en caso contrario, y con el cual se puede extender la función de interpretación a los nuevos predicados:

$I(t, F(A)) = 1$ si y sólo si hay un t' tal que $T(t, t')$ e $I(t', A) = 1$
$I(t, P(A)) = 1$ si y sólo si hay un t' tal que $T(t', t)$ e $I(t', A) = 1$

En informática, la lógica temporal tiene aplicaciones para los trabajos sobre especificación y verificación de programas, así como para todos aquellos en los que interviene la concurrencia y la intercomunicación entre procesos.

3. LÓGICAS MULTIVALORADAS

La idea que motiva a todas las lógicas multivaloradas, también llamadas multivalentes o polivalentes, radica en el hecho de que a veces somos incapaces de asignar valores de «verdad» o «falsedad» absolutos a las sentencias. Ahora bien, esto es algo que ya habíamos previsto en nuestra definición de «interpretación» (definición 3.1.2 del capítulo 2): decíamos que el conjunto de valores semánticos, V, debe tener, como mínimo, dos valores. Hasta ahora, hemos considerado que $\text{card}(V) = 2$. Las lógicas multivaloradas serán aquellas en las que $\text{card}(V) > 2$.

Consideremos el caso más sencillo: $\text{card}(V) = 3$. Podemos adoptar en este caso, por ejemplo, los símbolos «1» para «verdadero», «0» para «falso» y «½» para «ni verdadero ni falso». Esta es, por cierto, una *interpretación semántica*; una *interpretación pragmática* sería: «1» para «sabemos que es verdadero», «0» para «sabemos que es falso» y «½» para «no sabemos si es verdadero o falso». Pues bien, en esta *lógica trivalorada* (o *trivalente*) la extensión del dominio de la función de interpretación para las sentencias puede hacerse teniendo en cuenta la siguiente tabla que define las conectivas en el conjunto V:

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$
0	0	1	0	0	1	1
0	½	1	0	½	1	½
0	1	1	0	1	1	0
½	0	½	0	½	½	½
½	½	½	½	½	1	1
½	1	½	½	1	1	½
1	0	0	0	1	0	0
1	½	0	½	1	½	½
1	1	0	1	1	1	1

(Como puede observarse, si se eliminan las filas en las que A o B se interpretan «½», queda la tabla binaria clásica).

Para la interpretación de sentencias con variables cuantificadas bastará tener en cuenta que «$\forall$» y «$\exists$» son generalizaciones de «$\wedge$» y «$\vee$» respectivamente:

$$\begin{aligned} I(\forall x)(A(x)) &= 1 \text{ si } (\forall x)(I(A(x)) = 1) \\ &= 0 \text{ si } (\exists x)(I(A(x)) = 0) \\ &= \tfrac{1}{2} \text{ en otro caso} \end{aligned}$$

$$\begin{aligned} I(\exists x)(A(x)) &= 1 \text{ si } (\exists x)(I(A(x)) = 1) \\ &= 0 \text{ si } (\forall x)(I(A(x)) = 0) \\ &= \tfrac{1}{2} \text{ en otro caso} \end{aligned}$$

Invitamos al lector a considerar la justificación de estos significados de las conectivas. Por ejemplo, el condicional puede analizarlo sobre el ejemplo «si corro, entonces me canso» que considerábamos en el capítulo 2 para ilustrar el sentido del condicional en el caso de interpretación binaria.

La generalización para cualquier V tal que $\text{card}(V) > 3$ puede hacerse mediante las siguientes definiciones (llamadas «leyes de Lukasiewicz»):

$$I(\neg A) = 1 - I(A)$$

$$I(A \wedge B) = \text{mín}(I(A), I(B))$$

$$I(A \vee B) = \text{máx}(I(A), I(B))$$

$$I(A \rightarrow B) = 1 \text{ si } I(A) \leq I(B)$$

$$I(A \rightarrow B) = 1 - I(A) + I(B) \text{ si } I(A) > I(B)$$

$$I((\forall x)(A(x))) = \min_x(I(A(x)))$$

$$I((\exists x)(A(x))) = \max_x(I(A(x)))$$

que, como puede comprobarse fácilmente, coinciden con las dadas por la tabla para el caso particular de la lógica trivalorada ($\text{card}(V) = 3$).

Si hacemos $V = \{x \mid 0 \leq x \leq 1\}$ tendremos una lógica con infinitos valores de interpretación, que puede ponerse en relación con la *lógica probabilitaria*, en la que las conectivas se corresponden con operaciones de la teoría de probabilidades. Para el operador de negación, esta correspondencia es inmediata; así, por ejemplo, si tiramos un dado, en el *lenguaje de las probabilidades* decimos: «el suceso 'sacar un tres' tiene probabilidad 1/6, y el suceso contrario tiene probabilidad $1 - 1/6 = 5/6$», mientras que en el *lenguaje de la lógica* diríamos: «la sentencia 'al tirar un dado sale un tres' tiene el valor de verdad 1/6, mientras que la sentencia 'al tirar un dado sale un número distinto de tres' tiene el valor de verdad $1 - 1/6 = 5/6$».

Hay un modelo algebraico para este tipo de lógicas, las álgebras de Post, en el que no vamos a entrar aquí. Sí nos parece oportuno detenernos con cierto detalle en un tipo especial de lógica infinitamente valorada que tiene muchas posibilidades de aplicación en el diseño de sistemas basados en conocimiento: la lógica borrosa.

4. LÓGICA BORROSA

4.1. Justificación

Existe actualmente mucho interés (tanto científico-técnico como económico) por los llamados sistemas expertos, y su porvenir parece asegurado, a la vista de los programas de investigación en informática japoneses, americanos y europeos. Por ello, dedicaremos el siguiente capítulo a la exposición de sus principios básicos. Como allí veremos, un problema fundamental que se presenta es el de representar en la máquina los conocimientos y los procedimientos inciertos e imprecisos que utilizan los expertos humanos para resolver problemas. En la mayoría de los sistemas expertos que se han construido o se están diseñando se adoptan técnicas «ad hoc» para atacar ese problema. Pero existen intentos teóricos para introducir en la lógica formal la imprecisión y la subjetividad propia de la actividad humana, y parece obvio que en el futuro se tienda a basar rigurosamente el diseño de los sistemas expertos en tales teorías. La más conocida de ellas es la lógica borrosa («fuzzy logic»). Las consideraciones anteriores, unidas a la escasez de bibliografía en español sobre el tema, nos han incitado a dedicar una atención especial a la lógica borrosa en este capítulo de «otras lógicas».

La lógica borrosa va aún más allá que las lógicas con infinitos valores de verdad. Porque ya no sólo se trata de considerar que hay una infinidad de valores semánticos entre «verdadero» y «falso», sino también de tener en cuenta que estos mismos valores de verdad son imprecisos. Por ejemplo, a la sentencia «este párrafo es de difícil comprensión» podría asignársele el valor de verdad 0.7 en lógica multivalorada. Pero generalmente hacemos inferencias imprecisas, como «si alguien encuentra muy difícil comprender un párrafo, casi con seguridad abandona la lectura; este párrafo es de difícil comprensión para tal persona, luego es probable que esa persona abandone la lectura». La lógica multivalorada no nos permite hacer inferencias como esa, porque intervienen en ella matices (relaciones entre «difícil» y «muy difícil», entre «casi con seguridad» y «es probable») imposibles de abordar con la simple extensión del conjunto de valores de verdad.

Ese tipo de inferencia imprecisa es el que pretende abordar la lógica borrosa. Y para ello parte de una reconsideración del mismo pilar básico de las matemáticas: el concepto de *conjunto*. En la realidad se presentan situaciones (particularmente, cuando aparecen consideraciones subjetivas) en las que resulta difícil determinar la pertenencia o no de un elemento a un conjunto. Por ejemplo:

* el conjunto de los números naturales mucho mayores que 100: parece claro que 101 no pertenece al conjunto, y que 10^{10} sí, pero ¿y 500?;
* el conjunto de las personas pobres: ¿pertenezco yo a ese conjunto?;
* el conjunto de las mujeres preciosas, etc.

Tales conjuntos pueden denominarse «borrosos» (o «difusos») para indicar que no existe un criterio que determine exactamente un límite entre pertenencia y no pertenencia al conjunto.

Pero, en este momento, el lector atento puede objetar: «lo que ocurre es que no se ha establecido el criterio para definir los conjuntos. Así, en el primer ejemplo, puede

decirse (por convenio, o por hipótesis de trabajo) que el límite está en 1.000; en el segundo, que es pobre toda persona que, sin tener patrimonio, perciba unos ingresos inferiores al salario mínimo; y en el último ejemplo ya es más difícil establecer un criterio, porque ¿a quién se le ocurre tratar de determinar matemáticamente tal conjunto?».

Ahora bien, si tratamos de formalizar las relaciones del hombre con su entorno siempre vamos a encontrarnos con elementos imprecisos o «borrosos», sobre todo en la actividad más típicamente humana, el lenguaje. Cuando alguien está aprendiendo a conducir, en un determinado momento puede recibir una orden del instructor como ésta: «levante ligera y lentamente el pie del embrague», pero nunca una como ésta: «levante el pie 8° a una velocidad de 2°30′ por segundo». Y, sin embargo, el hombre, considerado como sistema, se comporta bien (aprende) con entradas «borrosas» como las del primer caso, mientras sería difícil que lo hiciera con las del segundo tipo.

A poco que se reflexione, se llegará a la conclusión de que si pretendemos analizar sistemas muy complejos como el hombre, las sociedades, etc. (ya sea con espíritu puramente científico, ya sea con fines utilitarios: diseño de máquinas que puedan razonar, tomar decisiones, comprender el lenguaje natural, etc.), resulta imprescindible introducir en los modelos la imprecisión y la subjetividad propias de la actividad humana. Porque, además, en estos sistemas complejos se da el hecho de que «significación» y «precisión» son incompatibles. Esto quiere decir que los resultados muy precisos suelen tener poco significado, y lo que interesa más bien son resultados cualitativos.

Estas consideraciones conducen a una reformulación del concepto básico de conjunto, admitiendo grados de pertenencia de los elementos a los conjuntos. Puesto que la lógica borrosa se apoya en la teoría de conjuntos borrosos, será preciso que veamos previamente los elementos básicos de esa teoría.

4.2. Subconjuntos borrosos

En la teoría clásica, dado un elemento x de un universo U, y un subconjunto, $A \subset U$, hay dos posibilidades: $x \in A$ o $x \notin A$. Puede definirse una función característica de pertenencia, μ_A, tal que $\mu_A(x) = 1$ si $x \in A$ y $\mu_A(x) = 0$ si $x \notin A$. Es fácil demostrar que:

$$\mu_{\overline{A}}(x) = 1 - \mu_A(x)$$
$$\mu_{A \cap B}(x) = \mu_A(x) \cdot \mu_B(x)$$
$$\mu_{A \cup B}(x) = \mu_A(x) + \mu_B(x) \quad \text{(suma lógica)}$$

Definición 4.2.1. Dado un universo U, un *subconjunto borroso*, $\underset{\sim}{A}$, de U, es un conjunto de pares

$$\{(x|\mu_A(x))\}, \forall x \in U,$$

donde $\mu_A(x)$ es una función que toma sus valores en un conjunto M llamado *conjunto de pertenencia*.

Generalmente se toma $M = \{m|m \in [0, 1]\}$; si se hace $M = \{0, 1\}$, entonces $\underset{\sim}{A}$ se reduce a un subconjunto ordinario, de manera que la teoría clásica de conjuntos es un caso particular de la teoría de conjuntos borrosos.

Ejemplo 4.2.2. Si $U = \{1, 2, 3, \ldots, 10\}$, podemos definir el subconjunto «varios», $\underset{\sim}{V}$, como:

$$\underset{\sim}{V} = \{3|0{,}5;\ 4|0{,}8;\ 5|1;\ 6|1;\ 7|0{,}8;\ 8|0{,}5\}$$

La asignación de valores a $\mu_V(x)$ es totalmente subjetiva, de manera que otra persona daría, con toda probabilidad, otras cifras.

Ejemplo 4.2.3. Si $U = \{x|x \in [0{,}150]\}$ se interpreta como el conjunto de edades posibles de un ser humano, podrían definirse los subconjuntos borrosos $\underset{\sim}{J}$ (joven) y $\underset{\sim}{V}$ (viejo) así:

$$\underset{\sim}{J} = \{(x|1)_{0\leq x\leq 40};\ (x|(1 + (x - 40)^2/40)^{-1})_{x>40}\}$$
$$\underset{\sim}{V} = \{(x|0)_{0\leq x\leq 40};\ (x|(1 + 40/(x - 40)^2)^{-1})_{x>40}\}$$

Podemos tener una visión gráfica de estos conjuntos representando $\mu_J(x)$ y $\mu_V(x)$ (figura 5.1):

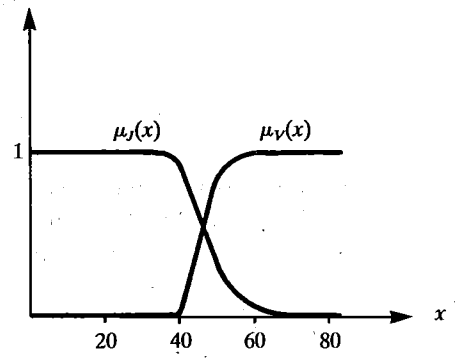


FIGURA 5.1.

Definición 4.2.4. Se definen las relaciones de igualdad e inclusión entre subconjuntos borrosos del siguiente modo:

$$\text{Igualdad: } \underset{\sim}{A} = \underset{\sim}{B} \text{ si } \mu_A(x) = \mu_B(x),\ \forall x \in U$$
$$\text{Inclusión: } \underset{\sim}{A} \subset \underset{\sim}{B} \text{ si } \mu_A(x) \leq \mu_B(x),\ \forall x \in U$$

Definición 4.2.5. Se definen las operaciones de complementación, intersección, unión, producto y potenciación del siguiente modo:

Complementación: $A = \overline{B}$ si $\mu_A(x) = 1 - \mu_B(x)$, $\forall x \in U$
(supuesto que $M = [0, 1]$)

Intersección: $\underset{\sim}{C} = \underset{\sim}{A} \cap \underset{\sim}{B}$ si $\mu_C(x) = \text{mín}(\mu_A(x), \mu_B(x))$, $\forall x \in U$

Unión: $\underset{\sim}{C} = \underset{\sim}{A} \cup \underset{\sim}{B}$ si $\mu_C(x) = \text{máx}(\mu_A(x), \mu_B(x))$, $\forall x \in U$

Producto: $C = \underset{\sim}{A}\,\underset{\sim}{B}$ si $\mu_C(x) = \mu_A(x) \cdot \mu_B(x)$, $\forall x \in U$

Potenciación: $C = \underset{\sim}{A}^{\alpha}$ si $\mu_C(x) = \mu_A^{\alpha}(x)$, $\forall x \in U$

Es fácil comprobar que las tres primeras se reducen a las definiciones clásicas en conjuntos ordinarios para $M = \{0, 1\}$. Asimismo, que todas las propiedades de estas operaciones en los conjuntos ordinarios (asociatividad, distributividad, etc.) se siguen cumpliendo, salvo en lo que respecta a dos muy importantes:

$$\underset{\sim}{A} \cap \overline{\underset{\sim}{A}} \neq \Phi$$

(el conjunto vacío, Φ, es aquel subconjunto borroso de U tal que $(\forall x \in U)$ $(\mu_\Phi(x) = 0)$), y

$$\underset{\sim}{A} \cup \overline{\underset{\sim}{A}} \neq U$$

Ejemplo 4.2.6. El complemento de «varios» tal como se definió más arriba sería:

$$\overline{\underset{\sim}{V}} = \{1|1;\ 2|1;\ 3|0{,}5;\ 4|0{,}2;\ 7|0{,}2;\ 8|0{,}5;\ 9|1;\ 10|1\}$$

Y las representaciones gráficas de $\underset{\sim}{V}$ y $\overline{\underset{\sim}{V}}$ serían las de la figura 5.2.

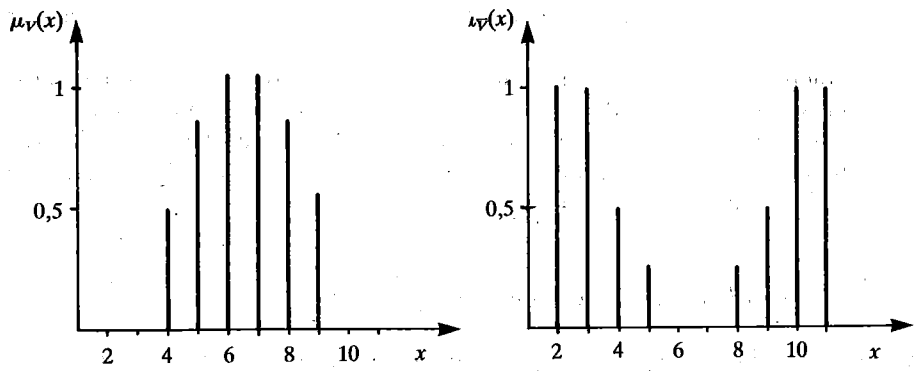


FIGURA 5.2.

Ejemplo 4.2.7. Con las definiciones dadas para «joven» y «viejo» en el Ejemplo 4.2.3, en la figura 5.3 pueden verse las representaciones gráficas de la intersección y la unión de ambos conjuntos.

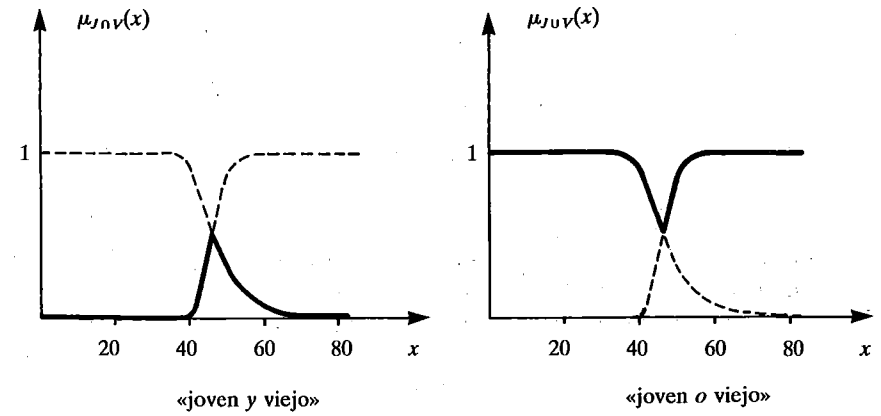


FIGURA 5.3.

En este ejemplo, tal como se definieron los subconjuntos, se cumple que $\underset{\sim}{V} = \overline{\underset{\sim}{J}}$, porque $(\forall x)(\mu_{J(x)} + \mu_{V(x)} = 1)$.

4.3. Relaciones borrosas

4.3.1. *Relaciones ordinarias entre conjuntos borrosos*

Más arriba hemos definido las relaciones de igualdad e inclusión entre conjuntos borrosos. Tales conjuntos son, realmente, subconjuntos borrosos de un universo (no borroso) U. Se trata, por tanto, de relaciones definidas sobre el conjunto de las partes (borrosas) de U, $P(U)$.

Ahora bien, aunque las partes o subconjuntos sean borrosos, las relaciones así definidas *no* son borrosas: dados $\underset{\sim}{A}$, $\underset{\sim}{B} \subset U$, o bien $(\underset{\sim}{A}, \underset{\sim}{B}) \in R$, o bien $(\underset{\sim}{A}, \underset{\sim}{B}) \notin R$; concretamente, para las relaciones R de igualdad e inclusión definidas, dados dos subconjuntos borrosos, $\underset{\sim}{A}$ y $\underset{\sim}{B}$, o bien $\underset{\sim}{A} = \underset{\sim}{B}$, o bien $\underset{\sim}{A} \neq \underset{\sim}{B}$, y, del mismo modo, o bien $\underset{\sim}{A} \subset \underset{\sim}{B}$, o bien $\underset{\sim}{A} \not\subset \underset{\sim}{B}$.

4.3.2. *Relaciones borrosas entre conjuntos ordinarios*

Vamos a definir ahora el concepto de *relación borrosa* entre conjuntos *no borrosos*. Sean A y B dos subconjuntos ordinarios, en general de universos diferentes: $A \subset U$, $B \subset V$, y sean $a \in A$ y $b \in B$ elementos genéricos de cada uno. Una relación

borrosa entre A y B se define como un conjunto de pares ordenados (a, b), cada uno con un determinado grado de pertenencia, μ_R, a la relación $\underset{\sim}{R}$:

$$\underset{\sim}{R} = \{(a, b)|\mu_R(a, b)\},\ a \in A,\ b \in B,\ 0 \leq \mu_R \leq 1 \qquad [1]$$

μ_R indica así en qué grado, o con qué intensidad, los elementos a y b están en la relación $\underset{\sim}{R}$.

Por ejemplo, existe una relación entre la estación del año en que nos encontramos y el calor o frío que se siente. Esta relación es subjetiva (y, por lo tanto, borrosa), y una determinada persona podría expresarla explícitamente así:

U = {estaciones}
V = {sensaciones}
$A = U$ = {primavera, verano, otoño, invierno}
B = {calor, frío}

$$\begin{array}{c} \\ A \end{array} \begin{array}{c} \overbrace{\qquad\qquad}^{B} \\ \begin{array}{cc} c & f \end{array} \\ \left\{\begin{array}{c} p \\ v \\ o \\ i \end{array}\right. \left[\begin{array}{cc} 0{,}7 & 0{,}4 \\ 1 & 0 \\ 0{,}6 & 0{,}5 \\ 0{,}1 & 1 \end{array}\right] \end{array}$$

Obsérvese que la relación propiamente dicha se escribe con mayor comodidad en forma matricial, en lugar de hacerlo así:

$$\underset{\sim}{R} = \{(p, c)|0{,}7;\ (p, f)|0{,}4;\ \text{etc.}\}$$

Las relaciones así definidas son binarias; la generalización a órdenes superiores (relaciones entre más de dos conjuntos) es inmediata. Así, en el ejemplo anterior un tercer conjunto podría ser el de países; su representación matricial se haría escribiendo varias matrices.

Siguiendo con la relación binaria definida por [1], se observa fácilmente que $\underset{\sim}{R}$ es, realmente, un subconjunto borroso del producto cartesiano de A y B:

$$\underset{\sim}{R} \subset A \times B$$

En efecto, como A y B son ordinarios, su producto cartesiano consta de todos los pares (a, b), y la relación borrosa le da un grado de pertenencia a cada uno.

En este caso binario la relación se suele escribir también así: $A\ \underset{\sim}{R}\ B$.

En general, una relación borrosa n-aria entre n conjuntos ordinarios, A_1, A_2, ..., A_n (no tienen por qué ser todos distintos) será un subconjunto borroso del producto cartesiano de todos ellos:

$$\underset{\sim}{R} \subset A_1 \times A_2 \times \ldots \times A_n$$

O, también,

$$\underset{\sim}{R} = \{(a_1, a_2, \ldots, a_n) | \mu_R(a_1, a_2, \ldots, a_n)\},\ a_1 \in A_1, \ldots, a_n \in A_n$$

4.3.3. *Relaciones borrosas entre conjuntos borrosos*

En lo sucesivo vamos a prescindir del símbolo «~» bajo los nombres de los conjuntos y de las relaciones, y, salvo que digamos lo contrario, supondremos que todos los conjuntos y todas las relaciones son borrosos (y, de todos modos, ya sabemos que un conjunto ordinario es un caso particular de conjunto borroso, y una relación ordinaria lo es de relación borrosa).

Sean $A \subset U$, $B \subset V$, subconjuntos borrosos de universos U y V, y sean $a \in A$, $b \in B$. Se define el *producto cartesiano* de A y B del siguiente modo:

$$A \times B \triangleq \{(a, b) | \text{mín}(\mu_A(a), \mu_B(b))\} \qquad [2]$$

(Es, pues, un subconjunto borroso de $U \times V$).

Cualquier subconjunto de $A \times B$ será una relación borrosa entre ambos: $ARB \subset A \times B$. Obsérvese que como todo subconjunto debe satisfacer a la relación de inclusión, el grado de pertenencia de (a, b) a tal subconjunto debe ser igual o inferior a $\text{mín}(\mu_A(a), \mu_B(b))$.

La generalización al caso n-ario es también inmediata.

4.3.4. *Composición de relaciones*

Dados tres conjuntos A, B, C, una relación binaria entre A y B:

$$AR_1B = \{(a, b) | \mu_{R_1}(a, b)\},$$

y otra entre B y C:

$$BR_2C = \{(b, c) | \mu_{R_2}(b, c)\},$$

se define la *relación compuesta* de R_1 y R_2 del siguiente modo:

$$A(R_1 \circ R_2)C = \{(a, c) | \max_b [\text{mín}(\mu_{R_1}(a, b), \mu_{R_2}(b, c))]\} \qquad [3]$$

Es decir, para cada pareja (a, c) se toma como grado de pertenencia a $R_1 \circ R_2$ el resultado de la siguiente operación:

— Para cada uno de todos los $b \in B$ se toma el mínimo de las funciones de pertenencia a las relaciones R_1 y R_2 de las parejas (a, b) y (b, c), respectivamente.

— El máximo de todos los mínimos anteriores es el valor de la función de pertenencia a $R_1 \circ R_2$ de la pareja (a, c).

Si A, B, C son finitos, R_1 y R_2 pueden escribirse en forma matricial, y puede comprobarse que la matriz correspondiente a $R_1 \circ R_2$ se calcula como el «producto máx-mín» de las matrices R_1 y R_2. El producto máx-mín de dos matrices es el producto ordinario, pero sustituyendo la operación «suma» por la operación «máximo» y el «producto» por el «mínimo».

Estas definiciones, como las que daremos luego, pueden parecer a primera vista arbitrarias, y hasta cierto punto lo son. No obstante, en su elección se han tenido en cuenta los siguientes criterios:

a) que sean consistentes con las definiciones paralelas de la teoría de conjuntos ordinarios, es decir, que ésta pueda considerarse como un caso particular de la teoría de conjuntos borrosos;

b) que los modelos basados en la teoría reflejen razonablemente bien la realidad; y

c) que las computaciones que se derivan de la utilización de los modelos sean sencillas y, por tanto, se ejecuten con rapidez; en particular, las operaciones de selección de máximo o mínimo requieren mucho menos tiempo que las de suma o producto.

Veamos un ejemplo para ilustrar la composición de relaciones. Sean A, B y R_1 las definidas en el ejemplo anterior (relación entre estaciones del año y sensaciones de frío o calor). Obsérvese que en ese caso particular tanto A como B se han definido como conjuntos ordinarios; en cualquier caso, R_1 es un subconjunto borroso de $A \times B$, independientemente de que A y B sean ordinarios o borrosos. Podríamos seguir con esos mismos conjuntos y definir otro, borroso o no, relacionado con B y ver la relación compuesta. Pero para aplicar tanto la definición [2] como la [3], vamos a redefinir A y B como conjuntos borrosos:

$$U = \{p, v, o, i\},$$

donde p = primavera; v = verano; o = otoño; i = invierno.

$$V = \{T_1, T_2\},$$

donde T_1 = temperatura superior a 20°C; T_2 = temperatura igual o inferior a 20°C (universos no borrosos).

$$A = \text{estaciones frías} = \{p|0{,}3;\ v|0{,}1;\ o|0{,}4;\ i|0{,}9\}$$

$$B = \text{sensación de frío} = \{T_1|0{,}2;\ T_2|0{,}8\}$$

El producto cartesiano de A y B, según [2], expresado en forma matricial será:

$$A \times B = \begin{array}{c} \\ p \\ v \\ o \\ i \end{array} \begin{array}{c} \begin{array}{cc} T_1 & T_2 \end{array} \\ \begin{bmatrix} 0{,}2 & 0{,}3 \\ 0{,}1 & 0{,}1 \\ 0{,}2 & 0{,}4 \\ 0{,}2 & 0{,}8 \end{bmatrix} \end{array}$$

Una relación entre A y B es, según hemos dicho, cualquier subconjunto de $A \times B$. Por ejemplo:

$$R_1 = \begin{array}{c} \\ p \\ v \\ o \\ i \end{array} \begin{array}{c} \begin{array}{cc} T_1 & T_2 \end{array} \\ \begin{bmatrix} 0{,}2 & 0{,}2 \\ 0{,}1 & 0 \\ 0{,}2 & 0{,}3 \\ 0 & 0{,}8 \end{bmatrix} \end{array}$$

Obsérvese que R_1 relaciona bastante bien la sensación de frío alta con las estaciones, pero no tanto la sensación de poco frío con las mismas (particularmente, el grado de pertenencia de (v, T_1) a R_1 debería ser mucho más alto del 0,1 que resulta).

La explicación es que, tal como se han definido A y B, R_1 relaciona la sensación de *frío* con las estaciones *frías*, pero no la sensación de poco frío con las estaciones poco frías. Para que así fuera, deberíamos definir una nueva relación haciendo uso de las operaciones de unión y complementación. Volveremos sobre este ejemplo en el apartado 4.4.2.

Consideremos ahora un nuevo subconjunto borroso, C, en el universo $W = \{\text{ropas de abrigo}\}$, establecido así:

$$C = \{b|0{,}1;\ t|0{,}5;\ a|0{,}9\}$$

donde b = bañador; t = traje; a = abrigo.

El producto cartesiano de B y C es:

$$B \times C = \begin{array}{c} \\ T_1 \\ T_2 \end{array} \begin{array}{c} \begin{array}{ccc} b & t & a \end{array} \\ \begin{bmatrix} 0{,}1 & 0{,}2 & 0{,}2 \\ 0{,}1 & 0{,}5 & 0{,}8 \end{bmatrix} \end{array}$$

Tomemos $R_2 = B \times C$ (podría ser cualquier otro subconjunto).

Definidas así AR_1B y BR_2C, A y C estarán en la relación $R_1 \circ R_2$, que se calcula, según [3], haciendo el producto máx-mín de las matrices de R_1 y R_2:

$$R_1 \circ R_2 = \begin{array}{c} \\ p \\ v \\ o \\ i \end{array} \begin{array}{c} \begin{array}{cc} T_1 & T_2 \end{array} \\ \begin{bmatrix} 0{,}2 & 0{,}2 \\ 0{,}1 & 0 \\ 0{,}2 & 0{,}3 \\ 0 & 0{,}8 \end{bmatrix} \end{array} \cdot \begin{array}{c} \\ T_1 \\ T_2 \end{array} \begin{array}{c} \begin{array}{ccc} b & t & a \end{array} \\ \begin{bmatrix} 0{,}1 & 0{,}2 & 0{,}2 \\ 0{,}1 & 0{,}5 & 0{,}8 \end{bmatrix} \end{array} = \begin{array}{c} \\ p \\ v \\ o \\ i \end{array} \begin{array}{c} \begin{array}{ccc} b & t & a \end{array} \\ \begin{bmatrix} 0{,}1 & 0{,}2 & 0{,}2 \\ 0{,}1 & 0{,}1 & 0{,}1 \\ 0{,}1 & 0{,}3 & 0{,}3 \\ 0{,}1 & 0{,}5 & 0{,}8 \end{bmatrix} \end{array}$$

Al interpretar esta relación entre «prendas de *abrigo*» y «estaciones *frías*» hemos de hacer el mismo comentario que antes con respecto a R_1.

4.4. Lógica borrosa

4.4.1. *Sintaxis y semántica*

La forma de las sentencias en lógica borrosa es la misma que en lógica de predicados. Ello quiere decir que valen las mismas reglas de formación. Ahora bien, ¿podría construirse un sistema axiomático? La respuesta es «no» (sin «borrosidad»). En efecto, consideremos un simple predicado, como «viejo». Aplicado a un individuo concreto, tiene que tener una interpretación. En una lógica multivalorada podríamos decir, por ejemplo, que $I(V(\text{Juan})) = 0{,}6$. Pero en lógica borrosa ya hemos dicho al principio que los valores de «verdad» y «falsedad» son imprecisos. Esto quiere decir que $I(V(x))$ es el subconjunto borroso «viejo», definido, por ejemplo, como hacíamos más arriba. En una lógica multivalorada puede haber sentencias cuya interpretación sea siempre 1 (verdadera), que serán sentencias válidas (tautologías, en el caso de la lógica de proposiciones multivalorada), y, por tanto, se podrá construir un sistema axiomático que permita, a partir de unos axiomas, derivar otras sentencias válidas. Pero en lógica borrosa no puede definirse el concepto de «sentencia válida», y por ello no puede haber sistema axiomático, y, por lo mismo, tampoco tiene sentido hablar de «completitud» ni de «consistencia». Esto aleja tanto a la lógica borrosa de la «tradición lógica» que muchos autores discuten que se le pueda llamar «lógica». La única defensa de la lógica borrosa (por ahora, mientras no se revise todo el cuerpo de la lógica) es pragmática: con ella se pueden modelar situaciones y construir sistemas que no se pueden abordar con otras lógicas, y si estos modelos permiten comprender mejor las situaciones y diseñar sistemas que funcionan, tanto peor para la «tradición lógica».

El conjunto de valores semánticos en la lógica borrosa no es, pues, el conjunto de puntos en el intervalo [0, 1], sino un conjunto de subconjuntos borrosos de ese conjunto. Esos subconjuntos borrosos no son todos los posibles, sino lo que se llaman *valores de verdad lingüísticos*. Concretamente,

V = {verdadero, falso, no verdadero, no falso, bastante verdadero, bastante falso, poco verdadero, muy verdadero, más o menos verdadero, más bien verdadero, ...}.

Para cada predicado, se definirá el subconjunto correspondiente a «verdadero» (por ello, se dice que los valores de verdad en lógica borrosa son «locales»). Así, en el caso de la sentencia «Juan es viejo», el subconjunto correspondiente a que esta sentencia sea verdadera puede ser el definido en los ejemplos anteriores. Este subconjunto es el *significado* del predicado. Definido este subconjunto, los correspondientes a los otros valores de verdad lingüísticos pueden calcularse en función de él definiendo previamente unos convenios para «falso» y para las partículas lingüísticas «muy», «bastante», etc.:

* «falso» es el subconjunto tal que $\mu_{\text{falso}}(x) = \mu_{\text{verdadero}}(c(x))$, donde $c(x)$ es alguna función complementaria definida sobre U (por ejemplo, si $U = [1, 150]$, $c(x) = 150 - x$);
* «no verdadero» = $\overline{(\text{«verdadero»})}$;
* «muy verdadero» = («verdadero»)^2;
* «algo verdadero» = («verdadero»)$^{1/2}$; etc.

Por ejemplo, en el caso de que la sentencia sea el predicado «viejo» aplicado a alguien, tendremos para «muy viejo» el significado:

$$S(\text{muy } v) = S^2(v) = \{x|\mu_{S^2(v)}(x)\},$$

con $\mu_{S^2(v)}(x) = \mu^2_{S(v)}(x)$.

En la figura 5.4 puede verse la representación gráfica de este convenio para «muy», que, más o menos, corresponde a la interpretación intuitiva de la partícula.

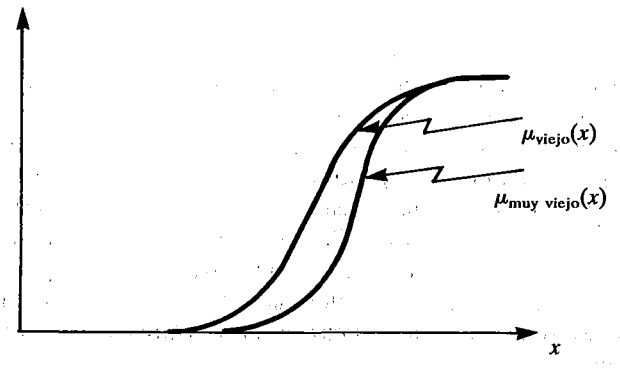


FIGURA 5.4.

4.4.2. *Interpretación de sentencias*

Para toda sentencia se tiene que poder calcular un *significado*, es decir, un subconjunto borroso que corresponde a la interpretación «verdadera» de tal sentencia. Empecemos por los casos más sencillos: los de las sentencias construidas con la negación, la conjunción y la disyunción. El cálculo del significado de tales sentencias se hace del siguiente modo:

$$S(\neg A) = \overline{S(A)}$$
$$S(A \wedge B) = S(A) \cap S(B)$$
$$S(A \vee B) = S(A) \cup S(B)$$

Como vemos, basta con utilizar las operaciones de complementación, intersección y unión entre conjuntos definidas en el apartado 4.2.

Por ejemplo, interpretemos la sentencia «Manuel es un viejo que ronda los setenta

años», es decir, «Viejo (Manuel) ∧ ronda setenta (Manuel)». La interpretación (o significado) de viejo puede ser la que venimos utilizando. Hemos de dar significado a «ronda setenta años». Gráfica y aproximadamente, podría ser como indica la figura 5.5, en la que también hemos señalado la función de pertenencia al conjunto intersección de «viejo» y «ronda setenta», que es el significado de la sentencia.

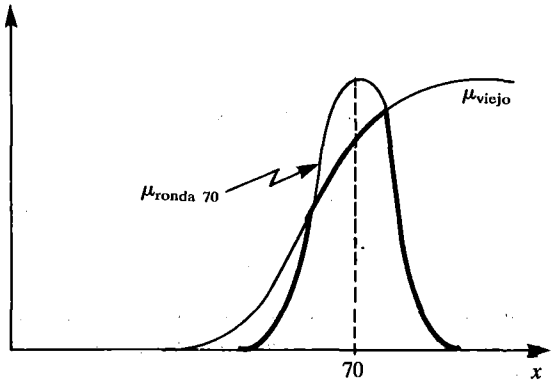


FIGURA 5.5.

Es fácil comprender que con ésto no hacemos sino abordar los casos más sencillos. Las sentencias, normalmente, manejan palabras cuyos significados son conjuntos borrosos de universos diferentes. Por ejemplo, en la sentencia «Manuel es un viejo extraordinariamente decrépito» tenemos dos variables borrosas: edad (con valor «viejo») y estado (con valor «decrépito»). La interpretación exige el conocer la de cada una de las variables (y definir «extraordinariamente», que podría ser, por ejemplo, el operador de potenciación con valor 3), y daría un resultado bidimensional.

Veamos otro ejemplo: Con los conjuntos A = {estaciones frías}, B = {sensación de frío} y C = {ropas de abrigo} definidos anteriormente, tendríamos:

S(estaciones muy frías) = A^2 = {p|0,09; v|0,01; o|0,16; i|0,81};
S(estaciones no muy frías) = $\overline{A^2}$ = {p|0,91; v|0,99; o|0,84; i| 0,19};
S(estaciones frías y no muy frías) = $A \cap \overline{A^2}$ = {p|0,3; v|0,1; o|0,4; i|0,19};
S(sensación de mucho frío) = B^2 = {T_1|0,04; T_2|0,64};
S(sensación de no mucho frío) = $\overline{B^2}$ = {T_1|0,96; T_2|0,36};
S(ropas que no abrigan) = $\overline{C}$ = {b|0,9; t|0,5; a|0,1};

etcétera.

Con ésto, podemos interpretar sentencias que contienen negaciones, conjunciones y disyunciones. El caso del condicional necesita un tratamiento aparte.

4.4.3. *Interpretación de sentencias condicionales*

Consideraremos el caso general «si A, entonces B, si no, C», es decir, «$(A \rightarrow B) \wedge (\neg A \rightarrow C)$», del cual «$(A \rightarrow B)$» será un caso particular. Cuando A, B y C son sentencias ordinarias con interpretaciones binarias, el cálculo de proposiciones clásico nos permite interpretar la sentencia como verdadera o falsa. Pero cuando A, B y C son sentencias borrosas, cada una tendrá un significado, y se trata de calcular el significado de la sentencia global.

Por ejemplo:

— si la humedad de la carretera es grande, el coche tiende a patinar;

o

— si estamos en una estación muy fría uso ropa de abrigo, si no, uso ropa de muy poco abrigo;

etcétera.

Se considera que este tipo de sentencia establece una relación entre los universos U (a que pertenece $S(A)$) y V (a que pertenecen $S(B)$ y $S(C)$), relación que se define de la siguiente manera:

* S (si A entonces B, si no C) = $[S(A) \times S(B)] \cup [\overline{S(A)} \times S(C)]$, donde «×» indica el producto cartesiano, definido en [2].

En el caso de que C no esté especificado («si A entonces B») se toma $S(C) = V$ (universo de C), es decir, se considera que el consecuente de «no A» puede ser cualquier subconjunto de V.

Ejemplos:

a) «Si la estación es fría se siente frío; si no, no se siente frío».

$$
\begin{aligned}
A &= S(\text{estación fría}) = \{p|0{,}3;\ v|0{,}1;\ o|0{,}4;\ i|0{,}9\} \\
B &= S(\text{se siente frío}) = \{T_1|0{,}2;\ T_2|0{,}8\} \\
\overline{A} &= S(\text{estación no fría}) = \{p|0{,}7;\ v|0{,}9;\ o|0{,}6;\ i|0{,}1\} \\
\overline{B} &= S(\text{no se siente frío}) = \{T_1|0{,}8;\ T_2|0{,}2\}
\end{aligned}
$$

La relación definida por la sentencia será:

$$
(A \times B) \cup (\overline{A} \times \overline{B}) = \begin{array}{c} \\ p \\ v \\ o \\ i \end{array} \begin{array}{c} \begin{array}{cc} T_1 & T_2 \end{array} \\ \begin{bmatrix} 0{,}2 & 0{,}3 \\ 0{,}1 & 0{,}1 \\ 0{,}2 & 0{,}4 \\ 0{,}2 & 0{,}8 \end{bmatrix} \end{array} \cup \begin{array}{c} \begin{array}{cc} T_1 & T_2 \end{array} \\ \begin{bmatrix} 0{,}7 & 0{,}2 \\ 0{,}8 & 0{,}2 \\ 0{,}6 & 0{,}2 \\ 0{,}1 & 0{,}1 \end{bmatrix} \end{array} = \begin{array}{c} \\ p \\ v \\ o \\ i \end{array} \begin{array}{c} \begin{array}{cc} T_1 & T_2 \end{array} \\ \begin{bmatrix} 0{,}7 & 0{,}3 \\ 0{,}8 & 0{,}2 \\ 0{,}6 & 0{,}4 \\ 0{,}2 & 0{,}8 \end{bmatrix} \end{array}
$$

b) «Si se siente frío, se usa ropa de abrigo; si no, no».

$$
\begin{aligned}
C &= S(\text{se usa ropa de abrigo}) = \{b|0{,}1;\ t|0{,}5;\ a|0{,}9\} \\
\overline{C} &= S(\text{no se usa ropa de abrigo}) = \{b|0{,}9;\ t|0{,}5;\ a|0{,}1\}
\end{aligned}
$$

$$(B \times C) \cup (\overline{B} \times \overline{C}) = \begin{matrix} \\ T_1 \\ T_2 \end{matrix} \begin{matrix} b & t & a \end{matrix} \begin{bmatrix} 0{,}1 & 0{,}2 & 0{,}2 \\ 0{,}1 & 0{,}5 & 0{,}8 \end{bmatrix} \cup \begin{matrix} b & t & a \end{matrix} \begin{bmatrix} 0{,}8 & 0{,}5 & 0{,}1 \\ 0{,}2 & 0{,}2 & 0{,}1 \end{bmatrix} = \begin{matrix} \\ T_1 \\ T_2 \end{matrix} \begin{matrix} b & t & a \end{matrix} \begin{bmatrix} 0{,}8 & 0{,}5 & 0{,}2 \\ 0{,}2 & 0{,}5 & 0{,}8 \end{bmatrix}$$

La composición de las relaciones correspondientes a las sentencias a) y b) será:

$$\begin{matrix} \\ p \\ v \\ o \\ i \end{matrix} \begin{matrix} T_1 & T_2 \end{matrix} \begin{bmatrix} 0{,}7 & 0{,}3 \\ 0{,}8 & 0{,}2 \\ 0{,}6 & 0{,}4 \\ 0{,}2 & 0{,}8 \end{bmatrix} \cdot \begin{matrix} \\ T_1 \\ T_2 \end{matrix} \begin{matrix} b & t & a \end{matrix} \begin{bmatrix} 0{,}8 & 0{,}5 & 0{,}2 \\ 0{,}2 & 0{,}5 & 0{,}8 \end{bmatrix} = \begin{matrix} \\ p \\ v \\ o \\ i \end{matrix} \begin{matrix} b & t & a \end{matrix} \begin{bmatrix} 0{,}7 & 0{,}5 & 0{,}3 \\ 0{,}8 & 0{,}5 & 0{,}2 \\ 0{,}6 & 0{,}5 & 0{,}4 \\ 0{,}2 & 0{,}5 & 0{,}8 \end{bmatrix}$$

4.4.4. *Reglas de inferencia borrosas*

Consideremos estas dos premisas:

*P*1: Estamos en una estación no muy fría.
*P*2: Si la estación es fría, se siente frío, si no, no.

¿Qué conclusión podríamos obtener? Obviamente, una conclusión borrosa referente al grado de frío que se siente en esta estación. De acuerdo con lo visto anteriormente, el significado de *P*1 es un subconjunto borroso de $U = \{p, v, o, i\}$, y el de *P*2 es una relación borrosa entre U y $V = \{T_1, T_2\}$. Y la conclusión será un subconjunto borroso de V.

La regla de inferencia borrosa para este caso (que es una generalización del modus ponens) es:

Si R es una relación borrosa entre U y V, y X es un subconjunto borroso de U, el subconjunto borroso inducido en V por X viene dado por la composición

$$Y = X \circ R$$

(tomando X como una relación unaria).

En el ejemplo dado,

$$X = \overline{S^2(\text{fría})} = \{p|0{,}91;\ v|0{,}99;\ o|0{,}84;\ i|0{,}19\}$$

$$Y = \begin{matrix} p & v & o & i \end{matrix} \begin{bmatrix} 0{,}91 & 0{,}99 & 0{,}84 & 0{,}19 \end{bmatrix} \begin{matrix} T_1 & T_2 \end{matrix} \begin{bmatrix} 0{,}7 & 0{,}3 \\ 0{,}8 & 0{,}2 \\ 0{,}6 & 0{,}4 \\ 0{,}2 & 0{,}8 \end{bmatrix} = \begin{matrix} T_1 & T_2 \end{matrix} \begin{bmatrix} 0{,}8 & 0{,}4 \end{bmatrix}$$

Si a las dos sentencias anteriores se añade: «si se siente frío se usa ropa de abrigo, si no, no», el resultado (grado de uso de las distintas prendas en las estaciones no muy frías) puede calcularse, de acuerdo con los resultados anteriores, así:

$$\begin{matrix} & & b & t & a \\ T_1 & T_2 & & & \end{matrix} \quad \begin{matrix} & & b & t & a \\ [0{,}8 & 0{,}4] & \begin{bmatrix} 0{,}8 & 0{,}5 & 0{,}2 \\ 0{,}2 & 0{,}5 & 0{,}8 \end{bmatrix} & = & [0{,}8 \quad 0{,}5 \quad 0{,}4] \end{matrix}$$

o así:

$$\begin{matrix} & & & & & b & t & a \\ p & v & o & i & & & & \\ [0{,}91 & 0{,}99 & 0{,}84 & 0{,}19] & \begin{bmatrix} 0{,}7 & 0{,}5 & 0{,}3 \\ 0{,}8 & 0{,}5 & 0{,}2 \\ 0{,}6 & 0{,}5 & 0{,}4 \\ 0{,}2 & 0{,}5 & 0{,}8 \end{bmatrix} & = & [0{,}8 \quad 0{,}5 \quad 0{,}4] \end{matrix}$$

4.4.5. *Aproximaciones lingüísticas*

Ahora bien, una vez interpretada una sentencia u obtenida una inferencia como hemos visto, lo que tenemos es un subconjunto borroso de un cierto universo. Pero lo que deberíamos obtener sería una sentencia como «se siente bastante frío», «se tienden a usar ropas de no mucho abrigo», etc. Para ello, es preciso asignar a cada subconjunto borroso una *aproximación lingüística*, que es otro subconjunto borroso que corresponde al significado de una sentencia construida con las partículas «muy», «bastante», etc.

En el último ejemplo habíamos llegado a las conclusiones

$$C1\text{: } \{T_1|0{,}8;\ T_2|0{,}4\}$$
$$C2\text{: } \{b|0{,}8;\ t|0{,}5;\ a|0{,}4\}$$

referentes al frío que se siente y al grado de uso de las ropas de abrigo. Por otra parte, los conjuntos borrosos correspondientes a los significados de «se siente frío» y «la ropa es de abrigo» se habían definido así:

$$\text{«frío»} = \{T_1|0{,}2;\ T_2|0{,}8\}$$
$$\text{«abrigo»} = \{b|0{,}1;\ t|0{,}5;\ a|0{,}9\}$$

El significado de «no hace frío» sería el de $\overline{\text{«frío»}}$, que, aproximadamente, se corresponde con el de $C1$. Para un mejor ajuste, tendríamos que combinar otras partículas (bastante, más bien, etc.). Igual ocurre con $C2$, que, en una primera aproximación, podría asimilarse al complementario de «se usan ropas de abrigo».

5. RESUMEN

Hemos pasado revista en este capítulo a algunas de las llamadas «lógicas no clásicas» (hay otras: lógica intuicionista, lógica dinámica, etc.), entendiendo por tales las que modelan aspectos de los procesos de razonamiento no contemplados en las lógicas de proposiciones y de predicados (aunque también hemos incluido aquí las lógicas de clases y de relaciones, que no son más que otra manera de contemplar la lógica de predicados).

Entre las que representan ampliaciones de la lógica clásica, es decir, que conservan todo su lenguaje y su sistema axiomático, hemos comentado la lógica de predicados de orden superior, la lógica modal y la lógica temporal. Y entre las que invalidan ese sistema axiomático, la multivalorada y la borrosa. Esta última, además de contemplar infinitos valores entre la verdad y la falsedad, considera que esos mismos predicados semánticos («verdadero» y «falso») son, en sí mismos, imprecisos. Para modelar esta situación hay una herramienta matemática, la teoría de conjuntos borrosos, cuya base hemos estudiado.

6. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

A pesar del adjetivo «no clásicas», muchos conceptos (nociones de «modalidades» y de «contingente» como algo que no puede considerarse verdadero ni falso) se remontan a la lógica aristotélica. Pero la formulación moderna de la lógica modal se debe a Lewis (1932), que introdujo los operadores modales y la «implicación estricta», y la de la lógica multivalorada a Lukasiewicz (1920, 1930). La formulación de Lukasiewicz sobre lógica multivalorada no es la única. Otros autores prefieren definir de otro modo el significado de las conectivas, como Bochvar (1939) y Kleene (1952).

Las «teorías borrosas» van indisolublemente unidas al nombre de Lofti Zadeh. Fue él quien introdujo los conjuntos borrosos («fuzzy sets») (Zadeh, 1965), los algoritmos borrosos (Zadeh, 1968), sus aplicaciones biológicas (Zadeh, 1969), sus aplicaciones a sistemas complejos y procesos de decisión (Zadeh, 1973), la lógica borrosa (Zadeh, 1974) y la «teoría de la posibilidad» (Zadeh, 1978), con la que se formaliza la idea de inferencia borrosa de una manera más rigurosa que la que hemos sugerido aquí.

El modelo relacional para las bases de datos se debe a Codd (1970). Un texto recomendable sobre este tema es el de Ullman (1982).

Como referencias generales sobre las lógicas no clásicas podemos citar los libros de Haak (1974, 1978), que aborda el tema desde el punto de vista teórico y filosófico, y el de Turner (1984), que lo hace centrándose en las aplicaciones a la informática.

La lógica modal se ha aplicado tanto en teoría de la programación (Manna y Pnueli, 1979) como en métodos de inferencia no monotónica (McDermott y Doyle, 1980) y en representación del conocimiento (Moore, 1984).

Igualmente, la lógica temporal se ha aplicado para la programación concurrente (Manna y Pnueli, 1981; Manna y Wolper, 1984) y para sistemas de inteligencia artificial (Allen, 1981; McDermott, 1982).

Sobre la teoría y las aplicaciones de los conjuntos borrosos hay un libro muy

completo, en tres volúmenes, de Kaufmann (1973-75): el primero incluye los elementos teóricos de base, el segundo, aplicaciones a la lingüística, la lógica y la semántica, y el tercero, al reconocimiento de formas, a los autómatas, a los sistemas y a las decisiones. Hay también libros en los que se recopilan muchos trabajos sobre aplicaciones: Zadeh *et al.* (1975), Gupta *et al.* (1977).

7. EJERCICIOS

7.1. Analizar en lógica de clases los razonamientos expuestos en el Ejercicio 7.3 del capítulo 4.

7.2. Formalizar los siguientes enunciados, teniendo en cuenta que algunos pueden ser ambiguos (y tener más de una formalización):

a) Si es necesario que si p entonces q, entonces si es necesario que p entonces es necesario que q.
b) No es posible ser valiente u osado si y sólo si no es posible ser valiente y no es posible ser osado.
c) Si Pepe siempre se ha mareado cuando ha bebido, entonces si mañana bebe se mareará.
d) Es posible que un programa haya funcionado siempre bien y que mañana no funcione.

7.3. Analizar mediante tablas de verdad si las leyes de modus ponens y modus tollens son tautologías en lógica trivalorada.

7.4. Definir los conjuntos borrosos que sean necesarios en cada caso y analizar los siguientes razonamientos:

a) *P*1: La mayoría de los hombres son heterosexuales.
*P*2: Sócrates es hombre.
C : Es muy posible que Sócrates sea heterosexual.

b) *P*1: La temperatura es un poco alta.
*P*2: Cuando la temperatura es alta hay que cerrar un poco la válvula.
C : Hay que cerrar ligeramente la válvula.

Capítulo 6
APLICACIONES EN INGENIERIA DEL CONOCIMIENTO

1. INTRODUCCIÓN

Se llama *sistema experto* a un sistema informático diseñado para resolver problemas en algún área muy específica del saber, con una competencia al menos similar a la que pueda tener un experto humano en ese área. Por ejemplo, MYCIN es un sistema experto en diagnóstico y tratamiento de un número muy reducido de enfermedades infecciosas de la sangre, PROSPECTOR lo es en determinar la probabilidad de la existencia de yacimientos de ciertos minerales a partir de las evidencias de campo, XCON en configurar sistemas informáticos con ordenadores VAX y PDP, etc.

Según el caso, el objetivo del sistema experto puede ser el de sustituir al experto humano (lo que puede tener un especial interés en aquellas aplicaciones en las que la experiencia tiene que estar disponible en lugares peligrosos o geográficamente remotos o inaccesibles) o el de ayudar a los expertos humanos a tratar con volúmenes de información que desbordan su capacidad (por ejemplo, en medicina). En cualquier caso, el objetivo final es el mismo de todas las aplicaciones informáticas: relevar al hombre de tareas mecanizables y proporcionarle instrumentos amplificadores de sus capacidades mentales.

Los sistemas expertos se construyen siguiendo una concepción modular: la de los *sistemas basados en conocimiento*. Cualquier programa que resuelva un problema también tiene que incorporar, en cierto modo, el conocimiento necesario para resolver ese problema. Pero el conocimiento en sí y los procedimientos que permiten manipular ese conocimiento para obtener una respuesta ante datos concretos son indistinguibles en el código que constituye el programa. Por el contrario, en los sistemas basados en conocimiento se separan ambos: por una parte, se construye una *base de conocimientos*, y luego se amplía y se modifica en interacción con el o los expertos humanos en el tema. Por otra, se diseña un conjunto de procedimientos que permiten, a partir de unos hechos o evidencias (que constituyen la *base de hechos*),

manipular el conocimiento almacenado en la base de conocimientos para extraer conclusiones; esta segunda parte corresponde a lo que se llama *motor de inferencias*.

Es fácil intuir que el primer problema en el diseño de un sistema experto es el de decidir los esquemas para la *representación del conocimiento*. Podemos distinguir varios tipos de conocimiento. Hay un *conocimiento descriptivo, o declarativo*, sobre los hechos concretos del dominio de experiencia o sobre los datos del problema concreto a resolver (este último constituirá el contenido inicial de la base de hechos). Hay también un *conocimiento procedimental, o normativo*, de tipo táctico, que permite obtener conclusiones a partir del conocimiento descriptivo. Y hay, finalmente, un *conocimiento estratégico, o de control*, que determina la manera en que se aplica el conocimiento procedimental. Por aclarar ideas sobre un ejemplo muy sencillo, recuérdese el Ejercicio 7.4 del capítulo 4. En ese caso, el conocimiento descriptivo sería el contenido en las relaciones de parentesco «madre de» y «padre de» que se establezcan, el procedimental sería el constituido por las sentencias condicionales que se escriban para determinar las condiciones bajo las cuales una pareja de individuos está en la relación «antepasado de», y el estratégico sería el definido por el sistema inferencial, por ejemplo, la regla de resolución con búsqueda exhaustiva.

Otro problema es el de la *adquisición del conocimiento*, que puede ser, como es lo más frecuente en los sistemas actuales, a partir de un experto humano (lo cual exige un arduo trabajo para llegar a traducir tal conocimiento, generalmente difícil de explicitar y de naturaleza imprecisa y heurística, al esquema de representación elegido), o bien por autoaprendizaje (inducción del conocimiento a partir de ejemplos resueltos).

Además está, por supuesto, el problema del diseño del motor de inferencias (que, en principio, contiene lo que hemos llamado «conocimiento estratégico»), dependiente del esquema de representación. Y, finalmente, el problema de la comunicación con el usuario final: para que el sistema sea aceptable debe «convencer», y para ello debe ser capaz de justificar la línea de razonamiento seguida en cada caso, y, si hace alguna pregunta (es decir, pide algún dato para completar la base de hechos), explicar el motivo de la misma. Todo este conjunto de problemas ha dado lugar a un campo de trabajo que se conoce por el nombre de *ingeniería del conocimiento*.

En este capítulo veremos cómo la formulación lógica es un posible esquema para la representación del conocimiento. Para simplificar la exposición, usaremos solamente la lógica de proposiciones. Veremos también, a grandes rasgos, la estructura de los motores de inferencia (en lógica de proposiciones) y cómo se aborda el problema de la representación del conocimiento impreciso. También daremos una idea de otros esquemas más estructurados para la representación del conocimiento.

2. SISTEMAS DE PRODUCCIÓN

2.1. Estructura

Un *sistema de producción* es un modelo de computación que incluye tres componentes: una *base de datos*, un conjunto de *reglas de producción* y un *sistema de control*. Estos tres componentes son, respectivamente, la «base de hechos», la «base de

conocimientos» y el «motor de inferencias» del sistema de producción. La llamada «base de datos» no es necesariamente una base de datos en el sentido informático habitual: según el sistema, puede ser desde una sencilla matriz de números hasta una verdadera base de datos. Las reglas de producción se aplican sobre la base de datos, cambiando su estado en cada aplicación, y el sistema de control gobierna esas aplicaciones y hace que la computación se detenga cuando el estado de la base de datos cumple con alguna *condición de terminación* predefinida.

2.2. Base de datos

La base de datos contendrá los hechos iniciales y los que se vayan obteniendo como consecuencias en el proceso inferencial. Como hemos dicho, nos vamos a limitar a la lógica de proposiciones, por lo que, al nivel en que nos vamos a mover, la base de datos estará formada, simplemente, por un conjunto de variables proposicionales negadas o no (es decir, un conjunto de literales), cada una de las cuales representará a un hecho concreto. (En el caso de lógica de predicados, en lugar de variables proposicionales tendríamos predicados aplicados sobre valores concretos de las variables, o sea, sobre constantes). No nos ocuparemos, a este nivel, de problemas como el de la estructuración de estos datos, acceso a los mismos, etc.

2.3. Reglas de producción

Las *reglas de producción* (no confundir con las reglas de inferencia, que estarán incluidas en el sistema de control) son pares ordenados (A, B). Según la aplicación, los elementos del par reciben los nombres de «antecedentes» y «consecuente», «condiciones» y «acción» o «premisas» y «conclusión». Su formalización lógica será la de sentencias condicionales: $A \rightarrow B$. Como puede adivinarse por los nombres dados a A y a B, A será normalmente una conjunción de literales y B un literal. Es decir, supondremos que nuestras reglas de producción serán sentencias de la forma:

$$l_{A1} \wedge l_{A2} \wedge \ldots l_{An} \rightarrow l_B$$

De todos modos, A podría ser una sentencia cualquiera; con unas transformaciones similares a las que hacíamos en el capítulo 2 (apartado 5.6) para llegar a la forma clausulada puede verse que siempre puede transformarse en un conjunto de sentencias de este tipo particular. Por su parte, B también podría ser cualquier sentencia. Sólo supondremos, de momento, y por las razones que veremos más adelante, en el apartado 3.1, que no contiene la conectiva «$\vee$». Si, por ejemplo, B fuera una conjunción de literales, $l_{B1} \wedge l_{B2} \wedge \ldots \wedge l_{Bm}$, la sentencia podría descomponerse en m sentencias con el mismo antecedente y cada una de ellas con uno de los literales como consecuente.

Obsérvese que esta forma de sentencia es la que en el capítulo 2 (apartado 5.6) llamábamos «cláusula de Horn con cabeza», salvo que ahora las variables proposicionales pueden ser literales. (Por otra parte, todo el conjunto de la base de datos puede

interpretarse como una cláusula de Horn sin cabeza y negada). Esta forma, aparte de que nos va a simplificar la estructura del sistema de control (o motor de inferencias), es, además, la que de manera natural se obtiene cuando se le pide a un experto que ponga su conocimiento en forma de reglas.

Por ejemplo, en la base de conocimientos del sistema XCON hay unas 2.500 reglas. Una de ellas es:

— Si el contexto actual es el de asignar una fuente de alimentación,
y se ha colocado un módulo SBI en un armario,
y se conoce la posición que ocupa el módulo,
y hay a su lado espacio disponible,
y se dispone de una fuente de alimentación,
entonces colocar la fuente en el espacio disponible.

Es claro que, independientemente de lo que signifiquen tales frases en el contexto de conocimiento de XCON, la regla puede formalizarse mediante la sentencia:

$$(p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5) \rightarrow q$$

Otro ejemplo, menos real pero quizás más sugestivo, podría ser el de un experto médico que contuviera las reglas:

*R*1: Si el paciente tiene fiebre,
y tose,
y tiene dolores musculares,
entonces padece gripe.

*R*2: Si el paciente padece gripe o resfriado,
y no tiene úlcera,
entonces recomendar aspirina y coñac.

La formalización de tales reglas sería:

$R1: f \wedge t \wedge m \rightarrow g$
$R2: (g \vee r) \wedge \neg u \rightarrow a \wedge c$

Y esta segunda regla puede descomponerse en:

$R2a: g \wedge \neg u \rightarrow a$
$R2b: g \wedge \neg u \rightarrow c$
$R2c: r \wedge \neg u \rightarrow a$
$R2d: r \wedge \neg u \rightarrow c$

2.4. Sistema de control (o motor de inferencias)

2.4.1. *Estrategias hacia adelante y hacia atrás*

Suponemos existente una base de conocimientos codificada como un conjunto de reglas de producción, R_1, R_2, ... del tipo que hemos visto, en las cuales interviene un conjunto de hechos diversos representados por literales, l_1, l_2, ..., l'_1, l'_2, ... Ante una situación, se tiene la evidencia de que un subconjunto de esos hechos, l_1, l_2, ... son verdaderos, y se trata de encontrar qué otros hechos, l'_1, l'_2, ... pueden inferirse de esa certidumbre y de las reglas, o dicho de otro modo, de encontrar los l'_k tales que

$$l_i \wedge R_j \rightarrow l'_k$$

sean tautologías.

Utilizando el último ejemplo, si tenemos los hechos *f*, *t*, *m* y $\neg u$, podemos hacer las siguientes inferencias:

$$R1 : \frac{f \wedge t \wedge m \qquad f \wedge t \wedge m \rightarrow g}{g}$$ (modus ponens)

$$R2a: \frac{g \wedge \neg u \qquad g \wedge \neg u \rightarrow a}{a}$$ (modus ponens)

$$R2b: \frac{g \wedge \neg u \qquad g \wedge \neg u \rightarrow c}{c}$$ (modus ponens)

(Conclusión: la terapia es aspirina y coñac).

(Planteado en el lenguaje del cálculo de proposiciones, se trataría de demostrar que

$$[(f \wedge t \wedge m \wedge \neg u) \wedge (f \wedge t \wedge m \rightarrow g) \wedge (g \wedge \neg u \rightarrow a \wedge c)] \rightarrow (a \wedge c)$$

es un teorema).

Normalmente, no tendremos dos, sino muchas reglas (en los sistemas expertos es frecuente que sean del orden de cientos o de miles), y, ante unos hechos, hay dos formas de enfocar el procedimiento de inferencia (es decir, dos *estrategias básicas*):

(a) Ir aplicando cuantas reglas de producción y cuantas reglas de inferencia se puedan para ir sucesivamente ampliando la base de hechos. Es lo que hemos hecho en el ejemplo, y corresponde a lo que se llama *encadenamiento hacia adelante*. Es también lo que sugeríamos en el capítulo 2 al hablar de la resolución con búsqueda exhaustiva.

(b) Fijarse un hecho como objetivo y tratar de deducirlo, viendo de qué reglas de producción es consecuente, si alguno de los antecedentes de esas reglas no figura en la base de hechos fijarlo como subobjetivo, etc. Este es el principio del *encadenamiento hacia atrás*.

En ambos casos, el sistema puede llegar a un punto en el que para poder deducir algo le falten hechos no contenidos inicialmente en la base de hechos, y entonces preguntaría sobre ellos al usuario. (Por ejemplo, el sistema: «¿tose mucho el paciente?»; el usuario: «sí», o «no». Obsérvese que es poco natural que el usuario responda así; en los sistemas reales se puede responder en una escala que permite expresar la respuesta de forma más matizada, pero para ello hay que introducir mecanismos de representación del conocimiento impreciso, de los que hablaremos más adelante, en el apartado 3. Aquí nos estamos limitando al caso más sencillo, en el que los hechos sólo pueden ser verdaderos o falsos, que es lo único que permite la lógica clásica).

Dentro de cada una de estas estrategias básicas caben una serie de variantes, es decir, de *estrategias concretas*.

2.4.2. *Un ejemplo*

Antes de entrar en la explicación de los algoritmos de inferencia, veamos sobre un ejemplo cómo se aplicarían los procedimientos de encadenamiento hacia adelante y hacia atrás. El ejemplo es abstracto, en el sentido de que partiremos de un conjunto de reglas de producción y de hechos escritos en forma simbólica, prescindiendo de su significado en un contexto de conocimiento.

Supongamos la base de conocimientos constituida por las siguientes reglas de producción:

$R1: A \rightarrow C$
$R2: A \rightarrow H$
$R3: C \rightarrow D$
$R4: D \rightarrow E$
$R5: B \wedge F \rightarrow X$
$R6: D \wedge G \rightarrow B$
$R7: C \wedge F \rightarrow B$
$R8: A \wedge H \rightarrow D$
$R9: A \wedge C \wedge H \rightarrow B$
$R10: A \wedge B \wedge C \wedge H \rightarrow F$

(Aunque en este ejemplo utilicemos A, B, C, ... en lugar de p_1, p_2, ..., debe entenderse que se trata de variables proposicionales).

Y supongamos que tenemos como hecho inicial el A, es decir, $BH_o = \{A\}$. (La base de hechos inicial sólo contiene a A), y que nos planteamos como objetivo el ver si se puede inferir el hecho X.

Con encadenamiento hacia adelante, la estrategia más sencilla es la de ir recorriendo las reglas desde la primera hasta llegar a una que pueda aplicarse (de acuerdo con la regla de inferencia de modus ponens), ampliar la base de hechos con la consecuencia, empezar de nuevo con la primera regla, y así sucesivamente hasta incluir el objetivo en la base de hechos, o hasta que ya no puedan aplicarse reglas. En el caso del ejemplo, la $R1$ es aplicable, con lo que la base de hechos se incrementa con C,

luego se aplicaría la *R*2, etc. Escribiéndolo resumidamente, en forma de tabla, tendríamos:

ΔBH	*A*	*C*	*H*	*D*	*E*	*B*	*F*	*X*
R	1	2	3	4	9	10	5	

donde en la línea superior se indican los hechos con los que sucesivamente se va incrementando la base de hechos, y en la inferior las reglas que en cada momento se aplican. Como vemos, en este caso termina por incluirse *X*, por lo que del hecho inicial, *A*, se infiere *X*.

No es necesario volver a empezar siempre con la primera regla cada vez que se llega a aplicar una de ellas. Otra estrategia, por ejemplo, es la de buscar la regla que contenga más premisas incluidas en *BH*; con ella, la inferencia de *X* en el ejemplo seguiría esta otra secuencia:

ΔBH	*A*	*C*	*H*	*B*	*F*	*X*
R	1	2	9	10	5	

donde vemos que, en el momento en que se dispone de *A*, *C* y *H* en la *BH*, tras la aplicación de *R*1 y *R*2, ya no se aplica *R*3, sino *R*9, que tiene más premisas.

El otro procedimiento, más natural en un caso como el planteado en este ejemplo, en el que no se trata de derivar cuantas conclusiones se puedan, sino de inferir un objetivo, es el del encadenamiento hacia atrás. Sobre el ejemplo, y expresado informalmente, funcionaría así:

1. Se trata de ver si puede deducirse *X*. ¿En qué reglas figura como consecuente?
2. Vemos que sólo lo hace en la *R*5. Por tanto, nuestro objetivo (*X*) se descompone en los subobjetivos que figuran como antecedentes (en forma conjuntiva, es decir, tienen que darse *ambos*) de esa regla (*B* y *F*).
3. Empecemos con *B*. Figura como consecuente en tres reglas: *R*6, *R*7 y *R*9. Basta pues con deducir los antecedentes de *una* de esas tres reglas.
4. Para *R*6 hay que deducir *D* y *G*. Pero *G* no figura como consecuente en ningún sitio, por lo que podemos abandonar esta vía y mirar *R*7.

Y así sucesivamente. No seguimos porque la descripción verbal es extremadamente engorrosa, y hay un lenguaje gráfico muy adecuado para este proceso: el de los árboles «*Y-O*». El nodo raíz del árbol es el objetivo a deducir, los intermedios corresponden a los subobjetivos, y los terminales, u hojas, a los hechos finales, contenidos o no en la base de hechos. De cada nodo no terminal, si corresponde a un objetivo o subobjetivo que puede deducirse de varias reglas, sale una ramificación de tipo «*O*» (y si sólo hay una regla, sólo sale una rama). De cada una de estas ramas, a su vez, sale una ramificación de tipo «*Y*», correspondiente al conjunto de premisas que hay en cada regla. Y así hasta dar con las hojas correspondientes a hechos que no pueden demostrarse o que están en la base de hechos. El árbol «*Y-O*» para el caso de nuestro ejemplo sería el de la figura 6.1.

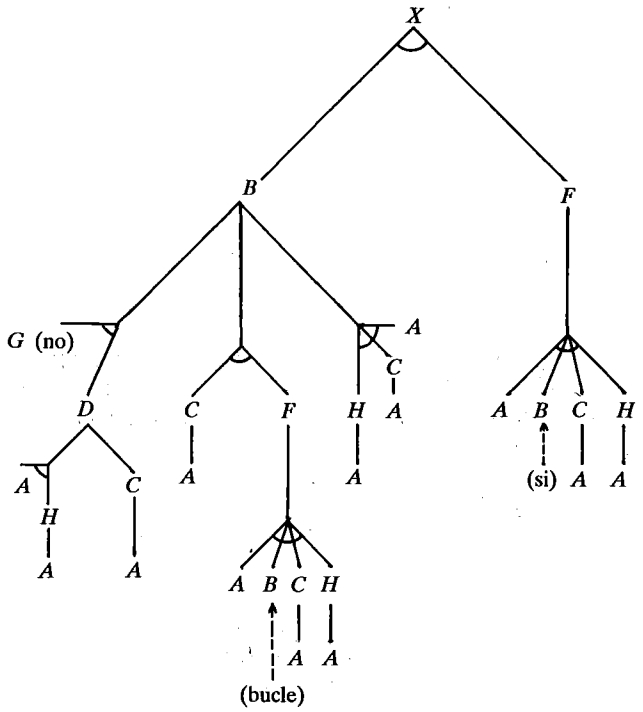


FIGURA 6.1.

2.4.3. *Estrategias de resolución impulsadas por los hechos*

En el ejemplo anterior hemos utilizado siempre la regla de inferencia de modus ponens. Como vimos en el capítulo 2, todas las reglas de inferencia pueden resumirse en una, la de resolución. Allí estudiamos la resolución en general, aplicada sobre dos cláusulas cualesquiera (generatrices). Vemos que ahora una de las cláusulas es un condicional y la otra un hecho (un literal). Por ejemplo, una vez que la base de hechos contiene B y F, $R5$ nos permite inferir, por modus ponens, X. Con la resolución necesitaríamos dos pasos (puesto que B y F son dos cláusulas). Concretamente, $R5$ en forma clausulada sería:

$$\neg B \vee \neg F \vee X$$

Resolviendo con B resultaría:

$$\neg F \vee X$$

y resolviendo con F, quedaría X.

Pero estos dos pasos se pueden encadenar. En efecto,

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \wedge (\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n \vee q) \rightarrow q$$

es una tautología (y, por tanto, una ley, y, consecuentemente, la formalización de una regla de inferencia). Ello nos permite disponer de una regla de resolución adaptada a nuestro caso. En efecto, nuestras reglas de producción son, como hemos dicho, de la forma

$$l_{A1} \wedge l_{A2} \wedge \ldots \wedge l_{An} \rightarrow l_B$$

o, puesta en forma disyuntiva,

$$\neg l_{A1} \vee \neg l_{A2} \vee \ldots \vee \neg l_{An} \vee l_B$$

Si la base de hechos contiene l_{A1}, l_{A2}, ..., l_{An}, la ley anterior nos permite inferir l_B.

Decimos que ésta es una *resolución impulsada por los hechos*: es una restricción de la regla de resolución, porque sólo una de las generatrices es una cláusula cualquiera (disyunción de literales), pero, al mismo tiempo, es una ampliación porque admite un número indefinido de otras cláusulas siempre que éstas sean simplemente literales (hechos).

2.4.4. *Un algoritmo con encadenamiento hacia adelante*

El algoritmo que vamos a dar aquí en términos muy generales, utilizando pseudocódigo, da por supuesto que todas las reglas de producción están en la forma disyuntiva.

Diremos que un literal l_i está *demostrado* si figura en la base de hechos (*BH*). Si figura como tal estará *demostrado como cierto*, y si lo que figura es su negación estará *demostrado como falso*.

Diremos que una regla $l_1 \vee l_2 \vee \ldots \vee l_n$ puede *dispararse* si todos los l_i menos uno están demostrados como falsos (y, en tal caso, nuestra regla de inferencia nos permite decir que el que queda es cierto). Una regla *se elimina* si o bien se ha disparado (y en ese caso no sirve más)* o bien alguno de sus *componentes* (literales) se ha demostrado que es cierto (y en ese caso no nos permite inferir nada). Las reglas *activas* son las que no están eliminadas.

Llamaremos *valor de una regla* al número de componentes no demostradas, si está activa; si una regla está eliminada su valor será 0. El *valor de una componente* (dentro de una determinada regla) será 0 ó 1, según que esa componente figure negada o sin negar, respectivamente, en la regla.

El algoritmo utiliza dos procedimientos. Uno de ellos, al que llamaremos «inferir», recorre todas las reglas para ver si alguna puede dispararse, y, en caso afirmativo,

---

* Obsérvese que ésto sería incorrecto en el caso de utilizar la lógica de predicados.

infiere la conclusión, la introduce en la base de hechos, elimina la regla y llama al segundo procedimiento, «actualizar». Lo que hace éste es, dado un hecho, actualizar el valor de todas las reglas en las que figura. Tenemos así:

```
— Procedimiento inferir;
  mientras haya reglas activas
     para cada regla activa
        si valor [regla] = 1, entonces
           conclusión: = componente no demostrado;
           valor [conclusión]: = valor del componente en la regla;
           introducir conclusión con su valor en la BH;
           valor [regla]: = 0;
           actualizar (conclusión);

— Procedimiento actualizar (hecho);
  para cada regla activa
     si regla contiene hecho, entonces
        si valor [hecho] = valor del hecho en la regla, entonces
        valor [regla]: = 0
        si no, valor [regla]: = valor [regla] − 1;
```

El programa se limitaría a ir leyendo las premisas (base de hechos inicial), y, para cada una, llamar a «actualizar» y luego a «inferir» (que, a su vez, y en su caso, también llamará a «actualizar»):

```
— Programa encadenamiento adelante;
  mientras haya premisas,
     leer (premisa);
     actualizar (premisa);
     inferir.
```

2.4.5. *Un algoritmo con encadenamiento hacia atrás*

Este otro algoritmo utiliza dos procedimientos. El llamado «nodoo» resuelve un nodo del árbol de tipo «*O*», examinando todas las reglas que contienen al objetivo en cuestión; si no puede demostrarlo (variable booleana «demostrado» con el valor falso tras examinar todas las posibilidades), pregunta por él al usuario. El procedimiento «nodoy» se aplica a un conjunto de objetivos (los que corresponden a una ramificación de tipo «*Y*»), a cada uno de los cuales aplica el «nodoo»:

```
— Procedimiento nodoo (objetivo, demostrado);
  demostrado: = falso;
     si objetivo incluido en BH, entonces demostrado: = cierto;
        si no, para todas las reglas que incluyan al objetivo,
           mientras demostrado sea falso
              elegir una regla, Ri;
              nodoy (términos de Ri, demostrado);
```

si demostrado = falso, entonces
preguntar (objetivo);
si hay respuesta, entonces
añadirla a la *BH*;
demostrado: = cierto;

— Procedimiento nodoy (conjunto de objetivos, demostrado);
para todos los objetivos del conjunto,
nodoo (objetivo, demostrado).

El programa se limitaría a llamar a nodoo con el objetivo final a demostrar y desde ese momento actuarían ya recursivamente los dos procedimientos.

3. INFERENCIA PLAUSIBLE

3.1. Fuentes de imprecisión y de incertidumbre

Ocurre con mucha frecuencia que, desde el primer momento en que se empieza a pensar en el diseño de un sistema basado en conocimiento, hay que enfrentarse ineludiblemente al problema de la incertidumbre y de la imprecisión, en dos facetas distintas:

(a) En el experto, cuando tiene que expresar su conocimiento (conocimiento «procedimental»). Si el esquema de representación elegido es el de las reglas de producción (el único que aquí estamos tratando) la incertidumbre y la imprecisión se traducen en la forma de las reglas.

En efecto, decíamos en el apartado 2.3 que las reglas de producción tenían que poder expresarse como sentencias condicionales en cuyo consecuente no figurase la conectiva «$\vee$». La explicación de ello es que una sentencia como

$$A \rightarrow q_1 \vee q_2$$

correspondería a una «duda» en el conocimiento de la persona que ha establecido la regla de la que procede esa sentencia: ante la evidencia A se deduce que o bien q_1 o bien q_2, ¿pero cuál de los dos? Sin embargo, tales dudas son frecuentes cuando se trata de explicitar el conocimiento de los expertos. Así, una de las reglas utilizadas como ejemplo al final del mismo apartado 2.3 sería seguramente más razonable redactarla de este otro modo:

— Si el paciente tiene fiebre,
y tose,
y tiene dolores musculares,
ıtonces
padece gripe,
o padece bronquitis,
o padece tuberculosis,
...

Desde luego, reglas de producción expresadas de ese modo no permitirían llegar a ninguna conclusión. Pero hay algo más en la mente del experto (si no lo hubiera, tampoco él podría concluir nada): ante unas evidencias (signos y síntomas en el caso del diagnóstico médico), puede albergar dudas, pero normalmente «cree» más en unas alternativas que en otras. El problema es cómo cuantificar ese «grado de creencia», y cómo trabajar con él para poder hacer deducciones «plausibles».

(b) En el usuario, cuando tiene que decir si cierto hecho está presente o no (conocimiento «declarativo»). Por ejemplo, decíamos más arriba (apartado 2.4.1) que parece poco razonable que el usuario tenga que responder «sí» o «no» a una pregunta del sistema sobre si el paciente tose mucho.

«Incertidumbre» e «imprecisión» son dos conceptos diferentes: una proposición es incierta si su valor de verdad o falsedad no se conoce o no se puede determinar, y es imprecisa si se refiere a alguna variable cuyo valor no puede determinarse con exactitud (por tanto, una proposición incierta puede ser precisa, y una imprecisa no ser incierta). En el ejemplo de (a), los condicionales son inciertos, mientras que en el ejemplo de (b), la respuesta del usuario sería imprecisa.

Veremos aquí tres de los mecanismos que se utilizan en los sistemas basados en conocimiento para abordar estos problemas. Al nivel introductorio en que nos vamos a mover no será necesario entrar en diferenciaciones entre incertidumbre e imprecisión.

3.2. Inferencia bayesiana

El método bayesiano es el adoptado en el sistema PROSPECTOR (y en otros derivados de él), y se basa en la teoría de la probabilidad para cuantificar la imprecisión y/o la incertidumbre y trabajar con ella.

La idea básica consiste en asociar probabilidades a las reglas de producción, e identificar tales reglas con probabilidades condicionales. Si llamamos H al suceso consistente en que cierta *hipótesis* sea verdadera y E al consistente en que cierta *evidencia* esté presente, podemos identificar la probabilidad condicional:

$$P(H|E) \text{ (probabilidad de que se dé } H \text{ supuesto } E)$$

con la sentencia condicional:

> «Si la evidencia E está presente, entonces H es verdadera con probabilidad $P(H|E)$».

Ahora bien, obsérvese que esto es, precisamente, lo que el experto debe deducir: si hay varias hipótesis posibles (por ejemplo, aquí debajo hay un yacimiento de molibdeno, o de cobre, o de oro, o... no hay nada) y se presenta una evidencia E compuesta por otras, $E = E_1$ y E_2 y... (por ejemplo, el terreno es arcilloso y en un arroyo cercano alguien se ha encontrado una pepita de oro y...), de lo que se trata justamente es de llegar a saber cuáles son las probabilidades $P(H_1|E)$, $P(H_2|E)$, ... Si el experto (humano) pudiera darnos todas esas probabilidades para *todas y cada una*

de las posibles combinaciones de E, entonces el sistema experto se limitaría a tenerlas almacenadas y a efectuar una búsqueda cuando se le diera una determinada combinación de E. Pero ello es impensable, por el gran número de combinaciones posibles que pueden formar E: no hay experto dispuesto a pasarse años enumerando las distintas posibilidades (terreno arcilloso o no, presencia de ciertas rocas o no, etc.) y dando para cada una de ellas y para cada hipótesis posible una probabilidad.

Lo que sí puede darnos el experto humano son unas estimaciones de las probabilidades *a priori* de cada una de las hipótesis ($P(H_j)$: probabilidad de que se dé H sin saber nada más) y de las probabilidades de que se presenten cada uno de los elementos atómicos de evidencia supuesto que cada una de las hipótesis es verdadera ($P(E_i|H_i)$: probabilidad de que, supuesto que hay molibdeno, el terreno sea arcilloso, etc.). Con esta información, el cálculo de las probabilidades a posteriori, que es lo que interesa, se puede hacer aplicando el conocido teorema de Bayes: si aparece la evidencia E_i, entonces

$$P(H_j|E_i) = \frac{P(H_j)P(E_i|H_j)}{\sum_j P(H_j)P(E_i|H_j)}$$

Sabemos que una suposición de partida en la demostración de este teorema es que las H_j son mutuamente excluyentes, suposición que, normalmente, es poco razonable en las aplicaciones prácticas (por ejemplo, la existencia de un cierto mineral no excluye la posibilidad de que haya otros). Ahora bien, si conseguimos que el experto humano nos dé no sólo las $P(H_j)$ y las $P(E_i|H_j)$ sino también las $P(E_i|\overline{H}_j)$ podemos escribir el teorema en esta otra forma:

$$P(H_j|E_i) = \frac{P(H_j)P(E_i|H_j)}{P(H_j)P(E_i|H_j) + P(\overline{H}_j)P(E_i|\overline{H}_j)} \tag{1}$$

en la que H_j y $\overline{H}_j$ son siempre mutuamente excluyentes, y donde $P(\overline{H}_j)$ se puede sustituir por $1 - P(H_j)$.

Análogamente,

$$P(H_j|\overline{E}_i) = \frac{P(H_j)P(\overline{E}_i|H_j)}{P(H_j)P(\overline{E}_i|H_j) + P(\overline{H}_j)P(\overline{E}_i|\overline{H}_j)} \tag{2}$$

donde $P(\overline{E}_i|H_j) = 1 - P(E_i|H_j)$ y $P(\overline{E}_i|\overline{H}_j) = 1 - P(E_i|\overline{H}_j)$.

Pero aún no hemos terminado, porque lo normal no es que aparezca un solo átomo aislado de evidencia, E_i, sino varios, de modo que $E = E_1 \cup E_2 \cup \ldots$, y lo que queremos calcular es $P(H_j|E)$. En este caso, hay dos modos de proceder:

(a) global: se calculan

$$P(E|H_j) = \prod_i P(E_i|H_j),$$

tras lo cual se aplica la fórmula anterior; para que ese cálculo sea correcto es preciso que todos los E_i sean independientes entre sí;

(b) por pasos: para E_1, se calculan con la fórmula (1) $P(H_j|E_1)$ (si es que E_i es verdadera; si no, se utiliza la fórmula (2)); estos valores se toman como nuevos valores de las probabilidades a priori, $P(H_j)$, y se vuelve a aplicar la fórmula con E_2, y así sucesivamente. Puede demostrarse que el resultado es el mismo que con el procedimiento anterior.

Este método, con algunas variantes que luego explicaremos, constituye el mecanismo inferencial utilizado en PROSPECTOR. Para ayudar a aclarar las ideas, vamos a concretar sobre un ejemplo sencillo: el de la regla a que antes nos referíamos sobre la fiebre, la gripe, etc. Puestos a concretar dando probabilidades, podríamos pensar en descomponer la regla en las nueve siguientes:

*R*1: Si tiene fiebre, entonces padece gripe con probabilidad 0,5.
*R*2: Si tiene fiebre, entonces padece bronquitis con probabilidad 0,1.
*R*3: Si tiene fiebre, entonces padece tuberculosis con probabilidad 0,4.
*R*4: Si tose mucho, entonces padece gripe con probabilidad 0,1.
*R*5: Si tose mucho, entonces padece bronquitis con probabilidad 0,7.
*R*6: Si tose mucho, entonces padece tuberculosis con probabilidad 0,2.
*R*7: Si tiene dolores musculares, entonces padece gripe con probabilidad 0,7.
*R*8: Si tiene dolores musculares, entonces padece bronquitis con probabilidad 0,2.
*R*9: Si tiene dolores musculares, entonces padece tuberculosis con probabilidad 0,1.

Si llamamos E_i a las *evidencias* (en este caso, signos y síntomas: E_1 = tiene fiebre, etc.) y H_j a las *hipótesis* (en este caso, enfermedades: H_1 = gripe, etc.), cada una de las reglas equivale a una probabilidad condicional (por ejemplo, *R*1: $P(H_1|E_1) = 0{,}5$: la probabilidad de que tenga gripe supuesto que manifiesta fiebre es 0,5).

Pero obsérvese que no es ésta la información que se precisa para aplicar el mecanismo de inferencia bayesiano. En efecto, ¿qué ocurre si unas evidencias están presentes y otras no?; es decir, ¿cómo calcular, por ejemplo, $P(H_1|E_1$ y E_2 y no $E_3)$? La información que se precisa no son las $P(H_j|E_i)$, sino las $P(E_i|H_j)$ (además de las $P(H_j)$).

Por tanto, supongamos que nuestro experto médico (humano) nos ha provisto con los siguientes elementos de conocimiento:

$$P(H_1) = 0{,}02; \quad P(H_2) = 0{,}01; \quad P(H_3) = 0{,}001,$$

donde H_1 = gripe, H_2 = bronquitis y H_3 = tuberculosis. (Dicho sea de paso, estas probabilidades a priori son lo que en términos médicos se llaman «prevalencias» de las distintas enfermedades). Y asimismo, llamando E_1 a la fiebre, E_2 a la tos y E_3 a los dolores musculares,

$$P(E_1|H_1) = 0{,}95 \qquad P(E_1|\overline{H}_1) = 0{,}01$$
$$P(E_1|H_2) = 0{,}8 \qquad P(E_1|\overline{H}_2) = 0{,}03$$

$$P(E_1|H_3) = 0{,}6 \qquad P(E_1|\overline{H}_3) = 0{,}02$$
$$P(E_2|H_1) = 0{,}3 \qquad P(E_2|\overline{H}_1) = 0{,}03$$
$$P(E_2|H_2) = 1 \qquad P(E_2|\overline{H}_2) = 0{,}01$$
$$P(E_2|H_3) = 0{,}8 \qquad P(E_2|\overline{H}_3) = 0{,}02$$
$$P(E_3|H_1) = 0{,}7 \qquad P(E_3|\overline{H}_1) = 0{,}02$$
$$P(E_3|H_2) = 0{,}5 \qquad P(E_3|\overline{H}_2) = 0{,}01$$
$$P(E_3|H_3) = 0{,}4 \qquad P(E_3|\overline{H}_3) = 0{,}05$$

La interpretación de estas probabilidades condicionales en términos estadísticos es muy fácil: las dos de la primera línea vienen a decir «el 95% de los afectados por la gripe tienen fiebre», y «el 1% de todos los pacientes que *no* tienen gripe presentan fiebre».

¿Cómo se diagnosticaría a un paciente que no presentase fiebre, pero sí tos y dolores musculares? Procedamos, como hemos indicado, por pasos. En primer lugar, la ausencia de fiebre nos permite, utilizando la fórmula (2), calcular:

$$P(H_1|\overline{E}_1) = \frac{P(H_1)P(\overline{E}_1|H_1)}{P(H_1)P(\overline{E}_1|H_1) + P(\overline{H}_1)P(\overline{E}_1|\overline{H}_1)} =$$

$$= \frac{0{,}02 * (1 - 0{,}95)}{0{,}02 * (1 - 0{,}95) + (1 - 0{,}02) * (1 - 0{,}01)} = 1{,}02961 * 10^{-3}$$

y, similarmente, $P(H_2|E_1) = 2{,}0783 * 10^{-3}$; $P(H_3|E_1) = 4{,}084 * 10^{-3}$.

Ahora tomamos como nuevos valores de $P(H_j)$ estas probabilidades a posteriori calculadas, y aplicamos la fórmula (1) para tener en cuenta la segunda evidencia (tos), resultando como nuevos valores para $P(H_j)$ los siguientes:

$$P(H_1) = 0{,}03322; \qquad P(H_2) = 0{,}17237; \qquad P(H_3) = 0{,}03956.$$

Y, finalmente, considerando que E_3 también está presente, otra aplicación de la fórmula (1) nos conduce a:

$$P(H_1) = 0{,}78; \qquad P(H_2) = 0{,}91; \qquad P(H_3) = 0{,}25.$$

Es decir, se infiere que la enfermedad más probable es la bronquitis, seguida de cerca por la gripe.

Resumiendo, el conocimiento del experto humano quedaría codificado en la base de conocimientos por las probabilidades a priori de cada una de los posibles resultados, o hipótesis, $P(H_j)$; y por las probabilidades condicionales de cada una de las evidencias elementales a cada una de las hipótesis, $P(E_i|H_j)$ y $P(E_i|\overline{H}_j)$. El sistema preguntará al usuario por algún elemento de evidencia, E_i, y el mecanismo de inferencia calcula las probabilidades a posteriori, $P(H_j|E_i)$ de acuerdo con la fórmula explicada, valores que sustituyen a los previos de $P(H_j)$; luego el sistema preguntará por otro E_i, y así sucesivamente.

¿Cuándo pregunta el sistema por un elemento de evidencia u otro, y cuándo se

detiene el proceso? La idea básica es la siguiente: a la vista de las $P(H_j)$ actualizadas en cada momento, y de todos los valores de $P(E_i|H_j)$ y de $P(E_i|\overline{H}_j)$ el sistema calcula cuál de las E_i tiene más influencia (teniendo en cuenta que la respuesta puede ser positiva o negativa) en las posibles modificaciones de las $P(H_j)$, y pregunta por ella. Pero si las posibles modificaciones son tales que ninguna de las $P(H_j)$ resultantes puede llegar a ser mayor que la $P(H_j)$ que actualmente es máxima, entonces el proceso termina con H_j como hipótesis más probable. Si no se hiciera así, el usuario estaría obligado a dar todas las E_i cada vez que hiciese una consulta.

Hemos dicho al principio que éste es el mecanismo de inferencia básico de PROSPECTOR, y así es, pero con una variante que permite que el experto humano pueda comunicar su conocimiento de forma algo más cómoda que dando probabilidades condicionales. Esta variante utiliza el concepto de *potencialidad* («odds»): si la probabilidad de un suceso es P, su potencialidad es $S = P/(1 - P)$ (por tanto, S será un número comprendido entre 0 y ∞). Sustituyendo P por S en las fórmulas de Bayes (1) y (2) y operando, resulta:

$$S(H_j|E_i) = MS_{ij} * S(H_j) \tag{3}$$

y

$$S(H_j|E_i) = MN_{ij} * S(H_j), \tag{4}$$

con

$$MS_{ij} = P(E_i|H_j)/P(E_i|\overline{H}_j) \quad (\textit{medida de suficiencia})$$
$$MN_{ij} = P(\overline{E}_i|H_j)/P(E_i|\overline{H}_j) \quad (\textit{medida de necesidad})$$

Puede comprobarse (teniendo en cuenta las leyes de las probabilidades) que tiene que cumplirse que si una es mayor o igual que 1 la otra tiene que ser menor o igual que 1; para homogeneizar, se definen las E_i de tal modo que siempre resulte $MS_{ij} \geq 1$ y $MN_{ij} \leq 1$.

Entonces, MS_{ij} estará comprendida entre 1 y ∞, y su interpretación es la siguiente: si $MS_{ij} = 1$, de acuerdo con (3) la potencialidad de que la hipótesis H_j sea cierta, tras saber que se da la evidencia E_i, es la misma que antes de saberlo, por lo que en este caso es indiferente conocer E_i o no, mientras que si MS_{ij} tiende a infinito, la potencialidad a posteriori de H_j tiende también a infinito cualquiera que sea su potencialidad a priori, es decir, E_i es lógicamente suficiente para inferir H_j; por tanto, MS_{ij} es, como su nombre indica, una medida de la suficiencia del conocimiento de E_i para la inferencia de H_j, que el experto humano ha de evaluar entre 1 y ∞.

En cuanto a MN_{ij}, si vale 1, según (4) la ausencia de E_i no afecta a la potencialidad de la hipótesis, es decir, no es en absoluto necesario que E_i esté presente, mientras que si vale 0 la ausencia de E_i reduce a cero la potencialidad de H_j: E_i sería lógicamente necesaria. El experto humano habrá de evaluar, en una escala de 0 a 1 la necesidad de que E_i esté presente para inferir H_j.

En resumen, la base de conocimientos de PROSPECTOR consta de las probabilida-

des a priori de cada hipótesis* y de un conjunto de reglas que se pueden esquematizar así:

$$H_j \rightarrow E_i(MS_{ij}, MN_{ij}),$$

es decir, reglas de producción con dos números asociados, y la inferencia consiste en ir recopilando E_i y actualizando las probabilidades de cada hipótesis.

Hay un aspecto final a considerar: el de la incertidumbre en el usuario cuando se le pregunta por la existencia o no de un determinado elemento de evidencia, E_i. Según (3) y (4), si la respuesta es que E_i está presente (con seguridad), entonces $S(H_j|E_i) = MS_{ij} * S(H_j)$, mientras que si no hay duda de que E_i no está presente, entonces $S(H_j|E_i) = MN_{ij} * S(H_j)$. Pues bien, lo que se hace es permitir al usuario que responda en una escala entre +5 y −5. Una respuesta +5 equivale a una seguridad absoluta en la presencia de E_i, y se aplicaría la primera fórmula. Una respuesta de −5 (seguridad absoluta de que E_i está ausente) conduciría a aplicar la segunda fórmula. Una respuesta 0 (ignorancia total) dejaría inalterada $S(H_j|E_i)$. Y para valores de R intermedios, el sistema hace una interpolación entre MS_{ij}, 1 y MN_{ij}. Si, por ejemplo, se hacen interpolaciones lineales, tendríamos las siguientes fórmulas:

$$\text{si } R > 0,\ M_{ij} = \frac{R * (MS_{ij} - 1) + 5}{5}$$

$$\text{si } R = 0,\ M_{ij} = 1$$

$$\text{si } R < 0,\ M_{ij} = \frac{R * (1 - MN_{ij}) + 5}{5}$$

y actualizaríamos así la potencialidad:

$$S(H_j|E_i) = M_{ij} * S(H_j)$$

3.3. Inferencia mediante factores de certidumbre

Una objeción crítica que puede hacerse al mecanismo explicado en el apartado anterior es que utiliza una base matemática rigurosa (la teoría de la probabilidad) para aplicarla a unas «creencias subjetivas» que no cumplen las leyes de las probabilidades.

Los diseñadores del sistema MYCIN estimaron más oportuno ensayar un enfoque heurístico que, resumidamente, es el siguiente:

---

* En principio, y según lo que hemos dicho, las «hipótesis» corresponderían a la existencia de determinados minerales. Pero en realidad PROSPECTOR es algo más complicado: para cada mineral hay una jerarquía de hipótesis que se combinan formando una red, y en esta red hay también relaciones lógicas imprecisas, para las que se aplican las «leyes de Lukasiewicz» que veíamos en el capítulo 5.

Las reglas de producción se expresan en la forma:

$$E_i \rightarrow H_j(C_{ij})$$

donde C_{ij} es el *factor de certidumbre* de la regla, número comprendido entre -1 y $+1$ que expresa el «grado de confianza» en esa regla: supuesto que E_i sea verdadero, $C_{ij} = +1$ correspondería a una seguridad absoluta de que se deduce H_j, y $C_{ij} = -1$ a una seguridad absoluta de que H_j es falsa ($C_{ij} = 0$ correspondería a una incertidumbre o ignorancia total sobre el asunto).

Por ejemplo, una de las aproximadamente 500 reglas contenidas en la base de conocimientos de MYCIN es la siguiente:

— Si la infección es bacteriemia primaria,
y la toma del material a cultivar es una toma estéril,
y se cree que la puerta de entrada del organismo es el tracto gastrointestinal,
entonces
hay bastante evidencia (0,7) de que la identidad del organismo sea bacteroides.

El número 0,7 que figura en el consecuente es el factor de certidumbre asociado a esa regla.

Los distintos elementos de evidencia pueden también llevar asociados factores de certidumbre (que introducirá el usuario cuando el sistema le pregunte por esos elementos de evidencia).

El mecanismo inferencial de MYCIN consiste en un algoritmo de encadenamiento hacia atrás como el explicado en el apartado 2.4.5, al que se añaden algunos «heurísticos»* para ir combinando los factores de certidumbre y terminar dando una certidumbre final a cada una de las posibles hipótesis. Los heurísticos principales son:

* Si existe la regla $A \rightarrow B(C_R)$ y se ha calculado un factor de certidumbre C_A para A, si $C_A < 0$, la regla no se aplica, y en caso contrario, se asigna a B un factor $C_B = C_A * C_R$.

* En general, A estará compuesto por otros hechos unidos por conectivas, $\vee$, $\wedge$ y $\neg$; el cálculo del factor de certidumbre resultante se hace de acuerdo con las fórmulas

$$C(A1 \vee A2) = \text{máx}(C_{A1}, C_{A2})$$
$$C(A1 \wedge A2) = \text{mín}(C_{A1}, C_{A2})$$
$$C(\neg A) = -C(A)$$

* En los nodos «Y», de acuerdo con los heurísticos anteriores, el C de la regla se multiplica por el menor de los C de las premisas.

* En las técnicas de inteligencia artificial, se llama heurístico a cualquier truco, o regla empírica, que se ha comprobado que sirve de ayuda en la solución de un problema.

* En los nodos «*O*», si sólo hay dos ramas para cuyos elementos se han calculado C_1 y C_2, el factor de certidumbre del resultado es:

$$C = C_1 + C_2 - C_1 * C_2 \qquad \text{si } C_1 * C_2 > 0$$

$$C = \frac{C1 + C2}{1 - \text{mín}(|C_1|, |C_2|)} \qquad \text{si } C1 * C2 < 0$$

Si hay más de dos ramas, se calcula el C para las dos primeras, el resultado se combina con el de la tercera, etc.

* Siempre que como consecuencia de un cálculo resulte $|C| \leq 0{,}2$, se hace $C = 0$. (Esto acelera los algoritmos y hace más claras las explicaciones y justificaciones del sistema).

Para ilustrar con un caso simplificado cómo se aplican estos heurísticos, volvamos a nuestro ejemplo de la fiebre, la gripe, etc. Supongamos que la base de conocimientos está formada por las 9 reglas enumeradas en el apartado 3.2, pero interpretando que lo que allí se llaman «probabilidades» son factores de certidumbre. Es decir:

*R*1: $f \rightarrow g$ (0,5)	*R*4: $ts \rightarrow g$ (0,1)	*R*7: $d \rightarrow g$ (0,7)
*R*2: $f \rightarrow b$ (0,1)	*R*5: $ts \rightarrow b$ (0,7)	*R*8: $d \rightarrow b$ (0,2)
*R*3: $f \rightarrow tb$ (0,4)	*R*6: $ts \rightarrow tb$ (0,2)	*R*9: $d \rightarrow tb$ (0,1)

Y supongamos que la evidencia presente es: fiebre (−0,8), tos (0,9), dolores (1). (Bastante seguro que no tiene fiebre, casi seguro que tiene tos y con seguridad que tiene dolores musculares).

Para cada una de las tres hipótesis (gripe, bronquitis, tuberculosis) tenemos un árbol muy sencillo: un simple nodo «*O*» con tres ramas. Para la primera (gripe) resulta:

Rama 1 (*R*1): −0,8 * 0,5 = −0,4
Rama 2 (*R*4): 0,9 * 0,1 = 0,09
Rama 3 (*R*7): 1 * 0,7 = 0,7

Combinando la Rama 1 con la 2,

$$\frac{-0{,}4 + 0{,}09}{1 - 0{,}09}$$

y combinando este factor de certidumbre con el de la Rama 3,

$$Cg = \frac{0{,}7 - 0{,}34}{1 - 0{,}34} = 0{,}54$$

Procediendo de igual modo con las otras hipótesis se obtiene

$$C_b = 0{,}68 \qquad \text{y} \qquad C_{tb} = -0{,}08,$$

es decir, algo más de certidumbre en que sea bronquitis que gripe, y una incertidumbre prácticamente absoluta en cuanto a que pueda o no ser tuberculosis.

3.4. Inferencia borrosa

Si a PROSPECTOR se le acusa de utilizar una teoría en exceso rigurosa y elaborada con otros fines para formalizar un conocimiento esencialmente impreciso y subjetivo, las objeciones a MYCIN van en sentido opuesto: se basa en unos heurísticos bastante arbitrarios.

Por ello, muchos de los trabajos actuales buscan su fundamento en unas herramientas teóricas que tratan de acometer la imprecisión y la subjetividad de manera rigurosa: la teoría de conjuntos borrosos y la lógica borrosa, de las cuales ya hemos dado alguna explicación en el capítulo 5.

Existen ya varios sistemas diseñados según esta teoría, y muchos otros están en proceso de desarrollo. Pero ninguno de ellos está tan establecido aún como PROSPECTOR y MYCIN (que han servido de patrón para otros muchos desarrollos posteriores). Por esta razón, no entraremos aquí en la descripción de ninguno, remitiendo al lector interesado a las referencias del apartado 6.

4. OTROS ESQUEMAS PARA LA REPRESENTACIÓN DEL CONOCIMIENTO

El objetivo de este capítulo era ver un campo de aplicación de la lógica: los sistemas basados en conocimiento. Pero en muchos de estos sistemas se utilizan otras técnicas más estructuradas para la representación del conocimiento, y parece oportuno, para completar el capítulo, dar una idea de dos de las más conocidas: las redes semánticas y las estructuras («frames»).

Una preocupación común en todas estas técnicas es la de tener una representación lo más estructurada posible a fin de facilitar el almacenamiento, modificación y búsqueda en las bases de conocimiento. Vamos a ver sobre un ejemplo cómo de la representación lógica puede pasarse a otras representaciones más estructuradas.

Supongamos que tenemos los siguientes hechos:

La Memoria Transfiere Datos e Instrucciones al Procesador.
El Procesador Transfiere Datos a la Memoria.
El Procesador Interpreta Instrucciones.
La Memoria Almacena Datos e Instrucciones.

Pensando en lógica de predicados, podemos definir dos predicados ternarios: $T(x, y, z)$ para representar «x transfiere a y, z» y $F(x, y, z)$ para «una función de x es hacer y con z». Nuestros hechos se traducirán entonces a predicados aplicados sobre las constantes M (Memoria), P (Procesador), D (Datos), I (Instrucciones), R (Interpreta) y A (Almacena). (También podríamos utilizar letras minúsculas, si quisiéramos seguir al pie de la letra los convenios del capítulo 4 para los nombres de las constantes).

La representación en lógica de predicados sería:

$$T(M, P, D); \quad T(P, M, D); \quad T(M, P, I);$$
$$F(P, R, I); \quad F(M, A, I); \quad F(M, A, D).$$

Esta base de hechos es pequeña, pero en un caso real podría contener cientos de predicados, y para mejorar el acceso convendría estructurarla. Por ejemplo, podríamos agrupar separadamente (aun cuando aparezcan repeticiones) los hechos referentes a la memoria y los referentes al procesador.

Memoria:

$$T(M, P, D);\ T(P, M, D);\ T(M, P, I);\ F(M, A, I);\ F(M, A, D)$$

Procesador:

$$T(M, P, D);\ T(P, M, D);\ T(M, P, I);\ F(P, R, I)$$

Ahora los hechos están indexados por objetos del dominio del discurso. Se dice que es una representación *centrada en los objetos*, en este caso, en los objetos físicos. También podemos adoptar una representación centrada en objetos abstractos: transferencias y funciones.

Todos los predicados de este ejemplo son ternarios. Pero en las representaciones estructuradas es preferible trabajar sólo con predicados binarios. Una de las razones para ello es que si se quiere perfeccionar el conocimiento diciendo, por ejemplo, que la transferencia de datos de la memoria al procesador se hace a través del bus de datos, ello exigiría convertir el predicado ternario en otro cuaternario, y habría que modificar también los procedimientos de inferencia. Vamos a ver que es posible expresar todo con predicados binarios y conseguir un sistema más modular y fácil de actualizar.

Definamos, para nuestro ejemplo, dos conjuntos, {transferencias} y {funciones}. Todo lo que se dice sobre transferencias y funciones puede expresarse mediante predicados binarios que relacionan a los argumentos de los predicados ternarios originales con un elemento arbitrario de esos conjuntos. Por ejemplo, la fórmula atómica $T(M, P, D)$ se transformará en la sentencia

$$(\exists x)(\text{Pertenece}(x, \{\text{transferencias}\}) \wedge \text{Fuente}(x, M) \wedge$$
$$\wedge\ \text{Destino}(x, P) \wedge \text{Objeto}(x, D)$$

y la $F(P, R, I)$, en

$$(\exists x)(\text{Pertenece}(x, \{\text{funciones}\}) \wedge \text{Unidad}(x, P) \wedge$$
$$\wedge\ \text{Función}(x, R) \wedge \text{Objeto}(x, D)$$

Los cuantificadores existenciales pueden eliminarse creando constantes de Skolem: $T1$ en sustitución de x en la primera sentencia, y $F1$ en la segunda. Por otra parte,

los predicados binarios que relacionan los argumentos originales con algún elemento arbitrario de los conjuntos también pueden expresarse como funciones definidas sobre esos conjuntos. Es decir, en vez de decir, por ejemplo, «Fuente(x, M)», podemos definir la función «fuente» y decir que «fuente(x) = M». (Obsérvese el cambio de notación, pero no de contenido semántico: «Fuente» es un predicado; Fuente($T1$, M) se evalúa como verdadero o falso; «fuente» es una función; fuente($T1$) es un individuo; «=» es un predicado binario, de modo que también habríamos podido escribir la igualdad en la forma Igual(fuente($T1$), M), y que se evalúa como verdadero o falso).

Haciendo esas dos transformaciones, las sentencias anteriores se escribirán así:

$$\text{Pertenece}(T1, \{\text{transferencias}\}) \wedge (\text{fuente}(T1) = M) \wedge \\ \wedge (\text{destino}(T1) = M) \wedge (\text{objeto}(T1) = D)$$

$$\text{Pertenece}(F1, \{\text{funciones}\}) \wedge (\text{unidad}(F1) = P) \wedge \\ \wedge (\text{función}(F1) = R) \wedge \text{objeto}(F1) = D)$$

Esta representación puede parecer, de momento, bastante más farragosa que la que inicialmente teníamos, pero obsérvense dos cosas: que hemos limitado a dos los predicados y que éstos son binarios (si bien a costa de introducir arbitrariamente funciones) y que si, como decíamos antes, se quieren ampliar las relaciones con nuevos elementos del dominio del discurso (por ejemplo, buses) basta con definir nuevas funciones sobre los conjuntos de base y seguir utilizando los mismos predicados binarios «Pertenece» e «Igual».

Poniendo en grupos separados todos los hechos que identifican a $T1$, $T2$, etc., tendremos una representación más estructurada:

$T1$ Pertenece($T1$, {transferencias})
fuente($T1$) = M
destino($T1$) = P
objeto($T1$) = D

etcétera.

Como ahora todas las funciones figuran dentro de un grupo identificado por su argumento, no hace falta especificar éste, y escribiremos «fuente: M», etc. Además, para abreviar el predicado «Pertenece($T1$, {transferencias}) escribiremos «tipo: transferencias». Llegamos así a la representación estructurada:

$T1$ tipo: transferencias
fuente: M
destino: P
objeto: D

$T2$ tipo: transferencias
fuente: P
destino: M
objeto: D

*T*3 tipo: transferencias
fuente: *M*
destino: *P*
objeto: *I*

*F*1 tipo: funciones
unidad: *P*
función: *R*
objeto: *I*

*F*2 tipo: funciones
unidad: *M*
función: *A*
objeto: *I*

*F*3 tipo: funciones
unidad: *M*
función: *A*
objeto: *D*

Se dice que *T*1, *T*2 y *T*3 son *casos* («instances») de la *estructura* («frame») general *T*, y *F*1, *F*2 y *F*3 lo son de la *F*. Cada estructura tiene como componentes *ranuras* («slots») de la forma nombre_ranura: valor_ranura.

Hay informaciones implícitas sobre pertenencia a conjuntos en nuestra representación original que podemos añadir ahora:

M tipo: unidades

P tipo: unidades

R tipo: funciones

A tipo: funciones

D tipo: informaciones

I tipo: informaciones

Una *red semántica* es un grafo en el que los nodos pueden representar objetos, conceptos o conjuntos y los arcos relaciones entre ellos. Para nuestro ejemplo, la red semántica podría ser la que indica la figura 6.2.

Aquí sólo hemos introducido la cuestión de las representaciones estructuradas. En el ejemplo, lo que hemos representado es conocimiento declarativo. Las estructuras hay que ampliarlas con nuevos conceptos para representar el conocimiento procedimental, y hay que utilizar métodos inferenciales especiales que operen sobre esas estructuras. De nuevo remitimos al lector interesado a la bibliografía sobre el tema.

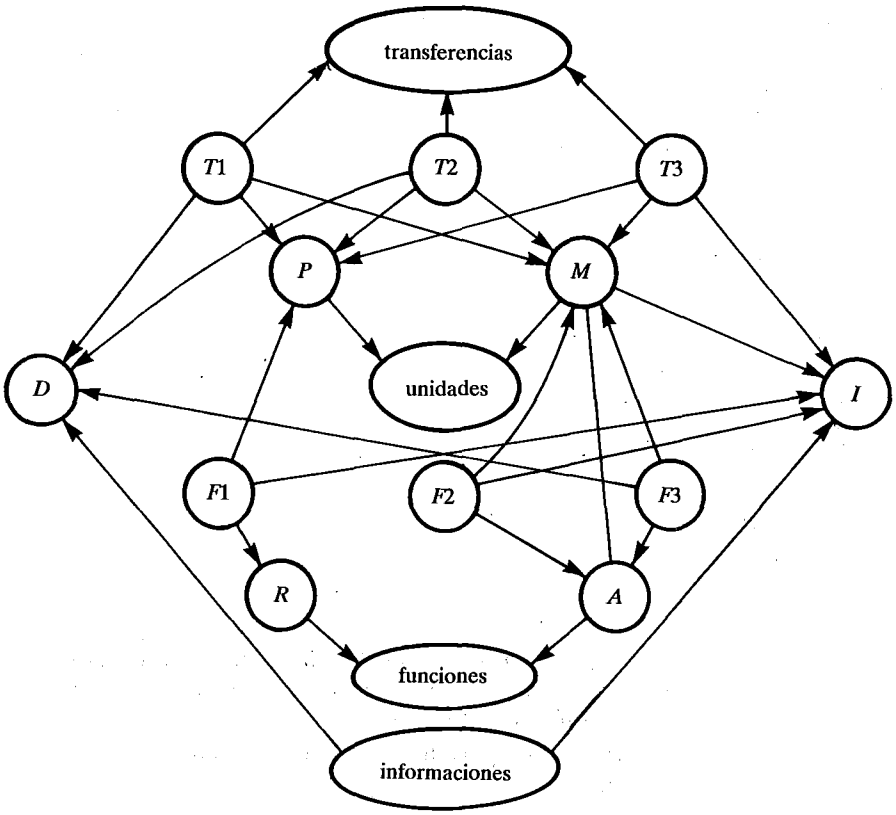


FIGURA 6.2.

5. RESUMEN

La lógica formal es una herramienta adecuada para la representación del conocimiento declarativo, procedimental y de control y, por tanto, para el diseño de sistemas basados en conocimiento y sistemas expertos.

Uno de los modelos más utilizados para el diseño de sistemas basados en conocimiento es el de los *sistemas de producción*. Las reglas de producción en estos sistemas pueden formalizarse como sentencias condicionales, y, por tanto, todo lo que la lógica nos enseña sobre sistemas inferenciales es aquí aplicable.

Los expertos humanos suelen trabajar con reglas no muy bien definidas y con elementos de evidencia imprecisos o inciertos. Para diseñar un sistema experto es preciso modelar esos aspectos de la actividad humana, y para ello existen varias técnicas: factores de certidumbre, probabilidades, lógica borrosa, etc.

Hay otras técnicas para representar el conocimiento de una manera más estructurada que con la lógica. Las más conocidas son las *estructuras* («frames») y las *redes semánticas*.

6. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

Post (1943) fue el primero que, desde un punto de vista teórico, propuso el modelo de «sistema de producción» como un mecanismo computacional general.

El «paradigma» de los sistemas basados en conocimiento comenzó a tomar cuerpo a finales de los años 60. Suelen citarse como «precursores» los programas DENDRAL y MACSYMA. El primero, desarrollado en la Universidad de Stanford, permite deducir la estructura química molecular de un compuesto orgánico a partir de su fórmula y de datos espectrográficos y de resonancia magnética nuclear (Buchanan *et al.*, 1969), y actualmente lo utilizan varias empresas farmacéuticas americanas. MACSYMA, del M.I.T., realiza cálculo diferencial e integral mediante manipulación simbólica de expresiones algebraicas (Martin y Fateman, 1971), y es también un producto comercializado. Los sistemas expertos «pioneros» más conocidos son MYCIN, para diagnóstico y tratamiento de enfermedades infecciosas (Shortliffe, 1976) e INTERNIST, para diagnóstico en Medicina Interna (Pople *et al.*, 1975). Actualmente puede hablarse de una «etapa de industrialización», en la que empresas ya establecidas se interesan por este nuevo campo, se crean otras dedicadas exclusivamente a él, y aparecen en el mercado herramientas software para el desarrollo de sistemas expertos.

Las redes semánticas se propusieron inicialmente como modelo de la memoria humana (Quillian, 1968), y se han utilizado en los sistemas expertos PROSPECTOR (Duda *et al.*, 1978), CASNET (Weiss *et al.*, 1978), CADUCEUS (Myers *et al.*, 1982), IRIS (Trigoboff y Kulikowski, 1981), etc., y las estructuras (frames) fueron introducidas por Minsky (1975) con idea parecida («cuando a la mente se le plantea una situación nueva, busca en la memoria alguna estructura estereotipada de información»), y en ellas se han basado otros diseños, como los sistemas PIP (Pauker *et al.*, 1976), RADEX (Chandrasekaran *et al.*, 1980), CENTAUR (Aikins, 1983), etc., y otras representaciones estructuradas, como las *escenas* (scripts), de Schank y Abelson (1977).

Hayes (1977, 1979) puso de manifiesto las relaciones entre las representaciones en lógica de predicados y mediante estructuras. En la presentación de estas relaciones en el apartado 4 hemos seguido a Nilsson (1982).

En realidad, la ingeniería del conocimiento está aún en un estado incipiente, y en cada diseño se adopta un esquema híbrido buscando la mejor solución posible al problema concreto. Por ejemplo, PROSPECTOR no sólo utiliza, como hemos visto, un mecanismo de inferencia bayesiano y, en otras partes, una lógica multivalorada, sino también una red inferencial que guarda cierta relación con las redes semánticas de Quillian.

El lector interesado en conocer más sobre estas técnicas de representación del conocimiento puede consultar el libro ya citado de Nilsson (1982), o el de Charniak y McDermott (1985), donde encontrará explicaciones muy claras tanto sobre las técnicas como sobre los procedimientos de inferencia asociados, y, para profundizar en el asunto, MacCalla y Cercone (1983) o Sowa (1984).

Sobre el tema de los sistemas basados en conocimiento, y a un nivel introductorio, pueden encontrarse informaciones adicionales a lo que aquí hemos expuesto en los artículos de Cuena (1984, 1985a, b), Cordier (1984), Hayes-Roth (1894a, b, c) y Nau

(1982). El libro de Gondran (1984) contiene varios ejemplos ilustrativos de pequeños sistemas, y en el de Naylor (1983) pueden encontrarse listados en BASIC para Apple II y Spectrum de sistemas muy sencillos (uno de ellos introduce al interesante tema del aprendizaje por inducción a partir de ejemplos).

Para mayor profundidad y detalle, pueden verse los textos de Hayes-Roth *et al.* (1983) (que contiene estudios comparativos sobre diversas herramientas), Buchanan y Shortliffe (1984) (centrado sobre MYCIN, EMYCIN y demás desarrollos del «Stanford Heuristic Programming Project»), Weiss y Kulikowski (1984) (que ofrece una visión complementaria del anterior, al estar más influenciado por la experiencia de sus autores en los desarrollos de CASNET y EXPERT), o, en español, el de Cuena *et al.* (1986) (que contiene los temas de aprendizaje y de inferencia imprecisa), el de Alty y Coombs (1986) (donde se resumen los conceptos básicos y se estudian con algún detalle varios sistemas expertos) y el coordinado por Mompín (1987) (escrito por varios autores, y que cubre aspectos básicos y de aplicación).

La lógica borrosa no es la única herramienta formal que se investiga actualmente para diseñar sistemas que trabajen con imprecisión o incertidumbre. Otra es, por ejemplo, la «teoría de la evidencia», de Dempster-Shaffer (Shaffer, 1976). Una síntesis de este tema puede encontrarse en un artículo de Prade (1985), o, con mayor extensión, en el libro de Dubois y Prade (1985).

Segunda parte
AUTOMATAS

Capítulo 1
IDEAS GENERALES

1. AUTÓMATAS E INFORMACIÓN

La palabra «autómata», en el lenguaje ordinario, normalmente evoca algo que pretende imitar funciones propias de los seres vivos, especialmente las relacionadas con el movimiento. (Véase, para corroborar esta aserción, la definición de un diccionario cualquiera). En este sentido, un ejemplo de autómata sería el típico robot antropomorfo o zoomorfo dotado de capacidades autónomas de movimiento que le permiten ejecutar las órdenes o seguir el programa establecido por un ser inteligente.

En el campo de la Informática lo fundamental no es la simulación del movimiento, sino la simulación de los procesos de tratar la información, y el ejemplo típico de autómata no es ya el robot mecánico sino el ordenador.

Pensemos en la naturaleza del trabajo que realizan los ordenadores. Bastan unos conocimientos básicos de informática para llegar a la conclusión de que un ordenador no es más que un dispositivo que manipula símbolos. Un ejemplo será útil para reforzar esta idea:

Consideramos un ordenador con registros de 16 bits. Supongamos que este ordenador recibe de un periférico una serie de impulsos que se graban en un registro dejando en él la siguiente configuración de bits:

1010010001000001

¿Qué significa ésto para el ordenador? Entre otras muchas cosas puede ser:

— El número −23487 expresado en binario con convenio de complemento a 2.
— Los caracteres «ñ, ü» codificados en código ASCII ampliado a 8 bits.
— Una instrucción del lenguaje de máquina del ordenador.

El que sea una u otra cosa depende de dos personas:

— el diseñador del ordenador (para ser realista habría que hablar del equipo de diseño), que decidió que los números se representen en binario y complemento a 2, o que, interpretado como instrucción, ese código de operación signifique, por ejemplo, «sumar», y no otra cosa, etc.;
— el utilizador, que, al hacer su programa, decide que en un momento dado el ordenador lleve esa configuración de bits de la memoria al acumulador, o al registro de instrucción, o a la unidad de salida, etc. Al tomar tal decisión, el utilizador está dotando al conjunto de bits de una significación y, por consiguiente, de una capacidad para representar información. Para el ordenador, sin embargo, los bits no son más que símbolos materializados por los niveles de tensión en los circuitos.

A veces se define al ordenador por sus supuestas capacidades para «tratar» o «procesar» automáticamente la información. De acuerdo con lo visto más arriba, debe entenderse que este tratamiento o procesamiento de la información sólo tiene sentido para nosotros, que decidimos qué tal «tira» o «cadena» de símbolos significa tal cosa, es decir, que *codificamos la información en cadenas de símbolos*; el ordenador se limita a manipular esas cadenas, dando normalmente como resultado otras cadenas que nosotros decodificamos.

Y hablando ya en términos generales, podemos considerar a un autómata como un dispositivo que manipula cadenas de símbolos que se le presentan a su entrada, produciendo otras tiras o cadenas de símbolos como salida. En el apartado 3.1 del capítulo 2 formalizaremos matemáticamente esta definición.

Un ordenador es un ejemplo de autómata, pero también son autómatas dispositivos más sencillos, como sumadores, contadores, etc., que veremos como ejemplos en el capítulo 2, o las mismas partes constituyentes de un ordenador: la unidad aritmética-lógica o la unidad de control son autómatas. Por otra parte, existen autómatas más complejos que un ordenador, como los robots controlados por ordenador. También pueden estudiarse como autómatas determinadas funciones de los seres vivos, e incluso complejos sistemas ecológicos y socioeconómicos.

2. AUTÓMATAS Y MÁQUINAS SECUENCIALES. CONCEPTO DE ESTADO

El autómata recibe los símbolos de entrada uno detrás de otro, distribuidos en el tiempo, es decir, secuencialmente. Además, en general, el símbolo de salida que en un instante determinado produce este autómata no sólo depende del último símbolo recibido a la entrada, sino de toda la secuencia o cadena, distribuida en el tiempo, que ha recibido hasta ese instante. Quiere esto decir que un autómata es una máquina secuencial, en el sentido de que opera sobre secuencias de símbolos, «recordando» en todo instante la «historia» de símbolos llegados hasta ese instante. Esto le diferencia de una máquina puramente combinatoria como sería, por ejemplo, un circuito lógico de los estudiados en el tema «Lógica».

Así pues, ante un determinado símbolo de entrada un autómata puede producir

diferentes símbolos de salida, dependiendo de la historia o secuencia de todos los símbolos de entrada anteriores.

Obsérvese que hasta ahora venimos hablando de un autómata como una «caja negra» con una entrada en la que recibe símbolos y una salida, sobre la que deposita otros símbolos. Otra manera de enfocar el estudio de los autómatas es considerando lo que hay dentro de la caja negra (aunque sea de una manera abstracta, es decir, prescindiendo de la naturaleza de los componentes físicos y atendiendo sólo a las transformaciones de símbolos), y esto nos lleva a definir un concepto fundamental: el estado del autómata. *El estado es toda la información necesaria en un momento dado para poder deducir, dado un símbolo de entrada en ese momento, cual será el símbolo de salida.* Es decir, conocer el estado es lo mismo que conocer toda la historia de símbolos de entrada*. Un autómata tendrá un determinado número de estados (en teoría, puede tener infinitos), y se encontrará en uno u otro según sea la historia de símbolos que le han llegado; si encontrándose en un estado determinado, recibe un símbolo también determinado, producirá un símbolo de salida y efectuará un cambio o *transición* a otro estado (también puede quedarse en el mismo). Estas ideas son fáciles de formalizar matemáticamente a partir de los conceptos de conjunto y función, y conducen a la definición de autómata que desarrollaremos en el capítulo siguiente.

3. AUTÓMATAS Y LENGUAJES

Un campo importante dentro de la Informática, al que dedicaremos el último tema, está constituido por el estudio de los lenguajes y las gramáticas que los generan. Los elementos de un lenguaje son sentencias, palabras, etc., formados a partir de un *alfabeto* (capítulo 1, apartado 4 del tema «Lógica»). Establecidas unas reglas gramaticales, una cadena de símbolos pertenecerá al correspondiente lenguaje si tal cadena se ha formado obedeciendo esas reglas; puede entonces pensarse en la posibilidad de construir un *autómata reconocedor* de ese lenguaje, tal que cuando reciba a su entrada una determinada secuencia de símbolos produzca, por ejemplo, un «1» a la salida si la secuencia es correcta, y un «0» si no lo es. De este modo, como veremos en su momento, a cada tipo de gramática corresponde un tipo de autómata.

4. AUTÓMATAS Y ÁLGEBRA

Las cadenas de entrada y salida de un autómata se forman a partir de los correspondientes alfabetos mediante una operación que consiste en poner los símbolos unos a continuación de otros. Esta operación se llama concatenación, y es asociativa. Por consiguiente, el conjunto de todas las cadenas con la concatenación

* Realmente, no basta con conocer toda la historia de símbolos de entrada para saber cuál es la salida; es necesario conocer también el «estado inicial», es decir, el estado en que se encontraba el autómata al recibir el primero de los símbolos de entrada.

tiene una estructura algebraica de semigrupo; si, además, definimos un elemento neutro, tendremos un monoide.

Por otra parte, como veremos en el capítulo siguiente, si el autómata es finito (es decir, si tiene un número finito de estados) puede determinarse un número finito de clases de equivalencia en el semigrupo (o monoide) de entrada, lo cual permite definir un semigrupo (o monoide) cociente llamado el semigrupo (o monoide) de la máquina, a partir del cual pueden formalizarse muchas cuestiones relativas al funcionamiento de los autómatas.

Siguiendo esta línea de trabajo, se ha elaborado en las dos últimas décadas una teoría abstracta de autómatas con una fuerte base algebraica que, según Arbib (1969), constituye «la matemática pura de la Informática».

5. RESUMEN

La Teoría de Autómatas, también llamada Teoría algebraica de máquinas, permite estudiar de un modo sistemático las máquinas, más o menos complicadas, que realizan un procesamiento de la información y que actúan de manera discreta, es decir, la información se supone codificada a partir de un conjunto finito de símbolos que el autómata trata secuencialmente, uno detrás de otro. La Teoría de Autómatas proporciona métodos para el análisis y la síntesis de tales máquinas.

Los trabajos sobre lenguajes y gramáticas formales han evolucionado en una dirección que les ha conducido a encontrarse con la Teoría de Autómatas como herramienta matemática de gran utilidad.

La Teoría de Autómatas no sólo puede aplicarse a las «máquinas», en el sentido estricto que normalmente damos a esta palabra, sino también a muchos sistemas naturales, y, en general, permite estudiar procesos que dependen de una «historia», es decir, cuyo comportamiento presente es función del pasado.

En sus veinte años de historia la Teoría de Autómatas se ha constituido en una disciplina muy formalizada que sigue en evolución. Mientras la teoría básica (autómatas finitos deterministas) puede considerarse definitivamente establecida, se abren nuevas vías que actualmente son objeto de estudio de los investigadores y que ofrecen amplias perspectivas de aplicación: autómatas estocásticos, borrosos, adaptativos, de aprendizaje, etc.

Evidentemente, en este tema no podemos exponer ni siquiera resumir, toda la Teoría de Autómatas. Nuestro objetivo será presentar los principios básicos, desarrollando algunos ejemplos de aplicación para ver su utilidad práctica en diversos campos, especialmente el de la Informática.

Capítulo 2
AUTOMATAS FINITOS

1. DEFINICIÓN Y REPRESENTACIÓN DE LOS AUTÓMATAS

1.1. Definición

Un *autómata* es una quíntupla:

$$A = \langle E, S, Q, f, g \rangle, \qquad [1.1.1]$$

donde:

- E es un conjunto finito, llamado *conjunto de entradas* o *alfabeto de entrada*, cuyos elementos llamaremos *entradas* o *símbolos de entrada*.
- S es un conjunto finito, llamado *conjunto de salidas* o *alfabeto de salida*, cuyos elementos llamaremos *salidas* o *símbolos de salida*.
- Q es un conjunto llamado *conjunto de estados*.
- f es una función f: $E \times Q \rightarrow Q$, llamada función de *transición* o *función de estado siguiente*.
- g es una función g: $E \times Q \rightarrow S$, llamada *función de salida*.

Esta definición formal puede interpretarse como la descripción matemática de una máquina que, si en el instante t recibe una entrada $e \in E$ y se encuentra en el estado $q \in Q$, entonces da una salida $g(e, q)$, y pasa al estado $f(e, q)$ en el instante $t + 1$. (Suponemos una escala discreta de tiempos arbitraria: $t = 1, 2, 3...$). Expresando de una manera explícita el tiempo, y si llamamos s a un elemento genérico de S, podemos escribir:

$$q(t + 1) = f[e(t), q(t)]; \; s(t) = g[e(t), q(t)]$$

A es un autómata finito si Q es un conjunto finito. En lo sucesivo, y hasta el capítulo 5, trataremos sólo con autómatas finitos, y abreviaremos escribiendo «AF».

1.2. Representación

1.2.1. *Tabla de transiciones*

Las funciones f y g pueden representarse mediante una tabla con tantas filas como estados y tantas columnas como entradas. Si la fila i corresponde al estado q_i y la columna j corresponde a la entrada e_j, en la intersección de ambas se escribirá $f(e_j, q_i)/g(e_j, q_i)$. Por ejemplo, sea el AF definido por los conjuntos

$$E = \{a, b\}$$
$$S = \{0, 1\}$$
$$Q = \{q_1, q_2, q_3\}$$

y las funciones de estado y de salida

$$f(a, q_1) = q_1; \qquad g(a, q_1) = 0$$
$$f(b, q_1) = q_2; \qquad g(b, q_1) = 1$$
$$f(a, q_2) = q_3; \qquad g(a, q_2) = 0$$
$$f(b, q_2) = q_2; \qquad g(b, q_2) = 0$$
$$f(a, q_3) = q_3; \qquad g(a, q_3) = 1$$
$$f(b, q_3) = q_1; \qquad g(b, q_3) = 0$$

En lugar de esto, es más cómodo representar f y g por la tabla de transiciones de la figura 2.1.

e / q	a	b
q_1	$q_1/0$	$q_2/1$
q_2	$q_3/0$	$q_2/0$
q_3	$q_3/1$	$q_1/0$

FIGURA 2.1.

1.2.2. *Diagrama de Moore*

Otra forma de representar las funciones f y g es mediante un grafo orientado en el que cada nodo corresponde a un estado, y si $f(e, q_i) = q_j$ y $g(e, q_i) = s$, existe un arco dirigido del nodo correspondiente a q_i al correspondiente a q_j, sobre el que pondremos la etiqueta *e/s*. Por ejemplo, el AF definido por la tabla anterior puede representarse por el grafo de la figura 2.2. Este grafo suele llamarse diagrama de transiciones o diagrama de Moore.

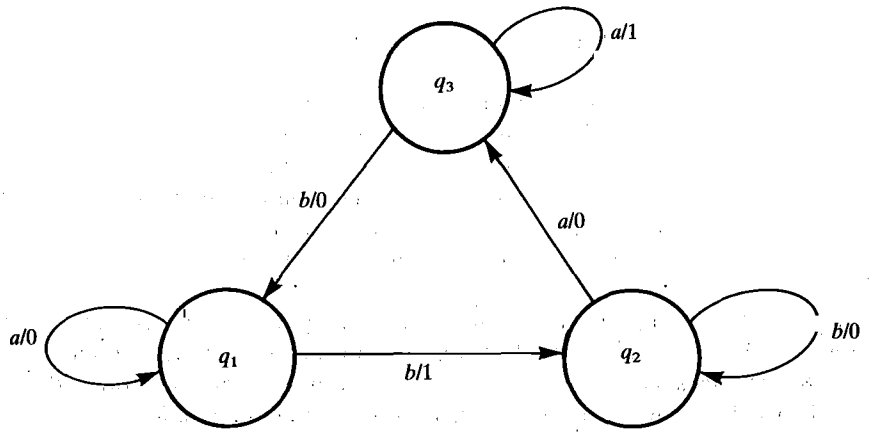


FIGURA 2.2.

1.3. Máquinas de Moore y de Mealy

El modelo general de autómata que hemos definido se llama *máquina de Mealy*. Las funciones f y g determinan la salida y el estado siguiente cuando la máquina se encuentra en un estado $q \in Q$ y recibe una entrada $e \in E$. Ahora bien, por conveniencia matemática, es interesante considerar, además de los símbolos o elementos de E, un elemento neutro, λ; físicamente, el decir que la entrada es λ, es lo mismo que decir que no hay ninguna entrada. Es inmediato entonces plantearse la siguiente pregunta: ¿qué ocurre si, estando un autómata en el estado $q \in Q$, recibe como entrada λ? Para responder a esto, matemáticamente, habría que ampliar el dominio de f, que es $E \times Q$, a $\{E \cup \{\lambda\}\} \times Q$, y lo mismo el dominio de g. La ampliación del dominio de f no plantea ningún problema: se puede convenir que $f(\lambda, q) = q$ (es decir, físicamente, que si no hay entrada no se cambia de estado). Pero no ocurre lo mismo con g; y ello se ve fácilmente si nos referimos al ejemplo desarrollado más arriba (figura 2.1): si llegamos a q_1 ya sea de q_3 (por efecto de entrada b) o de q_1 (por a) la salida es 0, por lo que podemos asociar la salida 0 al estado q_1 y decir $g(\lambda, q_1) = 0$; sin embargo, no podemos definir $g(\lambda, q_2)$, ya que si llegamos a q_2 desde

q_1 la salida es 1, mientras que si llegamos desde el propio q_2 la salida es 0. Es evidente que, en general, sólo puede definirse $g(\lambda, q)$ en el caso en que se cumpla que

$$[q = f(e_1, q_1) = f(e_2, q_2)] \rightarrow [g(e_1, q_1) = g(e_2, q_2)] \qquad [1.3.1]$$

es decir, que a q se le pueda asociar una salida y una sola. Si esto ocurre para todo $q \in Q$ podemos definir una función inyectiva h: $Q \rightarrow S$ tal que $g(e, q) = h[f(e, q)]$, $e \in \{E \cup \{\lambda\}\}$, $q \in Q$. En este caso, podemos decir que la salida sólo depende del estado, y el autómata se llama *máquina de Moore*. Expresando el tiempo de manera explícita:

$$s(t) = g[e(t), q(t)] = h[q(t)] = h[f[e(t-1), q(t-1)]]$$

En una máquina de Mealy las salidas están asociadas a las transiciones, mientras que en una máquina de Moore las salidas están asociadas a los estados, o, lo que es lo mismo, todas las transiciones que conducen a un mismo estado tienen asociada la misma salida. También podemos decir que una máquina de Mealy, en el instante de efectuar una transición necesita conocer una entrada $e \in E$ (ya que, en general, $g(\lambda, q)$ no está definida), mientras que en una máquina de Moore la entrada puede ser $e \in E$ o $e = \lambda$.

Puesto que toda máquina de Moore es una máquina de Mealy que cumple la condición [1.3.1] para todo $q \in Q$, parece en principio que las primeras son un subconjunto de las segundas. Sin embargo, vamos a demostrar que, dada una máquina de Mealy, siempre podremos encontrar una máquina de Moore equivalente (normalmente, a costa de aumentar el número de estados). En efecto, si tenemos una máquina de Mealy

$$A = \langle E, S, Q, f, g \rangle,$$

siempre podemos definir un nuevo autómata

$$\hat{A} = \langle E, S, \hat{Q}, \hat{f}, \hat{g} \rangle,$$

en el que $\hat{Q}$ se obtiene escindiendo cada $q \in Q$ en tantos estados q^s * como salidas s puedan asociarse a q:

$$\hat{Q} = \{q^s | \exists (q' \in Q \text{ y } e \in E) \text{ tales que } f(e, q') = q, \text{ y } g(e, q') = s\}$$

y en el que $\hat{f}$ y $\hat{g}$ se definen así:

$$\hat{f}(e, q^s) = [f(e, q)]^{g(e,q)}$$
$$\hat{g}(e, q^s) = g(e, q)$$

---

* s aquí es un superíndice, no un exponente.

De este modo, a cada $q^s \in \hat{Q}$ se le puede asociar una sola salida, s, y así tendremos una función de salida $\hat{h}$: $\hat{Q} \to S$ tal que $\hat{g}(e, q^s) = \hat{h}[\hat{f}(e, q^s)]$, por lo que $\hat{A}$ será una máquina de Moore.

Concretemos estas ideas con un ejemplo. Tomemos el autómata cuyo diagrama es el de la figura 2.2. Como ya hemos visto, con q_1 siempre se puede asociar la salida 0; sin embargo, q_2 lo escindiremos en q_2^0 y q_2^1, ya que la salida asociada es 0 ó 1, según que vengamos de q_2 o de q_1, y, del mismo modo, escindiremos q_3 en q_3^0 y q_3^1. De acuerdo con esto, y teniendo en cuenta las definiciones de f y g obtenemos el diagrama de la figura 2.3.

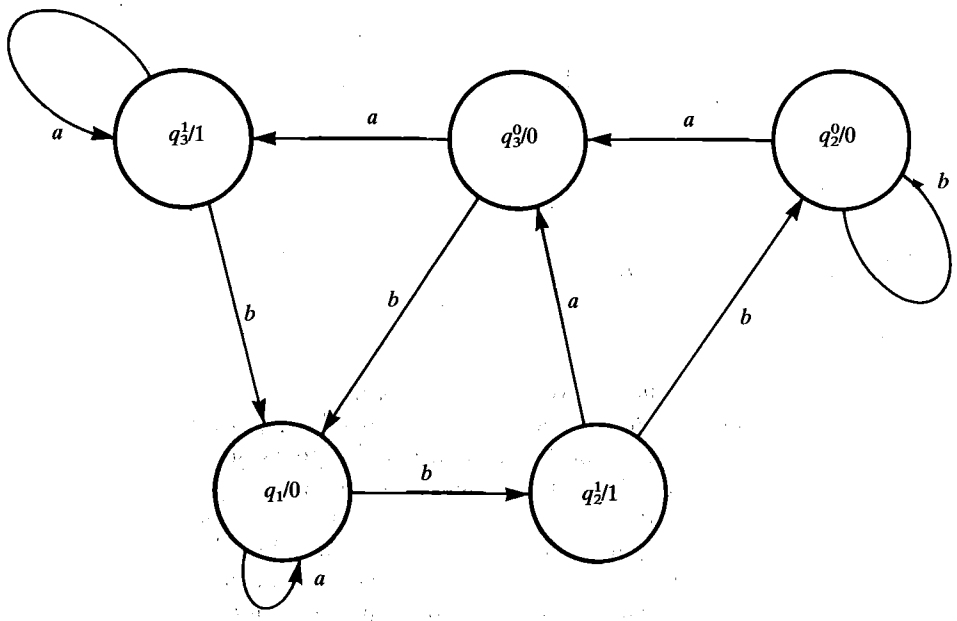


FIGURA 2.3.

Obsérvese que, al estar las salidas asociadas con los estados, todas las transiciones que conducen a un estado producen la misma salida, por lo que en lugar de rotular las salidas sobre los arcos las hemos incluido en los nodos. Del mismo modo, en la tabla de transiciones podemos incluir las salidas en la misma columna de estados; la tabla de esta máquina de Moore es entonces la de la figura 2.4.

En una máquina de Moore podemos considerar el monoide $\langle E^* \rangle$, al que se llama *monoide libre de entrada*. En una máquina de Mealy sólo podemos hablar del *semigrupo libre de entrada*, $\langle E^+ \rangle$.

En lo sucesivo siempre que hablemos de un autómata supondremos, a menos que se diga lo contrario, que se trata de una máquina de Moore, es decir, representaremos indistintamente la salida por la función de salida $g(e, q)$ o por la función de salida $h(q)$ teniendo en cuenta que $g = h \circ f$.

e / q/s	a	b
$q_1/0$	q_1	q_2^1
$q_2^0/0$	q_3^0	q_2^0
$q_2^1/1$	q_3^0	q_2^0
$q_3^0/0$	q_3^1	q_1
$q_3^1/1$	q_3^1	q_1

FIGURA 2.4.

2. EJEMPLOS DE AUTÓMATAS COMO MODELOS

2.1. Detector de paridad

Un procedimiento sencillo y muy utilizado para detectar errores en una transmisión digital (por ejemplo, en una transferencia de datos de un periférico remoto al bus de datos), consiste en enviar un bit de paridad. Este bit puede ser tal que haga par el número total de «unos» enviados (paridad par), o que lo haga impar (paridad impar). Por ejemplo, supongamos que el periférico envía caracteres codificados en código ASCII de 8 bits con paridad par (es decir, el código es ASCII de 7 bits, y el octavo bit es el de paridad). El carácter «A», en ASCII de 7 bits, se codifica 1000001; luego en 8 bits será 01000001 (el bit de paridad se hace 0 para que el número total de «unos» sea par). Por el contrario, el código de «C» es 1000011, por lo que el bit de paridad deberá ser 1 y por consiguiente en 8 bits será 11000011.

En el punto emisor deberá existir un *generador de paridad*, y en el receptor, un *detector de paridad*. Este detector deberá dar una señal de error en el caso de que la paridad recibida no sea correcta, cosa que ocurrirá cuando en la transmisión haya habido una alteración en un bit (o en un número impar de bits). Si la paridad es correcta no dará error. (Obsérvese que si hay un número par de alteraciones en la transmisión este sistema no detecta el error, pero la probabilidad de que ocurra más de una alteración es muy pequeña. Existen, desde luego, otros procedimientos mejores de detección e incluso corrección de errores).

El alfabeto de entrada del detector es, evidentemente, $E = \{0, 1\}$. El alfabeto de salida constará de dos elementos («error» y «no error»); podemos tomar el convenio de que sea también $S = \{0, 1\}$, donde «0» significa «no error» y «1» significa «error». El conjunto de estados puede ser $Q = \{q_0, q_1, q_2\}$, donde q_0 es el estado inicial, del que sólo se sale al recibir el primer bit; en q_1 se estará si se ha recibido un número par de bits y en q_2 si se ha recibido un número impar, de manera que, al finalizar la

transmisión, si la máquina se ha quedado en q_1 es que no ha habido error ($s = 0$), y si se ha quedado en q_2 es que sí lo ha habido ($s = 1$). De acuerdo con esto, es fácil establecer el diagrama de Moore de la figura 2.5.

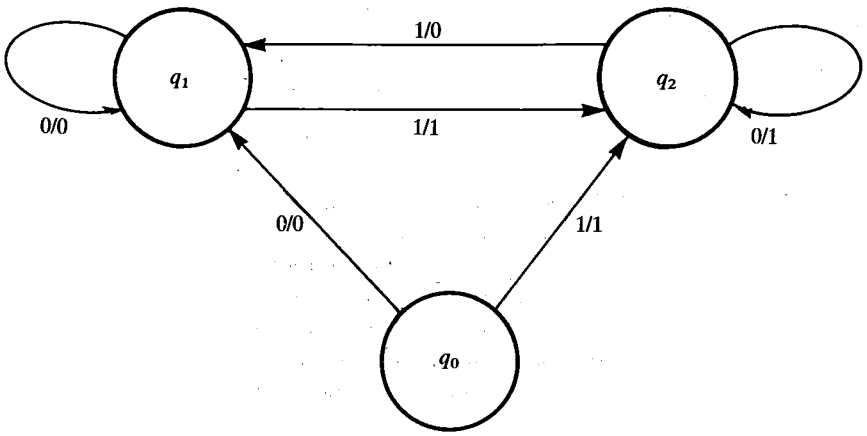


FIGURA 2.5.

Ahora bien, el estado q_0 puede fundirse con el q_1. Ello equivale a convenir en que inicialmente, cuando no se ha recibido ningún bit (es decir, cuando se ha recibido λ) la salida es 0. Obtenemos así el diagrama de Moore de la figura 2.6, en el que cada estado tiene una salida, y sólo una, asociada (es una máquina de Moore).

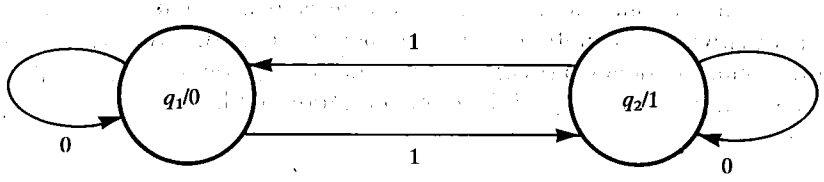


FIGURA 2.6.

2.2. Sumador binario serie

Un sumador binario es un dispositivo que suma dos números codificados en forma binaria y da el resultado también en binario. En el sumador serie los bits de los sumandos se presentan secuencialmente y por parejas, es decir, primero se presentan los dos bits de menor peso, el sumador los suma y obtiene el bit de menor peso del resultado (y toma nota del arrastre, si lo hay), luego los siguientes, etc. (figura 2.7).

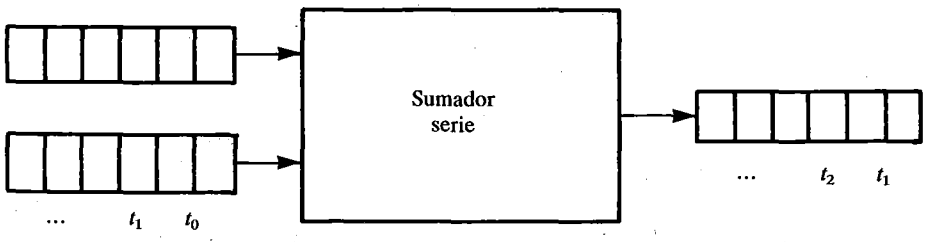


FIGURA 2.7.

Evidentemente, el sumador serie es un autómata, puesto que, en todo momento, debe recordar si ha habido arrastre de los bits sumados anteriormente, es decir, la salida no sólo depende de la entrada actual, sino también de las anteriores. También es fácil ver que sólo necesita dos estados. En efecto, en cada momento, para efectuar una suma de dos bits, sólo hay dos posibles situaciones a considerar: que no exista arrastre de los anteriores o que sí lo haya; llamemos q_1 y q_2, respectivamente, a los estados correspondientes a esas situaciones. Tenemos, por tanto:

$$E = \{00, 01, 10, 11\}$$
$$S = \{0, 1\}$$
$$Q = \{q_1, q_2\}$$

Inicialmente, el autómata estará en el estado q_1 (al recibir la primera pareja de bits no tiene que considerar ningún arrastre anterior). Si la primera pareja es 00, la salida deberá ser 0, y, como no hay arrastre, se quedará en q_1; si es 01 ó 10 deberá dar salida 1 y también quedarse en q_1, pero si recibe 11 la salida deberá ser 0, y habrá arrastre, por lo que pasará a q_2. Estando en q_2, si recibe 00, como hay arrastre de la suma anterior, deberá dar como salida 1 pero ya no habrá arrastre para la suma siguiente, por lo que pasará a q_1; sin embargo, en cualquier otro caso (01, 10, 11) se quedará en q_2, ya que sigue existiendo arrastre. Toda esta descripción se puede expresar con mayor concisión y claridad con el diagrama de Moore de la figura 2.8.

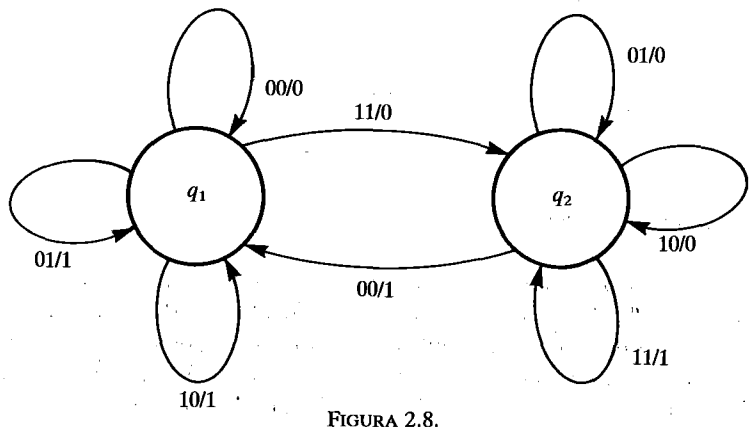


FIGURA 2.8.

Este AF es una máquina de Mealy, puesto que tanto q_1 como q_2 tienen asociadas las salidas 0 y 1. La máquina de Moore equivalente puede encontrarse siguiendo el procedimiento expuesto en el apartado 1.3, y resulta ser la descrita por el diagrama de la figura 2.9.

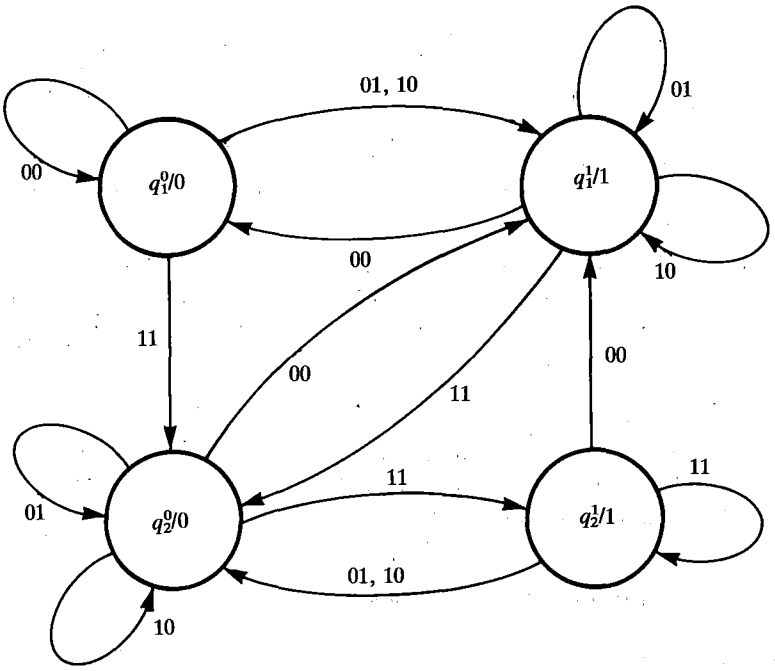


FIGURA 2.9.

2.3. El castillo encantado

El siguiente ejemplo, tomado de Ashby (1956), nos servirá para ilustrar cómo la Teoría de Autómatas tiene un campo de aplicación muy extenso: todo lo que se refiera a sistemas (en el más amplio sentido de la palabra) discretos con memoria; en este caso particular veremos cómo permite formalizar y resolver un problema de lógica en el que interviene el tiempo.

El problema es el expuesto en esta carta:

«Querido amigo: Al poco tiempo de comprar esta vieja mansión tuve la desagradable sorpresa de comprobar que está hechizada con dos sonidos de ultratumba que la hacen prácticamente inhabitable: un canto picaresco y una risa sardónica.

Aún conservo, sin embargo, cierta esperanza, pues la experiencia me ha demostrado que su comportamiento obedece a ciertas leyes, oscuras pero infalibles, y que puede modificarse tocando el órgano y quemando incienso.

En cada minuto, cada sonido está presente o ausente. Lo que cada uno de ellos

hará en el minuto siguiente depende de lo que pasa en el minuto actual, de la siguiente manera:

El canto conservará el mismo estado (presente o ausente) salvo si durante el minuto actual no se oye la risa y toco el órgano, en cuyo caso el canto toma el estado opuesto.

En cuanto a la risa, si no quemo incienso, se oirá o no según que el canto esté presente o ausente (de modo que la risa imita al canto con un minuto de retardo). Ahora bien, si quemo incienso la risa hará justamente lo contrario de lo que hacía el canto.

En el momento en que le escribo estoy oyendo a la vez la risa y el canto. Le quedaré muy agradecido si me dice qué manipulaciones de órgano e incienso debo seguir para restablecer definitivamente la calma».

La carta, especialmente en su tercer párrafo, describe un sistema lógico secuencial que puede formalizarse como un autómata finito. Hay dos variables de entrada (órgano e incienso), y como cada una de ellas tiene dos valores posibles, tendremos cuatro entradas diferentes; llamémoslas e_0, e_1, e_2, e_3.

e_0: no tocar el órgano ni quemar incienso;
e_1: no tocar el órgano pero quemar incienso;
e_2: tocar el órgano pero no quemar incienso;
e_3: tocar el órgano y quemar incienso.

También son cuatro los estados posibles:

q_0: ni risa, ni canto;
q_1: no risa, sí canto;
q_2: sí risa, no canto;
q_3: risa y canto.

En cuanto a la salida, podemos considerar dos situaciones:

1: que haya algún sonido (salida asociada a los estados q_1, q_2, q_3);
2: que no haya ningún sonido (salida asociada al estado q_0).

Con esta nomenclatura, el problema se puede expresar diciendo que nos encontramos en un estado inicial, el q_3, queremos pasar a un estado final, el q_0, y se trata de encontrar la secuencia de entrada adecuada.

Siguiendo el enunciado, podemos obtener la tabla y el diagrama de transiciones, pero ello resulta mucho más fácil si utilizamos un formalismo lógico. Designemos por I y O unas variables booleanas que representen el incienso y el órgano, respectivamente (es decir, $I = 0$ si no se quema incienso, $I = 1$ si se quema, etc.), y por R y C otras variables que representan la risa y el canto ($R = 0$ si no se oye la risa, etc.). Tenemos una correspondencia inmediata entre los valores de estas variables y los conjuntos de entradas y estados definidos más arriba:

e	$O\ I$	q	$R\ C$
e_0	0 0	q_0	0 0
e_1	0 1	q_1	0 1
e_2	1 0	q_2	1 0
e_3	1 1	q_3	1 1

Si en el minuto t los valores de estas variables son O_t, J_t, C_t, R_t, en el minuto $t + 1$ tomarán los valores dados por las siguientes expresiones lógicas, que no son más que otra forma de expresar los párrafos 4 y 5 de la carta:

$$O_t \cdot \overline{R}_t = 1 \rightarrow C_{t+1} = \overline{C}_t \tag{1}$$

$$O_t \cdot \overline{R}_t = 0 \rightarrow C_{t+1} = C_t \tag{2}$$

$$I_t = 0 \rightarrow R_{t+1} = C_t \tag{3}$$

$$I_t = 1 \rightarrow R_{t+1} = \overline{C}_t \tag{4}$$

De (1) y (2) se deduce que

$$C_{t+1} = (O_t \cdot \overline{R}_t) \oplus C_t \tag{5}$$

y de (3) y (4)

$$R_{t+1} = I_t \oplus C_t \tag{6}$$

De (5) y (6) se pueden sacar inmediatamente las tablas de verdad de C_{t+1} y R_{t+1} en función de O_t, I_t, C_t, R_t:

O_t	I_t	R_t	C_t	R_{t+1}	C_{t+1}
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	1	1	1
0	1	0	0	1	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	1
1	0	0	0	0	1
1	0	0	1	1	0
1	0	1	0	0	0
1	0	1	1	1	1
1	1	0	0	1	1
1	1	0	1	0	0
1	1	1	0	1	0
1	1	1	1	0	1

Volviendo ahora a la correspondencia establecida entre O, I y e, y entre C, R y q, la anterior tabla de verdad nos conduce a la tabla de transiciones de la figura 2.10:

e / q/s	e_0	e_1	e_2	e_3
$q_0/0$	q_0	q_2	q_1	q_3
$q_1/1$	q_3	q_1	q_2	q_0
$q_2/1$	q_0	q_2	q_0	q_2
$q_3/1$	q_3	q_1	q_3	q_1

FIGURA 2.10.

Y de aquí podemos dibujar el diagrama de Moore de la figura 2.11.

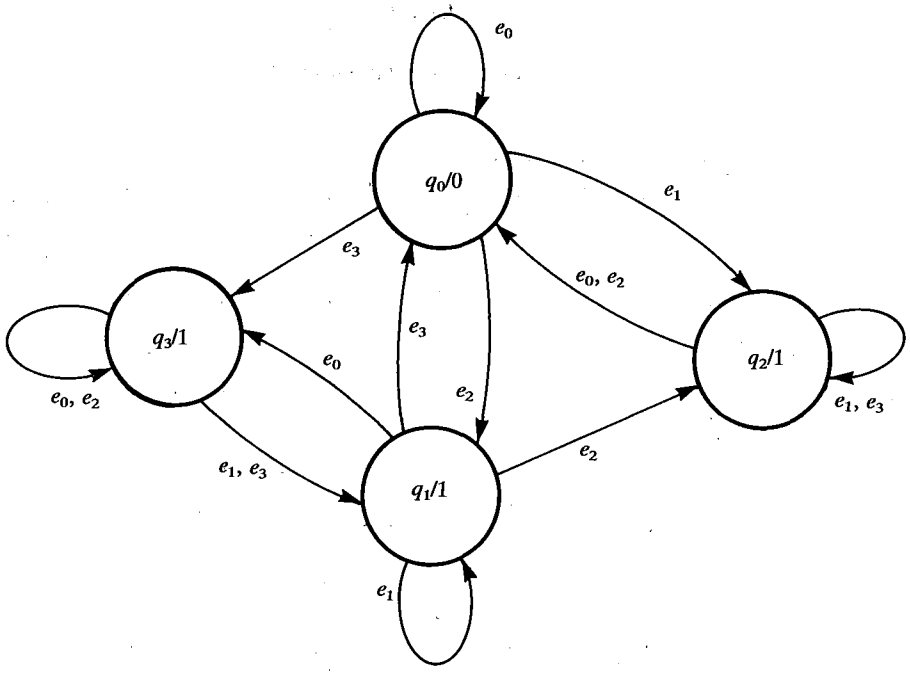


FIGURA 2.11.

A la vista de este diagrama, la solución al problema planteado en la carta aparece fácilmente: pasar primero de q_3 a q_1 mediante e_1 o e_3, y luego a q_0 mediante e_3; o bien, respondiendo en los mismos términos epistolares, «durante un minuto, queme usted incienso (tocando o no el órgano, es indiferente), y desaparecerá la risa; durante el minuto siguiente, queme incienso y toque el órgano, y al finalizar ese minuto cese toda actividad, con lo que, si no vuelve a manipular ni el órgano ni el incienso, se habrá librado para siempre de tan molestos moradores».

Hay desde luego, otras soluciones, pero todas ellas con secuencias de entrada más largas que la propuesta; por ejemplo, de q_1 puede pasarse a q_2 con e_2, y de aquí a q_0 con e_0 ó e_2. Obsérvese que, afortunadamente para el propietario de la casa, el estado q_0 es estable, en el sentido de que con la entrada e_0 (es decir, $I = 0$, $O = 0$) el siguiente estado es el mismo q_0.

3. COMPORTAMIENTO DE UN AUTÓMATA

3.1. Otra definición de autómata

En el capítulo 1 comenzamos hablando de los autómatas como dispositivos que producen cadenas de símbolos a la salida en respuesta a cadenas de símbolos presentadas a la entrada. Según esto, podríamos definir un autómata como una función:

$$F^*: E^* \rightarrow S^* \ (*) \qquad [3.1.1]$$

que hace corresponder a cada cadena de entrada, $x \in E^*$, una cadena de salida, $F^*(x) = y \in S^*$.

Ahora bien, vamos a ver que el autómata queda perfectamente definido restringiendo el rango de la función de S^* a S, es decir, podemos definir un autómata como una función

$$F: E^* \rightarrow S \qquad [3.1.2]$$

que hace corresponder a cada cadena de entrada, $x \in E^*$, el último símbolo obtenido como salida, $F(x) = s \in S$. En efecto, si $x = e_0, e_1, \ldots, e_{n-1}$, tendremos como símbolos de salida

$F(e_0)$ en el instante 1
$F(e_0, e_1)$ en el instante 2
$F(e_0, e_1, \ldots, e_{n-1}) = F(x)$ en el instante n

(*) Utilizamos la notación para lenguajes introducida en el capítulo 1, apartado 4 del tema «Sistemas lógicos».

Por consiguiente, la cadena de salida $F^*(x)$ se obtendrá concatenando todos estos símbolos:

$$F^*(x) = F^*(e_0, e_1, \ldots, e_{n-1}) = F(e_0)F(e_0, e_1) \ldots F(e_0, e_1, \ldots, e_{n-1}),$$

lo que nos demuestra que F^* queda determinada conociendo F.

Consideremos, por ejemplo, el autómata sumador binario serie estudiado en el apartado 2.2, y supongámoslo efectuando la suma 010110 + 011011 = 110001. En el instante t_0 recibirá por la entrada los bits de menor peso, es decir, $e_0 = 01$; en el instante t_1 tendremos $e_1 = 11$, etc. Así, la cadena de entrada será:

$$x = 01.11.10.01.11.00$$

(Obsérvese que, en contra de lo que es habitual, utilizamos un punto para indicar la concatenación, a fin de evitar ambigüedades en este caso, ya que los símbolos de entrada están formados por dos símbolos de nuestro alfabeto ordinario. Obsérvese también que las cadenas se escriben de izquierda a derecha en el tiempo, con lo que resulta un orden de escritura inverso al habitual en aritmética).

En el instante t_0 tendremos como salida:

$$\begin{aligned} & F(e_0) = F(01) = 1; \\ \text{en } t_1\text{: } & F(e_0, e_1) = F(01.11) = 0; \\ \text{en } t_2\text{: } & F(e_0, e_1, e_2) = F(01.11.10) = 0; \\ \text{en } t_3\text{: } & F(e_0, e_1, e_2, e_3) = F(01.11.10.01) = 0; \\ \text{en } t_4\text{: } & F(e_0, e_1, e_2, e_3, e_4) = F(01.11.10.01.11) = 1; \\ \text{y en } t_5\text{: } & F(e_0, e_1, e_2, e_3, e_4, e_5) = F(01.11.10.01.11.00) = 1. \end{aligned}$$

De modo que la cadena total de salida es:

$$F^*(x) = F(e_0)F(e_0, e_1) \ldots F(e_0, e_1, \ldots, e_6) = 100011,$$

que es el resultado de la suma escrito de izquierda a derecha según se van obteniendo los bits del resultado a partir del de menor peso.

La definición [3.1.2] considera al autómata exclusivamente desde el punto de vista de entrada-salida, es decir, como una «caja negra», a diferencia de la definición [1.1.1], en la que se contempla lo que sucede en el interior de la «caja». Algunos autores, para resaltar la diferencia entre ambos, les dan distintos nombres, y así, por ejemplo, al autómata definido por [3.1.2] le llaman «máquina», y al definido según [1.1.1], «circuito», y este mismo convenio seguiremos nosotros en adelante cuando nos interese destacar que nos referimos a una u otra definición.

En este punto, es natural que surjan dos preguntas inmediatamente: dada una máquina, ¿podemos encontrar su circuito? Y, evidentemente, la inversa. Para responder a ellas se hace precisa una consideración matemática previa sobre la definición [1.1.1]. En efecto, el dominio de las funciones f y g es $E \times Q$, lo que quiere decir que estas funciones nos permiten obtener el estado siguiente y la salida conociendo el

estado actual y el *símbolo* de entrada. Para conocer la respuesta del circuito no a un símbolo, sino a una *cadena* de entrada es necesario ampliar el dominio a $E^* \times Q$.

3.2. Ampliación del dominio de las funciones de un autómata

En el apartado 1.3 vimos que el dominio de g podía ampliarse de $E \times Q$ a $\{E \cup \{\lambda\}\} \times Q$ solamente si existe una función de salida h: $Q \to S$; en este caso decíamos que el autómata es una máquina de Moore, y teníamos:

$$f(\lambda, q) = q;\ g(\lambda, q) = h[f(\lambda, q)] = h(q)$$

Para extender ahora el dominio a $E^* \times Q$ basta con definir, para todo $x_1, x_2 \in E^*$,

$$f(x_1x_2, q) = f[x_2, f(x_1, q)]$$
$$g(x_1x_2, q) = g[x_2, f(x_1, q)] = h[f(x_1x_2, q)]$$

3.3. El comportamiento de entrada-salida, o las máquinas definidas por un circuito

Definición 3.3.1. Dado un autómata (circuito) $A = \langle E, S, Q, f, g\rangle$, definimos el comportamiento de entrada-salida de A inicializado en el estado q por la función

$$C_q\colon E^* \to S$$

que aplica a cada $x \in E^*$ un $s = g(x, q)$.

Es decir, si en el instante t_0 el autómata se encuentra en el estado q e introducimos la cadena $x = e_1, e_2, \ldots, e_n$, se obtendrán las salidas

$$C_q(e_1) \text{ en } t = t_0$$
$$C_q(e_1, e_2) \text{ en } t = t_0 + 1$$
$$C_q(e_1, e_2, \ldots, e_n) = C_q(x) \text{ en } t = t_0 + n - 1$$

Vemos así que para todo autómata (circuito) pueden definirse, en principio, tantas funciones de la forma [3.1.2] (es decir, tantas «máquinas») como estados tenga el autómata, aunque hay que advertir que algunas de estas funciones pueden ser idénticas entre sí (lo que correspondería a estados equivalentes, según la definición que daremos enseguida).

3.4. Equivalencia y accesibilidad

Damos a continuación una serie de definiciones cuyo sentido se captará mejor con ayuda de ejemplos, que desarrollaremos en el apartado 3.5.

Definición 3.4.1. Dados dos autómatas con los mismos alfabetos de entrada y salida, $A_1 = \langle E, S, Q_1, f_1, g_1\rangle$ y $A_2 = \langle E, S, Q_2, f_2, g_2\rangle$, $q_1 \in Q_1$ es *equivalente* a $q_2 \in Q_2$ si $C_{q_1} = C_{q_2}$.

La misma definición sirve para estados equivalentes dentro de un mismo autómata: basta considerar que $Q_1 = Q_2, f_1 = f_2, g_1 = g_2$.

Definición 3.4.2. Un autómata está en *forma mínima* (o es *observable*) si

$$(C_{q_1} = C_{q_2}) \rightarrow (q_1 = q_2)$$

Es decir, en un autómata en forma mínima no existen estados equivalentes.

Definición 3.4.3. Los autómatas $A_1 = \langle E, S, Q_1, f_1, g_1\rangle$ y $A_2 = \langle E, S, Q_2, f_2, g_2\rangle$ son *equivalentes* si

$$\{C_{q_1} | q_1 \in Q_1\} = \{C_{q_2} | q_2 \in Q_2\}$$

Definición 3.4.4. q_2 es *accesible* desde q_1 si existe $x \in E^*$ tal que $f(x, q_1) = q_2$.

Definición 3.4.5. El *subautómata conectado* de un autómata $A = \langle E, S, Q, f, g\rangle$ es $A^c = \langle E, S, Q^c, f^c, g^c\rangle$, con

$$Q^c = \{q_2 \in Q | (\exists x \in E^*)(\exists q_1 \in Q)(f(x, q_1) = q_2)\}$$

y f^c y g^c son las restricciones de f y g de $E^* \times Q$ a $E^* \times Q^c$.

Es decir, el subautómata conectado de un autómata está formado por todos los estados del autómata original que son accesibles desde algún otro estado.

Definición 3.4.6. Un autómata A es *fuertemente conectado* si $A = A^c$.

3.5. Ejemplos

3.5.1. *Autómata reconocedor de la cadena 010*

Considérense los AF dados por los diagramas de Moore de las figuras 2.12 y 2.13.

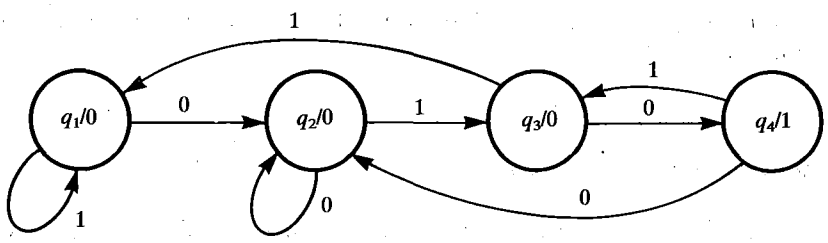


FIGURA 2.12.

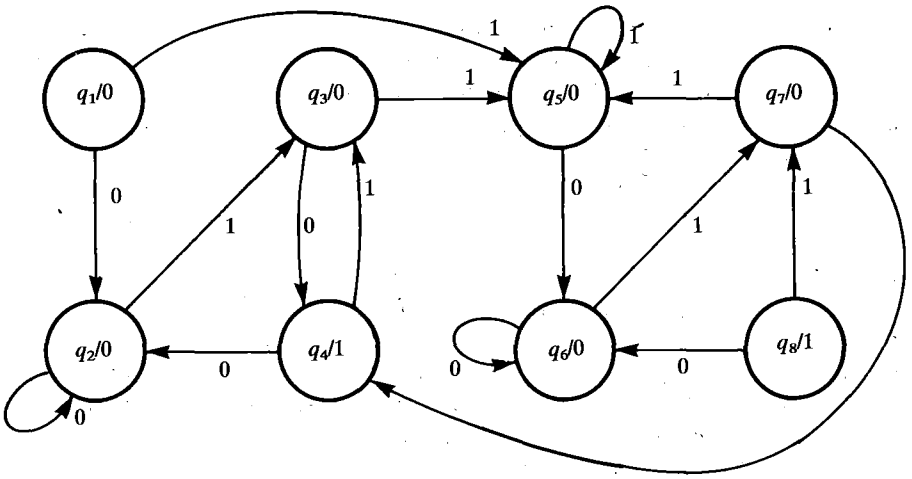


FIGURA 2.13.

a) Calcular C_{q_1}, C_{q_2}, C_{q_3} y C_{q_4} (en la figura 2.12) para cadenas de entrada de longitud igual o inferior a 3.
b) Comprobar que el AF de la figura 2.12 está en forma mínima y es fuertemente conectado. ¿Ocurre lo mismo con el de la figura 2.13?
c) Comprobar que ambos autómatas son equivalentes.

Pasemos a resolver las cuestiones planteadas.

a) Basta con seguir, para cada estado inicial, las transiciones provocadas sucesivamente por cada símbolo de la cadena (leída ésta de izquierda a derecha) y anotar la salida final. Resumimos los resultados en la tabla de la figura 2.14.

	λ	0	1	00	01	10	11	000	001	010	011	100	101	110	111
C_{q_1}	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
C_{q_2}	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0
C_{q_3}	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0
C_{q_4}	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0

FIGURA 2.14.

b) Para comprobar que un AF está en forma mínima hay que ver si el comportamiento es distinto para cada estado. Esto no es fácil con los conocimientos que tenemos hasta ahora, puesto que habría que ir ensayando con diferentes cadenas de entrada hasta que se observe una diferencia de comportamiento entre dos estados para la misma cadena. Pero como E^* es infinito, si no encontramos tal diferencia después de un número finito de cadenas no podemos garantizar que el AF esté en

forma reducida. (Empleando una terminología que se definirá con precisión en el tema siguiente, no tenemos un algoritmo). Ahora bien, en el apartado 5.3 demostraremos que basta con ensayar todas las cadenas x tales que $\lg(x) \leq n - 1$ (n = número de estados); y si $C_{q_1}(x) \neq C_{q_2}(x)$ para esas cadenas podremos asegurar que $C_{q_1}(x) \neq C_{q_2}(x)$, $\forall x \in E^*$. De este modo tendremos un algoritmo, ya que el número necesario de ensayos es finito.

En el autómata de la figura 2.12 se ve enseguida, con ayuda de la tabla de la figura 2.14, que

$$C_{q_1}(\lambda) = 0,\ C_{q_2}(\lambda) = 0,\ C_{q_3}(\lambda) = 0,\ C_{q_4}(\lambda) = 1,$$

por lo que q_4 no es equivalente a ninguno de los otros tres. Observando los comportamientos para la entrada $x = 0$ deducimos que q_3 tampoco es equivalente a ninguno, y, de igual modo, la no equivalencia de q_2 la deducimos de la entrada $x = 10$.

Por consiguiente, podemos asegurar que el AF de la figura 2.12 está en forma mínima. No ocurre lo mismo con el de la figura 2.13. En efecto, si se van ensayando cadenas diferentes de entrada se irá viendo que $C_{q_1}(x) = C_{q_5}(x)$, $C_{q_2}(x) = C_{q_6}(x)$, $C_{q_3}(x) = C_{q_7}(x)$ y $C_{q_4}(x) = C_{q_8}(x)$. Se puede comprobar que esto ocurre para todas las cadenas de longitud igual o inferior a 7 ($n = 8$), por lo que q_1 es equivalente a q_5, q_2 a q_6 y q_4 a q_8, y, por consiguiente, el AF no está en forma mínima.

El carácter de fuertemente conectado se ve por simple inspección de los diagramas. En efecto, en la figura 2.12 vemos que cualquiera de los cuatro estados tiene por lo menos un arco que entra en él, es decir, todos los estados son accesibles desde algún otro, y, por tanto, el autómata es fuertemente conectado. Sin embargo, en la figura 2.13 podemos observar que ni q_1 ni q_8 son accesibles desde ningún otro estado, por lo que el autómata no es fuertemente conectado.

c) Ensayando todas las cadenas de longitud igual o inferior a 7 deducimos que q_1 y q_4 de la figura 2.13 son equivalentes a q_1 de la figura 2.12, q_2 y q_6 a q_2, q_3 y q_7 a q_3 y q_4 y q_8 a q_4. Por tanto, ambos AF son equivalentes.

Finalmente, justifiquemos el título puesto a este ejemplo, aunque los autómatas reconocedores serán objeto de estudio en el capítulo 4. Puede comprobarse que siempre que en la cadena de entrada aparezca la sucesión 010 la salida del autómata es 1 al recibir el último símbolo de la sucesión (el segundo 0); en caso contrario la salida será 0, y no dará 1 hasta que en la cadena de entrada no vuelvan a aparecer seguidos un 0, un 1 y luego otro 0.

3.5.2. *Ejemplo 2: Equivalencia máquina de Moore-máquina de Mealy*

En el apartado 1.3 decíamos que para cualquier máquina de Mealy se puede obtener una máquina de Moore equivalente, y construíamos esta máquina escindiendo cada estado de la primitiva en tantos estados como salidas pudieran asociársele. También decíamos que en lo sucesivo nos referiríamos siempre a máquinas de Moore, en las que podemos considerar el elemento neutro λ en la entrada, y, por consiguien-

te, el monoide libre de entrada, E^*, mientras que en el caso de máquinas de Mealy tendríamos que trabajar con el semigrupo libre de entrada, E^+. Al definir la equivalencia entre autómatas (Definición 3.4.3) nos hemos basado en el comportamiento de entrada-salida (Definición 3.3.1), que se ha definido no con E^+, sino con E^*, suponiendo implícitamente que los autómatas considerados son máquinas de Moore. Lo que pretendemos en este ejemplo es ver que la máquina de Moore construida a partir de una de Mealy como se indica en el apartado 1.3 es efectivamente equivalente a ella, en el sentido de equivalencia de comportamiento. Para ello tenemos que ver que los conjuntos de los comportamientos de ambas máquinas son idénticos, con la salvedad de que no tiene sentido hablar de la entrada λ; es decir, por esta sola vez, digamos que el comportamiento para un estado q es

$$C_q\colon E^+ \to S$$

Pues bien, sobre el ejemplo de las figuras 2.2 y 2.3, y si llamamos A a la primera y $\hat{A}$ a la segunda, podemos comprobar que

$$C_{q_1}(a) = \hat{C}_{q_1}(a) = 0;\ C_{q_1}(b) = \hat{C}_{q_1}(b) = 1;\ C_{q_1}(aa) = \hat{C}_{q_1}(aa) = 0;$$
$$C_{q_1}(ab) = \hat{C}_{q_1}(ab) = 1;\ C_{q_1}(ba) = \hat{C}_{q_1}(ba) = 0;$$

etc., es decir,

$$C_{q_1} = \hat{C}_{q_1}$$

Análogamente se ve que $C_{q_2} = \hat{C}^0_{q_2} = \hat{C}^1_{q_2}$ y que $C_{q_3} = \hat{C}^0_{q_3} = \hat{C}^1_{q_3}$.

En general, lo que ocurre es que todos los estados q^s resultantes de una escisión de q tienen igual comportamiento (salvo, naturalmente, para $x = \lambda$), ya que se toma

$$\hat{g}(x, q^s) = g(x, q)$$

para todos los q^s.

4. CAPACIDAD DE RESPUESTA DE UN AUTÓMATA FINITO

4.1. Introducción

Hemos visto que un circuito con n estados puede realizar hasta n máquinas diferentes. Cada una de estas máquinas viene definida por el comportamiento de entrada-salida para el correspondiente estado.

El número de cadenas diferentes en E^* es infinito. Pero como el autómata es finito es imposible que responda de distinta manera a cada una de ellas. Es decir, debe ser posible particionar E^* en un número finito de subconjuntos tales que el autómata sea incapaz de distinguir dos cadenas pertenecientes al mismo subconjunto. Veremos que tales subconjuntos pueden considerarse como clases de equivalencia; naturalmente,

cuanto mayor sea el número de clases de equivalencia mayor será la capacidad del autómata para responder de distinta manera a cadenas diferentes.

Cuando decimos que el autómata «responde» a una cadena o «distingue» entre una u otra pensamos en el símbolo de salida asociado a cada cadena, de acuerdo con la Definición 3.1.2. Si consideramos exclusivamente máquinas de Moore, existirá una función de salida h: $Q \rightarrow S$. Vamos en este apartado a simplificar el análisis, suponiendo que h es biyectiva, es decir, que a cada estado podemos asignarle una salida diferente*. En este caso, dos cadenas pertenecerán a la misma clase de equivalencia (serán indistinguibles para todas las máquinas definidas por el circuito) si, considerando un estado inicial cualquiera, el estado final es el mismo para ambas cadenas. Nos vemos así conducidos a estudiar, para cada cadena, funciones de la forma $Q \rightarrow Q$, y, si llamamos Q^Q al conjunto de todas estas funciones, el comportamiento global del autómata vendrá determinado por una función K: $E^* \rightarrow Q^Q$, que asigna a cada cadena de entrada una función $Q \rightarrow Q$. Si el número de estados diferentes es n, habrá n^n funciones $Q \rightarrow Q$, por lo que el número máximo de clases de equivalencia en E^* será n^n.

Vamos a formalizar estas ideas, y para ello comenzaremos por recordar algunos conceptos de álgebra, pero antes debemos advertir al lector que este apartado, que es el más teórico del tema, no es imprescindible para la comprensión del resto, y, si lo desea, puede omitirlo y saltar al apartado 5 (minimización de AF).

4.2. Repaso de algunos conceptos de álgebra

4.2.1. *Monoide de transformaciones de un conjunto*

Una transformación t en un conjunto C se define como una función de C en sí mismo: t: $C \rightarrow C$.

Teorema 4.2.1.1. Sea C un conjunto cualquiera y sea $C^C = \{t: C \rightarrow C\}$ el conjunto de todas las transformaciones en C. Entonces $\langle C^C, \circ \rangle$, donde «$\circ$» representa la ley de composición de funciones t, es un monoide, llamado *monoide de transformaciones* de C.

La demostración es casi inmediata, teniendo en cuenta la evidencia de que la composición de funciones t es una operación cerrada y asociativa, es decir, si t_a, t_b, $t_c \in C^C$,

$$t_a \circ t_b \in C^C$$

* Esta simplificación no resta generalidad al análisis. En efecto, lo que haremos será estudiar la «respuesta de estados» en el sentido de que para cada entrada nos fijaremos no en la salida, sino en las transiciones que provoca entre los estados. Si h no fuera biyectiva lo único que podría ocurrir es que dos cadenas diferentes en cuanto a su «respuesta de estados» fueran indistinguibles en cuanto a la salida, pero esto es fácil de analizar conociendo la función h: $Q \rightarrow S$.

y

$$(\forall c \in C)[t_a \circ (t_b \circ t_c((c))) = (t_a \circ t_b) \circ t_c(c) = t_a(t_b(t_c(c)))]$$

Además, podemos definir un elemento neutro en C^C, t_1: $C \to C$, tal que $t_1 \circ t_i = t_i \circ t_1 = t_i$; este elemento neutro es $t_1(x) = x$. Por consiguiente, $\langle C^C, \circ \rangle$ cumple las condiciones para ser un monoide.

Ejemplo. Sea $C = \{0, 1\}$. Veamos cuál es el monoide de transformaciones de C. C^C tendrá cuatro elementos:

$$t_0\text{: } t_0(0) = 0,\ t_0(1) = 0$$
$$t_1\text{: } t_1(0) = 0,\ t_1(1) = 1$$
$$t_2\text{: } t_2(0) = 1,\ t_2(1) = 0$$
$$t_3\text{: } t_3(0) = 1,\ t_3(1) = 1$$

(Obsérvese que, acorde con la notación anterior, t_1 es el elemento neutro).

La composición de t_2 con t_3, por ejemplo, será:

$$t_2 \circ t_3(0) = t_2(t_3(0)) = t_2(1) = 0$$
$$t_2 \circ t_3(1) = t_2(t_3(1)) = t_2(1) = 0;$$

luego

$$t_2 \circ t_3 = t_0$$

Análogamente pueden hallarse las demás composiciones; el resultado puede ponerse en forma tabular (tabla del monoide):

$\circ$	t_0	t_1	t_2	t_3
t_0	t_0	t_0	t_0	t_0
t_1	t_0	t_1	t_2	t_3
t_2	t_3	t_2	t_1	t_0
t_3	t_3	t_3	t_3	t_3

Esta tabla representa la operación $\circ$ en el conjunto C^C, y por tanto, describe al monoide $\langle C^C, \circ \rangle$.

4.2.2. *Homomorfismo entre monoides*

Definición 4.2.2.1. Si $\langle S_1, * \rangle$ y $\langle S_2, \cdot \rangle$ son dos semigrupos, una función f: $S_1 \to S_2$ decimos que es un *homomorfismo* entre ambos semigrupos si preserva la ley de composición interna, es decir, si

$$(\forall a, b \in S_1)[f(a * b) = f(a) \cdot f(b)]$$

o, expresado gráficamente, si el diagrama

$$\begin{array}{ccc} S_1 \times S_1 & \xrightarrow{*} & S_1 \\ f \times f \downarrow & & \downarrow f \\ S_2 \times S_2 & \rightarrow & S_2 \end{array}$$

es conmutativo.

Definición 4.2.2.2. Un *isomorfismo* entre semigrupos es un homomorfismo en el que la función f es biyectiva.

Definición 4.2.2.3. Si S_1 es un monoide con elemento neutro e_1, S_2 es un monoide con elemento neutro e_2, f es un homomorfismo (isomorfismo) entre S_1 y S_2, y se cumple que

$$f(e_1) = e_2,$$

entonces f es un *homomorfismo (isomorfismo) monoide.*

Ejemplos

1) Si $P = \{1, 2, 3, \ldots\}$ y $N = \{0, 1, 2, 3, \ldots\}$, entonces $\langle P, \cdot\rangle$, $\langle N, +\rangle$ y $\langle N, \cdot\rangle$ son monoides, mientras que $\langle P, +\rangle$ es un semigrupo, pero no un monoide. (¿Por qué?). La multiplicación por un número natural a en $\langle N, +\rangle$, es decir, $\langle N, +\rangle \rightarrow \langle N, +\rangle$: $n \rightarrow a \cdot n$ es un homomorfismo, ya que $a \cdot (n_1 + n_2) = a \cdot n_1 + a \cdot n_2$. Sin embargo, en general, no es un homomorfismo para $\langle N, \cdot\rangle$. (¿En qué casos particulares lo es?).

2) Llamemos R al conjunto de los números reales. $\langle R, +\rangle$ y $\langle R, \cdot\rangle$ son monoides. (¿Por qué?). La función

$$\langle R, +\rangle \rightarrow \langle R, \cdot\rangle : r \rightarrow a^r \qquad (a, r \in R)$$

es un homomorfismo monoide, ya que

$$a^{r_1 + r_2} = a^{r_1} \cdot a^{r_2} \quad \text{y} \quad 0^r = 1$$

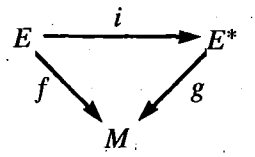
(¿es isomorfismo?).

4.2.3. *Monoide libre y homomorfismo*

Hemos definido en el apartado 4.2 del capítulo 1 del tema «Lógica» el monoide libre generado por un alfabeto E, $\langle E^*, \rangle$.

Teorema 4.2.3.1. Sea i: $E \rightarrow E^*$ la función que aplica todo elemento de E en la correspondiente cadena de longitud unidad, es decir, $(\forall a \in E)$ $(i(a) = a)$ y sea f

cualquier función de E en el conjunto de cualquier monoide $\langle M, * \rangle$. Entonces, existe un único homomorfismo monoide g: $\langle E^*, \ \rangle \rightarrow \langle M, * \rangle$ tal que $g \circ i = f$, es decir, tal que el diagrama



es conmutativo.

Demostración

Para cadenas de longitud 1 podemos definir $g(a) = f(a)$ para que se satisfaga $f(a) = g \circ i(a) = g(i(a)) = g(a)$. Si x es una cadena de longitud $l \geqslant 2$ podemos descomponerla en $x = ya_n$, donde $\lg(y) = l - 1$ y $\lg(a_n) = 1$, y tendremos:

$$g(x) = g(ya_n) = g(y) * g(a_n)$$

para que g sea un homomorfismo. Como $\lg(a_n) = 1$,

$$g(x) = g(y) * f(a_n)$$

Del mismo modo, podemos descomponer y en za_{n-1}, y así sucesivamente, con lo que, por inducción sobre la longitud de la cadena, si $x = a_1a_2 \ldots a_n$ podemos determinar $g(x)$ así:

$$g(x) = f(a_1) * f(a_2) \ldots * f(a_n)$$

Finalmente, si e es el elemento neutro de $\langle M, * \rangle$, haremos $g(\lambda) = e$ para obtener un homomorfismo monoide.

Este teorema nos permite extender el dominio de cualquier función $E \rightarrow M$ de alfabeto E en el conjunto de un monoide $\langle M, * \rangle$ a un homomorfismo monoide $\langle E^*, \ \rangle \rightarrow \langle M, * \rangle$, y nos será de utilidad más adelante.

4.2.4. *Relaciones de congruencia y monoide cociente*

Definición 4.2.4.1. Una *relación binaria* R en un conjunto C es un subconjunto de $C \times C$. Si c_1 y c_2 son elementos de C, decimos que c_1 y c_2 están relacionados por R si $(c_1, c_2) \in R$, y generalmente se escribe: $c_1 \ R \ c_2$.

Definición 4.2.4.2. R es una *relación de equivalencia* en un conjunto C si y sólo si R es una relación binaria que tiene las propiedades:

— Reflexiva : $(\forall c \in C)(c\ R\ c)$
— Simétrica : $(\forall c_1, c_2 \in C)[(c_1\ R\ c_2) \rightarrow (c_2\ R\ c_1)]$
— Transitiva: $(\forall c_1, c_2, c_3 \in C)[(c_1\ R\ c_2) \wedge (c_2\ R\ c_3) \rightarrow (c_1\ R\ c_3)]$

Definición 4.2.4.3. Dados un conjunto C y una relación de equivalencia R, la *clase de equivalencia* de $c \in C$ módulo R, es $[c]_R = \{x | x\ R\ c\}$. (En adelante, siempre que no se preste a ambigüedad, suprimimos el subíndice R). *El conjunto cociente* C/R es el conjunto de las clases de equivalencia de C módulo R. El número de elementos de C/R es el *índice* de la equivalencia.

Así, una relación de equivalencia origina una división de C en subconjuntos disjuntos, es decir, una partición.

Definición 4.2.4.4. Dado un monoide $\langle M, * \rangle$ y una relación de equivalencia, R, en M, R es una *relación de congruencia* en $\langle M, * \rangle$ si $a\ R\ b$ implica que $(a * c)\ R\ (b * c)$ y $(c * a)\ R\ (c * b)$ para todo $c \in M$.

Teorema 4.2.4.5. Si R es una relación de congruencia en el monoide $\langle M, * \rangle$, el conjunto cociente $M/R = \{[a] \mid a \in M\}$ con la operación «·» definida por

$$[a] \cdot [b] = [a * b]$$

es un monoide.

Demostración

En primer lugar hay que demostrar que la operación «·» está bien definida sobre las clases de equivalencia, es decir, que el resultado es independiente de que se tome un miembro u otro de la clase. Para ello, sean $a_1 \in [a]$, $a_2 \in [a]$, $b_1 \in [b]$, $b_2 \in [b]$. Tenemos pues que $a_1\ R\ a_2$ y $b_1\ R\ b_2$, y, como R es una relación de congruencia, podemos escribir:

$$(a_1 * b_1)\ R\ (a_1 * b_2) \quad \text{y} \quad (a_1 * b_2)\ R\ (a_2 * b_2),$$

y, por la propiedad transitiva de R,

$$(a_1 * b_1)\ R\ (a_2 * b_2),$$

es decir,

$$[a_1 * b_1] = [a_2 * b_2],$$

lo que indica que $[a] \cdot [b]$ está bien definida.

Para ver que $\langle M/R, \cdot \rangle$ es un monoide, basta con comprobar dos hechos:

1) «·» es asociativa. En efecto, como «*» es asociativa (pues $\langle M, * \rangle$ es un monoide),

$$[a] \cdot \{[b] \cdot [c]\} = [a] \cdot [b * c] = [a * (b * c)] = [(a * b) * c] = \{[a] \cdot [b]\} \cdot [c]$$

2) Existe un elemento neutro. En efecto, si el elemento neutro en $\langle M, * \rangle$ es e, tendremos que

$$[a] \cdot [e] = [a * e] = [a]$$

y

$$[e] \cdot [a] = [e * a] = [a]$$

por lo que $[e]$ es el elemento neutro para $\langle M/R, \cdot \rangle$.

Definición 4.2.4.6. El monoide $\langle M/R, \cdot \rangle$ se llama *monoide cociente* de M por R.

4.3. Comportamiento de entrada-estados

Sea un AF

$$A = \langle E, S, Q, f, g \rangle$$

Sabemos que dados un símbolo de entrada y un estado podemos obtener el estado siguiente mediante la función

$$f\colon E \times Q \to Q$$

Dado solamente un símbolo de entrada, $e \in E$, podemos definir una función que aplique a cada estado el siguiente bajo esa entrada determinada, es decir, una función de Q en Q, $k(e)\colon Q \to Q$. Existirá entonces una función

$$k\colon E \to Q^Q,$$

siendo Q^Q el conjunto de las funciones de Q en Q, que sabemos por el Teorema 4.2.1.1 que, con la composición de funciones, es un monoide. La función k se determina a partir de f del siguiente modo: para cada $e \in E$, y cada $q \in Q$, $[k(e)](q) = f(e, q)$.

Ahora bien, por el Teorema 4.2.3.1, la función k puede extenderse a un homomorfismo entre monoides:

$$K\colon \langle E^*, \rangle \to \langle Q^Q, \circ \rangle,$$

con $K(e_1, e_2, \ldots, e_n) = k(e_1) \circ k(e_2) \circ \ldots \circ k(e_n)$. Llamaremos a K *comportamiento de entrada-estados* del autómata.

Ejemplo. Consideremos el autómata detector de paridad descrito por el diagrama de la figura 2.6. La función $k\colon \{0, 1\} \to Q^Q$ se obtiene fácilmente del diagrama y se puede resumir en forma de tabla:

		Estado siguiente	
		$k(0)$	$k(1)$
Estado inicial	q_1	q_1	q_2
	q_2	q_2	q_1

Calculemos K para algunas cadenas:

$$K(0) = k(0);\ K(1) = k(1);$$
$$K(00) = k(0) \circ k(0) = k(0);$$
$$K(01) = k(0) \circ k(1) = k(1);$$
$$K(10) = k(1) \circ k(0) = k(1);$$
$$K(11) = k(1) \circ k(1) = k(0);$$

etcétera.

En general,

$K(x) = k(0)$ si se tiene un número par de unos
$K(x) = k(1)$ si se tiene un número impar de unos

Si la cadena es vacía, $K(\lambda)$ será la función identidad en Q, es decir, la función que aplica q_1 en q_1 y q_2 en q_2, que coincide con $k(0)$.

4.4. Relación equirrespuesta y monoide de un autómata

Sea un autómata A con comportamiento de entrada-estados K.

Definición 4.4.1. La *relación equirrespuesta* de A, $\cong$, es una relación binaria en E^* tal que

$$(\forall x, y \in E^*)[(x \cong y) \leftrightarrow (K(x) = K(y))]$$

$\cong$ es una relación de equivalencia, ya que se cumple:

$$(\forall x \in E^*)[K(x) = K(x)]$$
$$(\forall x, y \in E^*)[(K(x) = K(y)) \rightarrow (K(y) = K(x))]$$
$$(\forall x, y, z \in E^*)[(K(x) = K(y)) \wedge (K(y) = K(z)) \rightarrow (K(x) = K(z))]$$

Además es una relación de congruencia en el monoide $\langle E^*, \rangle$. En efecto, si $x \cong y$, $K(x) = K(y)$, y $K(xz) = K(x) \circ K(z) = K(y) \circ K(z) = K(yz)$. Análogamente, $K(zx) = K(zy)$. Por tanto, según el Teorema 4.2.4.5., $\langle E^*/\cong, \rangle$, es decir, el conjunto cociente $E^*/\cong$ con la operación de concatenación, es un monoide.

Definición 4.4.2. Dado un autómata con un comportamiento de entrada-estados K que origina una relación equirrespuesta $\cong$ en E^*, el monoide cociente $\langle E^*/\cong, \rangle$ se llama *monoide del autómata.*

Si el número de estados es n, el monoide del autómata tendrá como máximo n^n elementos. En efecto, el número de transformaciones en el conjunto Q, es decir, de funciones diferentes $Q \to Q$ es n^n. El homomorfismo K aplica entonces un conjunto infinito, E^*, en un conjunto, Q^Q, que tiene n^n elementos. Si esta aplicación es suprayectiva la relación equirrespuesta tendrá índice n^n, es decir, inducirá n^n clases de equivalencia en E^* (si la aplicación no fuera suprayectiva, es decir, si existieran una o más funciones $Q \to Q$ a las que no correspondiese ningún elemento de E^*, el número de clases de equivalencia sería menor). Por consiguiente, el número máximo de elementos del conjunto cociente $E^*/\cong$ es n^n.

El monoide de un autómata refleja la capacidad de éste para responder de distinto modo a las cadenas de entrada. En efecto, en E^* hay infinitas cadenas, mientras que en $E^*/\cong$ hay como máximo n^n elementos, que son las clases de congruencia de $\cong$. Si dos cadenas diferentes, x e y, están en la misma clase, es decir, $x \cong y$, entonces $K(x) = K(y)$, es decir, el homomorfismo K las aplica sobre el mismo elemento de Q^Q, o lo que es lo mismo, ambas producen la misma transformación $Q \to Q$, y el autómata será incapaz de distinguir una de la otra.

4.5. Ejemplos

4.5.1. *Detector de paridad*

Ya hemos estudiado el comportamiento de entrada-estados del detector de paridad, llegando a la conclusión de que sólo hay dos clases de equivalencia en E^*; en efecto, veíamos que $K(x) = k(0) = k(\lambda)$, si x tiene un número par de unos (o ninguno) y $K(x) = k(1)$, si x tiene un número impar de unos. Luego toda cadena x es equivalente a «λ» o a «1». Llamemos [λ] y [1] a estas dos clases de equivalencia, que serán los elementos del monoide del autómata. Al tener un número finito de elementos, podemos describirlo mediante una tabla que indique el elemento resultante de la concatenación de otros dos:

·	[λ]	[1]
[λ]	[λ]	[1]
[1]	[1]	[λ]

4.5.2. *Reconocedor de la cadena 010*

El autómata, representado en la figura 2.12, tiene 4 estados, por lo que el número total posible de transformaciones en Q, es decir, el número máximo de elementos del monoide del autómata es $4^4 = 256$. Veamos si se tienen todos ellos. Para ello, calculemos $K(x)$ para cadenas de longitud creciente desde $x = \lambda$. Resumimos los

resultados en la tabla de la figura 2.15. El procedimiento para construir una tabla de este tipo es el siguiente:

Tenemos tantas filas como estados tiene la máquina. Para cada cadena x ponemos en una columna los estados resultantes de la función $K(x)$: $Q \to Q$. Comenzamos por λ, para la que la función es la unidad (es decir, q_1 en q_1, q_2 en q_2, etc.). Para las cadenas de longitud 1 (en este caso 0 y 1) miramos en el diagrama de Moore; en este ejemplo, cuando se aplica $x = 0$ la imagen de q_1 es q_2, la de q_2 es q_2, etc. Teniendo ya $K(x)$ para cadenas de longitud 1 no hace falta consultar más el diagrama; por ejemplo, $K(01) = K(0) \circ K(1)$, y así, con 0, q_1 se aplica en q_2, y luego con 1, q_2 se aplica en q_3, por lo que la imagen de q_1 será q_3. Para cadenas de longitud 3 nos basamos en los resultados anteriores; por ejemplo, $K(010) = K(01) \circ K(0) = K(0) \circ K(10)$ (las dos formas de hacerlo son válidas). La tabla se va así rellenando por columnas, aumentando la longitud de las cadenas por postconcatenación de los símbolos del alfabeto a las cadenas ya tratadas, hasta que encontramos cadenas x tales que $K(x) = K(y)$, donde $\lg(y) \leq \lg(x)$; entonces podemos asegurar que $x \cong y$, y no es preciso que sigamos aumentando x, ya que, si la concatenamos con un símbolo cualquiera, e, tendremos $K(xe) = K(ye)$, es decir, $xe \cong ye$, con $\lg(ye) \leq \lg(x)$, por lo que la clase de ye o bien ya ha aparecido (si $\lg(y) < \lg(x)$), o bien va a aparecer (si $\lg(y) = \lg(x)$).

Estado final

Estado inicial x	λ	0	1	00	01	10	11	000	001	010
q_1	q_1	q_2	q_1	q_2	q_3	q_2	q_1	q_2	q_3	q_4
q_2	q_2	q_2	q_3	q_2	q_3	q_4	q_1	q_2	q_3	q_4
q_3	q_3	q_4	q_1	q_2	q_3	q_2	q_1	q_2	q_3	q_4
q_4	q_4	q_2	q_3	q_2	q_3	q_4	q_1	q_2	q_3	q_4

x	011	100	101	110	111	0100	0101
q_1	q_1	q_2	q_3	q_2	q_1	q_2	q_3
q_2	q_1	q_2	q_3	q_2	q_1	q_2	q_3
q_3	q_1	q_2	q_3	q_2	q_1	q_2	q_3
q_4	q_1	q_2	q_3	q_2	q_1	q_2	q_3

FIGURA 2.15.

Concretemos con nuestro ejemplo. Siete de las ocho cadenas de longitud 3 tienen el mismo comportamiento que cadenas de longitud 2: $K(000) = K(00)$; $K(001) = K(01)$, etc., por lo que [000] = [00]; [001] = [01], etc. La única cadena cuyo comportamiento difiere de los anteriores es 010. Por tanto, estudiamos 0100 y 0101, y vemos que $K(0100) = K(00)$ y $K(0101) = K(001)$. Nos quedan así solamente 8 clases de equivalencia: [λ], [0], [1], [00], [01], [10], [11] y [010]. El monoide del autómata tendrá pues 8 elementos, no 256: K dista mucho de ser suprayectiva. La tabla del monoide se obtiene sin dificultad de la figura 2.15, y es la representada en la

figura 2.16; por ejemplo, para saber el resultado de [11] [10], con ayuda de la figura 2.15 vemos que $K(1110) = K(11) \circ K(10) = K(00)$, por lo que [11] [10] = [00].

	[λ]	[0]	[1]	[00]	[01]	[10]	[11]	[010]
[λ]	[λ]	[0]	[1]	[00]	[01]	[10]	[11]	[010]
[0]	[0]	[00]	[01]	[00]	[01]	[010]	[11]	[010]
[1]	[1]	[10]	[11]	[00]	[01]	[00]	[11]	[010]
[00]	[00]	[00]	[01]	[00]	[01]	[10]	[11]	[010]
[01]	[01]	[010]	[11]	[00]	[01]	[00]	[11]	[010]
[10]	[10]	[00]	[01]	[00]	[01]	[010]	[11]	[010]
[11]	[11]	[00]	[11]	[00]	[01]	[00]	[11]	[010]
[010]	[010]	[00]	[01]	[00]	[01]	[010]	[11]	[010]

FIGURA 2.16.

Con la tabla del monoide se puede hallar rápidamente el comportamiento para cualquier cadena. Así, por ejemplo, si queremos saber C_q (010000110), tendremos:

$$[010000110] = [010]\,[00]\,[01]\,[10] = [00]\,[00] = [00],$$

y como [00] transforma cualquier estado inicial en q_2 (figura 2.15), y q_2 da salida 0, tendremos que $C_{q_i}(0100000110) = 0$, $i = 1, 2, 3, 4$.

4.5.3. *Contador en código binario natural módulo 10*

Un contador es un autómata que proporciona en cada momento información sobre el número de entradas de un determinado tipo recibidas hasta ese momento. Supondremos que $E = \{0, 1\}$, y que el contador debe dar como salida el código en BCD natural correspondiente al número total de «1» recibidos. Si el autómata es finito sólo podrá contar hasta un determinado número y a partir de él volver al principio; un contador módulo p contará de 0 a $p - 1$.

Consideremos $p = 10$, es decir, el contador contará de 0 a 9 y al recibir el décimo «1» volverá a contar desde 0. Cada vez que recibe un «1» deberá cambiar de estado (y, naturalmente, de salida). Así es fácil ver que el diagrama de Moore es el de la figura 2.17, en donde se supone que el estado inicial es q_0, y como símbolos de salida se han tomado $s_0, \ldots, s_9$ para abreviar la notación (en realidad serían 0000, 0001, ..., 1001).

Como hay 10 estados, el monoide del autómata podría tener hasta 10^{10} elementos. Ahora bien, si construimos la tabla de $K(x)$ (figura 2.18) vemos que sólo hay 10 clases de equivalencia.

En efecto, vemos que $0 \cong \lambda$, por lo que sólo hay que considerar cadenas compuestas por «1». Calculamos $K(1^2)$, $K(1^3)$, etc. (el exponente indica concatenación: $1^3 = 111$, etc.), y vemos que $1^{10} \cong \lambda$. Por consiguiente, las clases de equivalencia son:

$$[\lambda], [1], [1^2], [1^3], \ldots, [1^9]$$

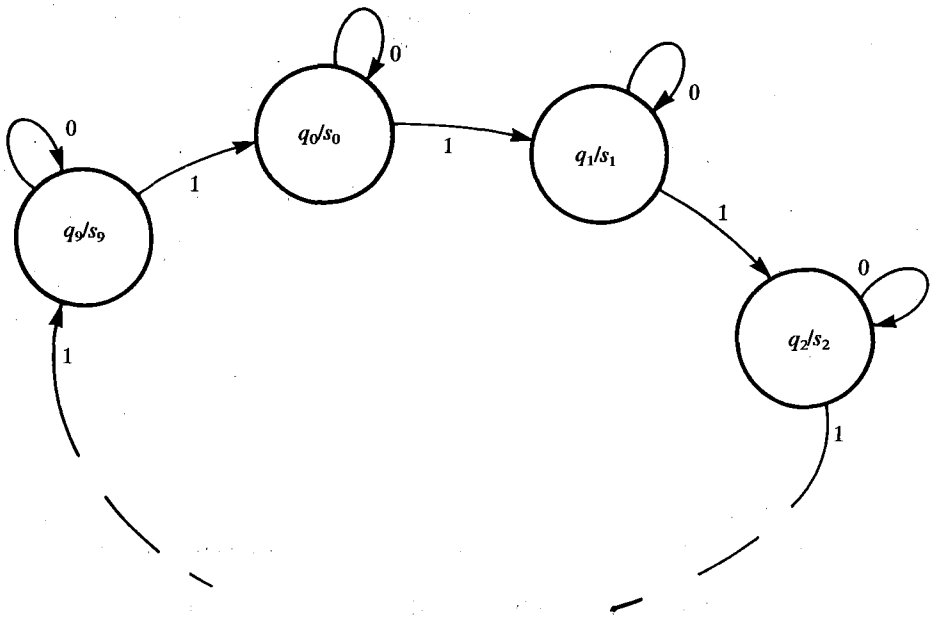


FIGURA 2.17.

Dejamos como ejercicio al lector la obtención de la tabla del monoide del autómata. Con ella podrá fácilmente comprobar que cualquier cadena que contenga n_0 «ceros» y n_1 «unos» en cualquier orden pertenece a la clase de equivalencia $[1^m]$, donde $m = n_1$ (mód. 10).

	λ	0	1	1^2	...	1^9	1^{10}
q_0	q_0	q_0	q_1	q_2		q_9	q_0
q_1	q_1	q_1	q_2	q_3		q_0	q_1
⋮	⋮	⋮	⋮	⋮		⋮	⋮
⋮	⋮	⋮	⋮	⋮		⋮	⋮
q_8	q_8	q_8	q_9	q_0		q_7	q_8
q_9	q_9	q_9	q_0	q_1		q_8	q_9

FIGURA 2.18.

4.6. Relación entre el comportamiento de entrada-salida y el comportamiento de entrada-estados

El comportamiento de entrada-salida lo definíamos para un estado inicial q:

$$C_q\colon E^* \to S$$

Así, para cada entrada, x, $C_q(x)$ es la función que aplica x en $g(x, q)$:

$$C_q(x)\colon x \to g(x, q)$$

El comportamiento de entrada-estados, sin embargo, se define para todo el conjunto de estados:

$$K\colon E^* \to Q^Q$$

De modo que, para cada entrada, x, $K(x)$ aplicará x en una función de transformación entre estados:

$$K(x)\colon x \to l\colon\ Q \to Q,$$

donde la función l está determinada por la función f restringida a la entrada x:

$$f\colon E^* \times Q \to Q,$$
$$f(x)\colon Q \to Q$$

El sentido físico es el siguiente: *C_q determina, dado un estado inicial, el símbolo final de salida para cada cadena de entrada; K determina, para cada cadena de entrada, la correspondiente transformación entre estados*, es decir, nos dice que de q_1 se pasará a q_i, de q_2 a q_j, etc. Por tanto, K nos proporciona más información acerca del comportamiento interno del autómata. Por otra parte, la relación de equivalencia en E^* inducida por K y el monoide del autómata resultante nos permiten deducir fácilmente C_q para cualquier cadena de entrada. Así, en el ejemplo del reconocedor de la cadena 010 veíamos que [010000100] = [00], y de aquí que $C_{q_i} = 0$; para llegar a la misma conclusión con el procedimiento que seguíamos en el apartado 3.5 hubiera sido preciso reconocer sobre el diagrama (figura 2.12) todas las transiciones producidas por la cadena.

Por otra parte, podría pensarse en estudiar el comportamiento de entrada-salida para todo el conjunto de estados. Para ello podemos definir un comportamiento global de entrada-salida, $K_S(x)$ como una aplicación de E^* en el conjunto de funciones $l\colon Q_0 \to S_T$, donde Q_0 son los estados iniciales y S_T las salidas finales. K_S no será ahora un homomorfismo entre monoides, como lo era K, ya que el conjunto de funciones l no es un monoide; por tanto, K_S no nos permite definir un monoide de la máquina, y por ello hemos utilizado K en el análisis anterior. En cualquier caso, K_S también nos define una relación de equivalencia en E^*, que podría llamarse «relación equisalida»,

$\cong_S$, tal que $x \cong_S y$ si y sólo si $K_S(x) = K_S(y)$. $\cong_S$ particiona E^* en clases de equivalencia. Ahora bien, suponiendo máquinas de Moore, tenemos una función h: $Q_T \to S_T$ que nos aplica el estado final en la salida final correspondiente a ese estado, y por tanto, si llamamos $[x]_i$ a la clase de equivalencia de $E^*/\cong$ que conduce a una determinada transformación k_i: $Q_0 \to Q_T$ y $[y]_j$ a la clase de equivalencia de $E^*/\cong_S$ que corresponde a una determinada l_i: $Q_0 \to S_T$, tendremos:

$$[y]_j = \cup\, [x]_i \text{ para todo } i \text{ tal que } k_i \circ h = l_j$$

Esto demuestra que las clases de equivalencia de $\cong_S$ están perfectamente determinadas por $\cong$ y h, y justifica lo que decíamos en nota a pie de página en el apartado 4.1: que al considerar sólo la respuesta de los estados el estudio no pierde generalidad.

5. MINIMIZACIÓN DE UN AUTÓMATA FINITO

5.1. Planteamiento del problema

En el ejemplo 3.5.1 veíamos dos AF equivalentes, uno de los cuales estaba en forma mínima. Si estos AF describen un sistema secuencial que debe realizarse físicamente escogeremos el que está en forma mínima, ya que el coste de la realización, como se verá en el capítulo 3, crece con el número de estados. Es, pues, importante poder saber si un AF está en forma mínima, y, si no lo está, hallar un AF en forma mínima equivalente a él. Comenzaremos por ver que éste existe siempre; a continuación veremos que para detectar equivalencia entre estados no es preciso realizar infinitos ensayos con cadenas de entrada, y, finalmente, veremos un algoritmo para minimizar un AF, es decir, para hallar otro AF en forma mínima equivalente.

Ante todo, debemos señalar que la equivalencia entre estados de un AF, definida por

$$(\forall x \in E^*)[(q_1 = q_2) \leftrightarrow (C_{q_1}(x) = C_{q_2}(x))]$$

es una relación de equivalencia en el conjunto Q, pues, como es inmediato comprobar, es reflexiva, simétrica y transitiva. Por consiguiente, puede definirse un conjunto cociente, $Q/\equiv$, cuyos elementos serán las clases de equivalencia: $[q]$ será la clase que contiene a q.

5.2. Autómata en forma mínima de un autómata dado

Vamos a demostrar que, dado $A = \langle E, S, Q, f, h\rangle$, el AF definido por $A_M = \langle E, S, Q_M, f_M, h_M\rangle$, donde

$$Q_M = Q/\equiv$$
$$f_M(x, [q]) = [f(x, q)]$$
$$h_M([q]) = h(q),$$

es equivalente a A y está en forma mínima.

En primer lugar, habrá que demostrar que f_M y h_M están bien definidas, es decir, que son independientes del elemento q que se tome dentro de la clase $[q]$, o, lo que es lo mismo, que si $q_1 \equiv q_2$, entonces $(\forall x)$ $([f(x, q_1)] = [f(x, q_2)])$, y $h(q_1) = h(q_2)$. En efecto, por la definición de la equivalencia, $(\forall x \in E^*)$ $[(q_1 = q_2) \rightarrow (g(x, q_1) = g(x, q_2))]$. Si tomamos $x = \lambda$ resulta $h(q_1) = h(q_2)$. Y si tomamos $x = x_1x_2$,

$$g(x_1x_2, q_1) = g(x_2, f(x_1, q_1))$$
$$g(x_1x_2, q_2) = g(x_2, f(x_1, q_2)),$$

por lo que $(\forall x_1)$ $([f(x_1, q_1) = f(x_1, q_2)])$, es decir, $(\forall x \in E^*)$ $([f(x, q_1)] = [f(x, q_2)])$.

Por otra parte, es evidente que A y A_M son equivalentes, puesto que todo estado q_i de A será equivalente al estado $[q_i]$ de A_M.

Finalmente, A_M está en forma mínima, ya que si los estados $[q_1]$ y $[q_2]$ de A_M fueran equivalentes, q_1 y q_2 de A deberían ser también equivalentes, por lo que $[q_1] = [q_2]$.

5.3. Comprobación de la equivalencia entre estados de un autómata

5.3.1. *Teorema de las particiones sucesivas*

Teorema 5.3.1.1. Si $P_0, P_1, P_2, \ldots$ es una secuencia infinita de particiones en un conjunto *finito* Q tal que, para todo k, se cumple:

a) P_{k+1} es un *refinamiento* de P_k, es decir, todo bloque de P_{k+1}, B^i_{k+1}, está contenido en un bloque, B^j_k, de P_k: $B^i_{k+1} \subset B^j_k$.
b) $(P_{k+1} = P_k) \rightarrow (P_{k+2} = P_{k+1})$,

entonces existe un número entero $k_0 < \text{card}(Q)$ tal que $(\forall k \geq k_0)$ $(P_k = P_{k_0})$.

Para demostrarlo, supongamos que $\text{card}(P_0) = 0$, $\text{card}(P_1) = 1$, $\text{card}(P_2) = 2$, ... Pero como, para todo k, $\text{card}(P_k) \leq \text{card}(Q) = n$ (finito), deberá existir un entero $k_0 < n$ tal que $\text{card}(P_{k_0+1}) = \text{card}(P_{k_0})$, es decir, $P_{k_0+1} = P_{k_0}$, y, por la condición b), $(\forall k \geq k_0)$ $(P_k = P_{k_0})$.

5.3.2. *Equivalencia de orden k entre estados*

Definición 5.3.2.1. Dos estados de un AF son *equivalentes de orden k* si y sólo si conducen a la misma salida para cadenas de entrada de longitud igual o inferior a k:

$$(q_1 \equiv_k q_2) \leftrightarrow [(\lg(x) \leq k) \rightarrow C_{q_1}(x) = C_{q_2}(x)]$$

Obsérvese que para comprobar la equivalencia de orden k ya no hay que hacer un número infinito de comprobaciones o «experimentos».

Teorema 5.3.2.2. Dado un AF y $q_1, q_2 \in Q$, existe un $k_0 < \text{card}(Q)$ tal que q_1 y q_2 son equivalentes $(q_1 \equiv q_2)$ si y sólo si q_1 y q_2 son equivalentes de orden k_0 $(q_1 \equiv_{k_0} q_2)$.

Para demostrar este teorema nos apoyaremos en el 5.3.1.1. Veamos pues que las particiones inducidas en Q por las sucesivas equivalencias de orden 0, 1, ..., k, $k + 1$... cumplen las condiciones a) y b) de aquel teorema.

a) Si $q_1 \equiv_{k+1} q_2$, entonces $(\forall x, \lg(x) \leq k + 1)\,[C_{q_1}(x) = C_{q_2}(x)]$ y, evidentemente, $\forall x$, $\lg(x) \leq k$, por lo que $q_1 \equiv_k q_2$. Por tanto, todo bloque de la partición P_{k+1} inducida por $\equiv_{k+1}$ está contenido en un bloque de P_k, es decir, P_{k+1} es un refinamiento de P_k.

b) Supongamos que $P_{k+1} = P_k$, es decir $(q_1 \equiv_k q_2) \rightarrow (q_1 \equiv_{k+1} q_2)$. Si q_1 y q_2 son equivalentes de orden $k + 1$, los estados $f(e, q_1)$ y $f(e, q_2)$ serán, sea cual sea e, equivalentes de orden k, pero como $P_{k+1} = P_k$ también serán equivalentes de orden $k + 1$, y por ello q_1 y q_2 son equivalentes de orden $k + 2$, de donde $P_{k+2} = P_{k+1}$. (El razonamiento es una sucesión de condicionales. Definiendo, para abreviar la notación, las variables proposicionales

$$\begin{aligned} a &: P_{k+1} = P_k \\ b &: P_{k+2} = P_{k+1} \\ c &: q_1 \equiv_k q_2 \\ c' &: q_1 \equiv_{k+1} q_2 \\ c'' &: q_1 \equiv_{k+2} q_2 \\ d &: (\forall e \in E)(f(e, q_1) \equiv_k f(e, q_2)) \\ d' &: (\forall e \in E)(f(e, q_1) \equiv_{k+1} f(e, q_2)) \end{aligned}$$

podemos expresarlo formalmente así:

$$[(a \rightarrow (c \rightarrow c')) \wedge (c' \rightarrow d) \wedge (a \rightarrow (d \rightarrow d') \wedge (c' \rightarrow c'')) \rightarrow \\ \rightarrow ((c' \rightarrow c'') \rightarrow b)] \rightarrow (a \rightarrow b),$$

que puede comprobarse que es una tautología).

En definitiva, se cumplen las condiciones a) y b) del Teorema 5.3.1.1, y, por tanto, existe $k_0 < \text{card}(Q) = n$ tal que $P_k = P_{k_0}$, $k \geq k_0$, con lo que $(q_1 \equiv_{k_0} q_2) \rightarrow (q_1 \equiv q_2)$.

La conclusión importante es que, sin necesidad de conocer k_0, como $k_0 < n$ (número de estados del AF), para comprobar si dos estados son equivalentes bastará con comprobar si son equivalentes de orden $n - 1$.

5.4. Algoritmo para la minimización de un autómata finito

El algoritmo para hallar el AF en forma mínima de un AF dado se basa en los resultados anteriores:

1. $k = 0$. Formar P_0, poniendo en el mismo bloque los estados que tengan asociada la misma salida ($q_i \equiv_0 q_j$ si y sólo si $h(q_i) = h(q_j)$).
2. Formar P_{k+1}, teniendo en cuenta que dos estados estarán en el mismo bloque

de P_{k+1} si y sólo si para cada entrada los estados siguientes están en el mismo bloque de P_k ($q_i \equiv_{k+1} q_j$ si y sólo si $f(e, q_i) \equiv_k f(e, q_j)$, $\forall e \in E \cup \{\lambda\}$).

3. Si $P_{k+1} \neq P_k$, incrementar k en una unidad y volver al paso 2.
4. $k_0 = k$. Proceso terminado. El AF en forma mínima tiene como estados $Q_M = Q/\equiv_{k_0}$.

5.5. Ejemplos

Ejemplo 5.5.1

Consideremos el autómata reconocedor de 010 descrito en 3.5.1, y partamos del diagrama no mínimo (figura 2.13). A efectos de aplicación del algoritmo es más cómodo representar el AF por la tabla de transiciones:

	0	1
$q_1/0$	q_2	q_5
$q_2/0$	q_2	q_3
$q_3/0$	q_4	q_5
$q_4/1$	q_2	q_3
$q_5/0$	q_6	q_5
$q_6/0$	q_6	q_7
$q_7/0$	q_4	q_5
$q_8/1$	q_6	q_7

1. Observando las salidas asociadas a cada estado podemos poner:

$$P_0 = \{q_1, q_2, q_3, q_5, q_6, q_7\}, \{q_4, q_8\}$$

2. Con entrada 0, de q_1, q_2, q_5 y q_6 se pasa a estados del primer bloque de P_0, y con 1 también. De q_3 y q_7, con 0 se pasa a estados del segundo bloque de P_0, y con 1 a otros del primer bloque. Finalmente, de q_4 y q_8, con 0 se pasa al primer bloque de P_0 y con 1 también. Luego

$$P_1 = \{q_1, q_2, q_5, q_6\}, \{q_3, q_7\}, \{q_4, q_8\}$$

2'. En lugar de con palabras, pongamos simbólicamente los resultados:

$$\left.\begin{array}{l} f(0, q_1) = q_2 \\ f(0, q_2) = q_2 \\ f(0, q_5) = q_6 \\ f(0, q_6) = q_6 \end{array}\right\} \text{(1.}^{\text{er}}\text{ bloque)} \qquad \begin{array}{l} f(1, q_1) = q_5 \text{ (1.}^{\text{er}}\text{ bloque)} \\ f(1, q_2) = q_3 \text{ (2.}^{\text{o}}\text{ bloque)} \\ f(1, q_5) = q_6 \text{ (1.}^{\text{er}}\text{ bloque)} \\ f(1, q_6) = q_7 \text{ (2.}^{\text{o}}\text{ bloque)} \end{array}$$

$$\left.\begin{array}{l} f(0, q_3) = q_4 \\ f(0, q_7) = q_4 \end{array}\right\} \text{(3.}^{\text{er}}\text{ bloque)} \qquad \left.\begin{array}{l} f(1, q_3) = q_5 \\ f(1, q_7) = q_5 \end{array}\right\} \text{(1.}^{\text{er}}\text{ bloque)}$$

$f(0, q_4) = q_2$, $f(0, q_8) = q_6$ } (1.er bloque) $\quad$ $f(1, q_4) = q_3$, $f(1, q_8) = q_7$ } (2.º bloque)

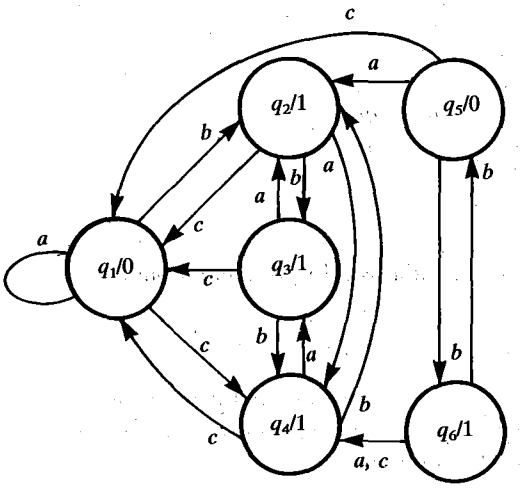
Luego,

$$P_2 = \{q_1, q_5\}, \{q_2, q_6\}, \{q_3, q_7\}, \{q_4, q_8\}$$

2″. Si repetimos la operación, viendo a qué bloques de P_2 se pasa con cada entrada veremos que $P_3 = P_2$. Luego $k_0 = 2$, y el AF minimizado tiene como estados los bloques definidos en P_2; dando los nuevos nombres $\{q_1, q_5\} = q_1$, etc., se obtiene el diagrama de la figura 2.12.

Ejemplo 5.5.2

Sea el AF dado por el diagrama y la tabla de la figura 2.19.



	a	b	c
$q_1/0$	q_1	q_2	q_4
$q_2/1$	q_4	q_3	q_1
$q_3/1$	q_2	q_4	q_1
$q_4/1$	q_3	q_2	q_1
$q_5/0$	q_2	q_6	q_1
$q_6/1$	q_4	q_5	q_4

FIGURA 2.19.

Siguiendo el algoritmo, encontramos sucesivamente:

$$P_0 = \{q_1, q_5\}, \{q_2, q_3, q_4, q_6\}$$
$$P_1 = \{q_1\}, \{q_5\}, \{q_2, q_3, q_4\}, \{q_6\}$$
$$P_2 = P_1$$

Llamando $q_1 = \{q_1\}$; $q_2 = \{q_2, q_3, q_4\}$; $q_3 = \{q_5\}$ y $q_4 = \{q_6\}$ tenemos el diagrama de la figura 2.20, que representa el AF en forma mínima equivalente al dado.

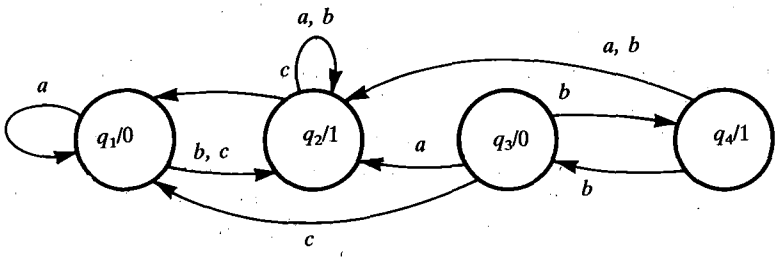


FIGURA 2.20.

6. RESUMEN

Hemos expuesto la teoría básica de los autómatas finitos procurando buscar la mayor generalidad, generalidad que se conserva aunque el estudio se restrinja a máquinas de Moore.

El comportamiento de entrada-salida es un concepto importante en relación con los lenguajes, que utilizaremos en el capítulo sobre autómatas reconocedores. Otros conceptos importantes son los de equivalencia entre estados y entre autómatas, pero el interés de éstos radica más bien en la realización tecnológica.

El estudio del comportamiento global del autómata, es decir, inicializado en cualquier estado, nos ha llevado a establecer el concepto de monoide del autómata, estructura algebraica que refleja su capacidad para responder de distinto modo a cadenas diferentes de entrada.

Finalmente, hemos visto cómo se puede comprobar la equivalencia entre estados con un número finito de experimentos, lo que nos ha permitido establecer un algoritmo para obtener el AF en forma mínima equivalente a un AF dado.

7. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

Quizás el primero en expresar el concepto moderno de autómata haya sido Leonardo Torres Quevedo, que, en 1915, respondía así en una entrevista de una revista americana: «Los antiguos autómatas... imitaban al aspecto y los movimientos de los seres vivos, pero esto no tiene mucho interés en la práctica, y lo que buscamos es un tipo de aparato que deje los meros gestos visibles del hombre e intente conseguir los resultados que obtiene una persona, para, de este modo, reemplazar a un hombre por una máquina» («Torres and his remarkable automatic devices», *Scientific American*, n.° 113, 6 nov. 1915, p. 296).

El primer estudio riguroso sobre autómatas fue el publicado por Moore (1956). Con anterioridad, debido al desarrollo de los primeros ordenadores, se habían estudiado diversos métodos para la síntesis de circuitos secuenciales (Huffman, 1954; Mealy, 1955). A finales de los años 50 se comenzó a ver la utilidad de los autómatas en relación con los lenguajes, y la mayor parte de los trabajos sobre teoría de autómatas

finitos se realizó durante los años 60. Por esta razón, las referencias más importantes que pueden darse sobre este campo son libros publicados entre 1965 y 1970. El de Harrison (1965) cubre tanto la parte combinacional como los circuitos secuenciales, con un tratamiento muy riguroso y fácil de seguir, aunque se limita a estudiar autómatas reconocedores. Otra obra recomendable es la de Booth (1967), que, además de autómatas finitos, trata también las máquinas de Turing, lenguajes artificiales y autómatas estocásticos. Muchas de las definiciones y terminología de este capítulo las hemos tomado de Arbib (1969), que cubre muy ampliamente los diversos estudios desarrollados hasta aquella fecha. La parte que trata del monoide del autómata está basada en el capítulo sobre monoides de Gilbert (1976), libro muy interesante sobre aplicaciones del álgebra moderna.

Para no alargar excesivamente el tema hemos presentado el algoritmo de minimización reducido al caso de máquinas de Moore (la máquina de Mealy mínima tendrá, normalmente, menos estados). El caso general viene expuesto en cualquiera de los libros citados en éste o en el siguiente capítulo.

8. EJERCICIOS

8.1. Obtener la tabla de transiciones y el diagrama de Moore de la máquina de Moore equivalente a la de Mealy, descrita por la siguiente tabla:

q \ e	e_1	e_2
q_1	q_3/s_1	q_2/s_2
q_2	q_4/s_2	q_3/s_2
q_3	q_4/s_1	q_2/s_2
q_4	q_2/s_2	q_4/s_1

8.2. Definir las máquinas de Mealy y de Moore de un restador binario.

8.3. Considérese un sistema eléctrico cuyas entradas son dos pulsadores, a y c (apertura y cierre), que pueden estar en reposo (0) o pulsados (1) (se supone que no puede ser $a = c = 1$), y cuyas salidas son tres lámparas: verde, ámbar y roja (V, A, R), de las que, en cada momento, una y sólo una está encendida. Inicialmente, está encendida la verde; al pulsar c, ha de pasar de verde a ámbar, y, si se pulsa otra vez (o más veces), a rojo. El pulsador a hace pasar siempre a verde.

a) Dibujar el diagrama de Moore, con $Q = \{V, A, R\}$ y $E = \{a, c\}$.
b) El modelo anterior tiene un inconveniente: si la transición de un estado a otro es instantánea (o casi), entonces el hecho de pulsar C estando en verde va a provocar el paso a rojo sin detenerse (casi) en ámbar. Las especificaciones dadas se pueden completar diciendo que hasta que no «se suelte» el pulsador no puede hacerse otra transición. Ahora va a ser necesario un estado más: «ámbar con $c \leftarrow 1$» y «ámbar con $c = 0$». Dibujar el nuevo diagrama de Moore.

8.4. Obtener los monoides del sumador y del restador binario y compararlos.

8.5. Un autómata retardador es aquel en que

$$s(t) = e(t - n)$$

Suponiendo $E = S = \{0, 1\}$ y $n = 2$, obtener el diagrama de Moore y el monoide del autómata.

8.6. Obtener el monoide del autómata cuya tabla de transición es:

q/s \ e	a	b	c
$q_1/1$	q_2	q_2	q_3
$q_2/0$	q_1	q_2	q_3
$q_3/1$	q_3	q_3	q_2

8.7. (Gilbert, 1976). En primavera, un brote de planta requiere unas condiciones adecuadas para su desarrollo. En una determinada especie, el brote necesita que se presente un día de lluvia seguido de dos días calurosos sin ser interrumpido por ningún día frío o de helada. Además, si hay un día de helada después de que se haya desarrollado el brote, éste muere. Dibujar el diagrama de Moore de este proceso.

Puede utilizarse $E = \{L, C, F, H\}$, donde L quiere decir «día lluvioso», C, «caluroso», etc., y $S = \{T, B, M\}$, donde T = «latente», B = «brote», M = «muerto». ¿Cuál es el número de elementos en el monoide de este autómata?

8.8. (Gilbert, 1976). Un perro puede estar tranquilo, irritado, asustado, o irritado y asustado, en cuyo caso muerde. Si le damos un hueso se queda tranquilo. Si le quitamos uno de sus huesos se pone irritado, y, si ya estaba asustado, nos muerde. Si le amenazamos se asusta y, si ya estaba irritado nos muerde. Obtener el diagrama de Moore y el monoide del perro.

8.9. Dibujar un diagrama de Moore para un autómata reconocedor de la cadena «321», suponiendo que $E = \{1, 2, 3\}$. Obtener el monoide del autómata. Comprobar si está en forma mínima.

8.10. Considérese un autómata finito definido por

$$E = \{a, b\};\ S = \{0, 1\};\ Q = \{q_1, q_2, q_3, q_4\}$$

y la tabla de transiciones:

q \ e	a	b
q_1	$q_2/1$	$q_1/0$
q_2	$q_3/0$	$q_4/0$
q_3	$q_3/0$	$q_1/1$
q_4	$q_3/0$	$q_1/1$

1.° Este autómata, ¿es una máquina de Moore o de Mealy? ¿Por qué?
2.° Si es una máquina de Mealy, dibujar el diagrama de Moore de la máquina de Moore equivalente.
3.° Minimizar el autómata resultante de la pregunta anterior.
4.° Si se ha seguido el algoritmo de minimización expuesto aquí se habrá obtenido la máquina de Moore en forma mínima. ¿Existe una máquina de Mealy equivalente que tenga menos estados? Si es así, dibujar su diagrama de Moore.

Capítulo 3
CIRCUITOS SECUENCIALES

1. LA REALIZACIÓN DE AUTÓMATAS FINITOS

Los AF que hemos estudiado en el capítulo anterior no son, en definitiva, más q modelos matemáticos para sistemas digitales con memoria. Ya mencionábamos en el capítulo 1 el doble interés de su estudio: teórico (herramienta matemática para formalizar el estudio de los lenguajes) y práctico (realización de sistemas digitales que cumplen funciones especificadas).

La realización de un sistema digital puede enfocarse de dos maneras:

a) Puede construirse físicamente el sistema, partiendo de elementos sencillos.
b) Puede utilizarse un ordenador, que es un sistema digital de uso general, y programarse para que efectúe las funciones que se desean.

Por ejemplo, si el autómata que se quiere realizar es un ordenador, se seguirá, normalmente, el enfoque a). (También puede seguirse el enfoque b), y en ese caso se trataría de simulación o emulación en otro ordenador). Si el autómata es muy complicado (como es un ordenador), se descompondrá generalmente en subautómatas; uno de ellos podría ser el sumador binario serie, ya estudiado a nivel teórico, del que veremos su realización en el apartado 6.1. (Conviene señalar que los ordenadores suelen utilizar otro tipo de sumador, el paralelo, que exige unos circuitos más complicados, pero que es mucho más rápido).

Otro ejemplo puede ser el reconocimiento de cadenas. Hemos visto un AF que reconoce la cadena «010». Podemos pensar en un AF que reconozca varias cadenas, dando una salida diferente para cada una. Así, si $E = \{A, B, \ldots, Z\}$ y $S = \{00000, 00001, \ldots, 11111\}$, podría dar $s = 00000$ para $x =$ CAR (cargar), $s = 00001$ para $x =$ CMP (complementar), $s = 00010$ para $x =$ SUM (sumar) y así, sucesivamente, todos los códigos binarios de operación de algún ordenador para los correspondientes códigos de su lenguaje ensamblador. Este AF sería parte de un AF más completo,

llamado «ensamblador», que traduciría a lenguaje de máquina los programas escritos en lenguaje ensamblador. Pues bien, los autómatas ensambladores, caso particular de procesadores de lenguajes, no se construyen físicamente, sino mediante un programa de ordenador.

En este capítulo nos vamos a dedicar especialmente a las técnicas para realizar autómatas finitos físicamente. La realización del AF, salvo raras excepciones, es con tecnología electrónica o electromecánica, y el sistema digital con memoria resultante se llama circuito secuencial.

2. ELEMENTOS DE UN CIRCUITO SECUENCIAL

2.1. Tipos de elementos

La realización de las funciones definidas por un AF implica dos tipos de elementos:

a) Elementos combinacionales, que realizan las funciones lógicas en las que no interviene el tiempo. Por ejemplo, la función h: $Q \rightarrow S$ debe dar la salida asociada a cada estado, sin intervención del tiempo, por lo que es una función lógica puramente combinacional, representable por una tabla de verdad y realizable con los métodos estudiados en el capítulo 3 del tema «Lógica».

b) Elementos con memoria, para realizar las funciones en las que interviene el tiempo. Así, f: $E \times Q \rightarrow Q$ debe dar, para unos valores dados de E y Q *en el instante* t el valor resultante de Q *en* $t + 1$ (recordemos que este «1» se refiere a una escala de tiempos arbitraria), es decir, debe calcular el valor resultante de Q y *memorizarlo* para darlo en $t + 1$.

Pasemos a estudiar el funcionamiento de algunos de estos elementos.

2.2. Elementos combinacionales

Son las puertas lógicas NOT, AND, OR, NAND, NOR, etc., ya estudiadas, cuyo funcionamiento debe ser bien conocido. Realizadas generalmente mediante tecnología electrónica, para aplicaciones especiales aún se utilizan otras tecnologías: electromecánica, hidráulica, neumática, etc.

2.3. Elementos con memoria

2.3.1. *Líneas de retardo*

El elemento con memoria más sencillo (desde el punto de vista conceptual) es aquel en que se tiene $E = Q = S = \{0, 1\}$ y la salida en cualquier instante es igual al estado en ese instante e igual a la entrada retardada un intervalo de tiempo θ:

$$s(t + \theta) = q(t + \theta) = e(t),$$

con e, q, $s \in \{0, 1\}$. Este sencillo modo de funcionamiento se puede ilustrar gráficamente con un cronograma como el de la figura 3.1.

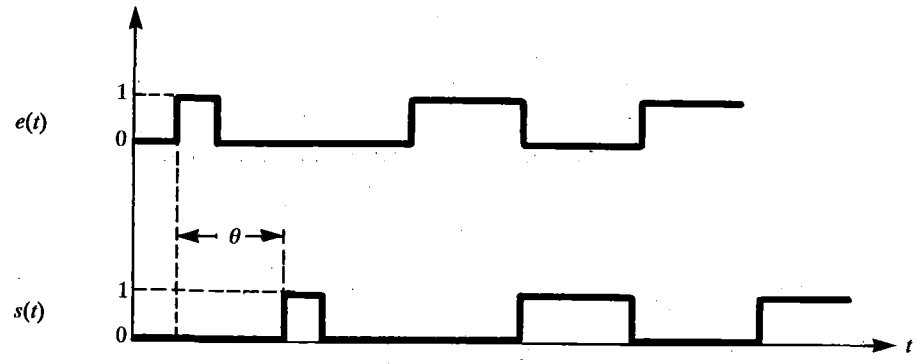


FIGURA 3.1.

Estos elementos (que, de una manera general, se llaman líneas de retardo) pueden realizarse mediante líneas de retardo acústicas, líneas de transmisión, redes de condensadores e inductancias, puertas lógicas conectadas en serie, etc., y también con biestables, como veremos enseguida.

2.3.2. *Biestables*

Los elementos de memoria más utilizados en los circuitos secuenciales son los biestables electrónicos. Un biestable es un circuito con dos estados estables que son también su salida y que, simbólicamente, se denominan «0» y «1» (en la realidad, cada uno de ellos corresponderá a un determinado nivel de tensión). «Estables» quiere decir que el elemento sólo cambia de estado bajo la acción de las entradas. Hay muy diversos tipos de biestables, que difieren entre sí por los posibles símbolos de entrada y las funciones de transición. Todos ellos tienen dos líneas de salida: en una de ellas, denominada habitualmente «Q», se tiene el nivel de tensión correspondiente en cada momento al estado, 0 ó 1; en la otra, $\overline{Q}$, se tiene su complemento. Pasemos a ver el funcionamiento de algunos biestables.

Biestable tipo SR («set-reset»). En este biestable $E = \{00, 01, 10\}$. Como las señales en los circuitos secuenciales son binarias, tendrá dos líneas de entrada, S y R (figura 3.2 a). Su funcionamiento puede expresarse formalmente, mediante la función de transición:

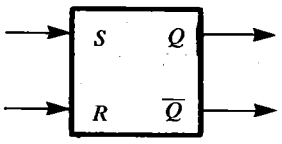
$$Q(t + 1) = S(t) + \overline{R}(t) \cdot Q(t) \qquad (R \cdot S = 0),$$

(aquí la unidad de tiempo es el intervalo que tarda el biestable en cambiar de estado o «tiempo de basculamiento»), o gráficamente, mediante el diagrama de Moore (figura

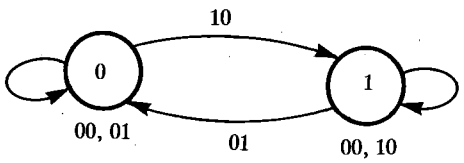
3.2 b), o mediante un cronograma (figura 3.2 c). Expresado con palabras, el funcionamiento es:

— $S = 0$, $R = 0$ hace que el biestable no cambie de estado.
— $S = 1$, $R = 0$ pone a 1 («set») el biestable.
— $S = 0$, $R = 1$ lo pone a 0 o repone («reset»).
— $S = 1$, $R = 1$ es una entrada no permitida.

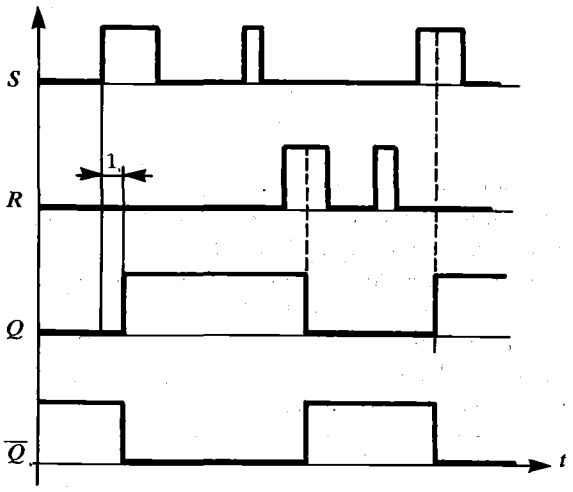
Es fácil diseñar, con los métodos ya conocidos, un circuito lógico que realice al biestable *RS*: basta minimizar la función de transición, función de S, R y Q y realimentar $Q(t + 1)$ a la entrada. Un posible circuito es el de la figura 3.2 d.



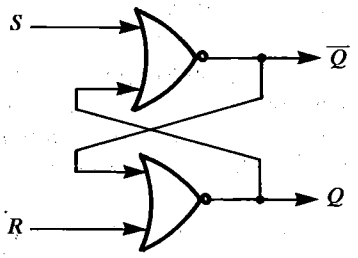
(a)



(b)



(c)



(d)

FIGURA 3.2.

Biestable tipo JK. Es como el *SR*, sólo que aquí la entrada $J = 1$, $K = 1$ está permitida, y su efecto consiste en cambiar el estado que tuviera anteriormente el biestable (figura 3.3; en el cronograma, para no complicar demasiado su interpretación, se ha despreciado el tiempo de basculamiento).

Hay que advertir que este biestable no se utiliza nunca con la realización de la

figura 3.3 d, sino sincronizado, como se verá más adelante. La razón es que si se mantienen las entradas $J = K = 1$ el circuito se convierte en un oscilador con una frecuencia que depende del tiempo de basculamiento del biestable.

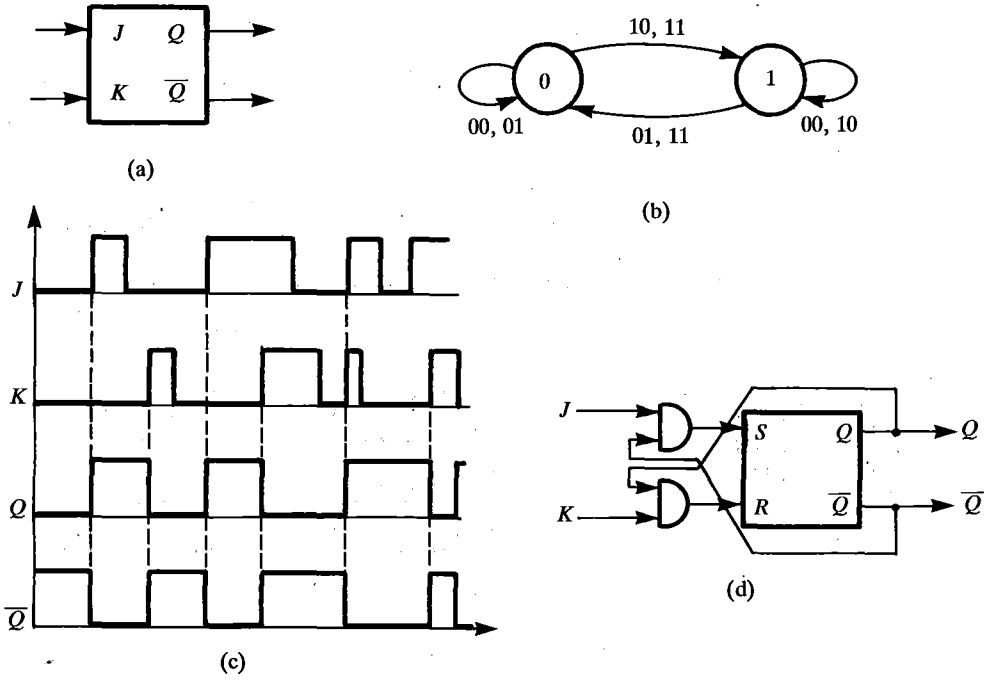


FIGURA 3.3.

Biestable tipo D. También tiene dos líneas de entrada, ya que $E = \{00, 01, 10, 11\}$. Estas líneas se llaman D («data») y C («clock») (figura 3.4 a). Su funcionamiento puede representarse formalmente así:

$$Q(t + 1) = D(t) \cdot C(t) + Q(t) \cdot \overline{C}(t),$$

que corresponde al diagrama de la figura 3.4 b. Ahora bien, aquí hay una diferencia muy importante en cuanto al significado de los símbolos «0» y «1» para D y para C. Para D, «0» significa, por ejemplo, una tensión de 0 v, y «1», 5 v; sin embargo, para C «1» significa que hay *una variación de tensión* de 0 v a 5 v, y «0» que no hay tal variación. En otras palabras, el valor de C es cero *salvo en el instante* en que su tensión cambia de nivel bajo a nivel alto. Véase la figura 3.4 c, para ilustrar este significado de «0» y «1».

Un cronograma que ilustra el funcionamiento del biestable tipo D es el de la figura 3.4 d, donde hemos, también, despreciado el tiempo de basculamiento. Obsérvese que si los cambios en D van retardados ligeramente un intervalo constante $\varepsilon \ll \theta$ con

relación a los cambios de 0 a 1 de C, lo que hace el biestable es exactamente retardar la entrada D en θ segundos, siendo θ el período del reloj (C).

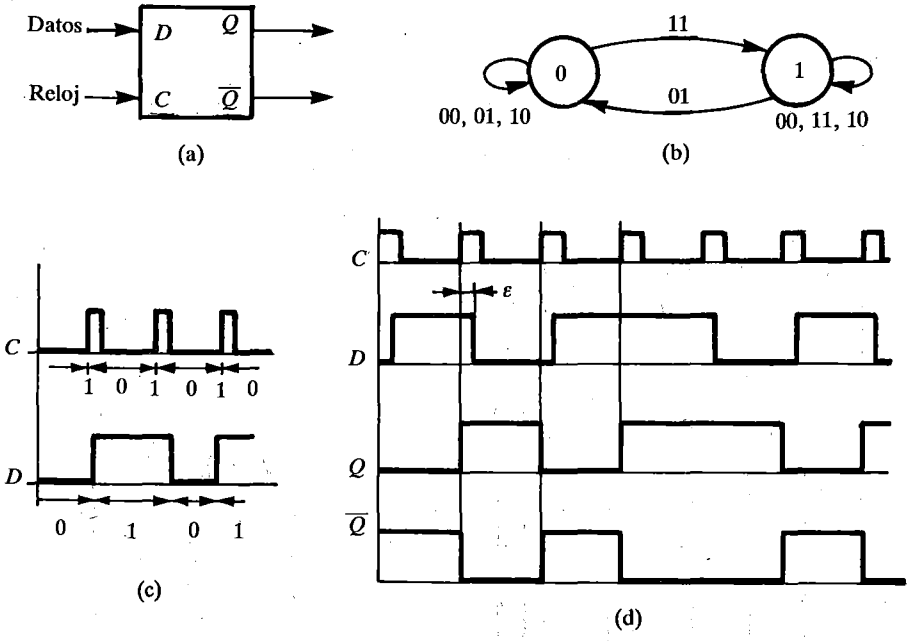


FIGURA 3.4.

Este tipo de biestable, que sólo cambia de estado cuando cambia el nivel de la señal de reloj se llama *sincronizado por flancos* («edge triggered»). El que hemos visto está sincronizado por el flanco de subida; también lo hay sincronizado por el flanco de bajada (es decir, cambia de estado cuando C pasa de 1 a 0). No entramos ya en su circuitería lógica, que el lector interesado puede consultar en la bibliografía.

Biestable tipo JK sincronizado. Tiene 5 líneas de entrada: S, R, J, K y C (figura 3.5 a). S y R se utilizan para ponerlo a 1 ó a 0 de una manera asíncrona, es decir, independientemente de C, mientras que J y K son la puesta a 1 ó a 0 síncronas. La ecuación que describe su comportamiento es:

$$Q(t+1) = S + \overline{R}\,\overline{K}\,Q + \overline{R}\,\overline{C}\,Q + \overline{R}\,J\,C\,\overline{Q}$$

Aquí el valor lógico «1» para C significa una doble transición nivel bajo-nivel alto-nivel bajo. Esto es debido a la constitución interna, en la que no entramos, que incluye dos biestables en una configuración llamada «maestro-esclavo». Por ello, las transiciones de la señal de salida, Q, se dan en el flanco de bajada de C. El cronograma de la figura 3.5 b ilustra el funcionamiento síncrono (S y R actúan como en un SR normal, y, en caso de conflicto, predominan sobre las JK).

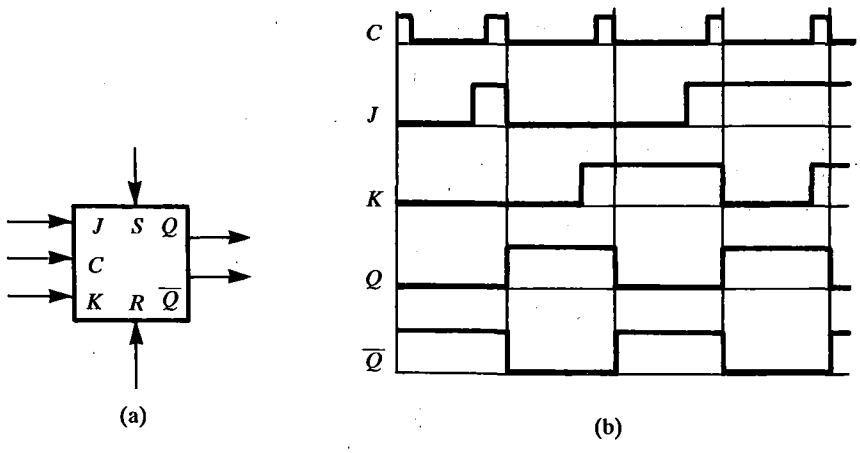


FIGURA 3.5.

3. MODELOS BÁSICOS DE CIRCUITOS SECUENCIALES

Recordemos que las funciones que definen un AF son:

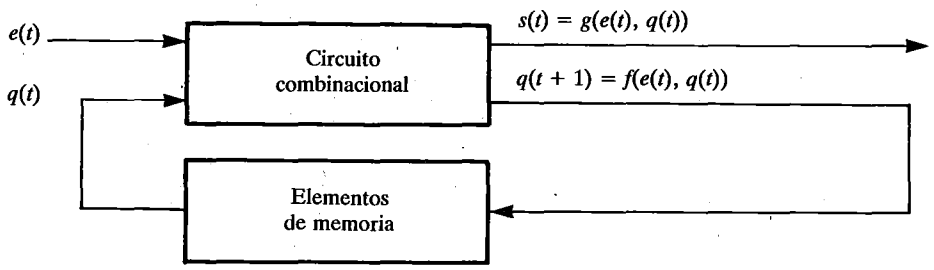
$$q(t+1) = f(e(t), q(t))$$
$$s(t) = g(e(t), q(t))$$

De ellas podemos deducir un primer modelo general para circuitos secuenciales (figura 3.6 a); consiste en disociar la parte combinacional, realizable mediante circuitos lógicos que calculan f y g, de la parte de memoria, que retarda en una unidad de tiempo $q(t)$ y $s(t)$. Este sería el modelo de Mealy, porque si existe una función de salida h: $Q \rightarrow S$, tal que $s(t) = h(q(t))$, podemos establecer el modelo de Moore (figura 3.6 b).

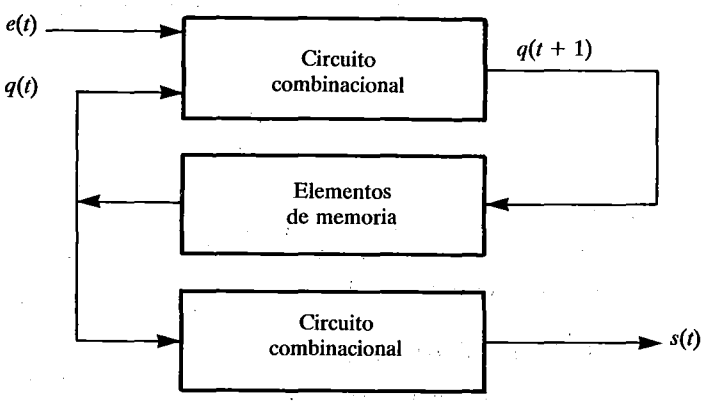
Supongamos que E tiene 3 símbolos, a, b, c; como los circuitos combinacionales son binarios, tendremos que codificarlos, haciendo, por ejemplo, $a = 00$, $b = 01$, $c = 10$. Por tanto, la entrada «e» será en realidad, 2 hilos de entrada binaria: uno para el primer dígito del código, y otro para el segundo. En general, si E tiene ℓ elementos, tendremos n hilos, con n tal que $2^{n-1} < \ell \leq 2^n$. Del mismo modo, habremos de suponer que en la realización con tecnología binaria serán necesarios, en general, m hilos para la salida y p hilos para el estado. De acuerdo con esto, el modelo básico de Mealy en forma de diagrama de bloques será el de la figura 3.7, en donde ya se supone que por todos los hilos las señales son binarias, y de igual modo podríamos dibujar el modelo básico de Moore. Para prescindir en la escritura de la variable t hemos adoptado el convenio, que seguiremos en adelante, de llamar z_i a $q_i(t+1)$.

Con estas nuevas variables binarias la función f será:

$$z_i = f_i(e_1, \ldots, e_n, q_1, \ldots, q_p), \qquad i = 1, \ldots, m$$



(a)



(b)

FIGURA 3.6.

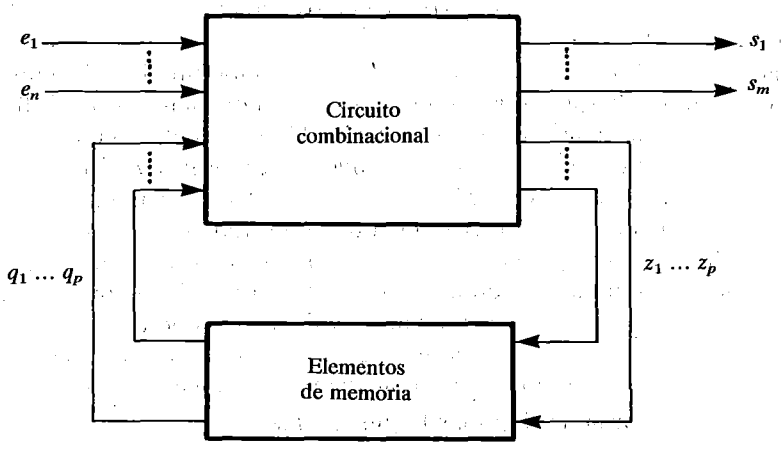


FIGURA 3.7.

y análogamente g y h. Si definimos los vectores columna **z, q, e, s** tendremos con notación vectorial:

$$\mathbf{z} = f(\mathbf{e}, \mathbf{q})$$
$$\mathbf{s} = g(\mathbf{e}, \mathbf{q})$$
$$\mathbf{s} = h(\mathbf{g})$$

4. TIPOS DE CIRCUITOS SECUENCIALES

En el apartado 2.3 hemos visto dos tipos de elementos con memoria: los no sincronizados, y los sincronizados con una señal de reloj, C. Por otra parte, las señales de entrada pueden ser de cuatro tipos: impulsionales síncronas o asíncronas y de nivel síncronas o asíncronas (figura 3.8). Si se utilizan unidades de memoria sincronizadas lo normal es que las señales de entrada sean síncronas, y así tenemos cuatro tipos de circuitos secuenciales:

— Tipo 1. *Síncronos impulsionales.* (Elementos de memoria sincronizados y entradas impulsionales síncronas).
— Tipo 2. *Síncronos de nivel.* (Igual que los anteriores, pero entradas de nivel).
— Tipo 3. *Asíncronos impulsionales.* (Elementos de memoria no sincronizados y entradas impulsionales asíncronas).
— Tipo 4. *Asíncronos de nivel.* (Como el tipo 3, con entradas de nivel).

Las salidas serán síncronas (impulsionales o de nivel) para el tipo 1, y lo mismo para el 2, asíncronas (impulsionales o de nivel) para el tipo 3 y necesariamente asíncronas de nivel para el tipo 4.

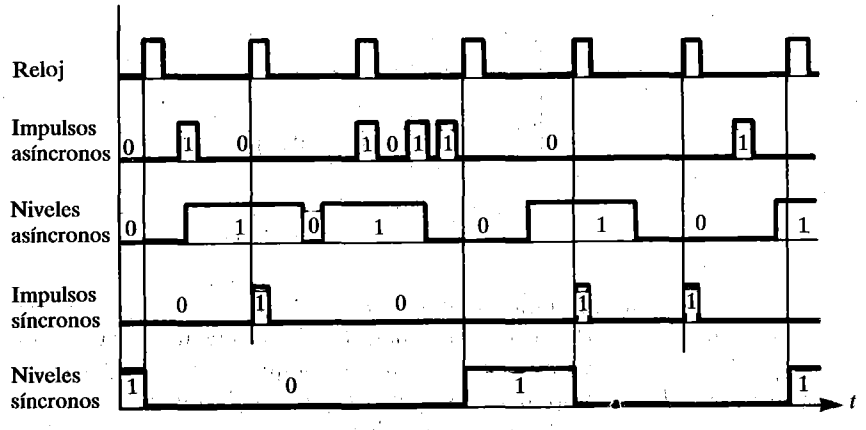


FIGURA 3.8.

Dependiendo de las características particulares de cada aplicación se utiliza un tipo u otro de circuito secuencial. Aquí nos vamos a limitar para los ejemplos exclusivamente a los tipos 1 y 2.

5. ANÁLISIS DE CIRCUITOS SECUENCIALES

El análisis de un circuito secuencial consiste en obtener su salida para una determinada cadena de entrada, o bien obtener su representación ya sea formal, ya en diagramas o tablas. El problema es conceptualmente muy fácil, y vamos a ilustrarlo con un ejemplo sencillo.

Consideremos el circuito secuencial de la figura 3.9.

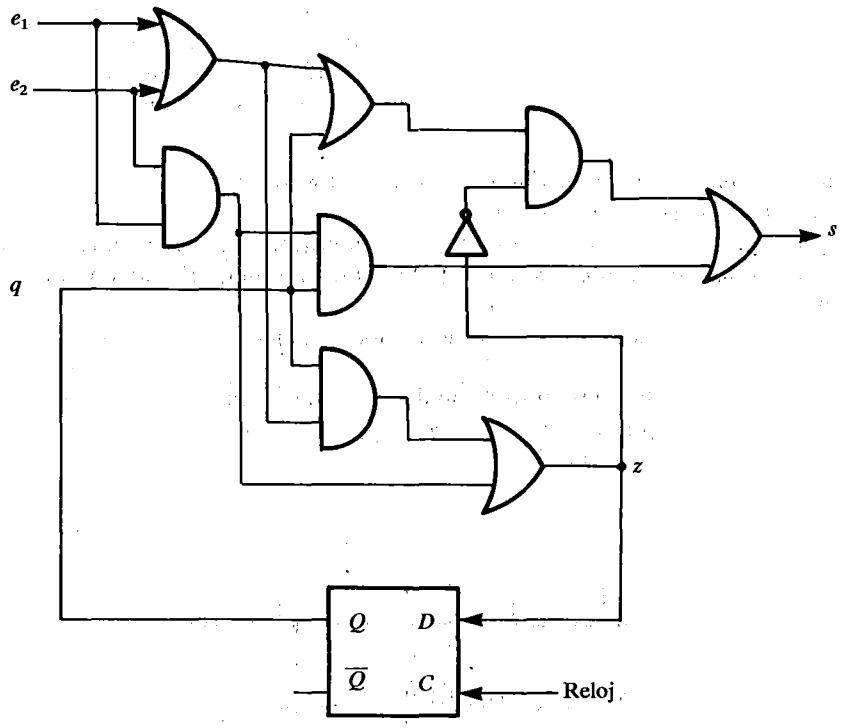


FIGURA 3.9.

De la parte combinacional vemos que $s = e_1 \cdot e_2 \cdot q + \overline{z} \cdot (e_1 + e_2 + q)$, y $z = e_1 \cdot e_2 + q \cdot (e_1 + e_2)$, y de la parte de memoria, como es un biestable tipo D, $q = z \cdot C + q \cdot \overline{C}$, es decir, q mantiene su valor hasta que C pasa de 0 a 1, instante en el que q pasa a valer z. Supongamos que las señales de entrada son de nivel (sincronizadas con C), que el estado inicial del biestable es $q = 0$, y que introducimos las cadenas de entrada $x_1 = 100110$, $x_2 = 101110$. Durante el primer intervalo de

tiempo $q = 0$, $z = 1 \cdot 1 + 0 \cdot (1 + 1) = 1$, $s = 1 \cdot 1 \cdot 0 + 0 \cdot (1 + 1 + 0) = 0$. Al llegar el siguiente impulso de reloj q pasa a valer 1 (pues $z = 1$), $z = 0 \cdot 0 + 1 \cdot (0 + 0) = 0$; $s = 0 \cdot 0 \cdot 1 + 1 \cdot (0 + 0 + 1) = 1$. Si vamos anotando gráficamente estos resultados obtenemos el cronograma de la figura 3.10.

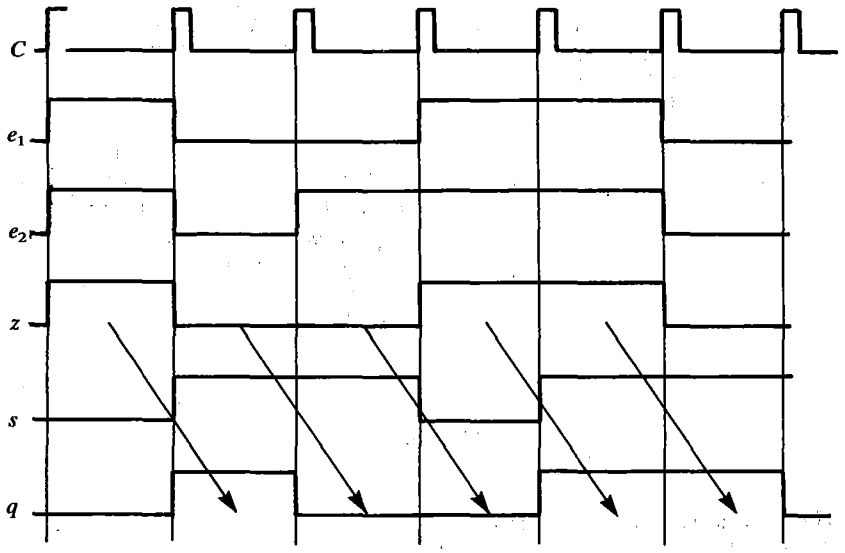


FIGURA 3.10.

El cronograma es muy útil para ver gráficamente el comportamiento del circuito, pero sólo lo representa para una determinada cadena de entrada. Para tener una representación más general de este circuito podemos poner en forma de tabla todas las posibles combinaciones de e_1, e_2 y q y las resultantes para s y z. Esta tabla (figura 3.11 a) nos conduce inmediatamente a la tabla de transición (figura 3.11 b) y al diagrama de Moore (figura 3.11 c). Obsérvese que este diagrama es precisamente el del autómata sumador binario, es decir, el circuito de la figura 3.9 es la realización física de tal autómata. (La parte combinacional es una etapa de sumador).

6. SÍNTESIS DE CIRCUITOS SECUENCIALES

6.1. Pasos de la síntesis

El diseño de un circuito secuencial se lleva a cabo siguiendo los siguientes pasos:

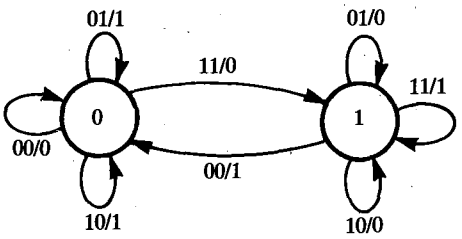
a) A partir de las especificaciones, obtener una tabla de transiciones y/o un diagrama de Moore.
b) Minimizar el AF.

e_1	e_2	q	s	z
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a)

q \ e	00	01	10	11
0	0/0	0/1	0/1	1/0
1	0/1	1/0	1/0	1/1

(b)



(c)

FIGURA 3.11.

c) Hacer una asignación de estados.
d) Escribir la tabla de transición en binario.
e) En función del tipo de biestable utilizado, obtener las tablas de excitación, que dan las entradas necesarias al biestable para cada transición.
f) De las tablas de excitación, obtener las ecuaciones lógicas de la parte combinacional.
g) Obtener las ecuaciones lógicas de $s = h(q)$.

Dada la diversidad de posibles especificaciones, no existe un método para el paso a).

Para el paso b) puede seguirse el algoritmo estudiado en el apartado 5 del capítulo 2.

El paso c) significa lo siguiente: Supongamos que el AF minimizado en el paso b) tiene N estados. Como los elementos de memoria de que disponemos son binarios, necesitaremos p elementos, con $2^{p-1} < N \leq 2^p$. La asignación de estados es una codificación arbitraria de los N estados con p dígitos binarios. El problema aquí es que según se haga una u otra codificación de las $(2^p!)/(2^p - N)!$ posibles, el circuito combinacional resultante puede ser más o menos complicado, y que no existe un método para saber cuál es la codificación óptima, aunque sí hay algunas reglas en las que no vamos a entrar, que el lector interesado puede estudiar en la bibliografía.

El paso e) consiste en obtener las tablas de verdad de la parte combinacional.

Para los pasos f) y g) aplicaremos las técnicas ya conocidas de diseño de circuitos lógicos.

6.2. Ejemplos

6.2.1. *Detector de paridad*

El diagrama de Moore es el de la figura 2.6. Si asignamos a los estados los valores $q_1 = 0$ y $q_2 = 1$, la tabla de transición en binario será:

q \ e	0	1
0/0	0	1
1/1	1	0

Escribamos la tabla de una forma lineal, es decir, con una línea para cada combinación posible de e y q:

e	q	z
0	0	0
0	1	1
1	0	1
1	1	0

Supongamos ahora que vamos a utilizar un biestable tipo D (sólo hace falta uno, puesto que sólo hay dos estados). Deberemos anotar en cada línea la entrada necesaria al biestable para que tenga lugar la transición correspondiente. Por ejemplo, en la primera línea, de $q(t) = 0$ se debe pasar a $z = q(t + 1) = 0$. Para ello, $D = 0$; en la segunda línea $D = 1$, etc. Así, podemos poner la tabla:

e	q	D
0	0	0
0	1	1
1	0	1
1	1	0

que nos define D (entrada del biestable) en función de $e(t)$ y $g(t)$, y se llama *tabla de excitaciones*. De ella obtenemos la función D:

$$D = \overline{e} \cdot q + e \cdot \overline{q} = e \oplus q,$$

y, como $s = q$, la realización del circuito será la de la figura 3.12 a, o bien la de la figura 3.12 b. En realidad, hemos tomado $s = z$, lo que nos permite obtener s con un intervalo de adelanto.

Obsérvese que, debido al modo de funcionamiento del biestable D, la tabla de excitaciones se obtiene, simplemente, sustituyendo «z» por «D», ya que la salida del biestable en $t + 1$ es justamente su entrada en t. No ocurre lo mismo con el JK. En efecto, si tenemos que realizar la transición de $q = 0$ a $z = 0$, las entradas J y K

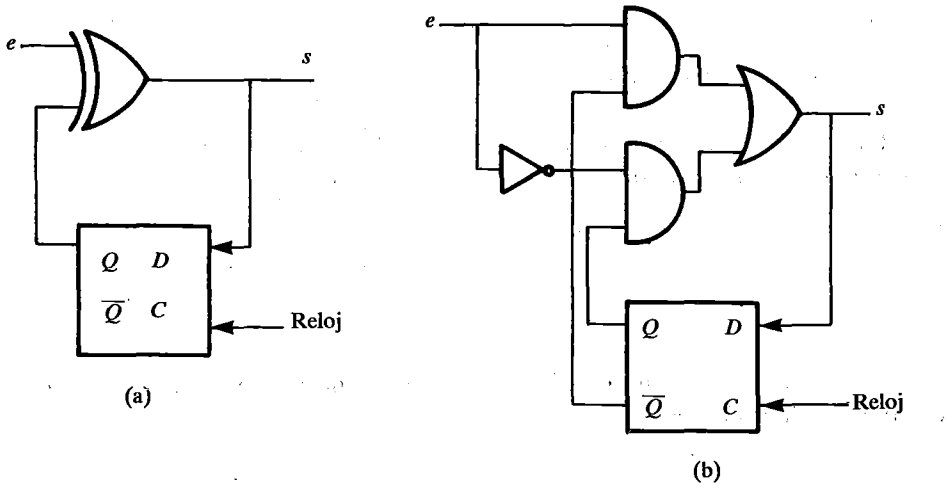
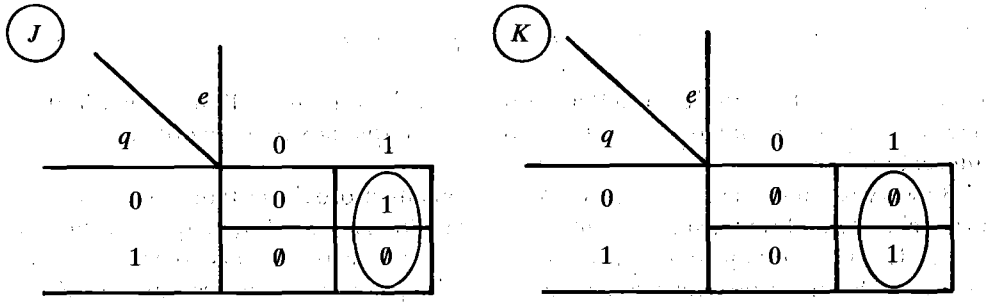


FIGURA 3.12.

pueden ser ambas 0, o bien $J = 0$, $K = 1$ (puesta a cero); para la transición de $q = 0$ a $z = 1$ podemos hacer $J = 1$, $K = 0$ (puesta a uno), o bien $J = 1$, $K = 1$ (complementación), etc. Así, podemos escribir la tabla de excitaciones para este ejemplo:

e	q	z	J	K
0	0	0	0	∅
0	1	1	∅	0
1	0	1	1	∅
1	1	0	∅	1

en la que hemos puesto «∅» en los lugares donde es indiferente que sea «0» ó «1»; como ya sabemos, esto nos ayuda en la minimización del circuito. Las tablas de Karnaugh de J y K en función de e y q son:



de las que obtenemos

$$J = K = e,$$

por lo que la realización con JK será la de la figura 3.13

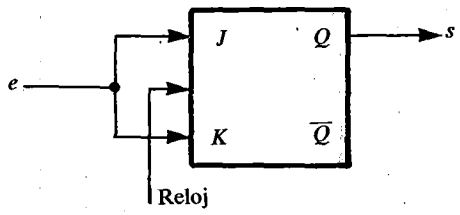


FIGURA 3.13.

6.2.2. *Reconocedor de 010*

El diagrama de Moore del AF reconocedor de la cadena 010 es el de la figura 2.12. Como hay 4 estados necesitaremos 2 biestables. Hagamos la siguiente asignación de estados: $q_1 = 00$; $q_2 = 01$; $q_3 = 10$; $q_4 = 11$. Con ello podemos escribir la tabla de transiciones en binario, y, al mismo tiempo, para cada transición, las entradas de los biestables; si éstos van a ser JK obtenemos:

e	q	s	z	$J_1\ K_1$	$J_2\ K_2$
0	00	0	01	0 ∅	1 ∅
0	01	0	01	0 ∅	∅ 0
0	10	0	11	∅ 0	1 ∅
0	11	1	01	∅ 1	∅ 0
1	00	0	00	0 ∅	0 ∅
1	01	0	10	1 ∅	∅ 1
1	10	0	00	∅ 1	0 ∅
1	11	1	10	∅ 0	∅ 1

Las 4 últimas columnas con las dos primeras definen las tablas de excitaciones. Si llamamos Q_1 y Q_2 (salidas de los biestables) al primero y segundo dígito de q, minimizando por Karnaugh podemos hallar las ecuaciones lógicas de las entradas de los biestables en función de e, Q_1 y Q_2; el resultado es:

$$J_1 = e \cdot Q_2;\ K_1 = e \oplus Q_2;\ J_2 = \overline{e};\ K_2 = e$$

Por otra parte, de las 3 primeras columnas se tendrá la función de salida, es decir, s en función de e, Q_1, Q_2. Minimizando resulta:

$$s = Q_1 \cdot Q_2$$

Con lo cual podemos trazar el circuito de la figura 3.14.

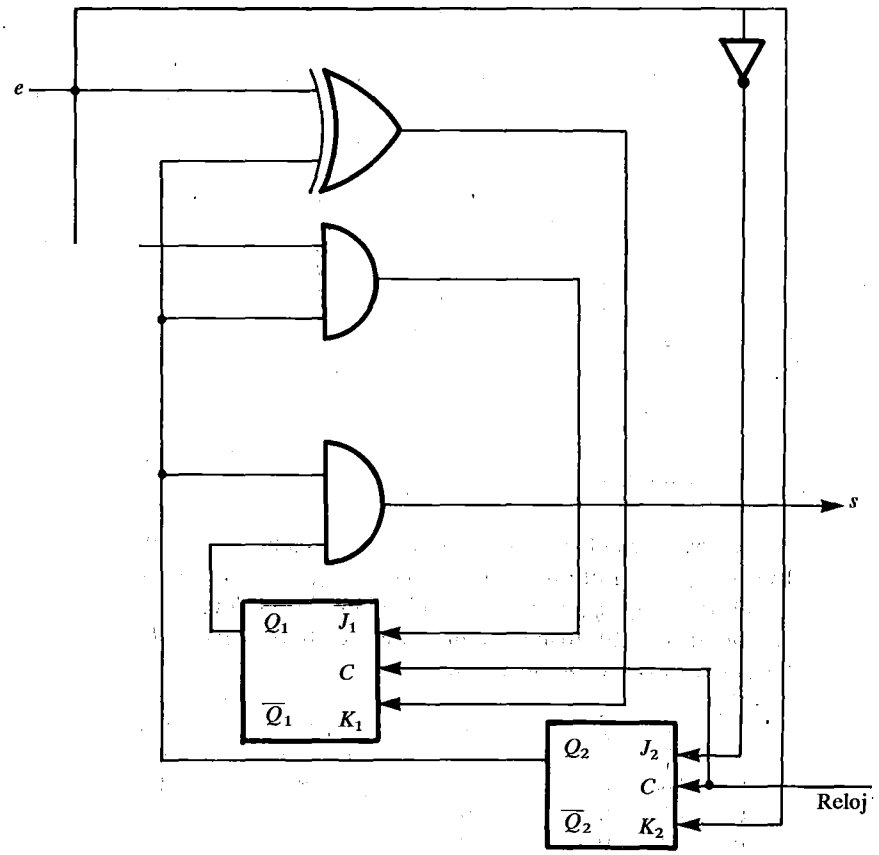


FIGURA 3.14.

6.2.3. *Contador BCD módulo 10*

Partimos del diagrama de Moore de la figura 2.17. Como hay 10 estados necesitaremos 4 biestables ($2^3 < 10 < 2^4$), que supondremos del tipo *JK*. Codifiquemos los estados con los mismos dígitos que las salidas asociadas, es decir:

$$\begin{aligned} q_0 &= s_0 = 0000 \\ q_1 &= s_1 = 0001 \\ q_2 &= s_2 = 0010 \\ &\vdots \\ q_9 &= s_9 = 1001 \end{aligned}$$

De esta manera se simplifica al máximo el circuito que realiza la función de salida, puesto que será $s = q$.

Escribimos a continuación la tabla de transición junto con la de excitaciones:

e	Q_3	Q_2	Q_1	Q_0	Z	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	0000	0	∅	0	∅	0	∅	0	∅
0	0	0	0	1	0001	0	∅	0	∅	0	∅	∅	0
0	0	0	1	0	0010	0	∅	0	∅	∅	0	0	∅
0	0	0	1	1	0011	0	∅	0	∅	∅	0	∅	0
0	0	1	0	0	0100	0	∅	∅	0	0	∅	0	∅
0	0	1	0	1	0101	0	∅	∅	0	0	∅	∅	0
0	0	1	1	0	0110	0	∅	∅	0	∅	0	0	∅
0	0	1	1	1	0111	0	∅	∅	0	∅	0	∅	0
0	1	0	0	0	1000	∅	0	0	∅	0	∅	0	∅
0	1	0	0	1	1001	∅	0	0	∅	0	∅	∅	0
1	0	0	0	0	0001	0	∅	0	∅	0	∅	1	∅
1	0	0	0	1	0010	0	∅	0	∅	1	∅	∅	1
1	0	0	1	0	0011	0	∅	0	∅	∅	0	1	∅
1	0	0	1	1	0100	0	∅	1	∅	∅	1	∅	1
1	0	1	0	0	0101	0	∅	∅	0	0	∅	1	∅
1	0	1	0	1	0110	0	∅	∅	0	1	∅	∅	1
1	0	1	1	0	0111	0	∅	∅	0	∅	0	1	∅
1	0	1	1	1	1000	1	∅	∅	1	∅	1	∅	1
1	1	0	0	0	1001	∅	0	0	∅	0	∅	1	∅
1	1	0	0	1	0000	∅	1	0	∅	0	∅	∅	1

Tenemos así en forma de tablas de verdad J_i y K_i en función de e, Q_1, Q_2, Q_3, Q_4. Obsérvese que no están todas las combinaciones de variables binarias de estado, debido a que sólo tenemos 10 estados de los 16 posibles con 4 variables. Para tales combinaciones podemos tomar cualquier valor (∅) en las J y K.

La tabla de Karnaugh de J_3 será:

Q_1Q_0 \ Q_3Q_2	00	01	11	10
00	0	0	∅	∅
01	0	0	∅	∅
11	0	0	∅	∅
10	0	0	∅	∅

$e = 0$

Q_1Q_0 \ Q_3Q_2	00	01	11	10
00	0	0	∅	∅
01	0	0	∅	∅
11	0	1	∅	∅
10	0	0	∅	∅

$e = 1$

Y de ella deducimos:

$$J_3 = e \cdot Q_2 \cdot Q_1 \cdot Q_0$$

Y, análogamente,

$$J_2 = e \cdot Q_1 \cdot Q_0; \qquad K_2 = e \cdot Q_1 \cdot Q_0$$
$$J_1 = e \cdot \overline{Q}_3 \cdot Q_0; \qquad K_1 = e \cdot Q_0$$
$$J_0 = e; \qquad K_0 = e$$
$$K_3 = e \cdot Q_0$$

Resulta así el circuito de la figura 3.15.

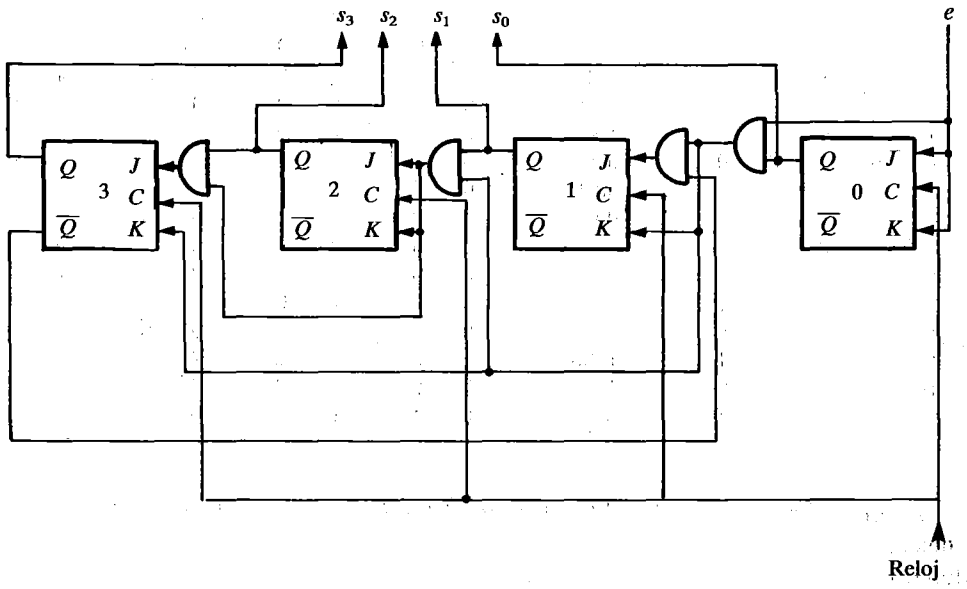


FIGURA 3.15.

7. RESUMEN

Hemos estudiado algunos de los elementos utilizados como memorias en la realización física de los AF (circuitos secuenciales). Conociendo el funcionamiento de los elementos que lo componen, el análisis de un circuito secuencial se puede llevar a cabo deduciendo paso a paso el cronograma correspondiente a una determinada cadena, y también se puede obtener el diagrama de Moore, que tendrá, si p es el número de elementos binarios de memoria, 2^p estados.

Hemos visto el procedimiento general para la síntesis de circuitos secuenciales, ilustrándolo con algunos ejemplos. No hemos abordado, por su mayor complejidad, el diseño de máquinas incompletamente especificadas ni el problema de la asignación de estados.

8. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

Véase el apartado 11 del capítulo 3 del tema «Lógica».

9. EJERCICIOS

9.1. Diseñar el sumador binario serie utilizando un biestable *JK*.

9.2. En el ejercicio 10 del capítulo 2, supóngase que los tres posibles símbolos de entrada se codifican en binario: 1 = 01; 2 = 10; 3 = 11 (es decir, habrá dos hilos de entrada). Diseñar el circuito secuencial correspondiente.

9.3. Diseñar un contador módulo 10 en exceso de 3, utilizando *JK*. (Véase el ejercicio 12.4 del tema «Lógica»).

9.4. Diseñar contadores módulo 10 en BCD natural y en exceso de 3, con biestables tipo *D*.

9.5. Diseñar circuitos con biestables *JK* y *D* para un autómata retardador de 2 intervalos ($s(t) = e(t-2)$).

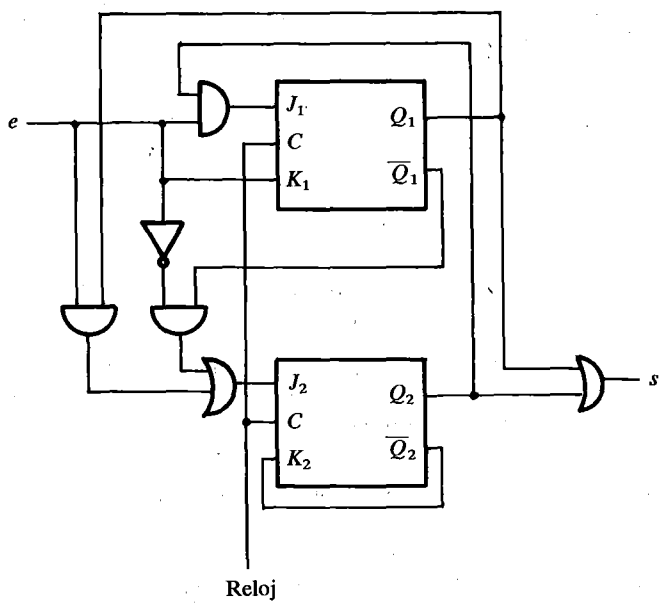
9.6. Diseñar un circuito que gobierne el funcionamiento de tres luces (verde, roja y amarilla) reguladoras de tráfico. La luz verde deberá permanecer encendida durante 40 seg, con la roja apagada; al cabo de este tiempo la situación cambiará (verde apagada, roja encendida) durante otros 40 seg, y así sucesivamente. La luz amarilla sólo se encenderá (simultáneamente con la verde) durante los diez segundos que preceden al encendido de la roja. Para ello, el circuito tendrá tres salidas binarias (*V, R, A*) y una sola entrada consistente en impulsos de período 10 seg.

9.7. Diseñar un circuito secuencial que simule la situación descrita para el «castillo encantado» (apartado 2.3 del capítulo 2).

9.8. Dado el autómata definido por la tabla que hay a continuación, expresar con palabras su funcionamiento (es decir, cuándo da 0 ó 1 a la salida dependiendo de la secuencia de ceros y unos a la entrada), y diseñar un circuito para materializarlo.

	0	1
$q_1/0$	q_2	q_1
$q_2/0$	q_3	q_1
$q_3/1$	q_3	q_1

9.9. Analizar el circuito de la figura que se acompaña y ver si es posible diseñar otro que, realizando la misma función, sea más sencillo.



9.10. Hacer el análisis de los diseños de los contadores BCD natural (apartado 6.2.3) y BCD exceso de 3 (ejercicios 3 y 4) para poder decir qué ocurre si, por causa de una perturbación exterior o de un fallo de funcionamiento, se pasa a alguno de los 6 estados no permitidos.

Capítulo 4
AUTOMATAS RECONOCEDORES Y LENGUAJES REGULARES

1. RECONOCEDOR FINITO

1.1. Definición

Hemos definido al principio del libro un *lenguaje*, L, sobre un alfabeto, A, como un subconjunto cualquiera de A^*. En este capítulo vamos a ver que ciertos lenguajes pueden asociarse con autómatas finitos que sirven como reconocedores de las cadenas pertenecientes a tales lenguajes.

Un reconocedor finito de un lenguaje L es un AF que sólo acepta las cadenas de dicho lenguaje, en el sentido de que, inicializado en un estado predeterminado, q_1, si se le introduce una cadena de entrada $x_1 \in L$, da un símbolo final de salida que corresponde a «aceptación» (por ejemplo, $s = 1$), mientras que para $x_1 \notin L$ produce una salida de «no aceptación» (por ejemplo, $s = 0$). En adelante supondremos siempre que hablemos de reconocedores que se trata de máquinas de Moore. Podemos entonces hacer abstracción del alfabeto de salida y de la función g, y considerar el subconjunto de estados $F \subset Q$ que producen la salida de aceptación, a los que llamaremos estados finales. De acuerdo con esto, daremos la siguiente definición formal:

Un *reconocedor finito* es una quíntupla

$$R = \langle E, Q, f, q_1, F \rangle$$

donde:

E es un conjunto finito (alfabeto de entrada).
Q es un conjunto finito (conjunto de estados).

f es una función f: $E \times Q \rightarrow Q$ (función de transición).
$q_1 \in Q$ es un estado designado como estado inicial.
$F \subset Q$ es un conjunto de estados designados como estados finales.

Llamaremos $L(R)$ al conjunto de cadenas aceptadas por R, es decir,

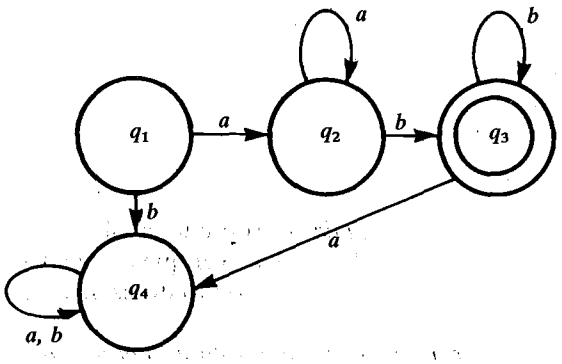
$$L(R) = \{x \in E^* | f(x, q_1) \in F\}$$

(con el dominio de f ampliado a E^*).

Seguiremos el convenio de representar en los diagramas de Moore los estados de aceptación con un círculo doble, y en las tablas de transición, encerrados en un círculo.

1.2. Ejemplos

1.

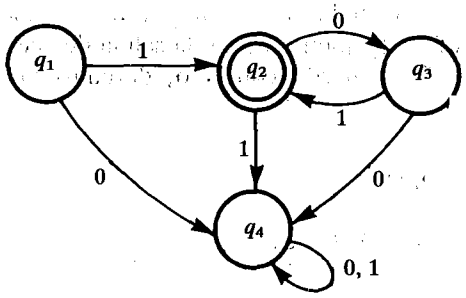


	a	b
q_1	q_2	q_4
q_2	q_2	q_3
(q_3)	q_4	q_3
q_4	q_4	q_4

FIGURA 4.1.

$$L(R_1) = \{ab, aab, \ldots, abb, abbb, \ldots, aabb, \ldots\} = \{a^n b^m | n \geq 1, m \geq 1\}$$

2.

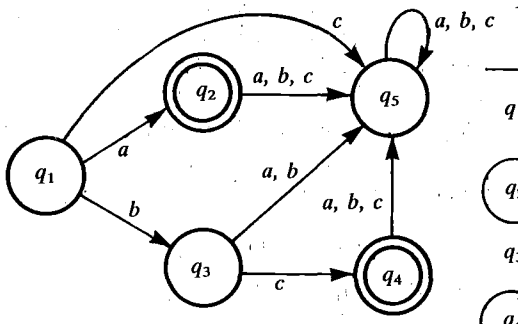


	0	1
q_1	q_4	q_2
(q_2)	q_3	q_4
q_3	q_4	q_2
q_4	q_4	q_4

FIGURA 4.2.

$$L(R_2) = \{1, 101, 10101, \ldots\} = \{1(01)^n | n \geq 0\}$$

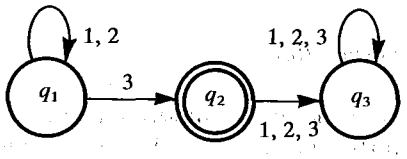
3.



	a	b	c
q_1	q_2	q_3	q_5
q_2	q_5	q_5	q_5
q_3	q_5	q_5	q_4
q_4	q_5	q_5	q_5
q_5	q_5	q_5	q_5

FIGURA 4.3.

$$L(R_3) = \{a,\ bc\}$$



	1	2	3
q_1	q_1	q_1	q_2
q_2	q_3	q_3	q_3
q_3	q_3	q_3	q_3

FIGURA 4.4.

$$L(R_4) = \{3, 13, 113, 1113, \ldots, 123, 1223, \ldots, 213, 2113, \ldots\} =$$
$$= \{1^{n_1}, 2^{n_2}, 1^{n_3}, 2^{n_4}, \ldots, 3 | n_1, n_2, \ldots, \geq 0\}$$

2. LENGUAJES ACEPTADOS POR RECONOCEDORES FINITOS

2.1. Planteamiento del problema

Podemos preguntarnos si, dado un lenguaje cualquiera, $L \subset E^*$, podemos siempre encontrar un reconocedor finito, R, tal que $L(R) = L$. Observemos que esa pregunta ya la habíamos dejado planteada, en términos más generales, en el capítulo 2, cuando, al final del apartado 3.1, decíamos: dada una máquina ¿podemos encontrar su circuito?

Baste un ejemplo para demostrar que existen lenguajes a los que no corresponde ningún reconocedor finito. Consideremos $E = \{0, 1\}$, y sea $L = \{1^{n^2} | n \geq 1\}$. Supongamos que existe un R, con p estados, tal que $L(R) = L$. Para cadenas del tipo $x = 1^i$ tendremos que $f(1^i, q_1) \in F$ si i es un cuadrado perfecto. Para las $p + 1$ cadenas $x_0 = 1^0, x_1 = 1^1, \ldots, x_p = 1^p$ tendremos como estados resultantes $f(x_0, q_1), f(x_1, q_1), \ldots, f(x_p, q_1)$, pero como el reconocedor sólo tiene p estados, al menos dos de estos estados resultantes deberán ser el mismo: $f(1^i, q_1) = f(1^j, q_1)$, con $j - i \leq p$. Entonces, si el reconocedor acepta 1^{n^2} también deberá aceptar $1^{n^2 + (j-i)}$, ya que

$$f(1^{n^2}, q_1) = f(1^i 1^{n^2-i}, q_1) = f[1^{n^2-i}, f(1^i, q_1)] =$$
$$= f[1^{n^2-i}, f(1^j, q_1)] = f(1^j 1^{n^2-i}, q_1) = f(1^{n^2+(j-i)}, q_1).$$

Ahora bien, siempre podemos tomar n suficientemente grande como para que $(n + 1)^2 - n^2 > p$, y, por tanto $(n + 1)^2 - n^2 > (j - i)$, con lo que $n^2 + (j - i) < (n + 1)^2$ no será un cuadrado perfecto, y el reconocedor responderá incorrectamente al aceptar $1^{n^2+(j-i)}$.

Vemos pues que *los lenguajes aceptados por algún reconocedor finito son un subconjunto de todos los lenguajes posibles sobre* E^*.

Ahora podemos preguntarnos: *¿existe alguna propiedad que caracterice a los lenguajes que son aceptados por un reconocedor finito?* Si tal propiedad existe parece lógico pensar que pueda deducirse de la capacidad del reconocedor para responder de distinto modo a diferentes cadenas de entrada.

2.2. Relación equirrespuesta de un reconocedor finito

En el capítulo 2 definimos de una manera general el comportamiento de entrada-estados de un AF como un homomorfismo K: $\langle E^*, \rangle \rightarrow \langle Q^Q, \circ \rangle$, tal que a cada $x \in E^*$ corresponde una transformación entre estados, $K(x)$: $Q \rightarrow Q$, y esto nos permitió definir una relación equirrespuesta en E^* tal que $x \cong y$, si y sólo si $K(x) = K(y)$, es decir, las cadenas x e y estarán en relación si y sólo si producen la misma transformación entre estados:

$$(\forall q_i \in Q)[(x \cong y) \leftrightarrow f(x, q_i) = f(y, q_i)].$$

Veíamos que esta relación es una relación de congruencia en $\langle E^*, \rangle$ y particiona E^* en un número finito de clases de equivalencia (es decir, es de índice finito, menor o igual a n^n, siendo n el número de estados).

En el caso del reconocedor finito, el estado inicial, q_1, es fijo, y el *comportamiento de entrada-estados* será más bien una función K_R: $E^* \rightarrow Q$ que aplica a cada $x \in E^*$ un $q \in Q$ de tal manera que $K_R(x) = f(x, q_1)$. Podemos igualmente definir una *relación equirrespuesta,* $\cong_R$, en E^*:

$$(\forall x, y \in E^*)[(x \cong_R y) \leftrightarrow (f(x, q_1) = f(x, q_2))]$$

Esta relación, como es fácil ver, es una relación de equivalencia. Además $(x \cong y) \rightarrow (x \cong_R y)$, pero no a la inversa (se dice que $\cong$ *refina a* $\cong_R$), por lo que en la partición de

E^* inducida por $\cong_R$ habrá un número igual o inferior de clases de equivalencia que en la inducida por $\cong$.

Si bien $\cong$ es una relación de congruencia en $\langle E^*, \rangle$, $\cong_R$ sólo es una *relación de congruencia derecha*. Esto quiere decir que $(x \cong_R y) \rightarrow (xz \cong_R yz)$, pero en general, no es cierto que $(zx \cong_R zy)$. En efecto, si llamamos $q_i = f(x, q_1) = f(y, q_1)$ y $q_j = f(z, q_i)$, tendremos: $f(xz, q_1) = f(z, f(x, q_1)) = f(z, q_i) = q_j$ y análogamente $f(yz, q_1) = q_j$; sin embargo, $f(zx, q_1) = f[x, f(z, q_1)]$, que en general será diferente de $f[y, f(z, q_1)]$.

Así pues, a cada reconocedor finito corresponde una partición de E^* en clases de equivalencia tal que si dos cadenas están en la misma clase ambas cadenas conducen al mismo estado. Además, esta relación de equivalencia es de índice finito (menor o igual que n^n, siendo n el número de estados del reconocedor) y es una relación de congruencia derecha en E^*. Veamos cómo pueden aplicarse estas conclusiones para caracterizar a los lenguajes aceptados por reconocedores finitos.

2.3. Condición para que un lenguaje sea aceptado por un reconocedor finito

Dado un lenguaje $L \subset E^*$, definimos la *relación de congruencia derecha inducida por L*, $\cong_L$, así:

$$(\forall x, y, z \in E^*)[(x \cong_L y) \leftrightarrow (xz \in L \leftrightarrow yz \in L)]$$

Es evidente que se trata de una relación de equivalencia. Para ver que también es una congruencia derecha basta suponer que $z = z_1 z_2$, con lo que

$$(\forall x, y, z_1, z_2 \in E^*)[(x \cong_L y) \leftrightarrow (xz_1z_2 \in L \leftrightarrow yz_1z_2 \in L) \leftrightarrow (xz_1 \cong_L yz_1)]$$

Vamos a demostrar ahora la principal conclusión de este apartado 2:

$L \subset E^*$ es un lenguaje aceptado por un reconocedor finito si y sólo si la relación de congruencia derecha inducida por L tiene índice finito.

Veamos primero que si $L = L(R)$, entonces $\cong_L$ tiene índice finito:

a) Por definición de $\cong_R$,

$$(x \cong_R y) \leftrightarrow (f(x, q_1) = f(y, q_1))$$

b) Por ser $\cong_R$ una congruencia derecha,

$$(x \cong_R y) \rightarrow [f(xz, q_1) = f(yz, q_1)]$$

c) Es claro que

$$[f(xz, q_1) = f(yz, q_1)] \rightarrow [(f(xz, q_1) \in F) \leftrightarrow (f(yz, q_1) \in F)]$$

d) Y también que

$$[(f(xz, q_1) \in F) \leftrightarrow (f(yz, q_1) \in F)] \leftrightarrow [(xz \in L) \leftrightarrow (yz \in L)]$$

e) Por definición de $\cong_L$,

$$[(xz \in L) \leftrightarrow (yz \in L)] \leftrightarrow (x \cong_L y)$$

f) El razonamiento constituido por las premisas b), c), d) y e) nos lleva a la conclusión de que

$$(x \cong_R y) \rightarrow (x \cong_L y)$$

($\cong_R$ refina a $\cong_L$). Por consiguiente, el índice (número de clases de equivalencia) de la partición inducida en E^* por $\cong_L$ es igual o inferior al de la partición inducida por $\cong_R$, y como ésta es de índice finito, $\cong_L$ también lo será.

A la inversa, supongamos que L es un lenguaje que induce una relación de congruencia derecha en E^* que es de índice finito. Vamos a construir un reconocedor finito, R_L, tal que $L(R_L) = L$. Llamemos $[x]$ a la clase de equivalencia de $E^*/\cong_L$ que contiene a $x \in E^*$. Entonces definimos

$$R_L = \langle E, Q_L, f_L, q_1, F_L \rangle,$$

donde

$Q_L = E^*/\cong_L = \{[x]\}$ (finito, puesto que $\cong_L$ es de índice finito)
$(\forall y \in E^*)(f_L(y, [x]) = [xy])$
$q_1 = [\lambda]$
$F_L = \{[x] \mid x \in L\}$

(F_L está bien definida, pues si $x_1 \in L$ y $x_2 \in L$, según la definición de $\cong_L$, haciendo $z = \lambda$, vemos que $x_1 \cong_L x_2$, es decir, $[x_1] = [x_2]$).

El lenguaje aceptado por este reconocedor será:

$$L(R_L) = \{y \in E^* \mid f_L(y, q_1) \in F_L\} = \{y \mid f_L(y, [\lambda]) \in F_L\} =$$
$$= \{y \mid [y] \in F_L\} = \{y \mid y \in L\} = L$$

Además, R_L está en forma mínima. En efecto, si $q_{L_1} = [x_1]$ fuera equivalente a $q_{L_2} = [x_2]$, esto querría decir que

$$(\forall y \in E^*)((f_L(y, [x_1]) \in L) \leftrightarrow (f_L(y, [x_2]) \in L)),$$

y por la definición de f_L,

$$(\forall y \in E^*)(([x_1, y] \in L) \leftrightarrow ([x_2, y] \in L)),$$

es decir, $x_1 \cong_L x_2$; x_1 y x_2 están en la misma clase de equivalencia, por lo que $q_{L_1} = q_{L_2}$.

3. CONJUNTOS REGULARES Y EXPRESIONES REGULARES

3.1. Los problemas de análisis y de síntesis

Acabamos de ver una condición necesaria y suficiente para que un lenguaje sea aceptado por un reconocedor finito. El problema de análisis consiste en deducir el lenguaje asociado a un determinado reconocedor, y el de síntesis en encontrar un reconocedor cuyo lenguaje sea un lenguaje dado. Ambos problemas los tenemos en teoría resueltos. En efecto, para el análisis, basta enumerar las cadenas que son aceptadas, como vimos en los ejemplos de 1.2; para la síntesis, hay que encontrar las clases de equivalencia de la relación de congruencia derecha inducida por L y construir el reconocedor como se ha indicado más arriba. El inconveniente está en que, al poder ser L un conjunto infinito, no es fácil trabajar con él, y no siempre podemos representarlo de una manera condensada, como hacíamos en los ejemplos de 1.2; además, salvo en el caso de que L sea finito (como en el ejemplo 3), no tenemos un algoritmo para encontrar las clases de equivalencia inducidas por L.

En este apartado vamos a exponer una herramienta, las expresiones regulares, especialmente introducida para trabajar con los lenguajes aceptados por reconocedores finitos, y que nos permitirá llegar a algoritmos para resolver los problemas de análisis y de síntesis.

3.2. Conjuntos regulares

En primer lugar vamos a definir tres operaciones en el conjunto $\{L_1, L_2, \ldots\}$ de subconjuntos de E^* (lenguajes sobre E):

a) Unión:

$$L_1 \cup L_2 = \{x | x \in L_1 \vee x \in L_2\}$$

b) Concatenación:

$$L_1 L_2 = \{x_1 x_2 | x_1 \in L_1 \wedge x_2 \in L_2\}$$

c) Cierre u operación estrella:

$$L^* = \{\lambda\} \cup \{L\} \cup \{LL\} \cup \{LLL\} \ldots = \bigcup_{n=0}^{\infty} L^n$$

Decimos que un subconjunto de E^*, $L_R \subset E^*$, es regular si y sólo si:

a) L_R es un subconjunto *finito* de E^* (puede ser $L = \emptyset$), o bien;
b) L_R puede obtenerse a partir de subconjuntos finitos de E^* mediante un número finito de operaciones de unión, concatenación y cierre.

3.3. Expresiones regulares

Las expresiones regulares se introducen para describir los conjuntos regulares, y como éstos son lenguajes, las expresiones regulares serán metalenguajes.

Seguiremos el convenio de designar por $|\alpha|$ al conjunto descrito por la expresión regular α.

Definimos ahora el conjunto de expresiones regulares sobre un alfabeto $E = \{e_1, \ldots, e_n\}$ y las operaciones suma, concatenación y cierre de la siguiente manera recursiva:

a) λ, $\emptyset$ y e_i $(i = 1, \ldots, n)$ son expresiones regulares tales que $|\lambda| = \{\lambda\}$; $|\emptyset| = \emptyset$ y $|e_i| = \{e_i\}$ $(i = 1, \ldots, n)$ (*);
b) si α y β son expresiones regulares, $\alpha + \beta$ es una expresión tal que $|\alpha + \beta| = |\alpha| \cup |\beta|$;
c) si α y β son expresiones regulares, $\alpha\beta$ es una expresión regular tal que $|\alpha\beta| = |\alpha|\,|\beta|$;
d) si α es una expresión regular, α^* es una expresión regular tal que $|\alpha^*| = |\alpha|^*$.

Como estas tres operaciones corresponden a las utilizadas para definir los conjuntos regulares, a todo conjunto corresponderá al menos una expresión regular.

Veamos algunos ejemplos.

	E	*Expresión regular,* α	*Conjunto regular,* $\lvert\alpha\rvert$
1.	$\{a, b\}$	aa^*bb^*	Conjunto de todas las cadenas de E^* constituidas por «*a*» seguido de «*a*» cualquier número de veces (o ninguna), seguido de «*b*» y seguido de «*b*» cualquier número de veces (o ninguna).
2.	$\{0, 1\}$	$1(01)^*$	Conjunto de cadenas que empiezan por «1» y sigue (01) cualquier número de veces (o ninguna).
3.	$\{a, b, c\}$	$a + bc$	$\{a, bc\}$.

(*) λ es la *cadena vacía* (elemento neutro para la concatenación de cadenas), y $\emptyset$ es el *conjunto vacío* (elemento neutro para la unión de conjuntos).

	E	*Expresión regular,* α	*Conjunto regular,* $\|\alpha\|$
4.	{1, 2, 3}	(1 + 2)*3	Conjunto de cadenas formadas con los símbolos 1 y 2 sucediéndose cualquier número de veces (y en cualquier orden), y siempre terminando la cadena con el símbolo 3.
5.	$\{e_1, e_2, \ldots, e_n\}$	$(e_1 + e_2 + \ldots + e_n)^*$	E^*.
6.	{0, 1}	(01)*	Conjunto formado por λ y todas las cadenas constituidas por la cadena 01 repetida cualquier número de veces.
7.	{0, 1}	0*10*	Conjunto de todas las cadenas que tienen un «1» (y sólo uno).

Obsérvese que los cuatro primeros ejemplos corresponden exactamente a los cuatro ejemplos del apartado 1.2.

Dos expresiones regulares son iguales si designan al mismo conjunto regular:

$$(\alpha = \beta) \leftrightarrow |\alpha| = |\beta|$$

Teniendo esto presente, es fácil demostrar las siguientes propiedades de las expresiones regulares:

1. Asociatividad de la concatenación:

$$\alpha(\beta\gamma) = (\alpha\beta)\gamma$$

2. Distributividad de la suma:

$$\alpha\beta + \alpha\gamma = \alpha(\beta + \gamma);$$
$$\beta\alpha + \gamma\alpha = (\beta + \gamma)\alpha$$

3. $\emptyset$ es elemento neutro para la suma:

$$\alpha + \emptyset = \emptyset + \alpha = \alpha$$

4. $\emptyset$ es un cero para la concatenación:

$$\alpha\emptyset = \emptyset\alpha = \emptyset$$

5. λ es elemento neutro para la concatenación:

$$\alpha\lambda = \lambda\alpha = \alpha$$

6. Propiedades de la operación cierre:

a) $(\alpha + \beta)^* = (\alpha^* + \beta^*)^* = (\alpha^*\beta^*)^*$
b) $(\alpha + \lambda)^* = \alpha^* + \lambda = \alpha^*$
c) $\alpha\alpha^* + \lambda = \alpha^*$
d) $\lambda^* = \emptyset^* = \lambda$

Por ejemplo, para demostrar que $(\alpha + \beta)^* = (\alpha^*\beta^*)^*$ tendremos en cuenta que:

$$|(\alpha + \beta)^*| = \{\lambda\} \cup |\alpha + \beta| \cup |\alpha + \beta|^2 \cup \ldots =$$
$$= \{\lambda\} \cup |\alpha| \cup |\beta| \cup |\alpha|^2 \cup |\beta|^2 \cup |\alpha||\beta| \cup |\beta||\alpha| \cup \ldots$$

y, por otra parte,

$$|(\alpha^*\beta^*)^*| = \{\lambda\} \cup |\alpha^*\beta^*| \cup |\alpha^*\beta^*|^2 \cup \ldots =$$
$$= \{\lambda\} \cup |\alpha^*||\beta^*| \cup |\alpha^*||\beta^*||\alpha^*||\beta^*| \cup \ldots =$$
$$= \{\lambda\} \cup |\alpha| \cup |\beta| \cup |\alpha|^2 \cup |\beta|^2 \cup |\alpha||\beta| \cup |\beta||\alpha| \cup \ldots$$

Pasemos ahora a ver que *todos los lenguajes aceptados por reconocedores finitos son conjuntos regulares y que todo conjunto regular es un lenguaje aceptado por un reconocedor finito, lo que nos va a permitir resolver los problemas de análisis y de síntesis, respectivamente.*

4. RESOLUCIÓN DE LOS PROBLEMAS DE ANÁLISIS Y DE SÍNTESIS DE UN RECONOCEDOR FINITO

4.1. Análisis

4.1.1. *Teorema de análisis*

Todo lenguaje aceptado por un reconocedor finito es un conjunto regular.

Supongamos que el reconocedor finito tiene n estados, $Q = \{q_1, \ldots, q_n\}$, con estado inicial q_1 y con estados finales $F = \{q_{f_1}, q_{f_2}, \ldots, q_{f_r}\}$ $(n \geq r \geq 0)$. El lenguaje aceptado es:

$$L(R) = \{x | f(x, q_1) \in F\} = \cup \{R_{1j} | q_j \in F\},$$

con $R_{ij} = \{x | f(x, q_i) = q_j\}$ (conjunto de cadenas que llevan del estado q_i al estado q_j). Basta entonces demostrar que los R_{ij} son conjuntos regulares.

Vamos a definir R_{ij}^k $(0 \leq k \leq n)$ como el conjunto de cadenas que llevan de q_i a q_j sin pasar por ningún q_l tal que $l > k$. En particular, R_{ij}^0 serán las cadenas que llevan de q_i a q_j sin pasar por ningún otro estado, por lo que son símbolos, es decir, es un subconjunto de $E \cup \{\lambda\}$ y, por tanto, es regular. Supongamos que R_{ij}^{k-1} es regular

para todo i y j y $k \geq 1$ y demostremos que entonces R_{ij}^k es regular. En efecto, basta comprobar que

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

Entonces, R_{ij}^k es regular para todo k, y, en particular, $R_{ij} = R_{ij}^n$ es regular.

4.1.2. *Algoritmo de análisis*

Como corolario del teorema anterior se desprende un procedimiento para obtener la expresión regular del lenguaje aceptado por un determinado reconocedor. En efecto, llamemos α_{ij}^k a la expresión regular que designa al conjunto R_{ij}^k: $|\alpha_{ij}^k| = R_{ij}^k$; entonces, si q_1 es el estado inicial, la expresión regular de $L(R)$ será:

$$\begin{cases} \alpha_{1f_1} + \alpha_{1f_2} + \ldots + \alpha_{1f_r} & \text{si } r > 0 \\ \emptyset \text{ si } r = 0 \end{cases}$$

Los α_{ij} se calcularán recursivamente teniendo en cuenta los conjuntos a los que representan:

$$\begin{aligned} |\alpha_{ij}^0| &= \{e \in E \cup \{\lambda\} \mid f(e, q_i) = q_j\} \\ \alpha_{ij}^k &= \alpha_{ij}^{k-1} + \alpha_{ik}^{k-1}(\alpha_{kk}^{k-1})^* \alpha_{kj}^{k-1}, \quad 0 < k \leq n \\ \alpha_{ij} &= \alpha_{ij}^n \end{aligned}$$

4.1.3. *Ejemplos*

Vamos a tomar los mismos cuatro ejemplos del apartado 1.2: partimos de los diagramas y debemos llegar a las expresiones regulares que ya conocemos.

1. Sólo hay un estado final, q_3, luego la expresión regular será:

$$\begin{aligned} \alpha_{13} &= \alpha_{13}^4 \\ \alpha_{13}^4 &= \alpha_{13}^3 + \alpha_{14}^3(\alpha_{44}^3)^* \alpha_{43}^3 \end{aligned}$$

Pero $\alpha_{43}^3 = \emptyset$, como se ve directamente en el diagrama de Moore, por lo que

$$\alpha_{13}^4 = \alpha_{13}^3 = \alpha_{13}^2 + \alpha_{13}^2(\alpha_{33}^2)^* \alpha_{33}^2$$

Calculemos pues, α_{13}^2 y α_{33}^2:

$$\begin{aligned} \alpha_{13}^2 &= \alpha_{13}^1 + \alpha_{12}^1(\alpha_{22}^1)^* \alpha_{23}^1 \\ \alpha_{13}^1 &= \alpha_{13}^0 + \alpha_{11}^0(\alpha_{11}^0)^* \alpha_{13}^0 = \emptyset + \lambda(\lambda)^* \emptyset = \emptyset \end{aligned}$$

(también se ve directamente en el diagrama)

$$\alpha_{12}^1 = \alpha_{12}^0 + \alpha_{11}^0(\alpha_{11}^0)^*\alpha_{12}^0 = a + \lambda(\lambda)^*a = a$$
$$\alpha_{22}^1 = \alpha_{22}^0 + \alpha_{21}^0(\alpha_{11}^0)^*\alpha_{12}^0 = a + \emptyset(\lambda)^*a = a$$
$$\alpha_{23}^1 = \alpha_{23}^0 + \alpha_{21}^0(\alpha_{11}^0)^*\alpha_{13}^0 = b + \emptyset(\lambda)^*\emptyset = b$$

Luego:

$$\alpha_{13}^2 = \emptyset + a(a^*)b = aa^*b$$

Podríamos calcular α_{33}^2 de manera análoga, pero ya se ve directamente en el diagrama que saldrá $\alpha_{33}^2 = b^*$. Por tanto,

$$\alpha_{13} = \alpha_{13}^4 = \alpha_{13}^3 = aa^*b + aa^*b(b^*)^*b^* =$$
$$= aa^*b + aa^*bb^* = aa^*b(\lambda + b^*) = aa^*bb^*$$

2. Como sólo hay un estado final, q_2, la expresión regular será:

$$\alpha_{12} = \alpha_{12}^4 = \alpha_{12}^3 + \alpha_{14}^3(\alpha_{44}^3)^*\alpha_{42}^3$$

Se ve sobre el diagrama que $\alpha_{42}^3 = \emptyset$, por lo que ya no tenemos necesidad de calcularla, ni de calcular α_{14}^3 ni α_{44}^3:

$$\alpha_{12} = \alpha_{12}^3 = \alpha_{12}^2 + \alpha_{13}^2(\alpha_{33}^2)^*\alpha_{32}^2$$

Omitimos los cálculos de α_{12}^2, α_{33}^2 y α_{32}^2, que se hacen por el mismo procedimiento de reducción, resultando:

$$\alpha_{12}^2 = 1;\ \alpha_{13}^2 = 10;\ \alpha_{33}^2 = 10;\ \alpha_{32}^2 = 1$$

(En los cálculos intermedios resulta una expresión de la forma $\emptyset^*$; no olvidar que $\emptyset^* = \lambda$, no $\emptyset$). Sustituyendo y aplicando las propiedades de las operaciones,

$$\alpha_{12} = 1 + 10(10)^*1 = (\lambda + 10(10)^*)1 = (10)^*1 = 1(01)^*$$

Dejamos como ejercicio la obtención de las expresiones regulares de los otros dos ejemplos. (En el ejemplo 3 hay dos estados finales, q_2 y q_4, por lo que la expresión será $\alpha_{12} + \alpha_{14} = \alpha_{12}^5 + \alpha_{14}^5 = \ldots$).

Como se habrá observado, el procedimiento es bastante engorroso para aplicarlo manualmente, pero es un algoritmo general que puede programarse para su ejecución automática. En la ejecución manual a veces puede simplificarse intuitivamente; así, cuando decíamos, en el ejemplo 1, que se ve directamente en el diagrama que $\alpha_{43}^3 = \emptyset$, lo que nos evita muchos cálculos. De hecho, si el diagrama no es muy complicado, es preferible obtener la expresión regular por simple inspección.

4.2. Síntesis

4.2.1. *Teorema de síntesis*

Todo conjunto regular es un lenguaje aceptado por un reconocedor finito.

Hay publicadas varias demostraciones de este teorema, bastante laboriosas todas ellas, por lo que, a fin de no alargar excesivamente este tema, nos permitimos no incluir ninguna, remitiendo al lector a las notas bibliográficas del apartado 6. En cambio, nos parece interesante dar un algoritmo que permite deducir directamente el reconocedor de un conjunto regular denotado por su expresión regular. Para ello necesitamos introducir un nuevo concepto: el de derivadas de una expresión regular.

4.2.2. *Derivadas de una expresión regular*

Consideremos una expresión regular, α, que designa a un conjunto regular L_R, $|\alpha| = L_R$, y consideremos el subconjunto de L_R formado por todas las cadenas de L_R que empiezan por un determinado símbolo, e. Definimos el *cociente izquierdo de L_R por e*, $L_R \backslash e$, como el conjunto resultante de suprimir e en todas esas cadenas:

$$L_R \backslash e = \{x | ex \in L_R\}$$

y definimos la *derivada de α respecto al símbolo e*, $D_e(\alpha)$, como la expresión regular de $L_R \backslash e$. (Es fácil ver que si L_R es regular, $L_R \backslash e$ también lo es).

Por ejemplo, sea $\alpha = abc + d + a^*c$, es decir, $L_R = |\alpha| = \{abc, d, c, ac, aac, \ldots\}$. Entonces,

$$L_R \backslash a = \{bc, c, ac, \ldots\};\ L_R \backslash b = \emptyset;\ L_R \backslash c = \lambda;\ L_R \backslash d = \lambda,$$

y las derivadas serán las respectivas expresiones regulares:

$$D_a(\alpha) = bc + a^*c;\ D_b(\alpha) = \emptyset;\ D_c(\alpha) = \lambda;\ D_d(\alpha) = \lambda.$$

Veamos algunas propiedades de las derivadas así definidas. De la definición es inmediato comprobar que:

a) $D_{e_1}(e_2) = \begin{cases} \lambda \text{ si } e_1 = e_2 \\ \emptyset \text{ si } e_1 \neq e_2 \end{cases}$

b) $D_e(\lambda) = D_e(\emptyset) = \emptyset$

c) $D_e(\alpha + \beta) = D_e(\alpha) + D_e(\beta)$

Ya no es tan inmediato calcular la derivada de la concatenación de dos expresiones regulares ni la del cierre de una expresión regular.

Supongamos que $\gamma = \alpha\beta$. Si $|\alpha|$ no contiene la cadena vacía, λ, al suprimir «e» en las cadenas de $|\alpha\beta|$ resultarán las cadenas derivadas de α (es decir, todas aquellas que

comiencen por «*e*», suprimiendo «*e*») concatenadas con las cadenas de β: $D_e(\alpha\beta) = [D_e(\alpha)]\beta$. Pero si $\lambda \in (\alpha)$, entonces $|\alpha\beta|$ contiene también a todas las cadenas de β, por lo que habrá que añadir $D_e(\beta)$. Si introducimos la expresión $\delta(\alpha)$ tal que

$$\delta(\alpha) = \emptyset \text{ si } \lambda \notin |\alpha|$$
$$\delta(\alpha) = \lambda \text{ si } \lambda \in |\alpha|$$

entonces podemos representar ambos casos en una sola expresión:

d) $D_e(\alpha\beta) = [D_e(\alpha)]\beta + \delta(\alpha)D_e(\beta)$

Finalmente, veamos cómo se calcula la derivada de la operación cierre. Sabemos que

$$|\alpha^*| = |\alpha|^* = \{\lambda\} \cup |\alpha| \cup |\alpha||\alpha| \cup |\alpha||\alpha||\alpha| \cup \ldots$$

por lo que

$$D_e(\alpha^*) = D_e(\lambda) + D_e(\alpha) + D_e(\alpha\alpha) + D_e(\alpha\alpha\alpha) + \ldots$$
$$D_e(\lambda) = \emptyset.$$

Si $\lambda \notin |\alpha|$

$$D_e(\alpha^*) = D_e(\alpha) + [D_e(\alpha)]\alpha + [D_e(\alpha)]\alpha\alpha + \ldots =$$
$$= D_e(\alpha)[\lambda + \alpha + \alpha\alpha + \ldots] = [D_e(\alpha)]\alpha^*$$

Si $\lambda \in |\alpha|$ llegamos al mismo resultado. En efecto:

$$D_e(\alpha^*) = D_e(\alpha) + [D_e(\alpha)]\alpha + D_e(\alpha) + [D_e(\alpha)]\alpha\alpha + D_e(\alpha\alpha) + \ldots$$

que se reduce a la misma expresión anterior teniendo en cuenta la idempotencia de la suma. Luego:

e) $D_e(\alpha^*) = [D_e(\alpha)]\alpha^*$

Teniendo en cuenta estas cinco propiedades se puede calcular la derivada de cualquier expresión regular sin tener que formar previamente el conjunto cociente.

Hemos visto la derivación respecto de un símbolo, $e \in E$; la operación se puede extender a derivación respecto a una cadena definiendo:

$$D_\lambda(\alpha) = \alpha$$
$$D_{xe}(\alpha) = D_e[D_x(\alpha)]$$

Se puede demostrar (por inducción sobre la longitud de las cadenas) que el conjunto de derivadas diferentes de una expresión regular, es decir, $\{D_x(\alpha)|x \in E^*\}$ es finito.

4.2.3. *Algoritmo de síntesis*

Para construir un reconocedor en forma mínima del lenguaje denotado por la expresión regular α, se puede seguir el siguiente procedimiento:

1. Calcular $\{D_x(\alpha) | x \in E^*\}$.
2. El estado inicial es $q_1 = \alpha$; los otros son las diferentes $D_x(\alpha)$.
3. La función de transición es $f(e, \alpha) = D_e(\alpha)$; $f(e, D_x(\alpha)) = D_{xe}(\alpha)$.
4. El conjunto de estados finales es $F = \{D_x(\alpha) | \lambda \in |D_x(\alpha)|\}$.

4.2.4. *Ejemplos*

Para ilustrar la aplicación del algoritmo anterior vamos a tratar los mismos cuatro ejemplos del apartado 1.2. Teníamos allí cuatro reconocedores; las expresiones regulares de los correspondientes lenguajes las obtuvimos en los apartados 3.3 y 4.1.3. Pues bien, ahora partiremos de esas expresiones y comprobaremos que, con el algoritmo de síntesis, llegamos a los diagramas originales.

Ejemplo 1.

$$\alpha = aa^*bb^* = (a)(a^*bb^*)$$

$$D_a(\alpha) = D_a(a)a^*bb^* + \delta(a)D_a(a^*bb^*) = \lambda a^*bb^* + \emptyset D_a(a^*bb^*) = a^*bb^*$$

$$D_b(\alpha) = D_b(a)a^*bb^* + \delta(a)D_b(a^*bb^*) = \emptyset a^*bb^* + \emptyset D_b(a^*bb^*) = \emptyset$$

$$\begin{aligned} D_{aa}(\alpha) &= D_a(a^*)bb^* + \delta(a^*)D_a(bb^*) = \\ &= D_a(a)a^*bb^* + \lambda\emptyset = \lambda a^*bb^* = a^*bb^* = D_a(\alpha) \end{aligned}$$

Al repetirse la derivada, ya no seguimos derivando respecto de a. Tampoco es necesario derivar $D_b(\alpha)$, puesto que, al ser $\emptyset$, cualquier derivada posterior será $\emptyset$. Nos queda pues por calcular $D_{ab}(\alpha)$:

$$\begin{aligned} D_{ab}(\alpha) &= D_b(a^*bb^*) = D_b(a^*)bb^* + \delta(a^*)D_b(bb^*) = \\ &= D_b(a)a^*bb^* + \lambda[D_b(b)b^* + \delta(b)D_b(b^*)] = \\ &= \emptyset a^*bb^* + \lambda b^* + \emptyset\lambda b^* = b^* \end{aligned}$$

Calcularemos ahora las derivadas terceras a partir de $D_{ab}(\alpha)$ (puesto que ésta es la única derivada segunda que no es igual a ninguna anterior).

$$D_{aba}(\alpha) = D_a(b^*) = D_a(b)b^* = \emptyset b^* = \emptyset$$

$$D_{abb}(\alpha) = D_b(b^*) = D_b(b)b^* = b^* = D_{ab}(\alpha)$$

Al salir una $\emptyset$ y la otra repetida ya no es preciso que sigamos derivando.
Según el algoritmo, el conjunto de estados será:

$$Q = \{\alpha, D_a(\alpha), D_b(\alpha), D_{ab}(\alpha)\}$$

De las derivadas calculadas, la única que contiene la cadena vacía es $D_{ab}(\alpha) = b^*$, por lo que

$$F = \{D_{ab}(\alpha)\}$$

Y la función de transición será:

$$\begin{aligned}
&f(a, \alpha) = D_a(\alpha); f(b, \alpha) = D_b(\alpha)\\
&f(a, D_a(\alpha)) = D_{aa}(\alpha) = D_a(\alpha)\\
&f(b, D_a(\alpha)) = D_{ab}(\alpha)\\
&f(a, D_b(\alpha)) = D_{ba}(\alpha) = D_b(\alpha)\\
&f(b, D_b(\alpha)) = D_{bb}(\alpha) = D_b(\alpha)\\
&f(a, D_{ab}(\alpha)) = D_{aba}(\alpha) = D_b(\alpha)\\
&f(b, D_{ab}(\alpha)) = D_{abb}(\alpha) = D_{ab}(\alpha)
\end{aligned}$$

Podemos así dibujar el diagrama de la figura 4.5, que es el mismo de la figura 4.1, con $q_1 = \alpha$; $q_2 = D_a(\alpha)$; $q_3 = D_{ab}(\alpha)$; $q_4 = D_b(\alpha)$.

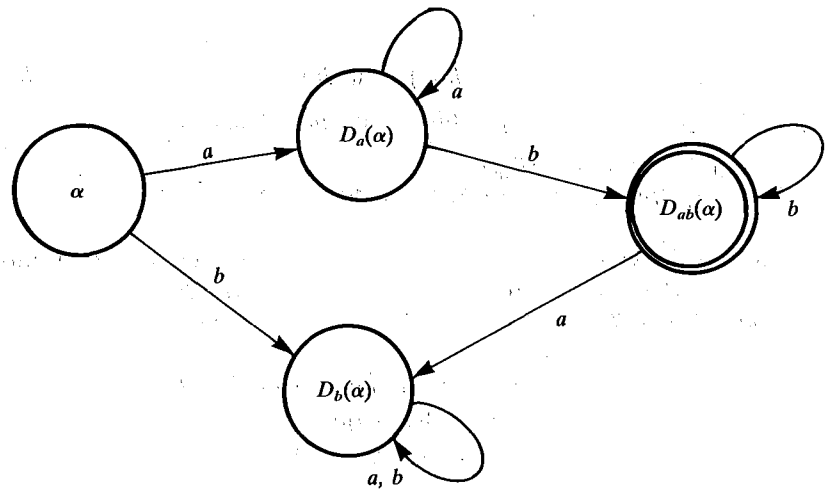


FIGURA 4.5.

Ejemplo 2.

La expresión regular es $\alpha = 1(01)^*$. Calculemos las derivadas hasta que aparezcan repetidas:

$$\begin{aligned}
D_0(\alpha) &= D_0(1)(01)^* + \delta(1)D_0(01)^* = \emptyset\\
D_1(\alpha) &= D_1(1)(01)^* + \delta(1)D_1(01)^* = (01)^*
\end{aligned}$$

$$D_{00}(\alpha) = D_{01}(\alpha) = \emptyset$$
$$D_{10}(\alpha) = D_0[(01)^*] = [D_0(01)](01)^* = 1(01)^* = \alpha$$
$$D_{11}(\alpha) = D_1[(01)^*] = \emptyset$$

Por tanto,

$$Q = \{\alpha, D_0(\alpha), D_1(\alpha)\}$$
$$F = \{D_1(\alpha)\}$$

La función de transición será:

$$f(0, \alpha) = D_0(\alpha)$$
$$f(1, \alpha) = D_1(\alpha)$$
$$f(0, D_0(\alpha)) = D_{00}(\alpha) = D_0(\alpha)$$
$$f(1, D_0(\alpha)) = D_{01}(\alpha) = D_0(\alpha)$$
$$f(0, D_1(\alpha)) = D_{10}(\alpha) = \alpha$$
$$f(1, D_1(\alpha)) = D_{11}(\alpha) = D_0(\alpha)$$

Llegamos así a un diagrama de Moore como el de la figura 4.6 que, aparentemente, corresponde a un reconocedor diferente del original (figura 4.2). No hay ningún error. Lo que ocurre es que este algoritmo que estamos aplicando nos da el reconocedor en forma mínima, y el de la figura 4.2 no lo está.

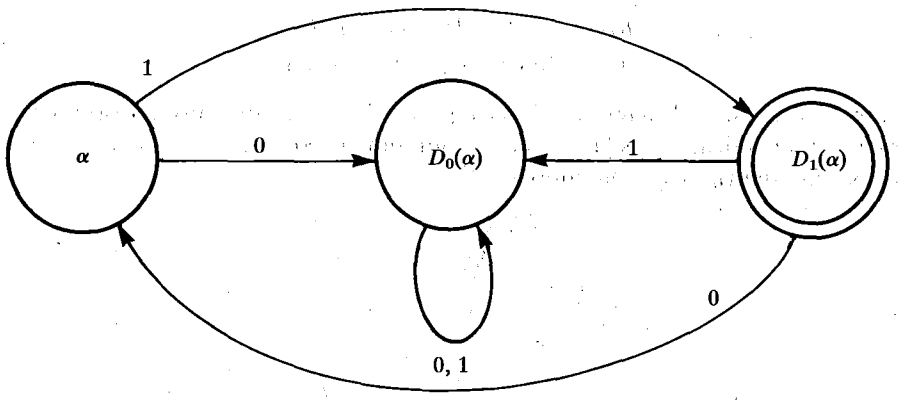


FIGURA 4.6.

Invitamos al lector a aplicar el algoritmo de minimización explicado en el capítulo 2 al AF de la figura 4.2 para ver que se llega al mismo AF de la figura 4.6.

Ejemplo 3.

$$\alpha = a + bc$$

Derivadas:

$$
\begin{aligned}
D_a(\alpha) &= D_a(a) + D_a(bc) = \lambda + \emptyset = \lambda \\
D_b(\alpha) &= D_b(a) + D_b(bc) = \emptyset + c = c \\
D_c(\alpha) &= D_c(a) + D_c(bc) = \emptyset \\
D_{aa}(\alpha) &= D_{ab}(\alpha) = D_{ac}(\alpha) = \emptyset = D_c(\alpha) \\
D_{ba}(\alpha) &= D_a(c) = \emptyset = D_c(\alpha) \\
D_{bb}(\alpha) &= D_b(c) = \emptyset = D_c(\alpha) \\
D_{bc}(\alpha) &= D_c(c) = \lambda = D_a(\alpha) \\
D_{ca}(\alpha) &= D_{cb}(\alpha) = D_{cc}(\alpha) = \emptyset = D_c(\alpha)
\end{aligned}
$$

$$
\begin{aligned}
Q &= \{\alpha, D_a(\alpha), D_b(\alpha), D_c(\alpha)\} \\
F &= \{D_a(\alpha)\}
\end{aligned}
$$

Función de transición:

$$
\begin{aligned}
f(a, \alpha) = D_a(\alpha);\ f(b, \alpha) &= D_b(\alpha);\ f(c, \alpha) = D_c(\alpha) \\
f(a, D_a(\alpha)) &= D_{aa}(\alpha) = D_c(\alpha) \\
f(b, D_a(\alpha)) &= D_{ab}(\alpha) = D_c(\alpha) \\
f(c, D_a(\alpha)) &= D_{ac}(\alpha) = D_c(\alpha) \\
f(a, D_b(\alpha)) &= D_{ba}(\alpha) = D_c(\alpha) \\
f(b, D_b(\alpha)) &= D_{bb}(\alpha) = D_c(\alpha) \\
f(c, D_b(\alpha)) &= D_{bc}(\alpha) = D_c(\alpha) \\
f(a, D_c(\alpha)) &= D_{ca}(\alpha) = D_c(\alpha) \\
f(b, D_c(\alpha)) &= D_{cb}(\alpha) = D_c(\alpha) \\
f(c, D_c(\alpha)) &= D_{cc}(\alpha) = D_c(\alpha)
\end{aligned}
$$

Con esto, resulta el diagrama de la figura 4.7, que es distinto del original de la figura 4.3. La explicación es la misma del ejemplo anterior: el de la figura 4.3 no está en forma mínima (q_2 y q_4 son equivalentes).

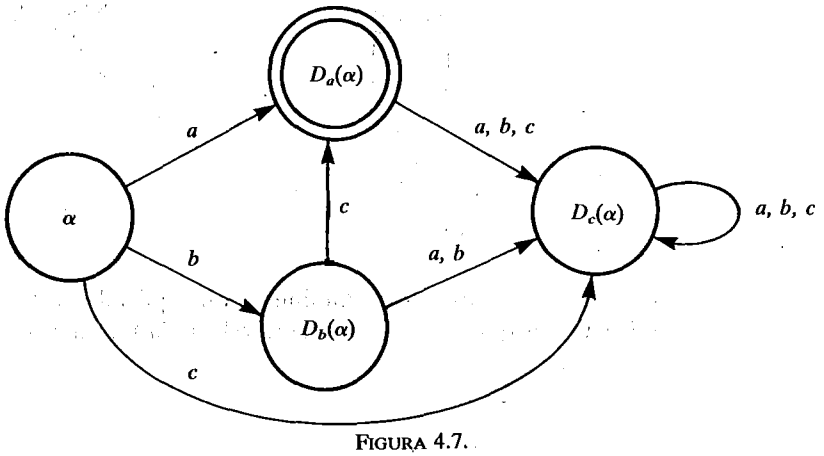


FIGURA 4.7.

Ejemplo 4.

$$\alpha = (1 + 2)^*3$$

Derivadas:

$$\begin{aligned}
D_1(\alpha) &= D_1[(1 + 2)^*]3 + \delta[(1 + 2)^*]D_1(3) = \\
&= [D_1(1 + 2)](1 + 2)^*3 + \lambda\emptyset = (1 + 2)^*3 = \alpha \\
D_2(\alpha) &= \ldots = (1 + 2)^*3 = \alpha \\
D_3(\alpha) &= D_3[(1 + 2)^*]3 + \delta[(1 + 2)^*]D_3(3) = \emptyset + \lambda\lambda = \lambda \\
D_{31}(\alpha) &= D_{32}(\alpha) = D_{33}(\alpha) = \emptyset
\end{aligned}$$

$$\begin{aligned}
Q &= \{\alpha, D_3(\alpha), D_{31}(\alpha)\} \\
F &= \{D_3(\alpha)\}
\end{aligned}$$

Calculando los valores de la función de transición se llega a un diagrama idéntico al de la figura 4.4, con $q_1 = \alpha$, $q_2 = D_3(\alpha)$, $q_3 = D_{31}(\alpha)$.

5. RESUMEN

Orientándonos ya hacia las relaciones entre lenguajes y autómatas, hemos definido un reconocedor finito como un tipo particular de autómata finito en el que existe un estado inicial fijo y el alfabeto de salida consta sólo de dos símbolos, correspondientes a la aceptación o no aceptación de la cadena de entrada.

Hemos demostrado la condición general que debe cumplir un lenguaje para que todas sus cadenas sean aceptadas por un reconocedor finito: que la relación de congruencia derecha inducida por el lenguaje tenga índice finito (ese índice es, precisamente, igual al número de estados del reconocedor).

Se han definido los conjuntos regulares, que, según los teoremas de análisis y síntesis, son justamente los lenguajes que pueden ser aceptados por un reconocedor finito. Las expresiones regulares constituyen un metalenguaje para describir de una manera cómoda a los conjuntos regulares. El algoritmo de análisis permite, dado un reconocedor finito, deducir la expresión regular del lenguaje que acepta ese reconocedor. A la inversa, hemos visto un algoritmo de síntesis mediante el cual se llega al diagrama de Moore de un reconocedor finito minimizado correspondiente a una expresión regular dada.

6. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

Los conceptos de conjunto regular y expresión regular, así como los teoremas de análisis y síntesis, se deben a Kleene (1956).

La demostración del teorema de análisis que hemos seguido es la de McNaughton y Yamada (1960).

Para el teorema de síntesis existen varias demostraciones. Quizá la más utilizada sea la debida a Rabin y Scott (1959), que precisa de la introducción de un tipo especial de autómata que no tiene interpretación física: el autómata no determinista (o «posibilístico», como prefiere llamarlo Arbib (1969), ya que no interviene la idea de probabilidad), que veremos en el tema «Lenguajes» (capítulo 4, apartado 5). Una generalización de este autómata es el llamado sistema de transición (Ott y Feinstein, 1961), que permite llegar a un algoritmo bastante cómodo para la síntesis. Sin embargo, nos ha parecido que el algoritmo de síntesis más manejable es el basado en el trabajo de Brozozowski (1962, 1965), que introdujo el concepto de derivada de una expresión regular.

Las anteriores referencias tienen un interés más bien histórico. Para ampliar este capítulo es preferible dirigirse a libros de carácter general, como los ya citados en el capítulo 2, o el de Hopcroft y Ullman (1979).

Los cuatro ejemplos utilizados reiteradamente están adaptados de Scala y Minguet (1974).

7. EJERCICIOS

7.1. Dada la tabla de transición

	a	b	c
q_1	q_2	q_6	q_1
q_2	q_3	q_5	q_1
q_3	q_3	q_4	q_1
q_4	q_7	q_7	q_1
q_5	q_6	q_5	q_1
q_6	q_7	q_6	q_1
(q_7)	q_7	q_7	q_1

correspondiente a un reconocedor finito, hallar la expresión regular correspondiente al lenguaje aceptado por ese reconocedor.

7.2. Aplicar el algoritmo de síntesis a la expresión regular encontrada en el ejercicio anterior.

7.3. Aplicar el algoritmo de análisis al reconocedor de la cadena 010 (capítulo 2, apartado 4.5.2) y al reconocedor de 321 (capítulo 2, ejercicio 10). En esos ejemplos se utilizó «reconocedor» con un sentido distinto del definido en este capítulo. ¿Cuáles son las diferencias?

7.4. Dar una expresión regular correspondiente al detector de paridad par y otra al detector de paridad impar, y aplicar el algoritmo de síntesis para obtener los correspondientes reconocedores.

7.5. Diseñar un circuito secuencial que dé una salida «1» cuando la cadena de entrada tenga un número de «unos» congruente a 0 (mód. 2). (La expresión regular es $\alpha = 0^*(10^*10^*)^*$).

7.6. Considérese un reconocedor finito definido por:

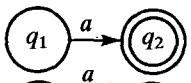
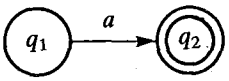
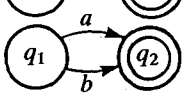
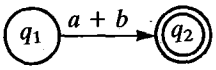
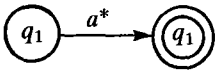
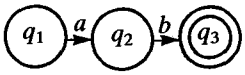
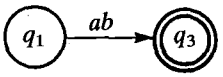
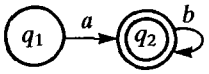
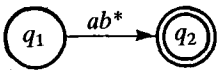
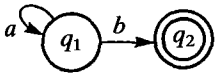
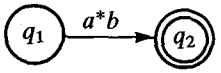
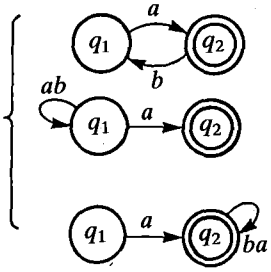
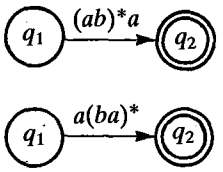
$$E = \{a, b, c, d\};\ Q = \{q_1, q_2, q_3, q_4\};\ F = \{q_4\}$$

q \ e	a	b	c	d
q_1	q_2	q_3	q_4	q_4
q_2	q_3	q_2	q_4	q_4
q_3	q_3	q_3	q_3	q_3
(q_4)	q_3	q_3	q_4	q_4

Hallar una expresión regular de las cadenas aceptadas por el reconocedor, y diseñar un circuito secuencial para la realización del reconocedor con biestables JK.

Los dos ejercicios siguientes, tomados de Booth (1967), desarrollan una técnica gráfica para obtener la expresión regular directamente del diagrama de Moore.

7.7. Las siguientes expresiones regulares básicas describen a los conjuntos regulares de los reconocedores correspondientes. (El estado inicial es siempre q_1):

	Expresión regular	*Diagrama*	*Diagrama reducido*
1)	a		
2)	$a + b$		
3)	a^*		
4)	ab		
5)	ab^*		
6)	a^*b		
7)	$(ab)^*a = a(ba)^*$		

a) Comprobar que las expresiones regulares corresponden efectivamente a los diagramas.
b) Utilizar esas relaciones para obtener la expresión regular del reconocedor dado por la tabla:

	a	b
q_1	q_1	q_2
q_2	q_4	q_5
q_3	q_5	q_1
q_4	q_3	q_2
(q_5)	q_4	q_5

7.8. Utilizando como punto de partida las relaciones básicas establecidas en el ejercicio anterior, desarrollar una técnica general gráfica de análisis para deducir la expresión regular de cualquier reconocedor finito.

Capítulo 5
OTROS AUTOMATAS

1. INTRODUCCIÓN

El autómata finito estudiado en los anteriores capítulos es un modelo aplicable a cualquier sistema dinámico, discreto y con memoria (siempre que la memoria sea finita). Ahora bien, para determinados sistemas, la descripción que se obtendría con tal modelo sería inútil, por demasiado compleja (número de estados y de transiciones desorbitado) o porque los estados engloban de forma poco natural un conjunto de condiciones internas del sistema. Tal ocurre, por ejemplo, cuando el sistema está compuesto por subsistemas con funcionamiento asíncrono y concurrente. Para modelar estos casos se han propuesto diversas generalizaciones del modelo. Una de las más conocidas y más aplicadas en informática es la llamada «red de Petri». La red de Petri puede, además, como veremos, modelar sistemas con infinitos estados, por lo que se aproxima, en cuanto a potencia de descripción, a la máquina de Turing.

Por otra parte, los modelos que hemos considerado hasta ahora son completamente deterministas. Es decir, dado un estado y un símbolo de entrada, el estado siguiente y el símbolo de salida vienen determinados por las funciones f y g, respectivamente. Pero se dan situaciones cuya complejidad funcional y la imposibilidad de analizar todos los factores que intervienen hacen que sea preferible modelarlas a partir de la idea de probabilidad. Tal ocurre, por ejemplo, cuando la fiabilidad de los componentes es pequeña, o cuando se quiere modelar órganos o procesos de los sistemas vivos. Así, si observamos que, estando en el estado q_1 y recibiendo entrada e_1 el sistema pasa al estado q_2 un 80% de las veces y al q_3 un 20%, diremos que $f(e_1, q_1) = q_2$ con probabilidad 0,8 y $f(e_1, q_1) = q_3$ con probabilidad 0,2. Esta es la idea básica del autómata probabilista o estocástico.

Si en lugar de «probabilidad» utilizamos «borrosidad» tendremos un autómata borroso. Uno de los campos de aplicación más interesantes de los autómatas estocásticos y de los borrosos está en los sistemas de aprendizaje: el autómata, en interacción con su entorno, puede ir modificando su estructura para adaptarse a tal entorno.

2. REDES DE PETRI

2.1. Estructura estática

Definición 2.1.1. Una *red de Petri* (RdP) es una cuádrupla

$$\langle L, T, \alpha, \beta \rangle$$

donde:

L: es un conjunto finito, no vacío, de *lugares*.
T: es un conjunto finito, no vacío, y disjunto de L, de *transiciones*.
α: $T \to \mathcal{P}(L)$ es la *función de incidencia anterior*, o *función de entrada*, que define, para cada transición, sus lugares de entrada.
β: $T \to \mathcal{P}(L)$ es la *función de incidencia posterior*, o función de salida, que define, para cada transición, sus lugares de salida.

Por ejemplo:

$$L = \{l_1, l_2, l_3, l_4, l_5\}$$
$$T = \{t_1, t_2, t_3, t_4\}$$
$$\alpha(t_1) = \{l_1, l_5\};\ \beta(t_1) = \{l_2\}$$
$$\alpha(t_2) = \{l_2\};\ \beta(t_2) = \{l_1, l_5\}$$
$$\alpha(t_3) = \{l_3, l_5\};\ \beta(t_3) = \{l_4\}$$
$$\alpha(t_4) = \{l_4\};\ \beta(t_4) = \{l_3, l_5\}$$

Una RdP puede representarse gráficamente como un grafo bipartido orientado, en el que hay dos tipos de nodos: círculos (para los lugares) y segmentos (para las transiciones), y los arcos representan a las funciones α y β. Así, el grafo del ejemplo anterior sería el de la figura 5.1.

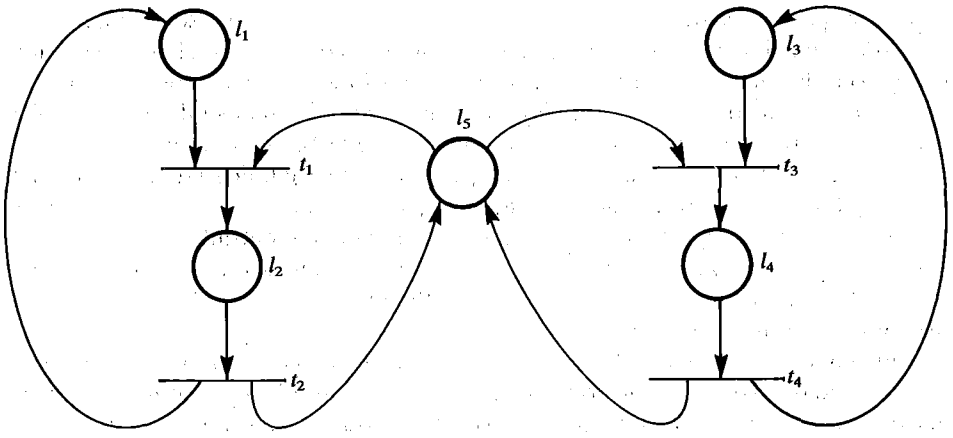


FIGURA 5.1.

Consideraremos solamente las llamadas «RdP puras», en las que no puede haber lugares que sean entrada y salida de una misma transición. Es decir,

$$(\forall t_i)(\alpha(t_1) \cap \beta(t_i) = \emptyset)$$

En este caso, la RdP puede representarse también mediante una *matriz de incidencia* cuyas filas corresponden a los lugares y cuyas columnas corresponden a las transiciones y cuyo elemento (i, j) es:

$$\begin{array}{r} 0 \text{ si } l_i \text{ no es entrada ni salida de } t_j \\ 1 \text{ si } l_i \text{ es salida de } t_j \\ -1 \text{ si } l_i \text{ es entrada de } t_j \end{array}$$

La matriz de incidencia del ejemplo anterior sería:

$$\begin{array}{c} \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & t_3 & t_4 \end{array} \\ \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix} \end{array}$$

El concepto de RdP tal como lo acabamos de definir está incompleto, porque no contempla ningún comportamiento dinámico. En particular, se observará que no se ha definido el estado.

2.2. Comportamiento dinámico

Definición 2.2.1. Un *marcado* de una RdP es una asignación de *marcas* (números naturales) a los lugares:

$$\mu: L \rightarrow N$$

El número de marcas en l_i será $\mu(l_i)$. El número y posición de las marcas varía con el tiempo según las reglas que definiremos enseguida. En cada instante, tendremos un vector $\mu = [\mu(l_1), \mu(l_2) \ldots]^T$ que indica el número de marcas en cada lugar.

Definición 2.2.2. Una RdP marcada es una quíntupla

$$\langle L, T, \alpha, \beta, \mu_0 \rangle,$$

donde los cuatro primeros elementos son los de la Definición 2.1.1, y μ_0 es un *marcado inicial.*

En adelante consideraremos siempre RdP marcadas.

Gráficamente, el marcado se indica mediante puntos en los círculos que representan a los lugares. Así, si $\mu_0 = [1\ 0\ 1\ 0\ 1]^T$, el grafo de la RdP marcada correspondiente al ejemplo anterior será el de la figura 5.2.

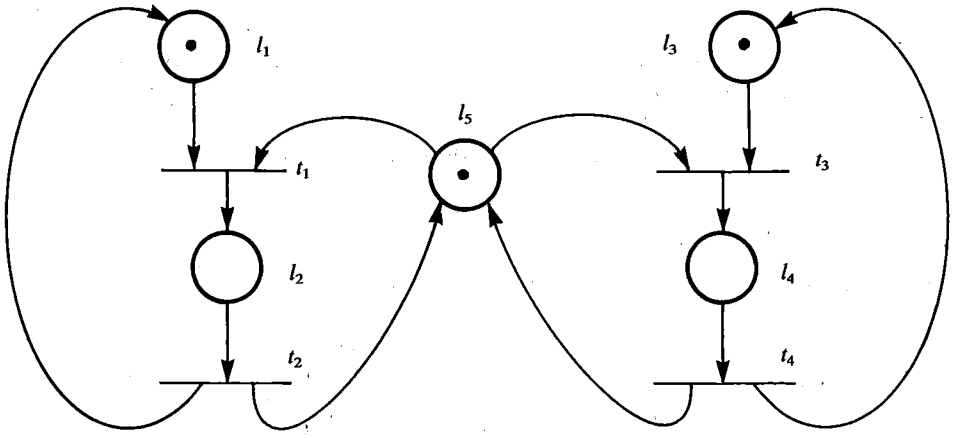


FIGURA 5.2.

Definición 2.2.3. Una transición t_i está *permitida* por el marcado si todos sus lugares de entrada tienen al menos una marca, es decir, si

$$(\forall l_j \in \alpha(t_i))(\mu(l_j) > 0)$$

En la figura 5.2, t_1 y t_3 están permitidas; t_2 y t_4, no.

Definición 2.2.4. El *disparo* de una transición permitida consiste en quitar una marca de cada uno de sus lugares de entrada y añadir una marca a cada uno de sus lugares de salida.

Por ejemplo, el disparo de t_1 en la figura 5.2 conduce a un nuevo marcado: desaparecen las marcas de l_1 y l_5 y aparece una en t_2; el nuevo marcado es $\mu_1 = [0\ 1\ 1\ 0\ 0]^T$. Obsérvese que el hecho de disparar t_1 hace que t_3 deje de estar permitida.

Definición 2.2.5. El *estado* de una RdP en un instante viene dado por su marcado μ_i en ese instante. El conjunto de estados, M, será, pues, el conjunto de todos los marcados posibles.

Definición 2.2.6. La *función de transición*, f, de una RdP es una función

$$f: M \times T \rightarrow M$$

tal que aplicada a un marcado μ_i y a una transición t_j da el marcado que resulta de disparar t_j. Es una función parcial, porque si t_j no está permitida por μ_i, entonces $f(\mu_i, t_j)$ no está definida.

Definición 2.2.7. Se llama *ejecución* de una RdP a la secuencia de acontecimientos que resulta de disparar transiciones sucesivamente a partir de un marcado inicial. Esta secuencia de acontecimientos se manifiesta en dos secuencias de objetos: una secuencia de marcados $(\mu_0, \mu_1, \ldots)$, y una secuencia de transiciones $(t_0, t_1, \ldots)$, tales que $f(\mu_i, t_i) = \mu_{i+1}$. En el ejemplo de la figura 5.2 podemos tener esta ejecución:

$$\begin{aligned}
&\mu_0 = [1\ 0\ 1\ 0\ 1]^T;\\
&\mu_1 = [0\ 1\ 1\ 0\ 0]^T;\\
&\mu_2 = \mu_0;\\
&\mu_3 = [1\ 0\ 0\ 1\ 0]^T;\\
&\mu_4 = \mu_0;\\
&\mu_5 = \mu_1;\\
&\ldots\ldots\ldots\ldots\ldots\\
&(t_1, t_2, t_3, t_4, t_1, \ldots)
\end{aligned}$$

obsérvese que la ejecución no es única: con el marcado μ_0 pueden dispararse indistintamente t_1 ó t_3; cuál de ellas lo hace depende de acontecimientos externos que no figuran en el modelo. Las RdP son, pues, modelos no deterministas, como los autómatas que veremos en el tema «Lenguajes».

Teorema 2.2.8. La función de transición de una RdP puede escribirse explícitamente en función de la matriz de incidencia, I, y de un vector, u_i, cuyas componentes (tantas como transiciones tenga la RdP) son todas nulas, salvo la que corresponde a la transición disparada en el instante i, que vale 1, y ello mediante la llamada *ecuación de estado de la RdP*:

$$\mu_{i+1} = \mu_i + I \cdot u_i$$

Para la demostración, que omitimos, basta con ver que todas las componentes de μ_i satisfacen la ecuación, teniendo en cuenta la definición de I (apartado 2.1).

En nuestro ejemplo, si, partiendo de μ_0, se dispara t_1, tendremos:

$$\mu_1 = \mu_0 + I \cdot u_0 = \begin{bmatrix}1\\0\\1\\0\\1\end{bmatrix} + \begin{bmatrix}-1 & 1 & 0 & 0\\ 1 & -1 & 0 & 0\\ 0 & 0 & -1 & 1\\ 0 & 0 & 1 & -1\\ -1 & 1 & -1 & 1\end{bmatrix} \cdot \begin{bmatrix}1\\0\\0\\0\end{bmatrix} =$$

$$= \begin{bmatrix}1\\0\\1\\0\\1\end{bmatrix} + \begin{bmatrix}-1\\1\\0\\0\\-1\end{bmatrix} = \begin{bmatrix}0\\1\\1\\0\\0\end{bmatrix}$$

Y si luego se dispara t_2:

$$\mu_2 = \mu_1 + I \cdot u_1 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} =$$

$$= \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

El marcado final resultante de una secuencia de disparos a partir de un marcado inicial, μ_0, puede obtenerse aplicando sucesivamente la ecuación de estado:

$$\begin{aligned} \mu_i &= \mu_{i-1} + I \cdot u_i = \mu_{i-2} + I \cdot (u_{i-1} + u_i) = \\ &= \mu_{i-3} + I \cdot (u_{i-2} + u_{i-1} + u_i) = \ldots \\ &= \mu_0 + I \cdot \sum_{j=1}^{i} u_i = \mu_0 + I \cdot \overline{s} \end{aligned}$$

donde $\overline{s}$ es el *vector característico* asociado a la secuencia $s = (t_1, t_2, \ldots)$: su j-ésima componente es igual al número de ocurrencias de t_j en S. En nuestro ejemplo, si $s = (t_1, t_2, t_3, t_4, t_1)$, $\overline{s} = [2\ 1\ 1\ 1]^T$,

$$\mu_s = \mu_0 + I \cdot \overline{s} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 1 \\ 1 \\ 1 \end{bmatrix} =$$

$$= \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \\ 0 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

2.3. Autómata finito equivalente a una red de Petri con espacio de estados finito

Tal como se ha definido el estado de una RdP, el número de estados posible puede ser infinito, porque el número de marcas en cada lugar no está, en general, limitado. Por ejemplo, en la RdP de la figura 5.3, la secuencia t_1, t_2, t_1, t_2 ... hace crecer indefinidamente el número de marcas en l_3.

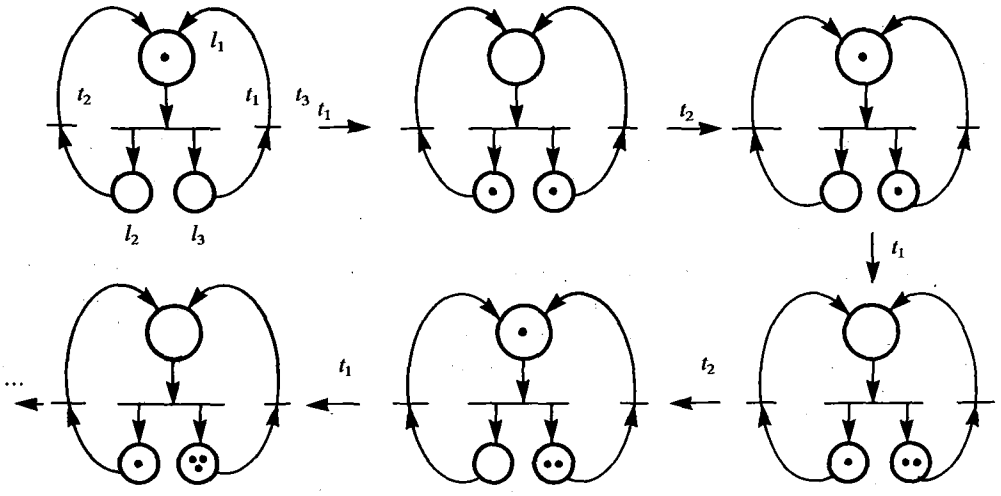


FIGURA 5.3.

Pero en ciertas RdP el número de marcados posibles a partir de uno inicial es finito. Así ocurre en el ejemplo de la figura 5.2. Esa RdP es, además, *binaria*: a partir de un marcado que no asigne a cada lugar más de una marca, todos los marcados sucesivos son tales que cualquier lugar tiene una marca o ninguna. En este caso, como hay cinco lugares, el número de marcados binarios posibles es $2^5 = 32$. Sin embargo, no todos ellos son alcanzables a partir de uno inicial dado. Por ejemplo, si μ_0 es el de la figura 5.2, es decir,

$$\mu_0 = [1\ 0\ 1\ 0\ 1]^T,$$

puede comprobarse que sólo son alcanzables otros dos marcados:

$$\mu_1 = [0\ 1\ 1\ 0\ 0]^T$$
$$\mu_2 = [1\ 0\ 0\ 1\ 0]^T$$

El AF equivalente a esta RdP sería el de la figura 5.4.

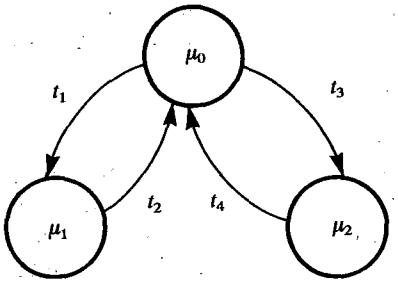


FIGURA 5.4.

2.4. Ejemplos de aplicación

2.4.1. *Redes de Petri interpretadas*

Para aplicar el concepto matemático de RdP a la modelación de sistemas hay que establecer una *interpretación* o convenio por el que se asocian las entradas del sistema a condiciones necesarias para que ciertas transiciones se disparen y las salidas al disparo de otras transiciones o a las marcas de determinados lugares.

Como ya hemos dicho en la introducción, las RdP son una herramienta de modelación útil para sistemas en los que se dan actividades asíncronas y concurrentes. Esto abarca campos de aplicación muy diversos. Aquí nos limitaremos a ilustrar con dos ejemplos su utilidad para abordar ciertos problemas que aparecen en el diseño de programas.

2.4.2. *Problemas de exclusión mutua: lectores y redactores*

Un problema que se presenta con frecuencia en los sistemas multiprogramados es el de evitar que dos procesos (programas en ejecución) puedan acceder simultáneamente a un recurso común no compartible: un periférico, un fichero, una tabla de datos en memoria, etc. Los segmentos de los programas que realizan ese acceso se llaman *secciones críticas*, y hay que dotar al sistema de mecanismos que aseguren la exclusión mutua de los procesos en la ejecución de tales secciones críticas.

Uno de los mecanismos más utilizados es el del *semáforo* acompañado de las operaciones «*P*» (o «wait») y «*V*» (o «signal»). Un semáforo es una variable entera no negativa, *S*, que, aparte de su inicialización, sólo puede modificarse mediante las operaciones «*P*» y «*V*»:

$P(S)$: si $S > 0$, disminuye S en una unidad
$V(S)$: incrementa S en una unidad

Si un proceso ejecuta «*P*» sobre un semáforo con $S = 0$, entonces debe esperar hasta que otro proceso ejecute «*V*» sobre el mismo semáforo. Por otra parte, son operaciones indivisibles: mientras un proceso no termina de ejecutar «*P*» o «*V*» otro no puede acceder al mismo semáforo.

Para asegurar la exclusión mutua de dos procesos a secciones críticas podemos utilizar un semáforo, «exmut», inicializado con el valor 1, del siguiente modo:

Programa 1	*Programa 2*
Sección A1	Sección A2
P(exmut)	*P*(exmut)
Sección crítica 1	Sección crítica 2
V(exmut)	*V*(exmut)
Sección B1	Sección B2

El primero de los procesos que ejecute P pondrá el semáforo a cero, y si el otro intenta entrar en su sección crítica antes de que el primero haya salido de la suya y ejecutado V, tendrá que esperar.

La modelación con una RdP puede hacerse considerando el semáforo como un lugar en el que el número de marcas es el valor del semáforo. Las operaciones P son transiciones de salida del lugar, y las V lo son de entrada. La RdP que modela el funcionamiento de los dos procesos anteriores es la de la figura 5.2, interpretada del siguiente modo:

l_5 es el semáforo «exmut».
t_1 es la «P» del Proceso 1; hay que añadir la condición de disparo: fin de la ejecución de la «Sección A1».
t_2 es la «V» del Proceso 1; la condición de disparo es el fin de la ejecución de la «Sección crítica 1».
l_1 es «entrada en la Sección B1».
l_2 es «entrada en la Sección crítica 1».
t_3, t_4, l_3 y l_4 son las análogas del Proceso 2.

Obsérvese que los lugares l_2 y l_4 quedan mutuamente excluidos.

Como un ejemplo más concreto del problema de la exclusión mutua, consideremos uno clásico: el de los lectores y redactores, procesos concurrentes que acceden a unos datos comunes con las siguientes restricciones:

a) Los lectores pueden acceder a los datos simultáneamente.
b) Por el contrario, los redactores, como modifican los datos, deben trabajar en exclusión mutua entre sí y excluyendo también a los lectores. Es decir, mientras un redactor está ejecutando la sección crítica (acceso a los datos) ningún otro proceso puede entrar en su sección crítica.
c) Los procesos pueden estar en alguno de estos tres estados:

   A (activo: ejecutando su sección crítica);
   E (esperando entrar en su sección crítica);
   R (reposo: no necesita los datos, aunque puede estar ejecutando otra parte del programa).

Supongamos que hay dos lectores y dos redactores. Cada uno de los cuatro procesos puede modelarse como una RdP, según la figura 5.5. Los tres lugares corresponden a los estados definidos, y las transiciones se disparan cuando en el proceso aparecen determinados acontecimientos: D (demanda de acceso a los datos), C (comienzo de ejecución de la sección crítica, o acceso a los datos), F (fin de acceso a los datos).

Ahora bien, para poder cumplir las restricciones enunciadas será preciso interconectar las cuatro subredes: el acceso de un lector a los datos tienen que excluir acceso de los dos redactores, pero no del otro lector, y el acceso de un redactor tiene que excluir el acceso de los otros tres. Podemos utilizar dos semáforos, S_1 y S_2. S_1 para excluir al lector 1 de los dos redactores y S_2 para el lector 2. La RdP global será la de la

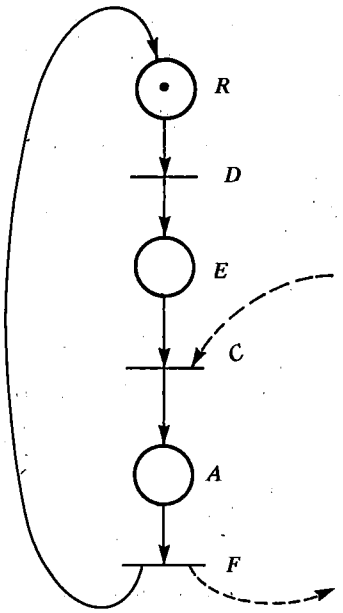


FIGURA 5.5.

figura 5.6, en la que *l*1, *l*2, *r*1, *r*2 significan «lector 1», «lector 2», «redactor 1» y «redactor 2».

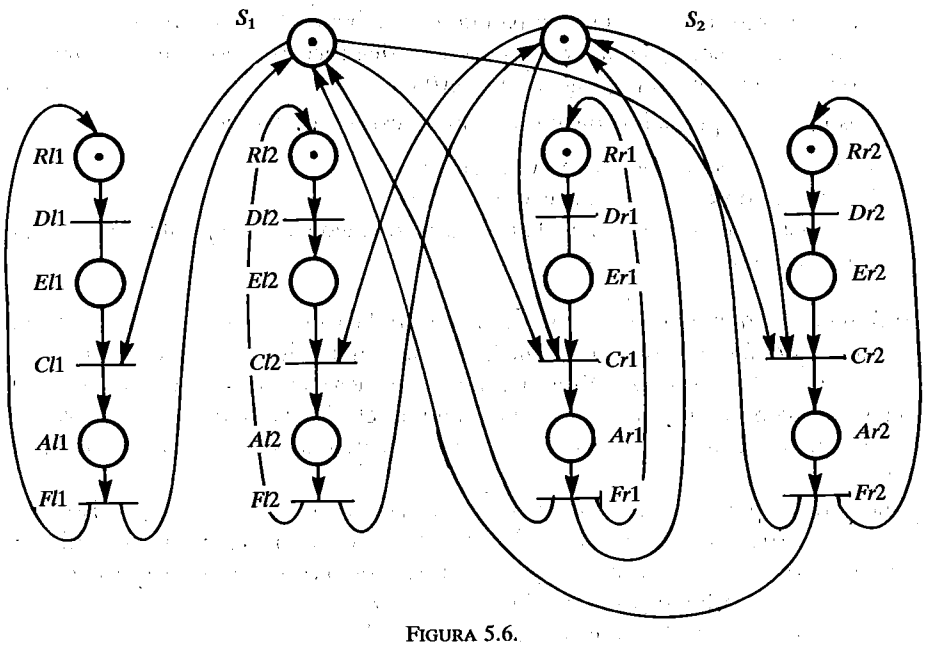


FIGURA 5.6.

Obsérvese que la RdP en este ejemplo es binaria, y, por ello, el número de estados es finito. Por tanto, podríamos hallar un AF equivalente, como hacíamos en el apartado 2.3. Los estados de tal AF corresponderían a combinaciones de estados de cada uno de los procesos, haciendo más difícil la interpretación del modelo. Además, la RdP es modular: si se ha comprendido la idea de la figura 5.6, es muy fácil extenderla a casos con más lectores y/o redactores. Finalmente, el paso a la programación es inmediato: basta con poner, en cada proceso, operaciones P y V sobre S_1 ó S_2 en las transiciones C y F.

2.4.3. *Problemas de sincronización: productores y consumidores*

En muchas aplicaciones, dos o más procesos pueden ejecutarse concurrentemente pero no de manera independiente: uno de los procesos no puede seguir a partir de un punto si el otro no ha realizado una cierta acción. Para concretar, consideremos otro problema clásico en informática: un proceso productor genera elementos que deposita en un *almacén* o zona de memoria («buffer»), mientras que un proceso consumidor extrae elementos del mismo almacén. El almacén tiene una capacidad limitada, N. El productor repite continuamente un ciclo: «producir un elemento-meterlo en el almacén-producir un elemento...»; el consumidor, otro: «extraer un elemento-consumirlo-extraer...». En la figura 5.7, la parte de la izquierda corresponde al ciclo del productor, con

E_p = productor esperando (a que haya sitio en el almacén o a que el consumidor finalice su acceso);
C_m = comienzo de la operación de meter un elemento;
M = meter un elemento;
F_m = fin de la operación de meter (y comienzo de producir);
P = producir un elemento;
F_p = fin de la producción,

y la de la derecha, al ciclo del consumidor, con

E_c = consumidor esperando (a que haya elementos, o a que el productor finalice su acceso);
C_s = comienzo de la operación de sacar;
S = sacar;
F_s = fin de sacar (y comienzo de consumir);
C = consumir un elemento;
F_c = fin de consumir.

Como indican las líneas de puntos, la RdP está incompleta. Es preciso añadir los mecanismos de sincronización entre ambos procesos:

a) C_m no puede dispararse si no hay sitio disponible;
b) C_s no puede dispararse si no hay elementos disponibles;
c) los procesos no pueden acceder simultáneamente al almacén (ello podría

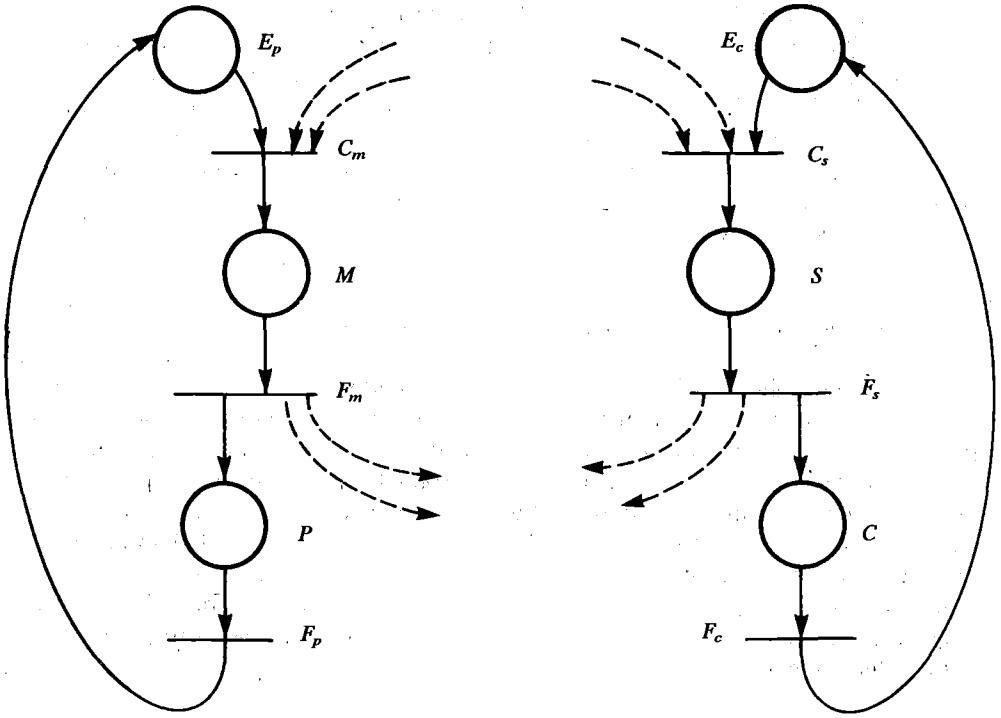


FIGURA 5.7.

provocar, en la realización del programa, problemas de actualización de punteros).

Para ello, añadiremos tres semáforos, modelados en la RdP como tres lugares:

a) S_s, que se inicializará con el valor N (capacidad del almacén), y cuyo valor (número de marcas) indicará el número de sitios disponibles. F_s lo incrementará en una unidad, y C_m lo decrementará.
b) S_e, que se inicializará con el valor 0, y cuyo valor indicará el número de elementos en el almacén. C_s lo decrementará, y F_m lo incrementará.
c) S_m, semáforo binario con el valor inicial 1, que servirá para la exclusión mutua de ambos procesos en el acceso al almacén.

La RdP completa es la de la figura 5.8, en la que se ha puesto un marcado inicial que supone que la capacidad del almacén es de 4 elementos.

Obsérvese también aquí que el número de estados es finito, pero que el AF equivalente, que podemos encontrar, modela de manera menos natural los fenómenos. Asimismo, que es fácil extender la RdP para que considere más productores y/o consumidores.

Un ejemplo típico de procesos que han de coordinarse de este modo se tiene en los sistemas operativos con los gestores («drivers») de dispositivos de entrada y salida: un

gestor de salida es un consumidor que, cuando tiene en el almacén alguna información para enviar al periférico de salida, se ejecuta; el productor es el programa que produce esas informaciones. Simétricamente, un gestor de entrada es un productor para los programas que consumen esas informaciones.

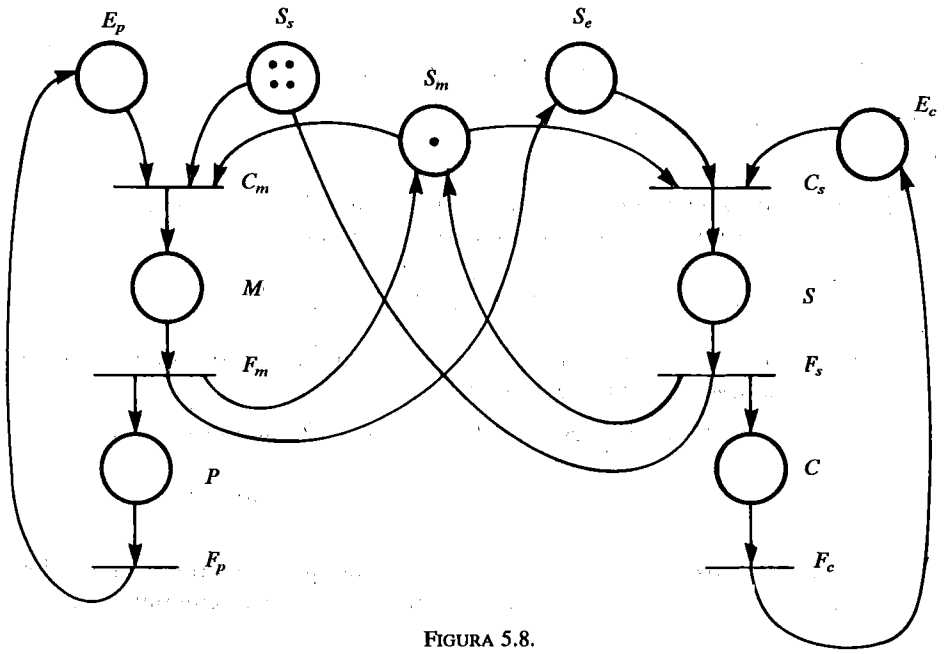


FIGURA 5.8.

El esquema para la programación de un productor y un consumidor sería:

Productor	*Consumidor*
Producir-elemento;	$P(S_e)$;
$P(S_s)$;	$P(S_m)$;
$P(S_m)$;	Sacar-elemento;
Meter-elemento;	$V(S_m)$;
$V(S_m)$;	$V(S_s)$;
$V(S_e)$;	Consumir-elemento.

Obsérvese que S_m es el equivalente al semáforo «exmut» del ejemplo anterior, mientras que S_e y S_s aseguran la coordinación de los dos procesos.

3. AUTÓMATAS ESTOCÁSTICOS

3.1. Definición

Un autómata estocástico o probabilista es una quíntupla

$$AP = \langle E, S, Q, P, h \rangle,$$

donde:

E, S, Q: son, como en los autómatas deterministas, los alfabetos de entrada y salida y el conjunto de estados, que supondremos finito:

$$Q = \{q_1, q_2, \ldots, q_n\}$$

h : $Q \to S$ es la función de salida (consideraremos sólo autómatas de tipo Moore), que se supone determinista.

P : $E \times Q \to [0, 1]^n$ es la *función de probabilidades de transición*:

$$P(e, q) = (p_1(e, q), p_2(e, q), \ldots, p_n(e, q)),$$

donde $0 \leq p_i(e, q) \leq 1$ es la probabilidad de que, estando en el estado q y recibiendo la entrada e, se pase al estado q_i. (Deberá cumplirse que:

$$\sum_{i=1}^{n} p_i(e, q) = 1).$$

Si consideramos todos los estados en que puede encontrarse el autómata cuando recibe una entrada e, podemos definir una *matriz de probabilidades de transición*, $M(e)$:

$$M(e) = \begin{bmatrix} p_1(e, q_1) & \ldots & p_n(e, q_1) \\ \vdots & & \vdots \\ p_1(e, q_n) & \ldots & p_n(e, q_n) \end{bmatrix}$$

de modo que el elemento $m_{ij} = p_j(e, q_i)$ de $M(e)$ es la probabilidad de pasar del estado q_i al q_j bajo la acción de la entrada e.

Si $x = e_1, e_2, \ldots, e_m \in E^*$, es fácil demostrar, por inducción sobre m, que $M(x) = (p_j(x, q_i)) = M(e_1) \cdot M(e_2), \ldots, M(e_m)$.

Un AF determinista es un caso particular del estocástico. En el caso estocástico, $m_{ij} \in [0, 1]$, y en el determinista $m_{ij} \in \{0, 1\}$. En cualquier caso, la suma de los elementos de cada fila en todas las matrices M debe ser la unidad.

3.2. Ejemplo

Sea un autómata estocástico definido por:

$$E = S = \{0, 1\}$$
$$Q = \{q_1, q_2\}$$
$$h(q_1) = 0;\ h(q_2) = 1$$
$$P(0, q_1) = (0{,}7;\ 0{,}3);\ P(1, q_1) = (0{,}4;\ 0{,}6)$$
$$P(0, q_2) = (0{,}6;\ 0{,}4);\ P(1, q_2) = (0{,}2;\ 0{,}8)$$

Podemos representar este autómata por un diagrama de Moore (figura 5.9), indicando sobre cada transición la probabilidad asociada.

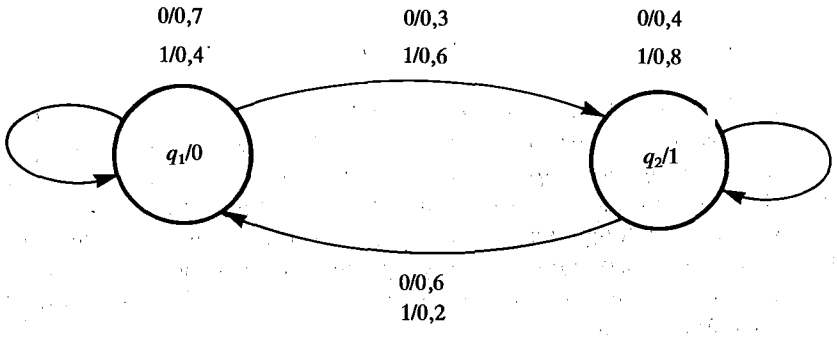


FIGURA 5.9.

Las matrices de probabilidades de transición correspondientes a los dos símbolos de entrada son:

$$M(0) = \begin{bmatrix} 0{,}7 & 0{,}3 \\ 0{,}6 & 0{,}4 \end{bmatrix}; \quad M(1) = \begin{bmatrix} 0{,}4 & 0{,}6 \\ 0{,}2 & 0{,}8 \end{bmatrix}$$

Si por ejemplo estamos interesados en la respuesta a la cadena $x = 011$, tendremos:

$$M(011) = \begin{bmatrix} 0{,}7 & 0{,}3 \\ 0{,}6 & 0{,}4 \end{bmatrix} \cdot \begin{bmatrix} 0{,}4 & 0{,}6 \\ 0{,}2 & 0{,}8 \end{bmatrix}^2 = \begin{bmatrix} 0{,}268 & 0{,}732 \\ 0{,}264 & 0{,}736 \end{bmatrix}$$

Así, la probabilidad de pasar de q_1 a q_2 bajo la acción de $x = 011$ es 0,732. Al mismo resultado se llega si se consideran las cuatro posibilidades existentes para pasar de q_1 a q_2 con la cadena 011:

$$\xrightarrow{0} q_1 \xrightarrow{1} q_1 \xrightarrow{1} q_2\text{: probabilidad} = 0{,}7 \times 0{,}4 \times 0{,}6 = 0{,}168$$
$$\xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{1} q_2\text{: probabilidad} = 0{,}7 \times 0{,}6 \times 0{,}8 = 0{,}336$$
$$\xrightarrow{0} q_2 \xrightarrow{1} q_2 \xrightarrow{1} q_2\text{: probabilidad} = 0{,}3 \times 0{,}8 \times 0{,}8 = 0{,}192$$
$$\xrightarrow{0} q_2 \xrightarrow{1} q_1 \xrightarrow{1} q_2\text{: probabilidad} = 0{,}3 \times 0{,}2 \times 0{,}6 = 0{,}036$$
$$\text{probabilidad total} = 0{,}732$$

3.3. Reconocedores estocásticos

Siguiendo la misma línea del capítulo 4, podemos definir un reconocedor estocástico como

$$R_p = \langle E, Q, q_1, P, F \rangle,$$

donde q_1 es el estado designado como inicial y $F \subset Q$ es el conjunto de estados finales. (Algunos autores introducen también la indeterminación de tomar un estado inicial u otro, y, en lugar de q_1, incluyen un conjunto, $\Psi = \{\psi_1, \psi_2, \ldots, \psi_n\}$, que representa las probabilidades asociadas a cada estado inicialmente).

El estudio de los lenguajes aceptados por reconocedores estocásticos es más complicado que en el caso determinista, ya que una misma cadena puede ser aceptada por el reconocedor en unas ocasiones y rechazada en otras. La probabilidad de que $x \in E^*$ sea aceptada, será:

$$p(x) = \sum_i p_i(q_1, x)$$

Así, en el ejemplo anterior, si q_1 es el estado inicial y q_2 el final, $p(011) = 0{,}732$.

Se define el lenguaje aceptado por un reconocedor estocástico haciéndolo depender de un parámetro, λ, llamado *punto de corte* del lenguaje:

$$L_{RP}(\lambda) = \{x \in E^* | p(x) > \lambda\}$$

y se demuestra que tales lenguajes son una generalización de los conjuntos regulares.

4. AUTÓMATAS ESTOCÁSTICOS DE ESTRUCTURA VARIABLE

Un autómata estocástico o probabilista de estructura variable, APEV, es una sextupla

$$\text{APEV} = \langle E, S, Q, P, h, A \rangle,$$

donde E, S, Q, P, h tienen el mismo significado anterior, y A es un algoritmo llamado *esquema de actualización o esquema de refuerzo* que genera P_{t+1} a partir de P_t, s_t y e_t. Así en un APEV las probabilidades de transición no son siempre las mismas, sino que van evolucionando en función de su valor anterior, de la respuesta última y de la entrada. *Esta característica es la que confiere al APEV la capacidad de adaptarse y de aprender.*

5. AUTÓMATAS DE APRENDIZAJE

5.1. Concepto de aprendizaje

Las palabras «adaptación» y «aprendizaje» tienen diferente significado según quién las utiliza. En psicología representan conceptos bastante diferentes, mientras que en ingeniería suelen tomarse como sinónimos.

Aquí nos limitaremos a definir el aprendizaje como la *capacidad de un sistema para mejorar la probabilidad de emitir una respuesta correcta como resultado de la interacción con su entorno*. Esto implica que el entorno tiene capacidad de evaluación de las respuestas del sistema. Un esquema general de un proceso de aprendizaje es el de la figura 5.10.

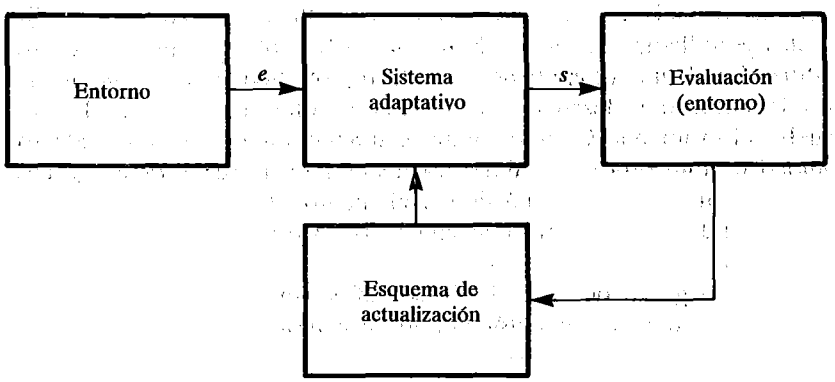


FIGURA 5.10.

5.2. Aprendizaje en un APEV

De acuerdo con lo dicho, un APEV en interacción con un entorno (figura 5.11) puede presentar un comportamiento adaptativo que le confiera capacidad de aprendizaje. *Así, un autómata de aprendizaje será un APEV que opera en un entorno y actualiza sus probabilidades de transición de acuerdo con las entradas recibidas del entorno para mejorar su comportamiento en algún sentido especificado.*

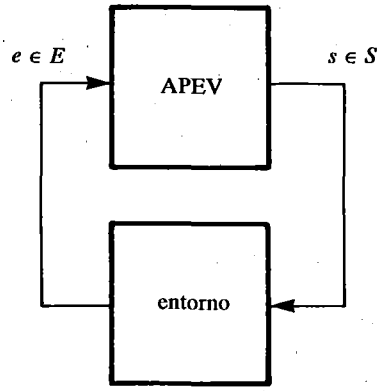


FIGURA 5.11.

En psicología, un autómata de aprendizaje puede ser un modelo del comportamiento de un organismo bajo estudio, donde el APEV será el modelo del organismo y el entorno estará representado por el experimentador. En una aplicación de ingeniería, tal como el control de un proceso, el controlador es el APEV, y el resto del sistema, con sus incertidumbres, constituye el entorno.

Las respuestas del entorno son frecuentemente binarias, es decir, $E = \{0, 1\}$, y a una de ellas se le llama «respuesta de premio» y a la otra «respuesta de castigo».

La utilización de un autómata de aprendizaje sólo tiene interés cuando el comportamiento del entorno es desconocido, puesto que si no fuera así, un AF determinista nos resuelve el problema. Generalmente se supone que el entorno es aleatorio, y la probabilidad de que produzca salida «1» dependerá de su entrada, es decir, de la respuesta s del autómata. Así, si S tiene r elementos, tendremos r probabilidades C_i ($i = 1, 2, \ldots, r$) de que el entorno de salida «1». A estas probabilidades se les llama *probabilidades de castigo*.

Para juzgar el grado de aprendizaje se define un *castigo medio recibido por el autómata*: en un instante t, si el autómata produce la salida s_i con probabilidad $p_i(t)$, el castigo medio condicionado a $P(t)$ es:

$$M(t) = E[e(t)|P(t)] = \sum_{i=1}^{r} p_i(t)C_i$$

Si suponemos que en $t = 0$ el autómata escogerá una salida u otra con igual probabilidad, se tendrá

$$M(0) = \frac{\Sigma C_i}{r}$$

El uso del término «aprendizaje» sólo tiene sentido si $M(t)$ se va haciendo menor que $M(0)$. Según como sea esa tendencia se definen unos tipos u otros de autómatas, y se estudian los esquemas de actualización necesarios para conseguirlos.

6. AUTÓMATAS BORROSOS

Podemos definir un autómata finito borroso como

$$\text{AB} = \langle E, S, Q, f, h \rangle,$$

donde E, S, Q y h: $Q \to S$ tienen el mismo significado que en el AF ordinario, y f es una *función de transición borrosa*:

$$f\colon E \times Q \times Q \to M$$

(en adelante tomaremos $M = \{x | 0 \leq x \leq 1\}$). Esta f se interpreta así:

si $f(e, q_i, q_j) = 0$, seguro que, con entrada e, *no hay* transición de q_i a q_j;
si $f(e, q_i, q_j) = 1$, seguro que, con entrada e, *hay* transición de q_i a q_j;
si $f(e, q_i, q_j) = \alpha$, α es el grado de confianza de que, para entrada e, haya transición de q_i a q_j.

(Obsérvese que también podríamos haber definido, de forma totalmente equivalente,

$$F\colon E \times Q \to M,$$

con

$$F(e, q) = (f_1(e, q), f_2(e, q), \ldots, f_n(e, q)),$$

donde $f_i(e, q)$ sería la $f(e, q, q_i)$ de la definición dada).

La función de transición se puede representar mediante una matriz para cada símbolo de entrada:

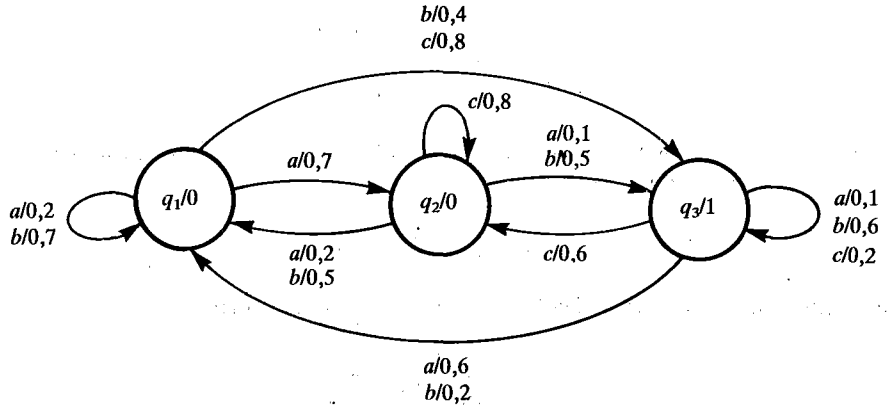
$$T(e) = \begin{bmatrix} f(e, q_1, q_1), f(e, q_1, q_2) \ldots f(e, q_1, q_n) \\ f(e, q_2, q_1), f(e, q_2, q_2) \ldots f(e, q_2, q_n) \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ f(e, q_n, q_1), f(e, q_n, q_2) \ldots f(e, q_n, q_n) \end{bmatrix}$$

Ejemplo:

$$E = \{a, b, c\};\ S = \{0, 1\};\ Q = \{q_1, q_2, q_3\}$$
$$h(q_1) = h(q_2) = 0;\ h(q_3) = 1$$

$$T(a) = \begin{bmatrix} 0{,}2 & 0{,}7 & 0 \\ 0{,}2 & 0 & 0{,}1 \\ 0{,}6 & 0 & 0{,}1 \end{bmatrix};\ T(b) = \begin{bmatrix} 0{,}7 & 0 & 0{,}4 \\ 0{,}5 & 0 & 0{,}5 \\ 0{,}2 & 0 & 0{,}6 \end{bmatrix};\ T(c) = \begin{bmatrix} 0 & 0 & 0{,}8 \\ 0 & 0{,}8 & 0 \\ 0 & 0{,}6 & 0{,}2 \end{bmatrix}$$

Gráficamente:



El concepto recuerda al de autómata probabilista (AP), pero hay dos diferencias importantes:

a) Puesto que no se trabaja con probabilidades, no es necesario que las filas de las matrices de transición tengan componentes cuya suma sea la unidad.
b) En el AP, para calcular las probabilidades de transición para una cadena $x = e_1, e_2, \ldots, e_n$ se multiplicaban las sucesivas matrices. Así, si $x = e_1, e_2,$

$$f(x, q_i, q_j) = \sum_k [f(e_1, q_i, q_k) \cdot f(e_2, q_k, q_j)]$$

En el borroso, se aplica la regla de composición de funciones:

$$f(x, q_i, q_j) = \max_k[\min(f(e_1, q_i, q_k), f(e_2, q_k, q_j))],$$

lo que equivale a sustituir el producto ordinario de matrices por el producto máx-mín.

Por ejemplo, la función de transición para la cadena $x = bca$ en el autómata anterior será:

$$T(bca) = T(b)T(c)T(a) =$$

$$= \begin{bmatrix} 0{,}7 & 0 & 0{,}4 \\ 0{,}5 & 0 & 0{,}5 \\ 0{,}2 & 0 & 0{,}6 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0{,}8 \\ 0 & 0{,}8 & 0 \\ 0 & 0{,}6 & 0{,}2 \end{bmatrix} \begin{bmatrix} 0{,}2 & 0{,}7 & 0 \\ 0{,}2 & 0 & 0{,}1 \\ 0{,}6 & 0 & 0{,}1 \end{bmatrix} =$$

$$= \begin{bmatrix} 0{,}7 & 0 & 0{,}4 \\ 0{,}5 & 0 & 0{,}5 \\ 0{,}2 & 0 & 0{,}6 \end{bmatrix} \begin{bmatrix} 0{,}6 & 0 & 0{,}1 \\ 0{,}2 & 0 & 0{,}1 \\ 0{,}2 & 0 & 0{,}1 \end{bmatrix} = \begin{bmatrix} 0{,}6 & 0 & 0{,}1 \\ 0{,}5 & 0 & 0{,}1 \\ 0{,}2 & 0 & 0{,}1 \end{bmatrix}$$

Si inicialmente estamos en el estado q_1, el subconjunto borroso de Q que expresa el grado de confianza de estar en cada estado será $Q_B^0 = \{q_1/1; q_2/0; q_3/0\}$; tras aplicar la cadena *bca*, se transformará en

$$Q_B^{bca} = [1 \quad 0 \quad 0] \begin{bmatrix} 0{,}6 & 0 & 0{,}1 \\ 0{,}5 & 0 & 0{,}1 \\ 0{,}2 & 0 & 0{,}1 \end{bmatrix} = [0{,}6 \quad 0 \quad 0{,}1] = \{q_1/0{,}6; q_3/0{,}1\}.$$

Y lo mismo que con el autómata probabilista, puede definirse un *autómata borroso de estructura variable*, ABEV = $\langle E, S, Q, f, h, A \rangle$, donde A es el *esquema de refuerzo o de actualización*, que genera f_{t+1} en función de f_t, e_t, s_t, y utilizarlo en un entorno aleatorio como sistema de aprendizaje. La ventaja con relación al APEV es que los esquemas de actualización están basados en productos máx-mín de matrices, lo que le confiere mayor velocidad. El principal inconveniente es la dificultad para establecer estudios analíticos sobre las propiedades de convergencia.

7. RESUMEN

El modelo básico de autómata finito (AF) expuesto en los anteriores capítulos puede generalizarse en varias direcciones. Las redes de Petri (RdP) y los modelos derivados de ellas son generalizaciones especialmente útiles porque permiten representar de forma más expresiva que con AF sistemas discretos compuestos por subsistemas que se comunican entre sí. Ello tiene gran interés, por ejemplo, en los campos de procesamiento distribuido, redes de ordenadores y protocolos de comunicación.

Otra vía de generalización surge al considerar modelos probabilistas o basados en la teoría de conjuntos borrosos. El interés aquí está en poder modelar sistemas de gran complejidad en los que no se conocen todas las relaciones causales. Y también el de servir de base para el diseño, mediante autómatas de estructura variable, de sistemas que «aprenden» (mejoran su comportamiento) al ir interactuando con el entorno.

8. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

Las RdP proceden del modelo propuesto por Carl Adam Petri (1962) en su tesis doctoral. Posteriormente han sido desarrolladas por muchos autores y han servido de base a otros modelos, como el de Bochmann (1979) para sistemas distribuidos. El lector interesado puede encontrar más detalles en un magnífico artículo de Peterson (1977), o, para mayor profundidad y extensión, en el libro de Silva (1985).

El concepto de autómata estocástico fue introducido al comienzo de los años 60 en USA (Rabin, 1963) pensando más bien en el objetivo de mejorar la fiabilidad de los circuitos secuenciales (Winograd y Cowan, 1963) que en su aplicación a sistemas de aprendizaje. Previamente, von Neumann (1956) había introducido la noción de probabilidad en el modelo de neurona formal de McCulloch y Pitts (1943) para

demostrar que, gracias a la redundancia, pueden sintetizarse sistemas muy fiables a partir de componentes poco fiables.

El interés por los sistemas de aprendizaje tiene su origen en la URSS. Tsetlin (1961) introduce la idea de utilizar AF deterministas en un entorno aleatorio como modelos de aprendizaje. Algo más tarde, Varshavskii y Vorontsova (1963) demuestran que el número de estados se reduce si se utilizan autómatas estocásticos con actualización de las probabilidades de transición (es decir, con estructura variable).

Una buena referencia para adquirir una visión global sobre autómatas de aprendizaje es el artículo de Narendra y Thathachar (1974). Y sobre sus aplicaciones, algunos capítulos del libro de Glorioso y Colón (1980).

Los autómatas de aprendizaje son un caso particular de los sistemas de aprendizaje en general, que también tienen aplicaciones dentro del campo de la ingeniería del conocimiento. En la introducción del capítulo 6 del tema «Lógica» mencionábamos el problema de la adquisición de conocimiento y la posible vía de su inducción automática, que, en definitiva, es aprendizaje. Una referencia muy completa para estudiar este asunto son los libros de Michalski *et al.* (1983, 1986).

Tercera parte
ALGORITMOS

Capítulo 1
IDEAS GENERALES

1. ALGORITMOS Y ORDENADORES

La palabra *algoritmo* procede del apellido latinizado de un matemático árabe, Mohamed ibn Mûsâ al-Khowârizmî (o al-Khârezmi) que en 820 y 825 d.C. escribió respectivamente dos tratados, el primero de cálculo con los números hindúes y el segundo de resolución de ecuaciones. La deformación del título de esta última obra ha originado el nombre de álgebra, con el que se conoce a la rama de las matemáticas consagrada al cálculo literal.

En nuestro siglo, no solamente se ha hecho más frecuente el uso del término *algoritmo*, sino que su contenido, su significación precisa, ha sido explorado profundamente. En la técnica de los computadores es palabra de uso cotidiano, aunque es necesario advertir que no todos los profesionales de la informática la conocen y emplean en su sentido cabal.

Lo cierto es que la aparición de los ordenadores con sus extraordinarias posibilidades de proceso y almacenamiento de información ha impulsado fuertemente el estudio de los algoritmos. Tanto es así, que a veces es difícil percibir que pueden distinguirse *dos campos de interés*: uno, de índole fundamental en matemáticas, en donde se plantea el problema de *si tienen solución ciertos tipos de problemas, lo que es equivalente a preguntarse si existe un algoritmo que conduzca siempre a una solución para esa clase de problema*. En el próximo capítulo se abordarán las definiciones formales del concepto de algoritmo, pero por el momento podemos aceptar que un algoritmo es, intuitivamente, la expresión de una secuencia precisa de operaciones que conduce a la resolución de un problema.

El otro campo de interés investiga o afronta *los problemas que surgen de la aplicación de las máquinas computadoras al procesamiento de algoritmos*. En este campo caen, entre otras, las relaciones de los algoritmos con las estructuras de los datos y con los lenguajes de programación, y la complejidad computacional (o

algorítmica), es decir, la cantidad de recursos necesaria para computar determinados problemas. El estudio de la complejidad algorítmica ha adquirido en las dos últimas décadas un nivel teórico considerable.

Puesto que hablamos de resolver problemas, será bueno tener una idea general de los principales aspectos implicados en dicha tarea. La resolución de un problema implica un proceso de varias etapas, que, a grandes rasgos, son las expresadas en el flujograma de la figura 1.1.

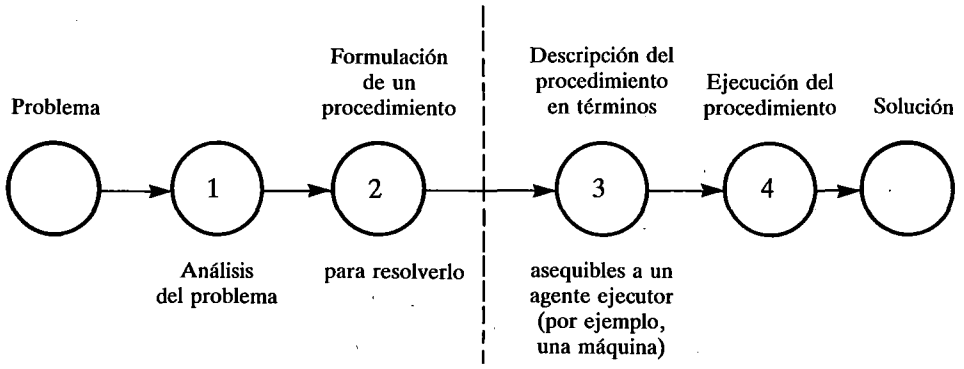


FIGURA 1.1.

Si el problema abordado tiene solución, la descripción del procedimiento para llegar a ésta estará fuertemente teñida por las características interpretativas y operativas del agente ejecutor del procedimiento que, en nuestro caso, será normalmente una máquina, y en particular un ordenador.

Así pues, el momento del proceso que se señala con una línea vertical de trazos marca la frontera habitual desde donde la principal preocupación del autor del algoritmo empieza a ser su comunicación con la máquina. Cuanto más evolucionada sea ésta, menor será el esfuerzo del autor. De hecho, podemos esperar que la etapa 3 pueda desglosarse en una secuencia $3'$, $3''$, $3'''$, $3''''$, ... tal que las reglas descriptivas en un lenguaje sean transformadas por una máquina M' en reglas descriptivas en otro lenguaje, etc., sucesivamente, hasta llegar a una máquina que sea capaz de ejecutar dichas reglas.

Por la misma razón, las etapas 2 y 3 pueden verse fundidas si se es capaz de formular el procedimiento directamente en términos de reglas o instrucciones para una máquina. En definitiva, la secuencia $3'$, $3''$, $3'''$, ..., supone contar con la apoyatura de un encadenamiento de algoritmos de transformación de expresiones simbólicas, y, por consiguiente, la repetición otras tantas veces, y a niveles y para problemas distintos, del proceso 1-2-3-4. Resumiendo, *el grado de concentración o multiplicación de las fases dependerá del conocimiento del sujeto acerca del problema a resolver y de la operatividad de los recursos ejecutores de que disponga.*

2. ALGORITMOS, LENGUAJES Y PROGRAMAS

La referencia anterior al papel del lenguaje en la expresión de la secuencia de operaciones que constituye el algoritmo hace inevitable entrar a considerar las relaciones existentes entre los tres términos que dan título a este apartado.

Con un programa, escrito en un lenguaje concreto, expresamos nuestro algoritmo para ser procesado por una máquina concreta, provista del adecuado procesador de lenguaje. De manera que *un programa para ordenador es la expresión de un algoritmo en un lenguaje artificial formalizado.*

Hablábamos en el primer apartado del esfuerzo necesario para describirle un algoritmo a un ordenador. La noción de lenguaje pone de manifiesto en forma nítida cómo el grado de ese esfuerzo, entre otros factores importantes, es función de la potencia operativa del lenguaje y de su grado de sintonía funcional con las clases de operadores implicadas en el algoritmo en cuestión. El siguiente ejemplo permitirá visualizar de una manera nítida esta cuestión. Se trata de un programa para calcular el máximo común denominador de dos números *A* y *B*, siguiendo el conocido algoritmo de Euclides. Las expresiones del programa difieren, a veces considerablemente, en su forma y volumen según el lenguaje utilizado. Si empleamos el lenguaje PL/I obtenemos el siguiente programa:

```
IF A = 0 THEN
   ULTIMO: DO;
      MCD = B; RETURN; END;
IF B = 0 THEN DO;
      MCD = A; RETURN; END;
   AQUI: G = A/B;
/*Suponiendo G variable entera*/
   R = A - B * G;
IF R = 0 THEN GO TO ULTIMO;
   A = B; B = R; GO TO AQUI;
```

Este mismo algoritmo, descrito en un lenguaje de máquina o en un lenguaje ensamblador, sería mucho más largo y en cierta manera incomprensible para una mente humana no entrenada. Si le aplicásemos un lenguaje inapropiado como DYNAMO, que es un lenguaje de simulación de sistemas continuos, la expresión de este algoritmo podría hacerse prácticamente imposible o por lo menos extraordinariamente enrevesada. En cambio, resultaría muy sencilla de escribir y de comprender si contásemos con una máquina capaz de entender y ejecutar la instrucción CALL MAX ( ). Este programa consistiría simplemente en la sentencia CALL MAX (A, B, MCD), que presupone la existencia de una máquina especializada, con dos entradas *A* y *B* para los dos números a procesar y una salida MCD para el resultado.

Es evidente que no siempre se dispone de una máquina así y cuando se dispone de ella no podríamos asegurar que tenga existencia física en la forma que sugiere la figura 1.2. Se dispondría más bien de una máquina virtual que, en términos generales, es el ordenador armado con un programa procesador de lenguaje de unas determinadas potencia y funcionalidad. En tales circunstancias, *el autor del algoritmo puede*

desentenderse de las características de la máquina física y expresarlo en un formato adecuado a las características funcionales de la máquina virtual, esto es, *del lenguaje escogido*. Esta idea puede tal vez expresarse de otra manera diciendo que el lenguaje transforma el ordenador en una máquina virtual.

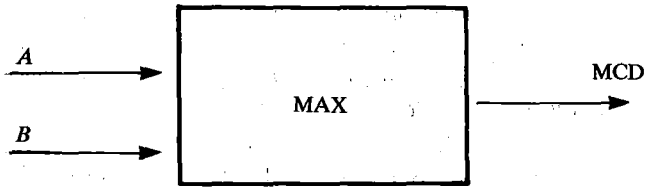


FIGURA 1.2.

Aunque la programación estructurada es una técnica ya muy conocida, se ha decidido incluirla en este tema porque nos proporciona al menos tres elementos didácticos importantes. En primer lugar, y en relación con lo que se acaba de decir, este estilo de programación, integrado en un diseño descendente, desarrolla la noción de *recurso abstracto*, conceptualmente similar en la práctica a la de máquina virtual. En segundo lugar, el ejercicio profesional de la programación adolece de serios problemas y deficiencias, tales como costes elevados, falta de fiabilidad, dificultades de mantenimiento y otros, sobre los que un buen estilo de programar ejerce una influencia positiva. Y, por último, el origen de la programación estructurada, debido a un trabajo de Böhm y Jacopini publicado en 1966, está conectado al concepto de autómata y, en particular, al de máquina de Turing, artefacto central en el tratamiento de este tema en este libro. A la programación estructurada le dedicaremos dos capítulos.

3. ALGORITMOS Y MÁQUINAS DE TURING

En el año 1936, antes del advenimiento de los ordenadores, el matemático inglés A. M. Turing inventó una máquina computadora de una increíble simplicidad en su estructura lógica. El concepto de algoritmo puede muy bien estudiarse en términos de esquemas funcionales de máquinas de Turing, ya que, como veremos, éstas permiten disecar los algoritmos en una secuencia de las operaciones más elementales que cabe imaginar.

La máquina de Turing se ha convertido en *la piedra angular de la moderna teoría de algoritmos*. Siendo una máquina ideal, que cada uno define y construye con papel y lápiz, no se ve afectada por los avances tecnológicos. Es, pues, un invariante de la informática y también un símbolo. La asociación de informática profesional más prestigiosa, la A.C.M. (Association for Computing Machinery), entrega todos los años el premio Turing a uno de los científicos entre quienes más hayan contribuido a generar avances significativos en el dominio de la informática.

Con la máquina de Turing dispondremos de un artefacto teórico con el que

formalizar el concepto de algoritmo a través del concepto de *función computable*: un problema computable es un problema para el que existe un algoritmo. En su forma más sencilla, la máquina de Turing es un autómata finito que controla una cinta infinita. Véase en la figura 1.3 cómo, por afán de hacerlo más tangible y próximo a las máquinas físicas, se ilustraba gráficamente este autómata en un artículo de Wang de 1965, en la revista «Scientific American».

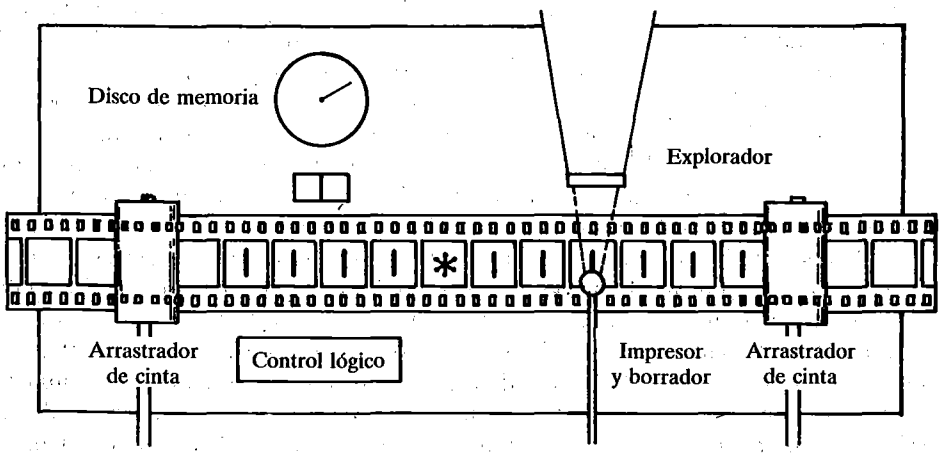


FIGURA 1.3.

Además de desarrollar algunas cuestiones de importancia teórica relacionadas con las máquinas de Turing y los algoritmos, habremos de enfatizar el análisis y el diseño de estas máquinas como una vía hacia la comprensión de la complejidad de la tarea de diseñar algoritmos cuando las máquinas virtuales son de muy bajo nivel (en otras palabras, cuando los lenguajes son de muy bajo poder expresivo) y de cómo, con un lenguaje tan elemental como el de la máquina de Turing, se puede expresar cualquier algoritmo. En tal sentido, existe una relación de la máquina de Turing con la complejidad algorítmica. Se dedicarán los capítulos 5 y 6 y algún apartado del séptimo a todas estas cuestiones.

No acaba aquí el interés de la máquina de Turing. Tratándose de un autómata, podría habérsela considerado en el tema «Autómatas». Sin embargo, su particular relevancia le ha hecho merecedora de estos capítulos específicos. Por parecidas razones, la relación de los autómatas con los lenguajes, brevemente abordada ya en «Autómatas», será ampliada en el tema de «Lenguajes». Ciertos autómatas son capaces de reconocer las cadenas de símbolos que constituyen el lenguaje generado por una determinada gramática. Las máquinas de Turing reconocen lenguajes generados por sistemas de escritura no restringidos, a los que se llama lenguajes de tipo 0. En el capítulo 6 encontraremos una breve consideración de este punto, a la espera de sistematizarlo en «Lenguajes».

4. COMPUTABILIDAD Y COMPLEJIDAD

Hemos dicho que un problema computable (o decidible) es aquel que admite solución algorítmica. *En los problemas computables es interesante estimar el orden de magnitud de los recursos computacionales* que requieren los distintos algoritmos que puedan resolverlos. La complejidad computacional es, en pocas palabras, ese orden de magnitud. Así, un problema concreto podría necesitar 500 años de cómputo continuado, por lo que, siendo un problema computable, lo consideraríamos un problema no factible, un problema de una complejidad intratable.

Los recursos habituales para procesar algoritmos son: tiempo, capacidad de memoria y velocidad de procesamiento. Por simplificar, fijémonos en el tiempo que, además, depende en alguna proporción de los otros dos factores señalados. Es evidente que, una vez fijado un algoritmo para un determinado problema, el tiempo de cálculo necesario depende del tamaño de los datos del problema. Por ejemplo, multiplicar dos números lleva más tiempo si éstos constan de 1.543 dígitos cada uno que si sólo constan de 10, y lo mismo puede decirse si de lo que se trata es de clasificar un conjunto de elementos: el tiempo es función del tamaño de los datos. La búsqueda de expresiones matemáticas para la estimación del tiempo de procesamiento de un algoritmo como función del tamaño de los datos de entrada entra en el campo de competencia de la teoría de la complejidad, que, entre otras actividades, evalúa por consiguiente el grado de factibilidad de los problemas. De esta manera se llega a clasificar los algoritmos por la expresión de su complejidad (comportamiento asintótico) en algoritmos de complejidad polinómica y exponencial. Sobre tales cuestiones hablaremos en el capítulo 7.

El lector debe comprender que el estudio de la complejidad algorítmica, pese a haber alcanzado un notable nivel teórico, no puede de ninguna manera ser catalogado como un campo especulativo, puesto que se ocupa de cuestiones eminentemente prácticas o proporciona herramientas para abordarlas. Imagínese un caso como el siguiente: para controlar por computador un reactor nuclear resulta vital conocer el máximo lapso con el que el algoritmo programado es capaz de responder a ciertos supuestos de emergencia. O el caso de tener que elegir un algoritmo cuya complejidad esté acotada entre un límite superior y uno inferior. O el de tener que evaluar la ventaja obtenida entre procesar un algoritmo en un ordenador de estructura secuencial o en otro de estructura paralela.

Otro tipo de complejidad, menos fundamental en sus implicaciones que el anterior, aunque de considerable importancia práctica, *es el de complejidad del software*. Le dedicaremos algún espacio en el mismo capítulo 7. En primera aproximación, esta complejidad tiene que ver con la estructura de predicados del programa construido para describir un determinado algoritmo. Cuanto más rica sea esta estructura parece intuitivamente evidente que la tarea del programador resulta más difícil, mayor su esfuerzo para desarrollarla y también mayor la probabilidad de error, puesto que se acrecienta el número de caminos de proceso que pueden tener que recorrer los datos tratados por el algoritmo. Existe un vínculo de proporcionalidad entre el número de predicados y el número de discriminaciones mentales del programador en su diseño y descripción del algoritmo. La fiabilidad y el coste del software guardan una fuerte relación con la riqueza de tal estructura.

Una forma de visualizar y hasta de medir esta complejidad consiste en asociar a la estructura de control de los programas un grafo orientado. Tomemos el ejemplo sencillo del programa en PL/I para calcular el m.c.d. de dos números, visto en el apartado 1.

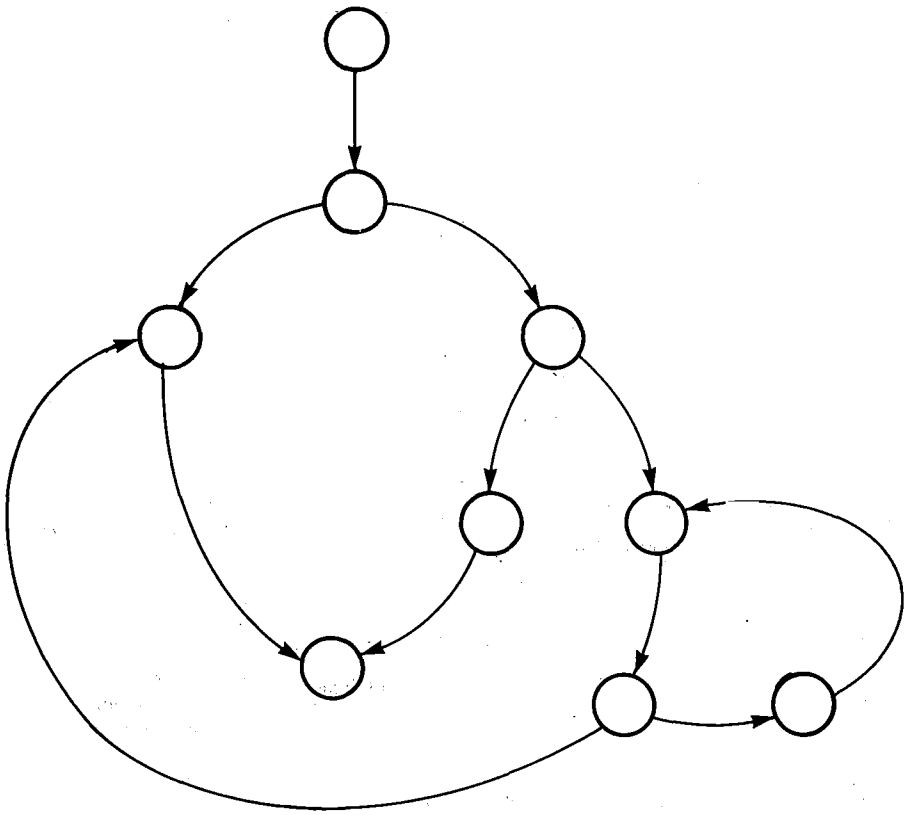


FIGURA 1.4.

La figura 1.4 es el grafo correspondiente a este programa, si suponemos que cada uno de sus nodos representa un bloque de código con flujo de control secuencial y los arcos representan ramas en el programa. No vamos a entrar en detalles ahora. Con lo dicho, cualquiera puede comprender que la dificultad asociada con el diseño y codificación de un programa ha de tener una relación con la magnitud y enrevesamiento del grafo. Esta es a grandes rasgos la complejidad del software. Al mismo tiempo se hace patente que dicha complejidad es función del nivel de lenguaje, puesto que la instrucción CALL MAX ( ), que resuelve el mismo problema, tiene un grafo absolutamente elemental.

5. RESUMEN Y CONCLUSIONES

No es fácil de resumir este capítulo. El esquema conceptual de la figura 1.5 intenta plasmar una imagen incompleta e imperfecta de la íntima relación entre algunos de los conceptos de que versa este libro, pero se pretende que sirva sobre todo como un plano-guía para los siguientes capítulos de este tema.

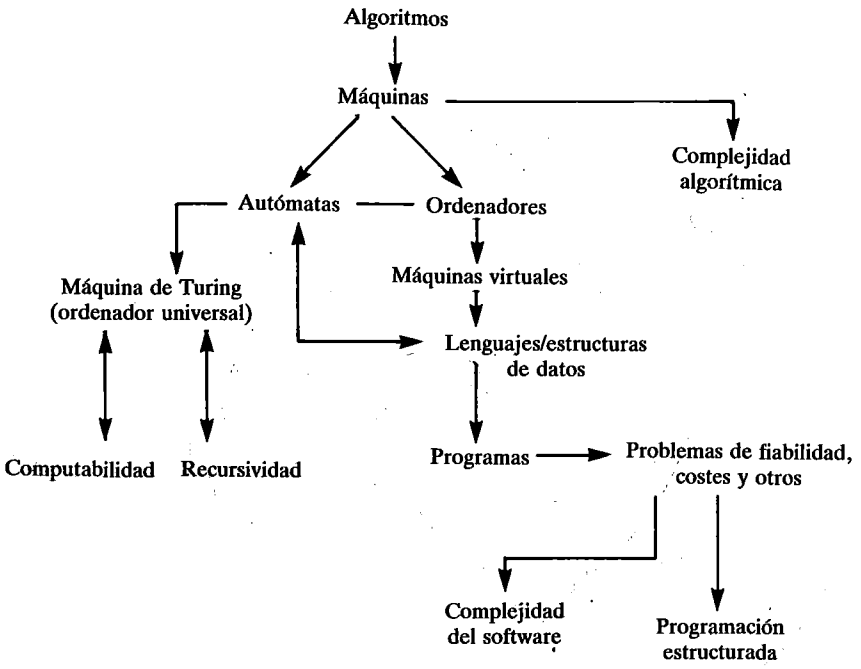


FIGURA 1.5.

El sentido de la flecha quiere significar algo parecido al sentido de una navegación conceptual. Así pues, partiendo del concepto de algoritmo, nos vemos conducidos al concepto de máquina ejecutora, bien sea un autómata, bien sea un ordenador, y éste último es básicamente un conjunto de autómatas. Un autómata muy especial, que resulta ser el ordenador universal, es la máquina de Turing, por cuyo intermedio se construye una teoría formal de los algoritmos y se llega a los conceptos de recursividad y computabilidad.

Los ordenadores llevan a la máquina virtual y al lenguaje (incluyendo las estructuras de datos) como niveles convenientes para describir procesos ejecutivos de los algoritmos. En cierta manera, y tal como ha escrito Wirth, «el propio ordenador consiste tan sólo en algoritmos y estructuras de datos» y éste es, en esencia, el núcleo de la naturaleza protéica (multiforme, multiuso) del ordenador. Lenguajes formales y autómatas se corresponden, siendo los distintos tipos de éstos, máquinas reconoce-

doras de aquellos, y por tanto, piezas básicas de los programas denominados procesadores de lenguaje (traductores, interpretadores y otros).

Los programas son en definitiva algoritmos descritos en un lenguaje para una máquina, que puede ser un ordenador o una máquina de Turing, por citar dos ejemplos extremos. Por cualquiera de los dos caminos nos encontramos con el problema de la complejidad algorítmica, entendida como cantidad de recursos computacionales necesarios para culminar la ejecución del algoritmo.

En el primer caso, además, tropezamos con problemas de costes y de fiabilidad. La programación estructurada es un método o estilo de programación, que, inicialmente justificado por estudios de teoría de autómatas, es representativo de un conjunto de técnicas ya clásicas para mejorar los resultados de la programación secuencial. Los mismos problemas han conducido a interesarse por definir y medir la complejidad de los programas. Naturalmente, la complejidad del software tiene relación con los algoritmos a través de la programación, del lenguaje y del nivel de máquina virtual disponible en el ordenador (siguiendo el esquema en sentido inverso), pero no tiene en principio una relación directa con el concepto de complejidad algorítmica. Para muchos autores, esta última complejidad constituye una parcela importante de la teoría de la programación, aunque otros le dan un carácter todavía más básico. En cambio, la complejidad del software es indiscutiblemente un aspecto práctico, metodológico, de la programación.

Un detalle final aparentemente anecdótico, pero que puede dar una cierta medida de la importancia informática atribuida a los conceptos aquí esbozados es que la contribución a su estudio ha proporcionado el premio Turing a varios investigadores, a saber: Knuth (algoritmos, estructuras de datos), Dijkstra (teoría de la programación), Wirth (lenguajes), Hoare (algoritmos, teoría de la programación) y Cook (complejidad algorítmica).

El último premio, correspondiente al año 1985, ha sido entregado a Karp por sus trabajos sobre complejidad en problemas de tipo combinatorio.

Capítulo 2
ALGORITMOS

1. INTRODUCCIÓN

El presente capítulo se dedica básicamente a definir el concepto de algoritmo. Primero empieza con unas definiciones diversas y no totalmente coincidentes, que se contrastarán. Se pasará a continuación a una definición formal en teoría de conjuntos, que comprende y precisa todas las anteriores.

De la definición se extraen unas propiedades o condiciones que debe cumplir todo algoritmo, a las que se añaden propiedades que se puede desear que cumplan los «buenos algoritmos».

El capítulo prepara ya el terreno para hablar de programas y de máquinas destacando en la definición formal aquellos elementos, como el número de orden de una ejecución y la existencia misma de los estados, que prefiguran la existencia de determinadas condiciones generales en cualquier máquina ejecutora de algoritmos.

2. DEFINICIONES DE ALGORITMO

No hay una, sino muchas definiciones de algoritmo. Aquí recuadramos varias de distintos autores, todas (¡y no es casualidad!) tomadas de libros sobre computación o computadores.

Definiciones de algoritmo

1. Lista de instrucciones que especifican una secuencia de operaciones que darán la contestación a cualquier problema de un tipo determinado.
2. Conjunto de reglas que define de manera precisa una secuencia de

operaciones tales que cada regla es efectiva y definida y tal que la secuencia termina en un tiempo finito.
3. Sucesión finita de prescripciones potencialmente ejecutables expresadas en un lenguaje definido que estipula cómo ejecutar un cierto encadenamiento de operaciones para resolver todos los problemas de un cierto tipo dado.
4. Sistema de reglas que permiten obtener una salida específica a partir de una entrada específica. Cada paso debe estar definido exactamente, de forma que pueda traducirse a lenguaje de computador.

Lo primero que sorprende es que las definiciones difieren no poco entre sí, aunque tal vez sólo sean las apariencias. Esto deja ya suponer que no son muy precisas. Pero fijémonos en las semejanzas y no en las diferencias.

1.° Hay unas *reglas*, o instrucciones, o prescripciones.
2.° Tales reglas, instrucciones, etc., especifican una *secuencia* (encadenamiento) *de operaciones* o pasos.
3.° Aunque de forma muy implícita (excepto en la cuarta definición) las operaciones se supone han de ser llevadas a cabo por un *agente ejecutor*, máquina o ser vivo, por sí o a través de otros agentes. Agente que es el destinatario de las instrucciones.
4.° Más implícitamente aún, pero contenido en las definiciones, se establece que la secuencia de operaciones tiene una *duración*, que podrá ser tan larga como se quiera, pero ha de ser *finita*.

Con estos elementos, que constituyen un común denominador de lo que hemos podido descubrir hasta ahora sobre la conceptualización moderna de los algoritmos, vemos que es posible elaborar algoritmos para resolver muchas clases de problemas. Así, por ejemplo, encontrar el máximo común divisor de dos números enteros, investigar si una palabra determinada figura en una tabla de palabras almacenada en la memoria de un computador, jugar una partida de ajedrez o tornear una pieza complicada.

La amplitud del campo de los problemas es tan grandiosa que cualquiera percibe la dificultad de aprehender la noción de algoritmo si uno se sitúa en medio de la diversidad de los problemas y la diversidad de los agentes ejecutores. Es obligatorio reducir el ámbito de atención e investigar el asunto como un proceso intelectual independiente del problema específico, por una parte. De ahí se desprende que, si bien hay algoritmos numéricos y no numéricos, en última instancia todos pueden reducirse a la especificación de operaciones sobre símbolos. Y por otra, es obligatorio independizarse en lo posible del agente ejecutor de estas operaciones simbólicas y, para ello, una solución ha consistido en definir un agente ejecutor único (veremos que será la máquina de Turing), al cual podrían reducirse en última instancia todos los demás.

3. ALGORITMOS Y MÁQUINAS

Fijémonos ahora en la ejecución del procedimiento o algoritmo, con independencia del problema y de la máquina de que se trate, para lo cual daremos unas definiciones más formales que en el apartado anterior.

3.1. El concepto de algoritmo, visto desde la teoría de conjuntos

Se define un algoritmo como una cuádrupla A,

$$A = \langle Q, E, S, F \rangle \tag{1}$$

con

- Q: Conjunto de todos los elementos simples y de todas las K-uplas que pueden describir el cálculo.
- E: Subconjunto de Q. Sus elementos son los datos de entrada al proceso de cálculo.
- S: Subconjunto de Q. Sus elementos son los diferentes resultados al término del cáiculo.
- F: $Q \rightarrow Q$, aplicación que describe la regla de cálculo propiamente dicha y que, a partir de cualquier elemento q_0, genera la construcción de una sucesión q_0, q_1, q_2 ... tal que:

$$q_{i+1} = F(q_i),\ i \in N \text{ con } q_0 \in E \subset Q \tag{2}$$

Para que A represente un algoritmo, cada sucesión (2) debe ser finita. Así pues, es preciso, como condición necesaria no suficiente, que la función F deje invariante el subconjunto S; es decir: $\forall s \in S$, $F(s) = s$. Si la sucesión es finita su punto de parada viene dado por el menor índice i para el que $q_i \in S$. No será finita si $\nexists i \in N$; $F(q_i) \in S$.

Ejemplo: Cálculo del m.c.d. de dos números enteros no negativos n_1 y n_2. El organigrama de la figura 2.1 expresa un procedimiento conocido para resolver este problema. (P.E. significa «parte entera» de dividir n por n').

Este algoritmo que se expresa en la figura 2.1 en una forma, en parte icónica (diagrama), en parte formal (expresiones matemáticas), puede representarse a través del lenguaje de los conjuntos así:

Q: $(\langle n \rangle, \langle n, n' \rangle, \langle n, n', r, 1 \rangle, \langle n, p, r, 2 \rangle, \langle p, n', r, 3 \rangle)$, $n, n', p, r \in N$

E: $(\langle n, n' \rangle)$

S: $(\langle n \rangle)$

F: $F(n_1, n_2) = (n_1, n_2, 0, 1)$ si $n_1 > n_2$, si no $F(n_1, n_2) = (n_2, n_1, 0, 1)$;
$F(n) = (n)$;
$F(n, n', r, 1) = (n)$ si $n' = 0$, si no

$F(n, n', r, 1) = (n, n', n - n' \times \text{P.E.}(n/n'), 2);$
$F(n, p, r, 2) = (p, p, r, 3);$
$F(p, n', r, 3) = (p, r, r, 1);$

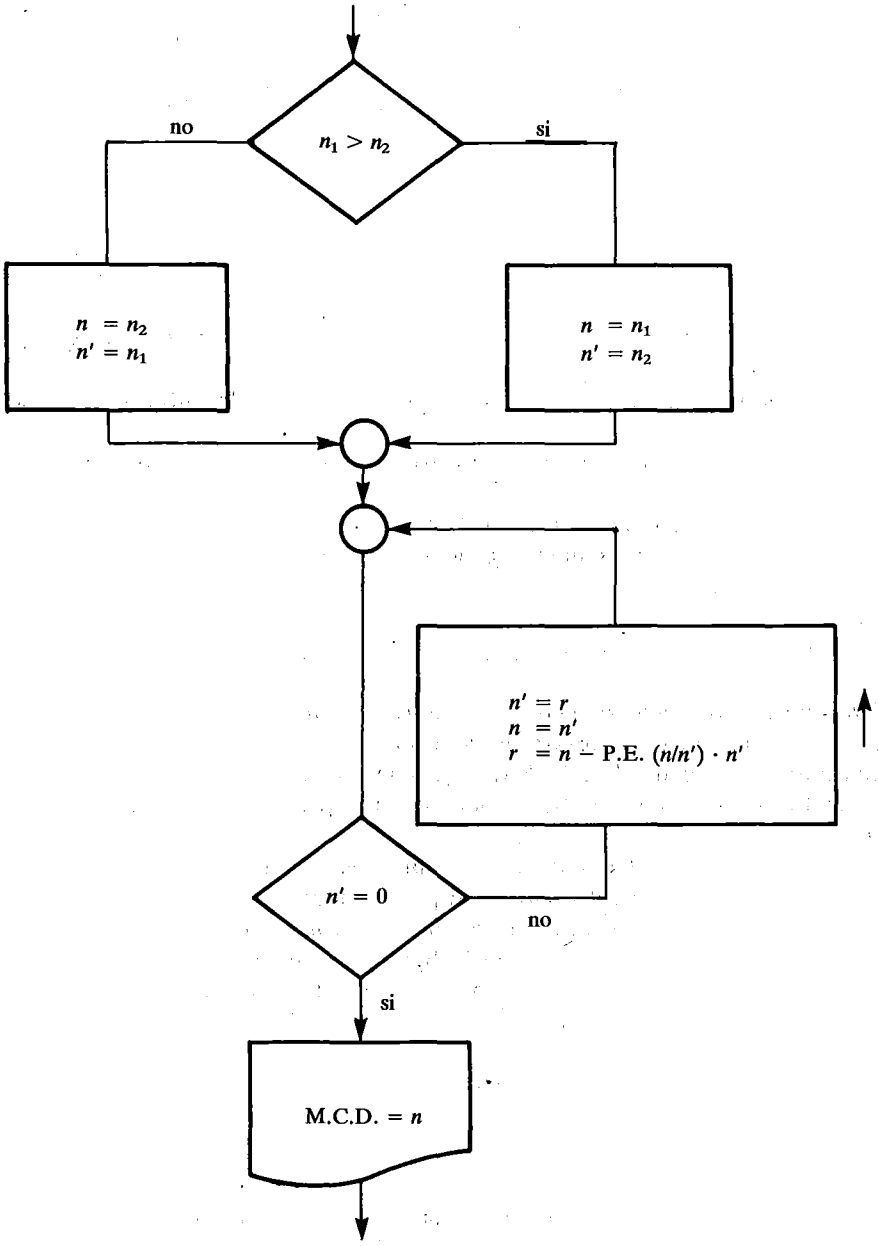


FIGURA 2.1.

El signo ; separa las distintas variantes de la regla de cálculo que el lector puede usar para hallar la sucesión $q_0, \ldots, q_i, \ldots$ partiendo de cualquier pareja de números enteros no negativos, familiarizándose así con la definición que se acaba de dar. Por cierto, que dicha definición habría que perfeccionarla en el sentido de que *la aplicación F no implique operaciones que no se sepa realizar*. En definitiva, las restricciones que habría que imponer a Q, E, S y F son de naturaleza tal que la cuádrupla A no contenga más que operaciones elementales simples (repare el lector en la relatividad del argumento de simplicidad, siendo la operación más simple la que pueda ejecutar un autómata).

Un ejemplo como el del cálculo del m.c.d. pone de manifiesto, mediante el empleo de los números 1, 2, 3, etc., la idea de que *una aplicación o un cálculo es una secuencia de aplicaciones o cálculos más elementales y que tal secuencia puede expresarse mediante un número asignado a las órdenes que deben ejecutarse*.

De forma general, podría decirse que el estado del algoritmo es un par (a, j), donde j indica el número de la orden que debe ejecutarse y a es la información que caracteriza el estado del algoritmo cuando hay que ejecutar la orden j (información que comprende combinaciones de datos, resultados intermedios y resultados finales). Así, a cada evaluación de la aplicación $F(q_i) = F(a_i, k) = (a_{i+1}, l)$ se le puede llamar ejecución de la orden k del algoritmo. El conjunto S incluiría la información correspondiente a los resultados finales y el conjunto E, la información de los datos de entrada.

El concepto de algoritmo puede examinarse formalmente también en relación con un alfabeto y en términos de una máquina de Turing, cosa esta última que haremos más adelante. De momento, veamos más de cerca y en forma intuitiva las implicaciones de los algoritmos sobre las máquinas.

3.2. Las máquinas, como estructuras capaces de ejecutar algoritmos

Toda máquina capaz de ejecutar algoritmos (fase 4 del proceso de la figura 1.1) debe tener una estructura con los subsistemas que se destacan en la figura 2.2.

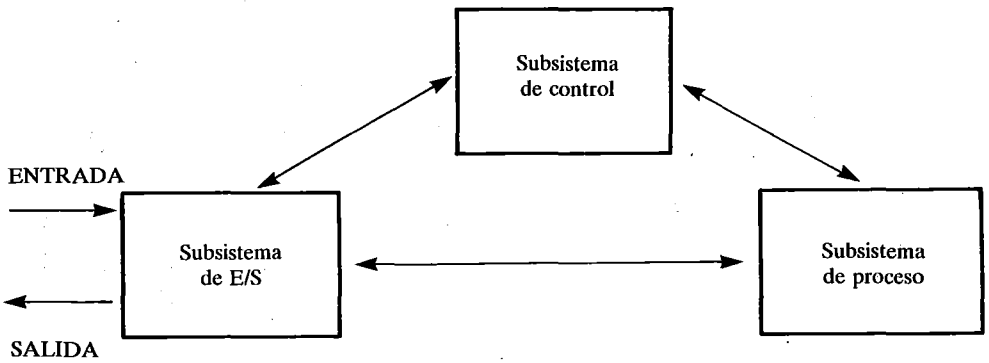


FIGURA 2.2.

Siguiendo literalmente este enfoque, se definen a continuación los subsistemas básicos de una máquina general ejecutora de algoritmos y el subsistema de memoria que resulta de agrupar todos (o parte) de los elementos de memorización necesarios a la máquina.

3.2.1. *Subsistema de entrada/salida (E/S)*

Donde se ejecutarán las órdenes de comunicación de la máquina y su mundo exterior.

3.2.2. *Subsistema de proceso*

Donde se ejecutarán las órdenes de tratamiento de datos que dan lugar a la realización de operaciones del algoritmo.

3.2.3. *Subsistema de control*

Donde se consigue el secuenciamiento adecuado en la ejecución de las órdenes y donde se generan las señales de control adecuadas para el funcionamiento de los subsistemas definidos en los dos apartados anteriores.

Por lo visto en el apartado 3.1, la ejecución de las órdenes de un algoritmo genera un nuevo estado q_{i+1} a partir del estado anterior q_i. Ello implica que hay que memorizar en la máquina el estado del algoritmo. Esto es lo mismo que decir que los

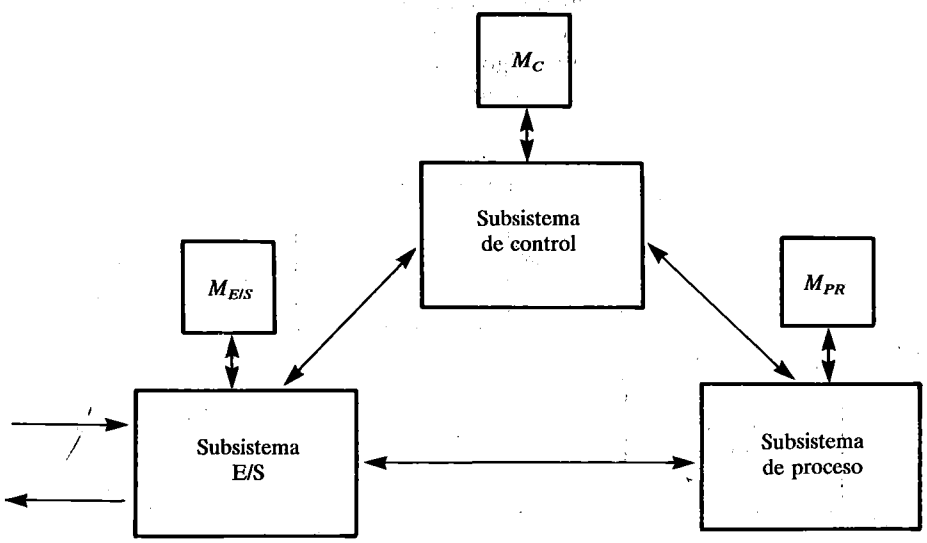


FIGURA 2.3.

subsistemas podrán llevar asociados elementos de memoria (asociados siempre, como se sabe, a todo circuito secuencial) para memorizar las informaciones de estado que corresponden a la misión de cada subsistema.

En muchas máquinas (y esto es precisamente lo que ocurre en los ordenadores) resulta más conveniente agrupar todos o una parte importante de los mencionados elementos de memorización en un subsistema específico, el subsistema de memoria (figura 2.4).

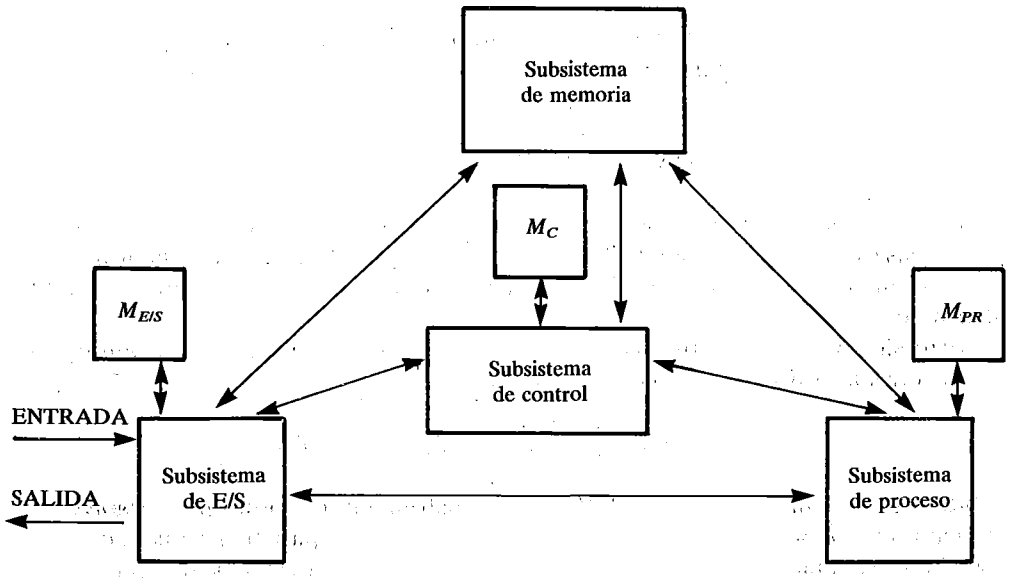


FIGURA 2.4.

4. PROPIEDADES DE LOS ALGORITMOS

Como resumen y ampliación de lo dicho, describimos en este apartado las cuatro propiedades que debe poseer todo buen algoritmo.

4.1. Propiedad de finitud

Un algoritmo debe siempre terminarse. Todo algoritmo puede subdividirse en un número de subalgoritmos tan grande como se quiera, pero finito, admitiendo cada uno de estos subalgoritmos cadenas últimas que se terminan.

4.2. Propiedad de definitud

Toda regla de un algoritmo debe definir perfectamente la acción a desarrollar, para aplicarla sin que pueda haber lugar a ambigüedad alguna de interpretación.

4.3. Propiedad de generalidad

Un algoritmo no debe contentarse con resolver un problema particular aislado sino, por el contrario, toda una *clase de problemas* para los que los datos de entrada y los resultados finales pertenecen respectivamente a conjuntos específicos.

4.4. Propiedad de eficacia

Aun cuando un algoritmo posea las tres propiedades anteriores, se busca mejorarlo por razones de economía, de realizabilidad o de rapidez.

5. PROBLEMAS SIN ALGORITMO

Los matemáticos han mostrado históricamente su deseo de resolver tipos de problemas cada vez más generales. Este deseo, como se ha visto, inspira la propiedad de generalidad, que es una propiedad relacionada con el grado de potencia de un algoritmo.

Esto significa, así en abstracto, que sería preferible construir un algoritmo para hallar todas las raíces de una ecuación del tipo

$$a_n n^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 = 0$$

donde n es un entero positivo arbitrario, a construir un algoritmo para resolver la ecuación $x^n - a = 0$, o, más sencillo todavía, un algoritmo para hallar raíces cuadradas. Ante este planteamiento surge inmediatamente una restricción de orden pragmático en escoger el nivel de generalidad congruente con los propósitos de quien tenga que resolver un problema. También, como se verá, al elevar el nivel de generalidad puede penetrarse en un entorno donde no existan soluciones o, al menos, soluciones conocidas.

Ahora bien, cuando el propósito es de índole teórica, se puede llegar a elevar dicho propósito hasta el nivel del sueño de Leibniz, que consistía en buscar un algoritmo para resolver cualquier problema matemático. Refinado este enunciado, dio en uno de los más famosos problemas de la lógica matemática, *el problema de la deducción*.

Es sabido que, utilizando símbolos, cualquier proposición de una teoría matemática puede escribirse mediante una fórmula y esta fórmula es una palabra definida en un alfabeto. Entonces, la derivación lógica de una proposición se convierte en una cadena de transformaciones de palabras (cálculo lógico). El problema de la deducción se puede formular así:

Para dos palabras cualesquiera (fórmulas) R y S de un cálculo lógico, determinar si existe o no una cadena deductiva de R a S. (S es la proposición y R es la premisa).

Se supone que la solución es un algoritmo para resolver cualquier problema de este tipo. Dicho algoritmo daría un método general para resolver problemas en todas las

teorías matemáticas que se construyen de forma axiomática. La validez de cualquier proposición *S* en tal teoría sólo significa que puede deducirse del sistema de axiomas. Después, la aplicación del algoritmo determinaría si la proposición *S* era válida o no. Además, si la proposición *S* fuese válida, entonces podríamos encontrar un encadenamiento deductivo correspondiente en el cálculo lógico y de ahí recuperar un encadenamiento de inferencias que probaría la proposición.

Hasta ahora no se ha encontrado tal algoritmo. De hecho, no se han encontrado algoritmos para problemas menos generales. El matemático alemán Hilbert presentó en 1901, en un congreso que se celebraba en París, una lista de 20 problemas no resueltos. El décimo problema se enunciaba de la forma siguiente: *Hallar un algoritmo para determinar si cualquier ecuación diofántica dada tiene una solución entera.*

Precisamente se conoce un algoritmo para resolver una ecuación diofántica con una incógnita: $a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 = 0$, pero no cuando contiene varias incógnitas, como es el caso de la siguiente ecuación: $a^2 + b^2 - c^2 = 0$.

Otro problema que no tiene solución conocida es el de averiguar si dos sucesiones de ceros y unos están relacionadas.

Podemos formar una sucesión de ceros y unos que tenga «descendencia» mediante las siguientes reglas:

1. Si la sucesión tiene menos de tres símbolos, parar.
2. Si la sucesión comienza por 0, borrar los tres primeros símbolos y añadir 00 al final.
3. Si la sucesión comienza por 1, borrar los tres primeros símbolos y añadir 1101 al final.

¿Hay un algoritmo para determinar si una de las dos sucesiones dadas es descendiente de la otra?

011010001001	101110110011
01000100100	1101100111101
0010010000	11001111011101
001000000	011110111011101
00000000	11011101110100
0000000	111011101001101
000000	0111010011011101
00000	101001101110100
0000	0011011101001101
000	10110100110100
00	1101001101001101
(parar)	(descendencia continua)

Si prescindimos de los detalles operativos relacionados con la ejecución de los algoritmos (detalles de enorme importancia en la práctica) puede decirse: a) que *un problema tiene solución cuando se es capaz de demostrar formalmente que existe un método que genera siempre, a partir de todo elemento del conjunto E, un elemento del conjunto S (expresión (1)) en un número finito de pasos*; b) *no existe solución cuando se es capaz de demostrar formalmente la inexistencia de algún método que pueda generar un elemento de tal conjunto S en las mismas condiciones expresadas en a).* Y, por último, c) *no existe solución conocida cuando no se es capaz de demostrar una cosa o la otra.*

6. RESUMEN

Para empezar *se han definido verbalmente los algoritmos* como un conjunto o lista de reglas, instrucciones o prescripciones que especifican a un agente ejecutor una secuencia finita de operaciones para la resolución de un problema. El problema ha de ser lo más general que sea posible, aunque esta condición es siempre bastante relativa.

Cuando se pretende *resolver un problema, se desarrolla un proceso de varias etapas* en las que normalmente unas son más próximas a la estructura propia del problema (etapas de análisis y de formulación de un procedimiento de solución) y otras más decantadas del lado de las capacidades operativas del (o de los) agente ejecutor (en muchas ocasiones, una máquina y modernamente casi siempre un computador, al menos en nuestro caso). Este proceso resulta tanto más largo o complejo cuanto mayor sea la distancia entre la complejidad del problema y la capacidad operativa del agente ejecutor disponible.

Si se hace caso omiso de este último factor, puede plantearse *una definición formal de algoritmo* como una cuádrupla $A = \langle Q, E, S, F \rangle$, donde la regla de cálculo F: $Q \rightarrow Q$ lleva al algoritmo desde un estado inicial ($\in E$) a un estado final ($\in S$) en un número finito de pasos, supuesto que se sepa realizar F con todos los elementos de Q.

Pensando ahora precisamente en términos del factor anteriormente descartado, vemos que el uso y el concepto de estado del algoritmo, en donde puede distinguirse un número de orden de ejecución, lleva a concebir un *agente ejecutor-máquina como una estructura con tres o cuatro subsistemas* (entrada/salida, proceso, control y memoria).

No se tiene un algoritmo si el procedimiento elaborado no cumple las condiciones de *finitud y definitud*, a las que pueden añadirse, si se quiere hablar de «buenos algoritmos», las de generalidad y eficacia, relacionadas con el grado de ambición o de optimización con que se enfoque la resolución de un problema.

De todas formas *hay problemas que no tienen solución o*, al menos, *solución conocida*. Esta última parte del capítulo nos muestra cómo el grado de generalidad de un algoritmo tiene unos límites y nos pone delante de temas fundamentales de las matemáticas modernas.

7. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

Aparte la mención al origen de la palabra *algoritmo*, creada en homenaje a un matemático árabe, no es fácil establecer una evolución histórica del concepto moderno que tal palabra contiene. Referencias a matemáticos célebres como Leibniz, Hilbert y otros, ya más entrado el presente siglo, no parecen pertinentes a la redacción de este libro.

Es indudable, y esto conviene resaltarlo de nuevo, que la aparición de los ordenadores hacia la década de los cuarenta impulsó un gran interés por trabajos anteriores teóricos de Turing y otros científicos.

En lo que se refiere a este capítulo, se han empleado los trabajos que a continuación se relacionan.

Las definiciones de algoritmo del apartado 2 se deben por el mismo orden a las

siguientes referencias: (Trakhtenbrot, 1960); (Stone, 1972); (Corge, 1975) y (Knuth, 1977).

Para la presentación de la relación entre algoritmos y máquinas hemos seguido el planteamiento de Alabau y Figueras (1975).

Las propiedades de los buenos algoritmos se encuentran en Corge (1975).

En cuanto a la formulación del problema de la deducción se ha tomado de Trakhtenbrot (1960) y, por último, el problema de las dos sucesiones de ceros y unos, de H. Wang (1974).

Capítulo 3
PROGRAMACION ESTRUCTURADA: CONCEPTOS TEORICOS

1. INTRODUCCIÓN

El tema de la programación estructurada se desarrolla en dos capítulos, dedicado este primero a presentar los *conceptos más importantes de la teoría de la P.E.* (así la llamaremos frecuentemente en adelante) y el capítulo 4, a describir cómo aplicar en forma coherente dichos conceptos en un *proceso de diseño de programas.*

En el capítulo 3 se contesta a las preguntas siguientes: *¿Qué es un programa y cuál es su relación con los algoritmos? ¿Qué es un ordinograma y cuál es su relación con los programas? ¿Qué es un programa estructurado y si se puede y cómo se puede transformar un programa no estructurado en un programa estructurado?* Y, por último, *¿qué ventajas e inconvenientes comporta la P.E.?*

2. DEFINICIÓN FORMAL DE PROGRAMA PARA ORDENADOR

Es una sucesión de reglas llamadas instrucciones que definen completamente un tratamiento para ser ejecutado en ordenador y que, en la ejecución, es la realización de un cierto algoritmo $A = \langle Q, E, S, F \rangle$. Se puede representar por una cuádrupla

$$P = \langle X, x_0, Y, \varphi \rangle$$

con:

X : Conjunto de todos los estados del ordenador.
$x_0 \in X$: Estado inicial de la máquina, programa cargado y datos iniciales.
Y : Conjunto de los estados finales de la máquina después del cálculo.
φ : Función de transición de un estado interno al estado interno siguiente.

P representa a A si existe una aplicación g de E en x_0, una aplicación h de X sobre Q aplicando Y en S y una función r aplicando X en N tal que:

— Si $e \in E$ produce el resultado s a partir de e si y solamente si existe un y en Y que pueda ser producido como resultado por P a partir de $g(e)$ y tal que $h(y) = s$.
— Si $x \in X$, entonces $F(h(x)) = h(\varphi^{r(x)}(x))$, donde la notación $\varphi^{r(x)}$ significa que la función φ debe ser iterada $r(x)$ veces.

3. DIAGRAMAS DE FLUJO, ORGANIGRAMAS, ORDINOGRAMAS

La mayoría de los algoritmos se ejecutan modernamente con ordenador. Aunque un ordenador es en definitiva un autómata finito o, si se quiere, un conjunto de autómatas finitos interconectados, lo cierto es que sus usuarios no tienen normalmente conciencia de ello, limitándose a comunicarse con él por medio de lenguajes formales cuya estructura está más cerca del lenguaje matemático (por ejemplo) que del lenguaje del autómata.

Si bien es una práctica muy cuestionada y en retroceso a partir de los últimos años de la década de los setenta, ha sido muy popular describir el algoritmo en un principio (fase 3, figura 1.1., capítulo 1) mediante un diagrama de flujo, organigrama u ordinograma, que de todas estas maneras se viene designando. Desde la época de los primeros programadores, Condesa de Lovelace, Adela Goldstine, Grace Hopper, John von Neumann, etc., se han utilizado los grafos para expresar el flujo de desarrollo de cálculos.

Sin embargo, hasta hace muy poco la utilización de estos grafos carecía de un soporte riguroso.

3.1. Función, programa, función de programa

Una *función* es un conjunto de pares ordenados (a, b) tales que $(a, b') \in f \wedge (a, b'') \in f \Rightarrow b' = b''$. Si $(a, b) \in f$, se puede escribir $b = f(a)$, siendo a un argumento, b un valor de f. El conjunto de todos los argumentos es el dominio de f y el de todos los valores es la imagen de f. Damos ahora una definición de «programa» alternativa con respecto a la que acaba de estudiarse en el apartado 2 y que nos parece más conveniente a los efectos que siguen.

Un *programa* es un conjunto finito de funciones, llamadas instrucciones: cada instrucción opera sobre un dominio finito, contenido en un conjunto común D llamado espacio de los datos. Estas funciones toman valores en $Q = D \times P$, que podemos llamar espacio de estados. Así, una ejecución de programa es una secuencia de estados,

$q_i = (d_i, f_i)$, $i = 0, 1, \ldots$ tal que $q_{i+1} = (f_i(d_i))$ para $i = 0, 1, \ldots$
q_0 = Es el valor inicial de la ejecución.
q_n = Es el valor final de la ejecución, si la secuencia es finita.

Se llama función de programa $[P]$ a:

$[P] = \{(q_0, q_n), q_n$ es el valor final de P si el valor inicial es $q_0\}$.

El programa es, pues, una regla específica, pero no única para calcular la función $[P]$.

3.2. Normalización de organigramas

Vamos a formalizar de manera mejor y más completa la idea de grafo orientado, asociado a un programa, que ya se había presentado en el apartado 4 del primer capítulo.

Un programa P puede representarse por un grafo $G = (P, U)$ en que el conjunto de los nodos es el conjunto de instrucciones y donde el conjunto U de los arcos está definido por:

$$U = \{(f, g) \in P \times P, \exists d \in \text{dominio}(f), \exists d' \in D, f(d) = (d', g)\}$$

o, lo que es lo mismo, hay un arco de f a g si la instrucción g puede ejecutarse inmediatamente después de la instrucción f.

Este grafo puede adoptar, en último extremo, la forma de un *grafo orientado con tres tipos de nodos* (figura 3.1):

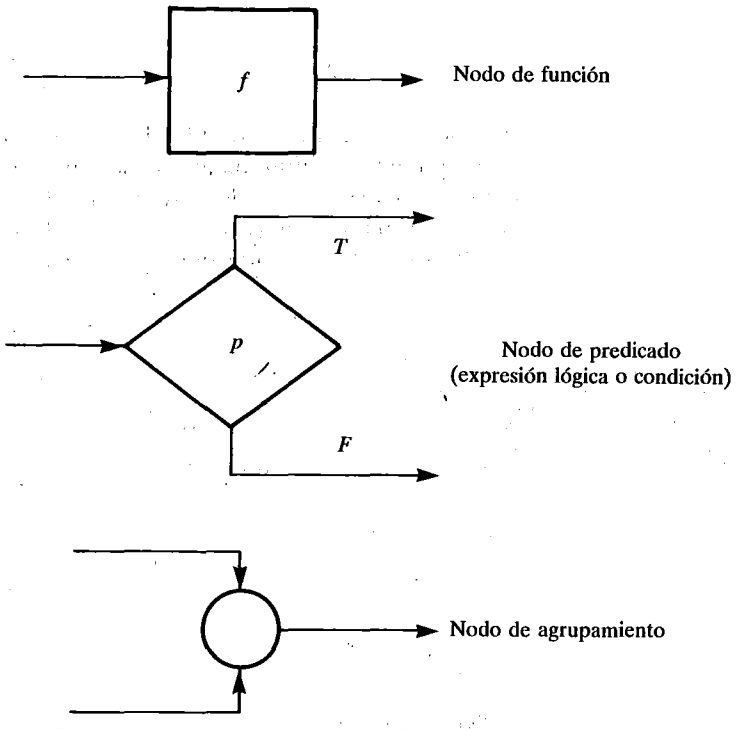


FIGURA 3.1.

El nodo de función, en estas condiciones, conduce siempre a la misma instrucción siguiente, por lo que puede considerarse que la aplicación de su dominio no lo es en el espacio $Q = D \times P$, sino solamente en el espacio D.

El nodo de predicado tiene dos arcos de salida y es una aplicación de su dominio en {True, False} ({cierto, falso}). (El arco de arriba o de derecha, esto último si se traza verticalmente, corresponde a True, y el de abajo o izquierda corresponde a False). (*Nota*: a veces se cambia un lado por otro, en la práctica).

Un nodo de agrupamiento simplemente transfiere control de sus dos líneas de entrada a la de salida.

Definamos un *programa limpio* como aquel cuyo grafo o diagrama posee un solo arco de entrada y un solo arco de salida, existiendo un camino de entrada hasta cualquier nodo y de cualquier nodo hasta la salida.

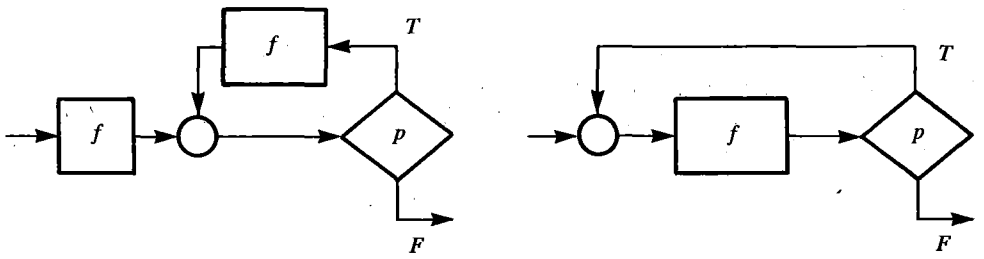


FIGURA 3.2.

Dos programas limpios pueden ser equivalentes si definen los mismos cálculos o si definen la misma función, aún cuando tengan distinto diagrama de flujo. Por ejemplo, los programas de la figura 3.2 definen los mismos cálculos y las mismas funciones. Los programas de la figura 3.3 definen la misma función (cuando los arcos de salida de p y q son ambos T, entonces f en ambos grafos) pero diferentes cálculos (en los demás casos).

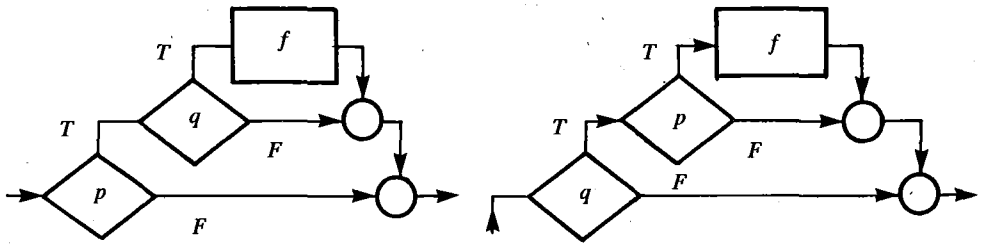


FIGURA 3.3.

Por tanto, pueden definirse distintos niveles de equivalencia, entre diagramas de flujo, unos que preservan cálculos, otros que preservan función, etc. Varios autores

han estudiado la potencia de diversas clases de diagramas en la definición de cálculos y funciones. El principal resultado de sus estudios es *que clases relativamente pequeñas y económicas de diagramas de flujo pueden definir los cálculos y funciones de la clase de todos los diagramas de flujo, posiblemente a costa de cálculos extra fuera de la descripción original del conjunto de estados.*

Se define una clase de diagramas *BJ* (por Böhm y Jacopini) sobre un conjunto de funciones $F = \{f_1, \ldots, f_m\}$ y sobre un conjunto de predicados $PR = \{P_1, \ldots, P_n\}$ en la forma del cuadro de la figura 3.4.

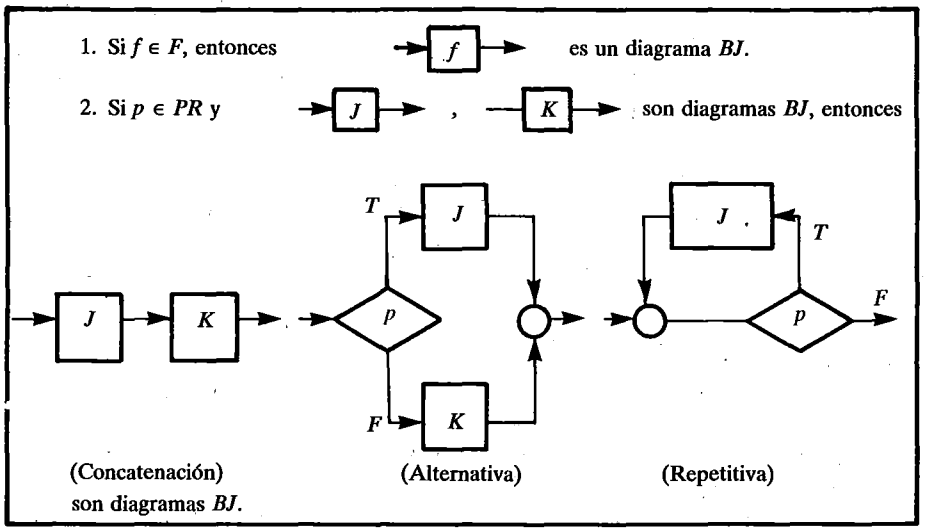


FIGURA 3.4.

La definición que se acaba de dar es recurrente y permite construir cualquier diagrama de flujo de programas limpios como un diagrama *BJ*. Esta es una forma de normalizar organigramas o diagramas de flujo.

4. ¿QUÉ ES UN PROGRAMA ESTRUCTURADO?

4.1. Diagramas o estructuras básicas, fórmulas o esquemas de programa

Los que hemos llamado diagramas *BJ* en el apartado 3.2 —algunos los llaman diagramas *D* (por Dijkstra, que es uno de los autores más reconocidos)— son estructuras básicas, porque se ha demostrado que con ellos es posible construir el grafo de cualquier programa. A semejanza de lo que ocurre con cualquier expresión lógica y los operadores AND, OR y NOT. A veces, es práctico utilizar dos estructuras más que resultan de una degradación o una reconstitución de las tres anteriores. En definitiva, puede contarse con las cinco estructuras del cuadro de la figura 3.5, en el que las estructuras 4 y 5 son derivadas de las tres primeras. A menudo resulta cómodo

designar estos grafos por una expresión o fórmula con un nombre. Los pioneros Böhm y Jacopini les llamaron π, Δ, Ω, Λ y Φ, respectivamente, por el orden en que los situamos en la figura 3.5. En la actualidad es más corriente denominarlos por una expresión que se corresponde de cerca con algunos de los lenguajes de alto nivel que describen tales estructuras. Concretamente, en la columna de la derecha del mismo cuadro pueden verse las expresiones en Pascal que corresponden a las cinco fórmulas.

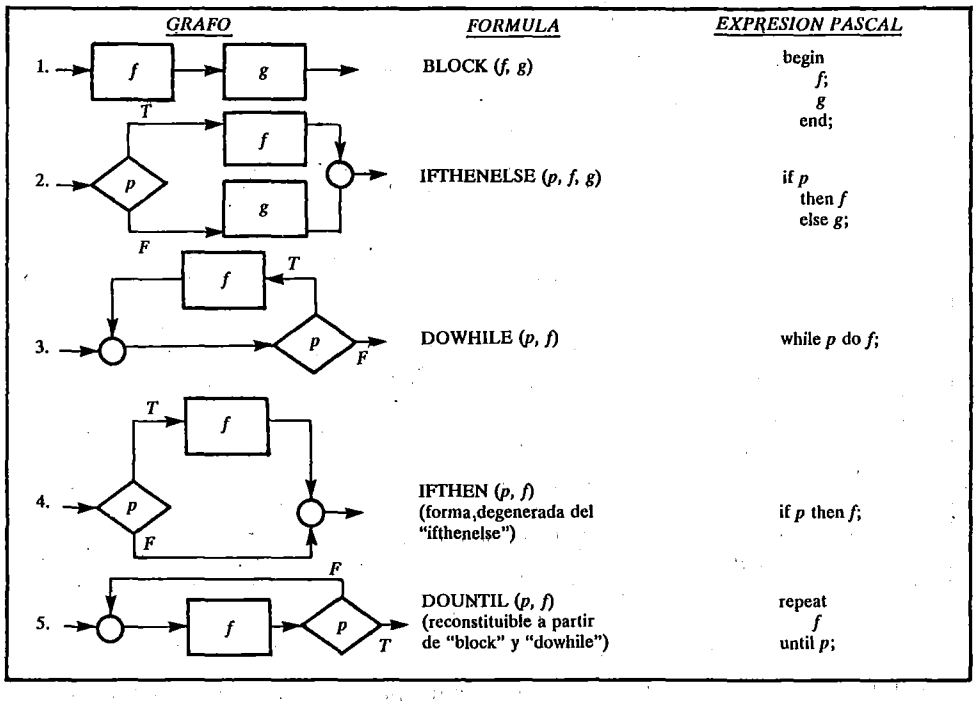


FIGURA 3.5.

4.2. Definición

Se dice que un programa es estructurado si se expresa únicamente por medio de los diagramas del cuadro anterior. De manera más general, *un programa es estructurado si es una fórmula compuesta en cualquier conjunto prescrito de fórmulas básicas de programa.*

De manera más concreta, un programa es estructurado (en el conjunto definido en el subapartado anterior) si tiene una de estas formas:

$$P = \text{BLOCK } (f, g)$$
$$P = \text{IFTHENELSE } (p, f, g)$$
$$P = \text{IFTHEN } (p, f)$$
$$P = \text{DOWHILE } (p, f)$$
$$P = \text{DOUNTIL } (p, f)$$

siendo p un predicado de P y f, g:

a) Función identidad.
b) Cálculos de P.
c) Subprogramas limpios de P, también estructurados.

El grafo de la figura 3.6 es estructurado puesto que puede escribirse:

$$P = \text{IFTHENELSE } (p, \text{ DOWHILE } (q, f), \text{ BLOCK } (g, h))$$

Codificándolo en Pascal, resultaría así:

```
if p then
    while q do f
  else
    begin
      g;
      h
    end;
```

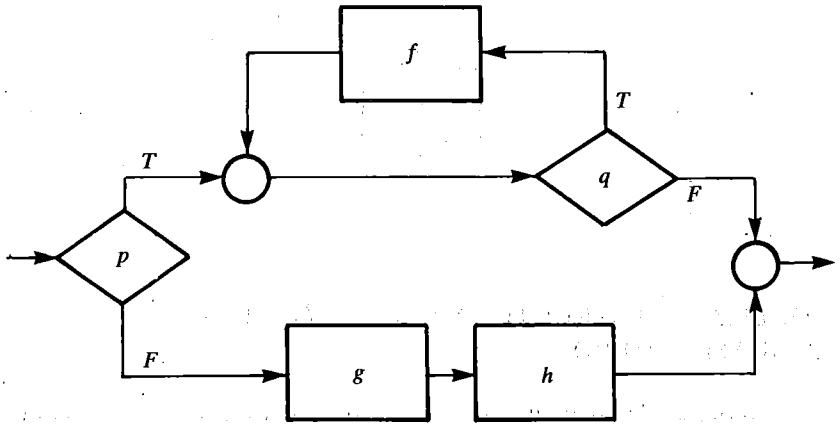


FIGURA 3.6.

El grafo de la figura 3.7 es estructurado y puede escribirse así:

$$P = \text{BLOCK } (f_1, \text{ DOWHILE } (\neg p, f_2), f_3)$$

y en Pascal:

```
begin
    f1;
    while not p do f2;
    f3
  end;
```

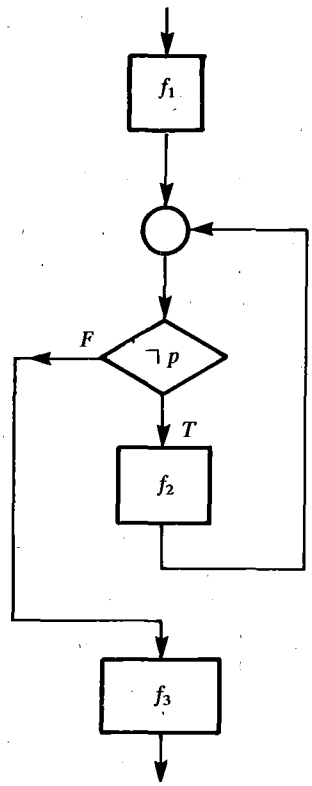


FIGURA 3.7.

5. TEOREMA DE ESTRUCTURA REFERIDO A UN PROGRAMA LIMPIO

Existe un teorema de estructura que demuestra la existencia de un diagrama *BJ* en conjuntos ampliados de funciones *F* y de predicados *PR*, que simula los cálculos de cualquier diagrama considerado.

5.1. Funciones estándar y teorema

Para el teorema que vamos a enunciar ahora sin demostración se exige el uso de las siguientes cuatro funciones estándar: TRUE, FALSE, POP y TOP. Aquí se presentarán estas cuatro funciones, el enunciado del teorema, una aplicación del mismo y un corolario.

TRUE : Añade el valor «true» (cierto) al conjunto de los datos que corresponden a su arco de entrada. Si $a \in A$, siendo A el conjunto de datos, TRUE $(a) = (a, \text{true})$.

FALSE: Añade el valor «false» (falso) a su dominio. FALSE (a) = (a, false).
POP : Se utiliza cuando una de las funciones TRUE o FALSE ha añadido una variable booleana al conjunto de los datos y su acción consiste en borrar esta variable. Si b es una variable booleana auxiliar, POP (a, b) = a.
TOP : Se utiliza cuando una de las funciones TRUE o FALSE ha añadido una variable booleana auxiliar al conjunto de los datos y su acción consiste en conservar el valor de esta variable. TOP (a, b) = (a, b). Es un predicado (b es la variable booleana).

Dice así el teorema: todo *programa limpio* es equivalente a un programa estructurado que contiene como máximo las fórmulas BLOCK, IFTHENELSE y DOWHILE (y/o DOUNTIL), las funciones estándar TRUE, FALSE, POP y TOP, así como las funciones y predicados del programa original.

Las funciones estándar hacen las veces de señalizadores para incorporar las estructuras básicas y guardar traza de la procedencia de las variables en el flujo de funciones.

5.2. Ejemplo de aplicación del teorema de estructura con funciones estándar

Se tiene el organigrama de la figura 3.8.

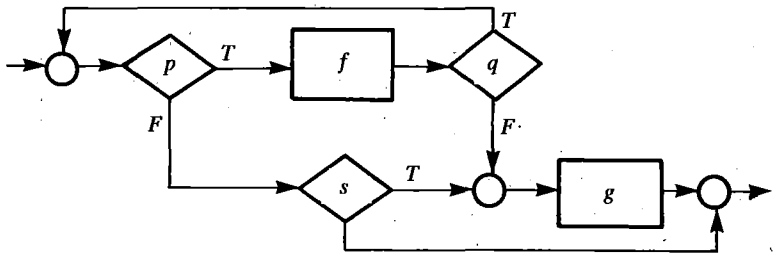


FIGURA 3.8.

El organigrama de la figura 3.9 es equivalente al de la figura 3.8. Al comenzar la ejecución, se realiza la secuencia (TRUE; TOP; POP; p), por lo que la función p tiene como dominio el conjunto de datos de partida, igual que en el programa original. En el ramal del grafo donde se encuentra f se marca uno de los arcos con TRUE y otro con FALSE para poder encaminar de nuevo hacia la función p o hacia la salida, respectivamente. El ramal inferior, el de la secuencia (s; g) se marca con FALSE para poder encaminar el tratamiento hacia la salida por medio del mecanismo TOP.

El organigrama de la figura 3.9 es una estructura cuya fórmula puede escribirse de esta manera:

BLOCK (TRUE, DOWHILE (TOP, BLOCK (POP, IFTHENELSE
(p, BLOCK (f, IFTHENELSE (q, TRUE, BLOCK (g, FALSE))),
BLOCK (IFTHEN (s, g), FALSE)))), POP).

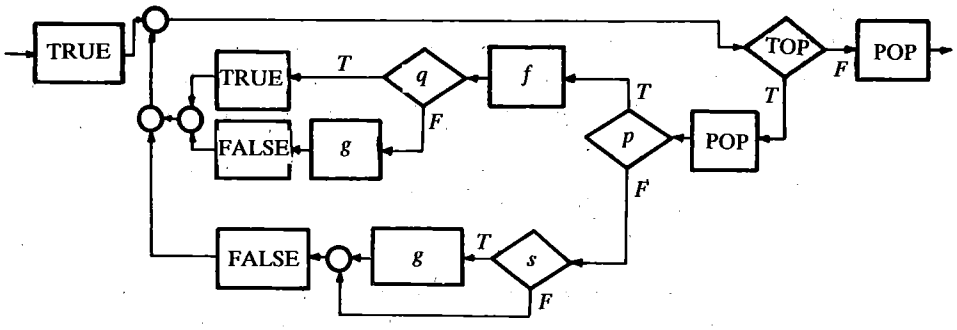


FIGURA 3.9.

El organigrama de la figura 3.10 es también equivalente al de la figura 3.8, aunque en esta ocasión se ha pivotado sobre la estructura DOUNTIL.

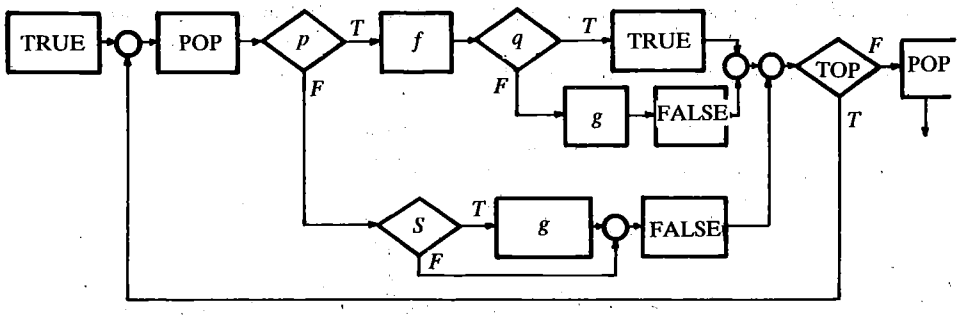


FIGURA 3.10.

5.3. Corolario

Todo programa limpio es equivalente a un programa en una de las formas:

BLOCK (*f*, *g*)
IFTHENELSE (*p*, *f*, *g*)
DOWHILE (*p*, *f*)
(y/o DOUNTIL (*p*, *f*))

donde *p* es un predicado del programa original o TOP, y donde *f* y *g* son programas limpios, cálculos del programa original, TRUE, FALSE o POP.

6. INTERÉS Y APLICABILIDAD DE LA PROGRAMACIÓN ESTRUCTURADA

La programación estructurada es un movimiento de la década de los setenta. Coincide su lanzamiento y primera difusión con una etapa de análisis crítico de los problemas de la programación, relacionados primordialmente con la *falta de fiabilidad de los programas*, es decir, del software en general, especialmente de los programas complejos. El software de sistema (sistema operativo, compiladores, etc...), por ejemplo, es un conjunto de programas muy complejos, y sus fallos conducen a consecuencias cuando menos desagradables y en muchos casos catastróficas.

La velocidad de expansión de la programación estructurada está condicionada al hecho histórico del momento de su aparición, cuando ya un cierto número de hábitos y técnicas ha adquirido volumen. Hoy día, ya va siendo por suerte menos frecuente que una persona formada en los principios de la P.E. y convencida de su utilidad tenga que luchar contra el entorno.

El organigrama de la figura 3.11 no emplea los nodos del apartado 3.2, no es un programa limpio y muchos menos estructurado, aunque pueda ser desde otro punto de vista un programa óptimo. En pocas palabras, no utiliza nada de lo que se ha estudiado en este capítulo. Representa un algoritmo de asignación de particiones en memoria, que ilustraba un conocido libro sobre sistemas operativos del año 1974.

Dirigidas al lector poco familiarizado con este concepto, haremos en las próximas páginas unas anotaciones muy sumarias con el fin de orientarle en cuanto al interés y aplicabilidad de la P.E., aunque globalmente es posible confirmar los efectos positivos de la P.E. y técnicas subsiguientes sobre la calidad del software.

6.1. Comunicabilidad de la programación estructurada

La comunicabilidad es una faceta dentro de la condición más general de *inteligibilidad*.

Este es un aspecto que nadie puede discutir. *La P.E. produce programas claros, expresados con un mínimo de fórmulas o diagramas, programas limpios.* La consecuencia práctica inmediata es una disminución de los costes de programación relacionados con la tarea de modificación de programas y una potenciación del trabajo en equipo.

Corolario de lo anterior es el favorecimiento del *trabajo en equipo*, del que la técnica de organización del equipo estructurado de programación (Chief Programmer Team, CPT en la terminología profesional), que utiliza diseño modular y programación estructurada, es un conocido ejemplo.

Otra faceta igualmente derivable es la *didáctica*. La P.E. puede enseñarse bien ya que, entre otras cosas, permite discutir y comparar programas. Puede decirse, sin temor a errar, que la aparición de la P.E. ha supuesto un impacto incalculable en la mejora de la enseñanza de la programación. Y, sin duda, en la propia evolución de la disciplina, científicamente hablando.

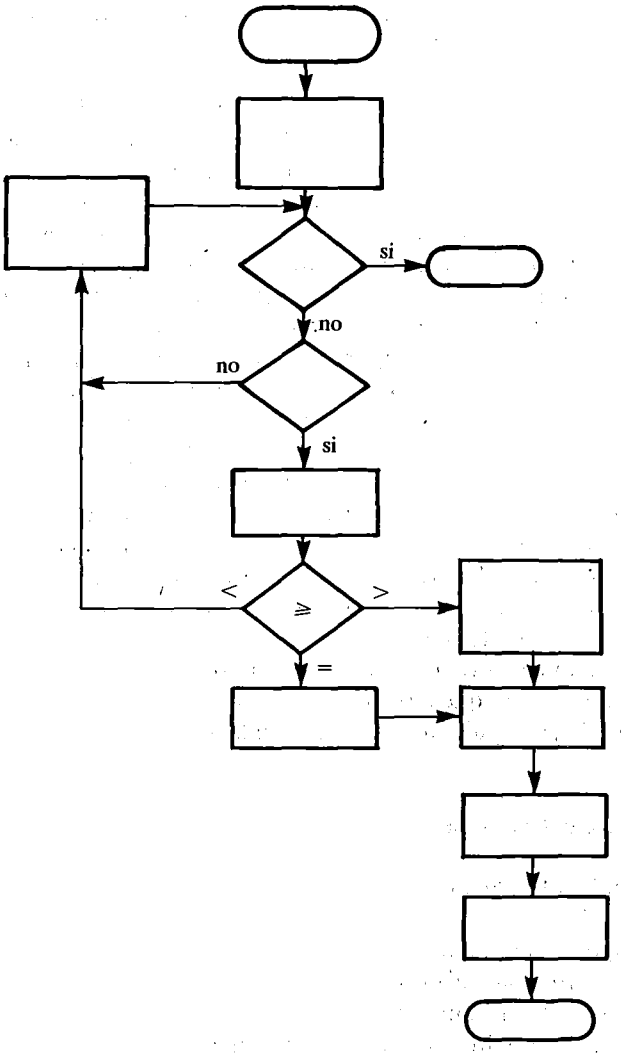


FIGURA 3.11.

6.2. Corrección de los programas

Tal vez sea *la posibilidad de demostrar formalmente si un programa es o no correcto el aspecto más importante de la programación estructurada.*

Efectivamente, si la fórmula de un programa emplea solamente los diagramas de la figura 3.5, puede probarse si aquella es o no es correcta mediante un censo de todos

los nodos del grafo. En cada nodo debe aportarse la prueba local de la descomposición. Por ejemplo, si f = BLOCK (g, h), demostrar que

$$[f] = \{(r, t), \exists s(r, s) \in [g] \wedge (s, t) \in [h]\}$$

Dejamos sugerida simplemente esta cuestión, de gran complejidad teórica y fuera por completo de los objetivos de estas páginas.

6.3. Los lenguajes de programación y la programación estructurada

Los lenguajes todavía más utilizados se han definido ajenos por completo, y en general anteriormente, a la programación estructurada. Hubiera sido una casualidad que su estructura reflejara precisamente las estructuras de la P.E. Y, en efecto, no la reflejan. Este ha sido el obstáculo objetivamente más considerable a la difusión de la P.E. Como se ve, es una consecuencia en la creación de dos tipos de lenguajes: el lenguaje gráfico formalizado (lenguaje de la P.E.) y el lenguaje formal lineal.

De los lenguajes formales existentes unos se adaptan mejor que otros a la P.E., pero de manera genérica hay que aceptar ciertos retoques en la manera de utilizarlos, retoques que generan habitualmente una pérdida de eficacia en su uso. A título de ejemplo, el lector puede hacerse una idea (figura 3.12) de la clase de manipulaciones que habría que emplear en el uso de las sentencias clásicas de Fortran para reflejar con fidelidad diagramas estructurados. Se pierde parte de la potencia del DO y sentencias tales como las de IF aritmético, IF lógico, GOTO calculado y GOTO asignado, por poner algunos ejemplos, sufrirían también serias erosiones al describirse diagramas estructurados mediante FORTRAN. De una u otra forma, se han diseñado versiones estructuradas de los lenguajes más comunes, como Fortran y Basic.

El Pascal es lenguaje definido a imagen y semejanza de la programación estructurada.

Quede dicho aquí, a modo de nota final, que los progresos en materia de lenguajes han sido gigantescos y se extienden más allá de la programación estructurada, la que prácticamente sólo atiende al control de la secuencia de pasos. Los lenguajes modernos integran en su estructura la estructura de los datos.

7. RESUMEN

Un programa es, mediando ciertas condiciones (véase apartado 1), *una representación de un algoritmo.* Análogamente, *un organigrama (ordinograma o diagrama de flujo) es un grafo representativo de un programa.*

Se ha expuesto cómo *un organigrama puede trazarse con sólo tres tipos de nodos (de función, de predicado y de agrupamiento).* A su vez, *una sola clase de diagramas compuesta por tres diagramas simples (concatenación, alternativa, repetitiva) bastaría para definir los cálculos de la clase de todos los diagramas.* Estos diagramas simples poseen la característica de tener *una entrada única y una salida única.*

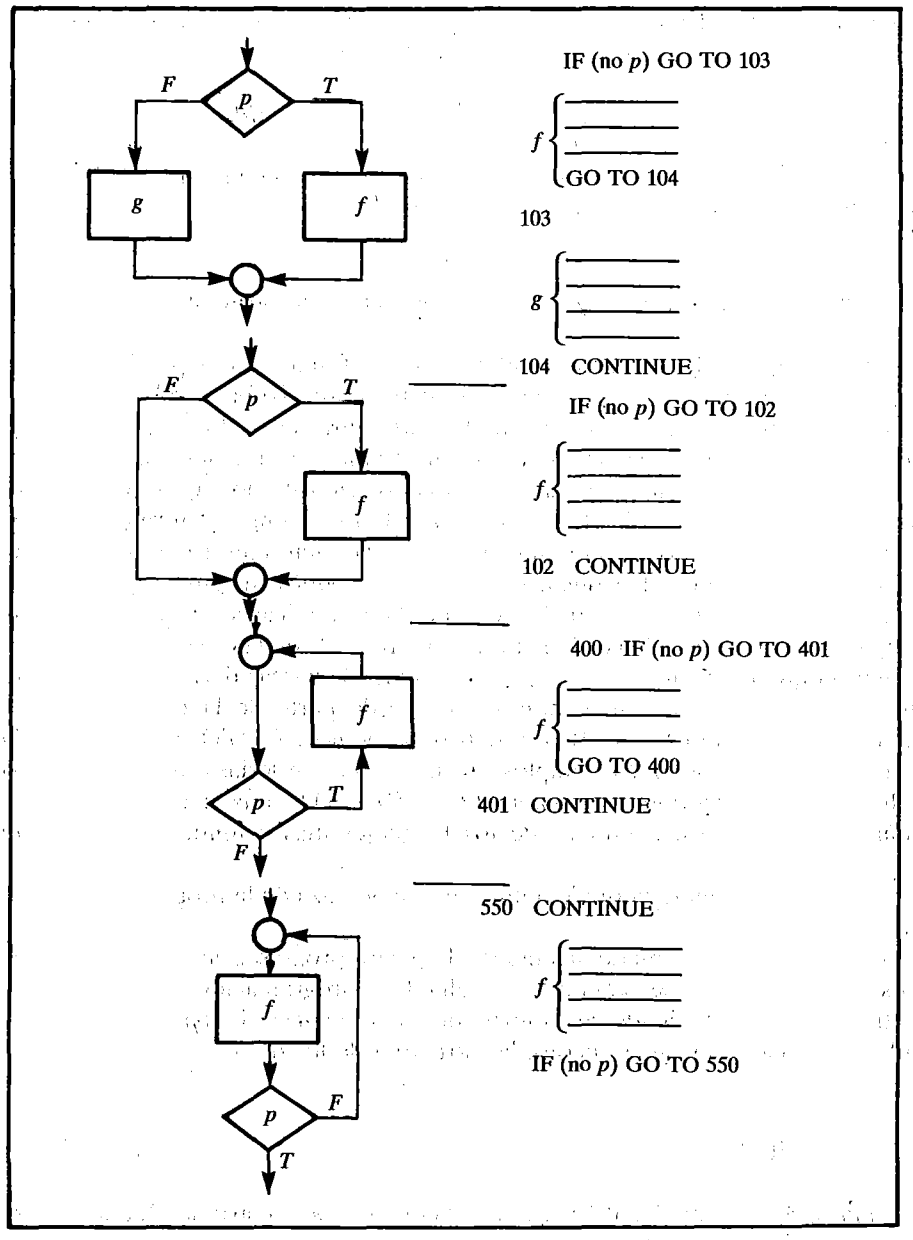


FIGURA 3.12.

Se define como estructurado un programa cuando está formado exclusivamente con los diagramas o fórmulas de un determinado conjunto, como por ejemplo, el constituido por {fórmula de concatenación, fórmula alternativa, fórmula repetitiva}. Natural-

mente, el organigrama de un programa estructurado se expresa mediante una de estas tres fórmulas.

Si un programa es limpio pero no estructurado, es factible llegar a un programa estructurado, sin distorsionar excesivamente la estructura original, introduciendo unas funciones concretas (TRUE, FALSE, POP, TOP), que amplían el conjunto de estados mediante la introducción de unas variables lógicas binarias.

La programación estructurada presenta grados diversos de interés y aplicabilidad, tema hace pocos años muy discutido en el sector profesional de la informática. El apartado 6 recoge algunos rasgos de esta discusión, resaltando la virtualidad teórica de poderse demostrar formalmente si un programa es o no es correcto. Debe observar el lector que actualmente se prueba si los programas funcionan o no con un juego más o menos significativo del espacio de los datos, pero no se demuestra si son o no son correctos*.

8. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

Fueron Böhm y Jacopini quienes, en 1966, pusieron las primeras piedras de la teoría de la programación estructurada, partiendo de un conjunto de bloques o diagramas de flujo entendido como un lenguaje de programación. Ha habido grandes autores e investigadores, que se han caracterizado por sus aportaciones teóricas en este terreno, entre quienes cabe citar a Dijkstra, Hoare, Wirth y Mills (Mills, 1975). Bibliográficamente, es especialmente notable el clásico libro de Dahl, Dijkstra y Hoare (1972). Otras obras muy interesantes, entre las muchas docenas que se han escrito, corresponden a Manna (1974), Dijkstra (1976) y Alagic y Arbib (1978).

Desde un punto de vista más pragmático —quiere decirse más al alcance de los programadores profesionales—, es reseñable el impacto causado a finales de 1973 (como se ve, con un cierto retraso) por un número de la revista «Datamation», que dedicó varios artículos a este tema, con el título genérico de «Revolución en la Programación» (*Datamation*, 1973).

No se pretende que todas las referencias que acaban de hacerse constituyan siquiera un conjunto mínimo básico en el campo de la programación. Este ha adquirido tal volumen, ramificaciones e importancia que sólo estudiando una obra colectiva preparada por distintos especialistas puede uno llegar a hacerse verdaderamente una idea de conjunto. A este fin merece citarse un estudio panorámico reciente en español, editado por Gamella (Gamella, 1985).

En lo concerniente a este capítulo, los ejemplos de equivalencia entre programas limpios del subapartado 3.2 se deben a Mills (1975), y este matemático fue también quien encabezó el primer y famoso experimento de equipo estructurado de programación, al que aludíamos en el subapartado 6.1.

El teorema de estructura del subapartado 5.1 se tomó de Tabourier *et al.* (1975), referencia utilizada de manera general en varios pasajes del capítulo.

---

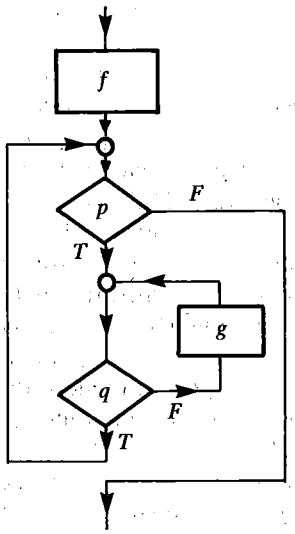
* Hay una «ley» (Gilb, 1975) en informática práctica que dice: *Todos los programas reales contienen errores mientras no se pruebe lo contrario, lo cual es imposible.*

En el libro de Madnick y Donovan (1974, p. 119), se encontrará el ordinograma de la figura 3.11. Y, para terminar, citemos a Tenny (1974), como un estudio, entre muchos otros, sobre las manipulaciones a que habría que someter las sentencias de Fortran no estructurado para reflejar diagramas estructurados.

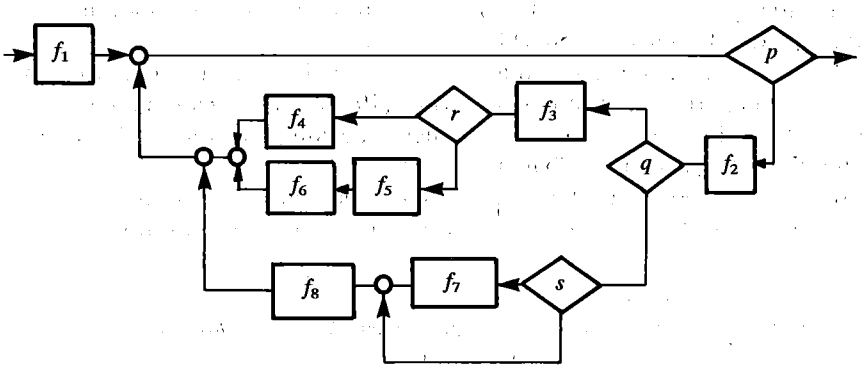
9. EJERCICIOS

9.1. Dado el siguiente grafo, indicar cuál de las siguientes opciones se corresponde con la fórmula de programación estructurada que define los mismos cálculos.

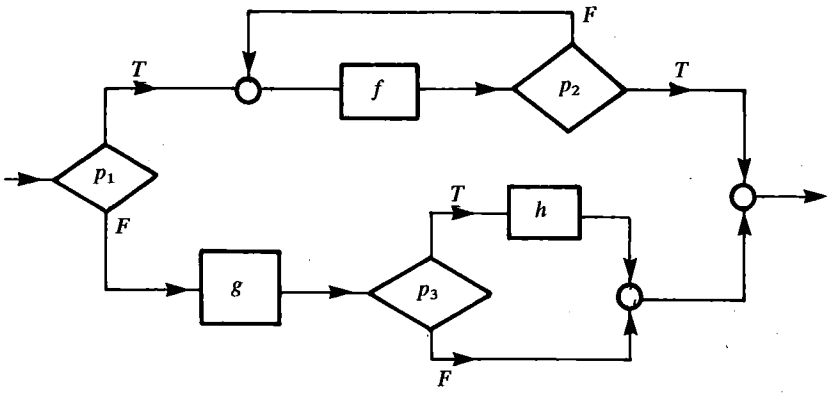
a) BLOCK (f, DOWHILE (p, $\neg q$, g))
b) BLOCK (f, IFTHEN (p, DOUNTIL (q, g)))
c) BLOCK (f, DOWHILE (p, DOUNTIL (q, g))
d) BLOCK (f, IFTHEN (p, IFTHEN $\neg q$, g))



9.2. Obtener la fórmula del siguiente programa estructurado.



9.3. Obtener la fórmula correspondiente al siguiente grafo estructurado:



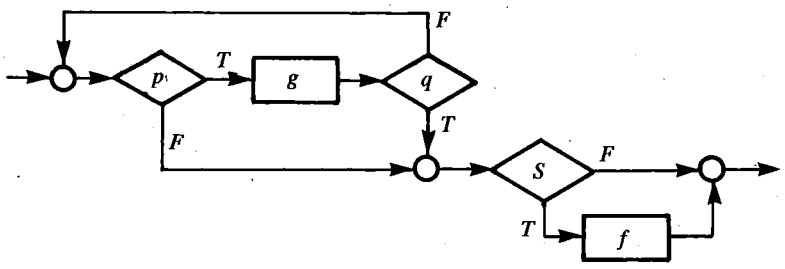
9.4. Obtener el organigrama del siguiente programa estructurado:

DOWHILE (p_1, IFTHENELSE (p_2, IFTHEN (p_3, BLOCK (DOUNTIL (p_4, DOWHILE (p_5, f_1)), IFTHENELSE (p_6, f_2, f_3)))))

9.5. Obtener el diagrama de flujos del programa estructurado siguiente:

BLOCK (f_1, DOWHILE (p_1, IFTHENELSE (p_2, f_2, IFTHENELSE (p_3, IFTHENELSE (p_4, f_3, f_4), IFTHENELSE (p_5, f_5, IFTHEN (p_6, f_6))))))

9.6. Transformar el siguiente organigrama en un organigrama estructurado utilizando el método de las cuatro funciones estándar TRUE, FALSE, POP y TOP. *Nota*: pivótese básicamente sobre la estructura DOUNTIL.



Escríbase la fórmula de la estructura resultante.

Capítulo 4
PROGRAMACION ESTRUCTURADA: CONCEPTOS METODOLOGICOS

1. INTRODUCCIÓN

En este capítulo nos proponemos describir, a través del desarrollo de un ejemplo muy sencillo, cómo se diseña un programa de nueva planta en forma estructurada. El método general consiste en manejar constantemente tres principios: a) el *razonamiento será deductivo*, de lo más general a lo más particular; b) a cada nivel de razonamiento se hará uso de los *recursos abstractos* necesarios; c) el razonamiento será guiado por la decisión de utilizar siempre *estructuras básicas* en la expresión gráfica de dicho razonamiento.

2. MÉTODO GENERAL DE DISEÑO DE PROGRAMAS ESTRUCTURADOS

Sobre las estructuras básicas está casi todo dicho en el capítulo anterior, pero llamamos la atención del lector para que recuerde que la estructura completa de un programa estructurado tiene la forma de una estructura básica. De ahí partirá el método. Tras el análisis del problema se expresa un procedimiento de solución del mismo mediante un esquema estructurado muy general. Sin romper esta estructura, sino profundizando en su interior, se irá refinando el esquema mediante niveles o pasos sucesivos («stepwise refinement») hasta llegar a un nivel en que ya no interesa refinar más. Veamos qué son el *recurso abstracto* y el *razonamiento deductivo*.

2.1. El recurso abstracto

Una de las circunstancias que más dificulta la concepción de programas reside en mantener simultáneamente «in mente» las especificaciones del programa, por un

lado, y por otro lado, los recursos concretos de que se dispone para codificarlas, es decir, las instrucciones del lenguaje de programación que se empleará. La tendencia natural consiste en plasmar directamente las especificaciones en términos de instrucciones. La experiencia demuestra que es realmente arduo luchar contra esta tendencia.

Para salvar la brecha existente entre el dominio de las ideas tal como ocurren en la mente humana y el dominio de los procesos reflejados por las instrucciones de un lenguaje de programación, el diseño o concepción de programas estructurados se auxilia de lo que se denomina «recursos abstractos», por contraposición a los recursos concretos de que se dispone (un ordenador con un determinado lenguaje).

Según Dijkstra, concebir un programa en términos de recursos abstractos consiste en descomponer una determinada acción compleja en términos de un número de acciones más simples, que podrían ser interpretadas como instrucciones para una supermáquina (inexistente) capaz de ejecutarlas. Puesto que nuestros recursos concretos son incapaces de procesar tales instrucciones concebidas para una máquina, el paso siguiente será, entonces, crear un programa que simule su funcionamiento, o, lo que es lo mismo, descomponer cada una de sus instrucciones en acciones todavía más simples que podrían ser a su vez instrucciones para otra máquina ideal que plantearía los mismos problemas. El proceso continuaría hasta que en un determinado nivel de descomposición, las subacciones obtenidas constituyan instrucciones para el ordenador actualmente disponible*.

Esta manera de concebir programas no es exclusiva de la programación estructurada. La programación modular la utiliza, no sistemáticamente, con otros fines. El programador experto que divide el programa en módulos hace algo parecido; los módulos serán codificados por distintos programadores, pero para él son como instrucciones de un lenguaje más potente que le permiten establecer más cómodamente el programa. Su problema, justamente, es establecer qué módulos son los más adecuados, qué condiciones deben satisfacer y cómo deben ser encadenados. En realidad, lo que establece son las especificaciones de la supermáquina que fuera capaz de procesar tales módulos, considerados como instrucciones. El resto de los programadores lo que hacen es simular la supermáquina con los recursos concretos disponibles.

Un aspecto que no debe ser olvidado en programación estructurada es el poseer unas estructuras, las cuales son válidas para todas las supermáquinas; es decir, cuando una acción compleja se descompone en acciones más simples, la lógica con que se establecen tales acciones debe de corresponder con alguna de las estructuras. Este es el principio c) que se señalaba en la introducción a este capítulo.

2.2. Razonamiento deductivo (diseño descendente)

Con este nombre se designa al proceso mental que permite concebir un programa por medio de una marcha analítica que se refleja en niveles o pasos consecutivos de

* En el capítulo 5 se verá hasta qué punto habría que llevar este proceso cuando el recurso concreto fuera una máquina de Turing.

refinamiento, cada uno de ellos poseyendo sus propios recursos abstractos que permiten resolver el programa por completo.

Esta marcha analítica, totalmente cartesiana, supone que, situados en un determinado nivel de razonamiento, todos los niveles anteriores y cada uno de ellos han resuelto totalmente el problema; de lo que se trata es de hacer un refinamiento que permita expresar esa solución de una manera más adecuada a ciertas limitaciones existentes (los recursos reales). Es por esto que *nunca será necesario volver atrás** para reconsiderar ciertos aspectos no tenidos en cuenta, salvo error o que se quiera intentar otra solución más conveniente.

El paso de un nivel al siguiente se hace por medio de un cambio en el «punto de vista», lo que permite introducir un refinamiento. Esto puede verse más claro reflexionando sobre cómo es una estructura, habida cuenta de que sólo tiene una entrada y una salida.

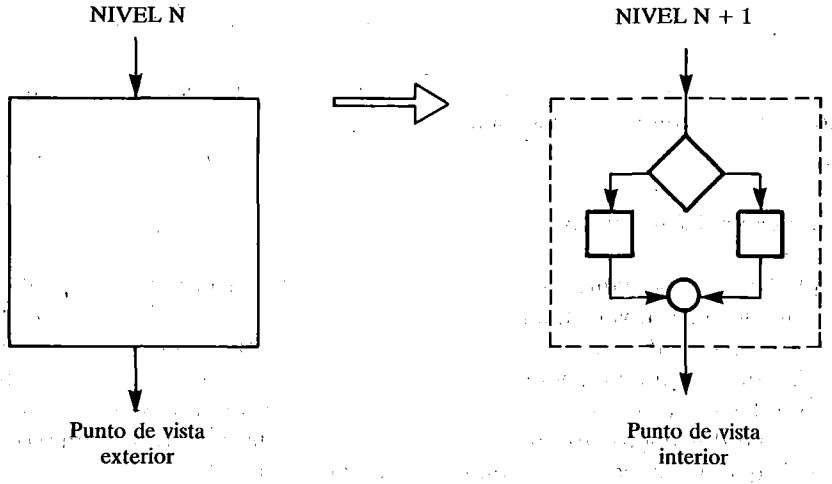


FIGURA 4.1.

Una estructura básica nos permite tomar dos puntos de vista con respecto a ella.

a) Punto de vista «exterior». En este caso la estructura es vista como una caja «negra» con una única entrada y una única salida por donde circula una información que la estructura manipula de alguna manera para entregar unos resultados en su salida. El aspecto fundamental a considerar es que la estructura *hace* algo, sin considerar *cómo lo hace*.
b) Punto de vista «interior». Ahora la posición es totalmente inversa. Dada una «caja negra» determinada que aporta una determinada acción, en un momento determinado debemos preguntarnos «de qué manera es hecha», si encadenan-

* En la práctica real, no se es capaz de realizar una marcha perfecta, sin vuelta atrás, pero éste es un principio que es importante seguir con el máximo grado de pureza que sea posible.

do, ofreciendo alternativas o repitiendo acciones simples o complejas. Cuando tomamos esta posición, mirando a *cómo lo hace* realmente, lo que tratamos es de averiguar la estructura interna de la caja negra primitiva.

2.3. Consejo al lector

Al lector que no estuviera anteriormente familiarizado con esta técnica le habrá parecido este apartado tan incomprensible y «abstracto» como el recurso al que se ha hecho referencia. Es recomendable seguir con atención el ejemplo práctico del siguiente apartado, para releer posteriormente éste, que resultará entonces mucho más «concreto».

3. EJEMPLO DE DISEÑO DE UN PROGRAMA ESTRUCTURADO

Se ha escogido un caso práctico que consiste en digitalizar el tiempo y en estructurar la programación de su simulación.

3.1. Enunciado del problema

Se pide simular mediante un programa para ordenador el funcionamiento de un reloj digital que posee dos ventanillas visualizadoras, la primera con tres campos, cada uno de dos dígitos decimales, que representan respectivamente, de izquierda a derecha, la hora, los minutos y los segundos (por ejemplo: 10:22:08, significa las diez horas, veintidos minutos, ocho segundos). La segunda ventanilla visualiza, mediante dos campos de dos dígitos decimales, el mes y el día del mes (por ejemplo: 04:30, quiere decir que nos encontramos en el mes de abril, día 30).

El reloj funciona mediante un circuito integrado, compuesto, entre otros elementos, de tantos registros como campos de dígitos decimales hay en las ventanillas de la esfera del reloj. Dicho en otras palabras, a cada campo le corresponde un registro electrónico donde se crean las representaciones binarias como consecuencia de la recepción de los impulsos procedentes de un oscilador ajustado a un ritmo de segundos. A estos registros y a su simulación en ordenador les llamaremos HORA, MINUTO, SEGUNDO, MES, DIA.

Suponemos que la batería que alimenta el circuito se agota al cabo de N impulsos, siendo N un dato numérico conocido. El programa, pues, leerá los datos N, HORA, MINUTO, SEGUNDO, MES y DIA, siendo estos cinco últimos los correspondientes al momento en que se pone en marcha el reloj. Se introducirá una variable de cuenta de impulsos llamada OSC.

Se pide que el programa escriba sucesivamente «HORA», N.º DE HORA, «MINUTO», N.º DE MINUTO, «SEGUNDO», N.º DE SEGUNDO, «DIA», N.º DE DIA, «MES», N.º DE MES, una línea de seis asteriscos, «HORA», N.º DE HORA, etc. Cuando se acabe la batería, el ordenador imprimirá «SE ACABO LA

BATERIA». Los caracteres entrecomillados denotan aquí textos a imprimir literalmente.

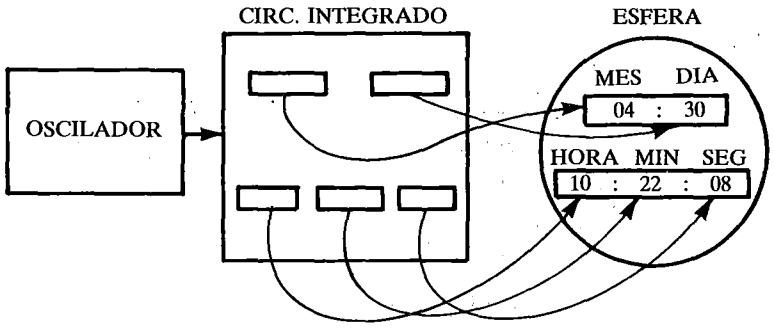


FIGURA 4.2.

Observaciones

Recordamos que febrero *tiene 28 días* (se despreciará la consideración de los años bisiestos), septiembre, abril, junio y noviembre tienen 30. El resto, 31.

Obsérvese que se trata de obtener como salida los estados sucesivos de la esfera del reloj, pero no hay que preocuparse de que la impresora escriba estas salidas a intervalos de tiempo fijo.

3.2. ¿Por qué este ejemplo?

Este es un ejemplo que, en algún sentido, es muy adecuado para los propósitos didácticos que animan este texto. En primer lugar, es sencillo intrínsecamente y, desde nuestra óptica docente, permite poner de manifiesto con claridad, en el espacio de unos 45 minutos que viene a durar una clase, los principios básicos del diseño de programas estructurados. De otra parte, se trata de un problema que plantea una situación familiar a casi cualquier lector y, por tanto, la dificultad de comprensión de su contenido no podrá ser una nube perturbadora que le impida concentrarse en lo esencial del ejemplo, que no es el ejemplo en sí, sino el método que se va a seguir con él.

Sin embargo, en otro sentido, es un ejemplo deficiente. Su misma sencillez animará al lector a creer que puede saltarse pasos del método y, lo que tal vez sea peor, a hacerle pensar que no aparenta ser tan ventajosa la P.E. como se pretende. Ciertamente, los beneficios de la P.E. se aprecian con mayor nitidez enfrentándola a problemas complejos.

3.3. Una solución no estructurada

Aquí sólo interesa estudiar el organigrama que, como se ha repetido, no es único. La codificación en un lenguaje, sea el que sea (ENSAM, FORTRAN, PASCAL, etc.), no entra en nuestros objetivos ahora.

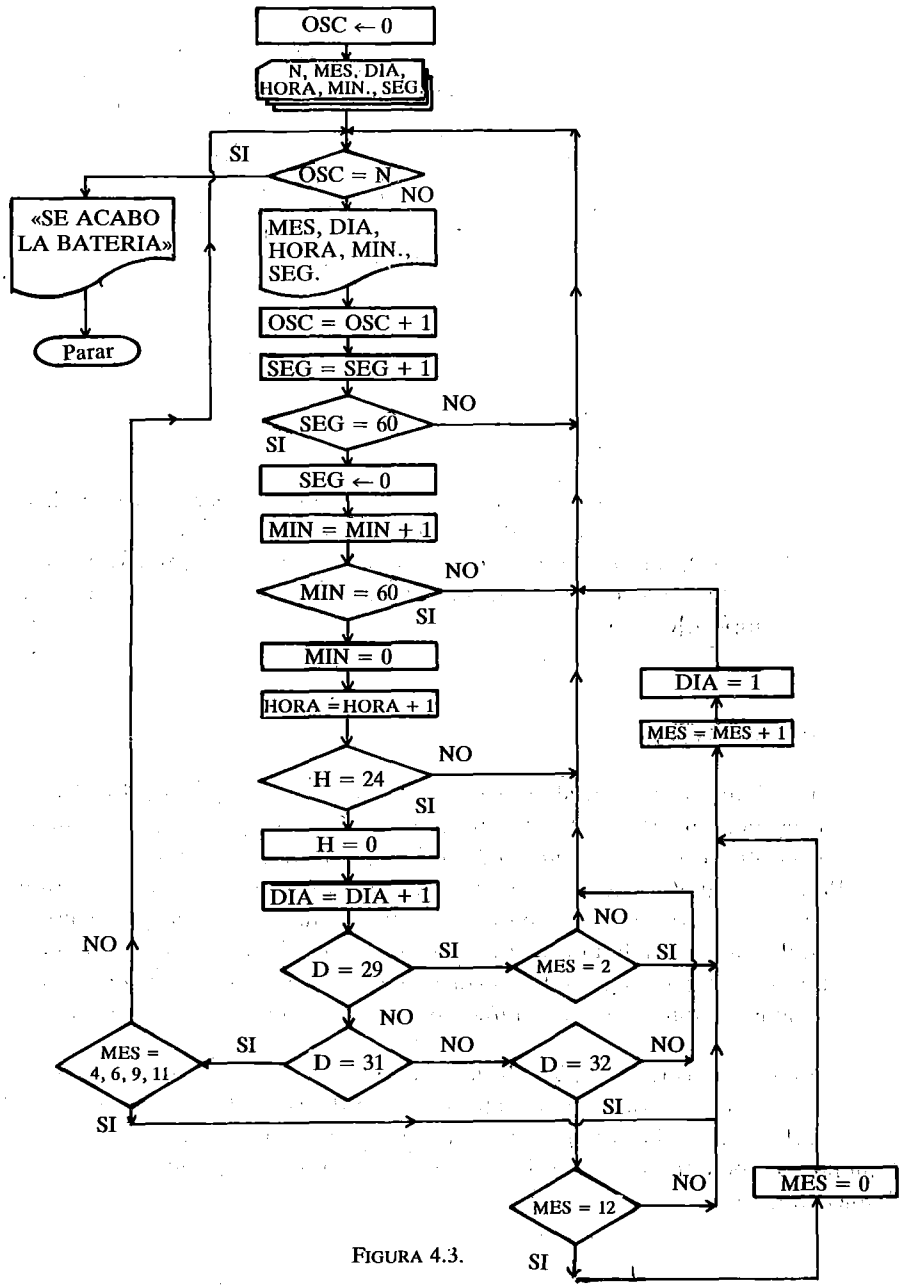


FIGURA 4.3.

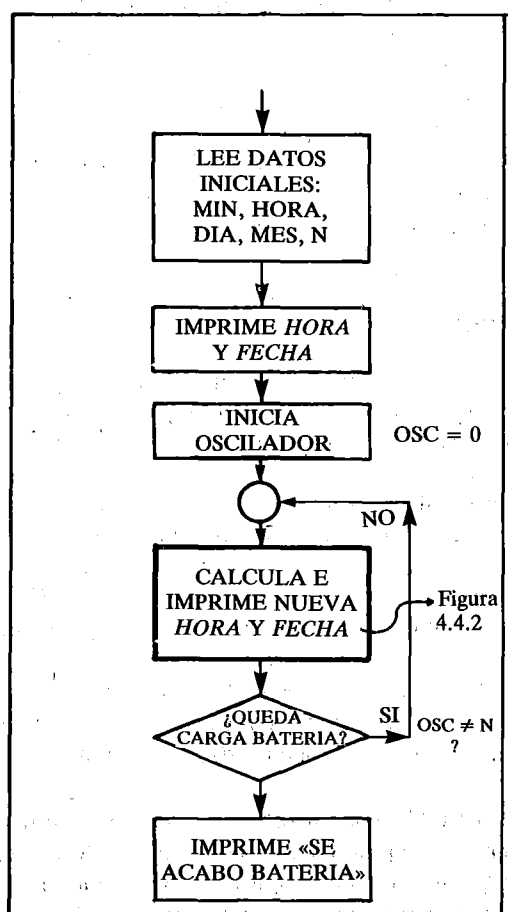


FIGURA 4.4.1.

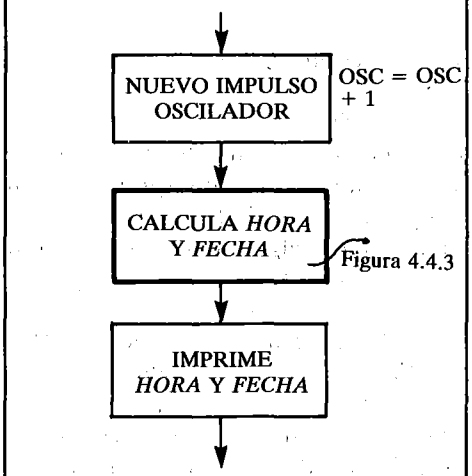


FIGURA 4.4.2.

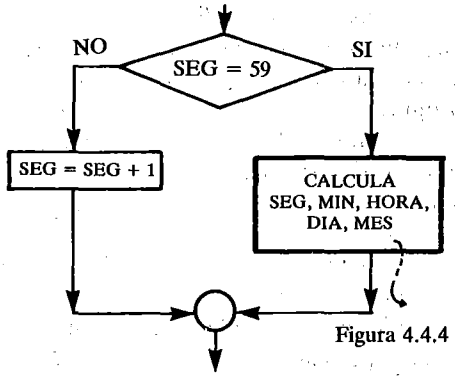


FIGURA 4.4.3.

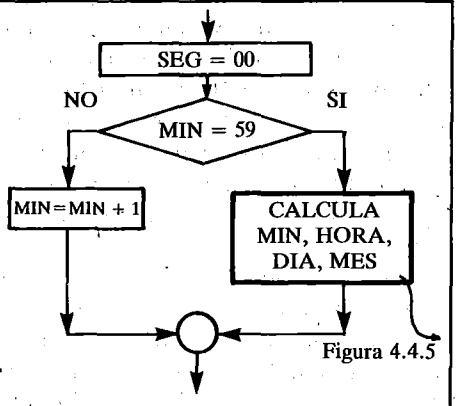


FIGURA 4.4.4.

Para que el lector pueda establecer comparaciones *estructurales*, reproducimos en la figura 4.3 un organigrama tal como ha sido diseñado por una persona que ha estudiado, codificado y probado este programa. (Piénsese que un programa estructurado no tiene necesariamente por qué ser óptimo desde un punto de vista de minimización del tiempo de ejecución o de la ocupación de memoria. Como ya se ha comentado en el capítulo anterior, la programación estructurada busca la optimización en otros terrenos: en la minimización del tiempo de diseño o modificación de programas, en el rigor y fiabilidad de éstos, en su transferibilidad, etc.).

3.4. Desarrollo de una solución estructurada

En el primer nivel se plantea una solución esquemática completa que resuelve totalmente el problema, en el caso de que fuera posible contar con una máquina que tuviera la capacidad de interpretar y ejecutar las instrucciones contenidas en los nodos del organigrama de la figura 4.4.1. Este organigrama tiene una estructura definitiva. Examinemos si algún nodo necesita un desarrollo más detallado, porque cuando así sea (y lo recuadraremos con trazo más fuerte) se tratará de una instrucción dirigida a un recurso abstracto.

En el esquema se entiende por *HORA* y *FECHA* los conjuntos (SEG, MIN, HORA) y (DIA, MES). Cuando HORA no aparece en cursiva es que se refiere al elemento HORA.

Si se exceptúa la instrucción «CALCULA E IMPRIME NUEVA *HORA* y *FECHA*», las demás son prácticamente órdenes traducibles muy sencillamente al lenguaje de cualquier ordenador. Así, pues, se desarrolla dicha instrucción en un nivel de desglose superior (figura 4.4.2).

La instrucción «CALCULA *HORA* Y *FECHA*» se va refinando sucesivamente, de manera que cada vez son menos las cosas que se piden a un recurso abstracto (figuras 4.4.3, 4.4.4 y 4.4.5). Al llegar al nivel representado en la figura 4.4.6 nos encontramos ante un pequeño obstáculo: todos los meses no tienen el mismo número de días. Este es un obstáculo normal. Lo que estaba ocurriendo hasta aquí no dejaba de ser una aburrida monotonía. Si el número de horas que constituyen un día dependiera del día de la semana o el de minutos que forman una hora de que fuera antes o después de mediodía, posiblemente nos haríamos un taco en la vida corriente, pero en lo tocante a la programación actuaríamos como en la figura 4.4.6. Se especificaría una instrucción a un recurso abstracto para que calculase el límite de la cuenta correspondiente a las circunstancias concretas. Y este desarrollo se dejaría para más adelante, en este caso para la figura 4.4.8, donde se plantea una solución estructurada entre varias posibles.

Una vez terminado el proceso de desglose, se recorre el camino en sentido inverso sustituyendo sistemáticamente los diagramas más sencillos en el diagrama de nivel inmediatamente anterior hasta recuperar la estructura completa de la figura 4.4.1, pero ya compuesta con todo el detalle de las operaciones fácilmente traducibles a instrucciones para un ordenador. El organigrama final es el de la figura 4.5, en donde por razones de falta de espacio no se ha sustituido el diagrama de la figura 4.4.8. (Es obvio —el lector puede comprobarlo fácilmente—, que este diagrama dista de ser el

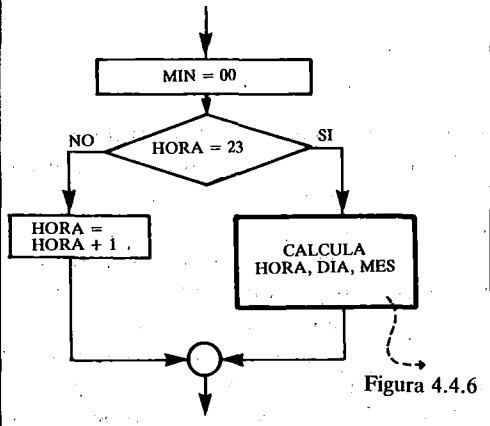


Figura 4.4.6

FIGURA 4.4.5.

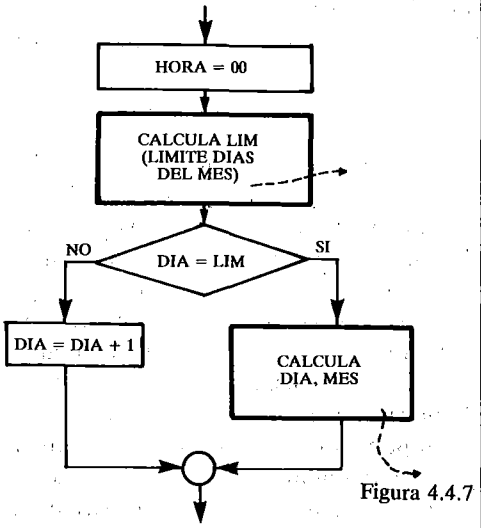


Figura 4.4.7

FIGURA 4.4.6.

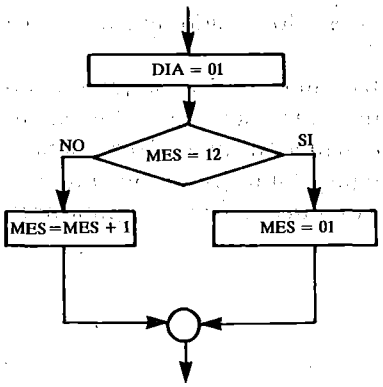


FIGURA 4.4.7.

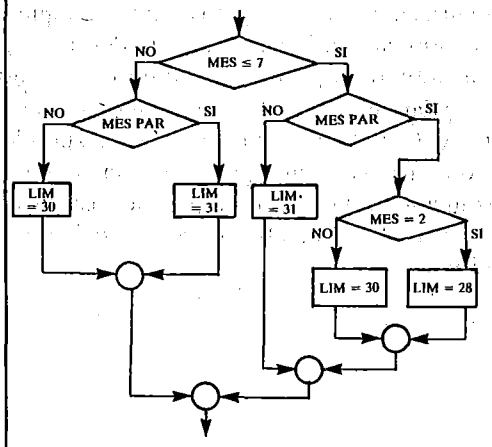


FIGURA 4.4.8.

mejor. Así, sin merma de la buena estructuración del programa podría haberse introducido el cálculo del «límite de los días del mes» al iniciarse cada nuevo mes, siempre que se hubiera introducido como dato el límite del mes en el que se inicializa el reloj).

4. OBSERVACIONES SOBRE EL MÉTODO

Alcanzar un cierto dominio en la aplicación de la reglas generales del método de diseño que se acaba de ver, es algo que se consigue con la práctica y no está exento de dificultades. Pero se hace necesario tomar siempre en cuenta que estas reglas no excluyen —sino más bien al contrario— el uso de la lógica que, en el caso de la programación, se vincula estrechamente a la estructura de los datos que ha de tratar el programa. El mismo ejemplo del reloj digital puede proporcionarnos la oportunidad de mostrar hasta donde nos llevaría el uso ciego (sin lógica) del método.

Supongamos que se nos pide modificar el organigrama de la figura 4.5 para simular el funcionamiento del reloj en el caso de que éste contase con una ventana visualizadora más, de dos campos, uno para el día de la semana (DIASEM) y otro para el día del año (DIAAÑO). (El primer día de la semana es el lunes y el año será siempre de 365 días).

El lector debe intentarlo por sí mismo, antes de mirar una solución que se le ofrece más abajo. Ciertamente el problema parece simple, y lo es. Sin embargo, un elevado porcentaje de los alumnos que tuvieron que resolverlo de manera imperativa en una experiencia nuestra lo hicieron mal. Más o menos, como se indica en las figuras 4.4'.5, 4.4'.6, 4.4'.7 y 4.4'.8. En ellas parece haberse aplicado sistemáticamente (mejor diríase ciegamente) el método, a imagen y semejanza del proceso seguido en las figuras 4.4.1 a 4.4.8. El resultado es que el programa modifica el «día del año» sólo cuando cae en lunes, y el «día del mes» y el «mes» sólo cuando, además de caer en lunes fuera el primer día de un nuevo año.

¿Qué ha ocurrido? Simplemente que no se ha tenido en cuenta que las variables DIASEM, DIAAÑO y DIA corresponden a tres enumeraciones distintas, sin relación entre sí, y que, por tanto, hay que llevar de manera disjunta. Véase una posible solución en la figura 4.4".6, que sustituiría a la figura 4.4.6. (Naturalmente, sería imprescindible leer DIASEM y DIAAÑO al principio del programa e imprimirlo a lo último).

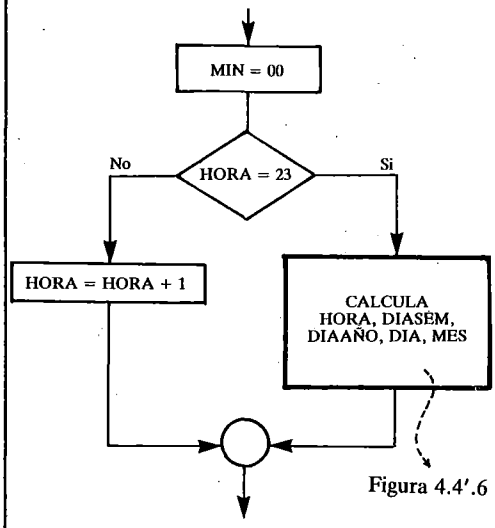


FIGURA 4.4'.5.

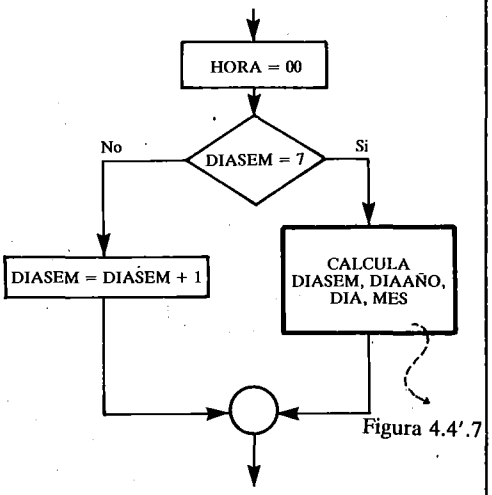


FIGURA 4.4'.6.

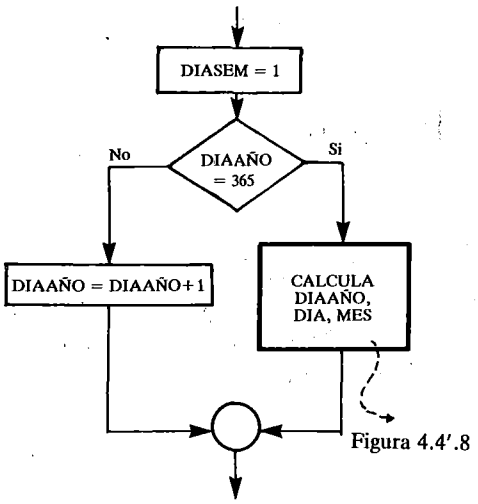


FIGURA 4.4'.7.

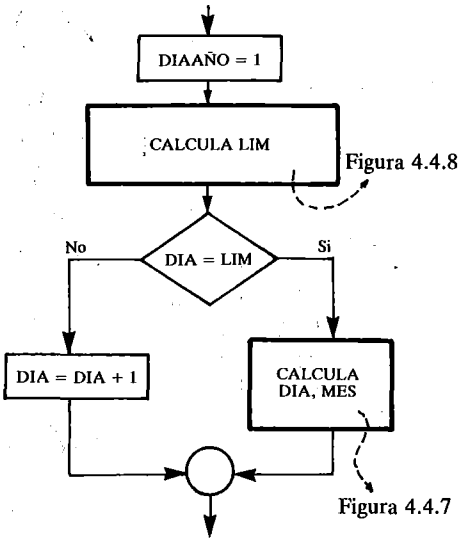


FIGURA 4.4'.8.

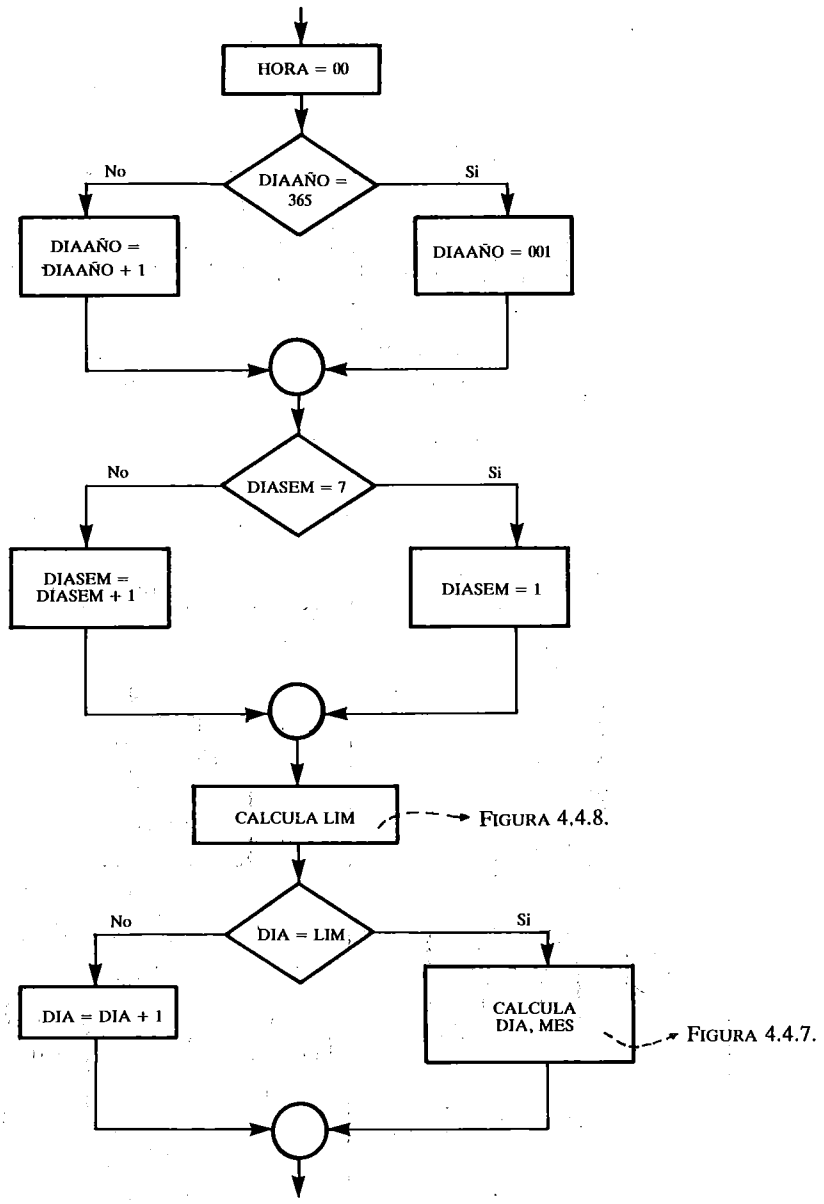

HORA = 00
DIAAÑO = 365
No
Si
DIAAÑO = DIAAÑO + 1
DIAAÑO = 001
DIASEM = 7
No
Si
DIASEM = DIASEM + 1
DIASEM = 1
CALCULA LIM
FIGURA 4.4.8.
DIA = LIM
No
Si
DIA = DIA + 1
CALCULA DIA, MES
FIGURA 4.4.7.

FIGURA 4.4".6.

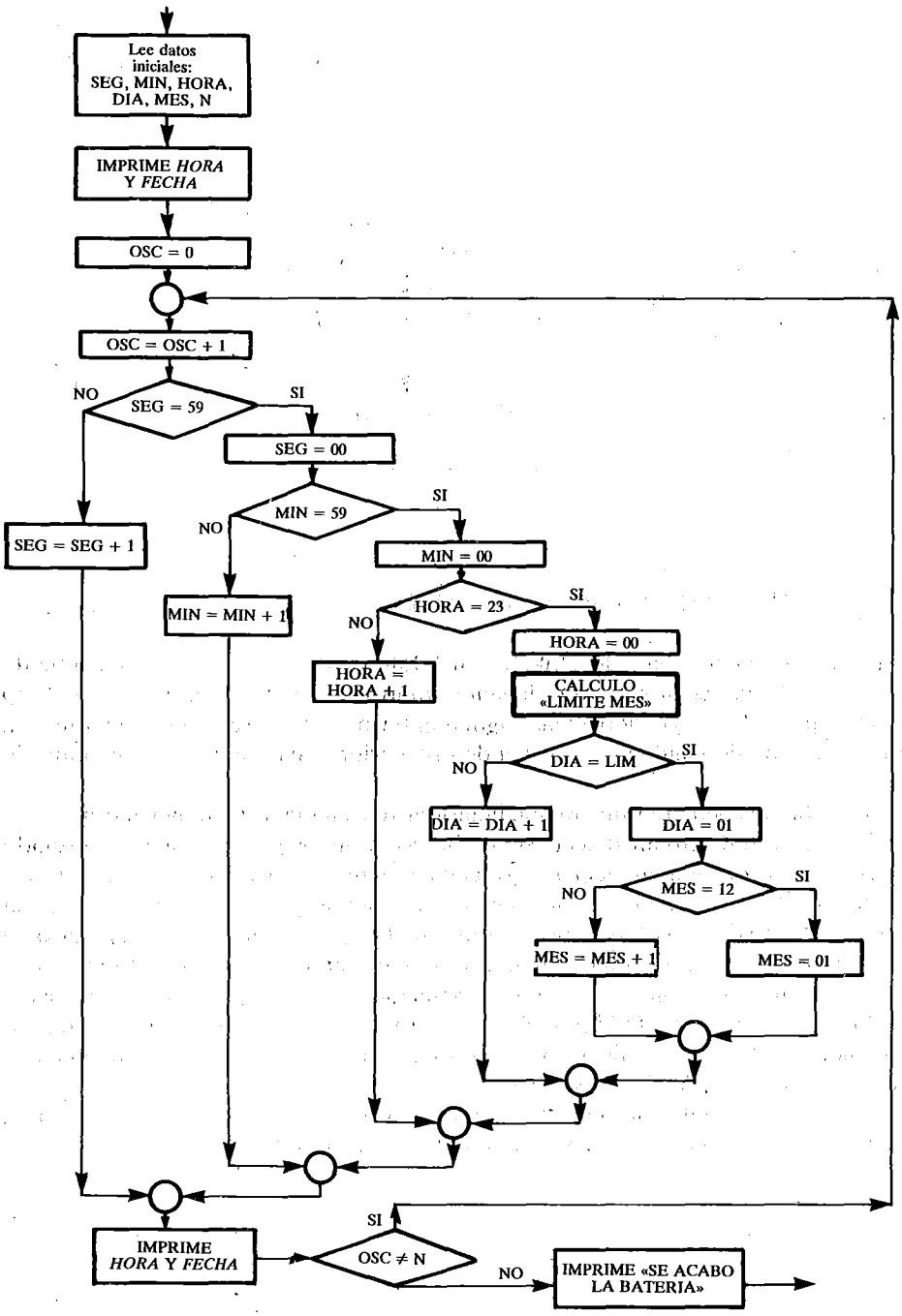

Lee datos iniciales: SEG, MIN, HORA, DIA, MES, N
IMPRIME *HORA* Y *FECHA*
OSC = 0
OSC = OSC + 1
NO
SEG = 59
SI
SEG = 00
SEG = SEG + 1
NO
MIN = 59
SI
MIN = 00
MIN = MIN + 1
HORA = 23
SI
NO
HORA = 00
HORA = HORA + 1
CALCULO «LIMITE MES»
NO
DIA = LIM
SI
DIA = DIA + 1
DIA = 01
NO
MES = 12
SI
MES = MES + 1
MES = 01
IMPRIME *HORA* Y *FECHA*
SI
OSC ≠ N
NO
IMPRIME «SE ACABO LA BATERIA»

FIGURA 4.5.

5. RESUMEN

El método general de diseño de programas estructurados se resume fácilmente por el esquema conceptual de la figura 4.6, sin necesidad de mayores palabras.

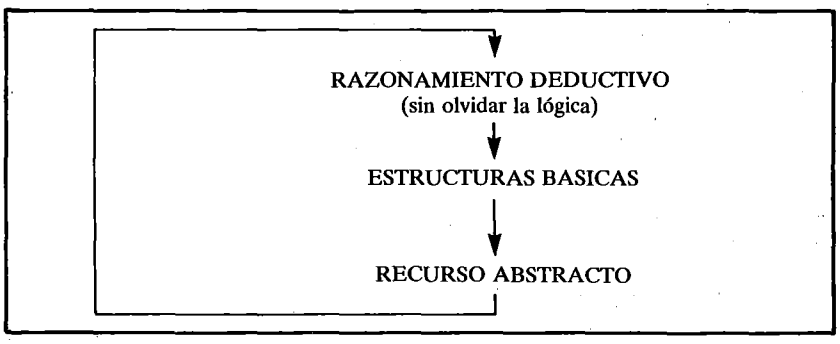


FIGURA 4.6.

6. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

A nuestro entender, ha sido Dijkstra en (Dahl *et al.*, 1972), quien inicialmente describió, y además de forma tan elegante como en él es costumbre, un método parecido al que se ha pergeñado en las páginas anteriores. Concretamente, él utilizaba la descomposición descendente en forma de instrucciones para una supermáquina sin existencia física.

Todos debemos entender que lo que hemos visto es un método muy general, y por ello utilizable en toda circunstancia. Sin embargo, motivos prácticos han aconsejado el desarrollo de procedimientos más detallados, sistemáticos y operativos para ciertas clases de problemas, singularmente algunos, frecuentes en el área de la llamada informática de gestión. Autores como Yourdon, Linger y Mills (EE.UU.), Jackson (G.B.), Warnier y Bertini (Francia) han desarrollado diferentes métodos prácticos de diseño de la programación basados en los principios de la P.E. Véanse al respecto Yourdon (1975), Jackson (1975) y Warnier (1973). Sáez Vacas ha estudiado las funcionalidades de algunos de estos métodos y su relación con los fundamentos científicos de la P.E. (Sáez Vacas, 1976).

El ejemplo de simulación del reloj digital, que nos ha servido de vehículo para explicar el método general, lo hemos adaptado de Arbib (1977).

Capítulo 5
MAQUINA DE TURING: DEFINICION, ESQUEMA FUNCIONAL Y EJEMPLOS

1. INTRODUCCIÓN

El propósito fundamental de este capítulo es definir *qué es, cómo funciona y cómo se diseña una máquina de Turing*. A través de varios ejemplos se familiarizará el lector con la estructura de esta máquina, los alfabetos externo e interno, su programación y las distintas representaciones de la misma (lista de quíntuplas, esquema funcional, diagrama de estados) y la representación y manejo de las informaciones en la cinta de la máquina.

Pondremos el máximo de énfasis en visualizar configuraciones sucesivas y significativas de la información almacenada en la cinta de la máquina.

2. DEFINICIÓN DE MÁQUINA DE TURING

Una máquina de Turing es un autómata finito, junto con una cinta de longitud infinita (que en cualquier momento contiene sólo un número finito de símbolos) dividida en casillas (cada casilla puede contener un solo símbolo o estar en blanco) y un aparato para explorar (leer) una casilla, imprimir un nuevo símbolo sobre ella, y mover la cinta una casilla a la derecha o a la izquierda. Veamos en detalle estos elementos:

a) *La cinta constituye una memoria infinita.* Los símbolos que sobre ella están escritos o se pueden escribir, e_i, $i \in I \subset N$, pertenecen a un conjunto finito E, al que llamamos alfabeto externo de la máquina. Hagamos $m = \text{Card}(E)$. Estos m símbolos sirven para codificar la información suministrada a la máquina. El símbolo blanco, o vacío (□ generalmente, a no ser que se designe expresamente por 0) forma parte del alfabeto, lo que no querrá decir que pueda considerarse infinita la información

contenida en la cinta. Por tanto, en un estadio cualquiera del funcionamiento de la máquina, toda información registrada sobre la cinta se presenta bajo la forma de una palabra escrita en el alfabeto externo E, a razón de un símbolo por casilla.

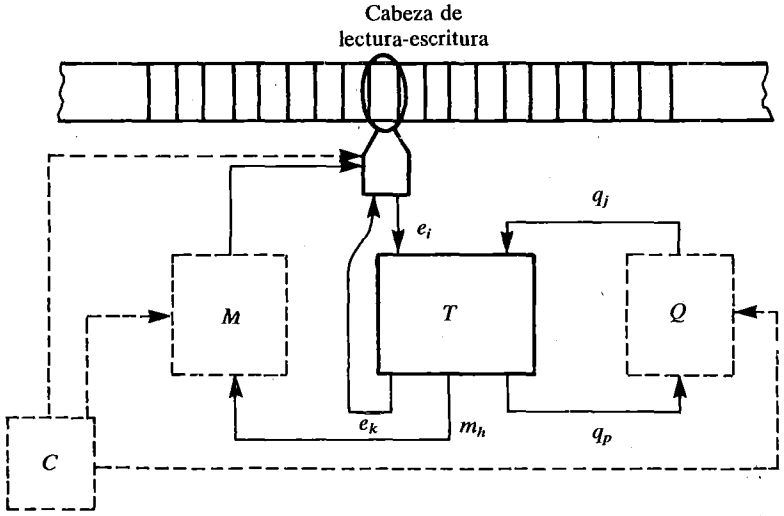


FIGURA 5.1.

b) En principio, había una cabeza de lectura y escritura inmóvil y la cinta se desplazaba bajo la misma. Pero en este texto —y no somos los únicos en hacerlo así—, supondremos de ahora en adelante que ocurre lo contrario: *la cinta estará inmóvil y se desplazará la cabeza*, con lo que resultará más sencilla la representación gráfica de la dinámica del contenido de la cinta. Dado que la cabeza se desplaza una posición a la derecha o a la izquierda o no se desplaza, es evidente que las instrucciones que puede ejecutar esta máquina tienen que estar compuestas de las especificaciones de una operación de lectura/escritura y de un movimiento (derecha (→), izquierda (←), inmóvil (↔); conjunto M, $M = \{\rightarrow, \leftarrow, \leftrightarrow\}$). (*Observación*: puede definirse la M.T. con $M = \{\rightarrow, \leftarrow\}$, pero aquí se ha optado por la primera versión, más flexible).

c) El bloque T, que puede atravesar diferentes estados q_j, de un conjunto finito Q, está conectado a la cabeza de lectura/escritura por un enlace de entrada (lectura, símbolo e_i) y un enlace de salida (escritura, símbolo e_k). El estado le es presentado por el bloque Q. La función lógica realizada por el bloque T hace corresponder a la pareja (e_i, q_j) un vector de salida (e_k, m_h, q_p) con $e_i, e_k \in E$, $m_h \in M$, $q_j, q_p \in Q$. Naturalmente, *el autómata, en un sentido formal, está constituido por los bloques T y Q*. Los bloques Q y M actúan como dos memorias internas, que conservan respectivamente el estado q_p y la orden de movimiento m_h, producidos por el autómata durante su trabajo, hasta el comienzo del instante siguiente, comienzo desencadenado por un secuenciador C, que controla los pasos o fases de trabajo. (*Nota:* aunque se designen por las mismas letras M y Q, no confundir bloque con conjunto. El contexto dirá de qué cosa se trata).

En un eje de tiempos se representan las fases del funcionamiento de una M.T. (figura 5.2).

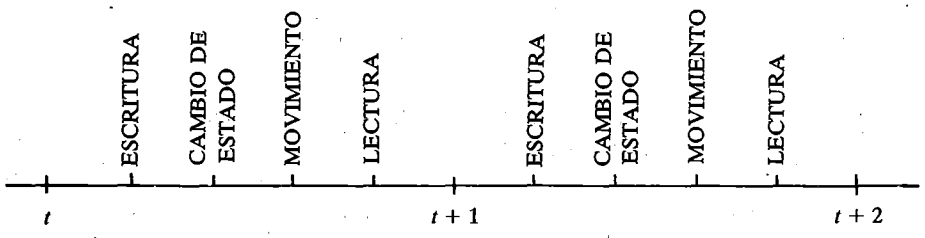


FIGURA 5.2.

No es esencial el orden en que ocurran «cambio de estado» y «movimiento de la cabeza» que, incluso, pueden suponerse simultáneos.

Con lo dicho, *podemos definir así el autómata finito* $T - Q$, que es el componente lógico de la máquina de Turing:

$$T - Q = \langle E, (E \times M) \cup (\text{Stop}), Q, f, g \rangle \quad (1)$$

E conjunto finito de símbolos en la cinta (alfabeto externo) (2)

M conjunto finito de movimientos de la cabeza (3)

Q conjunto finito de estados internos (alfabeto interno) (4)

f: $E \times Q \rightarrow Q$ función de transición (5)

g: $E \times Q \rightarrow (E \times M) \cup (\text{Stop})$ función de salida (6)

Cualquiera que sea la información, o palabra A escrita inicialmente en la cinta según el alfabeto E, pueden ocurrir dos cosas:

1. Bien, al cabo de un número finito de pasos se detiene la máquina dando la señal de stop. Sobre la cinta se encontrará una cierta palabra B escrita según el mismo alfabeto, que representa la información resultante. Se dice que *la máquina es aplicable a la información A, y que transforma A en B.*
2. Bien, la señal de stop no se produce nunca, en cuyo caso se dice que *la máquina es inaplicable a la información inicial A.*

El funcionamiento de una particular y concreta máquina de Turing podrá describirse especificando los items (1) a (6), junto con los siguientes elementos:

Información inicial registrada en la cinta (7)

Posición inicial de la cabeza (8)

Estado inicial del autómata (9)

o, lo que es lo mismo, mediante los items (7) a (9) y una tabla (también llamada esquema funcional) de todas las quíntuplas posibles, e_i, q_j, e_k, m_h, q_p. Esta tabla es un

programa para la máquina de Turing, cuyas instrucciones dicen: «si la máquina está en el estado q_j y lee el símbolo e_i, que escriba el símbolo e_k, mueva la cabeza una posición m_h y cambie al estado q_p».

(*Nota*: reflexione el lector que en la definición y esquema de la M.T. están presentes de una mera o de otra todos los subsistemas de la figura 2.3 del capítulo 2).

3. FUNCIONAMIENTO DE LA MÁQUINA DE TURING A TRAVÉS DE LOS EJEMPLOS

En este apartado aprenderemos, a través de cinco ejemplos, a analizar máquinas de Turing. En el subapartado 3.3 entraremos en contacto con el procedimiento de composición de máquinas, una especie de diseño modular de máquinas de Turing. A lo largo del apartado nos familiarizaremos adicionalmente con algunos conceptos, tales como la *descripción instantánea* y el cálculo de una M.T.

3.1. Suma de dos números enteros no nulos escritos en el alfabeto {|}

La tabla adjunta expresa el programa correspondiente a un algoritmo de suma de dos números enteros no nulos para una máquina de Turing, con las siguientes características:

$$E = \{\square, |, *\}$$
$$Q = \{q_0, q_1, q_2\}$$

e_i	q_j	e_k	m_h	q_p
|	q_0	□	→	q_2
|	q_1	|	←	q_1
|	q_2	|	→	q_2
□	q_0	□	→	q_0
□	q_1	□	→	q_0
□	q_2	|	↔	q_1
*	q_0	□	↔	stop
*	q_1	*	←	q_1
*	q_2	*	→	q_2

e_i \ q_j	q_0	q_1	q_2
|	□ → q_2	| ← q_1	| → q_2
□	□ → q_0	□ → q_0	| ↔ q_1
*	□ ↔ stop	* ← q_1	* → q_2

FIGURA 5.3. *Dos representaciones tabulares alternativas.*

— Información inicial en la cinta: los dos números enteros representados por grupos de | (ejemplo, 3 ≡ |||; 5 ≡ |||||) separados por un asterisco *.
— Posición inicial de la cabeza: sobre el primer | de la izquierda.
— Estado inicial del autómata: q_0.

En el primer instante, la pareja de entrada es (|, q_0), lo que ocasiona un trío ($\square \rightarrow q_2$), esto es, se borra el primer palote, se desplaza la cabeza una posición a la derecha y se pasa al estado q_2. Y así una y otra vez. La mejor forma de apreciar la mecánica será con un caso numérico concreto, por ejemplo, la suma de 2 y 3, situado

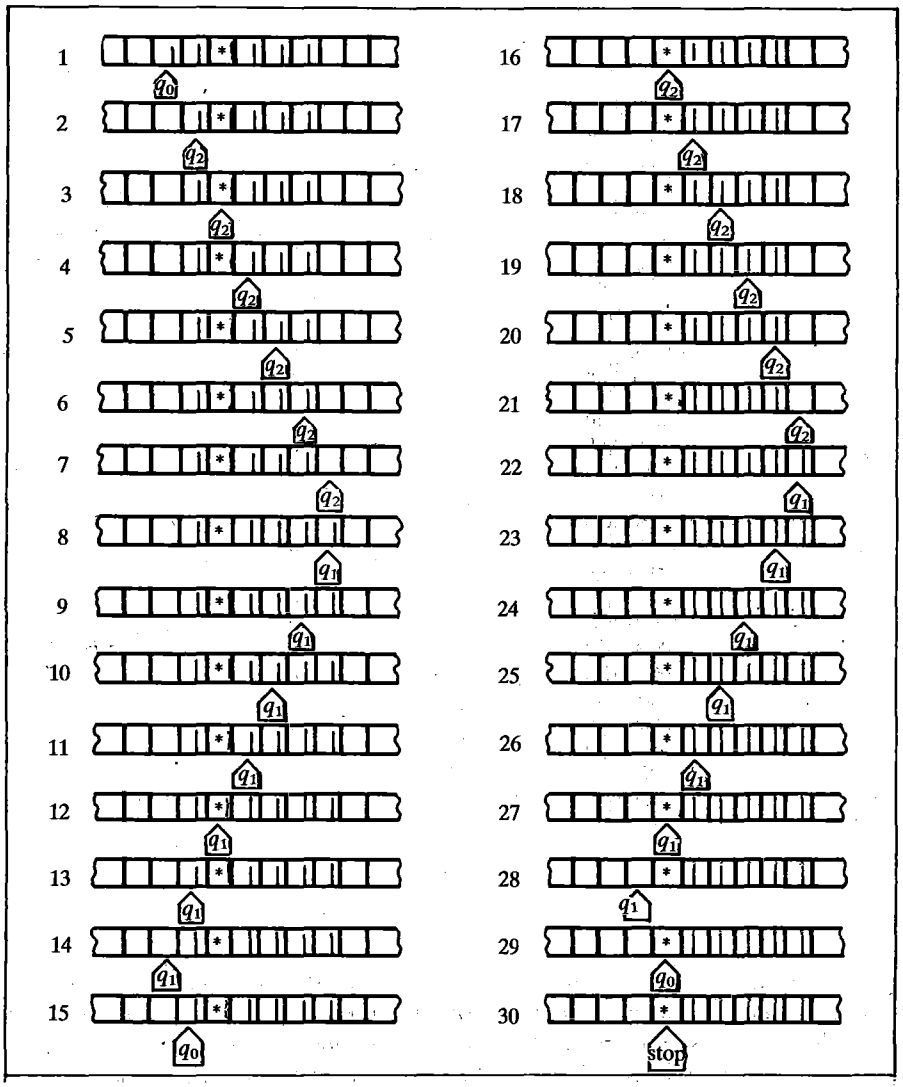


FIGURA 5.4. *Máquina de Turing para sumar dos números enteros no nulos.*

el primero de ellos a la izquierda. Véanse en la figura 5.4 las configuraciones sucesivas de la suma 2 + 3 en una máquina de Turing.

La máquina se detiene en el movimiento n.° 30. La información resultante, o solución del problema, o palabra B, aparece situada a la derecha de la cabeza cuando ésta se para.

Es evidente que la presentación de las configuraciones de la cinta bajo la forma de la figura 5.4 es poco manejable. Por tal razón, se le prefiere las de la figura 5.5 y 5.6, en las que prácticamente siempre el símbolo «blanco» sería el cero.

CINTA	*ESTADO*
00|*||00	q_0
000|*||00	q_2
000|*||00	q_2
000|*||00	q_2
000|*||00	q_2
000|*||00	q_2
000|*||00	q_2
000|*||0	q_1
000|*||0	q_1
000|*||0	q_1
000|*||0	q_1
000|*||0	q_1
000|*||0	q_1
000|*||0	q_1
000|*||0	q_0
0000*||0	q_2
0000*||0	q_2
0000*||0	q_2
0000*||0	q_2
0000*||0	q_2
0000*||0	q_2
0000*|||	q_1
0000*|||	q_1
0000*|||	q_1
0000*|||	q_1
0000*|||	q_1
0000*|||	q_1
0000*|||	q_1
0000*|||	q_0
00000||||	stop

FIGURA 5.5.

A las expresiones de la figura 5.6 se les llama *descripciones instantáneas*. La descripción instantánea da de manera absolutamente precisa el comportamiento de la M.T. En efecto, la descripción, si no se toma en cuenta q_j, da el contenido de la cinta y, puesto que el símbolo a la derecha de q_j en la expresión es e_i (por eso no es necesario el subrayado indicativo de la posición de la cabeza), se tiene además la

$00q_0||*|||00$
$000q_2|*|||00$
$000|q_2*|||00$
$000|*q_2|||00$
$000|*|q_2||00$
$000|*||q_2|00$
$000|*|||q_200$
$000|*|||q_1|0$
$000|*||q_1||0$
$000|*|q_1|||0$
$000|*q_1||||0$
$000|q_1*||||0$
$000q_1|*||||0$
$00q_10|*||||0$
$000q_0|*||||0$
$0000q_2*||||0$

FIGURA 5.6.

pareja (e_i, q_j) representativa de la situación de trabajo en que se encuentra la máquina.

Aprovechemos este ejemplo y esta última representación para definir el cálculo de una máquina de Turing. Encontramos esta definición en el cuadro de la figura 5.7.

Cálculo de una máquina de Turing

Se dice que una descripción instantánea γ es *terminal*, relativamente a una máquina Z, si no existe ninguna descripción ω tal que $\gamma \rightarrow \omega$ ($\rightarrow$ significa aquí *es seguida por*).

Se llama cálculo de una M.T. determinada, a una sucesión:

$$\gamma_1, \gamma_2, \ldots, \gamma_p$$

de descripciones instantáneas tales que $\gamma_i \rightarrow \gamma_{i+1}$, para $1 \leq i < p$ con γ_p terminal. Se puede escribir:

$$\gamma_p = \text{Res } Z(\gamma_1), \tag{10}$$

lo que se lee: «γ_p es el resultado de γ_1 para la máquina Z».

FIGURA 5.7.

Nota: Obsérvese que la expresión (10) es una forma más completa de decir que la máquina Z es aplicable a una determinada información, ya que incluye el estado y la posición de la cabeza en la situación inicial y en la final.

3.2. Algoritmo de Euclides para el cálculo del m.c.d. de dos números enteros escritos en el alfabeto {|}

$$E = \{\square, |, *, \alpha, \beta\}$$

$$Q = \{q_0, q_1, q_2, q_3, q_4\}$$

— Información inicial en la cinta: los dos números enteros representados por grupos de |, separados por un asterisco *.
— Posición inicial de la cabeza: sobre el primer | del número de la izquierda.
— Estado inicial del autómata: q_0.

El esquema funcional (los recuadros vacíos significan combinaciones imposibles o indiferentes) se representa en la figura 5.8.

e_i \ q_j	q_0	q_1	q_2	q_3	q_4
|	$\beta \rightarrow q_0$	$\alpha \leftrightarrow q_2$	$\beta \leftrightarrow q_1$	$\| \rightarrow q_1$	$\| \leftarrow q_1$
$\square$	$\square \rightarrow q_3$	$\square \rightarrow q_4$	$\square \leftarrow q_3$	$\square \rightarrow q_1$	$\square \leftrightarrow$ stop
*	$\alpha \leftarrow q_0$		$\| \rightarrow q_2$	$\square \rightarrow q_2$	—
α		$\alpha \leftarrow q_1$	$\alpha \rightarrow q_2$	$\| \leftarrow q_3$	$\square \rightarrow q_4$
β	$* \leftarrow q_0$	$\beta \leftarrow q_1$	$\beta \rightarrow q_2$	$\square \leftarrow q_3$	$\| \rightarrow q_4$

FIGURA 5.8. *Máquina de Turing para el algoritmo de Euclides aplicado a dos números enteros en* {|}.

La misma información que hay en el esquema funcional puede expresarse mediante un diagrama de estados que no es, en definitiva, otra cosa que un organigrama de los que típicamente se usan en programación (figura 5.9) o en la representación de una máquina de Mealy (figura 5.10). (Véanse en cuadro separado los dos tipos de bucles que ocurren en un diagrama de estados).

Sigamos ahora la explicación del proceso de cálculo, para lo que nos puede servir de ayuda la ilustración de algunos instantes significativos en el procesamiento de la cinta en el caso particular de los números 3 (izquierda) y 6 (derecha). (Véase figura 5.11).

La primera fase del proceso consiste en suprimir el asterisco que separa los dos números para hacer como si se quisiera yuxtaponer éstos, de manera que formen una palabra escrita en el alfabeto {|}, y en disponer la cinta de suerte que quedase bajo la cabeza el último símbolo |, que se habrá sustituido previamente por α. Esta fase acaba en el movimiento número 13 (¡atención, esto sólo es cierto en el caso práctico considerado!).

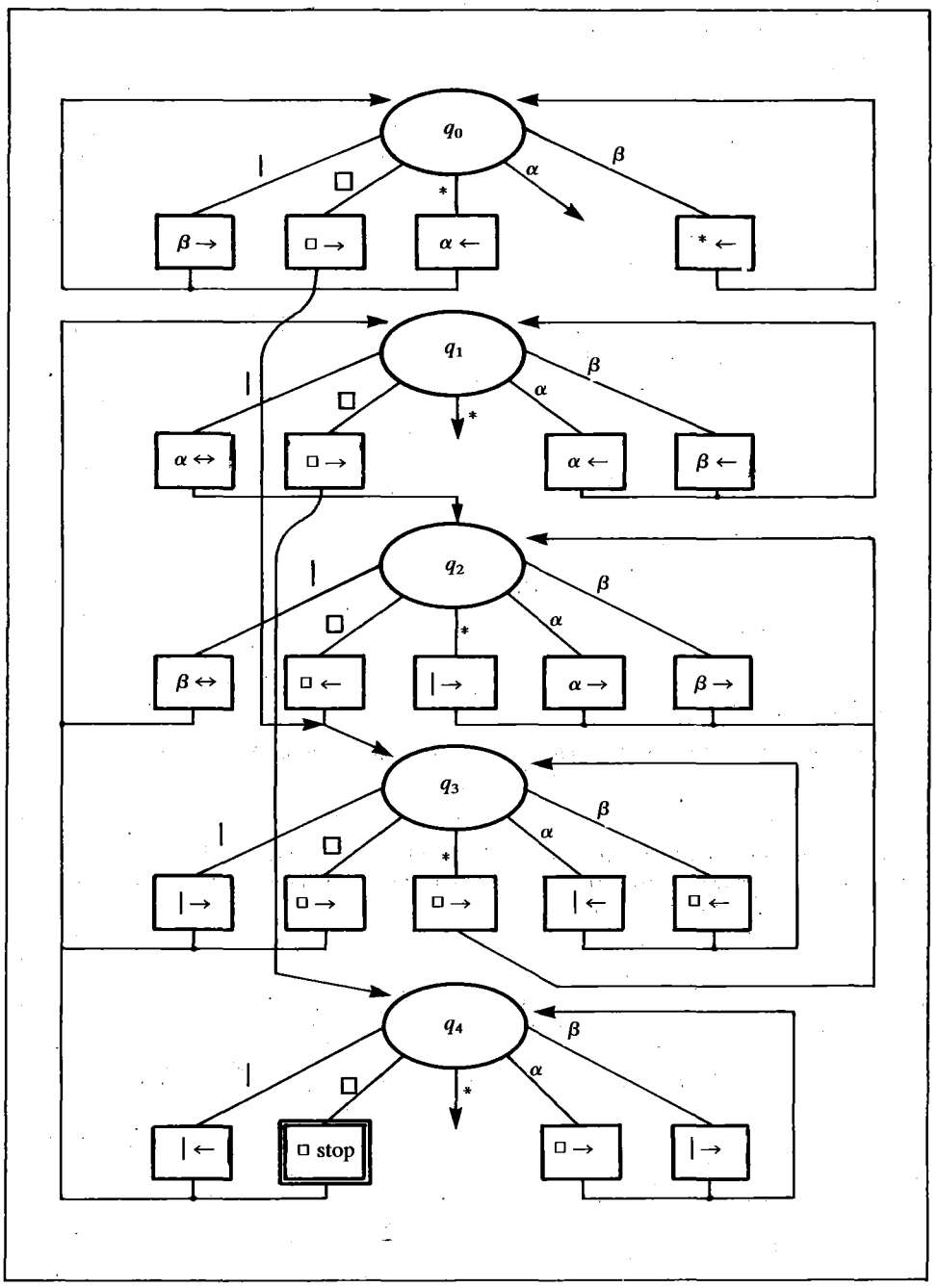


FIGURA 5.9. *Diagrama de estados de una máquina de Turing para el cálculo del m.c.d. de dos números enteros. Los rectángulos representan la acción sobre el contenido y la posición relativa de la cinta.*

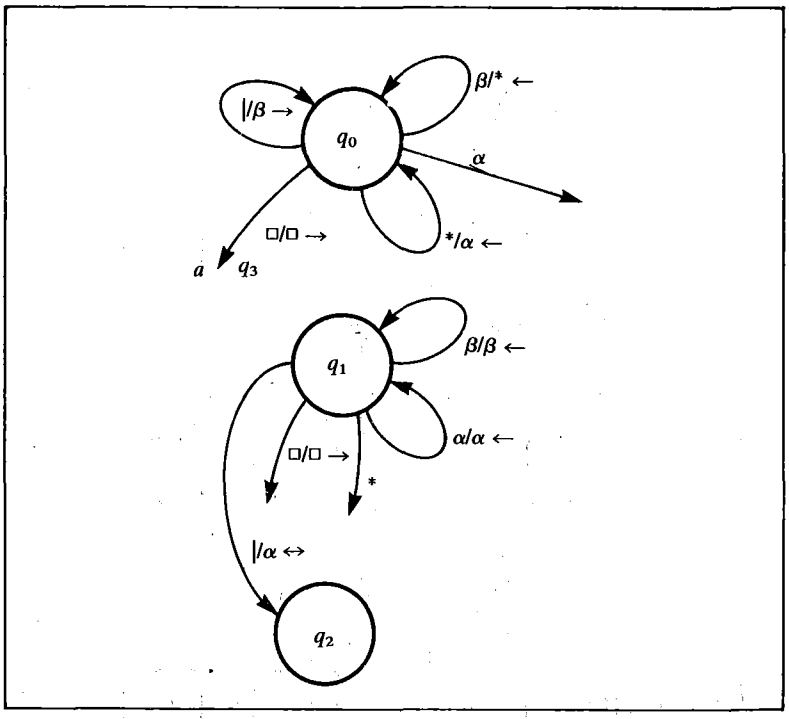
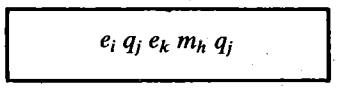
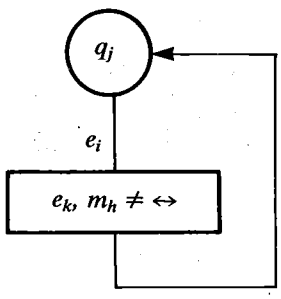


FIGURA 5.10. *Diagrama (incompleto) de estados en forma de máquina de Mealy.*

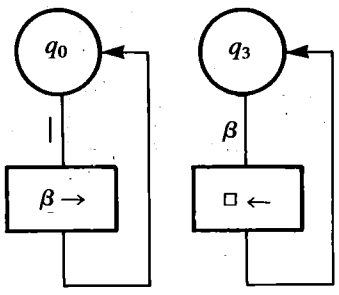
El diagrama de estados permite poner de manifiesto la existencia de dos tipos de bucles, que son aquellos procesos peculiares en que la máquina puede permanecer repetidamente en un mismo estado:

a)



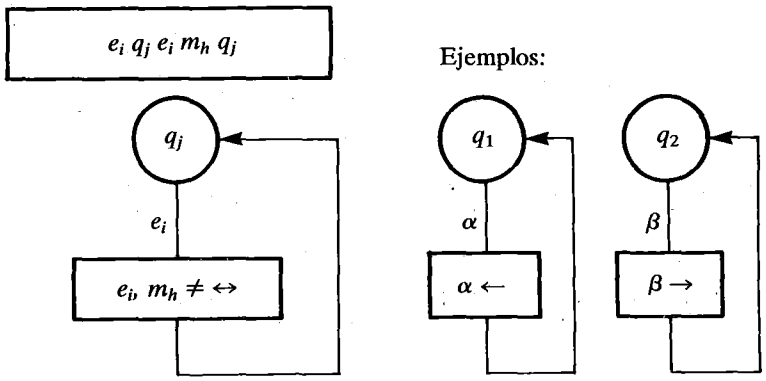


Ejemplos:



Se sustituye un símbolo e_i por otro símbolo distinto e_k sistemáticamente, mientras queden casillas rellenas con e_i en la dirección de exploración.

b)



Se explora la cinta en una determinada dirección sin producir cambios en la información que contiene, mientras la cabeza lea un símbolo determinado e_i.

A esta primera fase siguen los ciclos de comparación y sustracción que movilizan respectivamente los estados q_1, q_2 y los q_3, q_4.

Para comparar los dos números y detectar cual de ellos es mayor, la máquina examina los dos grupos de |, marcando con un símbolo diferente (α para el número de la izquierda, β para el de la derecha), alternativamente un símbolo | de cada grupo. El punto de partida del ciclo de comparación es el movimiento 13, en que ya se ha sustituido un primer símbolo | del número de la izquierda por α. Dos instantes más tarde (configuración 15), el primer símbolo | del segundo número ha sido sustituido por β. El proceso sigue así, en sustituciones alternadas, hasta la configuración 33, en la que vemos que ya no hay más sustituciones posibles en el primer número. Pero esto la máquina no lo descubre hasta transcurridos 6 instantes más (configuración 39).

El estado q_4 provoca, cuando en la cinta hay α o β, sucesivos desplazamientos a derecha con sustitución de α por □ (borrado) y de β por 1, respectivamente. Así se llega a la configuración 46 y un instante después, a la 47. Desde la configuración 40 a la 47 se ha desarrollado un ciclo de sustracción que, si no ha hecho uso del estado q_3 se debe, una vez más, a las características del caso práctico escogido. (El lector puede ensayar con otro par de números. Así se familiarizará con el funcionamiento de una máquina de Turing y, de paso ¡y con paciencia!, podrá verificar si el programa dado para el cálculo del m.c.d. es o no es correcto).

La configuración 48 marca el principio de un nuevo ciclo de comparación hasta el momento 74. Seguidamente tiene lugar otro ciclo de sustracción, hasta el instante 81. Las alternancias de ciclos de comparación y de sustracción seguirían desarrollándose hasta llegar a tener dos números iguales, en cuyo momento se acaba el proceso. Esto es precisamente lo que ya ha ocurrido en el caso del 3 y del 6, y en la configuración 82 se detiene la máquina, quedando el resultado escrito a la izquierda de la cabeza: 3.

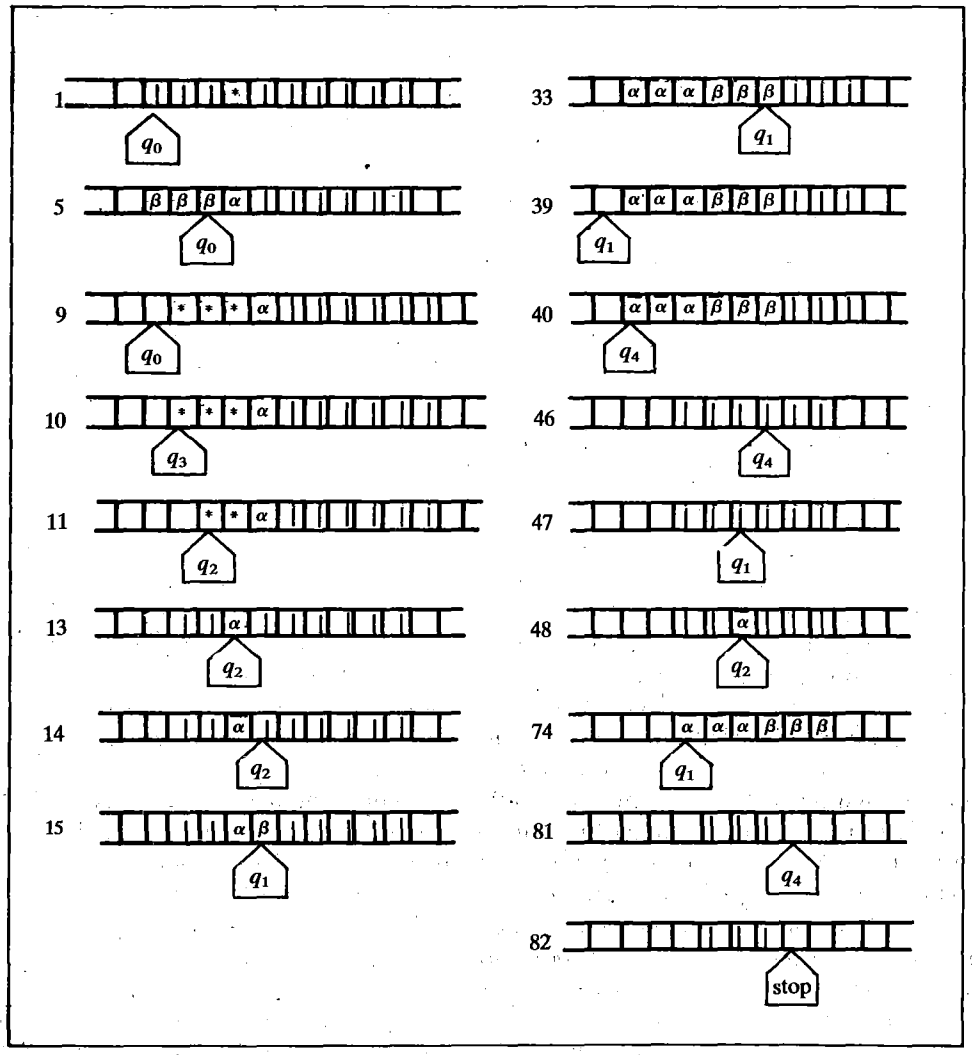


FIGURA 5.11. *Instantes significativos en el proceso de cálculo del m.c.d. de 3 y 6 por una máquina de Turing.*

3.3. Cálculo del m.c.d. de dos números enteros escritos en D (D = {0, 1, 2, 3, ..., 9}) por el procedimiento general de construir un programa a base de subprogramas. Composición de máquinas de Turing

Ahora nos podemos proponer definir una M.T., y su correspondiente programa, para calcular el m.c.d. de dos números enteros expresados en $D = \{0, 1, 2, 3, \ldots, 8, 9\}$, dando el resultado en el mismo alfabeto.

Esto se puede hacer por el sistema de elaborar un diseño original, considerando el problema en su totalidad, lo que no resulta demasiado fácil. También puede descomponerse el problema en partes más sencillas, resolver éstas y después ensamblar las soluciones en un conjunto coherente. Esta técnica es la de composición de M.T.'S. Vamos a seguir este segundo procedimiento, que es muy común en informática, debido al menos a estas tres razones:

a) Es más sencillo (pese a todo, podrá comprobarse cómo rápidamente se hace patente un considerable nivel de dificultad).

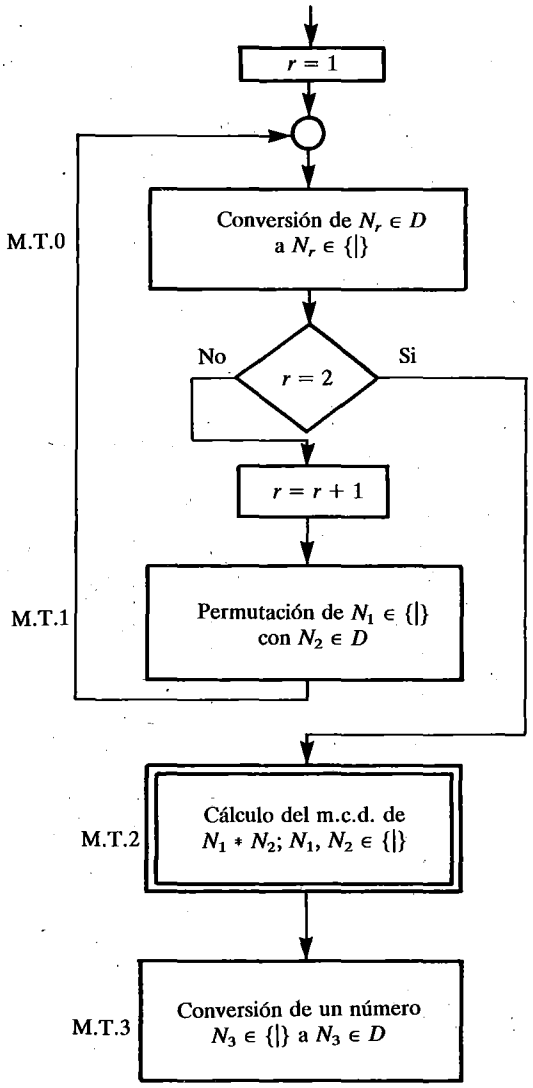


FIGURA 5.12.

b) Es más fiable.
c) Es más económico, por ser más sencillo, por ser más fiable y porque puede aprovecharse trabajo ya desarrollado anteriormente.

Y aquí, adicionalmente, es más didáctico. *Se utilizará como subprograma básico la M.T. definida en el apartado 3.2.*

La información inicial de la cinta será $N_2 * N_1$, con $N_1, N_2 \in D$. Si se pretende aprovechar el diseño de M.T. de 3.2 será necesario crear otros subprogramas (de manera más rigurosa, otras M.T.) con la secuencia general expresada por el organigrama de la figura 5.12, que representa, en definitiva, un algoritmo ejecutado por cuatro máquinas de Turing ensambladas.

(No se olvide que todas las máquinas de Turing son estructuralmente idénticas, lo que se materializa por el formato constante y el significado de sus instrucciones e_i, q_j, e_k, m_h, q_p. En virtud de ello es posible aspirar a fundir en una sola M.T. varias M.T. distintas, siempre que se compatibilicen sus elementos diferenciales o condiciones de contorno: los alfabetos, las condiciones iniciales, las funciones f, g y las condiciones terminales).

q_j / e_i	q_0	q_1	q_2
0	$9 \leftarrow$	$\rightarrow$	
1	$0 \rightarrow q_1$	$\rightarrow$	
2	$1 \rightarrow q_1$	$\rightarrow$	
3	$2 \rightarrow q_1$	$\rightarrow$	
4	$3 \rightarrow q_1$	$\rightarrow$	
5	$4 \rightarrow q_1$	$\rightarrow$	
6	$5 \rightarrow q_1$	$\rightarrow$	
7	$6 \rightarrow q_1$	$\rightarrow$	
8	$7 \rightarrow q_1$	$\rightarrow$	
9	$8 \rightarrow q_1$	$\rightarrow$	$\square \rightarrow$
|	$\leftarrow$	$\rightarrow$	stop
$\square$	$\rightarrow q_2$	$\| \leftarrow q_0$	$\rightarrow$

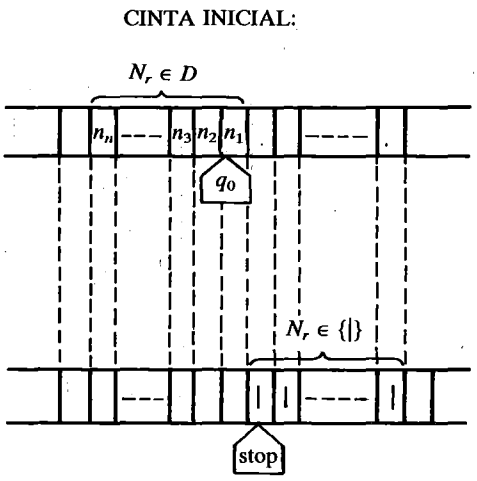


FIGURA 5.13. M.T.0. *Conversión de $N_r \in D$ a $N_r \in \{|\}$.*

A continuación damos los esquemas de M.T.0, M.T.1 y M.T.3. Para aligerar los contenidos de los esquemas, convendremos en suprimir los símbolos de escritura y de estado siguiente cuando sean iguales a los de entrada y el de desplazamiento cuando sea ↔. (Reproducimos M.T.2, también con este convenio. Una casilla vacía indica una combinación imposible, ya que nunca se puede producir ésta por causa del convenio).

e_i \ q_j	q_0	q_1	q_2	q_3
0	→	→	←	←
1	→	→	←	←
2	→	→	←	←
3	→	→	←	←
4	→	→	←	←
5	→	→	←	←
6	→	→	←	←
7	→	→	←	←
8	→	→	←	←
9	→	→	←	←
|		→	←	□ ← q_2
□		← q_3	| → q_1	* → q_1
*	← q_3	→	←	□ ← stop

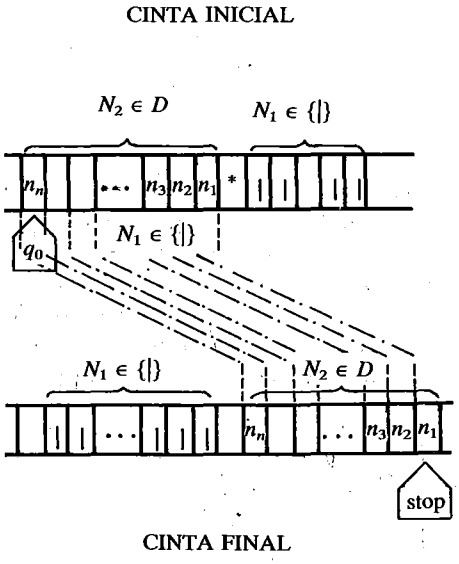


FIGURA 5.14. M.T.1. *Permutación de* $N_1 \in \{|\}$ *con* $N_2 \in D$.

Se recomienda al lector tome los esquemas M.T.0, M.T.1 y M.T.3, que resuelven tres problemas distintos, como ejercicios independientes de análisis del funcionamiento de máquinas de Turing y, al menos, realice el esfuerzo de captar la idea global del proceso de composición de varias M.T., según se expone a continuación. El esquema de la figura 5.18 puede considerarse como letra pequeña, aunque sería de aconsejar su estudio detallado para aquellos lectores que gusten de profundizar algo más.

Para encajar las cuatro máquinas en una sola y definitiva que resuelva el problema que nos hemos planteado, hay que introducir los cambios oportunos —amén de rebautizar los nombres de los estados al objeto de referirlos a un alfabeto interno más amplio— tendentes a adaptar las diferencias mencionadas entre la forma cómo empieza un subprograma y la forma cómo empieza el siguiente (incluyendo las diferencias de alfabeto externo entre máquina y máquina). La figura 5.17 recoge la configuración de la cinta de datos antes y después de aplicarle la M.T. correspondien-

e_i \ q_j	q_0	q_1	q_2	q_3	q_4
|	$\beta \rightarrow$	$\alpha\ q_2$	$\beta\ q_1$	$\rightarrow q_1$	$\leftarrow q_1$
□	$\rightarrow q_3$	$\rightarrow q_4$	$\leftarrow q_3$	$\rightarrow q_1$	stop
*	$\alpha \leftarrow$		$\mid \rightarrow$	$\square \rightarrow q_2$	
α		$\leftarrow$	$\rightarrow$	$\mid \leftarrow$	$\square \rightarrow$
β	$* \leftarrow$	$\leftarrow$	$\rightarrow$	$\square \leftarrow$	$\mid \rightarrow$

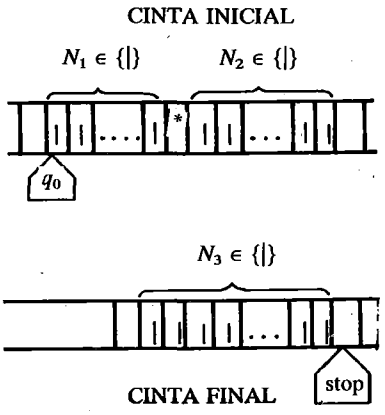


FIGURA 5.15. M.T.2. *Cálculo del m.c.d. de* $N_1 * N_2$; $N_1, N_2 \in \{|\}$.

te al proceso indicado en el organigrama de la figura 5.12 con señalamiento expreso de las modificaciones que hay que producir en aquella para ajustar la situación final de una máquina con la inicial de la siguiente.

En la figura 5.18 damos el esquema funcional de la M.T. solución (que esperamos sea correcto), utilizando superíndices en los nombres de los estados para que el lector vea de qué subprograma procede cada uno de ellos y recuadrando con trazo grueso las combinaciones introducidas para la adaptación a que acabamos de referirnos.

e_i \ q_j	q_0	q_1	q_2
0	$1\ q_2$	stop	$\rightarrow$
1	$2\ q_2$	stop	$\rightarrow$
2	$3\ q_2$	stop	$\rightarrow$
3	$4\ q_2$	stop	$\rightarrow$
4	$5\ q_2$	stop	$\rightarrow$
5	$6\ q_2$	stop	$\rightarrow$
6	$7\ q_2$	stop	$\rightarrow$
7	$8\ q_2$	stop	$\rightarrow$
8	$9\ q_2$	stop	$\rightarrow$
9	$0 \leftarrow$	stop	$\rightarrow$
|	$\leftarrow$	$\square \leftarrow q_0$	$\rightarrow$
□	$1\ q_2$	stop	$\leftarrow q_1$

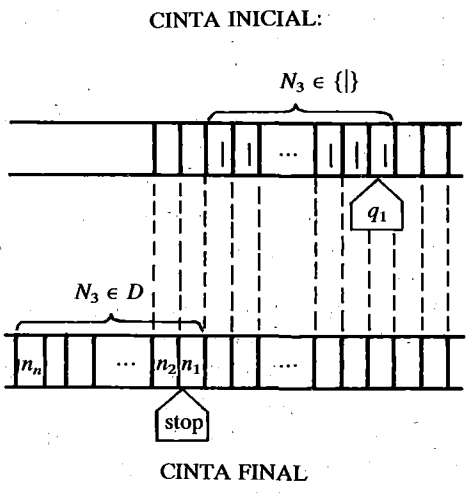


FIGURA 5.16. M.T.3. *Conversión de* $N_3 \in \{|\}$ a $N_3 \in D$.

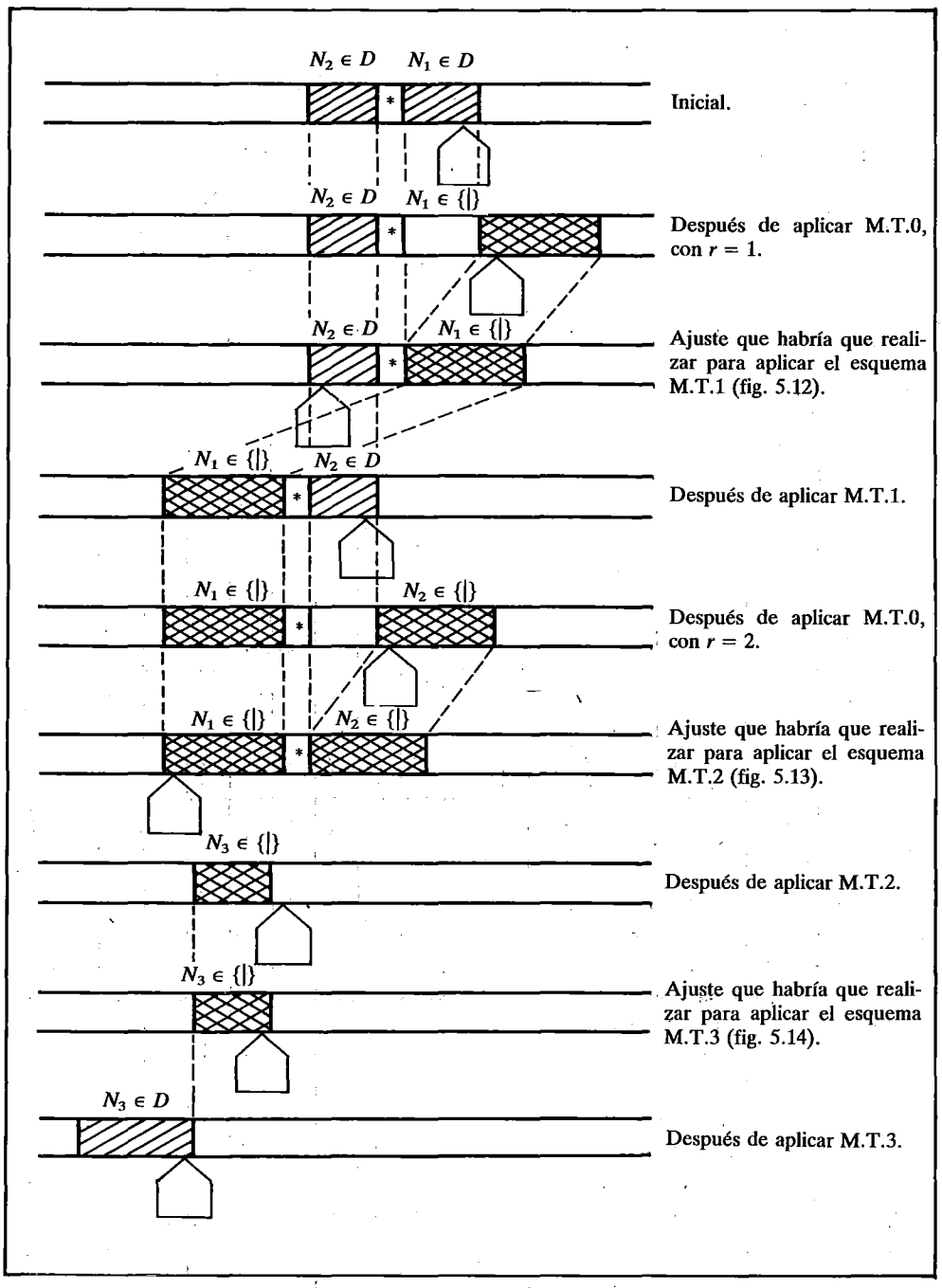


FIGURA 5.17. *Sucesivas configuraciones de la cinta al aplicar el proceso de la figura 5.12.*

	M.T.0′			M.T.1′						M.T.2′				M.T.3′			
q_j / e_i	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}	q_{12}	q_{13}	q_{14}	q_{15}	Nueva denominación de estados
	q_0	q_1	q_2	q_0'	q_1'	q_2'	q_3'	q_0''	q_1''	q_0'''	q_1'''	q_2'''	q_3'''	q_0''''	q_1''''	q_2''''	Denominación parcial
0	$9\leftarrow$	$\rightarrow$		$\leftarrow$	$\rightarrow$	$\leftarrow$	$\leftarrow$	$9\leftarrow$	$\rightarrow$					$1q_2''''$	stop	$\rightarrow$	
1	$0\rightarrow q_1$	$\rightarrow$		$\leftarrow$	$\rightarrow$	$\leftarrow$	$\leftarrow$	$0\rightarrow q_1''$	$\rightarrow$					$2q_2''''$	stop	$\rightarrow$	
2	$1\rightarrow q_1$	$\rightarrow$		$\leftarrow$	$\rightarrow$	$\leftarrow$	$\leftarrow$	$1\rightarrow q_1''$	$\rightarrow$					$3q_2''''$	stop	$\rightarrow$	
3	$2\rightarrow q_1$	$\rightarrow$		$\leftarrow$	$\rightarrow$	$\leftarrow$	$\leftarrow$	$2\rightarrow q_1''$	$\rightarrow$					$4q_2''''$	stop	$\rightarrow$	
4	$3\rightarrow q_1$	$\rightarrow$		$\leftarrow$	$\rightarrow$	$\leftarrow$	$\leftarrow$	$3\rightarrow q_1''$	$\rightarrow$					$5q_2''''$	stop	$\rightarrow$	
5	$4\rightarrow q_1$	$\rightarrow$		$\leftarrow$	$\rightarrow$	$\leftarrow$	$\leftarrow$	$4\rightarrow q_1''$	$\rightarrow$					$6q_2''''$	stop	$\rightarrow$	
6	$5\rightarrow q_1$	$\rightarrow$		$\leftarrow$	$\rightarrow$	$\leftarrow$	$\leftarrow$	$5\rightarrow q_1''$	$\rightarrow$					$7q_2''''$	stop	$\rightarrow$	
7	$6\rightarrow q_1$	$\rightarrow$		$\leftarrow$	$\rightarrow$	$\leftarrow$	$\leftarrow$	$6\rightarrow q_1''$	$\rightarrow$					$8q_2''''$	stop	$\rightarrow$	
8	$7\rightarrow q_1$	$\rightarrow$		$\leftarrow$	$\rightarrow$	$\leftarrow$	$\leftarrow$	$7\rightarrow q_1''$	$\rightarrow$					$9q_2''''$	stop	$\rightarrow$	
9	$8\rightarrow q_1$	$\rightarrow$	$\square\rightarrow$	$\leftarrow$	$\rightarrow$	$\leftarrow$	$\leftarrow$	$8\rightarrow q_1''$	$\rightarrow$					$0\leftarrow$	stop	$\rightarrow$	
|	$\leftarrow$	$\rightarrow$	$\leftarrow$	$\leftarrow$	$\rightarrow$	$\leftarrow$	$\square\leftarrow q_2'$	$\leftarrow$	$\rightarrow$	$\alpha q_1'''$	$\beta q_0'''$	$\rightarrow q_0'''$	$\leftarrow q_0'''$	$\leftarrow$	$\square\leftarrow q_0'''$	$\rightarrow$	
$\square$		$\|\leftarrow q_0$	$\|\rightarrow q_1'$	q_2'	$\leftarrow q_3'$	$\|\rightarrow q_1'$	$*\rightarrow q_1'$		$\|\leftarrow q_0''$	$\rightarrow q_3'''$	$\leftarrow q_2'''$	$\rightarrow q_0''''$	q_2''''	$1q_2''''$	stop	$\leftarrow q_1''''$	
*	$\rightarrow q_2$	$\leftarrow q_0$		$\rightarrow q_1''$	$\rightarrow$	$\leftarrow q_3'$	$\leftarrow q_0$	$\alpha\rightarrow q_2$	$\square\leftarrow q_0''$								
α						$*\leftarrow q_0'''$				$\leftarrow$	$\rightarrow$	$\|\leftarrow$	$\square\rightarrow$				
β										$\leftarrow$	$\rightarrow$	$\square\leftarrow$	$\|\rightarrow$				

FIGURA 5.18. *Esquema funcional de una M.T. para el cálculo del m.c.d. de N_1 y N_2; información inicial $N_2 * N_1$, N_1 y $N_2 \in D$.*

4. DISEÑO DE UNA MÁQUINA DE TURING

En el apartado anterior hemos estudiado, partiendo de unos datos en la cinta, el funcionamiento de la máquina ya diseñada. *Otro problema mucho más complejo es el de especificar precisamente las características de una M.T. para resolver un tipo de problema, esto es, definir los conjuntos Q, E, las funciones f, g, y las configuraciones iniciales y finales de la cinta.* Es el problema del diseño, para el cual no existe otra metodología que el propio razonamiento lógico.

En este apartado se verá un nuevo ejemplo de M.T., resaltándose la estrategia seguida en el razonamiento para su creación. Se trata de diseñar una M.T. con cinta direccionable por etiqueta. (Después de estudiar este apartado, se le recomienda al lector reexamine las máquinas de Turing de las páginas anteriores a través de la óptica de diseño).

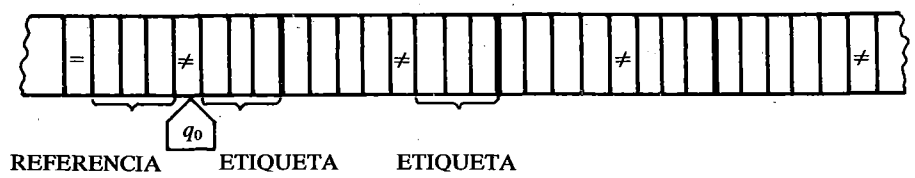


FIGURA 5.19.

Suponemos que la cinta es binaria (esto no resta generalidad al problema) y que sobre ella se tienen unos registros de longitud fija, separados por el símbolo ≠. Estos registros están constituidos por una etiqueta, igualmente de longitud fija, y la información, sin que entre ambas exista ninguna separación.

A la izquierda del primer registro, entre el símbolo ≠ de éste y otro =, se suponen escritos los bits correspondientes a la etiqueta del registro que se quiere localizar, conjunto de bits al que llamaremos «referencia». La cabeza de lectura se sitúa inicialmente sobre el primer ≠, con la máquina en un estado q_0 (figura 5.19).

El trabajo de la máquina consistirá en acudir a la referencia, recordar un bit y sustituirlo por otro símbolo (0 por α, 1 por β). Memorizado este bit, se desplazará la cabeza hacia la derecha en busca del primer registro no explorado y, dentro de él, del bit homólogo. Si coinciden bit de referencia y bit homólogo, éste último deberá ser anulado por la cabeza, que escribirá en su lugar α o β, volviendo a buscar el siguiente bit de referencia.

En el momento en que se encuentre una discrepancia, la máquina anulará todo el registro, volverá a =, restaurará todos los bits numéricos de la referencia en una pasada que le dejará sobre ≠ en condiciones de iniciar el proceso.

Si no se presentase la discrepancia, la cabeza no encontrará bits en la referencia, señal de que el registro ha sido localizado. La situación final es con la cabeza sobre el primer ≠ y toda la cinta escrita con símbolos literales (α, β) desde la referencia hasta la etiqueta del registro buscado (ambas inclusive). La primera información con bits es la deseada.

Veamos cómo se van asignando estados y construyendo el esquema funcional.

Con el estado q_0 se controla el retorno en el proceso de comparación de la referencia con una etiqueta, mediante un bucle a izquierdas, sin modificación de la información leída, hasta encontrarse con =, que conserva, iniciando un movimiento a derechas (véase 1.ª columna del esquema).

El estado q_1 permite buscar el primer bit no anulado de la referencia. Al encontrarlo, se memoriza, pasando a los estados q_2, q_3, según sea 0 ó 1 y escribiendo α o β, respectivamente.

Los estados q_2 y q_3, de movimiento a la derecha, deben ser insensibles a los bits que queden de la etiqueta controlando sendos bucles que terminan cuando la cabeza lee el símbolo $\neq$, pasando la máquina al estado q_4 o q_5, según el caso.

Precisamente con los estados q_4 y q_5 habrá de buscarse, en movimiento a derechas, el primer bit no anulado del registro, cambiarlo por un símbolo α o β, y, si coincidiera con el bit memorizado de la referencia, retornar (mediante el estado q_0) para buscar otro nuevo bit no anulado de la referencia (estado q_1). Si no hubiera coincidencia, la máquina debe concluir que la comparación con este registro ha dado resultado negativo, por lo que procederá a anular el resto del registro.

e_i \ q_j	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8
0	$\leftarrow$	$\alpha \rightarrow q_2$	$\rightarrow$	$\rightarrow$	$\alpha \leftarrow q_0$	$\alpha \rightarrow q_6$	$\alpha \rightarrow$	$\leftarrow$	$\rightarrow$
1	$\leftarrow$	$\beta \rightarrow q_3$	$\rightarrow$	$\rightarrow$	$\beta \rightarrow q_6$	$\beta \leftarrow q_0$	$\beta \rightarrow$	$\leftarrow$	$\rightarrow$
α	$\leftarrow$	$\rightarrow$			$\rightarrow$	$\rightarrow$		$\leftarrow$	$0 \rightarrow$
β	$\leftarrow$	$\rightarrow$			$\rightarrow$	$\rightarrow$		$\leftarrow$	$1 \rightarrow$
=	$\rightarrow q_1$							$\rightarrow q_8$	
$\neq$	$\leftarrow$	stop	$\rightarrow q_4$	$\rightarrow q_5$	$\rightarrow$	$\rightarrow$	$\leftarrow q_7$	$\leftarrow$	$\leftarrow q_0$

FIGURA 5.20. *Máquina de Turing con cinta direccionable.*

La anulación del resto del registro ha de controlarse por medio de un nuevo estado, q_6, con desplazamiento a derechas hasta llegar al término del registro, expresado por el símbolo $\neq$, en donde la M.T. cambiará de estado a q_7.

Se hace necesario un nuevo estado que llamamos q_7 para un retorno distinto del controlado por q_0, puesto que, habiendo resultado fallida una comparación referencia/etiqueta, es preciso restaurar los bits anulados de la referencia para iniciar un nuevo proceso de comparación de aquella con la siguiente etiqueta en la cinta. Tal restauración se hará gracias a un estado q_8 al que se pasa, junto con un desplazamiento a la derecha, cuando la cabeza, en su desplazamiento de retorno a izquierdas (bucle sin escritura) controlado por q_7, lee el símbolo =.

(*Observación*: Nótese que se ha diseñado el esquema de manera que no se toma en cuenta el símbolo □, que sólo afectaría aquí a los bordes).

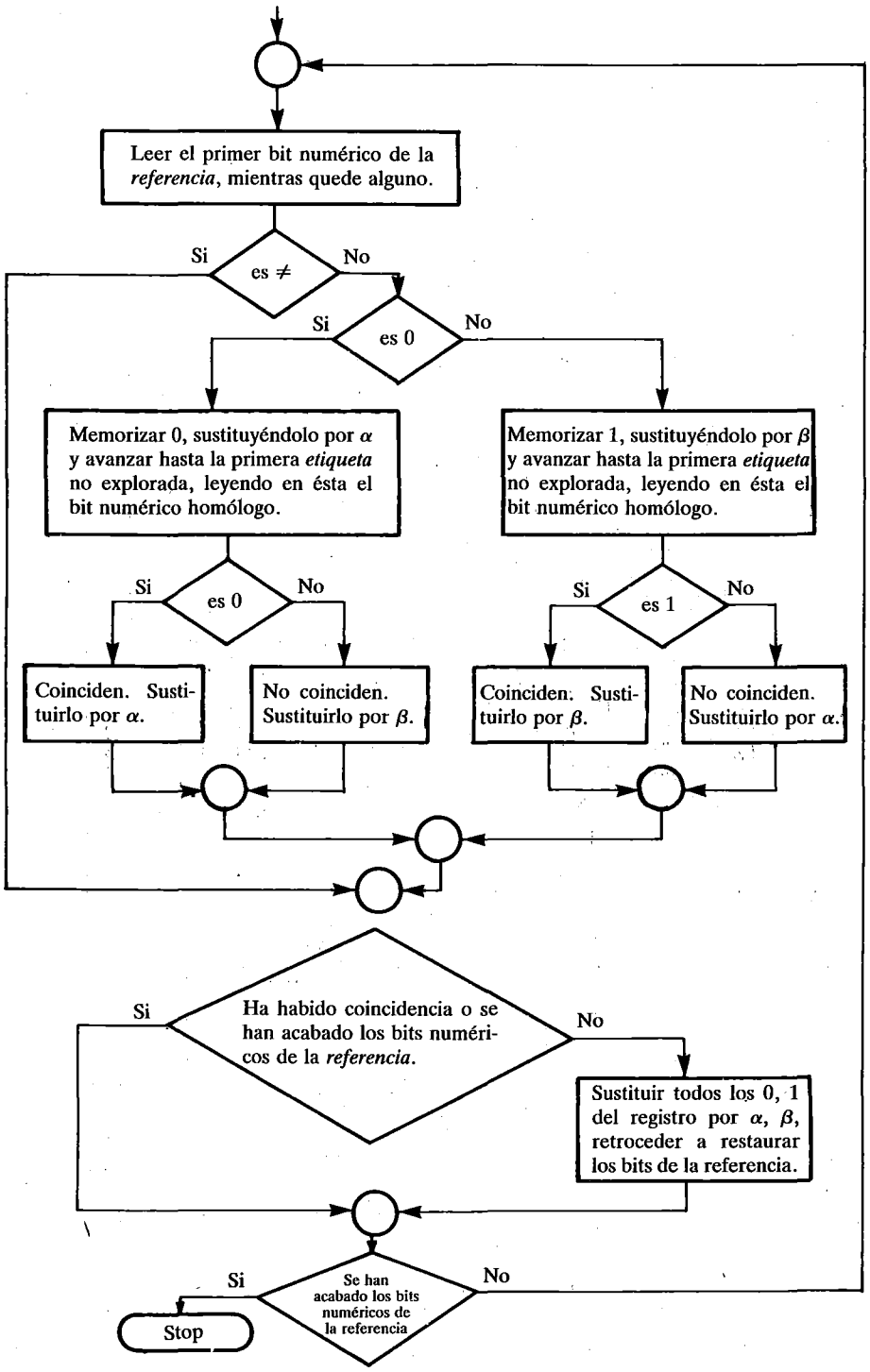

Leer el primer bit numérico de la *referencia*, mientras quede alguno.
Si
es ≠
No
Si
es 0
No
Memorizar 0, sustituyéndolo por α y avanzar hasta la primera *etiqueta* no explorada, leyendo en ésta el bit numérico homólogo.
Memorizar 1, sustituyéndolo por β y avanzar hasta la primera *etiqueta* no explorada, leyendo en ésta el bit numérico homólogo.
Si
es 0
No
Si
es 1
No
Coinciden. Sustituirlo por α.
No coinciden. Sustituirlo por β.
Coinciden. Sustituirlo por β.
No coinciden. Sustituirlo por α.
Si
Ha habido coincidencia o se han acabado los bits numéricos de la *referencia*.
No
Sustituir todos los 0, 1 del registro por α, β, retroceder a restaurar los bits de la referencia.
Si
Se han acabado los bits numéricos de la referencia
No
Stop

FIGURA 5.21.

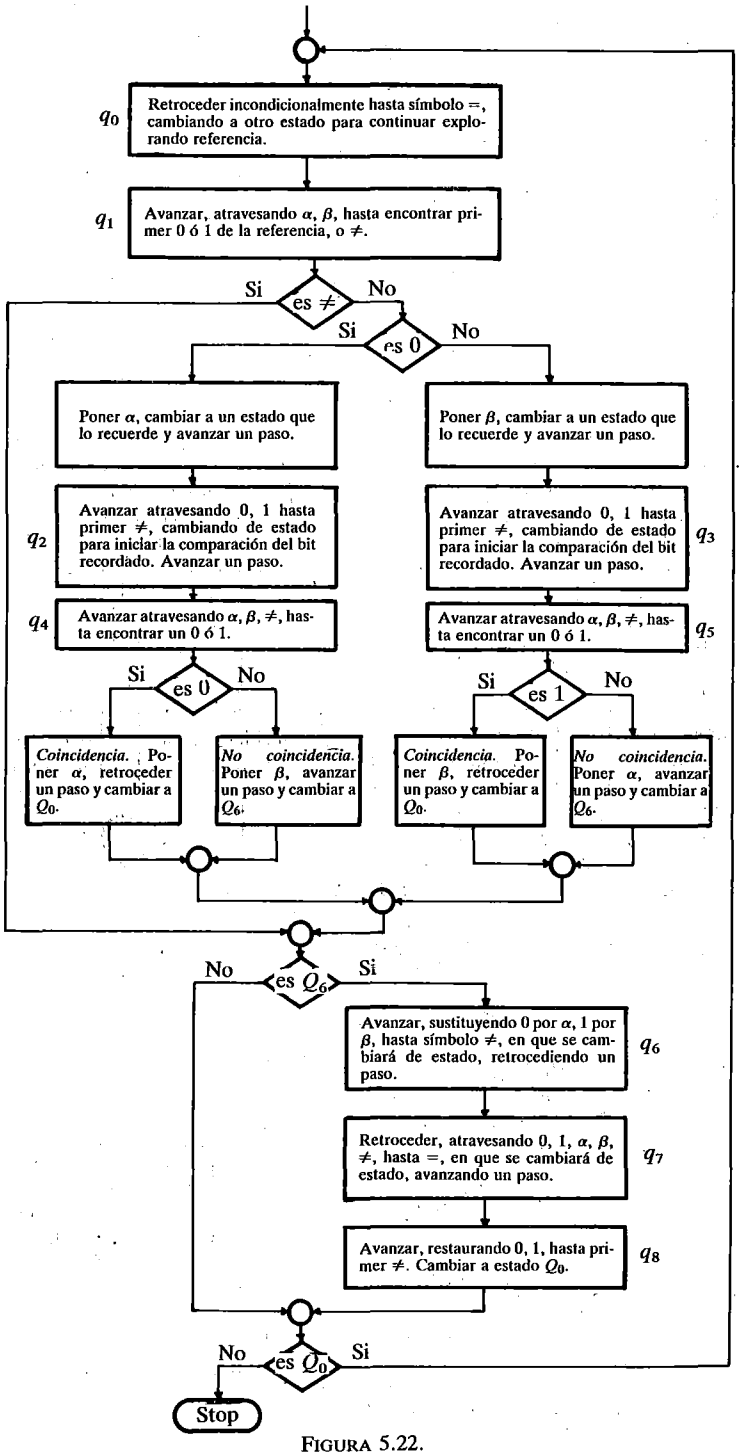

q_0
Retroceder incondicionalmente hasta símbolo =, cambiando a otro estado para continuar explorando referencia.
q_1
Avanzar, atravesando α, β, hasta encontrar primer 0 ó 1 de la referencia, o $\neq$.
Si
es $\neq$
No
Si
es 0
No
Poner α, cambiar a un estado que lo recuerde y avanzar un paso.
Poner β, cambiar a un estado que lo recuerde y avanzar un paso.
q_2
Avanzar atravesando 0, 1 hasta primer $\neq$, cambiando de estado para iniciar la comparación del bit recordado. Avanzar un paso.
q_3
Avanzar atravesando 0, 1 hasta primer $\neq$, cambiando de estado para iniciar la comparación del bit recordado. Avanzar un paso.
q_4
Avanzar atravesando α, β, $\neq$, hasta encontrar un 0 ó 1.
q_5
Avanzar atravesando α, β, $\neq$, hasta encontrar un 0 ó 1.
Si
es 0
No
Si
es 1
No
Coincidencia. Poner α, retroceder un paso y cambiar a Q_0.
No coincidencia. Poner β, avanzar un paso y cambiar a Q_6.
Coincidencia. Poner β, retroceder un paso y cambiar a Q_0.
No coincidencia. Poner α, avanzar un paso y cambiar a Q_6.
No
es Q_6
Si
Avanzar, sustituyendo 0 por α, 1 por β, hasta símbolo $\neq$, en que se cambiará de estado, retrocediendo un paso.
q_6
Retroceder, atravesando 0, 1, α, β, $\neq$, hasta =, en que se cambiará de estado, avanzando un paso.
q_7
Avanzar, restaurando 0, 1, hasta primer $\neq$. Cambiar a estado Q_0.
q_8
No
es Q_0
Si
Stop

FIGURA 5.22.

Como es natural, podría haberse seguido la técnica del organigrama para diseñar este proceso de cálculo. A fin de cuentas, el razonamiento lógico-literario que acaba de hacerse, el esquema funcional de la figura 5.20 y el organigrama de la figura 5.22 son equivalentes, si bien la segunda representación es, en el caso de la M.T., la más precisa y económica. Los organigramas de las figuras 5.21 y 5.22 son etapas sucesivas de un mismo razonamiento, el último más detallado y con especificación de estados. Para ser coherentes con los dos capítulos anteriores, los organigramas son estructurados con arreglo a la técnica de base allí expuesta.

Se le sugiere al lector la resolución del ejercicio siguiente: introducir en el esquema de la figura 5.20 las modificaciones necesarias y mínimas (sin ampliar el número de símbolos) para que la máquina siga realizando la misma función, pero deteniéndose en el símbolo $\neq$ que precede a la primera etiqueta que coincida con la referencia. *Nota*: No deje de observarse que la modificación pedida es del tipo de los ajustes que se han requerido en la composición de máquinas del apartado 3.3.

5. SIMULACIÓN DE MÁQUINAS DE TURING, MÁQUINA DE TURING UNIVERSAL Y OTRAS CONSIDERACIONES

5.1. Simulación de máquina de Turing por ordenador

Diseñar una M.T. para resolver un problema medianamente complejo es asunto difícil, pero desarrollar su funcionamiento a base de papel y lápiz es algo absolutamente tedioso. Lo primero podría tomarse como un desafío intelectual, como una prueba (necesaria y no suficiente) demostrativa del grado de capacidad lógica de un futuro especialista en informática, como un juego de sociedad en grupos humanos altamente intelectualizados o como entretenimiento inagotable de náufragos, solitarios y presos que no fueran delincuentes comunes. Lo segundo es un trabajo rutinario con una mecánica que, una vez conocida, no aporta nada especial al individuo e incluso puede resultar psicológicamente casi inaceptable en esta hora en que se dispone fácilmente de ordenadores.

Así pues, simular por ordenador la M.T. es una idea congruente con la eliminación de tarea rutinaria y que permite, dentro de ciertos límites, preservar y hasta acentuar los aspectos teóricos de la M.T. Entre otras cosas permite —es un ejemplo— ayudar a poner a punto el diseño de una M.T. como la del apartado 3.3.

Se han escrito algunos programas simuladores de M.T.'s, utilizando un lenguaje de programación con las siguientes características:

DN	Desplazamiento relativo de la cabeza *N* casillas a la derecha.
IN	Desplazamiento relativo de la cabeza *N* casillas a la izquierda.
E(*e*)	Escritura de *e* en la casilla situada bajo la cabeza, con $e \in E$.

$T(\alpha, e)$	Transferencia condicional. Si el símbolo leído es e, se transfiere control a la instrucción de etiqueta α, si no el programa se desarrolla en secuencia. Cuando no figure símbolo se tratará de una transferencia incondicional.
NOMBRE($A_1, A_2, \ldots, A_r$)	Nombre de un subprograma donde las A_k son etiquetas de instrucciones o símbolos de la M.T. NOMBRE es la etiqueta de la primera instrucción del subprograma.
END	Corresponde al estado «stop».

Para la representación en el ordenador de los alfabetos de la M.T. se emplearán los propios caracteres admitidos por el ordenador y el único problema es el que se deriva de la limitación física del ordenador respecto de la M.T.: la memoria del ordenador es finita y la memoria de la M.T. es infinita. Por consiguiente, los topes que el programa simulador establezca en la memoria central del ordenador serán los que marquen en cada simulación el espacio de validez de ésta.

Un ejemplo permitirá ver la forma en que habría que describir un esquema de M.T. con el lenguaje especificado más arriba (anotaremos el blanco por una B). Tomemos un caso muy sencillo, como el de la M.T. que sumaba dos números enteros no nulos en el alfabeto {|}. (Aquí se sustituirá el palote | por el 1, que es un símbolo admitido por cualquier ordenador). (Véase figura 5.23).

5.2. Máquina de Turing universal

Todas las M.T.'s pueden ser simuladas por otra máquina de Turing llamada M.T. universal, siempre que se le de a ésta la información necesaria sobre la primera, a saber:

- Contenido inicial de la cinta.
- Posición inicial de la cabeza.
- Estado inicial.
- Esquema funcional, programa o funciones f y g (como se prefiera llamar).

Los conceptos necesarios para definir una M.T. universal son éstos. Se demuestra que:

1.° Cualquier M.T. concreta puede ser simulada por otra con alfabeto binario ({0, 1}, {□, |} o los dos símbolos preferidos de cada uno. En general, los símbolos de un alfabeto cualquiera son codificables por paquetes de unos, paquetes distinguibles entre sí por paquetes convenidos de ceros).
2.° Las posibilidades de una M.T. no se restringen por el hecho de que su cinta sea ilimitada sólo por un extremo.

Una M.T. universal dispondrá de una cinta ilimitada por ambas partes, con un alfabeto externo $\{0, 1, \alpha, \beta, \neq, \neq, *, b\}$ correspondiéndose α con 0 y β con 1. El

```
M0   T(M3, 1)
     T(M4, B)
     T(M5, *)
M1   T(M6, 1)
     T(M7, B)
     T(M8, *)
M2   T(M9, 1)
     T(M10, B)
     T(M11, *)
M3   E(B)
     D1
     T(M2)
M4   D1
     T(M0)
M5   E(B)
     T(M20)
M6   I1
     T(M1)
M7   D1
     T(M0)
M8   I1
     T(M1)
M9   D1
     T(M2)
M10  E(1)
     T(M1)
M11  D1
     T(M2)
M20  END
```

FIGURA 5.23.

símbolo * representa la posición de la cabeza simulada, *b* es el blanco y al iniciar y terminar la simulación de un movimiento de la M.T. simulada sólo habrá símbolos numéricos {0, 1} en la M.T. universal. La información de datos de la cinta simulada se sitúa a partir de una posición a la izquierda de ≠ y la información sobre sus estados y funciones a la derecha de ese mismo símbolo. Esta es una versión de una máquina de Turing universal.

Con estos elementos se puede construir una M.T. universal que, combinando las posibilidades de una M.T. de cinta direccionable (apartado 4) y de una M.T. transcriptora de información, puede simular todas las M.T.'s definidas en un alfabeto binario.

5.3. Otras consideraciones

Ya se ha visto que la propiedad de cualquier M.T. de contar con una cinta infinita —característica eminentemente teórica— le confiere una variedad (en el sentido

cibernético de esta palabra) adaptable potencialmente al control de cualquier circunstancia de tratamiento de información.

Es conocido que, cuando se recorre un terreno más práctico como es el del uso de los ordenadores, el contar con una memoria principal grande es condición *sine qua non* para poder tratar, con una determinada velocidad, mayor variedad de problemas y problemas más complejos. Aumentando la capacidad de la memoria principal se incrementa el volumen y complejidad de los problemas que es posible tratar con una máquina concreta y a la inversa. Ahora bien, razones tecnológicas y económicas impiden aumentar todo lo que se desearía la capacidad de las memorias, para un precio y un instante histórico precisos. En todos los casos, la velocidad de la memoria representa un freno a la velocidad de que hace gala el procesador o unidad de cálculo, que tiene la virtud, además, a cada nuevo diseño de ordenador, de poder ejecutar un repertorio más amplio de instrucciones distintas. Lo cierto es que, incluso con una memoria finita, el incremento del número de instrucciones distintas lo que hace es añadir versatilidad y velocidad al tratamiento de los problemas, porque un número extraordinariamente reducido de instrucciones distintas basta para describir cualquier cálculo en este tipo de máquinas.

El tipo de máquinas que definió Turing (antes que se diseñara el primer ordenador, recuérdese) es un ordenador ideal, ya que no depende de ninguna característica física ni le preocupa la velocidad u otra clase de eficiencia. Visto desde la perspectiva

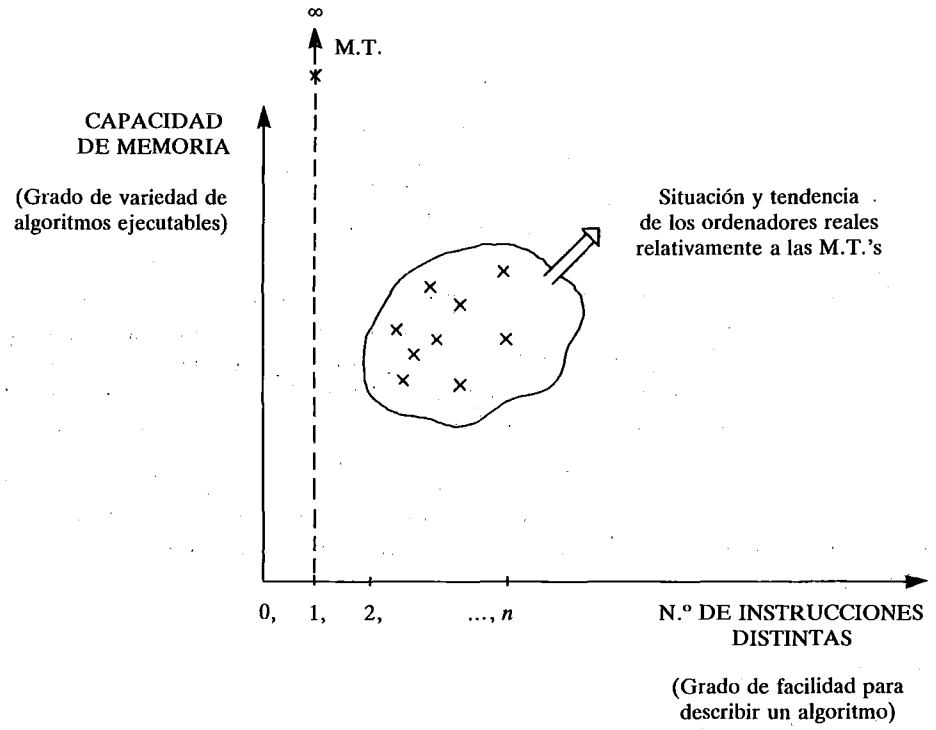


FIGURA 5.23.

de los ordenadores —esto es, a posteriori— *es un ordenador con una memoria infinita y una sola instrucción distinta*, la quíntupla, tantas veces utilizada en este capítulo. Así pues, no solamente goza de la virtud teórica de poder ejecutar cualquier algoritmo, como, por ejemplo, simular a un ordenador moderno que es una máquina más compleja dotada de un rico repertorio de instrucciones, sino que esto lo hace mediante los pasos más elementales de que se tiene noticia.

La genialidad de Turing consistió en poner su invento fuera de las limitaciones espacio-temporales (espacio para la información, infinito; tiempo, el que sea, pero un número finito de pasos). En tales circunstancias, una sola M.T. es capaz de reproducir el funcionamiento de todas las demás, siempre que disponga de la descripción de las mismas.

Esta última idea, inmanente a la M.T.U., de que una máquina pueda desarrollar procesos más complejos que los que su propia estructura parece permitirle, a condición de que se le suministre la información adecuada, despertó gran interés y ha sido trasladada por analogía al campo de la reproducción biológica para intentar explicar la construcción de la vida y su mantenimiento a partir de las informaciones genéticas.

6. SUCEDÁNEOS DE LA MÁQUINA DE TURING

Al objeto de que el lector tenga también noticia de ello, conviene que conozca la existencia de ciertos derivados de la versión clásica de M.T., que es con la que hemos venido trabajando en este capítulo. Son, entre otros:

a) M.T.'s con sólo dos de las tres salidas posibles e_k, m_h, q_p.
b) M.T.'s con cinta limitada por un extremo.
c) M.T.'s con más de una cinta y más de una cabeza.
d) M.T.'s no deterministas.

Se demuestra que ninguna de ellas restringe las posibilidades de la M.T. definida en el apartado 2. Más adelante, mencionaremos la utilidad teórica de algunos de estos tipos de M.T.'s, a propósito de ciertas cuestiones planteadas en el estudio de la complejidad algorítmica (capítulo 7).

7. RESUMEN

Una máquina de Turing es un artefacto computador constituido por un autómata finito que controla una cinta infinita. Cada paso en el cálculo de una M.T. consiste en escribir un símbolo en la cinta, desplazar la cabeza de lectura/escritura una casilla a la derecha o a la izquierda y asumir un nuevo estado. La acción concreta de cada paso viene determinada por el estado en curso de la máquina y por el símbolo que lee en ese instante la cabeza.

El funcionamiento de una M.T. se especifica completamente *por una lista de quíntuplas* e_i, q_j, e_k, m_h, q_p, donde están todas las combinaciones e_i, q_j que permiten los alfabetos externo E e interno Q, *por la cinta con la información inicial* y por la

situación inicial de la máquina expresada *por la posición de la cabeza y el estado del autómata*. A la lista de quíntuplas se le llama esquema funcional o programa de la M.T.

Distintos ejercicios a lo largo del capítulo han buscado familiarizar al lector con el funcionamiento y las diferentes formas de representar los resultados de una M.T. Una M.T. es capaz de realizar sólo operaciones muy elementales, pero secuencias adecuadas de estas operaciones pueden llegar a componer una amplia variedad de operaciones de manipulación de bloques. Estas últimas operaciones comprenden: formar copias de bloques especificados, sustituir un bloque por otro y comparar bloques. Empleando estas operaciones como subprogramas, es posible diseñar M.T.'s que realizan cálculos muy complejos.

El modelo de M.T. que se ha presentado puede modificarse en varios sentidos sin alterar sus posibilidades últimas como máquina computadora. Tal vez convenga subrayar que, *dado un algoritmo a ejecutar por una máquina de Turing*, en el diseño de ésta —supuesto escogido un modelo específico de M.T. (por ejemplo, con una sola cinta ilimitada por ambos extremos)— *se presenta, en principio, la disyuntiva de disminuir el cardinal del alfabeto externo a costa de aumentar el del interno, o viceversa*. Uno de los resultados más interesantes de esta propiedad es que siempre es posible simular una M.T. por otra M.T. definida sobre un alfabeto externo binario.

Por último, debe resaltarse la *máquina de Turing universal, diseñada para ejecutar un algoritmo de simulación de todas las otras M.T.'s que poseen su misma estructura*. Las especificaciones completas de la M.T. simulada y sus datos de trabajo figuran como datos en la cinta de la M.T.U.

8. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

La máquina de Turing se conoció, como ya se dijo en el primer capítulo de este tema, a raíz de un trabajo fundamental del matemático inglés A. M. Turing, publicado en 1936. Este trabajo, junto con aportaciones previas de Gödel y otras coetáneas, ha significado un impacto para la ciencia matemática, además de constituirse en argumento básico para lo que la informática puede tener de ciencia.

La confección de este capítulo se ha visto ayudada por algunos artículos y libros publicados casi siempre en los últimos años de la década de los sesenta y toda la década de los setenta. En particular, hay que mencionar el libro de Gross y Lentin (1967), del que se ha tomado la definición de cálculo de una máquina de Turing.

En Corge (1975) hemos encontrado interesante la explicación del proceso de cálculo, iniciada en el subapartado 3.2 con el ejemplo del cálculo del m.c.d., y también el diseño de las máquinas M.T.0, M.T.1 y M.T.3 del apartado 3.3, que hemos adaptado, corregido y simulado.

El ejemplo de diseño de una máquina de Turing con cinta direccionable del apartado 4 se encuentra en Scala, Minguet (1974). Asimismo, la versión de máquina de Turing universal del subapartado 5.2. Un desarrollo de la M.T.U. un poco diferente, aunque simple y bastante detallado, se hallará en el capítulo 2 de Hennie (1977).

Los programas simuladores de máquinas de Turing han tenido su época. Por

nuestra parte, hemos utilizado la descripción de Curtis (1965). Delgado, cuando terminaba sus estudios, desarrolló bajo nuestra dirección un simulador escrito en Fortran, con el cual pueden simularse máquinas de cierta complejidad. Con él se han simulado las máquinas de los subapartados 3.2 y 3.3.

A título de curiosidad, tal vez merezca la pena reflexionar acerca de la trascendencia del concepto de máquina de Turing universal acercándose a especulaciones en otros ámbitos disciplinares ajenos a la informática. En tal sentido, cabe mencionar la analogía que habla de construir vida compleja a partir de un núcleo básico y reducido de mecanismos sencillos e informaciones (Singh, 1976, cap. 13).

El lector que necesite o desee una presentación más formal y amplia de la máquina de Turing dispone de varias opciones en la bibliografía. Pero si tuviéramos que recomendar aquí un solo texto nos inclinaríamos por el libro de Hopcroft y Ullman (1979), por la simple razón de que, dejando a un lado su incuestionable calidad, tiene la ventaja de que trata conjuntamente la mayor parte de los temas abordados en nuestro libro.

El primero de estos autores ha escrito un artículo de alta divulgación sobre máquinas de Turing en la revista *Scientific American*, posteriormente traducido al castellano (Hopcroft, 1984), que consideramos muy recomendable. Se extiende también en aspectos básicos de computabilidad y complejidad, sobre los que versarán nuestros próximos capítulos.

9. EJERCICIOS

9.1. Completar la definición del autómata finito de una máquina de Turing de la que se conoce lo siguiente:

— El contenido inicial de la cinta es un número entero representado en el alfabeto {|}, seguido de un «*».
— El contenido final de la cinta será el mismo número en binario natural representado en {0, 1}, seguido de un «*».
— Conjunto de símbolos de entrada {□, |, 0, 1, *}.
— Conjunto de movimientos de la cabeza {→, ←, ↔}.

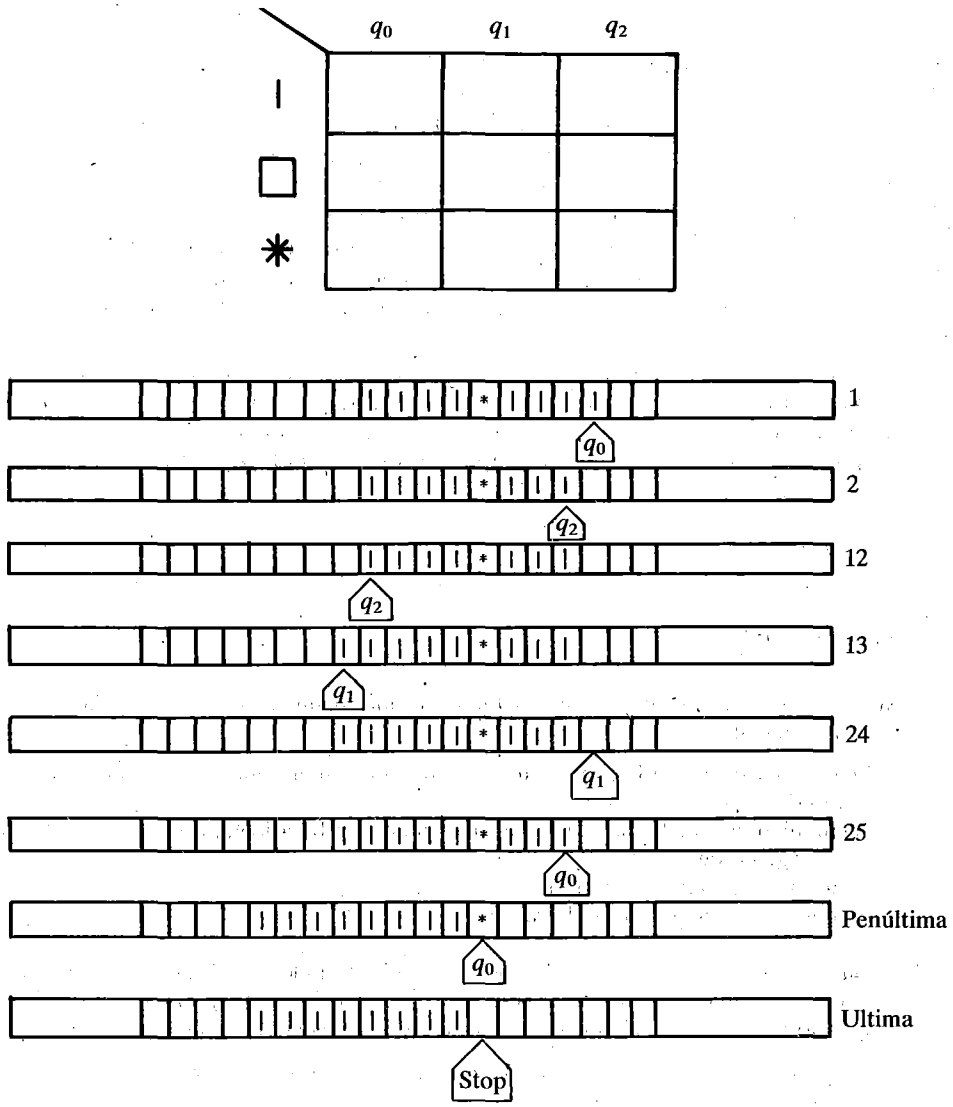
9.2. Escribir, *exclusivamente*, el contenido de la cinta (utilizando el formato de descripción instantánea) de una máquina de Turing para la resolución del algoritmo de Euclides (subapartado 3.2) con dos números enteros en el alfabeto | *en los instantes* que se indican:

1.° La máquina está en q_2, al que acaba de cambiar por segunda vez.
2.° La máquina está en q_2, al que acaba de cambiar por cuarta vez.
3.° La máquina está en q_1, al que acaba de cambiar por tercera vez.
4.° La máquina está en q_3, al que acaba de cambiar por segunda vez.

La máquina se inicia con los números 9 y 3 en la cinta, respectivamente, a izquierda y derecha del asterisco.

9.3. Obtener el esquema funcional de una máquina de Turing que pasa por las configuraciones de cinta abajo indicadas. Nótese que a la derecha de cada una de las configuraciones

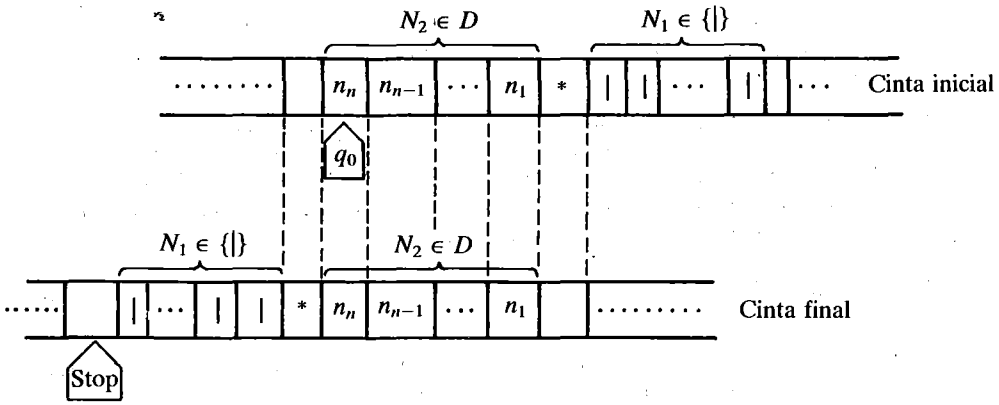
se escribe el instante correspondiente. Razonar por escrito la obtención del esquema funcional.



9.4. En la figura 5.14 puede consultarse el esquema funcional de una máquina de Turing que permuta los números N_2 (representado en el alfabeto decimal) y N_1 (expresado en palotes).

Partiendo de la misma posición: contenido inicial de la cinta, estado interno y posición de

la cabeza, se pide introducir las modificaciones pertinentes en el antecitado esquema funcional para que la máquina realice la siguiente transformación:



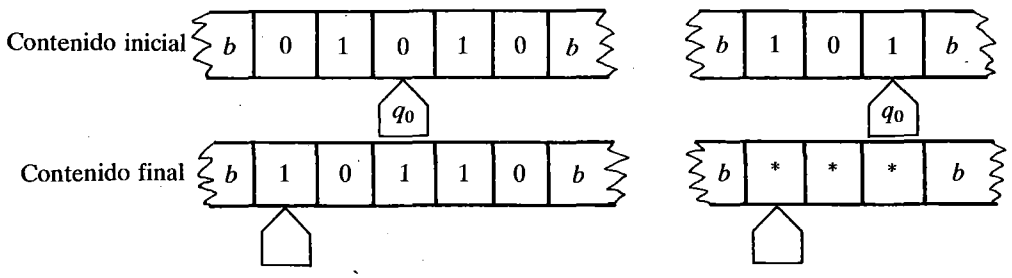
9.5. Diseñar una máquina de Turing cuya especificación es la siguiente:

— Contenido inicial de la cinta: Un número entero positivo representado en complemento a dos y delimitado por casillas en blanco.
— Contenido final de la cinta: La representación en complemento a dos del mismo número pero negativo.
— Posición inicial de la cabeza: Sobre cualquier símbolo de la representación del número positivo.
— Posición final de la cabeza: Sobre la casilla que contiene el bit de signo.

Notas:

1. El número de casillas que ocupa la representación del número es desconocido. Lo define el contenido inicial de la cinta.
2. En caso de que inicialmente la cinta contenga la representación de un número negativo, la máquina sustituirá los ceros y los unos por asteriscos, para indicar situación errónea.

Ejemplos:



9.6. Se propone introducir las modificaciones necesarias en la máquina de Turing con cinta direccionable, cuyo diseño se explica en el apartado 4, en orden a tomar en cuenta las siguientes condiciones:

— «Ha de preverse un final al cálculo cuando ninguna de las etiquetas de los registros de la cinta coincide con la referencia. Para ello supondremos que en la cinta inicial hay un asterisco (*) inmediatamente a la derecha del último registro».

Se pide introducir en el esquema de la figura 5.2 los cambios pertinentes, añadiendo para ello si es preciso un mínimo número de nuevos estados, de manera que la máquina se detenga sobre el asterisco en el caso en que se produjera la situación descrita más arriba entre comillas. Hágase el esquema funcional completo, recuadrando claramente los cambios o añadidos con respecto al esquema de la figura señalada.

9.7. Diseñar una máquina de Turing que sume enteros no negativos en las siguientes condiciones:

a) Los números n enteros no negativos, considerados como datos en la cinta, se representarán en el alfabeto $\{|\}$ con $(n + 1)$ palotes.
b) Los dos números, n_1 y n_2, que suma la M.T. se presentan en la cinta, de izquierda a derecha, separados por un cero (o blanco, □) y el resto a ceros (o blancos).
c) La M.T. inicia su operación en el estado q_0 con la cabeza sobre el primer palote a la izquierda.
d) El resultado de aplicar la M.T. será un bloque de $(n_1 + n_2)$ palotes.
e) La M.T. termina su operación con la cabeza sobre el primer palote a la izquierda.

Capítulo 6
MAQUINA DE TURING: ALGORITMOS Y COMPUTABILIDAD

1. INTRODUCCIÓN

Este capítulo aborda de manera muy esquemática la noción de computabilidad (calculabilidad) en el sentido de Turing (y conceptos relacionados), que formaliza la noción un tanto intuitiva de algoritmo de las definiciones de autores recogidas en el segundo capítulo. Respecto a la definición formal de algoritmo del mismo capítulo ésta es una alternativa más fértil, pues se expresa en términos de una máquina que, no por ser ideal o conceptual, es menos concreta.

2. FUNCIÓN COMPUTABLE Y FUNCIÓN PARCIALMENTE COMPUTABLE

2.1. Hipótesis de Church o de Turing

La idea intuitiva de procedimiento efectivo para desarrollar un cálculo es la misma que la de algoritmo. Pues bien, según la hipótesis de Turing/Church: *la noción intuitiva informal de un procedimiento efectivo sobre secuencias de símbolos es idéntica a la de nuestro concepto preciso de un procedimiento que puede ser ejecutado por una máquina de Turing.*

No existe prueba formal de esta hipótesis pero, hasta la fecha, siempre que en la teoría de las *funciones recursivas** ha sido intuitivamente evidente que existía un algoritmo, ha sido posible diseñar una M.T. para ejecutarlo.

---

* Funciones recursivas y funciones computables son equivalentes. El concepto de computabilidad se debe a Turing y el de recursividad a Kleene.

2.2. Función computable

Asociamos una función $F_Z^{(r)}$ con una máquina de Turing Z, definiendo $F_Z^{(r)}(n_1, n_2, \ldots, n_r)$ como el número $\langle \gamma_p \rangle$ de unos que hay en la cinta cuando Z se detiene, habiéndose iniciado en la siguiente situación.

$$\ldots 000\underbrace{11 \ldots 1}_{n_1+1}0\underbrace{11 \ldots 1}_{n_2+1}0\underbrace{11 \ldots 1}_{n_3+1}0\underbrace{1101 \ldots 1}_{n_r+1}0\ldots \qquad (1)$$

Si la máquina nunca se detuviera, $F_Z^{(r)}$ no estaría definida para la r-upla considerada.

Intentemos ver más de cerca el significado de los elementos que intervienen en esta función.

En el apartado 5.2 del capítulo anterior se estableció que un alfabeto binario es suficiente —a costa de aumentar el número de estados— para ejecutar cualquier cálculo con una M.T. Con mayor razón podrán tratarse los números enteros no negativos, utilizando un alfabeto $\{0, 1\}$ o $\{\square, |\}$. Escojamos la primera de estas dos representaciones.

Se representará un número entero no negativo, n, por $(n + 1)$ unos y a este bloque lo llamaremos $\overline{n}$, para distinguirlo.

$$\overline{0} = 1,\ \overline{1} = 11,\ \overline{2} = 111,\ \ldots,\ \overline{5} = 111111,\ \ldots$$

El cero hará las veces de separador, por lo que una r-upla $(n_1, n_2, \ldots, n_r)$ se convendrá en representar como en (1), lo que abreviadamente equivale a $\overline{n}_1 0 \overline{n}_2 0 \overline{n}_3 0 \ldots 0 \overline{n}_r$. Si ésta es una información A en la cinta de una máquina de Turing Z, Z será aplicable o no a A. En el primer caso, Z produce el cálculo

$$\gamma_1, \gamma_2, \ldots, \gamma_p, \text{ donde } \gamma_1 = q_0 \overline{n}_1 0 \overline{n}_2 0 \overline{n}_3 0 \ldots 0 \overline{n}_r$$

y el número entero $\langle \gamma_p \rangle$ es una función que depende de la máquina Z y de la r-upla inicial. Se escribe

$$\langle \gamma_p \rangle = F_Z^{(r)}(n_1, n_2, \ldots, n_r) \qquad (2)$$

que es una función de valores enteros no negativos, definida sobre N^r o sobre una parte de N^r (función parcialmente definida).

Si se hace ahora al revés, es decir, se parte de una función definida sobre N^r o sobre una parte de N^r se tienen las siguientes definiciones.

2.2.1. *Definición de función parcialmente computable*

Se dice que una función f definida sobre una parte de N^r, es parcialmente computable, para expresar que existe una máquina de Turing Z tal que, para toda r-upla a la que corresponda un valor de f, se tenga:

$$f(n_1, n_2, \ldots, n_r) = F_Z^{(r)}(n_1, n_2, \ldots, n_r) \tag{3}$$

2.2.2. *Definición de función computable*

Se dice que una función f es computable para expresar que está definida sobre N^r y es parcialmente computable.

2.3. Ejemplos

2.3.1. La función $f(n_1, n_2) = n_1 + n_2$ definida sobre las parejas de enteros no negativos, que es calculable (computable) en el sentido habitual de la palabra, lo es también en el sentido de la definición 2.2.2. Puede construirse una máquina de Turing Z, tal que

$$n_1 + n_2 = \mathrm{SUM}_Z^{(2)}(n_1, n_2) \tag{4}$$

y teniendo en cuenta que $n_1 + n_2 = \overline{n}_1 + \overline{n}_2 - 2$

e_i \ q_j	q_0	q_1	q_2	q_3	q_4	q_5	q_6
0		$\rightarrow q_5$	q_4	$1 \leftarrow q_4$	$\rightarrow$ stop	$\leftarrow q_6$	
1	$0 \rightarrow q_1$	$0 \rightarrow q_2$	$0 \rightarrow q_3$	$\rightarrow$	$\leftarrow$	$\rightarrow$	$0 \leftarrow q_4$

En el momento de detenerse la máquina (4) contiene $(n_1 + n_2)$ unos, cualesquiera que sean n_1 y n_2. La función f está definida sobre N^2 y es parcialmente calculable, luego es una función calculable.

2.3.2. La función $g(n_1, n_2) = n_1 \div n_2$, sustracción definida sobre el subconjunto $n_1 \geqslant n_2$ puede demostrarse que es parcialmente calculable sin más que construir la correspondiente M.T., situando, por ejemplo, el número n_1 a la izquierda y el n_2 a la derecha de un cero separador.

La función $n_1 \div n_2$, parcialmente computable podría prolongarse en una función computable $n_1 \div n_2$, definida sobre N^2, así:

$$n_1 \div n_2 = n_1 - n_2 \quad \text{si } n_1 \geq n_2$$
$$n_1 \div n_2 = 0 \quad \text{si } n_1 < n_2$$

3. NUMERABILIDAD DE LA COLECCIÓN DE TODAS LAS M.T.'S

Una M.T. está especificada por una lista de quíntuplas e_i, q_j, e_k, m_h, q_p, que forman un conjunto finito. Los valores posibles de *i, j, k, h, p* son todos numerables. Así pues, la colección de todas las quíntuplas es numerable. Consiguientemente, las listas de quíntuplas son numerables y, por ende, los autómatas por ellas representados.

Esto significa que *las M.T.'s pueden ordenarse numéricamente*. El problema es cómo escoger un código tal que, dado un número, puedan determinarse las especificaciones de la M.T. correspondiente, si la hubiere, y viceversa.

3.1. Números de Gödel

El método de establecer un código de esta naturaleza, que consiste en *numerizar lo no numérico*, fue propuesto por Gödel antes de que existieran las máquinas de Turing.

Kurt Gödel, lógico eminente desaparecido hace pocos años, escribió en 1930 un artículo que, cuando se publicó en una revista alemana en 1931, produjo el efecto de un paquete de dinamita colocado precisamente en la base de la viga maestra de los fundamentos de la matemática. Fundamentos que, con celo encomiable, estaban renovando los matemáticos de la época, con Hilbert a la cabeza.

Para el lector que no conozca en qué contexto propuso Gödel su técnica de codificación, la preocupación matemática del momento consistía en probar la consistencia de la teoría axiomática de conjuntos, para lo cual Hilbert propuso un programa completo. Pues bien, Gödel probó dos cosas:

1.° Si la teoría axiomática de conjuntos es consistente, existen teoremas que no pueden ser probados ni refutados.
2.° No existe ningún procedimiento constructivo que pruebe que la teoría axiomática de conjuntos es consistente.

El primer resultado prueba que los problemas no siempre son solubles, ni siquiera en principio; el segundo destrozó el programa de Hilbert para probar la consistencia.

En la teoría axiomática de conjuntos se utilizan símbolos con los que se forman expresiones o cadenas, que son objetos metamatemáticos. Para demostrar sus teoremas, Gödel se sirvió de una codificación numérica de dichas cadenas, que es el asunto que interesa aquí.

Supongamos que se cuenta con los símbolos que a continuación se reseñan. Cada uno de ellos es codificable por los números naturales en la manera indicada:

$$+ \quad - \quad \times \quad \div \quad (\quad) \quad = \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad a \quad b \quad \ldots \quad x_i \quad \ldots$$

$$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad 18 \quad 19 \quad \ldots \quad y_i \quad \ldots$$

Una de las formas más simples para representar cadenas arbitrarias $x_0, x_1, \ldots, x_n$

compuestas con estos símbolos, por ejemplo, sería la de utilizar el número natural

$$p_0^{\,y_0}\ p_1^{\,y_1}\ \ldots\ p_n^{\,y_n}$$

siendo p_i el i-ésimo número primo.

La fórmula

$$4 + 7 = 11$$

se codificaría por el número $2^{12} \cdot 3^1 \cdot 5^{15} \cdot 7^7 \cdot 11^9 \cdot 13^9$.

La fórmula

$$a(a - 1) = aa - a$$

tendría el siguiente número de Gödel: $2^{18} \cdot 3^5 \cdot 5^{18} \cdot 7^2 \cdot 11^9 \cdot 13^6 \cdot 17^7 \cdot 19^{18} \cdot 23^{18} \cdot 29^2 \cdot 31^{18}$.

La cadena $\div\ -\ +$ se codifica por $2^4 \cdot 3^2 \cdot 5^1$ igual a 720. Pero, a causa de la *unicidad de la factorización en números primos*, una cadena puede reconstruirse a partir de su código. Esto es, el número 720, factorizado, nos da $2^4 \cdot 3^2 \cdot 5^1$ que corresponde a la cadena $\div\ -\ +$, pues los símbolos que componen ésta tienen los códigos 4, 2 y 1 respectivamente.

En resumen, *cada cadena tiene un número, y números diferentes corresponden a cadenas diferentes*. Disponiendo éstas según el tamaño de sus números de Gödel, puede verse que el conjunto de cadenas es numerable.

3.2. Catálogo de las M.T.'s

Todas las quíntuplas que constituyen el esquema funcional de una máquina de Turing forman una cadena de símbolos codificable por un número de Gödel. O, lo que es lo mismo, cada M.T. recibe su número de Gödel z y con él puede catalogarse numéricamente la colección de las máquinas de Turing.

Tratándose de un procedimiento sistemático es posible diseñar M.T.'s con las siguientes propiedades:

a) *Existe una M.T. que es aplicable a toda secuencia escrita en su cinta (representativa de la lista de quíntuplas), transformándola en su número de Gödel.*

b) *Existe una M.T. que, para todo número escrito en su cinta, proporciona una de estas dos respuestas:*

— «no existe secuencia correspondiente a este número»;

— «la secuencia correspondiente a este número es...» (la lista de quíntuplas).

4. DE NUEVO, LA MÁQUINA DE TURING UNIVERSAL

Se ha visto en el capítulo anterior que una M.T.U. puede simular a todas las otras si se le suministran los datos y las especificaciones de éstas. En el punto 3 acabamos de

ver que todas las máquinas pueden ser catalogadas unívocamente, y que puede diseñarse una M.T. tal que, dado un número de Gödel z, entrega como resultado la lista de quíntuplas de una M.T., si la hay asociada a ese número. Con esta última perspectiva, es factible pensar en una M.T.U. a la que baste suministrar solamente el número de catálogo de las M.T.'s que se quiera simular. ¿Qué tiene que ver con las funciones computables? Veámoslo.

Llámese X al elemento $(n_1, n_2, \ldots, n_r) \in N^r$. A toda función $f(X)$ parcialmente computable puede asociársele el número de Gödel z de la M.T. que calcula dicha función (es evidente que puede haber varias máquinas que calculen la misma función).

Se define una función $Q_z^{(r)}$:

— Si z es el número de Gödel de una máquina Z que calcula la función parcialmente calculable $f(X)$, entonces $Q_z^{(r)}(X)$ coincide con $f(X)$.
— Si z no es el número de Gödel de ninguna máquina, entonces $Q_z^{(r)}(X)$ toma el valor cero.

Está claro que $Q_z^{(r)}$ es una función cuyo dominio es N^{r+1} y que toma sus valores en N. Puede calcularse por composición de dos máquinas de Turing (véase apartado 3.3 del capítulo 5).

A una M.T. como la definida en el apartado 3.2.b) se le suministra el número z. Si la respuesta es *no*, $Q_z^{(r)}$ toma el valor *cero*. Si la respuesta es *sí*, se alimenta otra M.T. con la secuencia producida por la anterior M.T. que será el programa de cálculo de Z, y con el elemento o dato X. El resultado de la ejecución de esta última máquina es $Q_z^{(r)}(X)$, igual a $f(X)$.

Esto es lo mismo que decir que existe una máquina de Turing universal* U, tal que

$$F_Z(X) = F_U(z, X) \tag{5}$$

para todas las M.T.'s Z y todos los enteros z.

4.1. Teorema

Las funciones $Q_z^{(r)}(X)$ son funciones parcialmente computables.

Una M.T. que calcula la función $Q_z^{(r)}(X)$ se llama universal: inscribiendo en su cinta el número z adecuado, ella puede calcular la función parcialmente computable correspondiente a este número.

5. CONJUNTOS RECURSIVOS Y RECURSIVAMENTE NUMERABLES

En este apartado se va a considerar muy esquemáticamente la aplicación de los conceptos de computabilidad a dos importantes clases de conjuntos de números naturales: los conjuntos recursivos y los conjuntos recursivamente numerables.

---

* El análogo de U es el ordenador cargado de programas al que se le da el nombre z de uno de éstos y los datos sobre los que tiene que operar.

Hay dos procesos fundamentales de cálculo que pueden asociarse a un conjunto específico G de números naturales. Uno es el proceso de determinar si, dado cualquier número natural x, éste pertenece a G. El otro es el proceso de generar, uno a uno, todos los elementos de G. Ambos procesos están relacionados, pero no son equivalentes.

Para expresar qué es un conjunto de alguna de estas clases debe recordarse previamente la definición de *función característica de un conjunto*.

Sea $G \subset N$. La función característica de G, C_G, se define así:

$$C_G(x) = \begin{cases} 1, & \text{si } x \in G \\ 0, & \text{si } x \notin G \end{cases}$$

De la misma manera se define la función característica de un conjunto $G \subset N^r$.

5.1. Conjunto recursivo

Decir que un conjunto es recursivo es expresar que su función característica es computable*.

Esta definición equivale obviamente a decir que existe una M.T. que, aplicada a la información X, la transforma en 1 o en 0. O, en otras palabras, que existe un procedimiento efectivo para decir si un elemento X pertenece o no a ese conjunto.

5.2. Conjunto recursivamente numerable

Es el dominio de una (por lo menos) función parcialmente computable*.

Dicho en otra forma, significa que existe un procedimiento efectivo para generar sus elementos, uno detrás de otro.

Por ejemplo, el conjunto de los cuadrados de los números enteros es recursivamente numerable —se toman los números 1, 2, 3, 4, ... y se van elevando al cuadrado sucesivamente. También es recursivo— dado un número entero cualquiera, se descompone en sus factores primos, viéndose entonces con facilidad si es o no un cuadrado.

5.3. Dos teoremas más

Enunciamos sin demostración que:

5.3.1. Un conjunto es recursivo si y sólo si el mismo y su complementario son recursivamente numerables.

5.3.2. Existen conjuntos recursivamente numerables que no son recursivos.

---

* Podría sustituirse «función computable» por «función recursiva» (véase pie de página del apartado 2.1); y «función parcialmente computable» por «función parcialmente recursiva».

6. DETERMINACIÓN DE LA FINITUD DEL PROCESO DE CÁLCULO. PROBLEMA DE LA APLICABILIDAD

No podemos terminar de hablar de algoritmos y máquinas de Turing sin mencionar un famoso problema que se enuncia así: dada cualquier M.T., una cinta con un número finito de símbolos X y una posición inicial de la cabeza, establecer un algoritmo que permita conocer si el proceso se detendrá o no. «The halting problem» (el problema de la aplicabilidad), como se le conoce en la literatura, *es un problema indecidible*.

Bien, este problema puede enunciarse de una forma distinta. En el enunciado que se acaba de dar se habla de una máquina de Turing, supongamos que su nombre es una vez más Z con el número de orden z. En él se habla también de un algoritmo, así que según la hipótesis de Turing (apartado 2.1), que hemos aceptado, eso es lo mismo que hablar de una M.T. A esta nueva M.T. la llamaremos Z_H.

Veamos un nuevo enunciado, más general, del problema: ¿Existe una máquina de Turing Z_H cuya información en cinta son parejas y que, cuando se le suministra la pareja (z, X), responde de una de estas formas:

a) Sí, la máquina Z, de número z, es aplicable al dato X.
b) No, la máquina Z, de número z, es inaplicable al dato X?

Demostración de que el problema es indecidible

Supóngase que existe tal máquina de Turing, Z_H.

Sea E un conjunto recursivamente numerable no recursivo (véase teorema 5.3.2). E es el conjunto de definición de una función parcialmente definida calculada por la máquina Z_α.

Sea x un entero cualquiera, ante el que Z_H podría dar una respuesta así:

— «Sí, $x \in E$, Z_α se parará».
— «No, $x \in E$, Z_α no se parará».

Entonces E sería recursivo, contrariamente a la hipótesis. Q.e.d.

Observación

El problema de la aplicabilidad es trasladable al terreno práctico de la programación de ordenadores. Dados un programa P y los datos D correspondientes, ¿existe un programa general P_H que permita saber si P, aplicado a D, se detendrá? La respuesta es que P_H no existe.

7. LAS MÁQUINAS DE TURING Y LOS LENGUAJES TIPO 0

La lingüística formal ofrece un campo de interesante aplicación de las M.T.'s. Los distintos lenguajes formales se tratan con cierta extensión en este mismo libro, en el

tema «Lenguajes», donde puede verse la clasificación de aquellos en niveles relacionados con las reglas gramaticales que los generan y con los mecanismos que los aceptan (autómatas de uno u otro tipo).

Por lo que se refiere al autómata especial llamado M.T. —asunto que concierne a los capítulos 5 y 6 de este tema— avancemos unas líneas de su relación con la lingüística matemática.

— Puede construirse una M.T. que acepte cualquier lenguaje generado por un sistema de escritura no restringido (lenguajes tipo 0).
— Cualquier lenguaje generado por una gramática del tipo 0 es recursivamente numerable.

Se dice que un lenguaje $L \subseteq E^*$ (esto se verá con detalle en «Lenguajes») es recursivamente numerable si se puede construir una M.T. que acepte todas las cadenas en L y ninguna otra. Un lenguaje es recursivo si tanto L como su complementario, $E^* - L$, son recursivamente numerables. Si L es recursivo se puede construir una M.T. (en la práctica, un programa) capaz de reconocer si una cadena dada es o no un elemento de L. Realmente, esta M.T. estaría compuesta de dos M.T.'s, M.T.1 y M.T.2. El conjunto de ambas o M.T. principal se limitaría a anotar cual de las dos máquinas acepta la cadena (M.T.1 reconocería cadenas en L y M.T.2, cadenas en $E^* - L$).

Supóngase que L es recursivamente numerable, pero no recursivo. En tal caso, no sería posible construir la M.T. principal a que se hacía referencia en el párrafo anterior, al no poderse construir la M.T.2. La consecuencia es que la M.T.1 resultaría insuficiente, ya que, de no pararse, no podría saberse si la máquina (en la práctica, el programa) se había encerrado en un bucle o, simplemente, necesitaba más tiempo para aceptar la cadena en cuestión.

Una línea lingüística como la que acaba de esbozarse se prolonga en importantes desarrollos en el campo de la Inteligencia Artificial.

8. RESUMEN

Se han definido las *funciones computables* como aquellas a las que corresponde una máquina de Turing aplicable a una r-upla $\overline{n}_1 0 \overline{n}_2 0 \overline{n}_3 \ldots 0 \overline{n}_r$, que produce un número $\langle \gamma_p \rangle$ de unos.

Las M.T.'s pueden catalogarse mediante alguna *técnica de codificación,* tal que a cada máquina le corresponda un número entero positivo y a cada número entero positivo le corresponda como máximo una sola M.T. La técnica clásica, a nivel teórico, es la de los *números z de Gödel.*

Así codificada la colección de M.T.'s, pueden diseñarse dos máquinas de Turing, una para codificar máquinas de Turing (es decir, transformar un esquema funcional en un número de Gödel) y otra para decodificar (es decir, transformar un número de Gödel en un esquema funcional de M.T., si ésta existe).

Con este instrumental *se ha redefinido la máquina de Turing universal* como aquella a la que sólo es necesario darle el número de catálogo de la M.T. a simular y los datos para ésta.

Las nociones de computabilidad y de parcial computabilidad se emplean en relación con los procesos de cálculo en conjuntos de números naturales para definir qué es un *conjunto recursivo* (función característica computable) y qué es un *conjunto recursivamente numerable* (dominio de función parcialmente computable).

Por último, se han dado unas muestras del grado de interés de los conceptos de los *conjuntos recursivos y recursivamente numerables* aplicándolos al razonamiento acerca de la *indecibilidad del problema de la aplicabilidad de cualquier maquina de Turing* y *a la aceptación de lenguajes generados por sistemas de escritura no restringidos.*

9. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

El problema de si existe un procedimiento efectivo para determinar si una fórmula arbitraria en el cálculo de predicados de primer orden aplicada a enteros es cierta, parece que estaba ya implícita en Leibniz, se hace explícita en Schröder (1895) y finalmente, en Hilbert, con el cambio de siglo. En 1931, Gödel publicó su famoso *teorema de incompletitud*, probando que tal procedimiento efectivo no podía existir.

La máquina de Turing es el concepto relacionado con la noción intuitiva informal de procedimiento efectivo y se debe a Turing (1936), pero la suposición de que tal noción de función computable pueda ser identificada con la clase de funciones parcialmente recursivas se conoce como la *hipótesis de Church* o la *tesis de Church-Turing* (Hopcroft, Ullman, 1979). Formulaciones alternativas a la de Turing se encuentran en Kleene (1936), Church (1936) o Post (1936).

Formalismos equivalentes a las funciones parcialmente recursivas incluyen el λ-cálculo (Church, 1941), las funciones recursivas (Kleene, 1952) y los sistemas de Post (1943).

Para este capítulo hemos utilizado con preferencia los libros de Arbib (1965), y de Gross y Lentin (1967). En particular, a este último le debemos la enunciación de los teoremas 5.3.1 y 5.3.2, así como la definición de función característica de un conjunto y el segundo de los enunciados del problema de la aplicabilidad.

Un libro bien escrito sobre introducción a las máquinas de Turing, a las funciones recursivas y temas conexos, es el de Hennie (1977), del que, entre otras cosas, se ha usado su descripción de los procesos fundamentales de cálculo asociables a un conjunto G de números naturales.

La leve referencia del final del apartado 7 a los desarrollos de la línea lingüística dentro de la Inteligencia Artificial se encontró en Hunt (1975).

Capítulo 7
COMPLEJIDAD

1. INTRODUCCIÓN

Este capítulo aborda el fundamento teórico de las limitaciones prácticas que se presentan en la aplicación del concepto de algoritmo. En pocas palabras, se dedica a estudiar la factibilidad de los algoritmos.

De entre todos los problemas que pueden plantearse, el conjunto de aquellos que son computables (decidibles), es decir, que pueden teóricamente ser resueltos aplicando un algoritmo, es muy reducido. Sin embargo, no todos los problemas computables son factibles en la realidad, por requerir a veces demasiados recursos, ya sean de tiempo, espacio de memoria o circuitos materiales.

La teoría de la complejidad algorítmica es la encargada de definir los criterios básicos para saber si un problema computable es *factible* o, dicho de otro modo, si tiene un algoritmo eficiente para su resolución y en este caso cuál es su grado de eficiencia (recuérdese la propiedad de eficacia de todo buen algoritmo, apartado 4 del capítulo 2). Aunque nada más una ínfima parte de los problemas son algorítmicamente factibles, el conjunto es lo suficientemente amplio y variado como para que esta teoría resulte no solamente interesante, sino del todo imprescindible para el conocimiento de las nociones esenciales de la informática.

Iremos de lo más general a lo más particular. Primero, se tratará el asunto de la complejidad en términos de la máquina de Turing, puesto que así el tamaño del problema puede expresarse de la manera más sencilla posible como el número n de casillas que ocupa la información de entrada. Lógicamente, la complejidad del algoritmo resolutor se podrá poner en función de n contabilizando (o estimando por algún procedimiento), bien el número de casillas que es necesario explorar, bien el número de movimientos de la máquina.

Con posterioridad, se trasladará la cuestión a un terreno más práctico, en el que se dispone de operadores más funcionales, por lo general en forma de máquinas

computadoras secuenciales, aunque no pueda dejarse sin mencionar la emergencia cada día más patente de máquinas de procesos físicamente concurrentes y su repercusión sobre la complejidad. En todo caso, y al igual que con las máquinas de Turing, *se medirá la complejidad por un orden de magnitud*, del tiempo (por ejemplo), expresado en forma abstracta como *función del tamaño del problema*. Decir «en forma abstracta» significa expresar el tiempo como un número de pasos operativos elementales, al objeto de independizarlo de la velocidad concreta de la máquina ejecutora.

A grandes rasgos, los problemas computables se clasifican en buenos *(complejidad polinómica)* y malos *(complejidad exponencial)*. Entraremos en algunos detalles sobre estos conceptos y conceptos derivados.

Para terminar, nos ocuparemos de un apartado de la complejidad teóricamente menor, la complejidad de los programas, a la que se ha dado en llamar —de manera a nuestro juicio incorrecta— *complejidad del software*. La complejidad del software tiene que ver, no con la eficiencia de los algoritmos, sino con la eficiencia de la programación de los algoritmos.

2. COMPLEJIDAD Y MÁQUINAS DE TURING

Es posible introducir los conceptos involucrados por la complejidad de los algoritmos sin necesidad de apelar a la máquina de Turing, pero su potencialidad de soportar la ejecución de cualquier problema computable la señala como argumento «natural» para aproximarse a criterios universales en lo referente a eficiencia algorítmica.

2.1. Dificultad de encontrar un criterio universal de complejidad

Recordemos que, en efecto, al estudiar la máquina de Turing no se impuso ningún límite de espacio ni tiempo, o, lo que era lo mismo, todo algoritmo que pueda ejecutarse en un número finito de pasos es simulable mediante una máquina de Turing. Esto es así porque la máquina de Turing no padece limitaciones en cuanto a la longitud de su cinta ni en cuanto al tiempo de ejecución.

Como sabemos, en la práctica ocurre que la cantidad de memoria viene limitada por razones diversas, de índole tecnológica, económica u otra. El tiempo de ejecución tampoco puede ser cualquiera, por obvias razones de pragmatismo.

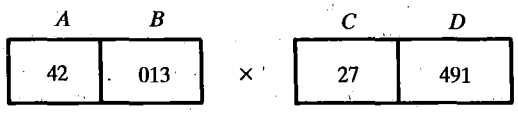
Establecer un criterio universal para poder saber cuán eficiente es un algoritmo no es sencillo a priori, porque un mismo problema puede resolverse sobre diferentes tipos de máquinas, con diferentes grados de eficiencia. Examinemos un momento este asunto por medio de un ejemplo sencillo. Multiplicar dos números de n dígitos decimales por el procedimiento que a todos nos enseñaron en la escuela requiere un tiempo de ejecución proporcional a n^2. El ejemplo de la figura 7.1.a) muestra un caso en el que $n = 5$. El número de operaciones necesarias es, de una parte, aproximadamente igual a $n \times n$ para obtener las n filas: cada fila resulta de efectuar n multiplicaciones de dos números de 1 dígito decimal. De otra parte, después hay que sumar las n filas interdesplazadas y esto puede costar alrededor de n^2 sumas de dos números de 1

dígito decimal. En resumen, el tiempo total se compone de $c \cdot n^2$ (c, constante) operaciones, que convencionalmente podemos suponer equivalentes a unidades de tiempo, aunque sabemos que hay involucradas dos tipos de operaciones distintas.

(a)

$$\begin{array}{r} 42013 \\ \times\ 27491 \\ \hline 42013 \\ 378117 \\ 168052 \\ 294091 \\ 84026 \\ \hline 1154979383 \end{array}$$

(b)



$$\begin{aligned} A \cdot C &= 1134 \\ (A + B) \cdot (C + D) - A \cdot C - B \cdot D &= 20973 \\ B \cdot D &= 6383 \\ \hline & 1154979383 \end{aligned}$$

FIGURA 7.1.

Existen algoritmos más rápidos para resolver el mismo problema. Uno de ellos está aplicado al caso práctico escogido en la figura 7.1.b). Este algoritmo requiere un tiempo proporcional a $n^{1,59}$ y se ha obtenido dividiendo el problema en subproblemas. El algoritmo más rápido conocido a este respecto ejecutable en una máquina secuencial tiene un tiempo proporcional a $n \cdot \log_2 n \cdot \log_2 \log_2 n$. Si la máquina fuera paralela, el tiempo podría ser proporcional a $\log_2 n$.

En resumen, la eficiencia de un algoritmo no depende sólo del tiempo y del espacio (memoria) utilizados, sino también de los circuitos disponibles (hardware y, en particular, del número y estructura de los procesadores), en suma, de los recursos concretos (véase capítulo 4).

La máquina de Turing nos permite, una vez más, liberarnos de la mayor parte de esas ligaduras materiales gracias a su estructura de operación de formato único (aunque, como sabemos, su estructura admite diversas variantes).

No obstante, utilizaremos la máquina de Turing solamente para iniciarnos en los conceptos básicos de *complejidad espacial y temporal*, la *complejidad en términos asintóticos* y algunos resultados sobre los efectos en la complejidad debido al *aumento*

lineal de velocidad, la *compresión de la información* y la *reducción del número de cintas*. El estudio detallado de estas cuestiones es objeto de obras especializadas. En el subapartado 2.4 se dará una breve idea sobre la cuestión de los efectos en la complejidad.

2.2. Complejidad espacial. Complejidad temporal

Las definiciones de complejidad algorítmica referidas a máquinas de Turing pueden asociarse con gran generalidad a un tipo de algoritmos tales como los reconocedores de lenguajes. En el apartado 7 del capítulo anterior hemos conocido que existe una relación entre la máquina de Turing y un cierto tipo de lenguajes, a los que se llama «lenguajes tipo 0». El estudio de esta relación y en particular la clasificación de los lenguajes de acuerdo con la complejidad estructural de las respectivas clases de autómatas reconocedores se detalla en el tema «Lenguajes».

Existe otra clasificación de los lenguajes, relativa ésta a la complejidad computacional (o algorítmica), que se basa en la cantidad de tiempo, espacio o cualquier otro recurso necesaria para reconocer un lenguaje en una máquina general, como es la máquina de Turing. Tal complejidad es la que nos interesa aquí.

En el capítulo 6 se usaba una codificación para representar cualquier expresión simbólica por un número, lo que nos llevaba a la posibilidad teórica indudable de catalogar numéricamente la colección de las máquinas de Turing. Por un principio análogo aunque en modo alguno semejante, podríamos representar los problemas reales, como el de la multiplicación de dos números visto antes, en términos de lenguajes.

Bastaría con definir adecuadamente una función de codificación tal que a cada entrada del problema (en el ejemplo serían los dos números a multiplicar) le correspondiera una determinada cadena de símbolos. Así:

Sea Σ un alfabeto, π un problema y E_π el conjunto de entradas del problema π. Se define la función de codificación α.

$$\alpha: E_\pi \rightarrow \Sigma^* \qquad (1)$$

El problema se convierte en un lenguaje L que una máquina de Turing concreta acepta o no. Lícitamente, podemos preguntarnos por la complejidad de las máquinas de Turing (o de los algoritmos) por referencia a los lenguajes.

Se define la complejidad espacial utilizando la máquina de Turing M de la figura 7.2, que consta de una cinta de entrada de sólo lectura con la información enmarcada por delimitadores en sus extremos y de K cintas semi-infinitas de memoria de lectura-escritura. La máquina M tiene una *complejidad espacial* $S(n)$ (o es $S(n)$-*limitada en espacio*) si, para cualquier entrada de longitud n, explora como máximo $S(n)$ casillas en cualquiera de las cintas de trabajo.

Por razones de conveniencia teórica, la máquina M de la figura 7.3 suministra el modelo adecuado para definir la complejidad temporal de una máquina de Turing. Esta consta de K cintas abiertas por ambos extremos, de lectura y escritura, con una de ellas dispuesta para contener la información de entrada, que, como siempre,

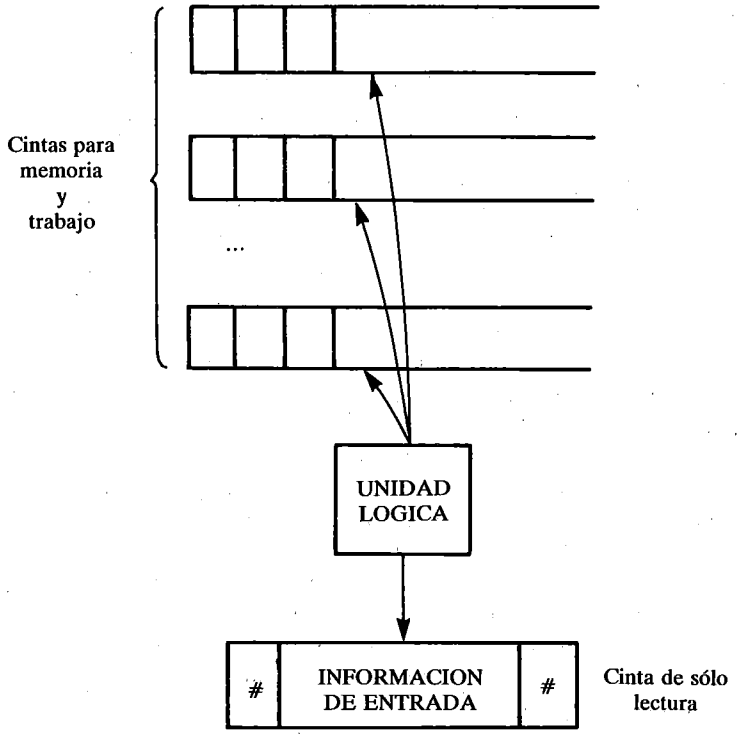


FIGURA 7.2.

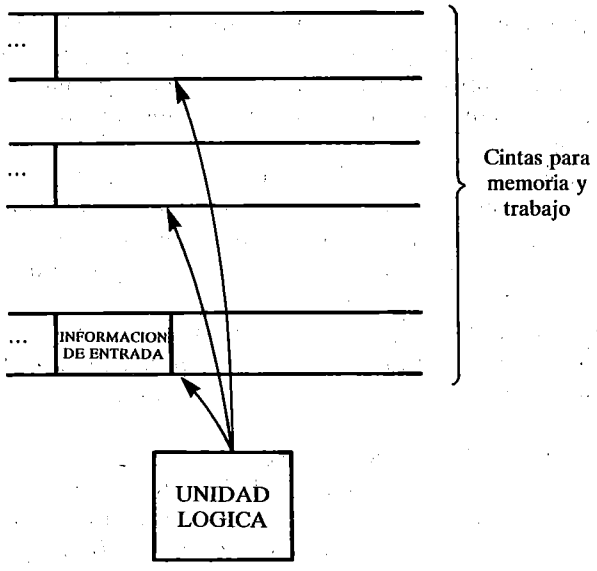


FIGURA 7.3.

suponemos de longitud n. Si para cualquier palabra de entrada de longitud n, M realiza como máximo $T(n)$ movimientos antes de detenerse, se dice que M tiene *complejidad temporal* $T(n)$ (o es $T(n)$-*limitada en tiempo*).

Puesto que la máquina M sirve para ejecutar un algoritmo, lo anterior viene a significar que la ejecución del algoritmo no requiere más de $S(n)$ posiciones en la cinta o más de $T(n)$ desplazamientos, respectivamente.

Obsérvese que las funciones $S(n)$ y $T(n)$ no necesitan ser una función exacta, sino sólo un indicador de la forma de variación en función de la longitud de la entrada. Por ejemplo, a efectos de estudiar la eficiencia de un algoritmo simulado por una máquina de Turing M, da igual que su tiempo de ejecución sea $14\ n^2 + 25\ n - 4$ ó $3\ n^2$, pues su forma de crecimiento es en ambos casos de tipo n^2. Diremos que la máquina M es de complejidad temporal (n^2), o (n^2)-limitada en tiempo. Esta segunda expresión se comprende intuitivamente, pues si nos fijamos en las funciones anteriores, vemos que en el límite (n infinito) tienden a las asíntotas $14\ n^2$ y $3\ n^2$ respectivamente, que son del tipo cn^2 (c, constante).

Así pues, al hablar de complejidad estamos hablando en realidad de *complejidad en términos asintóticos*. Parece razonable asumir que toda máquina de Turing será al menos de complejidad temporal $(n + 1)$, pues éste es el tiempo requerido para leer la entrada. Igualmente, toda máquina de Turing será al menos de complejidad espacial (1), pues debe poseer al menos una casilla de entrada.

Cuando decimos que una máquina es $T(n)$-limitada en tiempo, en realidad estamos diciendo Máx$[(n + 1), T(n)]$ limitada en tiempo, y lo mismo para la complejidad espacial, que será Máx$[(1), S(n)]$.

2.3. Máquinas deterministas, máquinas no deterministas y tipos de complejidad

Los conceptos que acabamos de ver son aplicables tanto a máquinas deterministas, en las que los pasos a seguir son siempre idénticos para toda entrada, como a máquinas no deterministas, en las que deben tomarse una serie de decisiones que provocan el que la máquina realice más o menos movimientos o escriba más o menos casillas de la cinta.

Para este segundo caso las definiciones son ligeramente diferentes. Diremos que una máquina de Turing no determinista es $S(n)$-limitada en espacio o de complejidad espacial $S(n)$ si para toda entrada de longitud n, la máquina no debe indagar más de $S(n)$ posiciones de la cinta, cualquiera que sea la secuencia de decisiones que pueda tomar durante la ejecución del algoritmo.

De forma similar, la máquina será $T(n)$-limitada en tiempo o de complejidad temporal $T(n)$ si, para toda entrada de longitud n, la cabeza de lectura de la máquina efectúa menos de $T(n)$ desplazamientos, cualquiera que sea la secuencia de decisiones.

La familia de lenguajes de complejidad espacial $S(n)$ se denomina DSPACE($S(n)$). Los lenguajes con complejidad espacial no determinista $S(n)$ se engloban en la clase NSPACE($S(n)$). En lo que respecta al tiempo, la familia de lenguajes de complejidad temporal $T(n)$ se denomina DTIME($T(n)$) y si es no determinista NTIME($T(n)$).

2.4. Algunos resultados de la teoría de complejidad en máquinas de Turing

Damos sin demostración determinados resultados. Los primeros vienen a expresar que, en lo referente a la complejidad, es la tasa funcional de crecimiento (es decir, lineal, cuadrática, exponencial) lo que importa, y que pueden despreciarse los factores constantes. Podrá hablarse, por ejemplo, de una complejidad $\log n$ sin especificar la base de los logaritmos, ya que $\log_b n$ y $\log_c n$ difieren en un factor constante. En su conjunto, estos resultados vienen a reforzar la idea intuitiva práctica de complejidad asintótica.

Un primer teorema es que si el lenguaje L es aceptado por una máquina de Turing $S(n)$-limitada en espacio dotada con K cintas de memoria, entonces, para cualquier $c > 0$, L es aceptado por una M.T. $c \cdot S(n)$-limitada en espacio.

Se prueba simulando la primera M.T. por otra que la simula y que en cada una de sus casillas comprime en un solo símbolo el contenido de r casillas de la cinta correspondiente de la primera M.T.

Otro teorema relacionado con la complejidad espacial en máquinas de Turing dice así: Si un lenguaje L es aceptado por una M.T. $S(n)$-limitada en espacio provista de K cintas de memoria, es aceptado por una M.T. $S(n)$-limitada en espacio con una sola cinta de memoria. De hecho, si $S(n) \geq n$ podemos suponer que cualquier M.T. $S(n)$-limitada en espacio es una M.T. con una sola cinta (que sirve de entrada y de memoria de trabajo).

Por último, puede demostrarse que, bajo ciertas condiciones, si L es aceptado por una M.T. con k cintas $T(n)$-limitada en tiempo, es aceptado por otra M.T. con k cintas $c \cdot T(n)$-limitada en tiempo, para cualquier $c > 0$.

Veamos ahora brevemente otro grupo de resultados.

Es indudable que los que en el apartado 6 del capítulo 5 hemos denominado sucedáneos de la máquina de Turing, algunos de los cuales se han utilizado líneas arriba para definir, por conveniencias teóricas, la complejidad, nos proporcionan una idea de la variabilidad de circuitos disponibles (número de cabezas, número de cintas de memoria y sólo lectura, límites en la longitud de las cintas, etc.) y consecuentemente de su influencia sobre la complejidad. Por fortuna, todos estos tipos acaban pudiéndose reducir a una M.T. mono-cinta.

Es así como se demuestra, por ejemplo, que si un lenguaje L está en la clase DTIME($T(n)$), es decir, posee complejidad temporal $T(n)$, entonces es aceptado en tiempo $T^2(n)$ por una M.T. mono-cinta.

Asimismo, si L es aceptado por una M.T. $T(n)$-limitada en tiempo de k cintas, es aceptado por una M.T. de dos cintas de memoria con complejidad temporal $T(n) \cdot \log T(n)$.

Resumiendo, comprobamos que alteraciones importantes de la estructura de una M.T., como puede ser el aumento o disminución del número de cabezas y de cintas de trabajo, si bien no restringen —como se estableció en capítulos anteriores— las posibilidades de la máquina de Turing básica, afectan en cambio a la complejidad temporal. Los dos teoremas que se acaban de enunciar son bien explícitos en este sentido: el paso de complejidad temporal $T(n)$ a complejidades $T^2(n)$ o $T(n) \cdot \log T(n)$ nos ayudan, por analogía, a comprender el cambio cuantitativo que cabe

esperar de procesar algoritmos en forma secuencial a procesarlos en forma concurrente (cuando ello es posible).

3. MEDIDAS DE LA COMPLEJIDAD ALGORÍTMICA

Desplacemos ahora nuestra atención a terrenos de interés práctico para cualquiera que se dedique a la informática.

En primer lugar, cabe plantearse la pregunta de si realmente medir (o estimar) la complejidad algorítmica tiene algún interés práctico. A diario, numerosos ejemplos nos demuestran que sí. Cualquier software implica cálculos, búsquedas, ordenaciones, manipulaciones u otras operaciones, que son especificados por una solución algorítmica y sometidos a un requisito de eficacia en cuanto a los recursos empleados.

Pueden citarse:

— Programas en tiempo real. La duración de su ejecución está limitada en el tiempo, como es evidente en un sistema que controla el movimiento de una antena que apunta a un satélite para una aplicación de transmisión de datos.
— Programas con volumen limitado de almacenamiento, tal como sucede en un sistema embarcado en un sobrecargado ingenio espacial.
— Programas de uso muy frecuente, por ejemplo, programas de software de base, compiladores, manejadores de dispositivos, y otros. En ellos no tiene por qué existir estrictamente una limitación de recursos (salvo por razones de competitividad económica), pero, por razones de rendimiento operativo, es en cambio muy importante mejorar los algoritmos utilizados y optimizar, llegado el caso, su codificación.

Como ya sabemos, se ha convenido en que sea el tamaño de los datos de entrada del problema el parámetro para medir su complejidad computacional. Lógicamente, una vez fijado el problema y definida su entrada, la complejidad dependerá de la clase (tiempo, espacio) y número de recursos considerados (máquinas secuenciales, máquinas paralelas) y del algoritmo elegido, y, por último, habrá un aspecto práctico digno de tenerse en cuenta: la variabilidad concreta de la medida de la complejidad para unos datos determinados (una vez fijados los recursos y el algoritmo). En todo caso, la complejidad siempre se expresa en unos términos independientes de la velocidad del operador concreto.

Vamos a empezar viendo los tipos clásicos de complejidad en función del tamaño n de la entrada.

3.1. Complejidad polinómica y complejidad exponencial

La figura 7.4 permite visualizar la variación del orden de magnitud de cuatro algoritmos de complejidad (que supondremos temporal) $\log_2 n$, n, n^2 y 2^n, respectivamente, en función de n.

n, tamaño de datos de entrada	$\log_2 n$ microsegundos	n microsegundos	n^2 microsegundos	2^n microsegundos
10	0,000003 seg.	0,00001 seg.	0,0001 seg.	0,001 seg.
100	0,000007 seg.	0,0001 seg.	0,01 seg.	10^{14} siglos
1.000	0,00001 seg.	0,001 seg.	1 seg.	Astronómico
10.000	0,000013 seg.	0,01 seg.	1,7 min.	Astronómico
100.000	0,000017 seg.	0,1 seg.	2,8 horas	Astronómico

FIGURA 7.4.

El problema cuyo algoritmo tiene complejidad 2^n se convierte en intratable aún para pequeños tamaños de los datos. No le ocurre así al algoritmo n^2-complejo, pero es evidente que para tamaños grandes puede requerir tiempo considerable.

La forma de crecimiento de los recursos necesarios en la ejecución de un determinado algoritmo se denota por $0(f(n))$, y ya hemos convenido anteriormente que sólo interesan los términos de mayor crecimiento de la función (el comportamiento asintótico).

Así, las complejidades de la figura 7.4 son denotadas $0(\log_2 n)$, $0(n)$, $0(n^2)$ y $0(2^n)$. Un tiempo de ejecución de $14\ n^2 + 25\ n - 4$ se resume en que la complejidad es $0(n^2)$.

Los algoritmos cuyo comportamiento asintótico es del tipo $0(n)$, $0(n^2)$, $0(n^3)$, ..., en general $0(n^c)$ para c constante, se llaman *algoritmos polinómicos*, o de *complejidad polinómica*. Los algoritmos que se comportan como 2^n (en general, c^n) son *algoritmos exponenciales*, o de *complejidad exponencial*.

Las figuras 7.4 y 7.5, examinadas conjuntamente, nos permiten apreciar que, siendo totalmente merecido el apelativo de «malos» con el que se conoce a los

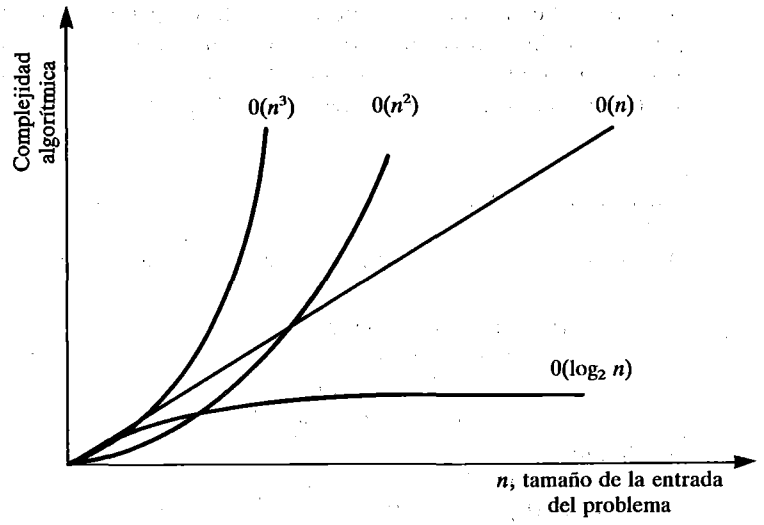


FIGURA 7.5.

algoritmos exponenciales, también entre los polinómicos los hay mejores y peores y algunos pueden llegar a resultar bastante malos en cuanto que crece un poco el tamaño *n*.

3.2. Ejemplos de algoritmos con complejidad polinómica

Los algoritmos de búsqueda y clasificación (ordenación) constituyen una clase muy favorable para suministrar ejemplos didácticos acerca de la complejidad algorítmica, o de su inversa, la eficiencia del algoritmo.

En primer lugar, hacen fácil lo que muchas veces no lo es: precisar el tamaño de la entrada del problema, que en esta clase coincide con el número de elementos del conjunto sobre el que hay que operar la búsqueda o la ordenación. En segundo lugar y no menos importante, un algoritmo de búsqueda o de ordenación se traduce en un plan de operaciones básicas de comparación entre valores de elementos y de movimientos de estos elementos (en la ordenación), con la particularidad de que ambos tipos de operaciones dependen del número *n* de elementos del conjunto de partida.

Huelga decir que en esta categoría existen algoritmos muy sofisticados, en los que intervienen más operaciones que las puramente comparativas y de movimiento. Forman parte de las técnicas habituales en el campo de la gestión de ficheros y en general de los sistemas de gestión de bases de datos. No es nuestra intención entrar en ellos, aunque es justo dejar aquí señalado que todo su estudio se relaciona con problemas de eficiencia bajo diversas circunstancias.

3.2.1. *Búsqueda secuencial*

Consiste en explorar consecutivamente y partiendo de un primero los elementos de una lista o conjunto de *n* elementos comparando cada uno con una clave, que es el elemento buscado. Este elemento puede estar o no en el conjunto explorado.

El código siguiente en Pascal permite resolver esta búsqueda.

```
BEGIN
    encontrado: = False;
    ind: = 1;
    WHILE (NOT encontrado) AND (ind < = n) DO
        BEGIN
            IF lista [ind] = clave
                THEN encontrado: = True                    (2)
                ELSE ind: = ind + 1
            END; {bucle de búsqueda}
    IF encontrado
        THEN BuscSec: = ind
        ELSE BuscSec: = 0
    END; {Búsqueda secuencial}
```

La salida de este algoritmo, que llamamos BuscSec, toma el valor 0 si no se encuentra la clave o el valor del lugar ocupado por el primer elemento coincidente con la clave, en caso positivo. "Encontrado" es una variable booleana e "ind" una variable de tipo intervalo formado por enteros. Los textos entre corchetes son comentarios. < = y > = significan menor o igual y mayor o igual, respectivamente.

El algoritmo codificado en la forma de (2) no requiere explorar más de n elementos, en el peor de los casos. Y esto es prácticamente todo lo que hay que hacer. Tanto la complejidad temporal como la espacial son $0(n)$. En adelante, sólo hablaremos de complejidad temporal.

3.2.2. *Búsqueda binaria*

El algoritmo de búsqueda binaria es mucho más eficiente (esto es, su complejidad mucho menor) que el de búsqueda secuencial. El algoritmo de búsqueda binaria tiene complejidad $0(\log_2 n)$. Véase la figura 7.5. Pero en realidad no son estrictamente comparables, porque el algoritmo de búsqueda secuencial es mucho más general, al no imponer condicionamiento alguno en cuanto al orden de los elementos en el conjunto. Por el contrario, en la búsqueda binaria se requiere previamente que el conjunto esté ordenado*.

Este algoritmo trabaja comparando la clave con el elemento situado en medio de la lista. Si coincide, lo da como encontrado y si no, descarta la mitad (de ahí el nombre) de la lista que no puede contener la clave y repite el proceso.

De nuevo en Pascal, el algoritmo de búsqueda binaria en una lista de enteros puede expresarse en la forma de (3). Las variables "alto", "medio" y "bajo" son de tipo intervalo con enteros, como antes "ind".

Obsérvese que con cada comparación se divide la lista por la mitad, de modo que ésta se va convirtiendo en la mitad, en la cuarta, la octava... parte de su tamaño inicial. En el peor de los casos, el proceso continuará hasta que la lista a explorar esté vacía (en el programa, sentencia "ELSE BuscBin: = 0"). En consecuencia, el mayor número de comparaciones que puede necesitar el método de búsqueda binaria será el primer valor entero k tal que $2^k \geq n$, es decir, $k \geq \log_2 n$.

```
BEGIN
    bajo: = 1;
    alto: = n;
    encontrado: = False;
    WHILE (NOT encontrado) AND (bajo < = alto) DO
        BEGIN
            medio: = (bajo + alto) DIV 2;
            IF lista [medio] = clave
```

* El interés de aplicar la búsqueda binaria (precedida de un algoritmo de ordenación) frente a la búsqueda secuencial dependerá del número de veces que haya que aplicar el algoritmo de búsqueda, y, por supuesto, también del orden de complejidad del algoritmo de ordenación.

```
                THEN encontrado: = True                    (3)
                ELSE IF lista [medio] > clave
                    THEN alto: = medio −1
                    ELSE bajo: = medio +1
            END; {bucle de búsqueda}
        IF encontrado
            THEN BuscBin: = medio
            ELSE BuscBin: = 0
        END; {búsqueda binaria}
```

3.2.3. *Ordenación por el método burbuja*

Ordenar es el proceso de reorganizar un conjunto de objetos según una determinada secuencia. Existen muchos métodos de ordenación y los más sencillos, denominados métodos directos, tienen complejidad $0(n^2)$. Normalmente, hay un número máximo de comparaciones y movimientos y un número mínimo, como resultante del grado inicial de ordenación del conjunto.

El método de la burbuja es un método de intercambio, que consiste en comparar dos elementos consecutivos, permutándolos cuando están desordenados y llegando hasta el final de la lista en cada una de las pasadas. Por ejemplo, si tenemos inicialmente la siguiente lista de números enteros (seguiremos utilizando por simplicidad este tipo de objetos):

(2, 4, 7, 6, 15, 12, 9, 10)

En una primera pasada (variable "Paso" en el código Pascal (4), la lista se reordenará así:

(2, 4, 6, 7, 12, 9, 10, 15)

La segunda pasada inicia los movimientos a partir de la pareja 12, 9, que es permutada, descendiendo 12 hasta el penúltimo lugar de la lista:

(2, 4, 6, 7, 9, 10, 12, 15)

El número de comparaciones en este algoritmo es $1/2\,n(n-1)$ y los números mínimo, medio y máximo de movimientos son 0, $3/4(n^2-n)$ y $3/2(n^2-n)$, respectivamente. Por consiguiente, la complejidad es del orden $0(n^2)$.

3.3. Sinopsis

El esquema de la figura 7.6 pretende sintetizar algunas de las ideas ya vistas o que se desprenden de los conceptos y ejemplos examinados.

```
BEGIN
    Intercambio: = TRUE;
    Paso: = 1;
    {El número de pasadas puede variar de 1 hasta n − 1, dependiendo
    de la disposición inicial}
    WHILE (Paso < = n − 1) AND (Intercambio)
        DO BEGIN
            {Por el momento no se han producido intercambios}
            Intercambio: = False;
            FOR j: 1 TO n − Paso DO
                {si un elemento es menor que el anterior,                (4)
                hay que intercambiarlo}
            IF a [j] > a [j + 1]
                THEN BEGIN
                    Intercambio: = TRUE;
                    Aux: = a [j];
                    a [j]: = a [j + 1];
                    a [j + 1]: = Aux
                END; { if then}
              Paso: = Paso + 1
            END {while}
END; {procedimiento burbuja}
```

Un aspecto que no habíamos comentado anteriormente es el que se refiere, no a la complejidad del algoritmo, sino a la complejidad del problema. Tal cuestión remite a diseñar y comparar (en tiempo secuencial, por ejemplo) todos los algoritmos posibles para resolver un problema. En el apartado 2 mencionábamos distintas soluciones al problema de la multiplicación de dos números enteros de longitud n. Sus complejidades oscilaban entre $0(n^2)$ y $0(n \cdot \log_2 n \cdot \log_2 \log_2 n)$ o, lo que es lo mismo, entre $0(n^2)$ y $0(n \cdot \log n \cdot \log \cdot \log n)$.

Se conoce como cota superior de la complejidad de un problema a la complejidad del mejor algoritmo que se haya podido encontrar. En el caso de la multiplicación, la cota superior es $0(n \cdot \log n \cdot \log \log n)$, para recursos de computación secuencial. Es posible probar, a veces, que no existe algoritmo que pueda resolver un determinado problema sin emplear como mínimo una cierta cantidad de recurso, a la que se llama cota inferior. Si seguimos razonando acerca del problema de la multiplicación, convendremos en que no puede existir algoritmo más eficiente que el que necesitase un tiempo proporcional a n, puesto que siendo éste el tamaño de la entrada, es evidente que, como mínimo, cada dígito debería ser considerado al menos una vez.

¿Es posible encontrar un algoritmo mejor que $0(n \cdot \log n \cdot \log \log_n)$ para la multiplicación? ¿Puede probarse la inexistencia de un algoritmo más rápido? En suma, ¿cuál es la exacta complejidad de la multiplicación de dos números enteros? Estas son preguntas abiertas dentro de la teoría de la complejidad.

Otra observación. El análisis de algoritmos se concentra muchas veces en un algoritmo concreto, investigando en el interior de su orden de complejidad cual es la

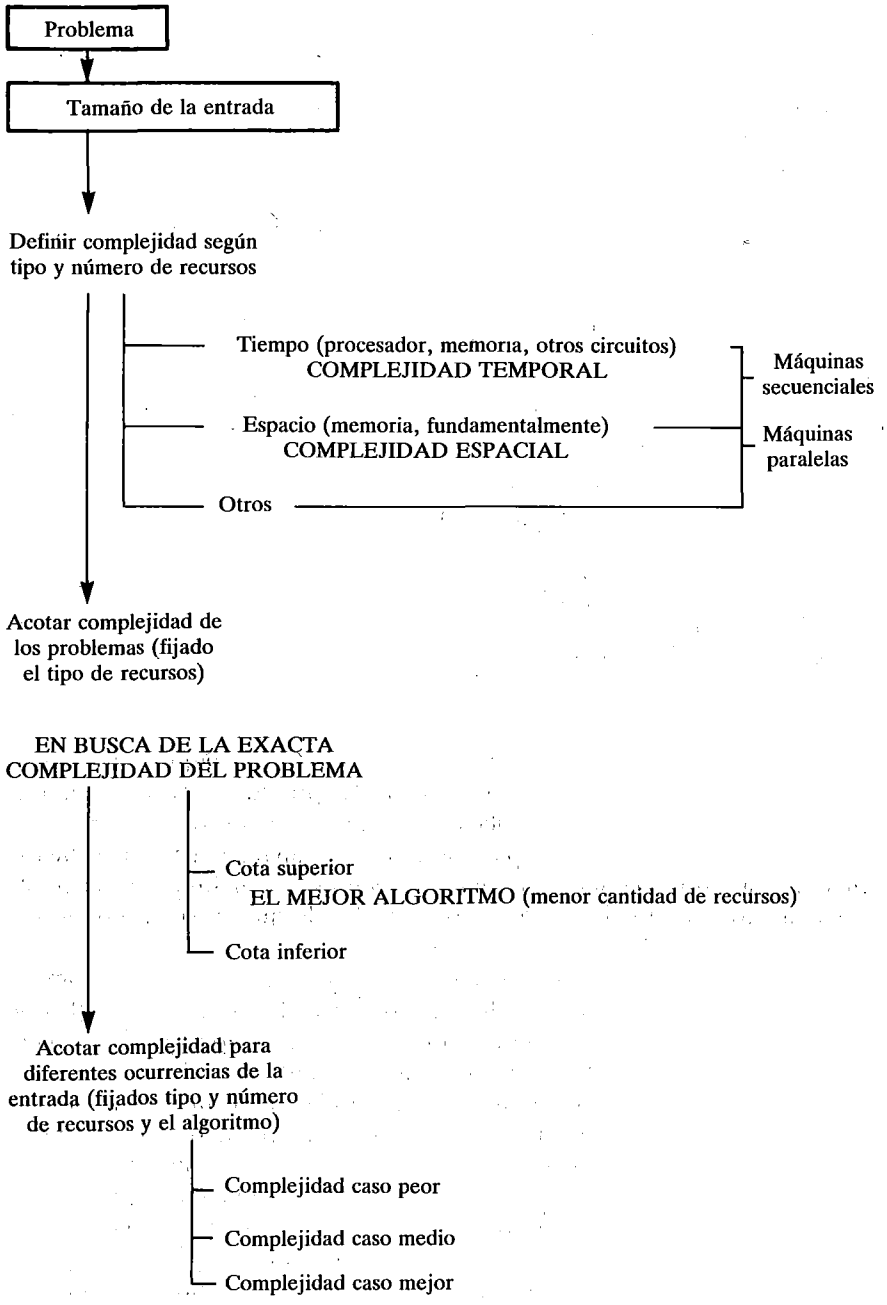

Problema
Tamaño de la entrada
Definir complejidad según tipo y número de recursos
Tiempo (procesador, memoria, otros circuitos) COMPLEJIDAD TEMPORAL
Espacio (memoria, fundamentalmente) COMPLEJIDAD ESPACIAL
Otros
Máquinas secuenciales
Máquinas paralelas
Acotar complejidad de los problemas (fijado el tipo de recursos)
EN BUSCA DE LA EXACTA COMPLEJIDAD DEL PROBLEMA
Cota superior
EL MEJOR ALGORITMO (menor cantidad de recursos)
Cota inferior
Acotar complejidad para diferentes ocurrencias de la entrada (fijados tipo y número de recursos y el algoritmo)
Complejidad caso peor
Complejidad caso medio
Complejidad caso mejor

FIGURA 7.6.

dinámica interna de ésta como resultado de las diversas configuraciones de datos. Esto conduce a evaluar medidas conocidas como casos peores, medios y mejores, de enorme interés en la práctica. El subapartado anterior nos ha ofrecido varios ejemplos en los que era patente una variabilidad de los tiempos de ejecución como producto de la configuración inicial de los datos. El lector puede imaginar hasta qué punto la dinámica de ejecución de predicados de un algoritmo complicado puede dificultar este tipo de análisis.

Por último, un comentario sobre la aparente contradicción que nos sugiere la definición de complejidad algorítmica. Está muy claro que un algoritmo es tanto menos complejo cuantos menos recursos de máquina requiera. De ello venimos tratando a lo largo de todo el capítulo. En este sentido, es menos complejo el algoritmo de ordenación Quicksort ($0(n \cdot \log n)$) que el de la burbuja. Sin embargo, el método de la burbuja se le ocurre a cualquiera, mientras que Quicksort lo inventó Hoare, precisamente uno de los más prestigiosos científicos de la informática. Desde una semántica antropológica, es más compleja (requiere más o por lo menos mejores recursos intelectuales) la solución Quicksort que la burbuja. En un caso estamos hablando de idear (inventar, diseñar, es un acto creativo) una solución y, en el otro, de ejecutarla mecánicamente.

4. PROBLEMAS P, NP Y NP-COMPLETOS

Hemos estado hablando de algoritmos más o menos eficientes, entendiendo que éstos son aquéllos cuyo crecimiento es polinómico. En general, diremos que un algoritmo es eficiente si existe una máquina de Turing M que lo simule en tiempo DTIME(n^i), donde i es una constante. A la clase de algoritmos eficientes se la denomina *la clase P*. Puede argumentarse, no sin razón, que un algoritmo (n^{50})-limitado en tiempo, no resulta muy eficiente. Esto es cierto efectivamente, pero en la práctica casi todos los problemas de P tienen un grado pequeño.

4.1. El problema P-NP

Podría pensarse que con esta clase P ya tenemos definida la eficiencia, y sin embargo, nada más lejos de eso, pues existe otra clase de problemas denominados NP que vienen a perturbar toda la teoría de complejidad.

Como ya sugeríamos en el subapartado 2.3, existen algoritmos (máquinas de Turing) llamados no-deterministas que, contrariamente a los algoritmos deterministas, no siguen un flujo fijo, sino que actúan en función de una serie de decisiones tomadas en tiempo real. *De entre los algoritmos no-deterministas existe un amplio conjunto de ellos que pueden considerarse eficientes, pero es indemostrable que estén en P* debido precisamente a que el algoritmo no es determinista. A esta clase de problemas se los llama NP. Dicho en palabras simples, la clase NP es el conjunto de problemas que tienen un algoritmo rápido (de complejidad polinómica) de verificación.

Un ejemplo de problema de tipo NP es el conocido problema de Hamilton. Dado

un grafo como el de la figura 7.7, ¿puede encontrarse un camino que una todos los puntos y que pase una sola vez por cada punto? Este problema no parece tener una solución determinista de tipo polinómico. Sin embargo, existen algoritmos no-deterministas que lo resuelven con eficiencia. Basta con elegir una posible solución al azar y verificar que cumple las condiciones del problema. Esta es precisamente una de las características más notables de los problemas de tipo NP. Encontrar el algoritmo determinista o la solución al problema es muy difícil, pero dada una posible solución, resulta sencillísimo comprobar que es válida. Esta diferencia es análoga a la que existe entre NP y P.

El problema principal en la teoría de los problemas intratables es, visto lo anterior, muy fácil de definir: ¿Es NP igual a P? *Este es el llamado problema P-NP*, y es tan difícil de resolver que casi nos vemos tentados a concluir que son dos clases diferentes. Lo que sí es demostrable es que todo problema de P es un problema de NP.

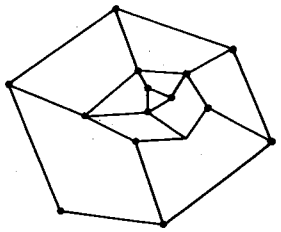


FIGURA 7.7.

Sin embargo, la cosa no acaba ahí, ya que existe una puerta abierta que permitiría demostrar que son iguales en el supuesto de encontrar la llave, si bien también podría demostrar que son distintos en caso de demostrar que la llave no existe. Vamos a ver esto con más detalle definiendo el concepto de completitud.

4.2. Completitud y problemas NP-completos

En la clase NP hay un cierto número de problemas que pueden catalogarse entre los más duros problemas, en el siguiente sentido: si se encontrase un algoritmo de tiempo polinómico para cualquiera de ellos habría un algoritmo de tiempo polinómico para todo problema en NP. Se dice que cualquier problema de esta categoría es *NP-completo*.

Un problema muy conocido entre los asiduos a la complejidad es el problema de «satisfactibilidad». Dada una expresión booleana, ¿es posible encontrar un conjunto de valores que den como resultado «true»?

El algoritmo NP es muy sencillo: cójase una posible solución al azar y verifique si el resultado es «true». Para demostrar que todo problema de NP puede reducirse al problema de satisfactibilidad (llamémoslo Lsat), hay que encontrar un algoritmo que permita simular cualquier otro problema de NP incrementando su complejidad por un

factor que como mucho habrá de ser log. Este algoritmo existe pero es bastante complicado y no lo expondremos en detalle. El principio básico es construir una nueva máquina Lsat a partir de la máquina original *M* en la cual las condiciones que deben verificarse para que *M* acepte la solución se convierten en una expresión booleana. De esta forma, la simulación de *M* con Lsat es como sigue: cójase una posible solución y verifíquese que la expresión booleana es «true». Si Lsat acepta la solución, *M* también acepta y viceversa, por lo que las dos máquinas son iguales. Si la complejidad de la máquina *M* era $T(n)$, la de la nueva máquina Lsat será como mucho $T(n) + \log(n)$.

Aparte del problema de Hamilton y el de satisfactibilidad hay otros problemas que también son NP-completos y son muy populares. Podemos mencionar unos cuantos:

— El problema del número cromático: Dado un número k, ¿existe una forma de dibujar un grafo con k colores de forma que dos vértices contiguos tengan un color diferente?
— El problema del agente de ventas: Dado un grafo, ¿cuál es el camino más corto que pasa una sola vez por cada uno de los puntos del grafo?
— El problema de las particiones: Dada una lista de enteros $(i1, i2, \ldots, ik)$, ¿existe algún subconjunto cuya suma sea exactamente $1/2(i1 + i2 + \ldots + ik)$?

Entre toda la variedad de problemas NP-completos, hay algunos para los que se han realizado enormes esfuerzos en busca de un algoritmo determinista de tipo polinómico, sin encontrarlo. Puesto que todos o ninguno de estos problemas están en P, parece normal conjeturar que ninguno lo está. Casi parece más razonable buscar mejoras en los algoritmos heurísticos, que pueden resolverlos con bastante eficiencia en aplicaciones prácticas.

4.3. Un tema no cerrado

Podríamos seguir tratando otros muchos problemas dentro del tema de la complejidad, pero éstos son —valga la redundancia— muy complejos y requieren unos conocimientos de lógica más profundos.

Existen oráculos que intentan estudiar el qué pasaría si P fuese igual a NP, conjeturas relativas a la complementación de un problema y otras muchas técnicas que permitirían poder llegar a la solución del problema P-NP.

También existen desde luego problemas que no son ni de P ni de NP, es decir, que requieren tiempos exponenciales sin posibilidad de reducción. Este tipo de problemas, llamados «intrínsecamente difíciles» forman un conjunto no menos importante dentro de la densa jerarquía que compone la complejidad espacio-temporal.

5. COMPLEJIDAD DEL SOFTWARE

Al final del subapartado 3.3, dedicado a una sinopsis de algunas de las ideas sobre complejidad algorítmica, se sugería una diferencia entre esta complejidad y la complejidad creativa específica del ser humano descubridor de algoritmos.

En informática, hay una tarea intermedia entre la de diseñar un algoritmo y la de

ejecutarlo por un instrumento mecánico. Es la tarea de programar y codificar el algoritmo por medio de un lenguaje artificial. Aunque los lenguajes empleados cada día tienden a ser de mayor potencia expresiva (lo que quiere decir que una parte importante del trabajo de programación lo hace la propia máquina), esta actividad todavía es mayoritariamente de carácter humano.

Por lo que sabemos, la codificación de un algoritmo puede introducir variaciones en su eficiencia de ejecución, pero no alterar el orden de magnitud de su complejidad. En cambio, determinadas circunstancias de esta tarea muestran una influencia profunda sobre el número de errores cometidos, la cantidad de esfuerzo requerido y en general sobre los costes del software. No existe una definición precisa de qué hay que entender por «complejidad del software», viniendo éste a ser un nombre genérico para sintetizar algunas de las mencionadas circunstancias, una especie de medida de cuán difícil es de comprender un programa y de trabajar con él.

La complejidad del software se traduce así, de una manera difusa, en un conjunto de recursos (en gran parte humanos) necesarios para producir software de calidad. Algún autor la llama «complejidad psicológica del software».

Se han propuesto diversas métricas, explayadas en unas 60 técnicas, para estimar la complejidad de la programación. Terminaremos este capítulo sobre «complejidad» resumiendo la métrica conocida como Física (o Ciencia) del Software y la métrica del número ciclomático. El cuadro de la figura 7.8 resume, por analogía con medidas literarias, las clases de complejidad de las que aproximadamente estamos hablando en este apartado. Como se verá enseguida, la medida por medio de operadores y operandos se corresponde con la Física del Software y ciertas propiedades de grafos son el sustrato teórico del método del número ciclomático.

Tipo	*Medida literaria*	*Medida software*
1. Tamaño	Número de páginas de libro.	Número de instrucciones.
2. Dificultad de texto	Estilo de autor (por ejemplo, Borges versus Rulfo).	Operadores más operandos.
3. Estructural	Recurrencias y tramas entrelazadas.	Propiedades de grafos de estructura de control.
4. Intelectual	Tema (por ejemplo, mecánica cuántica versus plantas de interior).	Dificultad del algoritmo.

FIGURA 7.8.

5.1. Física del software

Esta métrica está basada en contabilizar los operandos y los operadores con que se codifica un determinado algoritmo en un lenguaje concreto. Los *operadores* son elementos sintácticos tales como +, −, >, IF THEN ELSE, END, etc... *Operandos*

son las cantidades que reciben la acción de los operadores, variables y constantes, por ejemplo. En el programa en FORTRAN

```
      READ (5, 1) X
1     FORMAT (F 10.5)
      A = X/2
2     B = (X/A + A)/2
      C = B - A
      IF (C.LT.0) C = -C
      IF (C.LT.10.E-6) GOTO 3
      A = B
      GOTO 2
3     WRITE (6, 1) B
      STOP
      END
```

(5)

transcrito en (5) podemos contabilizar lo siguiente:

a) 13 operadores distintos. Al número de operadores distintos lo llamaremos n_1. (READ, FORMAT, =, /, ( ), +, −, .LT., IF, GOTO, WRITE, STOP, END).
b) 24 operadores en total: N_1. (Cinco =, tres /, dos de los siguientes: ( ), −, .LT., IF, GOTO, y uno de los seis restantes).
c) 11 operandos distintos: n_2. (X, F10.5, A, 2, B, C, 10.E−6, 0, y las etiquetas 1, 2 y 3).
d) 25 operandos en total: N_2. (Cinco de A y C, cuatro B, tres X, dos 2 y uno de los seis restantes).

A partir de estos elementos contables, la Física del software deriva funciones a las que denomina vocabulario y vocabulario potencial, nivel de programa, nivel de lenguaje, longitud de programa, volumen y volumen potencial del programa, contenido inteligente, esfuerzo de programación y otras, que presuntamente (éste es un tema todavía abierto) permiten predecir el esfuerzo mental y el tiempo requeridos para codificar diferentes programas y para codificar un mismo algoritmo por medio de distintos lenguajes.

El volumen del programa (figura 7.9) se mide en bits. El nivel del programa L representa una medida de lo sucinta que puede ser la codificación de un programa. Cuanto más elevado sea el nivel del lenguaje utilizado en el programa, tanto más se aproxima la implementación a una llamada de procedimiento. Recordemos al respecto la sentencia CALL MAX (*A, B,* MCD) frente al código del mismo programa en PL/I del apartado 2 del capítulo 1. Por definición $L = V^*/V$, donde V^* es el volumen del algoritmo codificado por una sentencia de llamada CALL (n_2^* suele representar el número de parámetros diferentes de entrada y salida; en el caso del algoritmo del m.c.d. $n_2^* = 3$) y $0 < L \leq 1$, siendo L tanto más próximo al valor 1 cuanto más potente es el lenguaje empleado. El algoritmo (5) se podría codificar con CALL RAIZCUA (*X, B*), con $n_2^* = 2$, siendo B la raíz cuadrada de X. $\hat{N}$ y $\hat{L}$ son fórmulas predictoras de la longitud y del nivel del programa, respectivamente.

Llama la atención la expresión para el esfuerzo E, que ha sido interpretado como

Número de operadores distintos	n_1
Número total de operadores	N_1
Número de operandos distintos	n_2
Número total de operandos	N_2
Tamaño del vocabulario	$n = n_1 + n_2$
Longitud del programa	$N = N_1 + N_2$
Volumen del programa	$V = N \log_2 n$
Vocabulario potencial	$n^* = 2 + n_2^*$
Volumen potencial	$V^* = n^* \log_2 n^*$
Longitud calculada del programa	$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$
Nivel del programa	$L = V^*/V$
Nivel calculado del programa	$\hat{L} = 2n_2/n_1N_2$
Nivel del lenguaje	$\lambda = LV^*$
Contenido inteligente	$I \simeq V^*$
Esfuerzo	$E = V/L$

FIGURA 7.9.

el número total de discriminaciones mentales requeridas para codificar un determinado programa. Su valor, que podría tomarse como el equivalente a una medida de la complejidad de programación es tanto mayor cuanto mayores son el tamaño del vocabulario y la longitud del programa (ambos se traducen en el valor de V) y más próximo a la máquina el lenguaje empleado. Diversos estudios empíricos demuestran la existencia de una fuerte correlación entre E y el tiempo necesario para la programación, y asimismo entre E y el número de errores hallados en los programas.

En cuanto al contenido de inteligencia I del programa, esta función tiende a ser constante e independiente del nivel de codificación.

Si aplicamos el cuadro anterior de definiciones al programa (5) obtenemos los valores de la figura 7.10.

$n_1 = 13$
$N_1 = 24$
$n_2 = 11$
$N_2 = 25$
$n = 24$
$N = 49$
$V = 49 \cdot \log_2 24 = 224{,}66$
$n^* = 2 + 2 = 4$
$V^* = 4 \log_2 4 = 8$
$\hat{N} = 13 \log_2 13 + 11 \log_2 11 = 86{,}16$
$L = 4 \log_2 4/49 \log_2 24 = 0{,}0356$
$\hat{L} = 2 \cdot 11/13 \cdot 25 = 0{,}0676$
$\lambda = (4 \log_2 4/49 \log_2 24) \cdot 4 \log_2 4 = 0{,}2848$
$I \simeq 4 \log_2 4 = 8$
$E = 49 \log_2 24 \div 4 \log_2 4/49 \log_2 24 = 6.310{,}67$

FIGURA 7.10.

5.2. Número ciclomático

La métrica que brevemente describiremos ahora se basa en la estructura decisional (flujo de predicados) de los programas. A esta estructura se le asocia un grafo orientado G. Recuérdese la figura 1.4 y las correspondencias allí establecidas entre bloques de código y ramas de programa con nodos y arcos.

Por la teoría de grafos, se calcula el número de circuitos de control linealmente independientes en G. Un grafo fuertemente conectado es aquel para el que existe un camino entre cualquier pareja de nodos.

El número ciclomático $V(G)$ de un grafo con n nodos, e arcos y p componentes conectados es:

$$V(G) = e - n + 2p \tag{6}$$

Existe un teorema que dice lo siguiente: en un grafo G fuertemente conectado, el número ciclomático es igual al número máximo de circuitos linealmente independientes.

Así pues, el número ciclomático da una medida de la complejidad del software, entendida como dificultad mental en la tarea de programación.

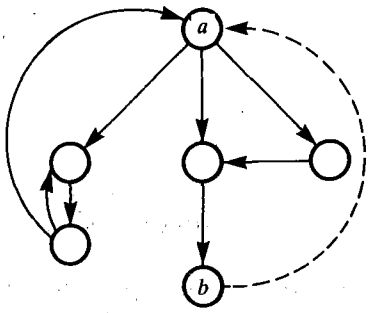


FIGURA 7.11.

El grafo de la figura 7.11 corresponde a un programa cualquiera, cuyos nodos de entrada y de salida son a y b. Si se cierra de forma ficticia por el arco b-a, el grafo resulta ser fuertemente conectado. Su número ciclomático es 5 ($e = 9, n = 6, p = 1$). El grafo de la figura 1.4 tiene un ciclomático de 4. El grafo de la figura 7.12, tiene un ciclomático de 8.

6. RESUMEN

Existe un conjunto bastante grande de algoritmos demasiado difíciles de llevar a la práctica por limitaciones de tiempo o espacio.

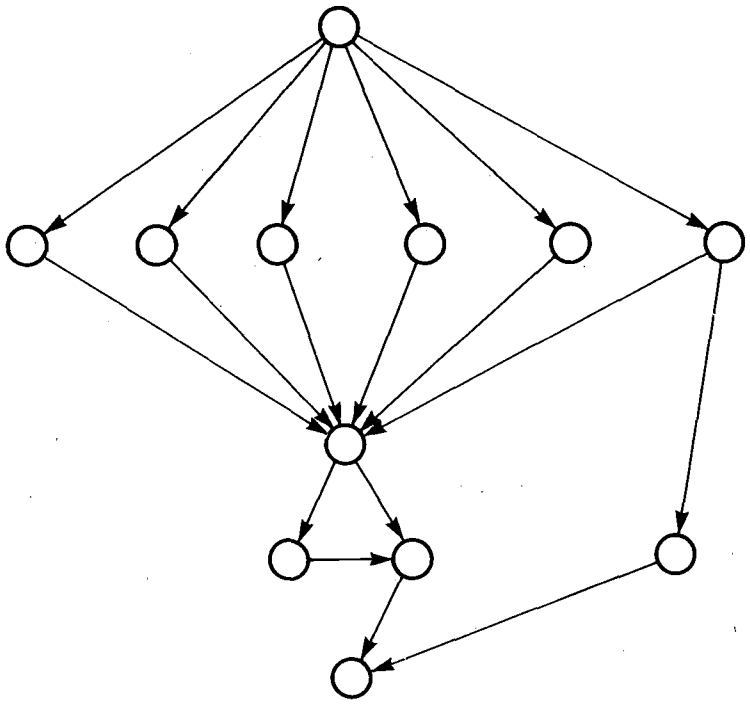


FIGURA 7.12.

La complejidad de un algoritmo no depende de la velocidad del ordenador, de su capacidad de memoria o formato de instrucciones, es decir, es independiente del hardware.

Existen numerosas clases de complejidad según sean los algoritmos deterministas —los pasos del algoritmo son siempre los mismos— o no deterministas —se toman decisiones que pueden variar la cantidad de pasos ejecutados.

Los algoritmos que pueden ejecutarse en un tiempo de orden polinómico en función de la magnitud de entrada, se consideran eficientes y constituyen la clase P, mientras que aquéllos que requieren algoritmos de tipo exponencial se consideran difíciles o complejos.

Algunos problemas pueden resolverse eficientemente con algoritmos no deterministas pero su solución determinista es compleja. A este tipo de algoritmo se le llama NP. En ciertos casos, un algoritmo NP es tal que todo problema NP puede simularse con él, en cuyo caso decimos que el algoritmo es NP-completo.

Si lográramos demostrar que un algoritmo NP-completo puede simularse mediante un algoritmo determinista eficiente —es decir, de P—, se demostraría automáticamente que todos los problemas NP pueden simularse con un algoritmo determinista eficiente.

Los estudios realizados hasta el momento no logran, sin embargo, aclarar este dilema llamado «problema P-NP», siendo una tendencia actual la de considerar que la

clase NP es diferente de la clase P. De esta forma pueden centrarse los estudios en la búsqueda de métodos no deterministas para resolver óptimamente toda esta gama de problemas.

En paralelo con la complejidad algorítmica surge una preocupación por la complejidad del software, correspondiente a una rama mucho menos teórica de la informática, conocida hoy por el nombre de ingeniería del software.

En este capítulo se han presentado dos clases de métrica, la primera de carácter lingüístico basada en la contabilidad de los operadores y operandos utilizados en la codificación de los programas, y la segunda, en la evaluación del grafo asociado a la estructura de control de los programas. Cualquiera de estas dos clases de técnicas y otras muchas que se han elaborado sólo son aplicables en el ámbito de los lenguajes clásicos de tipo imperativo y de los ordenadores de funcionamiento secuencial según el modelo de von Neumann.

Tanto el campo de la complejidad algorítmica como el de la complejidad del software siguen abiertos y muy activos.

7. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

El estudio de la complejidad algorítmica puede considerarse bastante reciente. Parece que el concepto de complejidad temporal se origina hacia 1965 en un artículo de Hartmanis y Stearns. La complejidad espacial se debe a Hartmanis, Lewis y Stearns, por un trabajo presentado a un simposio de Teoría y Diseño de circuitos lógicos en 1965, en el que estos autores analizaban la limitación de los cálculos por causa de la capacidad de memoria (Hopcroft y Ullman, 1979).

El número de investigadores y trabajos ha crecido y se ha diversificado sostenidamente desde entonces, por lo que su simple mención alargaría esta nota sin mayor provecho.

Sin embargo, no puede dejarse de citar a Cook, por su formulación de la clase de problemas NP-completos y del problema P-NP. Esto sucedió en 1971. Al año siguiente y sucesivos, Karp y otros matemáticos enunciaron listas de problemas NP-completos, entre los que se alinean todos los presentados en el subapartado 4.2.

Es Karp precisamente quien, en su conferencia de aceptación del premio Turing (Karp, 1986), ha expuesto una panorámica histórica de algunos de los hitos en la teoría de complejidad algorítmica, que merece ser leída. Un aliciente mayor adicional para aquellos lectores que deseen conocer un auténtico marco conceptual de los problemas planteados a la Informática Teórica en el eje computabilidad-complejidad lo constituye un rompecabezas resuelto por Karp, cuyas piezas son los mencionados problemas. Lo publica el mismo número de la revista *Communications of the A.C.M.*, que recoge la citada conferencia (Frenkel, 1986).

Díaz (1982) y Balcázar, en (Gamella, 1985, pp. 61-65) describen algunos de los campos a los que se extienden hoy día los trabajos sobre complejidad algorítmica. Citando a este último, nos encontramos con parcelas dedicadas: 1) al desarrollo de técnicas de análisis mediante funciones continuas interpoladoras del comportamiento asintótico de algoritmos dados; 2) al diseño de estructuras de datos eficientes, analizando el tiempo requerido por los distintos algoritmos; 3) al planteamiento de

problemas de investigación operativa (tales como búsqueda de rutas mínimas que permitan recorrer diversos puntos y otras funciones de coste) en términos combinatorios; 4) al análisis de comportamientos «caso medio»; 5) al análisis de algoritmos para problemas de tipo geométrico, frecuentes en las áreas de robótica (algoritmos para decidir formas y movimientos) y del diseño de circuitos integrados a gran escala (algoritmos para integrar en una misma pastilla de silicio una enorme cantidad de componentes y conectarlos adecuadamente con el mínimo posible de interconexiones cableadas).

En la parte de complejidad algorítmica, este capítulo ha sido confeccionado con los materiales siguientes.

Las grandes líneas del capítulo se han inspirado en una mezcla de los libros de Goldschlager y Lister (1982) y de Hopcroft y Ullman (1979), ayudada por un trabajo de síntesis de Díez Medrano (1984). El primero de estos libros es una introducción a la informática y presenta el tema de una manera mucho más intuitiva que el segundo, escrito en un nivel alto y considerablemente formalizado.

Para las definiciones de complejidad espacial y temporal en máquinas de Turing hemos seguido muy de cerca a Hopcroft y Ullman en su capítulo 12. También, los resultados del subapartado 2.4 han sido extractos de este libro. En gran parte, asimismo el apartado 4 es tributario de dichos autores a través de la versión de Díez Medrano.

Los ejemplos de algoritmos de búsqueda y ordenación se encuentran, con todos los detalles de codificación (declaración de variables y demás especificaciones) y otros muchos ejemplos, en un texto didáctico sobre algoritmos y complejidad, orientado a la enseñanza de la programación debido a Garijo y otros autores (Garijo *et al.*, 1985).

Por lo que respecta a la complejidad del software, los trabajos se inician aún más recientemente. Si no estamos equivocados, fue hacia 1972 cuando Halstead comenzó a publicar, bajo forma de informes técnicos en la Universidad norteamericana de Purdue, unas primeras aportaciones que, primero bajo los nombres de «Termodinámica de algoritmos», «Leyes naturales que controlan la estructura de los algoritmos» y otros parecidos, derivaron después en «Física del software» y finalmente, en «Ciencia del software». Halstead se inspiró en investigaciones sobre lenguajes naturales de Zipf en 1949 (seguidas y formalizadas posteriormente por Mandelbrot) y de Shannon sobre teoría matemática de la información.

La aplicación del número ciclomático a la complejidad de programas data de 1976 y se debe a Th. McCabe.

El cuadro de la figura 7.8 puede encontrarse, junto a las propuestas de Zipf y Mandelbrot en Shooman (1983, cap. 3).

Para la presentación de los postulados de la métrica de Halstead hemos utilizado el ejemplo (5), reproducido del capítulo 2 del libro de Shooman (1983), algunos elementos del artículo de Coulter (1983) y el libro en el que finalmente Halstead resumió su teoría (Halstead, 1977).

En cuanto al número ciclomático, nos hemos valido del artículo original de su propio autor (McCabe, 1976).

8. EJERCICIOS

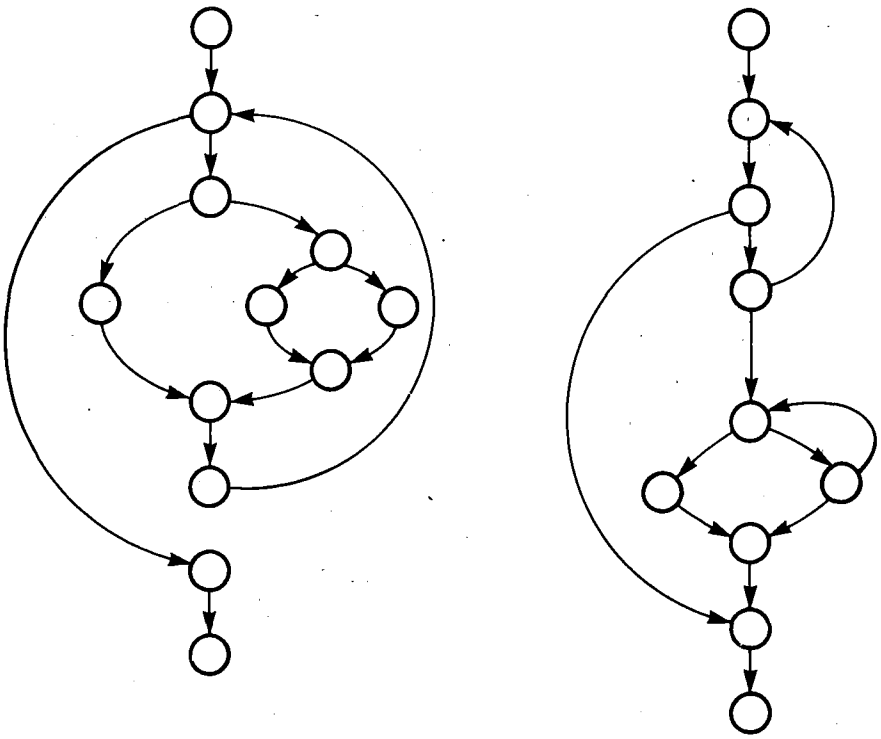
8.1. Buscar en la bibliografía sobre algoritmos de búsqueda y clasificación una selección de éstos, familiarizarse con ellos y con su complejidad, comparativamente a los vistos en este capítulo.

8.2. Aplicar la métrica de los operandos y los operadores a los códigos (2), (3) y (4) escritos en Pascal.

8.3. Tomar cualquiera de los algoritmos representados en (2), (3) y (4), codificarlo en un lenguaje ensamblador. Aplicar al código obtenido la métrica de Halstead y comparar los resultados con los resultados correspondientes al código en Pascal.

8.4. Aplicar la métrica del número ciclomático a los mismos algoritmos (2), (3) y (4).

8.5. Aplicar la métrica del número ciclomático a los siguientes grafos asociados a sendos programas.



8.6. Aplicar la métrica del número ciclomático a los programas de las figuras 4.3 y 4.5, comparar y extraer alguna conclusión.

Cuarta parte
LENGUAJES

Capítulo 1
IDEAS GENERALES

1. LENGUAJES E INFORMÁTICA

Ya al comienzo de este libro (tema «Lógica», capítulo 1, apartado 3) hemos definido informalmente un lenguaje como un sistema de símbolos que permite la comunicación entre personas, entre personas y máquinas o entre máquinas. Hemos introducido también (en el apartado 4, que debería releerse ahora) la terminología y los primeros conceptos de la teoría de lenguajes formales.

La importancia de la teoría de lenguajes en informática radica en la correspondencia biunívoca que existe entre máquinas programables y lenguajes. A cada máquina corresponde un lenguaje en el que se escriben sus programas; a la inversa, a cada lenguaje de programación le corresponde una máquina que interpreta los programas escritos con él*. La teoría de lenguajes permite desarrollar de manera científica tanto la creación y la producción de programas como el diseño de máquinas y lenguajes de programación.

Los tres capítulos centrales del tema que ahora comenzamos están dedicados a presentar los conceptos básicos de la teoría de lenguajes formales y su relación con la teoría de autómatas. Después, en el último capítulo, veremos la utilidad de estos conceptos en los lenguajes de programación de ordenadores.

2. DESCRIPCIONES DE LOS LENGUAJES

La descripción formal de un lenguaje finito es inmediata: basta enumerar las cadenas que lo forman. Pero si es infinito, será preciso inventar una descripción finita;

* El término «máquina» debe entenderse aquí en un sentido abstracto. Por ejemplo, una «máquina BASIC» es una «máquina virtual» cuyo lenguaje de máquina es el BASIC y que se materializa (se «implementa») mediante una máquina real (un ordenador) y unos programas adecuados.

esta descripción será, a su vez, una cadena de símbolos combinados de acuerdo con ciertas reglas (sintaxis) y con un determinado significado, tanto para los símbolos como para las reglas (semántica). Por tanto, esta cadena de símbolos pertenecerá a un *metalenguaje,* que servirá para describir nuestro lenguaje. Por ejemplo, una expresión regular (tema «Autómatas», capítulo 4) es una cadena del lenguaje de las expresiones regulares, que es un metalenguaje para describir los lenguajes regulares.

Ahora bien, podemos preguntarnos si, dado un lenguaje infinito cualquiera, $L \subset A^*$, es posible siempre describirlo de manera finita. La respuesta es «no». En efecto, sabemos (tema «Algoritmos», capítulo 6, apartado 3.1) que A^* (conjunto de todas las cadenas construidas sobre un alfabeto A) es recursivamente numerable (a cada cadena se le puede asociar, por ejemplo, su número de Gödel), y, por tanto, cualquier $L \subset A^*$ también lo es. Supongamos que existe un metalenguaje $L_M \subset A_M^*$ tal que cada cadena de L_M es una representación finita de un lenguaje sobre A^*, es decir, tal que a cada $L(M) \subset A^*$ corresponde al menos un $x_M \in L_M$. Si tal metalenguaje existiera debería ser, como todo lenguaje, recursivamente numerable. Esto implicaría que el conjunto de todos los $L \subset A^*$ fuera recursivamente numerable. Sin embargo, existe un teorema de la teoría de conjuntos según el cual el conjunto de los subconjuntos de un conjunto recursivamente numerable (y A^* lo es) no es recursivamente numerable, lo que contradice la conclusión anterior y refuta la hipótesis de que exista un L_M para todo $L \subset A^*$. Por consiguiente, podemos concluir que *no todos los posibles lenguajes tienen una representación finita en un metalenguaje.*

En los dos capítulos siguientes estudiaremos la descripción de ciertos tipos de lenguajes desde un punto de vista *generativo:* una *gramática, G,* es una descripción metalingüística con la que se puede desarrollar un algoritmo enumerativo para generar las cadenas del lenguaje $L(G)$, es decir, se puede diseñar una máquina de Turing que haga corresponder el número 1 a la cadena x_1, el 2 a la x_2, etc., de manera que genere el conjunto recursivamente numerable $L(G) = \{x_1, x_2, \ldots\}$.

En el capítulo 4 consideraremos un punto de vista complementario: el del *reconocimiento,* o sea, el estudio de algoritmos o estructuras de máquinas que permiten, dados un lenguaje L y una cadena x, determinar si $x \in L$ o $x \notin L$.

Finalmente, en el capítulo 5 abordaremos el aspecto *interpretativo:* un programa se escribe con el objetivo de que la máquina a la que va destinado *ejecute* un determinado algoritmo. Para que ello sea posible es necesario que el programador sepa con exactitud cómo la máquina va a reaccionar frente a cada una de las instrucciones de su programa, es decir, qué *significan* (y entramos así en el campo de la semántica) para la máquina las distintas cadenas del lenguaje.

3. SINTAXIS, SEMÁNTICA Y PRAGMÁTICA

Actualmente, en la teoría de lenguajes formales, el campo de la sintaxis está bien establecido: como veremos en el capítulo siguiente, existen metalenguajes (gramáticas) formales que describen con precisión varios tipos de lenguajes y que resultan de gran utilidad práctica en informática para el diseño de lenguajes de programación y para el reconocimiento, traducción e interpretación de programas.

La semántica, hasta ahora, ha resultado más difícil de formalizar. En la definición

de casi todos los lengujes de programación, la semántica se expresa de manera informal, mediante explicaciones verbales en sus manuales sobre lo que significa cada posible sentencia. La formalización de la semántica es, sin embargo, importante desde un punto de vista práctico. Con una definición formal de la semántica de un lenguaje se podrían elaborar métodos sistemáticos de comprobación de programas, es decir, de verificar, antes de su ejecución, que los programas son correctos, en el sentido de que van a hacer lo que se pretende que hagan. Y, eventualmente, se podrían diseñar herramientas (es decir, otros programas) basadas en estos métodos para automatizar la verificación de programas. Ello contribuiría a mejorar la productividad de los procesos de producción de software y la fiabilidad de los programas. Se han propuesto varios enfoques para definir formalmente la semántica de los lenguajes de programación (los comentaremos en el capítulo 5), pero éste es todavía un asunto sometido a numerosos trabajos de investigación.

En algunos textos sobre lenguajes de programación se habla de «pragmática», pero aún estamos lejos, no ya de disponer de una formalización, sino siquiera de estar de acuerdo sobre lo que representa: para unos, se refiere a los aspectos de comunicación con el usuario, para otros atañe a los problemas que se presentan cuando se pasa de la definición formal de un lenguaje a las particularidades de las máquinas en las que los programas han de ejecutarse, según otros trataría de los «aspectos prácticos» de la actividad de programar.

4. DOS EJEMPLOS

Los ejemplos que desarrollamos a continuación nos servirán para introducir de una manera intuitiva algunos de los conceptos que formalizaremos en capítulos posteriores.

Puesto que aún no hemos definido lo que es una gramática, haremos uso de una que, más o menos vagamente, todos conocemos: la gramática del lenguaje castellano.

Para describir la formación de sentencias (oraciones) utilizaremos *categorías sintácticas o sintagmas,* como «nombre», «verbo», etc. (o sintagma nominal, sintagma verbal, etc.). Está claro que, si bien «nombre» es un nombre, «verbo» no es un verbo. Esto es debido a que para describir el lenguaje castellano utilizaremos como metalenguaje el propio castellano, y ponemos las comillas para destacar que la palabra en cuestión se utiliza como elemento del metalenguaje (es la diferencia entre *uso* y *mención* del lenguaje, ver tema «Lógica», capítulo 1, apartado 4.3). La notación habitualmente seguida no es poner comillas, sino encerrar la palabra entre paréntesis angulares: ⟨nombre⟩, ⟨verbo⟩, etc.

Una sentencia castellana sintácticamente correcta estará compuesta por palabras concretas pertenecientes a las diversas categorías sintácticas (uno o varios sintagmas nominales, uno o varios sintagmas predicativos, etc.) combinadas de acuerdo con ciertas reglas.

Por ejemplo, una regla puede ser:

(1) *Para formar una sentencia, póngase un sintagma nominal y a continuación un sintagma predicativo.*

Y otras:

(2) *Un nombre propio es un sintagma nominal.*
(3) *«España» es un nombre propio.*
(4) *Un sintagma predicativo puede formarse con una forma verbal transitiva y un sintagma nominal.*
(5) *«es» es una forma verbal transitiva.*
(6) *Un sintagma nominal puede formarse con un determinante y un nombre común.*
(7) *Un artículo es un determinante.*
(8) *«un» es un artículo.*
(9) *«estado» es un nombre común.*

Del conjunto de estas nueve reglas puede deducirse que

España es un estado

es una sentencia del castellano.

Las reglas anteriores pueden expresarse formalmente así*:

(1) ⟨Sentencia⟩ → ⟨SN⟩ ⟨SP⟩
(2) ⟨SN⟩ → ⟨Npr⟩
(3) ⟨Npr⟩ → España
(4) ⟨SP⟩ → ⟨FVtr⟩ ⟨SN⟩
(5) ⟨FVtr⟩ → es
(6) ⟨SN⟩ → ⟨Det⟩ ⟨Ncom⟩
(7) ⟨Det⟩ → ⟨Art⟩
(8) ⟨Art⟩ → un
(9) ⟨Ncom⟩ → estado.

Y la sentencia obtenida puede descomponerse sintácticamente de acuerdo con las reglas; esta descomposición puede indicarse mediante corchetes etiquetados:

[	[España]]	[	[es]	[	[	[un]]	[estado]]]	
SN	Npr	SP	FVtr	SN	Det	Art	Ncom	

o bien, de una forma más gráfica, con un árbol llamado *árbol de derivación o árbol sintáctico* (figura 1.1).

* El símbolo «→» no tiene ahora nada que ver con el condicional de la lógica. En este contexto significa: «la parte de la izquierda puede sustituirse por la parte de la derecha», o «la parte izquierda puede tener la forma de la parte derecha».

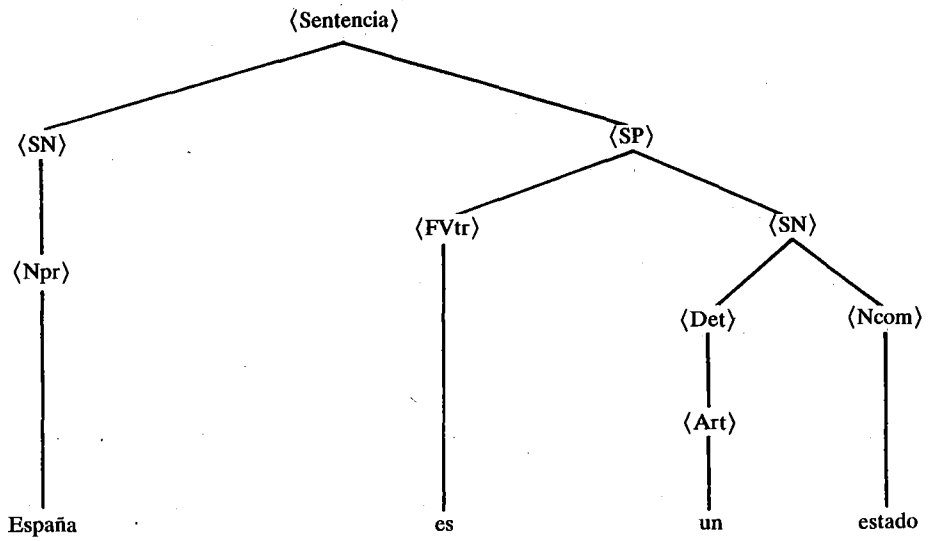


FIGURA 1.1.

Como segundo ejemplo, consideremos la sentencia

La soberanía nacional reside en el pueblo español.

Su descomposición sintáctica, o derivación a partir de las reglas, es la indicada por el árbol de la figura 1.2.

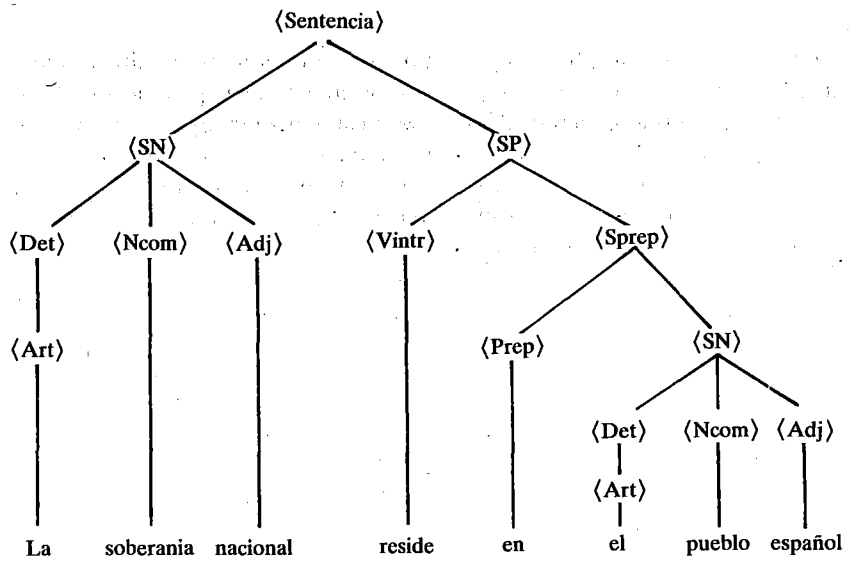


FIGURA 1.2.

Las reglas utilizadas para esta derivación han sido:

(1) ⟨Sentencia⟩ → ⟨SN⟩ ⟨SP⟩
(2) ⟨SN⟩ → ⟨Det⟩ ⟨Ncom⟩ ⟨Adj⟩
(3) ⟨Det⟩ → ⟨Art⟩
(4) ⟨SP⟩ → ⟨Vintr⟩ ⟨Sprep⟩
(5) ⟨Sprep⟩ → ⟨Prep⟩ ⟨SN⟩
(6) ⟨Art⟩ → la
(7) ⟨Art⟩ → el
(8) ⟨Ncom⟩ → soberanía
(9) ⟨Ncom⟩ → pueblo
(10) ⟨Adj⟩ → nacional
(11) ⟨Adj⟩ → español
(12) ⟨Vintr⟩ → reside
(13) ⟨Prep⟩ → en

Se obtiene una sentencia partiendo de la regla (1) y aplicando las demás hasta que resulta una cadena con sólo *símbolos terminales,* en este caso, las palabras concretas del castellano obtenidas por aplicación de las *reglas terminales* (6) a (13).

Con estas reglas pueden derivarse otras sentencias, como

«El pueblo español reside en la sobernía nacional»,
«La pueblo nacional residen en la pueblo español»,,

etc., que serían sentencias correctas en la gramática definida por tales reglas.

5. RESUMEN

La teoría de lenguajes formales es una herramienta matemática que permite abordar con rigor el diseño de lenguajes de programación y la producción de software. Si bien existe ya una buena elaboración de la misma en el campo de la sintaxis, queda aún mucho por hacer en cuanto a la semántica, y éste es un aspecto de gran trascendencia práctica y económica: no basta con que los programas sean sintácticamente correctos; es preciso que la máquina que los interpreta haga exactamente lo que el programador pretende que haga.

Capítulo 2
GRAMATICAS Y LENGUAJES

1. DEFINICIÓN DE GRAMÁTICA

Una gramática es una cuádrupla

$$G = \langle E_T, E_A, P, S \rangle,$$

donde

E_T es un conjunto finito llamado *alfabeto principal* o *alfabeto de símbolos terminales**.

E_A es un conjunto finito llamado *alfabeto auxiliar* o *alfabeto de variables.*

P es un conjunto finito de pares ordenados (α, β), donde $\alpha \in (E_T \cup E_A)^+$ y $\beta \in (E_T \cup E_A)^*$. Estos pares se llaman *reglas de escritura* o *producciones,* y generalmente se escriben con la notación $\alpha \rightarrow \beta$. α es el *antecedente* de la regla y β el *consecuente.* Si $\beta \in E_T^*$ se dice de la regla que es una *regla terminal.*

$S \in E_A$ es un símbolo destacado del alfabeto auxiliar llamado *símbolo inicial.*

Llamaremos $E = E_T \cup E_A$; supondremos que $E_T \cap E_A = \varnothing$, y para distinguir los elementos de los diferentes conjuntos adoptaremos el convenio de utilizar:

a) letras mayúsculas del alfabeto latino para los elementos de E_A. (O bien, palabras del metalenguaje castellano entre paréntesis angulares);

* En adelante representaremos con «*E*» a un alfabeto, porque «*A*» se utilizará frecuentemente como uno de los símbolos del alfabeto auxiliar.

b) letras minúsculas del comienzo del alfabeto latino (a, b, c, ...), o cifras, para los elementos de E_T. (O bien, palabras del castellano);
c) letras minúsculas del final del alfabeto latino (w, x, y, z) para los elementos de E_T^*;
d) letras minúsculas del alfabeto griego para los elementos de E^+.

2. RELACIONES ENTRE CADENAS E^*

2.1. Relación de derivación directa, $\underset{G}{\Rightarrow}$

Si $(\alpha \rightarrow \beta) \in P$ y $\gamma, \delta \in E^*$, las cadenas $\gamma\alpha\delta$ y $\gamma\beta\delta$ están en la *relación de derivación directa* en la gramática G. Escribiremos entonces

$$\gamma\alpha\delta \underset{G}{\Rightarrow} \gamma\beta\delta$$

y diremos que la cadena $\gamma\beta\delta$ *deriva directamente* de la $\gamma\alpha\delta$, o bien que $\gamma\alpha\delta$ *produce directamente* $\gamma\beta\delta$ en la gramática G. (De ahí el nombre de producciones para los elementos de P.)

2.2. Relación de derivación, $\overset{*}{\underset{G}{\Rightarrow}}$

Dados $\alpha_1, \alpha_m \in E^*$, diremos que están en la *relación de derivación* en la gramática G si existen $\alpha_2, \alpha_3 \ldots \alpha_{m-1}$ tales que

$$\alpha_1 \underset{G}{\Rightarrow} \alpha_2;\ \alpha_2 \underset{G}{\Rightarrow} \alpha_3;\ \ldots\ \alpha_{m-1} \underset{G}{\Rightarrow} \alpha_m$$

Se escribirá entonces

$$\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_m,$$

diciendo que α_m deriva de α_1, o que α_1 produce α_m

Por convenio, $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha \quad \forall\alpha \in E^*$

Siempre que sea evidente que nos referimos a una determinada gramática, G, escribiremos $\Rightarrow$ y $\overset{*}{\Rightarrow}$ en lugar de $\underset{G}{\Rightarrow}$ y $\overset{*}{\underset{G}{\Rightarrow}}$

Obsérvese que ni $\Rightarrow$, ni $\overset{*}{\Rightarrow}$, son relaciones de equivalencia, ya que, en general, no son simétricas.

3. LENGUAJE GENERADO POR UNA GRAMÁTICA. EQUIVALENCIA DE GRAMÁTICAS

Una cadena $\xi \in E^*$ es una *forma sentencial* de la gramática G si existe una derivación que produce ξ a partir de S, es decir, si $S \overset{*}{\underset{G}{\Rightarrow}} \xi$. Si además ξ sólo contiene símbolos terminales entonces es una *sentencia* o cadena válida. El conjunto de sentencias que pueden generarse en una gramática G se llama *lenguaje generado por la gramática G*, es decir:

$$L(G) = \{x \mid x \in E_T^* \text{ y } S \overset{*}{\underset{G}{\Rightarrow}} x\}$$

Dos gramáticas, G_1 y G_2, son equivalentes si ambas generan el mismo lenguaje: $L(G_1) = L(G_2)$.

4. EJEMPLOS

Ejemplo 4.1

Sea la gramática definida por $E_T = \{0, 1\}$; $E_A = \{S\}$;

$$P = \{(S \rightarrow 000S111); \quad (0S1 \rightarrow 01)\}$$

La única forma de generar sentencias es aplicando cualquier número de veces la primera regla y terminado con una aplicación de la segunda:

$$S \Rightarrow 000S111 \quad \Rightarrow 000000S111111 \quad \Rightarrow \ldots \Rightarrow 0^{3n-1}0S11^{3n1} \Rightarrow 0^{3n}1^{3n}$$

Por consiguiente, el lenguaje generado es el conjunto infinito

$$L(G) = \{0^{3n}1^{3n} \mid n \geq 1\}$$

Si la segunda regla fuera $S \rightarrow 01$, el lenguaje sería

$$L(G) = \{0^{3n+1}\, 1^{3n+1} \mid n \geq 0\}$$

Como podrá comprobarse en otros ejemplos, no siempre es posible expresar de esa manera (por intensión) $L(G)$.

Ejemplo 4.2

$$E_A = \{S, A\}; \quad E_T = \{a, b\}$$

$$P = \{(S \rightarrow abAS); \; (abA \rightarrow baab); \; S \rightarrow a; \; A \rightarrow b\}$$

Aquí, $L(G)$ está compuesto por cadenas que contienen (abb) y ($baab$) intercambiándose y reproduciéndose cualquier número de veces, y terminando siempre la cadena con el símbolo a.

Ejemplo 4.3

$$E_A = \{S, A, B\} \ ; \ E_T = \{a, b\}$$

$$P = \left\{ \begin{array}{ll} S \to aB & A \to bAA \\ S \to bA & B \to b \\ \ldots \to a & B \to bS \\ A \to aS & B \to aBB \end{array} \right\}$$

El lenguaje generado es el conjunto de todas las cadenas de E_T^+ que tienen igual número de a que de b, pero la demostración en este caso no es tan inmediata.

Ejemplo 4.4

$$E_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$
$$E_A = \{\langle \text{número} \rangle \ , \ \langle \text{dígito} \rangle\}$$
$$S = \langle \text{número} \rangle$$

$$\mathrm{P} = \left\{ \begin{array}{lll} \langle \text{número} \rangle \to \langle \text{dígito} \rangle \langle \text{número} \rangle & & (1) \\ \langle \text{número} \rangle \to \langle \text{dígito} \rangle & & (2) \\ \langle \text{dígito} \rangle \to 0 & & (3) \\ \langle \text{dígito} \rangle \to 1 & & (4) \\ \langle \text{dígito} \rangle \to 2 & & (5) \\ \langle \text{dígito} \rangle \to 3 & & (6) \\ \langle \text{dígito} \rangle \to 4 & & (7) \\ \langle \text{dígito} \rangle \to 5 & & (8) \\ \langle \text{dígito} \rangle \to 6 & & (9) \\ \langle \text{dígito} \rangle \to 7 & & (10) \\ \langle \text{dígito} \rangle \to 8 & & (11) \\ \langle \text{dígito} \rangle \to 9 & & (12) \end{array} \right\}$$

Damos a continuación algunos ejemplos de derivaciones de sentencias, poniendo bajo el símbolo $\Rightarrow$ el número de la regla o reglas utilizadas:

$\langle \text{número} \rangle \underset{2}{\Rightarrow} \langle \text{dígito} \rangle \underset{3}{\Rightarrow} 0$

$\langle \text{número} \rangle \underset{1}{\Rightarrow} \langle \text{dígito} \rangle \ \langle \text{número} \rangle \underset{12}{\Rightarrow} 9 \ \langle \text{número} \rangle \underset{2}{\Rightarrow} 9 \ \langle \text{dígito} \rangle \underset{12}{\Rightarrow} 99$

$\langle \text{número} \rangle \underset{1}{\Rightarrow} \langle \text{dígito} \rangle \ \langle \text{número} \rangle \underset{1}{\Rightarrow} \langle \text{dígito} \rangle \langle \text{dígito} \rangle \langle \text{número} \rangle \underset{1}{\Rightarrow}$
$\langle \text{dígito} \rangle \langle \text{dígito} \rangle \langle \text{dígito} \rangle \langle \text{número} \rangle \underset{2}{\Rightarrow} \langle \text{dígito} \rangle \langle \text{dígito} \rangle$
$\langle \text{dígito} \rangle \langle \text{dígito} \rangle \overset{*}{\underset{4,5,6,7}{\Rightarrow}} 1234$

Como los símbolos terminales son los diez dígitos decimales, el lenguaje que se obtiene es el conjunto infinito de cadenas que representan en decimal a los números naturales.

Obsérvese que es la regla (1) la que permite obtener una cadena de cifras de cualquier longitud.

Ejemplo 4.5

$$E_T = \{a, b\};\ E_A = \{A, S\};$$

$$P = \left\{ \begin{array}{ll} S \rightarrow aS & (1) \\ S \rightarrow aA & (2) \\ A \rightarrow bA & (3) \\ A \rightarrow b & (4) \end{array} \right\}$$

Un análisis del tipo de sentencias que pueden derivarse nos lleva fácilmente a la conclusión de que todas terminan con el símbolo b, por aplicación de (4), y todas empiezan por a, pudiendo tener en medio cualquier número de a seguido de cualquier número de b. Obsérvese que para el lenguaje así generado puede darse una expresión regular: $L(G) = aa^*bb^*$.

5. CLASIFICACIÓN DE LAS GRAMÁTICAS Y DE LOS LENGUAJES

5.1. Gramáticas de tipo 0 ó no restringidas

La gramática definida de una manera general en el apartado 1 se llama *gramática de tipo 0 ó no restringida.* Recordemos que las reglas de escritura o producciones son de la forma $\alpha \rightarrow \beta$, con $\alpha \in E^+$ y $\beta \in E^*$, es decir, la única restricción es que no puede haber reglas de la forma $\lambda \rightarrow \beta$.

Introduciendo restricciones adicionales en las reglas se obtienen sucesivamente gramáticas cada vez más restringidas. Pasemos a definir y comentar la clasificación debida a Chomsky y aceptada universalmente.

5.2. Gramáticas de tipo 1 ó sensibles al contexto

En las *gramáticas de tipo 1* las reglas son de la forma:

$$\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$$

con $A \in E_A$; $\alpha_1, \alpha_2 \in E^*$; $\beta \in E^+$.

Es decir, A puede reemplazarse por β siempre que esté en el contexto de α_1 y α_2. (Obsérvese que α_1 ó α_2, o ambas, pueden ser la cadena vacía, λ).

Una propiedad importante de las gramáticas de tipo 1 es que *las cadenas que se van*

obteniendo en cualquier derivación son de longitud no decreciente. En efecto, al definir las reglas más arriba hemos especificado que $\beta \in E^+$, es decir, $\beta \neq \lambda$, por lo que lg $(A) = 1 \leq$ lg (β), y lg $(\alpha_1 A \alpha_2) \leq$ lg $(\alpha_1 \beta \alpha_2)$, o sea, que la longitud del consecuente nunca puede ser menor que la del antecedente. En el siguiente capítulo demostraremos que la inversa es también cierta, en el sentido de que, *si todas las reglas de una gramática cumplen la condición de no decrecimiento, se puede hallar una gramática equivalente con reglas sensibles al contexto.*

Salvo en el primero y el segundo, todas las gramáticas definidas en los ejemplos del apartado 4 son sensibles al contexto. En el ejemplo 4.1, es la regla $0S1 \rightarrow 01$ la que no cumple la condición, ya que sustituye S por λ en el contexto (0, 1). En cuanto al segundo ejemplo, la regla $abA \rightarrow baab$ no es del tipo sensible al contexto (lo sería si fuera $abA \rightarrow abab$); sin embargo, la longitud del antecedente es menor o igual que la del consecuente en todas las reglas, por lo que habrá una gramática equivalente sensible al contexto. En el siguiente capítulo veremos cómo se puede encontrar.

5.3. Gramáticas de tipo 2 ó libres de contexto

Las *gramáticas de tipo 2* son un caso particular de las de tipo 1, con $\alpha_1 = \alpha_2 = \lambda$, es decir, las reglas son del tipo

$$A \rightarrow \beta$$

con $A \in E_A$, $\beta \in E^+$.

Estas gramáticas, también llamadas *gramáticas de Chomsky* o *C-gramáticas* juegan un papel muy importante tanto en la lingüística como en la teoría de lenguajes de programación, por lo que ya antes de la clasificación propuesta por Chomsky fueron descubiertas por diversos autores a partir de puntos de vista bastante diferentes.

Ejemplos de gramáticas libres de contexto son el 4.3, el 4.4 y el 4.5.

5.4. Gramáticas de tipo 3 ó regulares

Las gramáticas de tipo 3, también llamadas *gramáticas de Kleene* o *K-gramáticas* son un caso particular de las gramáticas de tipo 2, con reglas de la forma

$$A \rightarrow aB \quad \text{o} \quad A \rightarrow a$$

con $A, B \in E_A$; $a \in E_T$.

Un ejemplo de K-gramática es el ejemplo 4.5. Obsérvese que en ese ejemplo podíamos representar el lenguaje mediante una expresión regular; esto no es casualidad: como veremos en el capítulo 4, *los lenguajes generados por las gramáticas de tipo 3 son exactamente los lenguajes regulares que estudiábamos en el tema «Autómatas»* (y de ahí que a las K-gramáticas también se les llame gramáticas regulares).

6. JERARQUÍA DE LENGUAJES

Llamaremos lenguaje de tipo 0 al generado por una gramática de tipo 0, lenguaje de tipo 1 ó sensible al contexto al generado por una gramática de tipo 1, lenguaje de tipo 2 ó libre de contexto o C-lenguaje al generado por una gramática de tipo 2, y lenguaje de tipo 3 o regular o K-lenguaje, o también, lenguaje de estados finitos al generado por una gramática de tipo 3.

Según se han definido las gramáticas, es evidente que toda gramática regular es libre de contexto, toda gramática libre de contexto es sensible al contexto, y toda gramática sensible al contexto es de tipo 0. Por consiguiente, si llamamos $\{L(G3)\}$, $\{L(G2)\}$, $\{L(G1)\}$ y $\{L(G0)\}$ a los conjuntos de lenguajes de cada tipo, tendremos que:

$$\{L(G3)\} \subset \{L(G2)\} \subset \{L(G1)\} \subset \{L(G0)\} \subset P(E^*)$$

7. LENGUAJES CON LA CADENA VACÍA

Es fácil constatar que, tal como se han definido las gramáticas, la cadena vacía, λ, no puede figurar en ningún lenguaje de tipo 1, 2 ó 3. Una gramática es, esencialmente, una expresión en un metalenguaje que permite dar una descripción finita de ciertos lenguajes (no de todos) definidos sobre E. Es obvio que si L tiene una descripción finita, $L_1 = L \cup \{\lambda\}$ también puede tenerla: bastará añadir de algún modo «λ también está en L_1» a la descripción de L, y esto puede hacerse agregando $S \to \lambda$ a las reglas de la gramática que describe a L.

Ahora bien, si habíamos impuesto a las reglas de las gramáticas de tipo 1, $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, la condición de que $\beta \neq \lambda$ era para conseguir la importante propiedad de no decrecimiento. Si ahora añadimos $S \to \lambda$, será preciso que S no aparezca en el consecuente de ninguna regla si queremos que se conserve tal propiedad. A este respecto, es importante el siguiente teorema:

Teorema 7.1. Si G es una gramática de tipo 1, 2 ó 3, puede encontrarse otra gramática equivalente, G_1, de tipo 1, 2 ó 3 respectivamente, tal que $L(G_1) = L(G)$, y tal que su símbolo inicial, S_1, no aparece en el consecuente de ninguna regla de G_1. Si $G = \langle E_T, E_A, P, S\rangle$,

$$G_1 = \langle E_T, E_A \cup \{S_1\}, P_1, S_1\rangle,$$

donde

$$P_1 = P \cup \{S_1 \to \alpha \mid (S \to \alpha) \in P\}$$

Demostración:

a) Supongamos que $S \overset{*}{\underset{G}{\Rightarrow}} x$, y sea $S \to \alpha$ la primera regla utilizada en esa derivación; entonces, $S \underset{G}{\Rightarrow} \alpha \overset{*}{\underset{G}{\Rightarrow}} x$. Por la definición de P_1, $(S_1 \to \alpha) \in P_1$, de modo que $S_1 \underset{G_1}{\Rightarrow} \alpha$, y como $P \subset P_1$, $\alpha \overset{*}{\underset{G_1}{\Rightarrow}} x$. Por consiguiente, $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} x$, y $L(G) \subset L(G_1)$.

b) Supongamos que $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} y$, siendo $S_1 \to \beta$ la primera regla utilizada en G_1:

$$S_1 \underset{G_1}{\Rightarrow} \beta \overset{*}{\underset{G_1}{\Rightarrow}} y$$

Si $S_1 \to \beta$ es una regla en G_1, en G deberá existir la regla $S \to \beta$, por lo que $S \underset{G}{\Rightarrow} \beta$. Por otra parte, P_1 se ha definido de modo que S_1 no aparezca en el consecuente de ninguna regla, por lo que no estará incluido en α ni aparecerá en ninguna de las formas sentenciales de la derivación $\beta \overset{*}{\underset{G_1}{\Rightarrow}} y$; entonces, esta derivación será también válida en G: $\beta \overset{*}{\underset{G}{\Rightarrow}} y$. Vemos así que $S \overset{*}{\underset{G}{\Rightarrow}} \beta$, y por tanto, $L(G_1) \subset L(G)$. Teniendo en cuenta el resultado anterior podemos afirmar que $L(G) = L(L_1)$.

c) Tal como se ha definido P_1, es inmediato comprobar que si G es una gramática de tipo 1, 2 ó 3, G_1 es de tipo 1, 2 ó 3 respectivamente.

En virtud este teorema, dada una gramática cualquiera de tipo 1, 2 ó 3, G, podemos pasar a G_1, y de ésta a G_2 añadiendo la regla $S_1 \to \lambda$, con lo que tendremos $L(G_2) = L(G) \cup \{\lambda\}$, con G_2 del mismo tipo que G y de longitud no decreciente.

Ejemplo:

Sea G, definida por $E_A = \{S\}$; $E_T = \{0, 1\}$; $P = \{(S \to 0S1); (S \to 01)\}$. El lenguaje es $L(G) = \{0^n 1^n \mid n \geq 1\}$. Podemos construir G_1, con $E_{A_1} = \{S, S_1\}$; $E_{T_1} = E_T$; $P_1 = \{(S_1 \to 0S_1); (S_1 \to 01); (S \to 0S1); (S \to 01)\}$, siendo ahora S_1 el símbolo inicial; es fácil comprobar que $L(G_1) = L(G)$. Entonces, G_2 tendrá $E_{A_2} = E_{A_1} = \{S, S_1\}$; $E_{T_2} = E_T = \{0, 1\}$; $P_2 = \{(S_1 \to 0S1); (S_1 \to 01); (S \to 0S_1); (S \to 01); (S_1 \to \lambda)\}$, con lo que $L(G_2) = L(G) \cup \{\lambda\} = \{0^n 1^n \mid n \geq 0\}$.

De una manera general, del Teorema 7.1 se obtiene el siguiente

Corolario. Si L es un lenguaje de tipo 1, 2 ó 3, entonces $L \cup \{\lambda\}$ y $L - \{\lambda\}$ son lenguajes de tipo 1, 2 ó 3 respectivamente.

8. RESUMEN

En este capítulo hemos definido los conceptos de gramática, lenguaje generado por una gramática y gramáticas equivalentes. Hemos visto la clasificación de las gramáticas según las restricciones impuestas a sus producciones y cómo esta clasificación da lugar a una jerarquía de lenguajes. Finalmente, hemos estudiado la manera de modificar una gramática para que el lenguaje contenga la cadena vacía sin cambiar de tipo.

9. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

La teoría de lenguajes tiene su origen en un campo inicialmente bastante alejado de la informática: la lingüística. Los lingüistas distinguen, tradicionalmente, entre

gramática particular (propiedades de lenguajes concretos, como frecuencia de vocablos, reglas sintácticas, etc.) y *gramática universal* (propiedades generales que puedan aplicarse a cualquier lenguaje humano) (Chomsky, 1967).

Los lingüistas de la llamada «escuela estructuralista americana» habían elaborado por los años 50 algunas ideas informales acerca de la gramática universal. Por ejemplo, si un lenguaje (natural) es un conjunto innumerable de frases, para describirlo debería establecerse una *gramática generativa* o conjunto de reglas que subyacen en la composición de frases correctas y una *descripción estructural* para cada frase que permitiese explicar cómo puede componerse tal frase a partir de la gramática. El primer trabajo sobre la formalización de estos conceptos fue obra de Chomsky (1956), quien sin duda es la figura más destacada de la lingüística moderna, tanto por desarrollar sus fundamentos matemáticos (Chomsky y Miller, 1958; Chomsky, 1959) como por sus teorías sobre el origen y la naturaleza de los lenguajes naturales (Chomsky, 1968, 1975), aunque éstas son más discutidas. (El lector puede encontrar, traducidos al español, dos libros del más conocido crítico de Chomsky: Luria (1974a, b).) Por ejemplo, las gramáticas generativas formalizadas permiten explicar el «carácter creativo» de los lenguajes naturales, es decir, el hecho de que dispongan de mecanismos recursivos que les permiten expresar un número potencialmente infinito de ideas, sentimientos, etc. La falta de un formalismo para estudiar estos mecanismos había inclinado previamente a ciertos lingüistas de la escuela conductista a negar tal propiedad, y a otros, como Saussure, a considerarla como algo ajeno al campo de la lingüística (Chomsky, 1967).

En el campo de la informática, poco después de la aparición de las primeras publicaciones de Chomsky, el concepto de gramática formal adquirió gran importancia para la especificación de los lenguajes de programación; concretamente, se definió formalmente la sintaxis del lenguaje ALGOL 60 (con ligeras modificaciones sobre su versión primitiva) mediante una gramática libre de contexto (Naur, 1963). Ello condujo rápidamente, de una manera natural, al diseño riguroso de algoritmos de compilación (*cf.*, p. ej., Randell y Russell, 1964).

Desde una perspectiva bastante ambiciosa, hace tiempo se piensa que, puesto que todos los lenguajes de programación son lenguajes artificiales susceptibles de formalización, la teoría de lenguajes podría ser la base de una «teoría general de la programación» (Harrison, 1965) o de una «ciencia de la programación» (Gries, 1981).

Finalmente, y enlazando con el campo de la lingüística, la teoría de lenguajes es de gran utilidad para el trabajo en un campo de la inteligencia artificial: el procesamiento de lenguajes naturales (comprensión, generación y traducción) (*cf.*, p. ej. Hirst, 1981; Tennant, 1981; Carbonell *et al.*, 1981).

En cuanto a orientaciones bibliográficas para estudiar con mayor profundidad la teoría de lenguajes formales, tres libros clásicos y recomendables son el de Hopcroft y Ullman (1969), el de Harrison (1978) y el de Gross y Lentin (1967). El ejemplo 4.2 (como también los ejemplos 4.3 y 5.4 del capítulo siguiente, relacionados con él) está tomado del primero, donde pueden encontrarse algunos detalles y demostraciones que aquí omitimos. En otra obra posterior de Hopcroft y Ullman (1979) se incluyen también aspectos de la teoría de la computabilidad, y en la misma línea se orienta el libro, más completo, de Denning *et al.* (1978).

10. EJERCICIOS

10.1. Muchas veces interesa que los árboles de derivación en una gramática sean binarios, es decir, que de cada nodo sólo puedan salir uno o dos arcos. ¿Cómo deberán ser las reglas de escritura para que esto sea posible? Modificar la gramática de los ejemplos del capítulo 1 para que sus árboles sean binarios.

10.2. Dada la gramática $E_A = \{S, A\}$; $E_T = \{0, 1\}$;

$P = \{(S \rightarrow 0A); (A \rightarrow 0A); (A \rightarrow 1S); (A \rightarrow 0)\}$,

a) ¿de qué tipo es?;
b) expresar de algún modo el lenguaje que genera;
c) hallar otra gramática que genere el mismo lenguaje más la cadena vacía.

10.3. Definir una gramática que permita generar todos los números racionales escritos en decimal con el formato: ⟨signo⟩⟨parte entera⟩⟨parte fraccionaria⟩.

10.4. Definir una gramática que permita generar «identificadores»: secuencias de letras y dígitos que empiezan siempre por una letra.

10.5. Definir gramáticas para describir los lenguajes de la lógica de proposiciones y de la lógica de predicados.

10.6. Modificar las gramáticas de los ejemplos del apartado 4 para que se obtengan los mismos lenguajes con la cadena vacía.

Capítulo 3
DE ALGUNAS PROPIEDADES DE LOS LENGUAJES FORMALES

1. INTRODUCCIÓN

En el estudio de los lenguajes se plantean problemas que tienen o no solución dependiendo del tipo de gramática. Por ejemplo, para una gramática, G, de reglas sensibles al contexto, el problema de saber si $L(G)$ es vacío, finito o infinito es, en general, indecidible, mientras que para una gramática libre de contexto es decidible, es decir, existe un algoritmo que, aplicado a G, produce una de las tres respuestas; naturalmente, este problema será indecidible para las gramáticas de tipo 0 y decidible para las regulares. Otro ejemplo es el de averiguar si dos gramáticas son equivalentes, problema sólo decidible para las gramáticas regulares.

Al aumentar el grado de generalidad de la gramática, es decir, al ir desde el tipo 3 hacia el tipo 0, el estudio se hace más complejo y aumenta el número de problemas indecidibles.

Desde el punto de vista de aplicación práctica tanto a los lenguajes naturales como a los de programación, son particularmente importantes las gramáticas libres de contexto. Este capítulo se dedica, esencialmente, a dos características asociadas a tales gramáticas: la *recursividad* y la *ambigüedad.*

La recursividad, como veremos, es una propiedad válida no sólo para las gramáticas libres de contexto, sino también para las sensibles al contexto, ya que es una consecuencia del no decrecimiento de las cadenas. La ambigüedad sólo puede definirse y estudiarse cómodamente para las gramáticas de tipo 2 (y 3), en las que pueden describirse las derivaciones mediante árboles.

En aras de la brevedad omitiremos las demostraciones de algunos teoremas, que pueden encontrarse en la bibliografía referenciada en el apartado 9.

2. NO DECRECIMIENTO EN LAS GRAMÁTICAS SENSIBLES AL CONTEXTO

En el capítulo anterior vimos que como consecuencia de la misma definición de reglas sensibles al contexto ($\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta\alpha_2$, $\beta \neq \lambda$) se desprende inmediatamente que la longitud de las cadenas derivadas por aplicación de tales reglas no puede disminuir. Nos proponemos ahora demostrar la inversa, que ya no es tan evidente:

Teorema 2.1. Dada una gramática G_1 con reglas de longitud no decreciente, puede encontrarse otra gramática G_2 equivalente a G_1 cuyas reglas son sensibles al contexto.

Demostración:

Sea $\alpha \rightarrow \beta$ una regla de G_1, y sea $\alpha = p_1p_2 \ldots p_e$ y $\beta = q_1q_2 \ldots q_{e+m}$ con $p_i, q_j \in E$ y $m \geq 0$. Sustituiremos esta regla por un conjunto de reglas sensibles al contexto tales que juntas producen la derivación $\alpha \overset{*}{\underset{G_2}{\Rightarrow}} \beta$ sin aumentar la capacidad generativa.

Para ello ampliamos el alfabeto auxiliar de G_1 con un símbolo R y le añadimos también un símbolo P_1 por cada $p_1 \in E_T$.

Sustituimos en principio

$$p_1p_2p_3 \ldots p_e \rightarrow q_1q_2q_3 \ldots q_{e+m}$$

por

$$\begin{aligned}
&p_1p_2p_3 \ldots p_{e-1}p_e \rightarrow Rp_2p_3 \ldots p_{e-1}p_e \\
&Rp_2p_3 \ldots p_{e-1}p_e \rightarrow Rq_2p_3 \ldots p_{e-1}p_e \\
&Rq_2p_3 \ldots p_{e-1}p_e \rightarrow Rq_2q_3 \ldots p_{e-1}p_e \\
&\ldots\ldots\ldots\ldots\ldots\ldots \quad \ldots\ldots\ldots\ldots\ldots\ldots \\
&Rq_2q_3 \ldots q_{e-1}p_e \rightarrow Rq_2q_3 \ldots q_{e-1} \ldots q_{e+m} \\
&Rq_2q_3 \ldots q_{e+m} \rightarrow q_1q_2 \ldots q_{e+m}
\end{aligned}$$

Es evidente que:

a) Con este conjunto de reglas se obtiene la derivación

$$p_1p_2 \ldots p_e \overset{*}{\Rightarrow} q_1q_2 \ldots q_{e+m}$$

b) Al ser R un símbolo auxiliar no pueden producirse más sentencias que las que pueda producir $\alpha \rightarrow \beta$.

c) Las reglas introducidas sustituyen un p_i por un símbolo o una cadena en un contexto. Por tanto, son del tipo sensible al contexto, salvo si $p_i \in E_T$. En este caso, sustituimos p_i por $P_i \in E_A$ e introducimos la regla terminal $P_i \rightarrow p_i$.

Ejemplo:

Consideremos el ejemplo 4.2 del anterior capítulo. Sustituiremos la regla $abA \rightarrow baab$ por:

$$\begin{aligned} abA &\rightarrow RbA \\ RbA &\rightarrow RaA \\ RaA &\rightarrow Raab \\ Raab &\rightarrow baab \end{aligned}$$

Sustituimos a y b por P_1 y P_2 e introducimos las correspondientes reglas terminales, con lo cual obtenemos la gramática equivalente con reglas sensibles al contexto:

$$E_T = \{a, b\};\ E_A = \{S, A, R, P_1\ P_2\};$$

$$P = \left\{ \begin{array}{llll} S & \rightarrow P_1P_2AS & S & \rightarrow P_1 \\ P_1P_2A & \rightarrow RP_2A & A & \rightarrow P_2 \\ RP_2A & \rightarrow RP_1A & P_1 & \rightarrow a \\ RP_1A & \rightarrow RP_1P_1P_2 & P_2 & \rightarrow b \\ RP_1P_1P_2 & \rightarrow P_2P_1P_1P_2 & & \end{array} \right\}$$

3. RECURSIVIDAD DE LOS LENGUAJES SENSIBLES AL CONTEXTO

Una gramática es un algoritmo para generar un lenguaje, pero un problema de gran importancia es el del reconocimiento, es decir, dados un lenguaje $L \subset E^*$ y una cadena $x \in E^*$, ¿existe un algoritmo para determinar si $x \in L$ ó $x \notin L$? Si, por ejemplo, L es un lenguaje de programación y x un programa, es importante poder determinar si el programa es sintácticamente correcto ($x \in L$) o no ($x \notin L$). El problema del reconocimiento está íntimamente ligado al concepto de recursividad (ver tema «Algoritmos», capítulo 6). En efecto, decir que un lenguaje $L \subset E^*$ es recursivo es lo mismo que decir que existe un algoritmo para calcular la función característica de todo $x \in E^*$, o, lo que es lo mismo, para determinar si $x \in L$ o $x \notin L$.

Teorema 3.1. Todo lenguaje generado por una gramática de reglas sensibles al contexto es recursivo.

Aunque no daremos una demostración de este importante teorema, puede fácilmente intuirse que es una consecuencia de la propiedad de no decrecimiento. En efecto, un lenguaje de tipo 0 es recursivamente numerable, ya que existe un algoritmo (la gramática de tipo 0) para generar sus elementos (sentencias). Dado un elemento $x \in L$, podemos compararlo con cada uno de los elementos generados hasta comprobar que coinciden, pero si $x \notin L$ habría que generar las infinitas sentencias para ver que efectivamente $x \notin L$; es decir, en general, un lenguaje de tipo 0 no es recursivo. Sin embargo, si el lenguaje cumple la propiedad de no decrecimiento (tipo 1 en adelante), tenemos un algoritmo para generar primero *todas* las sentencias de longitud 1, luego

las de longitud 2, etc.; si lg $(x) = \ell$, generaremos todas las sentencias de longitud ℓ, y si x no se encuentra entre ellas, entonces $x \notin L$.

Podemos preguntarnos si la inversa es también cierta, es decir, si todo lenguaje recursivo puede ser generado por una gramática sensible al contexto. La contestación es «no»:

Teorema 3.2. Existen lenguajes recursivos que no son sensibles al contexto.

4. ARBOLES DE DERIVACIÓN PARA LAS GRAMÁTICAS LIBRES DE CONTEXTO

En el capítulo 1 ya hemos utilizado árboles como un método gráfico para ilustrar las derivaciones en ciertas gramáticas.

Definición 4.1. Un *árbol* es un conjunto finito de *nodos* unidos por *arcos* orientados (diremos que un arco *sale* del nodo n_i y *entra* en el nodo n_j), cumpliéndose tres condiciones:

1. Existe un nodo, y sólo uno, llamado *raíz,* en el que no entra ningún arco.
2. Para todo nodo n_m existe una secuencia de arcos, y sólo una, tal que el primero sale de la raíz y entra en n_i, el segundo sale de n_i y entra en n_{i+1}, etc., y el último entra en n_m.
3. En cada nodo (salvo la raíz) entra un arco y sólo uno.

Un nodo n_j es *descendiente directo* (o «hijo») de otro nodo n_i si existe un arco que sale de n_i y entra en n_j. Un nodo n_m es *descendiente* de otro n_0 si existe una secuencia $n_1, n_2, \ldots, n_{m-1}$ tal que n_m es descendiente directo de n_{m-1}, n_{m-1} lo es de $n_{m-2}, \ldots, n_1$ lo es de n_0.

Se llaman *hojas* a los nodos que no tienen ningún descendiente.

Entre los nodos de un árbol se puede establecer una *relación de orden:* todos los descendientes directos de n se pueden ordenar de *izquierda a derecha,* y si n_1 está a la izquierda de n_2 todos los descendientes de n_1 estarán a la izquierda de n_2.

Definición 4.2. Dada una gramática libre de contexto $G = \langle E_A, E_T, P, S \rangle$, un árbol es un *árbol de derivación o árbol sintáctico* en G si:

1. Cada nodo tiene una etiqueta que es un símbolo de $E = E_A \cup E_T$: Eti$(n) \in E$.
2. Eti(raíz) $= S$
3. Si n no es una hoja, Eti$(n) \in E_A$.
4. Si $n_1, n_2, \ldots, n_k$ son todos los descendientes directos de n de izquierda a derecha y Eti$(n) = A$, Eti$(n_1) = \ell_1$, Eti$(n_2) = \ell_2, \ldots,$ Eti$(n_k) = \ell_k$, entonces $(A \rightarrow \ell_1 \ell_2, \ldots, \ell_k) \in P$.

Llamaremos *resultado* de un árbol de derivación a la cadena compuesta por las etiquetas de las hojas leídas de izquierda a derecha.

Ejemplo 4.3

$$E_A = \{S, A, B\} \; ; \; E_T = \{a, b\}$$

$$P = \left\{ \begin{array}{llll} S \to aB & (1) & A \to a & (5) \\ S \to bA & (2) & A \to bAA & (6) \\ S \to aBS & (3) & B \to b & (7) \\ S \to bAS & (4) & B \to aBB & (8) \end{array} \right\}$$

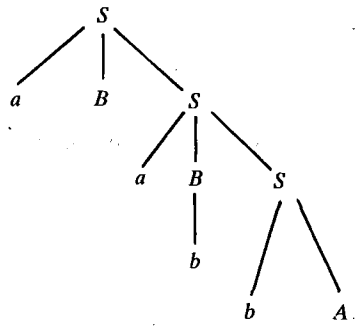


FIGURA 3.1.

El árbol de la figura 3.1 tiene como resultado la cadena *aBabbA*, y corresponde a la derivación

$$S \underset{3}{\Rightarrow} aBS \underset{3}{\Rightarrow} aBaBS \underset{7}{\Rightarrow} aBabS \underset{2}{\Rightarrow} aBabbA$$

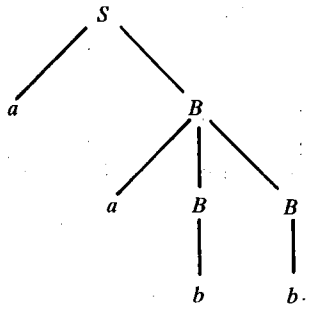


FIGURA 3.2.

El árbol de la figura 3.2 tiene como resultado la sentencia *aabb*, y corresponde a la derivación

$$S \underset{1}{\Rightarrow} aB \underset{8}{\Rightarrow} aaBB \underset{7}{\Rightarrow} aabB \underset{7}{\Rightarrow} aabb$$

Teorema 4.4. Dada una gramática libre de contexto, G, $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$ si y sólo si existe un árbol de derivación en G con resultado α.

5. AMBIGÜEDAD EN LAS GRAMÁTICAS LIBRES DE CONTEXTO

Definición 5.1. Una gramática libre de contexto, G, se dice que es ambigua si existe al menos una sentencia en $L(G)$ que puede obtenerse por dos o más derivaciones distintas, o lo que es lo mismo, le corresponden dos o más árboles diferentes.

Ejemplo 5.2

Los árboles de derivación suelen utilizarse en lingüística para el análisis sintáctico de oraciones. A la oración (ambigua) «Juan ve a Luis con gafas» puede corresponderle el árbol de la figura 3.3a o el de la figura 3.3b, según se interprete que es Juan o Luis, respectivamente, quien usa gafas.

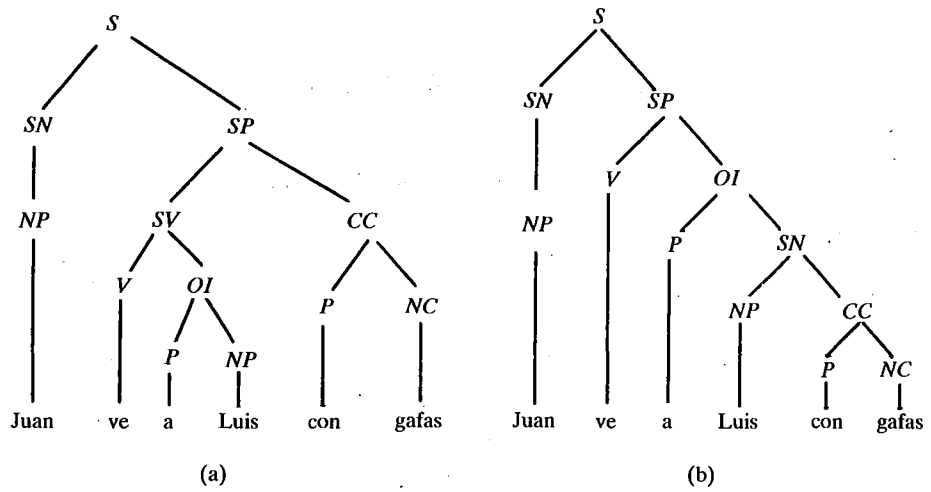


FIGURA 3.3.

Ejemplo 5.3

Consideremos la gramática definida por:

$$E_A = \{\langle \text{EA} \rangle\};\ E_T = \{|, +, *\};\ S = \langle \text{EA} \rangle$$

(«EA» significa «expresión aritmética»)

$$P = \left\{ \begin{array}{l} \langle \text{EA} \rangle \rightarrow \langle \text{EA} \rangle + \langle \text{EA} \rangle \\ \langle \text{EA} \rangle \rightarrow \langle \text{EA} \rangle * \langle \text{EA} \rangle \\ \langle \text{EA} \rangle \rightarrow \mid \langle \text{EA} \rangle \\ \langle \text{EA} \rangle \rightarrow \mid \end{array} \right\}$$

Si admitimos que una sucesión de n « | » representa el número natural n, esta gramática genera sentencias que representan combinaciones de los números naturales con las operaciones de suma y producto. Consideremos la sentencia |+| |*| | |; la derivación puede hacerse mediante el árbol de la figura 3.4.

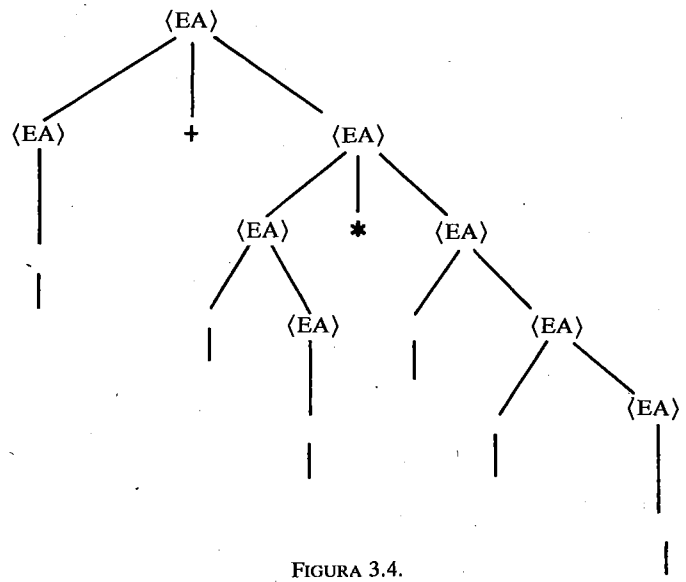


FIGURA 3.4.

O también mediante el de la figura 3.5.

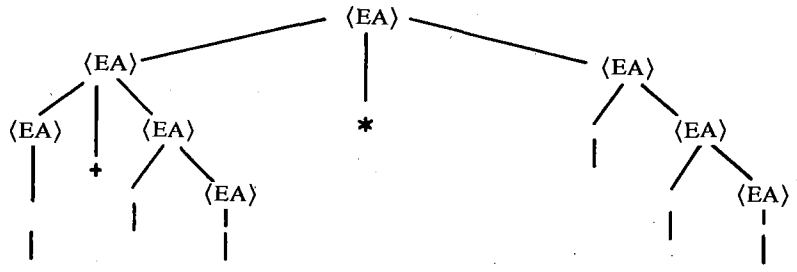


FIGURA 3.5.

El primero corresponde a la interpretación |+(| |*| | |) (prioridad de *), es decir, 7, mientras que el segundo corresponde a (|+| |) * | | | (prioridad de +), es decir 9.

Puede ocurrir que un mismo lenguaje pueda ser generado por una gramática ambigua y por otra gramática no ambigua. Un lenguaje es *inherentemente ambiguo* si todas las gramáticas que lo generan son ambiguas.

Ejemplo 5.4

La gramática del ejemplo 4.3 del capítulo 2 es ambigua. En efecto, la sentencia *aabbab*, por ejemplo, puede derivarse según indican los dos árboles de la figura 3.6.

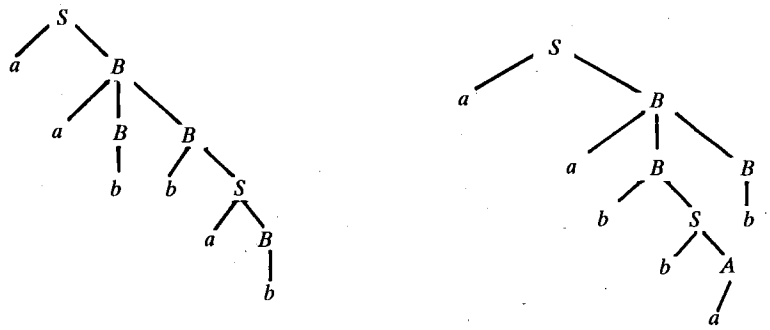


FIGURA 3.6.

Sin embargo, la gramática del ejemplo 4.3 de este capítulo es equivalente a ella y no ambigua, por lo que el lenguaje generado (conjunto de cadenas que tienen igual número de *a's* que de *b's*) no es inherentemente ambiguo. La derivación de la misma cadena en esta segunda gramática se representa por el árbol de la figura 3.7.

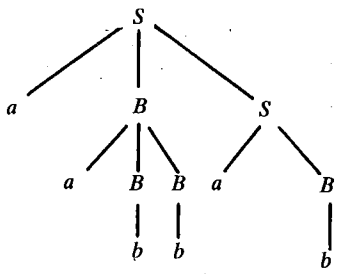


FIGURA 3.7.

Ejemplo 5.5

Una ambigüedad similar a la del ejemplo 5.3 puede ocurrir en las expresiones aritméticas decimales. Con una gramática definida por

$$E_A = \{\langle EA\rangle, \langle CTE\rangle, \langle DIG\rangle\}$$
$$E_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *\}$$
$$S = \{\langle EA\rangle\}$$

y las reglas

$$\langle EA\rangle \rightarrow \langle EA\rangle + \langle EA\rangle$$
$$\langle EA\rangle \rightarrow \langle EA\rangle * \langle EA\rangle$$
$$\langle EA\rangle \rightarrow \langle CTE\rangle$$
$$\langle CTE\rangle \rightarrow \langle CTE\rangle\langle DIG\rangle$$
$$\langle CTE\rangle \rightarrow \langle DIG\rangle$$
$$\langle DIG\rangle \rightarrow 0$$
$$\langle DIG\rangle \rightarrow 1$$
$$\langle DIG\rangle \rightarrow 2$$
$$\langle DIG\rangle \rightarrow 3$$
$$\langle DIG\rangle \rightarrow 4$$
$$\langle DIG\rangle \rightarrow 5$$
$$\langle DIG\rangle \rightarrow 6$$
$$\langle DIG\rangle \rightarrow 7$$
$$\langle DIG\rangle \rightarrow 8$$
$$\langle DIG\rangle \rightarrow 9$$

el lector puede fácilmente comprobar (de igual manera que en el ejemplo 5.3) la ambigüedad de, por ejemplo, «1 + 2 * 3».

Sabemos que una notación eficaz para eliminar la ambigüedad en casos como éste es el uso de los paréntesis. Podemos incluir esta posibilidad en nuestra gramática ampliando E_T con «(» y «)» y P con la producción «$\langle EA\rangle \rightarrow (\langle EA\rangle)$». Ahora, existe un árbol único para «1 + (2*3)» y otro para «(1 + 2)*3». Pero ello no impide que la sentencia «1 + 2*3» siga siendo válida en nuestra lenguaje y ambigua.

Sabemos también que la ambigüedad en una sentencia como «1 + 2*3» se elimina estableciendo una prioridad entre los operadores. Concretamente, se suele dar prioridad a «*» sobre «+», de manera que la sentencia se interpreta como «1 + (2*3)». Pues bien, podemos establecer una gramática equivalente a la anterior que refleja esta prioridad y que, por tanto, no es ambigua. Para ello, ampliamos E_A con dos nuevos símbolos: «$\langle TERM\rangle$» (término) y «$\langle FACT\rangle$» (factor), y definimos las nuevas producciones así:

$$\langle EA\rangle \rightarrow \langle EA\rangle + \langle TERM\rangle$$
$$\langle EA\rangle \rightarrow \langle TERM\rangle$$
$$\langle TERM\rangle \rightarrow \langle TERM\rangle * \langle FACT\rangle$$
$$\langle TERM\rangle \rightarrow \langle FACT\rangle$$

⟨FACT⟩ → ⟨CTE⟩
⟨FACT⟩ → (⟨EA⟩)
⟨CTE⟩ → ⟨CTE⟩⟨DIG⟩
⟨CTE⟩ → ⟨DIG⟩
⟨DIG⟩ → 0
.....
⟨DIG⟩ → 9

Ahora, la sentencia 1 + 2*3 sólo tiene un árbol, el de la figura 3.8. Y, en general, «*» tiene siempre precedencia sobre «+», excepto cuando se usan paréntesis para invalidar tal precedencia.

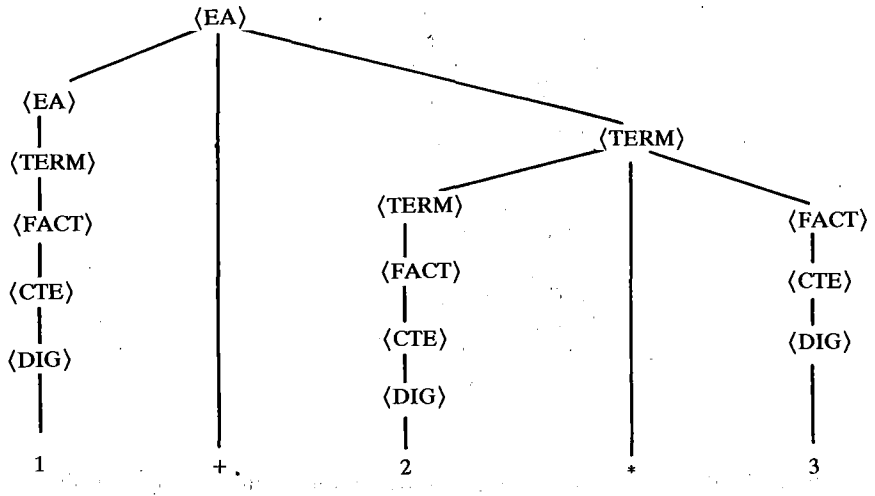


FIGURA 3.8.

El problema de la ambigüedad, en general, es indecidible:

Teorema 5.6. No existe un algoritmo que, aplicado a una *C*-gramática arbitraria, permita decidir si la gramática es ambigua o no.

6. RESUMEN

Los lenguajes de tipo 0 son recursivamente numerables pero no todos son recursivos. Los lenguajes de tipo 1 (y 2 y 3) son recursivos: existe un algoritmo para decidir si $x \in L$ o $x \notin L$. Sin embargo, hay lenguajes recursivos que no son de tipo 1. Es decir:

$$\{L(G3)\} \subset \{L(G2)\} \subset \{L(G1)\} \subset \{L_{\text{recursivos}}\} \subset \{L(G0)\} = \{L_{\text{recursivamente num}}\}$$

En los lenguajes libres de contexto las derivaciones se pueden representar gráficamente mediante árboles de derivación. A cada derivación corresponde un árbol y viceversa. Si alguna sentencia puede derivarse de dos o más formas diferentes se dice que la gramática es ambigua.

El problema de la ambigüedad es, en general, indecidible. No obstante, en muchos casos es posible, dada una gramática ambigua, encontrar otra equivalente a ella y no ambigua.

7. NOTAS HISTÓRICAS Y BIBLIOGRÁFICA

La mayor parte de los resultados concernientes a los lenguajes sensibles al contexto y, especialmente, a los libres de contexto, así como sus aplicaciones tanto a lenguajes naturales como a lenguajes de programación aparecieron durante los años 60. Las primeras propiedades indecidibles de las C-gramáticas se publicaron en Bar-Hillel *et al.* (1961), y la indecidibilidad de la ambigüedad se demostró por Cantor (1962), Floyd (1962) y Chomsky y Schutzenberger (1963), independientemente. Floyd (1964) publicó una revisión sobre la aplicación de las gramáticas formales a los lenguajes de programación.

La bibliografía, especialmente sobre C-gramáticas, es muy amplia; como botón de muestra podemos citar el libro de Ginsburg (1966), del cual se puede encontrar un resumen en Ginsburg (1968).

Hay gran cantidad de resultados sobre propiedades indecidibles de las C-gramáticas y sobre simplificación de las mismas o reducción a unas llamadas «formas normales», que pueden estudiarse en libros específicos, como el citado más arriba, o de carácter general, como los cinco recomendados en el capítulo anterior.

8. EJERCICIOS

8.1. Dada la gramática definida por

$$E_A = \{S\};\ E_T = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$P = \left\{ \begin{array}{r} S \rightarrow 12S \\ 12 \rightarrow 345 \\ 45 \rightarrow 678 \\ S \rightarrow \ \ 9 \end{array} \right\}$$

construir una gramática equivalente con reglas sensibles al contexto.

8.2. Dada la gramática definida por

$$E_A = \{S, A, B\};\ E_T = \{0, 1\};$$

$$P = \left\{ \begin{array}{ll} S \rightarrow 0A & B \rightarrow 1S0 \\ S \rightarrow 1B & B \rightarrow 0 \\ A \rightarrow 0S & \end{array} \right\}$$

determinar, de las siguientes sentencias, cuáles pertenecen y cuáles no al lenguaje: 00, 10; 100; 1100; 1010.

8.3. ¿De qué tipo es una gramática cuyas reglas son todas de la forma $A \to x\,B$ ó $A \to x$, con $A, B \in E_A$, $x \in E_T^*$?; demostrar que siempre puede encontrarse una gramática regular equivalente.

8.4. Con $E_A = \{S\}$; $E_T = \{a, b, c\}$, la gramática G_1 con reglas

$$P_1 = \left\{ \begin{array}{l} S \to aSa \\ S \to bSb \\ S \to aaSaa \\ S \to c \end{array} \right\}$$

es una gramática ambigua, mientras que G_2, con

$$P_2 = \left\{ \begin{array}{l} S \to aSa \\ S \to bSb \\ S \to c \end{array} \right\}$$

es equivalente a G_1 y no es ambigua. Comprobarlo haciendo algunas derivaciones. Describir informalmente la forma general de las sentencias del lenguaje.

8.5. Definir una gramática para las expresiones aritméticas como las del ejemplo 5.5, pero tal que «+» tenga preferencia sobre «*».

8.6. Definir una gramática para expresiones algebráicas que incluya tanto «constantes» como «identificadores» (ver el ejercicio 10.4 del capítulo anterior).

Capítulo 4
LENGUAJES Y AUTOMATAS

1. INTRODUCCIÓN

En este capítulo vamos a plantear el problema del reconocimiento de lenguajes desde un punto de vista material, ya que estudiaremos las relaciones entre el tipo de lenguaje y la estructura de la máquina capaz de reconocerlo; estas relaciones pueden enfocarse en dos sentidos:

a) dada una gramática, G, ¿qué estructura deberá tener una máquina, M, tal que $L(M) = L(G)$? (es decir, lenguaje reconocido por M igual a lenguaje generado por G);

b) dada una máquina, M. ¿Cuál será la gramática, G, tal que $L(G) = L(M)$?

De anteriores temas conocemos dos estructuras de máquina que pueden ser utilizadas como reconocedores de cadenas de símbolos: el autómata finito (tema «Autómatas») y la máquina de Turing (tema «Algoritmos»). Veremos aquí que la clase de lenguajes que pueden ser reconocidos por autómatas finitos es precisamente la clase de lenguajes que pueden ser generados por una gramática de tipo 3 (regular), y análogamente para las MT y las gramáticas de tipo 0. Para los dos tipos intermedios de gramáticas definiremos unos autómatas que pueden considerarse como restricciones de la máquina de Turing o como extensiones del autómata finito. Tendremos así una jerarquía de máquinas paralela a la jerarquía de lenguajes*.

Reflexione el lector en el hecho de que las «clases de lenguajes» (la clase de los lenguajes generados por una gramática regular, la clase de los lenguajes reconocidos por un autómata finito, etc.) son subconjuntos de $P(E^*)$ (conjunto de las partes de E^*).

---

* Obsérvese que un AF se puede considerar como un caso muy particular de MT que sólo puede leer de la cinta, ésta se desplaza en un solo sentido, y no para.

El desarrollo completo de las ideas expuestas exige la demostración de una serie de teoremas acompañada de las correspondientes construcciones (dada una gramática de tipo 0, diseñar la correspondiente MT y viceversa, etc.) que alargaría excesivamente este tema, por lo que nos contentaremos con desarrollar únicamente el caso más sencillo (gramáticas regulares y autómatas finitos), limitándonos a enunciar los teoremas en los otros tres.

2. LENGUAJES DE TIPO 0 Y MÁQUINAS DE TURING

2.1. Reconocedor de Turing

En el capítulo 5, apartado 2, del tema «Algoritmos», se define la parte de control de una MT como

$$T - Q = \langle E, (E \times M) \cup (\text{Stop}), Q, f, g \rangle$$

Al igual que en el tema «Autómatas» (capítulo 4) adaptábamos la definición general de autómata finito para especializarlo como reconocedor, adaptaremos también la definición de $T - Q$. Primero, explicaremos de una manera informal lo que entendemos por máquina de Turing reconocedora o reconocedor de Turing.

Un *reconocedor de Turing* será una MT que, inicializada en un estado inicial predeterminado, q_1, y teniendo la cadena $x \in E^*$ en su cinta (descripción instántanea inicial $0q_1x0$) opera de tal modo que después de un tiempo finito se para, tras escribir un 1, si $x \in L$. Si $x \notin L$ pueden ocurrir dos cosas: escribe 0 y se para (rechaza x), o no se para. (Suponemos que $\{0, 1\} \in E$).

$T - Q$ es un autómata finito que siempre se podrá describir como una máquina de Moore, con una función de salida:

$$h: Q \to (E \times M) \cup (\text{Stop})$$

Definiremos los estados finales de aceptación como aquellos para los que la función h es «stop», siendo la h del estado anterior «1». Tendremos así un conjunto $F_A \subset Q$ de estados de aceptación:

$$F_A = \{q_A \mid h(q_A) = \text{Stop y } h(q_{A'}) = 1, \text{ siendo } q_{A'} \text{ tal que } f(e, q_{A'}) = q_A \text{ para algún } e \notin E\}$$

El control de un reconocedor de Turing será definido como

$$R_{T-Q} = \langle E, Q, f, q_1, F_A \rangle,$$

y el reconocedor será $R_{\text{MT}} = R_{T-Q}$ + cinta.

2.2. Lenguaje aceptado por un reconocedor de Turing

Podemos ya definir el *lenguaje aceptado por un* R_{MT}:

$$L(R_{\mathrm{MT}}) = \{x \in E^* \mid \mathrm{Res}\ R_{\mathrm{MT}}\ (0q_1x0) = 0yq_Az0;\ q_A \in F_A;\ y, z \in E^*\}$$

y enunciar los dos teoremas que establecen la *equivalencia entre la clase de lenguajes aceptados por algún* R_{MT} *y la clase de lenguajes generados por una gramática de tipo 0.*

2.3. Teorema MT1

Para toda gramática de tipo 0, $G0$, existe un reconocedor de Turing, R_{MT}, tal que $L(R_{\mathrm{MT}}) = L(G0)$.

Obsérvese que, en general, no tiene por qué existir R'_{MT} tal que $L(R'_{\mathrm{MT}}) = E^* - L(G0)$; esto sólo ocurrirá en el caso de que $L(G0)$ sea recursivo, y en tal caso R'_{MT} puede ser el mismo R_{MT}, definiendo los estados de aceptación por $h(q_{A'}) = 0$.

2.4. Teorema MT2

Para todo reconocedor de Turing, R_{MT}, existe una gramática de tipo 0, $G0$, tal que $L(G0) = L(R_{\mathrm{MT}})$.

2.5. Conclusión

El teorema MT1 viene a decir que $\{L(G0)\} \subset \{L(R_{\mathrm{MT}})\}$, y el MT2, que $\{L(R_{\mathrm{MT}})\} \subset \{L(G0)\}$. Además, sabemos que la clase de lenguajes reconocibles por un R_{MT} coincide con la clase de los lenguajes recursivamente numerables (esto es una consecuencia de la hipótesis de Turing), y que los lenguajes recursivos son un subconjunto de éstos, luego, como conclusión, escribiremos:

$$\{L_{\mathrm{recurs.}}\} \subset \{L_{\mathrm{recurs.\ num.}}\} = \{L(R_{\mathrm{MT}})\} = \{L(G0)\}$$

3. LENGUAJES SENSIBLES AL CONTEXTO Y AUTÓMATAS LIMITADOS LINEALMENTE

3.1. Autómata limitado linealmente

Un *autómata limitado linealmente, ALL,* es una MT cuya cabeza no puede desplazarse fuera de los límites entre los que se sitúa inicialmente la cadena de entrada. Para señalar tales límites, se incluye en E un símbolo especial como marcador, # (figura 4.1)

Un reconocedor limitado linealmente, R_{ALL}, se definirá por estados finales de aceptación con un estado inicial q_1, igual que un R_{MT}.

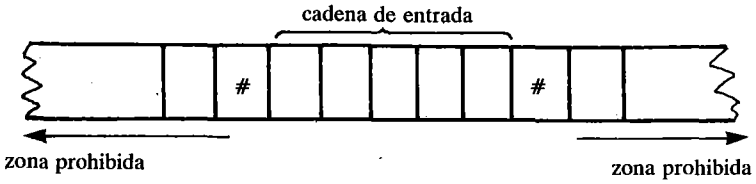


FIGURA 4.1.

3.2. Lenguaje aceptado por un reconocedor limitado linealmente

La definición del *lenguaje aceptado por un* R_{ALL} es análoga a la de $L(R_{MT})$, salvo que # no puede utilizarse como símbolo en las cadenas de entrada y que se preserva la distancia entre los dos marcadores:

$$L(R_{ALL}) = \{x \in (E - \#)^* \mid \text{Res } R_{ALL}(\#q_1x\ \#) = \#\ yq_Az\# \\ q_A \in F_A;\ y,\ z \in (E - \#)^*;\ \log(x) = \lg(yz)\}$$

3.3. Teorema ALL1

Para toda gramática sensible al contexto, $G1$, existe un reconocedor limitado linealmente, R_{ALL}, tal que $L(R_{ALL}) = L(G1)$.

Como las gramáticas de tipo 1 generan lenguajes recursivos, siempre existirá también R'_{ALL} tal que $L(R'_{ALL}) = E^* - L(G1)$.

3.4. Teorema ALL2

Para todo reconocedor limitado linealmente, R_{ALL}, existe una gramática sensible al contexto, $G1$, tal que $L(G1) = L(R_{ALL})$.

3.5. Conclusión

De los dos teoremas anteriores deducimos:

$$\{L(R_{ALL})\} = \{L(G1)\}$$

4. LENGUAJES LIBRES DE CONTEXTO Y AUTÓMATAS DE PILA

4.1. Autómata de pila

Una *pila* es un tipo de almacenamiento en el que sólo se pueden leer las informaciones en el orden inverso en que han sido escritas, siguiendo el principio de *«el último en entrar será el primero en salir»*, y de ahí que también se le llame memoria LIFO *(last-in, first-out)*.

Las dos operaciones posibles con una pila son introducir (escribir) o extraer (leer), como indica la figura 4.2.

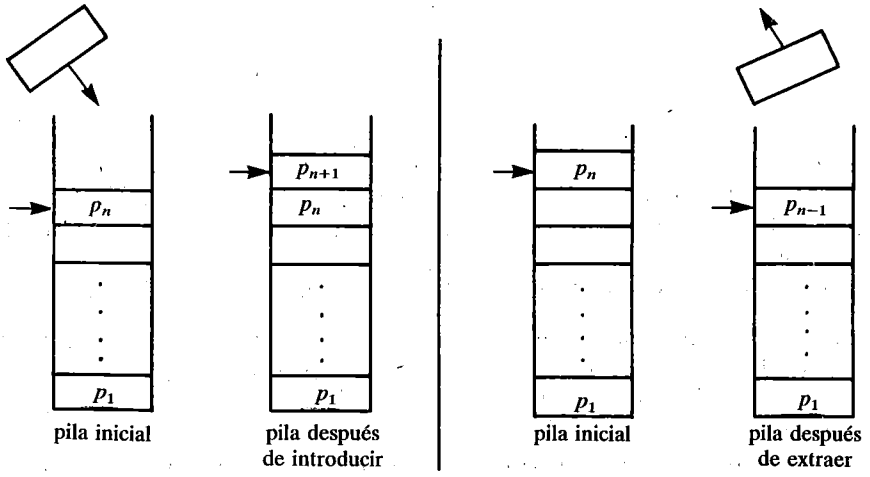


FIGURA 4.2.

El símbolo «⟷» en la figura indica en cada momento cuál es la «cabeza» o «cima» de la pila (*).

Un *autómata de pila*, AP, es un autómata finito al que se le añade una pila potencialmente infinita para guardar resultados intermedios. Puede considerársele como una MT con dos cintas y dos cabezas (figura 4.3). Pero no por ello es más potente que una MT. De hecho, su capacidad de procesamiento es inferior a la del ALL, debido a las siguientes restricciones sobre las posibles operaciones con las cintas:

* Si la pila se simula en la memoria central de un ordenador mediante un programa, existirá un puntero a la cima de la pila, es decir, una variable que se actualizará en cada operación para que su valor sea la dirección de la cima.

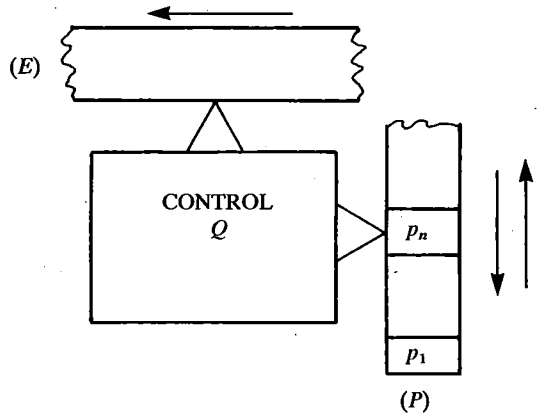


FIGURA 4.3.

La cinta (E) *se desplaza en un solo sentido,* y la correspondiente cabeza *sólo puede leer.* En la cinta (P) (*limitada en un extremo*) puede leerse, en cuyo caso desaparece p_n y se desplaza hacia arriba (según la figura) o escribirse, para lo cual se desplaza hacia abajo y se introduce p_{n+1} sobre p_n.

Las operaciones elementales que puede realizar un AP son de dos tipos:

— dependientes de E: se lee un $e_i \in E$, se desplaza la cinta (E), y, en función de e_i, q_j, p_k, el control pasa a otro estado q_e y en la pila (P) se introduce p_{k+1}, o se extrae p_k, o no se hace nada;
— independientes de E: lo mismo, sólo que e_i no interviene y (E) no se mueve, lo que permite manejar la pila sin tener en cuenta las informaciones de entrada.

En cualquier caso, si se vacía la pila (es decir, se extrae p_1) el AP se para.

4.2. Lenguaje aceptado por un reconocedor de pila

El reconocedor de pila, R_{AP}, será un AP inicializado en un estado q_1 y con p_1 como única información en la cinta (P). La aceptación de cadenas, y, consecuentemente, el *lenguaje aceptado por* R_{AP}, $L(R_{AP})$ se puede definir según dos criterios diferentes:

— *por pila vacía:* $x \in L(R_{AP})$ si al leer el último símbolo de x la pila queda vacía (y, por tanto, el AP se para);
— *por estados finales:* $x \in L(R_{AP})$ si al leer el último símbolo de x el control queda en un estado $q_A \in F_A$, siendo $F_A \subset Q$ un conjunto de estados definidos a priori como estados finales de aceptación.

En un estudio formalizado de los AP, que aquí no abordamos, se demuestra que *ambas definiciones de* $L(R_{AP})$ *son equivalentes,* en el sentido de que la clase de los lenguajes aceptados por las estructuras del tipo AP es la misma en ambos casos.

4.3. Teorema AP1

Para toda gramática libre de contexto, $G2$, existe un reconocedor de pila, R_{AP}, tal que $L(R_{AP}) = L(G2)$.

4.4. Teorema AP2

Para todo reconocedor de pila, R_{AP}, existe una gramática libre de contexto, $G2$, tal que $L(G_2) = L(R_{AP})$.

4.5. Conclusión

De los teoremas enunciados se desprende que

$$\{L(R_{AP})\} = \{L(G2)\}$$

5. LENGUAJES REGULARES Y AUTÓMATAS FINITOS

5.1. Autómata finito no determinista

En el tema «Autómatas» se han estudiado los reconocedores finitos y se ha visto que la clase de lenguajes que pueden reconocer son los lenguajes regulares. Aquí vamos a demostrar que esa clase de lenguajes coincide precisamente con la clase de lenguajes que pueden ser generados por gramáticas de tipo 3 ó regulares. Para ello tenemos que recurrir a un concepto auxiliar: el AF no determinista. Su necesidad aparece claramente teniendo en cuenta cómo se establecen las relaciones entre gramáticas de tipo 3 y autómatas. En efecto, si en un AF existe una transición del estado A al estado B bajo el efecto de la entrada e, veremos que la gramática correspondiente contiene la producción $A \rightarrow eB$, y a la inversa. Ahora bien, nada impide que una gramática regular contenga simultáneamente las reglas $A \rightarrow eB$ y $A \rightarrow eC$, lo que nos lleva a considerar la posibilidad de que *del estado A con entrada e se pase bien al estado B, bien al C.* Un AF en el que tal cosa puede ocurrir se llama *no determinista.* Quede bien claro que no es la idea de probabilidad la que aquí interviene. El AF no determinista debe ser considerado con un concepto abstracto, desprovisto de interpretación física. (El autómata estocástico estudiado en el tema «Autómatas», capítulo 5, apartado 3, podría verse como un autómata no determinista si atribuimos probabilidades de manera consistente a las producciones.)

Definición 5.1.1. Un *autómata finito no determinista* es una quíntupla

$$\text{AFND} = \langle E, S, Q, f, g \rangle,$$

donde todo es igual que en un autómata finito (tema «Autómatas», capítulo 2,

apartado 1.1), *salvo que* el rango de la función de transición no es Q, sino $P(Q)$ (siendo $P(Q)$ el conjunto de las partes de Q):

$$f\colon E \times Q \to P(Q)$$

Es decir, $f(e, q) = \{q_a, q_b, \ldots, q_e\} \subset Q$. (Obsérvese que puede ser $f(e, q) = \varnothing$.)

Un *reconocedor finito no determinista* se define siguiendo la misma línea que en el caso determinista: se considera un conjunto de estados finales, F, un estado inicial, q_1, y

$$\text{RFND} = \langle E, Q, f, q_1, F\rangle$$

El dominio de la función de transición puede extenderse a $E^* \times Q$: Recuérdese que en el caso determinista hacíamos (tema «Autómatas», capítulo 2, apartado 3.2) $f(\lambda, q) = q$; $f(x_1x_2, q) = f(x_2, f(x_1, q))$; ahora haremos

$$f(\lambda, q) = \{q\};\ f(x_1x_2, q) = \bigcup_{\substack{q_k \text{ en} \\ f(x_1, q)}} f(x_2, q_k)$$

También puede extenderse tal dominio a $E^* \times P(Q)$; para ello haremos:

$$f(x, \varnothing) = \varnothing$$

$$f(x, \{q_a, q_b, \ldots, q_e\}) = \bigcup_{j=a}^{e} f(x, q_j)$$

Ejemplo 5.1.2. Sea el RFND definido por

$$E = \{a, b\};\ Q = \{q_1, q_2, q_3, q_4\};\ F = \{q_4\}$$

y la función de transición de la figura 4.4

	a	b
q_1	$\{q_2, q_3\}$	$\varnothing$
q_2	$\varnothing$	$\{q_2, q_4\}$
q_3	$\{q_3\}$	$\{q_4\}$
(q_4)	$\{q_4\}$	$\varnothing$

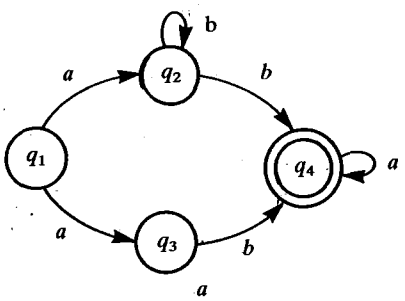


FIGURA 4.4.

Estudiemos, por ejemplo, las transiciones producidas por la cadena $x = abbab$

$$f(a, q_1) = \{q_2, q_3\}$$
$$f(ab, q_1) = f(b, q_2) \cup f(b, q_3) = \{q_2, q_4\}$$
$$f(abb, q_1) = f(b, q_2) \cup f(b, q_4) = \{q_2, q_4\}$$
$$f(abba, q_1) = f(a, q_2) \cup f(a, q_4) = \{q_4\}$$
$$f(abbab, q_1) = f(b, q_4) = \varnothing$$

Si quisiéramos calcular, por ejemplo, $f(a, \{q_1, q_4\})$, teniendo en cuenta cómo se ha definido la extensión a $E^* \times P(Q)$, tendríamos:

$$f(a, \{q_1, q_4\}) = f(a, q_1) \cup f(a, q_4) = \{q_2, q_3, q_4\}$$

5.2. Lenguaje aceptado por un reconocedor finito no determinista, y equivalencia con algún reconocedor finito determinista

Definiremos el *lenguaje aceptado por un RFND* como el conjunto de cadenas para las cuales la función de transición conduce a un subconjunto de Q dentro del cual se encuentra al menos un estado final:

$$L(\text{RFND}) = \{x \in E^* \mid [f(x, q_1) = \{q_a, q_b, \ldots, q_\ell\}] \bigwedge [\exists q_i (i = a, \ldots, \ell) \in F]\}$$

Así, en el ejemplo anterior vemos que la cadena a es rechazada, ab, abb y $abba$ son aceptadas, y $abbab$ es rechazada. Es fácil ver que el lenguaje admite una expresión regular: ab^*ba^* es la expresión regular del conjunto de cadenas que, pasando por el estado q_2, son aceptadas, mientras que aa^*ba^* es la de las que pasan por q_3; por tanto, la expresión regular de todas será: $ab^*ba^* + aa^*ba^* + aa^*ba^* = a(a^* + b^*)ba^*$.

Pasaremos ahora a ver que la clase de los lenguajes aceptados por reconocedores finitos no deterministas coincide con la de los acpetados por reconocedores finitos deterministas. Para ello será preciso demostrar dos cosas:

— que, dado un *RF* (determinista), siempre existe un RFND tal que $L(\text{RFND}) = L(\text{RF})$, (y, por tanto, $\{L(\text{RF})\} \subset \{L(\text{RFND})\}$), y

— que, dado un RFND, siempre existe un RF tal que $L(\text{RF}) = L(\text{RFND})$ (y, por tanto, $\{L(\text{RFND})\} \subset \{L(\text{RF})\}$, que, con lo anterior, demostrará que $\{L(\text{RFND})\} = \{L(\text{RF})\}$).

Lo primero es evidente, ya que un RF es un caso particular de RFND en el que los elementos del rango de f son subconjuntos unitarios de $P(Q)$. Lo segundo no es tan evidente, por lo que pasaremos a enunciarlo en forma de teorema y demostrarlo.

Teorema 5.2.1. Para todo reconocedor finito no determinista, RFND $= \langle E, Q, f, q_1, F\rangle$, puede construirse un reconocedor finito determinista, RF $= \langle E, Q', f_1', q_1', F'\rangle$, tal que $L(\text{RF}) = L(\text{RFND})$.

Hagamos: $Q' = P(Q)$ (de modo que RF tendrá, en general, $\text{card}(Q') = 2^{\text{card}(Q)}$ estados). Al estado de Q' que corresponda a $\{q_a, q_b, \ldots, q_e\}$ lo denotaremos $[q_a, q_b, \ldots, q_e]$.

$$f'(e, [q_a, \ldots, q_e]) = [q_m, \ldots, q_k] \text{ si y sólo si}$$
$$f(e, \{q_a, \ldots, q_e\}) = \{q_m, \ldots, q_k\}.$$

(Es decir, se calcula $f'(e, q')$ aplicando f a cada estado q de los que figuran en q' y haciendo la unión de todos los resultantes.)

$$q'_1 = [q_1]$$
$$F' = \{q' \in Q' \mid q_f \in q' \text{ y } q_f \in F\}$$

(Es decir, para que q' sea estado final basta que uno o más de los estados de Q que lo componen sea final.)

Para demostrar que ambos autómatas aceptan el mismo lenguaje bastará comprobar que, para todo $x \in E^*$, $f'(x, q_1) \in F'$ si y sólo si $f(x, q_1)$ contiene un estado (o varios) $q_f \in F$, y, teniendo en cuenta la definición de F', esto será evidentemente cierto si demostramos que

$$f'(x, q'_1) = [q_a, \ldots, q_e] \text{ si y sólo si } f(x, q_1) = \{q_a, \ldots, q_e\}$$

Tal demostración puede hacerse por inducción sobre la longitud de x: Para $\lg(x) = 0$ ($x = \lambda$) es inmediato, puesto que $f'(\lambda, q'_1) = q'_1 = [q_1]$, y $f(\lambda, q_1) = \{q_1\}$. Supongamos que es cierto para $\lg(x) \leq 1$; entonces para $e \in E$,

$$f'(xe, q'_1) = f'(x, q'_1)$$

Pero por la hipótesis de la inducción,

$$f'(x, q'_1) = [q_a, \ldots, q_e] \text{ si y sólo si } f(x, q_1) = \{q_a, \ldots, q_e\},$$

y, por definición de f',

$$f'(e, [q_a, \ldots, q_e]) = [q_m, \ldots, q_k] \text{ si y sólo si}$$
$$f(e, \{q_a, \ldots, q_e\}) = \{q_m, \ldots, q_k\}.$$

Por tanto,

$$f'(xe, q'_1) = [q_m, \ldots, q_k] \text{ si y sólo si } f(xe, q_1) = \{q_m, \ldots, q_k\},$$

con lo que queda demostrado.

Ejemplo 5.2.2.

Tomemos el RFND del ejemplo 5.1.2. Siguiendo la construcción del Teorema 5.2.1, el RF tendrá, en principio, $2^4 = 16$ estados:

$$Q' = \{\varnothing, [q_1], [q_2], [q_3], [q_4], [q_1, q_2] \ldots [q_1, q_2, q_3, q_4]\}$$

$$q_1' = [q_1]$$

$$F' = \{[q_4], [q_1, q_4], [q_2, q_4], [q_3, q_4], [q_1, q_2, q_4], \ldots [q_1, q_2, q_3, q_4]\}$$

y f' vendrá dada por la siguiente tabla:

	a	b
$\varnothing$	$\varnothing$	$\varnothing$
$[q_1]$	$[q_2, q_3]$	$\varnothing$
$\rightarrow [q_2]$	$\varnothing$	$[q_2, q_4]$
$[q_3]$	$[q_3]$	$[q_4]$
$[q_4]$	$[q_4]$	$\varnothing$
$\rightarrow [q_1, q_2]$	$[q_2, q_3]$	$[q_2, q_4]$
$\rightarrow [q_1, q_3]$	$[q_2, q_3]$	$[q_4]$
$\rightarrow [q_1, q_4]$	$[q_2, q_3, q_4]$	$\varnothing$
$[q_2, q_3]$	$[q_3]$	$[q_2, q_4]$
$[q_2, q_4]$	$[q_4]$	$[q_2, q_4]$
$\rightarrow [q_3, q_4]$	$[q_3, q_4]$	$[q_4]$
$\rightarrow [q_1, q_2, q_3]$	$[q_2, q_3]$	$[q_2, q_4]$
$\rightarrow [q_1, q_2, q_4]$	$[q_2, q_3, q_4]$	$[q_2, q_4]$
$\rightarrow [q_1, q_3, q_4]$	$[q_2, q_3, q_4]$	$[q_4]$
$\rightarrow [q_2, q_3, q_4]$	$[q_3, q_4]$	$[q_2, q_4]$
$\rightarrow [q_1, q_2, q_3, q_4]$	$[q_2, q_3, q_4]$	$[q_2, q_4]$

Ahora bien, en un RF los estados que no son accesibles desde el inicial pueden eliminarse. Así, eliminamos $[q_2]$ porque no lo vemos aparecer dentro de la tabla, y lo mismo $[q_1, q_2]$, etc. (los señalados con una flecha). Obsérvese que $[q_2, q_3, q_4]$ sólo es accesible desde otros que previamente han sido eliminados por lo que también puede eliminarse, y lo mismo ocurre con $[q_3, q_4]$. Naturalmente, el único del que no puede prescindirse en ningún caso es $[q_1]$. $\varnothing$, en este caso, no puede eliminarse, ya que es accesible desde $[q_1]$ (y desde $[q_4]$). Finalmente, de los 16 estados nos hemos quedado con 6, y el resultado puede representarse en forma de diagrama de Moore según indica la figura 4.5.

Comprobemos que el lenguaje aceptado es el mismo del RFND de partida (expresión regular $a(a^* + b^*)ba^*$). Al estado final $[q_2, q_4]$ sólo podemos ir pasando por $[q_2, q_3]$; tenemos así que las cadenas aceptadas en $[q_2, q_4]$ tienen por expresión

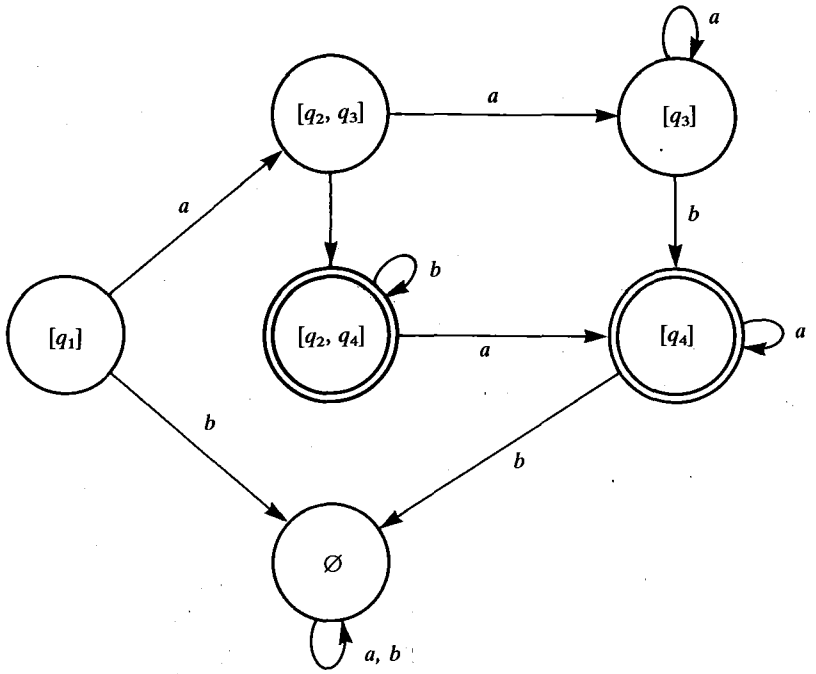


FIGURA 4.5.

regular: abb^*; a $[q_4]$ puede llegarse por $[q_2, q_4]$ (abb^*aa^*) o por $[q_3]$ (aaa^*ba^*). Tenemos, pues, en total:

$$\begin{aligned} abb^* + abb^*aa^* + aaa^*ba^* &= abb^*(\lambda + aa^*) + aaa^*ba^* = \\ &= abb^*a^* + aaa^*ba^* = ab^*ba^* + aaa^*ba^* = \\ &= a(b^* + aa^*)ba^* = a(b^* + a^*)ba^* \end{aligned}$$

(Esta última igualdad se ha obtenido teniendo en cuenta que $b^* + aa^* = b^* + \lambda + aa^* = b^* + a^*$). Naturalmente, llegaríamos al mismo resultado aplicando el algoritmo general de análisis desarrollado en el tema «Autómatas», capítulo 4, apartado 4.

5.3. Teorema AF1

Para toda gramática regular, $G3$, existe un reconocedor finito, RF, tal que $L(\text{RF}) = L(G3)$.

Para demostrar este teorema construiremos un RFND que reconoce exactamente el lenguaje aceptado por una $G3$ dada, al que le corresponderá un RF de acuerdo con el teorema 5.2.1.

Sea $G3 = \langle E_A, E_T, P, S\rangle$ una gramática regular. Definimos RFND = $\langle E, Q, f, q_1, F\rangle$ del siguiente modo:

$$E = E_T = \{a_1, a_2, \ldots, a_n\}$$
$$Q = E_A \cup \{X\} = \{A_1, A_2, \ldots, A_n, X\}$$
$$q_1 = S$$
$$F = \{X\}$$
Si $(A_i \rightarrow a_j) \in P$, entonces $X \in f(a_j, A_i)$
Si $(A_i \rightarrow a_j A_k) \in P$, entonces $A_k \in f(a_j, A_i)$
$(\forall a_i \in E_T)(f(a_i, X) = \varnothing)$

Veamos ahora que $L(\text{RFND}) = L(G3)$

a) Sea $x = a_1 a_2, \ldots, a_k \in L(G3)$; entonces, deberán existir $A_1, A_2, \ldots, A_{k-1} \in E_A$ tales que

$$S \Rightarrow a_1A_1 \Rightarrow a_1a_2A_2 \Rightarrow \ldots \Rightarrow a_1a_2 \ldots a_{k-1}A_{k-1} \Rightarrow a_1a_2 \ldots a_{k-1}a_k$$

y para ello P debe contener las reglas

$$\begin{aligned} S &\rightarrow a_1A_1 \\ A_1 &\rightarrow a_2A_2 \\ &\vdots \quad\quad \vdots \\ A_{k-1} &\rightarrow a_k \end{aligned}$$

Por la definición de f, $A_1 \in f(a_1, q_1)$; $A_2 \in f(a_2, A_1)$; ... $X \in f(a_k, A_{k-1})$; por consiguiente, $X \in f(a_1a_2, \ldots, a_k, q_1)$, y, como X es el estado final, $a_1a_2, \ldots, a_k \in L(\text{RFND})$. Esto nos dice que $L(G3) \subset L(\text{RFND})$.

b) A la inversa, si $x = a_1a_2, \ldots, a_k \in L(\text{RFND})$, deberá existir una secuencia de estados $A_1, A_2, \ldots, A_{k-1}, X$, tal que

$$\begin{aligned} &A_1 \in f(a_1, q_1) \\ &A_2 \in f(a_2, A_1) \\ &\vdots \quad\quad \vdots \\ &A_{k-1} \in f(a_{k-1}, A_{k-2}) \\ &X \in f(a_k, A_{k-1}) \end{aligned}$$

y, conforme a la construcción del RFND, $G3$ deberá tener las reglas: $S \rightarrow a_1A_1$; $A_1 \rightarrow a_2A_2$; ..., $A_{k-1} \rightarrow a_k$, con las cuales se podrá hacer la derivación $S \Rightarrow a_1A_1 \Rightarrow a_1a_2A_2 \Rightarrow \ldots \Rightarrow a_1a_2, \ldots, a_k$, es decir, $a_1a_2, \ldots, a_k \in L(G3)$. Es decir, $L(\text{RFND}) \subset L(G3)$.

En resumen, $L(\text{RFND}) = L(G3)$.

Ejemplo 5.3.1. Tomemos la gramática regular del capítulo 2, apartado 4.5.

$$E_A = \{A, S\};\ E_T = \{a, b\};$$

$$P = \left\{ \begin{array}{l} S \rightarrow aS \\ S \rightarrow aA \\ A \rightarrow bA \\ A \rightarrow b \end{array} \right\}.$$

El correspondiente RFND

$$E = E_T = \{a, b\};\ Q = \{A, S, X\};\ F = \{X\};\ q_1 = S$$

y la función de transición de la figura 4.6.

	a	b
S	$\{S, A\}$	$\varnothing$
A	$\varnothing$	$\{A, X\}$
X	$\varnothing$	$\varnothing$

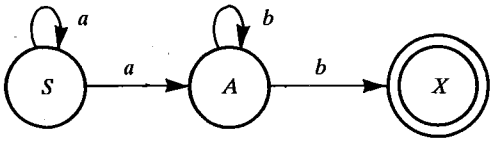


FIGURA 4.6.

Podemos construir un RF que acepte el mismo lenguaje, de acuerdo con el teorema 5.2.1:

	a	b
$\varnothing$	$\varnothing$	$\varnothing$
$[S]$	$[S, A]$	$\varnothing$
$\rightarrow [A]$	$\varnothing$	$[A, X]$
$\rightarrow [X]$	$\varnothing$	$\varnothing$
$[S, A]$	$[S, A]$	$[A, X]$
$\rightarrow [S, X]$	$[S, A]$	$\varnothing$
$[A, X]$	$\varnothing$	$[A, X]$
$\rightarrow [S, A, X]$	$[S, A]$	$[A, X]$

Eliminamos los estados $[A]$, $[X]$, $[S, X]$, $[S, A, X]$, inaccesibles desde $[S]$, y resulta el diagrama de Moore de la figura 4.7.

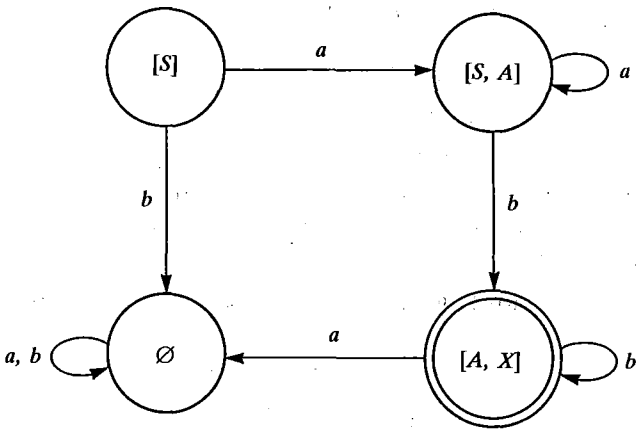


FIGURA 4.7.

Es fácil comprobar que el lenguaje viene dado por la expresión regular aa^*bb^*.

5.4. Teorema AF2

Para todo reconocedor finito, RF, existe una gramática regular, $G3$, tal que $L(G3) = L(\text{RF})$.

Dado RF = $\langle E, Q, f, q_1, F\rangle$, construiremos $G3 = \langle E_A\ E_T, P, S\rangle$ así:

$$E_A = Q;\ E_T = E;\ S = q_1$$

Si $f(a, q_i) = q_j$, entonces $q_i \rightarrow aq_j$
Si $f(a, q_i) = q_j$ y $q_j \in F$, entonces $q_i \rightarrow a$ (además de $q_i \rightarrow aq_j$)
Si $(\forall a)\ [(f(a, q_i) = q_i) \wedge (q_i \notin F)]$, entonces puede eliminarse q_i y todas las reglas en las que figure.

La demostración de que $L(G3) = L(\text{RF})$ se hace viendo que $(q_1 \overset{*}{\underset{G_3}{\Rightarrow}} x)$ si y sólo si $(\forall x \in E^*)\ [f(x, q_1) \in F]$, de manera similar a la del teorema anterior.

Ejemplo 5.4.1. Consideremos el RF al que llegábamos en el ejemplo 5.3.1 y retrocedamos ahora en busca de la gramática. Haciendo, para simplificar la escritura, $[S] = S$, $[S, A] = A$, $[A, X] = B$, tendremos:

$$E_A = \{S, A, B\};\ E_T = \{a, b\}$$

(Obsérvese que no hemos incluido $\varnothing$ en E_A porque $f(a, \varnothing) = f(b, \varnothing) = \varnothing$). Siguiendo las normas para la construcción de las reglas obtenemos:

$$\begin{aligned} S &\rightarrow aA \\ A &\rightarrow aA \\ A &\rightarrow bB \\ A &\rightarrow b \\ B &\rightarrow bB \\ B &\rightarrow b \end{aligned}$$

Esta gramática tiene un símbolo auxiliar (B) y dos reglas más que aquélla de la que habíamos partido, aunque, naturalmente, ambas deben ser equivalentes. La diferencia se ha originado en el paso a través del RF determinista. El lector puede extender la construcción del teorema AF2 al caso más general de RFND, y aplicarla al ejemplo 5.3.1 para ver que entonces sí se llega a la misma gramática original.

5.5. Conclusión

Según el teorema AF1, $\{L(G3)\} \subset \{L(\text{RF})\}$, y según el AF2, $\{L(\text{RF})\} \subset \{L(G3)\}$. Además, del tema «Autómatas» (capítulo 4, apartado 4.4.1; teorema de análisis) sabemos que $\{L(\text{RF})\} = \{L_{\text{regulares}}\}$. En resumen:

$$\{L_{\text{regulares}}\} = \{L(\text{RF})\} = \{L(G3)\}$$

5.6. Observaciones sobre la cadena vacía

En el teorema AF1 hemos supuesto implícitamente que $\lambda \notin L(G3)$. Si $\lambda \in L(G3)$, entonces, según se vio en el capítulo anterior, existirá la regla $S \rightarrow \lambda$ (y S no aparecerá a la derecha de ninguna otra regla). En este caso, la única diferencia es que habrá que considerar S también como estado final: $F = \{S, X\}$.

En cuanto al caso inverso (teorema AF2), si en el RF de partida $q_1 \in F$, entonces $\lambda \in L(\text{RF})$, y $L(G3) = L(\text{RF}) - \{\lambda\}$. Según vimos, puede construirse $G'3$ tal que

$$L(G'3) = L(G3) \cup \{\lambda\} = L(\text{RF})$$

6. JERARQUÍA DE AUTÓMATAS

Paralelamente a la jerarquía de lenguajes, aparece una jerarquía de autómatas, desde la MT hasta el AF. Para compararlos puede utilizarse como criterio la capacidad de memoria de cada uno.

La MT dispone, sobre la cinta, de una capacidad ilimitada de casillas para memorizar una cantidad ilimitada de símbolos.

La cinta de un ALL es también ilimitada, pero para una determinada cadena de entrada, x, de longitud $\ell = \lg(x)$, sólo puede utilizar ℓ casillas. Además, habrá que añadir a esta capacidad variable de memoria (dependiente, en cada caso, de cada entrada), una capacidad fija, n, que corresponde a la capacidad de la unidad central para memorizar estados. En resumen, la memoria de un ALL es una función de la longitud ℓ de la cadena de entrada: $\ell + n$. La forma lineal de esta función justifica el nombre del autómata.

El AP sólo puede utilizar la cinta de entrada para leer, pero dispone de una cinta de trabajo o pila potencialmente ilimitada que le sirve de memoria. Aquí también puede estimarse, para una longitud ℓ de la cadena de entrada, un límite superior de la memoria a utilizar. En efecto, si la cadena más larga que puede escribirse en la pila como consecuencia de la lectura de un símbolo de entrada tiene longitud k, la capacidad total de memoria será la suma de la parte variable (pila): $k\ell$ y de la parte fija, n; en total, $k\ell + n$. Esta es también una función lineal de ℓ, por lo que del AP puede también decirse que es un autómata limitado linealmente, pero con dos restricciones importantes: la cinta de entrada sólo se desplaza en un sentido, y la memoria tiene estructura de pila (si se quiere recuperar un símbolo que no está en la cima, hay que borrar toda la información comprendida entre la cima y el símbolo, cosa que no ocurre con la MT ni el ALL).

La capacidad de memoria de un AF es fija e independiente de la cadena de entrada.

Finalmente, es preciso que mencionemos un punto importante que, al omitir la demostración de la mayoría de los teoremas, hemos pasado por alto: que los autómatas de que venimos hablando son, en general, no deterministas, es decir, que la función de transición de la unidad de control tiene la forma que veíamos en el AF no determinista. Esto solamente serviría para demostrar los teoremas, y no tendría mayor importancia, si en todos los tipos de reconocedores ocurriese lo que en el RF:

que para todo RFND puede encontrarse un RF que acepte el mismo lenguaje, pero no es así:

a) En la MT, se demuestra que $\{L(R_{MT})\} = \{L(R_{MT}ND)\}$, es decir, ocurre lo mismo que en el RF.
b) En el ALL *no se sabe* si $\{L(R_{ALL})\} = \{L(R_{ALL}ND)\}$ o $\{L(R_{ALL})\} \subset \{L(R_{ALL}ND)\}$. Por tanto, cuando se habla de equivalencia de lenguajes sensibles al contexto y lenguajes aceptados por R_{ALL} debe entenderse que éstos son *no deterministas*. Por supuesto, para todo $R_{ALL}ND$ puede diseñarse un R_{MT} que acepte el mismo lenguaje; lo único que ocurre es que la longitud de cinta necesaria en el R_{MT} (determinista) puede ser una función exponencial de la longitud de la cadena de entrada, en lugar de lineal, como en el $R_{ALL}ND$.
c) En el AP, *se sabe* que $\{L(R_{AP})\} \subset \{L(R_{AP}ND)\}$, y, es más, se conoce el tipo de gramáticas correspondientes a los R_{AP} deterministas, que son una particularización del tipo general de gramáticas libres de contexto, llamadas *gramáticas deterministas*, y que, entre otras propiedades, no son ambiguas.

7. RESUMEN

La figura 4.8 resume las principales conclusiones de este capítulo.

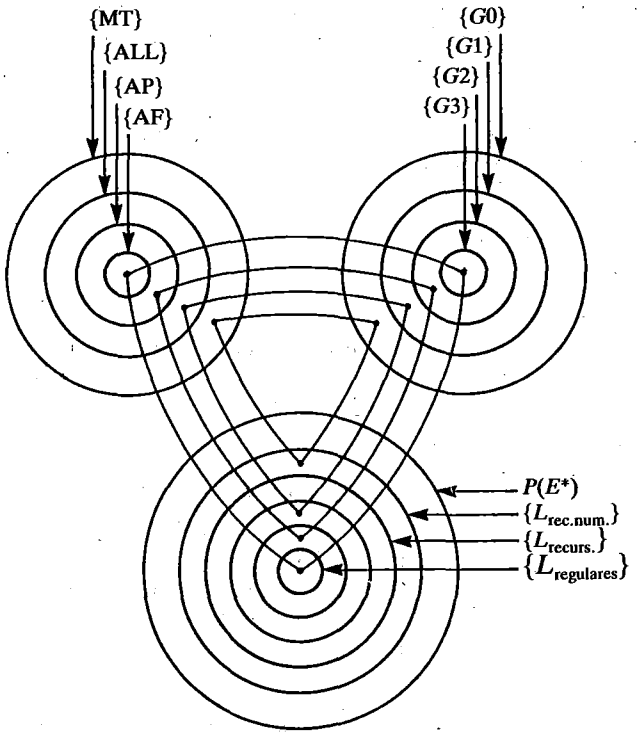


FIGURA 4.8

8. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

En los temas «Autómatas» y «Algoritmos» se han dado referencias históricas sobre los AF y las MT, respectivamente. El concepto de AF no determinista se debe a Rabin y Scott (1959), aunque las relaciones entre AF y gramáticas regulares ya habían sido establecidas por Chomsky y Miller (1958). La equivalencia entre lenguajes aceptados por MT y lenguajes de tipo 0 fue demostrada por Chomsky (1959).

El nombre y el concepto de ALL fueron introducidos por Myhill (1960). La generalización al caso no determinista y la equivalencia con las gramáticas sensibles al contexto se deben a Kuroda (1964).

Oettinger (1961) fue el primero en definir formalmente el AP, y Chomsky (1962) y Evey (1963), independientemente, demostraron su relación con los lenguajes libres de contexto.

Como bibliografía de consulta, son recomendables los mismos libros citados en los anteriores capítulos, y, especialmente, los de Hopcroft y Ullman (1969, 1979).

9. EJERCICIOS

9.1. Hallar una gramática regular que genere el lenguaje correspondiente al RF del ejemplo 5.2.2. Partir del RFND del ejemplo 5.1.2 y obtener otra gramática, equivalente pero más sencilla.

9.2. Hallar gramáticas regulares correspondientes a los cuatro ejemplos que se utilizaron en el capítulo 4 del tema «Autómatas».

9.3. Dada la expresión regular

$$\alpha = 00^* + 10^*10^*,$$

obtener un reconocedor finito del correspondiente lenguaje, y una gramática regular que lo genere.

9.4. Dada la gramática regular definida por:

$$E_A = \{S, A, B, C\};\ E_T = \{0, 1\};$$

$$P = \left\{ \begin{array}{ll} S \to 1A; & A \to 1 \\ S \to 1B; & B \to 1A \\ A \to 0A; & B \to 1C \\ A \to 0C; & B \to 1 \\ A \to 1C; & C \to 0 \end{array} \right\}$$

dar una expresión regular del lenguaje generado por ella y diseñar un circuito secuencial con biestables *JK* que sirva como reconocedor de tal lenguaje.

9.5. Repetir el ejercicio anterior para la gramática:

$$E_A = \{S, A\}; \quad E_T = \{a, b, c, d\};$$

$$P = \left\{ \begin{array}{ll} S \to aS; & S \to b \\ S \to aA; & S \to c \\ S \to bA; & S \to d \\ S \to cA; & A \to aA \\ S \to dA; & A \to bA \\ S \to a; & A \to a \\ & A \to b \end{array} \right\}$$

9.6. (Hopcroft y Ullman, 1969). Utilizar el concepto de RFND para demostrar que, si L es un lenguaje regular,

$$L^R = \{x \mid x^R \in L\}$$

es también regular

(x^R significa «reflejada» de x, es decir, si
$x = a_1a_2, \ldots, a_{n-1}a_n$, $x^R = a_na_{n-1}, \ldots, a_2a_1$).

9.7. (Hopcroft y Ullman, 1969). Apoyándose en el ejercicio anterior, demostrar que los lenguajes generados por gramáticas cuyas reglas son de la forma $A \to Ba$, ó $A \to a$ ($a \in E_T$; A, B, $\in E_A$) son lenguajes regulares (y viceversa: todo lenguaje regular puede generarse por una gramática de ese tipo).

Capítulo 5
APLICACIONES A LOS LENGUAJES DE PROGRAMACION

1. LENGUAJES E INTERPRETADORES

En los tres capítulos anteriores hemos presentado los fundamentos de la teoría de lenguajes formales y su relación con la teoría de autómatas. Veremos ahora cómo estos conceptos teóricos se aplican en la práctica a la problemática de la programación de ordenadores. El asunto es demasiado complejo y extenso como para pretender darle aquí un tratamiento completo. Por otra parte, es bastante probable que el lector tenga algunos (o muchos) conocimientos previos sobre programación de ordenadores. Por todo ello, nos limitaremos a apuntar las principales ideas, remitiendo a las orientaciones bibliográficas del apartado 7 a quien desee profundizar en alguno de los aspectos comentados.

Si L es un lenguaje de programación, $x \in L$ será un programa escrito en ese lenguaje, programa que corresponderá a algún algoritmo (tema «Algoritmos», capítulo 3, apartado 2) y que se habrá escrito para resolver algún problema. Entonces, lo que interesa es disponer de una máquina que *ejecute* x. Llamaremos *interpretador* de L* a una máquina, M, que reconoce si $x \in L$, y, en caso afirmativo, lo ejecuta. Y diremos que L es el *lenguaje de máquina* de M.

---

* En la terminología informática se utiliza más el término «intérprete». Preferimos, no obstante, «interpretador» por dos motivos. Uno es que, según el diccionario de Academia de la Lengua, «interpretador» es un adjetivo (que aquí sustantivamos) cuyo significado es «que interpreta», mientras que un «intérprete» es «una persona que interpreta». El otro es que resulta más uniforme con otros términos que designan conceptos relacionados, como «reconocedor», «procesador», etc. Por otra parte, hemos de advertir que vamos a utilizar la palabra «interpretador» con dos sentidos. Uno es el que acabamos de definir: una máquina que *reconoce* las órdenes o instrucciones de un programa escrito en su lenguaje de máquina *y las ejecuta*, una tras otra. El otro sentido, que veremos enseguida, y que es el más utilizado en la práctica, se refiere a un programa que va examinando y reconociendo una tras otra las instrucciones del programa original y generando las instrucciones oportunas para la máquina ejecutora.

Ahora bien, si afrontamos la realidad, nos encontramos con que los lenguajes de máquina de los ordenadores resultan prácticamente inutilizables para su uso directo. Codificar los algoritmos en un lenguaje cuyos únicos símbolos son «0» y «1» sería algo excesivamente tedioso para el programador, y de un coste prohibitivo por la cantidad de trabajo necesario, no sólo para la escritura de los programas, sino también para su depuración y mantenimiento. Por ello, desde los tiempos de los primeros ordenadores, uno de los campos de mayor actividad en informática ha sido el relacionado con el diseño de lenguajes que hagan más fácil y más eficaz la tarea de programar. Escrito un programa en uno de estos lenguajes, $L1$, antes de que pueda ejecutarse en una máquina, $M2$, cuyo lenguaje de máquina es $L2$, es preciso traducirlo. Un *traductor* de $L1$ a $L2$, $T12$, es un programa que recibe como dato de entrada $x \in L1$ y, tras comprobar que, efectivamente, $x \in L1$, produce como resultado $y \in L2$. Naturalmente, si este traductor ha de ejecutarse en $M2$ deberá estar escrito en $L2$ (o bien, haber sido traducido previamente a $L2$ desde el lenguaje en que fue escrito). El conjunto formado por $T12$ y $M2$ puede entonces verse como una «máquina virtual» (ver capítulo 1, apartado 1), $M1$, cuyo lenguaje de máquina es $L1$. Podemos, por tanto, decir que $M1$ es un interpretador (reconocedor y ejecutor) de $L1$.

De acuerdo con la descripción que acabamos de hacer de la máquina virtual $M1$, la interpretación de un programa $x \in L1$ se lleva a cabo en dos pasos: en el primero, la máquina real, $M2$, interpreta el programa traductor $T12$ dando como resultado de la ejecución $y \in L2$; después, la misma $M2$ interpreta y. Generalmente, a $L1$ se le llama *lenguaje fuente* y a $L2$ *lenguaje objeto.*

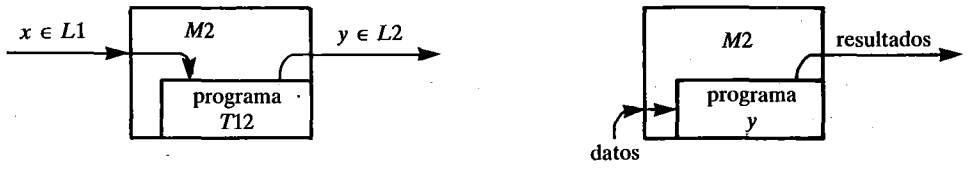
Pero hay otra alternativa para la ejecución en $M2$ de un algoritmo descrito por un programa $x \in L1$. Como sabemos (tema «Algoritmos», capítulo 3, apartado 3.1), x consta de una secuencia finita de instrucciones, instrucciones que tienen sentido para $M1$ (máquina virtual) pero no para $M2$ (máquina real). Podemos pensar en un programa para $M2$, $I12$, que vaya reconociendo, *una a una,* las instrucciones de x y haciendo que $M2$ ejecute las acciones oportunas para que dé el mismo resultado que hubiera dado $M1$. Este tipo de programa se llama *interpretador* (o «intérprete»)*.

La figura 5.1 ilustra la descripción que acabamos de hacer sobre el funcionamiento de un traductor y el de un interpretador. Ambos son *procesadores* de lenguaje, y, desde el punto de vista del programador, ambas permiten disponer de una máquina virtual $M1$, con lenguaje de máquina $L1$ (en el que el programador escribe sus programas) a partir de una máquina real $M2$, con lenguaje de máquina $L2$ (en el que sería mucho más difícil, o prácticamente absurdo, escribir los programas).

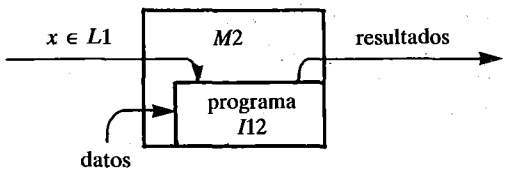
Desde un punto de vista pragmático, hay diferencias importantes entre procesar un programa escrito en lenguaje fuente mediante un traductor o mediante un interpretador. Como el traductor procesa todo el programa fuente, $x \in L1$, antes de que $M2$ ejecute el programa objeto, $y \in L2$, cualquier error que se observe durante la ejecución de y obliga a modificar x y volver a traducirlo (mediante una nueva ejecución de $T12$). Por el contrario, si se usa un interpretador, las modificaciones pueden hacerse «sobre la marcha» y de modo interactivo entre el programador y el

* Este es el segundo sentido de «interpretador» a que aludíamos en la nota anterior. Obsérvese que, funcionalmente, la diferencia entre un programa traductor y un programa interpretador es la misma que hay entre las profesiones de traductor e intérprete.

ordenador. El principal inconveniente del interpretador es que, para el mismo programa fuente y el mismo ordenador, la ejecución del programa es más lenta. Es fácil comprender por qué es así: si hay varias instrucciones dentro de un bucle, que van a ejecutarse muchas veces, el interpretador tiene que realizar con ellas las mismas operaciones de reconocimiento y de creación de instrucciones para *M2* una y otra vez (cada vez que se las encuentra en el bucle), mientras que el traductor las habría pasado de una vez por todas al lenguaje objeto.



a) Traducción



b) Interpretación

FIGURA 5.1.

2. LENGUAJES DE PROGRAMACION

2.1. Niveles de lenguajes

Salvo para arquitecturas muy especiales, que son actualmente objeto de investigación, todos los lenguajes de máquina son muy parecidos, si abstraemos los detalles concretos de cada ordenador. En este sentido, podemos hablar de un «lenguaje de máquina» en sentido genérico. Pues bien, imaginemos una escala arbitraria que abarcara desde el lenguaje de máquina hasta el lenguaje natural sobre la cual pudiésemos situar cualquier lenguaje: cuanto más cercano esté del lenguaje natural más fácil será escribir programas (mejorando con ello la productividad y la calidad del software), pero, al mismo tiempo, más complejo el procesador (traductor o interpretador) y, normalmente, menos eficaz el programa objeto. Esto último se debe a que cuanto más alejado está el lenguaje fuente del lenguaje objeto tanto más difícil es conseguir que el programa traducido esté optimizado (o sea, que se ejecute utilizando el mínimo de recursos de la máquina y en el menor tiempo posible). El diseño de un lenguaje de programación implica, por tanto, un compromiso entre facilidad de uso para el programador y posibilidad de realizar un procesador eficiente.

La «escala arbitraria» que hemos sugerido sólo es una ilustración, porque no existe una métrica para situar a los lenguajes en ella. Para empezar, ¿qué es «cercanía al lenguaje natural»? Depende de quién vaya a hacer uso del lenguaje: un lenguaje como FORTRAN, por ejemplo, orientado a cálculos y evaluaciones de fórmulas algebraicas, es mucho más asequible para científicos e ingenieros que para quien se dedica al tratamiento de transacciones comerciales. No obstante, es tradicional distinguir dos zonas en esa escala imaginaria: la de los llamados *lenguajes de bajo nivel,* más cercanos al lenguaje de la máquina y la de los conocidos como *lenguajes de alto nivel.* Por encima de éstos podemos situar a los *lenguajes declarativos.*

Bajo la etiqueta de «lenguajes de bajo nivel» suelen englobarse, aparte de los lenguajes de máquina, los lenguajes *ensambladores* y *macroensambladores.* En un lenguaje ensamblador, cada instrucción no es más que la codificación simbólica de una instrucción del lenguaje de máquina. Por tanto, cada ordenador tiene su lenguaje de máquina y su lenguaje ensamblador. Ello implica que el programador está obligado a conocer con bastante detalle las peculiaridades de la máquina: número y longitud de los registros, modos de direccionamiento, formatos de representación interna, etc. Los traductores para lenguajes ensambladores se llaman *ensambladores.*

Los lenguajes llamados de alto nivel son los más utilizados: FORTRAN, BASIC, COBOL, PL/1, Pascal, Ada, etc. En principio, son independientes de la estructura de la máquina en la que se ejecutan los programas. Por ello, se dice que éstos son *transportables:* un programa escrito en un lenguaje de alto nivel para un ordenador puede ejecutarse en otro cualquiera (siempre que se disponga del traductor o del intérprete adecuado). Los traductores para lenguajes de alto nivel se llaman *compiladores,* y, como veremos en el apartado 4, son bastante más complejos que los ensambladores.

Veamos con algunos ejemplos las características básicas de los distintos niveles de lenguajes.

2.2. Lenguajes de bajo nivel

2.2.1. *Un lenguaje de máquina*

Para ilustrar con un ejemplo muy sencillo cómo es un lenguaje de máquina y un lenguaje ensamblador y la programación con ellos, consideremos un modelo de ordenador con longitud de palabra* de 16 bits, con un solo registro de 16 bits llamado acumulador y en el que las instrucciones de máquina son también de 16 bits. Las instrucciones pueden hacer referencia a una dirección de memoria para operar con su contenido. En nuestro modelo no consideraremos más que un modo de direccionamiento, el «directo». Concretamente, de los 16 bits de cada instrucción, los cuatro primeros corresponden al código de operación (por lo que puede haber $2^4 = 16$ operaciones diferentes) y los 12 siguientes a una dirección de memoria (pueden

* Una «palabra» es el conjunto de bits que se trasfiere en paralelo entre la memoria y la unidad de procesamiento.

direccionarse, así, $2^{12} = 4.096$, o $4K$ palabras de la memoria). La dirección es, pues, un número entero comprendido entre 0 y 4.095. Algunos de los códigos de operación son:

0000: parar la ejecución (los 12 bits de dirección se ignoran);
0001: cargar en el acumulador, es decir, llevar el contenido (los 16 bits) de la palabra de memoria direccionada en el campo de dirección (los 12 bits que siguen a 0001) al acumulador, borrando lo que previamente hubiera en éste.
0010: almacenar el acumulador, es decir, llevar su contenido (los 16 bits) a la palabra de memoria direccionada, sin que se modifique lo que hay en el acumulador;
0011: sumar al acumulador, es decir, sumar su contenido con el de la palabra de memoria direccionada y dejar el resultado en el mismo acumulador;
0100: restar del contenido del acumulador el de la palabra direccionada y dejar el resultado en el acumulador;
0101: multiplicar el contenido del acumulador por el de la palabra direccionada y dejar el resultado en el acumulador.

Supongamos que queremos escribir en lenguaje de máquina un programa para realizar la operación $A*(B + C)$, donde A, B y C son datos numéricos que el programa debe leer de algún periférico de entrada. Entre los códigos de operación que acabamos de definir no hemos incluido operaciones de comunicación con los periféricos, aunque, naturalmente, todo ordenador debe disponer de ellas; el hacerlo nos habría obligado a entrar en demasiados detalles, porque lo que el ordenador comunica con los periféricos son códigos de control y de caracteres, y es preciso utilizar programas de conversión de caracteres a números en binario, y viceversa. Por tanto, obviando este aspecto, consideremos que los números A, B y C se han leído de algún modo y han quedado almacenados en la memoria en las palabras de dirección 0, 1 y 2, respectivamente. El «algoritmo» consistirá en llevar B al acumulador (cargar el contenido de la palabra de memoria de dirección 1), sumarle C (contenido de 2) y lo que resulte multiplicarlo por A (contenido de 0), con lo que quedará en el acumulador $A*(B + C)$, que podemos almacenar, por ejemplo, en la dirección 3 (para, posteriormente, con otras instrucciones, escribir el resultado por un periférico de salida). El programa en lenguaje de máquina sería:

```
...
...
0001000000000001
0011000000000010
0101000000000000
0010000000000011
...
...
0000000000000000
```

Las dos líneas de puntos iniciales indican que habrá unas instrucciones de entrada de datos para leer los números, convertirlos a binario y guardarlos en las direcciones

0, 1 y 2. A continuación vienen cuatro instrucciones, cada una de 16 bits. La primera tiene código de operación 0001 (cargar) y el número 1 en su campo de dirección; es decir, su ejecución provocará que el contenido de esa palabra de memoria se transfiera al acumulador. El código de operación de la segunda instrucción es 0011 (sumar) y su campo de dirección es 2 (10 en binario); por tanto, cuando la máquina la ejecute, su efecto será el de sumar el contenido de la palabra de dirección 2 a lo que en ese momento haya en el acumulador, y dejar la suma en el mismo acumulador. Con la siguiente, se multiplica ese resultado parcial por lo que haya en la dirección 0, y con la cuarta el resultado final se transfiere a la memoria, a la palabra de dirección 3 (11 en binario). A continuación se incluirían las instrucciones para sacar ese resultado por algún periférico, y, finalmente, viene la instrucción de parar.

Los ordenadores son máquinas de *programa almacenado*. Esto quiere decir que la secuencia de instrucciones que constituyen un programa se carga en la memoria y luego se ejecutan una detrás de otra, salvo cuando aparece una *instrucción de salto*. Se llaman así a aquellas que provocan una «ruptura de secuencia» en la ejecución del programa, es decir, que la siguiente instrucción a ejecutar no es la que viene a continuación (almacenada en la siguiente palabra de la memoria), sino la que está almacenada en la dirección indicada por la propia instrucción de salto. Estas instrucciones suelen utilizarse para controlar bucles o acciones condicionales. Por ejemplo, nuestro ordenador puede tener una instrucción de *salto incondicional,* con código 0110, y otra de *salto si acumulador es cero* (0111). La primera, al ejecutarse, hace que la siguiente instrucción a ejecutar sea la que está almacenada en la dirección especificada en su campo de dirección, y la segunda lo mismo pero condicionado al hecho de que el contenido del acumulador esté formado por 16 ceros, de lo contrario, se ejecuta la siguiente de la sentencia*. Obsérvese que ello obliga al programador a conocer exactamente las direcciones en que van a almacenarse las instrucciones de su programa. Otra instrucción interesante es la de *salto a subprograma* (con código, por ejemplo, 1000), cuyo efecto es el mismo del salto incondicional, pero guardando en algún sitio (en otro registro, o en una determinada palabra de la memoria) la *dirección de retorno,* dirección de la instrucción que le sigue en la secuencia; el *subprograma* es un conjunto de instrucciones que termina con una *instrucción de retorno,* y es muy útil cuando han de realizarse repetidamente ciertas operaciones a lo largo de un programa (por ejemplo, cálculo de la raíz cuadrada, lectura de datos, etc.).

Veamos un ejemplo de programación en lenguaje de máquina que hace uso de las instrucciones de salto. Se leen dos números, A y B (suponemos que $A < B$) y se trata de igualar el primero con el segundo a base de sumarle sucesivamente una unidad, como indica el organigrama de la figura 5.2. Supondremos que a partir de la dirección 20 (10100 en binario) hay almacenado un subprograma que permite leer números. La primera vez que «se le llama» (con la instrucción de salto a subprograma) lee un número y lo almacena en la dirección 0; la siguiente, lee otro y lo almacena en la dirección 1, y así sucesivamente. Nuestro programa puede ser el siguiente:

* Suponemos que la representación del número «0» en esta máquina se hace mediante 16 ceros.

dirección de memoria (en decimal)	*contenido (binario)*	*comentario*
0	----------	aquí se almacenará el primer dato
1	----------	aquí, el segundo
2	0000000000000001	esta es la constante 1
3	1000000000010100	salto al subprograma de lectura (lee un dato y lo almacena en la dirección 0)
4	1000000000010100	salto al subprograma de lectura (lee otro dato y lo almacena en 1)
5	0001000000000000	carga el primer número (A)
6	0100000000000001	le resta el segundo (B)
7	0111000000001100	si $A = B$, salta a ejecutar la instrucción almacenada en la dirección 12
8	0001000000000000	si no, carga A (almacenado en 0)
9	0011000000000010	le suma una unidad (almacenada en 2)
10	0010000000000000	almacena el resultado en la dirección de A (0)
11	0110000000000101	y vuelve a la instrucción almacenada en la dirección 5
12	0000000000000000	cuando $A = B$ se para

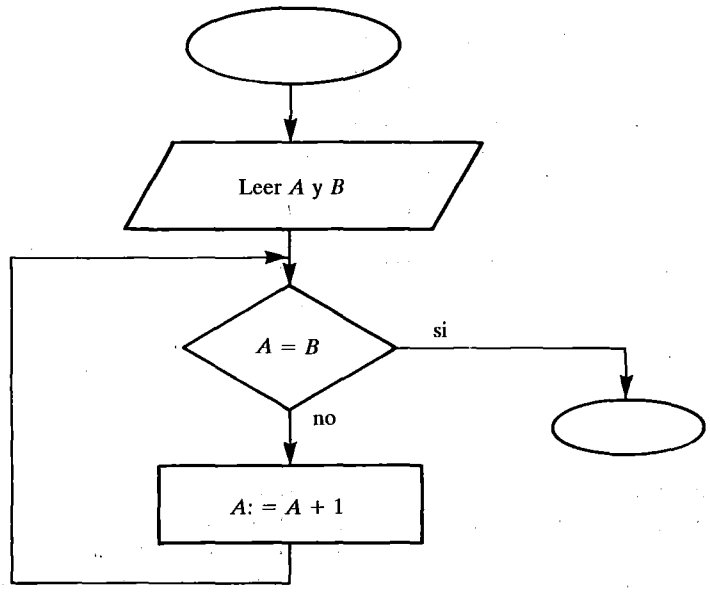


FIGURA 5.2.

2.2.2. *Un lenguaje ensamblador*

En un lenguaje ensamblador no hay que codificar en binario, porque a cada código de operación se le asigna un nombre nemotécnico, y se pueden también utilizar nombres simbólicos, llamados *etiquetas,* para designar a las direcciones de memoria. Por ejemplo, los códigos de operación de un lenguaje ensamblador para el modelo de ordenador definido en el apartado anterior podrían ser:

código de máquina	*código ensamblador*
0000	PAR
0001	CAR
0010	ALM
0011	SUM
0100	RES
0101	MUL
0110	SAI
0111	SAC
1000	SAS
...	...

Para el primero de los dos ejemplos anteriores, un posible programa en ensamblador sería:

```
A    MEM    1
B    MEM    1
C    MEM    1
D    MEM    1
     ...
     ...
     CAR    B
     SUM    C
     MUL    A
     ALM    D
     ...
     ...
     PAR
     FIN
```

El programa traductor (el ensamblador) deberá tomar estos códigos como datos de entrada y generar los códigos binarios en lenguaje de máquina.

Las cuatro primeras líneas del programa que hemos escrito no representan instrucciones de máquina, sino «pseudoinstrucciones». Se trata, en realidad, de instrucciones para el ensamblador. Concretamente, le dicen a éste que reserve cuatro palabras de la memoria y que las identifique mediante las etiquetas *A, B, C* y *D*. El

ensamblador, al ejecutarse, se encargará de atribuir direcciones reales a esos nombres. Si, por ejemplo, atribuye a «*B*» la dirección 1, entonces la instrucción «CAR *B*» se traducirá por «0001000000000001», que es la misma que aparecía en el programa escrito en lenguaje de máquina.

Como puede observarse, gracias al uso de etiquetas el programador no tiene que preocuparse de direcciones numéricas reales de la memoria.

«FIN» es otra pseudoinstrucción. Le indica al ensamblador el final del programa fuente.

Para el segundo ejemplo, podemos escribir este programa en ensamblador:

```
A       MEM   1
B       MEM   1
UNO     CEN   1
        SAS   LEER
        SAS   LEER
BUCLE   CAR   A
        RES   B
        SAC   FIN
        CAR   A
        SUM   UNO
        ALM   A
        SAI   BUCLE
FIN     PAR
LEER    (a partir de aquí vendrían las instrucciones del
        subprograma de lectura de datos)
        ...
        ...
        FIN
```

Hemos supuesto que el ensamblador reconoce la pseudoinstrucción «CEN»: crear una constante entera.

El modelo de ordenador que hemos utilizado para estos ejemplos es muy simple. La máquina reales disponen generalmente de varios registros y modos de direccionamiento que es preciso conocer para poder programarlas en lenguaje ensamblador.

2.2.3. *Lenguajes macroensambladores*

En los lenguajes *macroensambladores* hay *macroinstrucciones* (abreviadamente, «macros»), instrucciones que en el proceso de traducción (ensamblaje) se traducen no en una sino en varias instrucciones de máquina. Ello facilita algo la tarea de programación, pero sigue siendo válido lo dicho acerca de la necesidad de conocer los detalles de la estructura de la máquina.

Existen «macros del sistema», previstas en el propio lenguaje (especialmente para operaciones de comunicación con los periféricos), y, además, el programador puede definir sus propias «macros» para secuencias de instrucciones que tenga que repetir varias veces en su programa. La diferencia con los subprogramas es que el ensambla-

dor *expande* las «macros», es decir, en el proceso de ensamblaje, cada vez que encuentra una genera toda la secuencia de instrucciones de máquina que le corresponde.

2.3. Lenguajes de alto nivel

2.3.1. *Sentencias*

Un programa en un lenguaje de alto nivel está compuesto por una sucesión de *sentencias* (normalmente, el nombre «instrucción» se reserva para las instrucciones del nivel de máquina). Aquí hay un pequeño problema terminológico que conviene aclarar para evitar confusiones. En efecto, en teoría de lenguajes llamamos «sentencia» a cualquier cadena perteneciente a un lenguaje (capítulo 2, apartado 3); de acuerdo con esto, una sentencia sería cualquier programa *completo* que fuera sintácticamente correcto. Sin embargo, cuando hablamos de lenguajes de programación solemos entender por «sentencia» no un programa completo, sino cada una de las unidades elementales que forman parte de él y que tienen un significado en la definición del lenguaje. Así, en Pascal, existen sentencias de asignación como

$$x := 3$$

(sustituir el valor que tenga la variable x por la constante 3), sentencias de acción condicional, como

if EB then S1 else S2

(si el resultado de evaluar la expresión booleana EB es «true» (verdadero) entonces ejecutar la sentencia S1, si no, S2), sentencias de iteración, como

while EB do S

(ejecutar la sentencia S mientras el resultado de la expresión booleana EB sea «true»), etc.

El primero de los programas cuya codificación en lenguaje de máquina y en ensamblador vimos más arriba se podría codificar así en Pascal:

```
program aritmetico (input, output);
begin
   read (A, B, C);
   D := A*(B + C);
write (D)
end.
```

donde «begin» y «end» son «palabras clave» (símbolos terminales en la terminología de la teoría de lenguajes) que indican el principio y el final de un programa, y, en

general, de un grupo cualquiera de sentencias. El programa consta de tres sentencias: una de lectura («read»), otra de asignación y una final de escritura («write»). El símbolo «;» separa cada sentencia de la siguiente.

El programa en Pascal para el segundo de los ejemplos podría ser:

```
program igualar (input, output);
begin
   read (A, B);
   while A⟨ ⟩B do A : = A + 1
end.
```

Otros ejemplos de programas en Pascal pueden encontrarse en los capítulos 3 y 7 del tema «Algoritmos».

2.3.2. *Procedimientos y funciones*

Una característica muy importante de todos los lenguajes de alto nivel es que facilitan el uso de *procedimientos* («procedures»). Un procedimiento es la generalización de un subprograma: es una unidad programada independiente que se ejecuta cuando se la llama desde un programa, o desde otro procedimiento, o, incluso, desde el mismo procedimiento.

Un procedimiento consta de tres partes: su *nombre* (con el que se le llama), una lista de *parámetros formales* y el *cuerpo,* donde se incluyen las sentencias que forman el procedimiento. Por ejemplo, un procedimiento en Pascal para calcular el factorial de un número entero mediante una iteración es:

```
procedure factorial (n: integer; var fact: integer);
   var i: integer;
   begin
      fact: = 1;
      for i: = 2 to n do fact: = fact * i
   end;
```

El procedimiento utiliza dos parámetros formales: *n* es un parámetro de entrada, donde se da el número cuyo factorial hay que calcular; fact es un parámetro de salida, donde el procedimiento devuelve el valor calculado del factorial, y va precedido de la palabra clave «var» porque es una variable cuyo valor queda determinado tras la ejecución del procedimiento. Si, por ejemplo, desde un programa queremos calcular el factorial de 5, dejando el resultado en una variable de este programa llamada *f*, incluiremos en él la sentencia de llamada

```
factorial (5, f);
```

«5» y «*f*» son los *parámetros reales,* que, cuando el procedimiento se ejecute atendiendo a esta llamada, irán a sustituir a los parámetros formales.

Un procedimiento puede comunicarse también con el programa que le llama (o sea, recibir datos o entregar resultados) mediante el uso de *variables globales,* es decir, variables definidas en el programa y accesibles también desde el procedimiento. (En el procedimiento «factorial» que hemos escrito, «*i*» es una *variable local,* porque está definida dentro del cuerpo del procedimiento.)

Una *función* es otro tipo de subprograma. Al igual que un procedimiento, puede tener parámetros formales, y compartir variables globales, pero, a diferencia de él, devuelve directamente un valor al programa que la llama. Por ejemplo, para el factorial podemos escribir esta función en Pascal:

```
function factorial (n: integer): integer;
    var i, fact: integer;
    begin
        fact: = 1;
        for i: = 2 to n do fact: = fact * i;
        factorial: = fact
    end;
```

La llamada desde el programa para calcular el factorial de 5 y dejarlo en la variable *f* implica ahora el uso de una sentencia de asignación, para asignar a *f* el valor que devuelve la función:

```
f: = factorial (5)
```

Como hemos dicho, los procedimientos y las funciones pueden utilizarse en la mayoría de los lenguajes, y, desde luego, en Pascal, de manera recursiva. Por ejemplo, otra forma de escribir una función para el cálculo del factorial puede ser:

```
function factorial (n: integer): integer;
    begin
        if n < 2 then factorial: = 1
                 else factorial: = n * factorial (n - 1)
    end;
```

donde, como vemos, la función «se llama» a sí misma.

El uso de procedimientos y funciones permite desarrollar programas de manera modular y jerárquica, materializando así uno de los principios básicos de la programación estructurada (tema «Algoritmos», capítulo 4, apartado 2.2).

2.3.3. *Tipos de datos*

Los lenguajes modernos tienden a ser «fuertemente tipados» (desafortunada traducción de «strongly typed»). Esto quiere decir que todas las variables que intervienen en un programa, en una función o en un procedimiento, tienen un *tipo,* y este tipo debe ser declarado explícitamente mediante las oportunas *sentencias de*

declaración. Así, en los últimos ejemplos sobre procedimientos y funciones hemos incluido la declaración «integer» (entero) para las variables que intervenían, y también para la misma función, puesto que ésta devuelve un valor que también tiene un tipo (en este caso, entero). En general, un *tipo de datos* es un conjunto de valores posibles con unas operaciones asociadas. Aunque la obligación de declarar los tipos pueda, a primera vista, parecer un engorro innecesario (en relación con lenguajes clásicos, como FORTRAN, en los que no es preciso tal cosa) la ventaja es que la fiabilidad de los programas mejora, pues gracias a eso se ponen de manifiesto, durante la compilación o la ejecución, muchos errores que de otro modo pasarían inadvertidos o serían muy difíciles de localizar.

Pascal incluye en su definición cuatro tipos «estándar»: *integer* (el valor de la variable puede ser cualquier número entero comprendido en el rango de representación de la máquina), *boolean* (la variable sólo puede tomar dos valores, *true* y *false*), *real* (un número real comprendido en el rango de representación) y *char* (un carácter). A partir de ellos, el programador puede definir *tipos estructurados: array, record,* etc., cuyos componentes pueden ser también estructurados. Es posible, pues, construir *jerarquías de estructuras de datos,* insistiendo así en el diseño modular y jerárquico de los programas.

2.4. Lenguajes declarativos

2.4.1. *Programación imperativa y programación declarativa*

Los lenguajes de alto nivel comentados en el apartado anterior liberan, como hemos visto, al programador de las características concretas de la máquina en la que se ejecutan finalmente los programas. No obstante, lo mismo que los de bajo nivel, son *lenguajes imperativos.* Esto quiere decir que el programador, para resolver un problema, ha de diseñar un algoritmo y materializarlo en un programa que le diga a la máquina real (caso de los de bajo nivel) o a la máquina virtual (caso de los de alto nivel), paso a paso, qué tiene que hacer para resolver el problema. La idea de los *lenguajes declarativos* es que el programador sólo tenga que declarar o especificar el problema, y que sea el procesador de lenguaje el que se encargue del algoritmo. Dicho de otro modo, que el programador indique «qué» es lo que hay que resolver y la máquina (virtual) se ocupe de los detalles del «cómo».

La idea no es nueva, y está ya implícita en ciertos lenguajes diseñados para aplicaciones específicas, como cálculos estadísticos (SPSS, BMDP, etc.), consulta de bases de datos (IMS, NATURAL, etc.), simulación (CSMP, GPSS, etc.). Son también declarativos los lenguajes llamados *generadores.* Por ejemplo, un generador de informes (RPG: «report program generator») acepta descripciones de la estructura de representación de ciertos datos y del formato deseado de salida y genera un programa para imprimir los datos. Pero los dos «estilos» de programación declarativa más conocidos son los correspondientes a la programación funcional y a la programación lógica.

2.4.2. *Programación funcional*

El fundamento teórico de la programación funcional tiene su origen en el problema de la computabilidad de funciones matemáticas, y, concretamente, en el llamado «cálculo lambda». Sin entrar en esta base matemática, digamos simplemente que un programa funcional es un conjunto de ecuaciones que definen funciones a partir de otras funciones más sencillas o de primitivas. Por ejemplo, en el programa

$$\text{max2}[x, y]: = \text{if } x > y \text{ then } x \text{ else } y$$
$$\text{max3}[x, y, z]: = \text{max2}[\text{max2}[x, y], z]$$

la primera sentencia define la función «max2» a partir de las primitivas «if», «>»,, «then» y «else», y la segunda define la función «max3» a partir de «max2». Es importante señalar que el símbolo «:=» no representa, como en algunos lenguajes imperativos, una asignación; aquí significa «se define como». De hecho, en LISP, por ejemplo, no se utiliza este símbolo, sino la palabra clave «defun».

Las definiciones de funciones pueden ser recursivas. Por ejemplo:

$$\text{fact[N]} := \text{if N} = 0 \text{ then 1 else N} * \text{fact[N} - 1]$$

Un concepto fundamental en programación funcional es el de la *transparencia referencial.* Una función es referencialmente transparente, o *pura,* si su efecto queda definido exclusivamente por el valor dado a sus parámetros. Para ello es preciso que la función no tenga *efectos laterales,* es decir, que no altere ni el valor de sus parámetros ni variables globales. La idea básica de la programación funcional es la de expresar todos los algoritmos mediante aplicaciones de funciones puras (por ello, a los lenguajes funcionales se les llama también *lenguajes aplicativos*).

En principio, también puede hacerse programación funcional con lenguajes imperativos. Basta para ello utilizar funciones que no manejen variables globales ni alteren sus parámetros, como las dos funciones escritas en Pascal en el apartado 2.3.2. El problema está en que el único modo de obtener un resultado es a través del valor del objeto que devuelve la función, y en los lenguajes imperativos clásicos el tipo de este objeto está muy limitado.

En un lenguaje funcional los objetos pueden ser *listas.* Una lista es una secuencia finita de elementos, donde cada elemento es un «átomo» (numérico, como «109» o simbólico, como «ATOMO») u otra lista. Las listas permiten una representación uniforme y versátil de gran variedad de objetos, aunque a veces resulten de difícil lectura. Por ejemplo, las expresiones algebraicas

$$x + y$$

y

$$ax^2 + bx + c = 0$$

se escribirían así con notación de lista:

$$(\text{suma } x\ y)$$

y

$$(\text{igual}(\text{suma}(\text{mul } a\ (\text{cuad } x))(\text{suma}(\text{mul } b\ x)c))0)$$

Para manejar listas, los lenguajes funcionales suelen tener incorporada una *función de construcción* que permite encadenar dos listas. Esta función se puede escribir de forma prefija: cons[$L1$, $L2$] o de forma infija: $L1.L2$. Por ejemplo, la lista (a, b, c) se puede construir así a partir de los átomos a, b y c:

$$\text{cons}(a, \text{cons}(b, (\text{cons}(c,()))))$$

o bien:

$$(a.(b.(c.())))$$

donde ( ) representa una lista especial: la *lista vacía.*

La facilidad de manejo de expresiones simbólicas y de jerarquías de objetos representadas como listas es la razón por la que LISP, el lenguaje funcional más antiguo, es también el más utilizado en inteligencia artificial, donde el procesamiento simbólico predomina sobre el numérico. Hay otros lenguajes funcionales más modernos, pero mucho menos conocidos que LISP, como FP, HOPE, KRC, SETL, etc.

2.4.3. *Programación lógica*

El fundamento teórico de la programación lógica es el cálculo de predicados de primer orden. Actualmente, el único lenguaje para programación lógica (salvo algunas otras propuestas aún en estado embrionario) es Prolog. En Prolog, las sentencias son cláusulas de Horn (ver tema «Lógica», capítulo 4, apartado 5.6) compuestas con fórmulas atómicas del cálculo de predicados y escritas de un modo especial: primero se escribe la cabeza de la cláusula (o consecuente del condicional), si existe, y luego, separado por el símbolo «:−» (o «←»), el cuerpo (o antecedente). Así, la sentencia

$$A :- B, C, D.$$

equivale a la sentencia de la lógica

$$B \wedge C \wedge D \rightarrow A$$

donde A, B, C y D representan predicados.

Por ejemplo, un programa que declara unos hechos de parentesco y define la relación «abuelo» es:

```
madre (ana, luis).
padre (josé, ana).
abuelo (X, Z) :- padre (X, Y), padre (Y, Z).
abuelo (X, Z) :- padre (X, Y), madre (Y, Z).
```

Este programa corresponde exactamente al ejemplo 1.7.4 visto en lógica de predicados (tema «Lógica», capítulo 4), salvo que, como puede observarse, la definición de la relación «abuelo» se ha descompuesto en dos sentencias para que sean cláusulas de Horn.

Si se dispone de un interpretador de Prolog y del anterior programa en memoria, se pueden hacer consultas. Por ejemplo:

```
?- abuelo (josé, luis).
SI

?- abuelo (josé, ana).
NO

?- abuelo (X, luis)
X = josé
```

Para obtener las respuestas, el interpretador de Prolog aplica resolución y refutación (tema «Lógica», capítulo 4, apartado 4.5).

En Prolog se pueden definir también relaciones de modo recursivo. Por ejemplo, un programa para definir «antepasado» (tema «Lógica», capítulo 4, ejemplo 1.7.5) es:

```
progenitor  (X, Y) :- padre (X, Y).
progenitor  (X, Y) :- madre (X, Y).
antepasado (X, Y) :- progenitor (X, Y).
antepasado (X, Y) :- progenitor (X, Z), antepasado (Z, Y).
```

Además, con Prolog se pueden definir también funciones y manejar listas. Por ello, se usa cada vez más en inteligencia artificial como alternativa al LISP. Sin embargo, éste tiene la ventaja de contar ya con entornos de programación bastante elaborados.

2.5. Entornos de programación

Asociadas a un lenguaje, pueden diseñarse «herramientas», que son programas para facilitar su uso, y, en general, mejorar la productividad del desarrollo de programas. El conjunto integrado de herramientas asociado a un lenguaje se llama *entorno de programación,* y, dependiendo del número de herramientas disponibles y de las facilidades que ofrecen, puede ser más o menos potente. Ya hemos hablado de

un tipo de herramienta: los procesadores de lenguaje (traductores e interpretadores). Los *editores* permiten escribir y modificar con facilidad el programa fuente, los *depuradores* ayudan a detectar y corregir errores, etc. Otras herramientas más avanzadas ayudan a la especificación de programas y permiten demostrar, antes de su ejecución, si son correctos (satisfacen las especificaciones) o no.

Para el diseño sistemático y riguroso de todas estas herramientas es necesario disponer de una *definición formal* del lenguaje.

3. DEFINICIONES SINTÁCTICAS

3.1. Notación de Backus

La sintaxis de la práctica totalidad de los lenguajes de programación puede expresarse mediante una gramática libre de contexto (salvo algunas excepciones que comentaremos en el apartado 3.2). La notación de Backus, generalmente conocida como «BNF» (de «Backus Normal Form», o «Backus-Naur Form») no es más que una forma de escribir las producciones, que sólo difiere de la que hemos visto en el capítulo 2 en que en lugar de «→» se utiliza «::=», y, además, se puede abreviar la escritura de varias reglas que compartan el mismo antecedente, $A ::= \alpha_1$, $A ::= \alpha_2$, $A ::= \alpha_3$, ... en una sola línea, así: $A ::= \alpha_1|\alpha_2|\alpha_3|$ ...

Normalmente, para mayor claridad, no se utilizan letras aisladas para los símbolos, sino palabras completas. Para distinguir a los símbolos del alfabeto auxiliar de los del alfabeto terminal, los primeros se escriben, como ya hemos hecho en algunos ejemplos anteriores, encerrados entre paréntesis angulares: ⟨sentencia⟩, ⟨identificador⟩, etc., y los del terminal sin paréntesis: begin, if, etc. Concretamente, el símbolo inicial de una gramática para un lenguaje de programación puede ser ⟨programa⟩. Todo programa correcto $x \in L$, podrá derivarse en esa gramática mediante aplicación primero de la regla que tenga ⟨programa⟩ como antecedente y después de otras reglas en el orden adecuado para obtener tal programa.

Hay, además, algunas extensiones de la notación BNF original muy útiles para abreviar la escritura de las reglas recursivas. Los corchetes indican que lo que figura dentro puede repetirse cualquier número de veces (o no aparecer), y los paréntesis cuadrados lo mismo, pero apareciendo al menos una vez. Así, en la gramática para expresiones aritméticas con números positivos y operaciones «+» y «*» definida en el ejemplo 5.5 del capítulo 3, las dos primeras reglas:

$$\langle EA \rangle ::= \langle EA \rangle + \langle TERM \rangle$$
$$\langle EA \rangle ::= \langle TERM \rangle$$

(una expresión es o bien un sólo término o bien una suma de términos) pueden escribirse en BNF así:

$$\langle EA \rangle ::= \langle TERM \rangle \ \{+\langle TERM \rangle\}$$

Y las que generan ⟨CTE⟩:

$$\langle \text{CTE} \rangle ::= \langle \text{CTE} \rangle \langle \text{DIG} \rangle$$
$$\langle \text{CTE} \rangle ::= \langle \text{DIG} \rangle$$

(una constante es una secuencia de uno o más dígitos) así:

$$\langle \text{CTE} \rangle ::= \langle \text{DIG} \rangle \{ \langle \text{DIG} \rangle \}$$

o bien:

$$\langle \text{CTE} \rangle ::= [\langle \text{DIG} \rangle]$$

3.2. Gramáticas para lenguajes de programación

Decíamos más arriba que la sintaxis de los lenguajes de programación, salvo excepciones, puede expresarse mediante gramáticas libres de contexto. Estas excepciones se refieren a algunos aspectos de la definición de los lenguajes. Por poner un ejemplo, en un lenguaje como Pascal es obligatorio que se declare el tipo de las variables previamente a su uso en el programa (*cf.* apartado 2.3.3). Formalmente, esto se traduce en que dada una cadena (programa) de la forma

$$\alpha \text{ begin } \beta \; x \; T \text{ end.}$$

donde x es la variable y α, β, T representan el resto del programa, si x figura en α (y suponiendo que α, β, T están correctamente construidas) entonces la cadena pertenece al lenguaje, y si no, no. Pues bien, se puede demostrar que un lenguaje así no es libre de contexto.

Lo que se suele hacer es ignorar estas excepciones y definir, en cualquier caso, el lenguaje mediante una gramática libre de contexto. Por ejemplo, en los compiladores la detección de variables no declaradas no tiene lugar en la fase de «análisis sintáctico» (en la que se aplican las reglas libres de contexto), sino después, en la de «análisis semántico» (*cf.* apartado 5.2.1).

Por otra parte, dentro del tipo general de gramáticas libres de contexto existen clases definidas por ciertas restricciones sobre la forma de las reglas. Por ejemplo, en el apartado 6 del capítulo 3 hemos mencionado un subconjunto llamado «gramáticas deterministas». Los reconocedores, en la práctica, van a materializarse en algoritmos que realizan un análisis sintáctico del programa fuente, y estos algoritmos pueden ser más sencillos y más eficaces si la gramática cumple con ciertas restricciones. De hecho, cuando se diseña un lenguaje nuevo las decisiones sobre la gramática del mismo vienen muy condicionadas por los algoritmos de reconocimiento que luego habrán de programarse.

En los ejemplos que desarrollaremos a continuación vamos a imponer dos condiciones que conducen a un tipo de gramática determinista y a un algoritmo de reconocimiento sencillo, como veremos en el apartado 5.2.3.

La primera condición es que si existe una regla de la forma

$$A ::= \alpha|\beta$$

entonces no puedan derivarse a partir de α y β cadenas que comiencen con un mismo símbolo terminal. En particular, no sería lícita una regla como

$$A ::= \alpha\beta_1|\alpha\beta_2$$

En éste, como en otros muchos casos, es fácil encontrar otras reglas que generen las mismas cadenas y que cumplan la condición enunciada. Concretamente, basta con introducir un nuevo símbolo auxiliar, R, y sustituir la regla anterior por:

$$A ::= \alpha R$$
$$R ::= \beta_1|\beta_2$$

Ese procedimiento de «sacar factor común» puede conducir a algo que normalmente está prohibido en la definición de las gramáticas libres de contexto, pero que pese a todo se utiliza para cumplir la condición. Se trata de que, excepcionalmente, el consecuente de una regla puede ser la cadena vacía. Por ejemplo, en el caso de la regla

$$A ::= \alpha|\alpha\beta$$

que sustituiremos por

$$A ::= \alpha R$$
$$R ::= \beta|\lambda$$

(Obsérvese que, pese a tener la regla $R ::= \lambda$, las cadenas derivadas a partir de A siguen siendo de longitud no decreciente).

Este caso se da, por ejemplo, con la sentencia «if» de los lenguajes de programación, con sus dos posibilidades:

⟨sentencia⟩ ::= if ⟨expresión⟩ then ⟨sentencia⟩|
if ⟨expresión⟩ then ⟨sentencia⟩ else ⟨sentencia⟩

La segunda condición es que la gramática no sea *recursiva por la izquierda,* lo que quiere decir que no puedan darse derivaciones de la forma $A \overset{+}{\Rightarrow} A\alpha$. Existe un algoritmo general para transformar la recursión por la izquierda en recursión por la derecha. Veamos aquí el caso más sencillo y más frecuente, que es el de la existencia de dos reglas como:

$$A ::= \alpha|A\beta$$

que pueden sustituirse por estas otras:

$$A ::= \alpha R$$
$$R ::= \beta R|\lambda$$

o bien, usando la notación BNF ampliada:

$$A ::= \alpha\{\beta\}$$

De hecho, esta transformación es la que hemos utilizado al final del apartado anterior cuando hemos reescrito en notación BNF algunas reglas de una gramática para expresiones aritméticas.

3.3. Gramática para un lenguaje ensamblador

Como primer ejemplo de uso de la notación BNF para la definición formal de un lenguaje de programación, vamos a considerar un lenguaje ensamblador muy sencillo: el mismo cuya sintaxis y semántica hemos definido informalmente en el apartado 2.2.2.

El inconveniente de las definiciones informales es que suelen ser incompletas y ambiguas. Por ejemplo, podemos decir que un programa en nuestro lenguaje ensamblador es una secuencia de instrucciones terminada por la pseudoinstrucción «FIN». Esto podría conducirnos a escribir así la primera regla:

⟨programa⟩ ::= ⟨instrucción⟩⟨programa⟩|FIN

Ahora bien, esta regla indicaría que las instrucciones deberían ir una tras otra en la misma línea, mientras que, como hemos visto en los ejemplos del apartado 2.2.2, se escriben cada una en una línea separada. Para reflejar esta imposición en la forma de la regla, introduciremos un símbolo terminal especial, «cl» (cambio de línea» entre ⟨instrucción⟩ y ⟨programa⟩. Por otra parte, la definición anterior de programa como secuencia de instrucciones es incompleta: puede haber también pseudoinstrucciones (MEM y CEN). Además, la regla anterior tiene un problema: la última línea de un programa sería el símbolo terminal «FIN» con la letra «F» en la primera columna de la línea. Pero, como se desprende de los ejemplos, las etiquetas se escriben justamente desde la primera columna, por lo que el ensamblador no sabría que «FIN» es una pseudoinstrucción y no una etiqueta. La regla sintáctica es que los códigos de las instrucciones y pseudoinstrucciones no pueden empezar en la primera columna: tienen que llevar delante al menos un espacio en blanco. Llamaremos ⟨separ⟩ a la categoría sintáctica que significa «uno o más espacios en blanco». Finalmente, vamos a considerar la posibilidad de escribir comentarios. Un comentario es un texto que el procesador (ensamblador, en este caso) debe ignorar. Supondremos que los comentarios se escriben en líneas diferentes de las que corresponden a las instrucciones y pseudoinstrucciones. De acuerdo con todo esto, escribimos dos reglas:

(1) ⟨programa⟩ ::= ⟨línea⟩cl⟨programa⟩|⟨separ⟩FIN
(2) ⟨línea⟩ ::= ⟨instrucc⟩|⟨pseudoinstrucc⟩|⟨coment⟩

Tenemos ahora que introducir nuevas reglas para obtener derivaciones a partir de ⟨instrucc⟩, ⟨pseudoinstrucc⟩ y ⟨coment⟩. Sabemos que una instrucción (y una pseudoinstrucción) puede llevar etiqueta o no. La etiqueta se escribe a partir de la primera columna de la línea. Supondremos que el código de operación no tiene por qué escribirse en columnas determinadas: basta con que esté separado del final de la etiqueta por uno o más espacios en blanco («⟨separ⟩»), y si no hay etiqueta debe ir precedido por ⟨separ⟩. Hay, además, ciertas instrucciones que tienen que ir acompañadas de una dirección de memoria (SUM, RES, etc.); las pseudoinstrucciones CEN y MEM tienen que ir seguidas de un número. Se separará esta dirección o este número del código de operación mediante ⟨separ⟩. Otras instrucciones, como PAR, CMP (complementar el acumulador), etc., no llevan nada más. En cuanto a los comentarios, supondremos que las líneas correspondientes se distinguen porque en la primera columna tienen el carácter «*» y luego cualquier texto (cadena de caracteres). Todo ello se puede formalizar con las siguientes reglas:

(3) ⟨instrucc⟩ ::= ⟨eti⟩⟨separ⟩⟨resto-instr⟩|
⟨separ⟩⟨resto-instr⟩
(4) ⟨resto-instr⟩ ::= ⟨oper-con-direcc⟩⟨separ⟩⟨direcc⟩|
⟨oper-sin-direcc⟩
(5) ⟨pseudoinstrucc⟩ ::= ⟨eti⟩⟨separ⟩⟨resto-pseudo⟩|
(6) ⟨resto-pseudo⟩ ::= ⟨cod-pseudo⟩⟨separ⟩⟨num⟩
(7) ⟨coment⟩ ::= *{⟨carácter⟩}

Tenemos ahora nuevas categorías sintácticas para las que hemos de introducir nuevas reglas: ⟨eti⟩, ⟨direcc⟩, etc. Escribamos ya, sin extendernos en más explicaciones, el resto de las reglas:

(8) ⟨separ⟩ ::= bl|{bl} («bl» indica «espacio en blanco»)
(9) ⟨letra-may⟩ ::= A|B|C| ... |Z
(10) ⟨letra-min⟩ ::= a|b|c| ... |z
(11) ⟨dígito⟩ ::= 0|1|2|3|4|5|6|7|8|9
(12) ⟨carác-esp⟩ ::= *|?|/|bl| ...
(13) ⟨carácter⟩ ::= ⟨letra-may⟩|⟨letra-min⟩|⟨dígito⟩|⟨carác-esp⟩
(14) ⟨eti⟩ ::= ⟨letra-may⟩{⟨letra-may⟩|⟨dígito⟩}
(15) ⟨num⟩ ::= ⟨dígito⟩|{⟨dígito⟩} (comparar con el ejemplo 4.4 del cap. 2)
(16) ⟨direcc⟩ ::= ⟨eti⟩|⟨num⟩
(17) ⟨oper-con-direcc⟩ ::= CAR|ALM|SUM| ...
(18) ⟨oper-sin-direcc⟩ ::= PAR|CMP| ...
(19) ⟨cod-pseudo⟩ ::= MEM|CEN

Es interesante destacar algunas observaciones:

a) La gramática definida por las anteriores reglas es del tipo libre de contexto, porque en todas ellas el antecedente es un símbolo del alfabeto auxiliar; además, cumple las restricciones enunciadas en el apartado 3.2.

b) La formalización obliga a dejar perfectamente claros todos los detalles, que en

una descripción informal pueden olvidarse. Por ejemplo, una etiqueta, según la regla 14, tiene que estar formada por letras mayúsculas y dígitos, empezando por una letra, cosa que no habíamos dicho en nuestra presentación informal. Esto, ya de por sí, es una ventaja. Otra no menos importante es que la definición formal permite diseñar de modo sistemático los procesadores de lenguaje.

c) Según la regla 1, un programa puede tener un número ilimitado de líneas. De igual modo, las etiquetas (regla 14) y los números (regla 15) no tienen limitación de caracteres. Evidentemente, en la práctica (en la «implementación») han de introducirse restricciones.

d) Las constantes de la pseudoinstrucción CEN (reglas 19, 6 y 15) son siempre números positivos. Es fácil ampliar las gramática para que se admitan también los negativos. Por ejemplo, con una nueva regla:

$$\langle \text{cte} \rangle ::= \langle \text{num} \rangle | + \langle \text{num} \rangle | - \langle \text{num} \rangle$$

y las oportunas modificaciones para que en la generación de CEN (no así en la de MEM) aparezca ⟨cte⟩.

e) Existen otras muchas posibilidades de definición de reglas y categorías sintácticas (símbolos auxiliares) para el mismo lenguaje, que corresponden a gramáticas equivalentes (capítulo 2, apartado 3).

3.4. Gramática para un lenguaje de alto nivel

Tomaremos como ejemplo sencillo de un lenguaje de alto nivel un pequeño subconjunto de Pascal. Presentaremos primero de manera informal su sintaxis y su semántica.

Para simplificar, sólo consideraremos un tipo de dato: entero («integer»). En el programa pueden figurar constantes literales (p. ej., −3, 100, etc.) y constantes y variables denominadas mediante identificadores; estas últimas deben declararse al principio mediante sentencias de declaración. Por ejemplo:

```
program ejemplo;
const menosdos = −2;
      diez = 10;
var   i, j : integer;
begin
      {sentencias del programa}
end.
```

Las constantes y variables pueden combinarse con los operadores aritméticos «+», «−», «*» y «/» formando expresiones aritméticas, y con los operadores relacionales «=», «⟨», «⟩», «⟨=», «⟩=», «⟨ ⟩» formando condiciones que se evalúan como verdaderas (p. ej., 3⟩2) o falsas (p. ej., 3⟨2).

Las sentencias del lenguaje (o «mandatos») posibles son:

* La sentencia de asignación:
  Sintaxis: v := expresión
  Semántica: asocia a la variable v el valor resultante de evaluar la expresión (sustituyéndolo por el que tenía asociado anteriormente).

* La sentencia de acción condicional:
  Sintaxis: if condición then sentencia
  o bien:
  if condición then sentencia1 else sentencia2
  Semántica: si la condición se evalúa como verdadera, entonces se ejecuta la «sentencia1»; si se evalúa como falsa no se hace nada (primer caso) o se ejecuta la «sentencia2» (segundo caso).

* La sentencia de iteración:
  Sintaxis: while condición do sentencia
  Semántica: se ejecuta la sentencia repetidamente siempre que la evaluación de la condición sea verdadera.

* La sentencia de entrada de datos:
  Sintaxis: read(v)
  Semántica: lee un valor del periférico de entrada y lo asigna a la variable v.

* La sentencia de salida de resultados:
  Sintaxis: write (expresión)
  Semántica: evalúa la expresión y escribe el resultado por el periférico de salida.

En todos los casos anteriores, «sentencia» puede ser una sentencia simple o compuesta por una secuencia de varias; en este caso deben ir separadas por «;», precedidas por «begin» y terminadas por «end». Los espacios en blanco y los cambios de línea no tienen significado, y pueden introducirse arbitrariamente para mejorar la legibilidad.

Una definición formal del lenguaje es la dada por las siguientes reglas en notación BNF:

(1) ⟨programa⟩ ::= program ⟨identif⟩; ⟨bloque⟩.
(2) ⟨bloque⟩ ::= [const ⟨decl-const⟩]$_0^1$[var ⟨decl-var⟩]$_0^1$
begin⟨sentencia⟩{;⟨sentencia⟩}end
(3) ⟨decl-const⟩ ::= [⟨identif⟩ = ⟨cte⟩;]
(4) ⟨decl-var⟩ ::= [⟨identif⟩{,identif}:⟨tipo⟩;]
(5) ⟨identif⟩ ::= ⟨letra⟩{⟨letra⟩|⟨dígito⟩}
(6) ⟨cte⟩ ::= ⟨num⟩|+⟨num⟩|−⟨num⟩
(7) ⟨num⟩ ::= [⟨dígito⟩]
(8) ⟨letra⟩ ::= $a|b|$ ... $|z|A|B|$ ... $|Z$
(9) ⟨dígito⟩ ::= 0|1| ... |9
(10) ⟨tipo⟩ ::= integer

(11) ⟨sentencia⟩ ::= begin⟨sentencia⟩{;⟨sentencia⟩}end|
⟨identif⟩ := ⟨expresión⟩|
if ⟨condición⟩ then ⟨sentencia⟩⟨resto⟩|
while ⟨condición⟩ do ⟨sentencia⟩|
read(⟨identif⟩)|write(⟨expresión⟩)
(12) ⟨resto⟩ ::= else ⟨sentencia⟩|λ
(13) ⟨condición⟩ ::= ⟨expresión⟩⟨oper⟩⟨expresión⟩
(14) ⟨oper⟩ ::= = | ⟨ | ⟩ | ⟨= | ⟨ ⟩
(15) ⟨expresión⟩ ::= ⟨expr-s-sg⟩|+⟨expr-s-sg⟩|−⟨expr-s-sg⟩
(16) ⟨expr-s-sg⟩ ::= ⟨término⟩|{+⟨término⟩|-⟨término⟩}
(17) ⟨término⟩ ::= ⟨factor⟩|{*⟨factor⟩|/⟨factor⟩}
(18) ⟨factor⟩ ::= ⟨cte⟩|⟨identif⟩|(⟨expresión⟩)

También aquí debemos señalar algunas observaciones:

a) La notación $[...]_0^1$ significa «una vez o ninguna».

b) Tal como se ha definido «⟨expresión⟩», los operadores «*» y «/» tienen prioridad sobre «+» y «−», a menos que se anule tal prioridad con paréntesis (última parte de la regla 18). El procedimiento seguido para evitar la ambigüedad es el mismo que en el ejemplo 5.5 del capítulo 3.

c) Los símbolos «begin» y «end» cumplen una función con respecto a las sentencias similar a la que tienen los paréntesis con respecto a las expresiones. Si no existieran, una sentencia como

while *C* do *S*1; *S*2

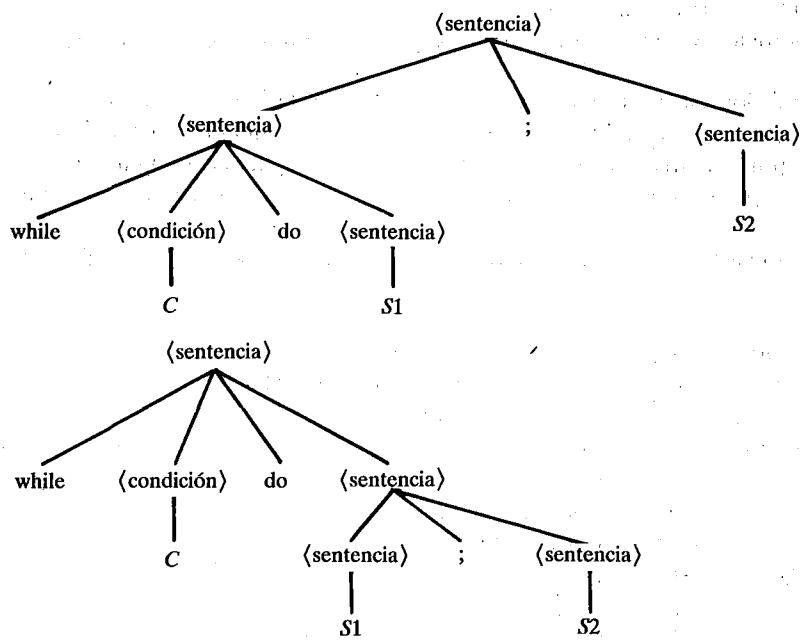


FIGURA 5.3.

sería ambigua, puesto que admitiría las dos derivaciones de la figura 5.3. Con nuestra gramática la única derivación posible es la primera, porque para la segunda habría que escribir:

while C do begin $S1$; $S2$ end

d) Sin embargo, nuestra gramática es ambigua en la parte que genera las sentencias «if». Por ejemplo, para la sentencia

if $C1$ then if $C2$ then $S1$ else $S2$

tenemos los dos árboles de derivación de la figura 5.4. Normalmente, en los lenguajes con este tipo de construcciones se prefiere la interpretación correspondiente al primero de los árboles. En general, se sigue el criterio de emparejar cada «else» con el «then» inmediatamente anterior que no esté emparejado. El lector puede comprobar que si las reglas (11) y (12) se sustituyen por estas otras:

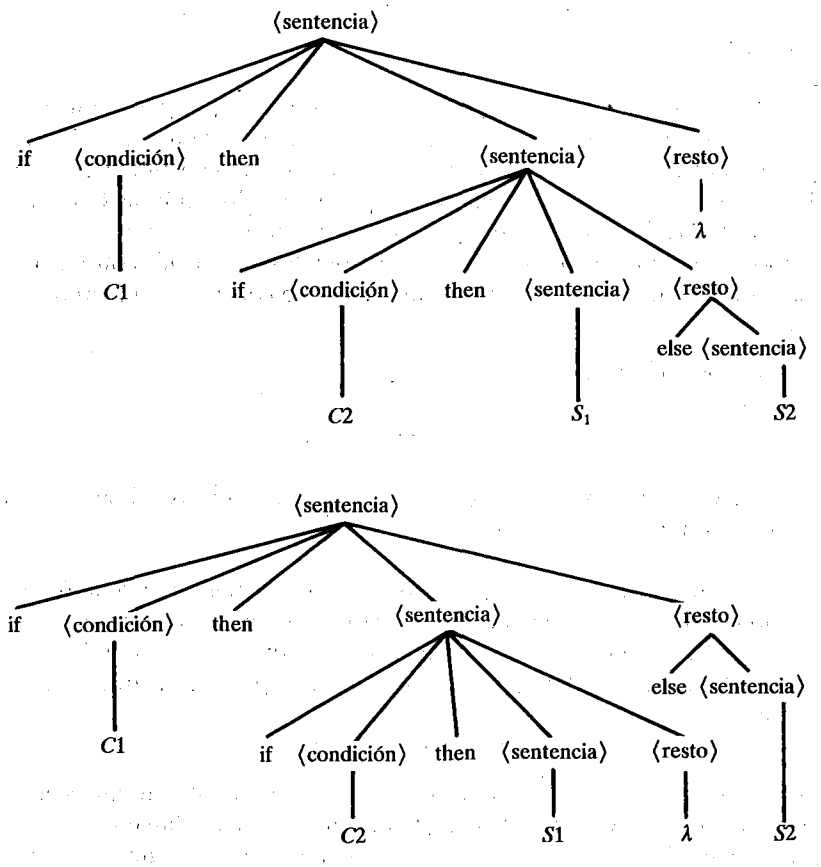


FIGURA 5.4.

```
⟨sentencia⟩ ::= ⟨s-emparejada⟩|⟨s-no-emparejada⟩
⟨s-emparejada⟩ ::= begin ⟨sentencia⟩{;⟨sentencia⟩} end|
                   ⟨identif⟩ := ⟨expresión⟩|
                   if ⟨condición⟩ then ⟨s-emparejada⟩ else
                                  ⟨s-emparejada⟩|
                   while ⟨condición⟩ do ⟨sentencia⟩|
                   read(⟨identif⟩)|write(⟨expresión⟩)
⟨s-no-emparejada⟩ ::= if ⟨condición⟩ then ⟨resto⟩
⟨resto⟩ ::= ⟨sentencia⟩|⟨s-no-emparejada⟩ else ⟨s-emparejada⟩
```

se obtiene una gramática equivalente y no ambigua en la que las derivaciones para los «if anidados» siguen el criterio anterior.

e) Si el lector consulta en algún libro sobre Pascal su definición sintáctica, observará que no aparecen las sentencias «read» y «write». La explicación es que, en realidad, no son sentencias, sino procedimientos estándar. Aquí las hemos incluido como sentencias porque para simplificar hemos prescindido de los procedimientos.

3.5. Diagramas sintácticos

Los diagramas sintácticos son construcciones gráficas que permiten presentar la misma información sintáctica que la notación BNF pero de una manera que resulta más fácil de interpretar para las personas. Un diagrama sintáctico es un grafo orientado con dos tipos de nodos: unos corresponden a los símbolos auxiliares, y se dibujan como cuadrados o rectángulos, y otros a los símbolos terminales, y se dibujan como círculos o rectángulos con ángulos redondeados. Dado un conjunto de reglas BNF, el diagrama sintáctico correspondiente se puede construir siguiendo las siguientes normas:

a) A una producción del tipo

$$A ::= \alpha_1|\alpha_2| \ldots |\alpha_n$$

le corresponde el diagrama de la figura 5.5(a), donde cada rectángulo se sustituye según las normas que siguen.

b) Si α_i tiene la forma $\beta_1\beta_2 \ldots \beta_n$, se sustituye por el diagrama de la figura 5.5(b), en el que cada rectángulo se sustituye de acuerdo con las otras normas.

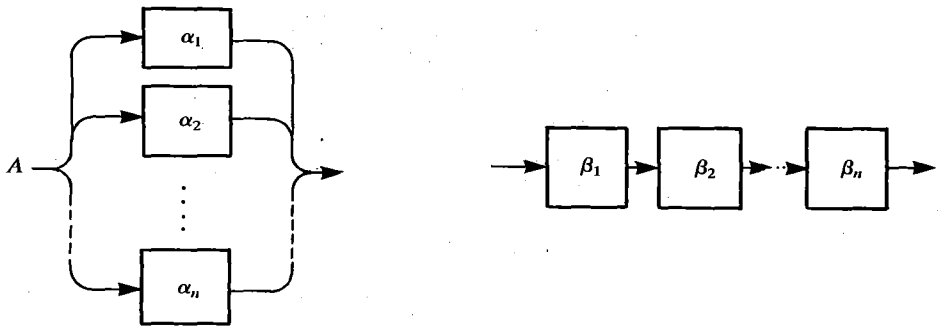
c) Si α_i (o β_i) tiene la forma $\alpha\{\beta\}$, se sustituye por la figura 5.5(c).

d) Si tiene la forma $\alpha\{\alpha\}$ (o, lo que es lo mismo, $[\alpha]$), se sustituye según indica la figura 5.5(d).

e) En el caso de $\alpha\{\beta\alpha\}$, tendríamos la figura 5.5(e).

El diagrama completo se presenta como un conjunto de subdiagramas en cada uno de los cuales pueden figurar símbolos auxiliares que se desarrollan en otros subdiagramas. La recursividad se manifiesta en el hecho de que el subdiagrama correspondiente a un símbolo auxiliar puede contener el nodo correspondiente a ese mismo símbolo, u

otro cuyo subdiagrama lo contiene. Como ejemplo, en la figura 5.6 reproducimos el diagrama sintáctico correspondiente al lenguaje definido en el apartado 3.4.

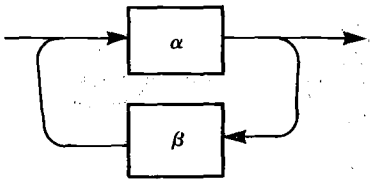


a) $A ::= \alpha_1|\alpha_2| \ldots | \alpha_n$

b) $\beta_1\beta_2 \ldots \beta_n$

c) $\alpha\{\beta\}$

d) $\alpha\{\alpha\} = [\alpha]$



e) $\alpha\{\beta\alpha\}$

FIGURA 5.5.

4. DEFINICIONES SEMÁNTICAS

4.1. Objetivos

La definición sintáctica de un lenguaje permite comprobar si un determinado programa pertenece o no al lenguaje, y, como veremos en el apartado 5.2, reconstruir el árbol de derivación de ese programa. Pero no nos dice nada sobre el efecto de la ejecución del programa. Esto pertenece al campo de la semántica. Hemos visto en los anteriores ejemplos que una primera ventaja de la presentación formal de la sintaxis de un lenguaje procede del rigor y la concisión del lenguaje matemático. El primer

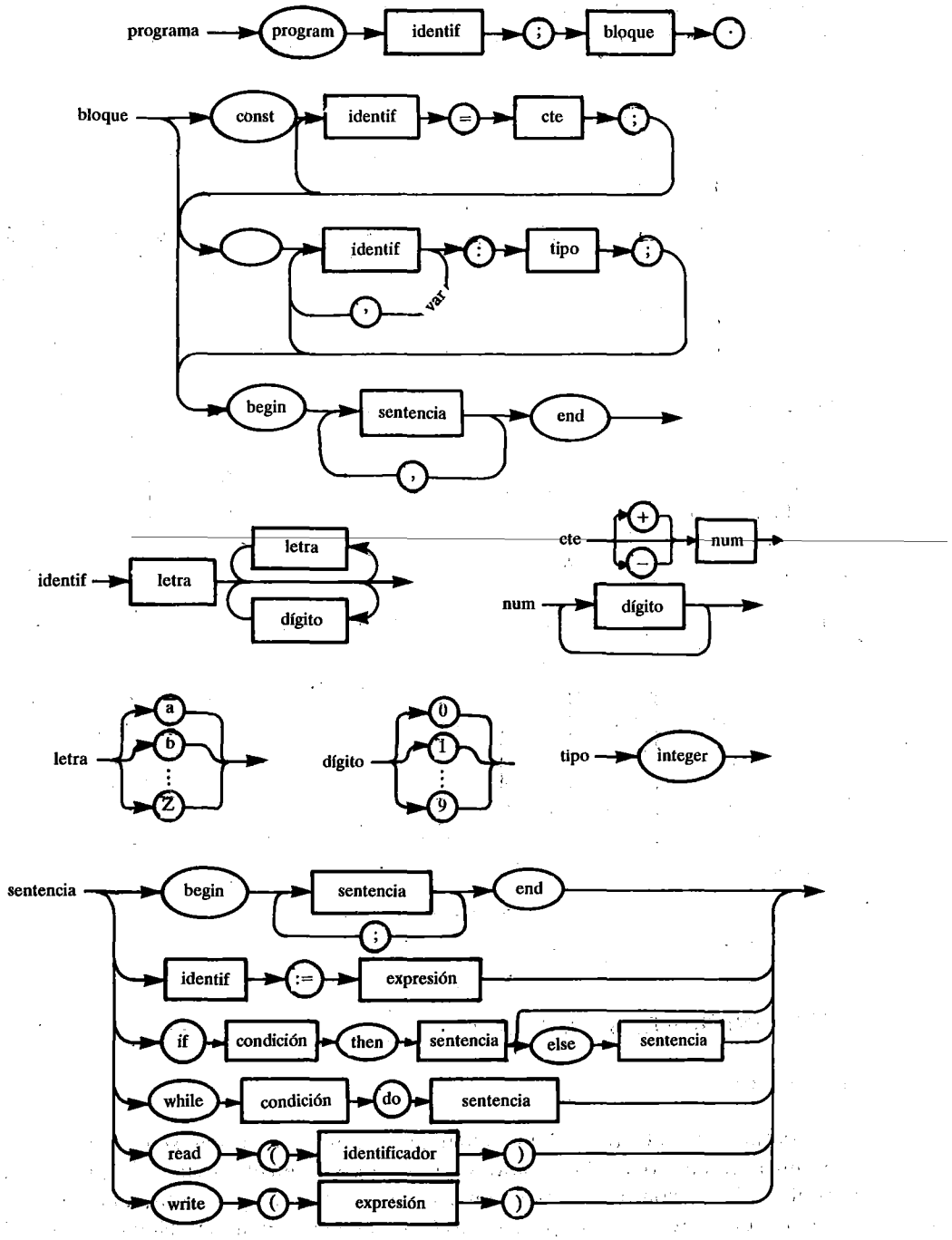

programa
program
identif
;
bloque
.
bloque
const
identif
=
cte
;
identif
:
tipo
;
,
var
begin
sentencia
end
,
identif
letra
letra
dígito
cte
+
−
num
num
dígito
letra
a
b
Z
dígito
0
1
9
tipo
integer
sentencia
begin
sentencia
end
;
identif
:=
expresión
if
condición
then
sentencia
else
sentencia
while
condición
do
sentencia
read
(
identificador
)
write
(
expresión
)

FIGURA 5.6.

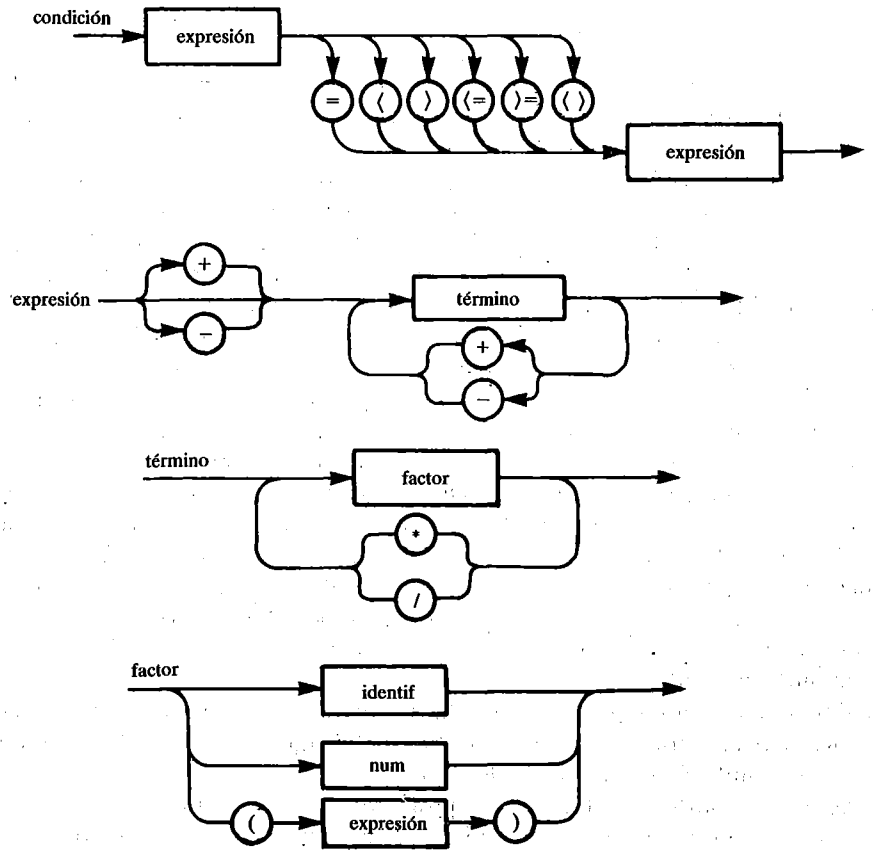


FIGURA 5.6. *(Continuación).*

objetivo de la formalización de la semántica es, asimismo, definir el lenguaje sin la ambigüedad y la pérdida de detalles con que se hace cuando, como es habitual, se utiliza el lenguaje natural para ello. Por ejemplo, en la referencia básica del lenguaje Pascal (Jensen y Wirth, 1985, p. 28) puede leerse con relación a la sentencia de asignación: «especifica que un valor a computar se asigne a una variable. El valor está especificado por una expresión». Pero esta definición es incompleta: ¿qué ocurre si la expresión no puede evaluarse, o si los tipos de la variable y del resultado de evaluar la expresión son distintos? Desde luego, en el libro citado se indican, también informalmente, las distintas posibilidades de compatibilidad entre tipos (p. 33), pero no cabe duda de que lo mejor sería expresar escueta, clara y completamente el significado de la sentencia.

Está claro que no sólo es importante que un programa sea sintácticamente correcto; es preciso que haga justamente lo que se pretende de él. Típicamente, en el proceso de desarrollo de un programa, una vez depurados los errores sintácticos se le somete a una serie de pruebas para examinar su comportamiento y se va refinando

hasta que se está «razonablemente» seguro (recuérdese la «ley de Gilb», tema «Algoritmos», capítulo 3, apartado 7) de que funciona correctamente. Este es un procedimiento rudimentario, costoso e inseguro. Una alternativa es desarrollar y aplicar métodos de prueba formal de programas. Es decir, demostrar, matemáticamente, que un programa dado hace lo que se supone que tiene que hacer. O, inversamente, sabiendo lo que tiene que hacer, demostrar constructivamente el programa. Este sería el segundo objetivo de la definición formal de la semántica: proporcionar una base rigurosa para razonar sobre los programas, y, por tanto, poder diseñar programas que, por el propio método de diseño, sean correctos*.

Un tercer objetivo está en relación con las ayudas informáticas al desarrollo de programas y los entornos de programación. La formalización de la sintaxis ha permitido diseñar herramientras para mejorar la productividad del desarrollo de software. Con la formalización de la semántica podrían conseguirse herramientas mucho más potentes. Por ejemplo, un método de diseño de programas es el basado en transformaciones: a partir de una especificación formal de lo que el programa debe hacer (escrita en un lenguaje de muy alto nivel), mediante transformaciones sucesivas con la ayuda de unas herramientas que elijan las reglas de transformación adecuadas, llegar al código de máquina final. Es como una generalización del proceso de compilación que veremos en el siguiente apartado. Pero al ser mucho más complejo, también resulta mucho más difícil diseñar las herramientas de modo que la semántica se mantenga en estas transformaciones. Una definición formal de la semántica es casi imprescindible para ello.

Finalmente, la definición semántica formal es útil en el proceso de diseño de nuevos lenguajes, para contrastar distintas posibilidades en la definición de las construcciones sintácticas básicas del lenguaje.

Como ya decíamos en el primer capítulo de este tema (apartado 3), se han propuesto varios enfoques para la definición formal de la semántica. Estos enfoques no son competitivos, sino complementarios: según el objetivo que se pretende con la formalización, resulta más adecuado un enfoque u otro. Resumimos a continuación las ideas básicas de los más conocidos.

4.2. Semántica operacional

La semántica operacional, también llamada interpretativa, se basa en una definición formal del interpretador del lenguaje, entendiendo «interpretador» en el sentido explicado al comienzo de este capítulo: una máquina que reconoce y ejecuta los programas escritos en ese lenguaje. Entonces, la semántica de cada construcción

---

* Hablar de «programa correcto» es común en el lenguaje cotidiano de la informática, y algo que todos entendemos: se trata de una propiedad que atañe tanto a la sintaxis del lenguaje (el programa es una cadena que puede derivarse en la gramática) como a la semántica (el programa cumple con las especificaciones). No obstante, podemos objetar el mismo reparo terminológico que expresábamos en el tema «Lógica» (capítulo 4, apartado 2.4) con respecto a «fórmula bien formada»: ¿acaso un «programa incorrecto» puede llamarse «programa»?

sintáctica se expresa mediante la descripción del funcionamiento de esa máquina virtual al interpretar la construcción.

Por ejemplo, para un lenguaje libre de contexto podemos definir un autómata de pila. El estado estará constituido por el estado de la parte de control y el estado de la pila, incluido el valor del puntero. Parte del estado está formada por el conjunto de los identificadores declarados en el programa y el valor que tengan. Esta parte se puede considerar como una ampliación del concepto de «interpretación» tal como se entiende en lógica: del mismo modo que la interpretación de variables proposicionales es una función que asigna a cada variable un valor del conjunto V, con $V = \{0, 1\}$ en el caso de interpretación binaria (tema «Lógica», capítulo 2, apartado 3), y cuyo dominio se puede ampliar a las sentencias, podemos ahora definir una función, q, que asigne a cada identificador declarado en el programa un valor del conjunto de valores, y cuyo dominio se puede ampliar a las expresiones. El conjunto de estados sería el conjunto de todas esas funciones:

$$Q = \{q\text{: } \{\text{ident}\} \rightarrow V\}$$

donde {ident} es el conjunto de identificadores y V es el conjunto de valores (enteros, reales, booleanos, etc.). Sobre esta función sería preciso hacer la restricción de que cada identificador sólo puede aplicarse en el subconjunto de valores determinado por su tipo.

Otra parte del estado sería la referente al estado de la pila: identificadores cuyo valor está almacenado y posición del puntero. El funcionamiento del autómata (que define la semántica del lenguaje) quedaría determinado especificando completamente la función de transición, $f(e, q)$, que depende del estado actual, q, y de la entrada, e (sentencia o dato).

Veamos algunos ejemplos con relación al lenguaje definido en el apartado 3.4. Para no extendernos en demasiados detalles, nos referiremos sólo a la primera parte del estado (por lo que las definiciones serán incompletas). Para la sentencia de asignación podríamos escribir:

$$f(\text{ident:=expr}, q_1) = q_2$$

donde, si q_1 (ident) $= v_i$ y q_1 (expr) $= v_e$, entonces

$$q_2(\text{ident}) = v_e$$
$$q_2(\text{ident}') = q_1(\text{ident}'), \forall\ \text{ident}' \neq \text{ident}$$

Esta definición de la sentencia de asignación ya dice algo que no se suele decir en la definición informal: que la sentencia no tiene «efectos laterales», o sea, que no se modifica más que el valor de «ident». No obstante, está incompleta, y no sólo por el asunto de la pila, cuyo estado, eventualmente, tendría que modificarse, sino porque no indica lo que pasa cuando ident y expr son de diferente tipo. En nuestro lenguaje esto no importa, porque sólo hay un tipo («integer»), pero en general, no es así, y entonces hay que completar la definición del lenguaje con otra función, «tipo», que se aplica sobre los identificadores y las expresiones y que toma valores en la interpreta-

ción de sentencias de declaración y de aquellas en las que hay evaluación de expresiones. También añadiríamos un estado especial de error, y completaríamos la definición de la sentencia de asignación formalizando la idea de que se va al estado de error si los tipos de ident y de expr son incompatibles. Si convenimos que la sentencia sólo es válida en el caso de que la evaluación de expr pertenezca al mismo tipo que ident, tendríamos:

$$f(\text{ident:=expr}, q_1) = q_2 \text{ si } t(\text{ident}) \text{ definida y } t(\text{ident}) = t(\text{expr})$$
$$= \text{error si } t(\text{ident}) \text{ indefinida o } t(\text{ident}) \neq t(\text{expr})$$

donde t es la función «tipo» a la que nos referíamos más arriba. Inicialmente, está indefinida para todo identificador. La función de transición le da valores para los identificadores al intepretar las sentencias de declaración y para las expresiones al interpretar sentencias donde intervengan tales expresiones.

Para la construcción que consiste en concatenar sentencias basta indicar que con la primera se va a un estado intermedio:

$$f(\text{sent1; sent2}, q) = f(\text{sent2}, f(\text{sent1}, q))$$

La semántica de la sentencia de iteración quedaría definida por la función de transición:

$$f(\text{while cond do sent}, q) = f(\text{sent}, q) \text{ si } (q(\text{cond}) = \text{verd})$$
$$= q \text{ si } (q(\text{cond}) = \text{falso})$$
$$= \text{error en otro caso (p. ej., si } q(\text{cond}) \text{ es un número)}$$

Insistimos en que la definición completa exigiría una descripción también completa del autómata, incluyendo el estado de la pila. El principal inconveniente de la semántica operacional radica, precisamente, en su excesiva dependencia de una estructura de máquina (aunque ésta sea abstracta). La tendencia a buscar modelos más matemáticos y menos ligados a estructuras de máquinas conduce a la semántica denotacional.

4.3. Semántica denotacional

Se llaman *denotaciones* a entidades matemáticas que modelan los significados de las construcciones sintácticas del lenguaje. Así, escribiremos S[ident:=expr] para representar el significado de la sentencia de asignación. El principal problema para definir la semántica denotacional de un lenguaje determinado es la elección de las entidades matemáticas adecuadas. Veamos resumidamente cómo podríamos proceder en el caso de nuestro lenguaje del apartado 3.4.

Definimos primero los *dominios sintácticos,* que son los conjuntos de identificado-

res, I, de expresiones, X, y de sentencias o comandos, C. Normalmente, lo que en la definición sintáctica del apartado 3.4 llamábamos «condición» (que interviene en las sentencias «if» y «while») se suele considerar como perteneciente a la misma categoría sintáctica que «expresión», es decir, se consideran pertenecientes al mismo dominio las expresiones aritméticas y las booleanas. Definiremos a continuación los *dominios semánticos,* que son los conjuntos donde van a tomar valores las denotaciones.

El primer dominio semántico es V, el conjunto posible de valores. Así, $S[I]$ será una función que aplica cada identificador en su valor. El nuestro lenguaje sólo teníamos un tipo de datos, «integer», pero normalmente, V será la unión de los enteros, booleanos, caracteres, etc. Suponemos que V contiene también el elemento «indefinido».

Otro dominio semántico es el *estado.* Aunque no se haga referencia a ninguna máquina concreta, ni real ni virtual, es preciso, de todas formas, representar de algún modo el efecto de la ejecución. Podemos partir de una definición de estado similar a la de la semántica operacional (una función de I en V). Pero al objeto de definir también la semántica de las sentencias «read» y «write» vamos a incluir el estado de la entrada y de la salida:

$$E = S = V^*$$

Esto indica que E (y S) es el dominio de todas las posibles cadenas (incluyendo la vacía) de elementos de V. Entonces, definiremos el *dominio estado* como

$$Q = M \times E \times S$$

donde M es el *dominio memoria,* formado por todas las funciones de I en V:

$$M = \{f : I \rightarrow V\}^*$$

o, escrito más abreviadamente,

$$M = I \rightarrow V$$

Es decir, el dominio estado, Q, es el conjunto de todas las triplas $q = (m, e, s)$, donde $m \in M$ es una función $I \rightarrow V$ y $e \in E$ y $s \in S$ son cadenas de V a la entrada y la salida.

Las denotaciones de los dominios sintácticos se definen mediante *funciones semánticas,* funciones que aplican dominios sintácticos en dominios semánticos. Así, las denotaciones de expresiones (consideramos que los identificadores son casos particulares de expresiones) son funciones de estados en valores (o sea, el significado de una expresión es que toma un valor que depende del estado):

$$S[x]: Q \rightarrow V \cup \{\text{error}\}$$

---

* En todo este apartado, el símbolo «→» representa la aplicación de un conjunto en otro.

donde $x \in X$, y se ha considerado que la evaluación de una expresión puede dar lugar a un error (por ejemplo, si se intenta sumar un entero con un booleano). Las denotaciones de sentencias o comandos son funciones de estados en estados (el significado de una sentencia es que cambia el estado, pudiendo también dar un error):

$$S[c]: Q \rightarrow Q \cup \{\text{error}\}$$

donde $c \in C$.

Utilizamos paréntesis cuadrados para las funciones semánticas sólo para mejorar la legibilidad. Para representar, por ejemplo, la función S de la sentencia $A:=B$ sobre un estado (m, e, s) escribiremos:

$$S[A:=B](m, e, s) = \ldots$$

Para un lenguaje determinado, se llaman *cláusulas semánticas* a las definiciones concretas de las anteriores funciones semánticas para todas y cada una de las construcciones sintácticas del lenguaje (dadas por la definición sintáctica). Veamos cómo podrían ser algunas de las cláusulas semánticas para nuestro lenguaje.

Empezando por las expresiones, la construcción más sencilla es ⟨expr⟩ ::= ⟨ident⟩. La correspondiente cláusula semántica es:

$$\begin{aligned} S[i](m, e, s) &= \text{error si } m(i) = \text{indefinido} \\ &= m(i) \text{ si no} \end{aligned}$$

Es decir, el significado del identificador i es el valor que tenga definido en M, $m(i)$, y si está indefinido se produce un error.

Para ⟨expr⟩ ::= ⟨expr⟩ + ⟨expr⟩ definimos la cláusula:

$$\begin{aligned} S[e1 + e2]q &= S[e1]q + S[e2]q \text{ si num}(S[e1]) \text{ y num}(S[e2]) \\ &= \text{error si no} \end{aligned}$$

donde num es un predicado que se aplica sobre el conjunto V y que es verdadero si el elemento es un número (entero o real) y falso si no. Tal como se definió la sintaxis de nuestro lenguaje esta condición no sería necesaria, al ser todos los valores enteros.

Veamos algunas cláusulas semánticas para sentencias. Para la de asignación tenemos:

$$\begin{aligned} S[i: = x](m, e, s) &= (m[v/i], e, s) \text{ si } S[x]q = v \text{ y compat}(S[i], v) \\ &= \text{error si no} \end{aligned}$$

donde compat es un predicado binario sobre V que es verdadero si los tipos de sus argumentos son compatibles y falso si no, y $m[v/i]$ es la función

$$\begin{aligned} m[v/i](j) &= v \text{ si } i = j \\ &= m(j) \text{ si no} \end{aligned}$$

que da a cada identificador el valor que tenía salvo a i, que le da v, resultado de la evaluación de x; si ésta da error (o si los tipos son incompatibles), también lo da $i{:=}x$.

Para la concatenación de sentencias,

$$S[c1; c2]q = \text{error si } S[c1]q = \text{error}$$
$$S[c2](S[c1]q) \text{ si no}$$

Para la sentencia «while»,

$$S[\text{while } x \text{ do } c]q = S[\text{while } x \text{ do } c]q' \text{ si}$$
$$S[x] = v \text{ y } v = \text{verd y } S[c] = q'$$
$$= q \text{ si } S[x] = v \text{ y } v = \text{falso}$$
$$= \text{error en otro caso}$$

En «otro caso» está comprendido el que $S[x]$ o $S[c]$ den error o que v no sea ni «verd» ni «falso». Obsérvese que la definición es recursiva.

Veamos, como último ejemplo, el caso de la sentencia de salida:

$$S[\text{write}(x)](m, e, s) = (m, e, s.v) \text{ si } S[x] = v$$
$$= \text{error si no}$$

donde «$s.v$» es la cadena resultante de poner v detrás de s.

La semántica denotacional, de la que aquí sólo hemos presentado la idea básica, es una herramienta muy útil para el diseño de nuevos lenguajes. Sabiendo lo que se quiere del lenguaje, se diseñan denotaciones alternativas, se comparan, y cuando se está satisfecho con sus propiedades se les da nombre y se define la sintaxis. Para la verificación de programas escritos en un lenguaje ya definido, o para el desarrollo de métodos formales de diseño de los programas, resulta más útil la semántica axiomática.

4.4. Semántica axiomática

En la semántica operacional el estado es un concepto básico, ligado al concepto de máquina. En la semántica denotacional abstraemos el concepto de máquina, y hablamos de estado como un dominio semántico sobre el que se aplican las funciones semánticas, definidas sobre los dominios sintácticos. En la semántica axiomática sigue siendo básico el concepto de *estado**, pero no se trata de establecer denotaciones de las construcciones sintácticas, sino de encontrar relaciones lógicas entre estados iniciales y finales que sirvan para expresar el significado de los programas, o, lo que es lo mismo, para decir formalmente lo que hacen. Como hablamos de propiedades de los estados y de relaciones entre estados, la base teórica para formalizar aquí la semántica es el cálculo de predicados.

* Si se nos presenta siempre como básico el concepto de estado es porque nos estamos refiriendo a lenguajes imperativos.

Sabemos que un predicado (tema «Lógica», capítulo 4, apartado 1.3) es la formalización de una propiedad (predicado monádico) o de una relación (predicado poliádico) entre elementos de un «universo del discurso». Nuestro universo del discurso es el conjunto V de valores posibles que pueden tomar las variables, que, en nuestro caso, son los identificadores. Como el conjunto de todos los estados posibles es el conjunto de todas las funciones de identificadores en valores, un predicado aplicado a unos identificadores representa a un conjunto de estados. Por ejemplo, si i es un identificador y $M(i)$ es el predicado «i es mayor o igual que cero», $M(i)$ es falso para todos los estados en los que $i < 0$ y verdadero para todos los estados en los que $i \geq 0$. Por tanto, $M(i)$, o, más nemotécnicamente, «$i \geq 0$», representa el conjunto de estados tal que $i \geq 0$. Predicados de este tipo son *predicados sobre el estado.*

Lo que interesa es establecer *predicados sobre programas* y poder demostrar que son verdaderos. Si Q y R son predicados sobre estados y P es un programa, la notación

$$\{Q\}P\{R\}$$

significa: «si la ejecución de P comienza en un estado que satisface Q, entonces termina en un tiempo finito en un estado que satisface R». Esto, realmente, es un predicado sobre P: es verdadero o falso según sea P. Lo que pretende la semántica axiomática es encontrar métodos para, a partir de Q y R (*especificaciones*), y dado un programa P, demostrar que el predicado $\{Q\}P\{R\}$ es verdadero.

Q es la *precondición,* o *aserción de entrada.* R es la *postcondición* o *aserción de salida.* Y como P está compuesto por una secuencia de sentencias, podemos considerar que cada sentencia va transformando un predicado en otro a partir de Q. Si P satisface al predicado $\{Q\}P\{R\}$, entonces el resultado de la última transformación será R. Tenemos así que definir, para cada sentencia y combinaciones de sentencias, un *transformador* de predicados sobre el estado.

Dados una sentencia S y un predicado R que representa el resultado de ejecutar S, se define otro predicado, pmd(S, R), que representa al conjunto de *todos* los estados tales la ejecución de S a partir de uno cualquiera de ellos termina en un tiempo finito es un estado que satisface R. Entonces, escribir $\{Q\}S\{R\}$ es lo mismo que escribir $Q \rightarrow$ pmd (S, R), donde «$\rightarrow$» es el condicional de la lógica (si... entonces). pmd (S, R) es la *precondición más débil* de S con respecto a R.

Por ejemplo, si S es $i := i + 1$ y $R = i \leq 1$,

$$\text{pmd } (i:=i + 1, i \leq 1) = i \leq 0$$

En efecto, para que tras la ejecución de $i:=i + 1$ resulte un valor de $i \leq 1$ es necesario y suficiente que, antes de la ejecución, $i \leq 0$. La postcondición se cumple con cualquier precondición Q tal que $i \leq 0$ (como $Q = i \leq -1$, $Q = i \leq -2$, etc.), pero la más restrictiva de ellas (o *más débil*) es $i \leq 0$.

Obsérvese que pmd, además de ser un *transformador de predicados,* es también un predicado (puesto que su resultado es una aserción, verdadera o falsa, que representa al conjunto de estados en los que es verdadera) con dos argumentos: una sentencia, S,

y un predicado, R. Estamos, pues, en lógica de predicados de segundo orden (tema «Lógica», capítulo 5, apartado 2.1).

La primera tarea, dada la definición sintáctica el lenguaje, es definir pmd para cada una de las posibles construcciones. Así, para la sentencia de asignación podemos definir:

$$\text{pmd } (i{:=}e, R) = Rs$$

donde $s = \{e/i\}$ es la sustitución en R de e por i tal como fue definida en el tema «Lógica», capítulo 4, apartado 4.3. Esto quiere decir que la postcondición, R, será verdadera si y *sólo si* R con el valor de i sustituido por e es verdadera. O, lo que es lo mismo, que el conjunto de estados tras la ejecución de la sentencia es el inicial con el valor de i sustituido por el de e.

Para la sentencia condicional «ifthenelse»,

$$\begin{aligned}\text{pmd (if } c \text{ then } S1 \text{ else } S2, R) = &\ \text{pmd } (S1, R) \text{ si } c \text{ verdadera} \\ &= \text{pmd } (S2, R) \text{ si } c \text{ falsa}\end{aligned}$$

Para la concatenación de sentencias,

$$\text{pmd } (S1; S2, R) = \text{pmd } (S2, \text{pmd } (S1, R))$$

Para las sentencias iterativas (como «while») y de entrada y salida se hace precisa una elaboración más detallada en la que no vamos a entrar, y para la que, como siempre, remitimos a la bibliografía comentada en el apartado 7. Lo importante a recordar es que este enfoque de la semántica permite verificar si un programa P cumple o no unas especificaciones de entrada (Q) y salida (R), viendo si se satisface $\{Q\}P\{R\}$, y, yendo un poco más lejos, permite desarrollar reglas para escribir P a partir de Q y R.

5. PROCESADORES DE LENGUAJES

5.1. Ensambladores

5.1.1. *Traducción, carga y ejecución*

Antes de entrar en el estudio de la estructura de un ensamblador conviene que nos detengamos un momento a ver su función en un contexto global, describiendo a grandes rasgos los procesos que tienen lugar desde la traducción hasta la ejecución de un programa.

Por razones ya explicadas tanto en este tema como en el de «Algoritmos», el desarrollo de un programa suele hacerse descomponiéndolo en módulos que se prueban independientemente. Por ello, la mayoría de los ensambladores no generan directamente un programa en lenguaje de máquina, sino lo que se llama un *módulo objeto*. Este módulo sólo difiere del programa final en lenguaje de máquina en que las direcciones simbólicas no se han sustituido aún por las direcciones absolutas de

memoria, sino que figuran en una tabla junto con sus desplazamientos con relación al comienzo del programa (como si éste fuera a cargarse a partir de la dirección 0). Además, puede haber direcciones simbólicas, conocidas como *referencias externas,* que corresponden a llamadas a otros módulos o subprogramas que se ensamblan aparte o que son rutinas de utilidad ya incluidas en el sistema operativo, como las de comunicaciones con los periféricos. Cuando se han ensamblado todos los módulos, un programa llamado *montador de enlaces* («linker») «resuelve» estas referencias externas y genera un *módulo de carga.* Finalmente, otro programa, el *cargador reubicador* («relocating loader») asigna las direcciones absolutas e introduce en la memoria el programa en lenguaje de máquina listo para ser ejecutado. La figura 5.7 ilustra esta secuencia de procesos. Los rectángulos representan programas que se ejecutan, y los bloques ovalados datos o resultados de esos programas. Como se indica en la figura,

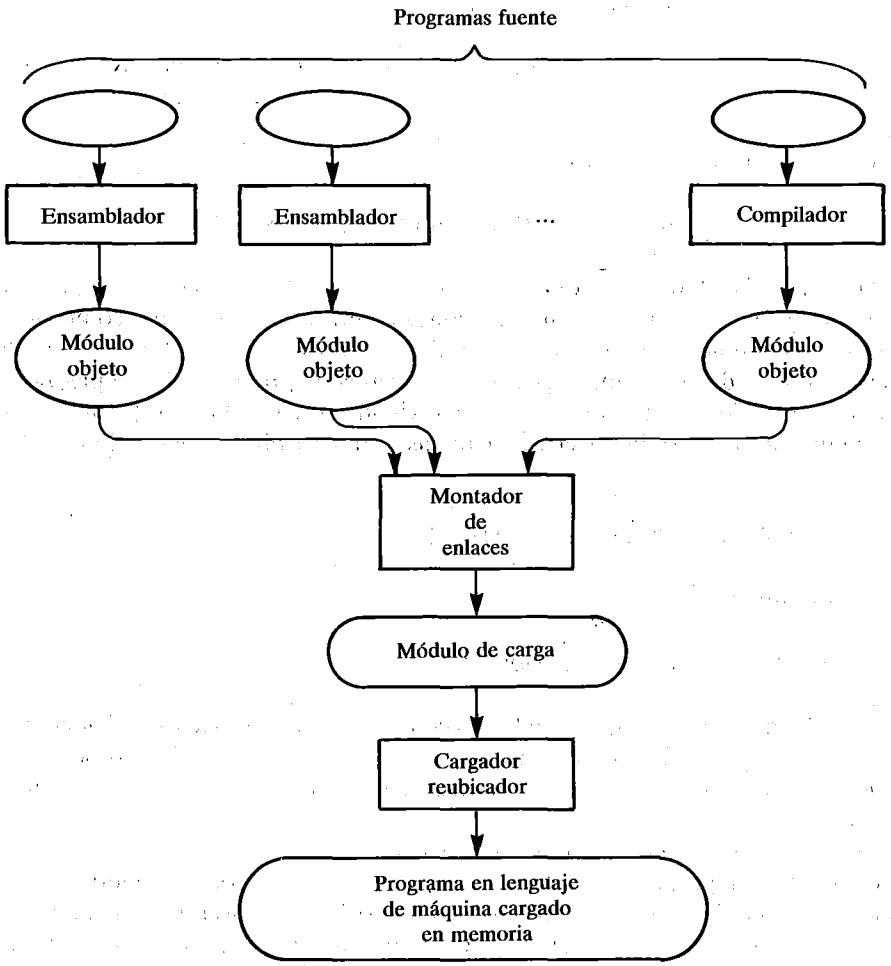


FIGURA 5.7.

los «módulos objeto» pueden haber sido generados por un ensamblador o por un compilador.

Por limitaciones de capacidad, normalmente no es posible que coexistan en la memoria central todos los programas que intervienen: programas fuente, módulos objeto, ensamblador, compilador, montador y cargador. Por esta razón, durante el ensamblaje (o la compilación) de cada módulo sólo está presente, además de una parte del sistema operativo llamado *residente* (que incluye al cargador), el ensamblador (o el compilador). Este lee las instrucciones del programa fuente, una detrás de otra, de un periférico (teclado, tarjetas, disco...), y deposita el módulo objeto en algún medio de almacenamiento auxiliar (disco, cinta, ...). Cuando todos los módulos objeto están disponibles, se carga el montador de enlaces, que genera el programa reubicable. La figura 5.8 insiste en esta secuencia de procesos, suponiendo que el almacenamiento auxiliar es sobre disco. No debe interpretarse erróneamente el hecho de que en la figura aparezca cinco veces el símbolo del disco: los diferentes programas pueden estar en varios discos o en uno solo, en distintas zonas o ficheros.

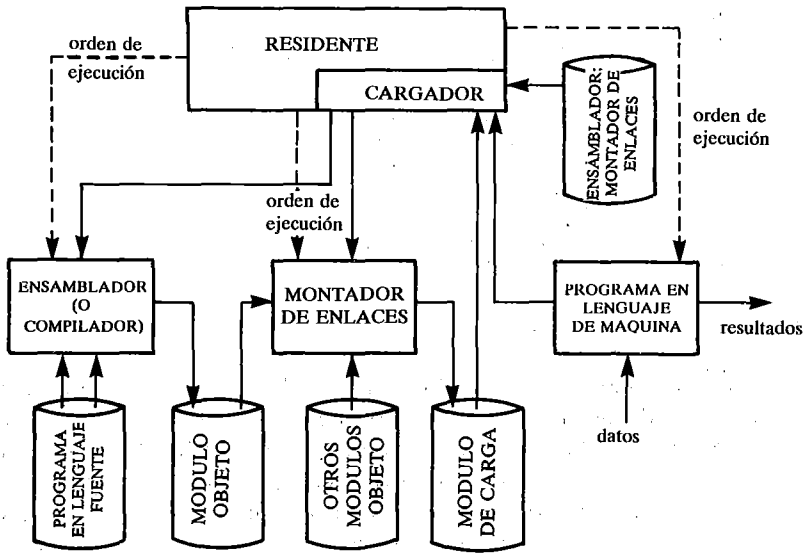


FIGURA 5.8.

5.1.2. *Ensambladores de un paso y de dos*

La mayoría de los ensambladores son de dos pasos. Esto quiere decir que leen dos veces el programa fuente. El motivo es que las instrucciones pueden hacer referencia a etiquetas que aparecen más adelante. Por ejemplo, consideremos el siguiente

segmento de un programa escrito en el lenguaje ensamblador que hemos descrito en los apartados 2.2.2 y 3.3:

```
            SAI         PRINCIP
UNO         CEN         1
N           MEM         1
A           MEM         100
PRINCIP     SAS         CALCULO
            ...
            ...
```

El ensamblador ha de generar un módulo objeto en el que la primera instrucción tiene que llevar como dirección el número que corresponda a la dirección de la etiqueta PRINCIP relativa al comienzo del programa. Es decir, a «SAI PRINCIP» le corresponde la dirección 0, a «UNO CEN 1» la 1, a «N MEM 1» la 2, y como después se reservan 100 direcciones, a «PRINCIP SAS CALCULO» le corresponderá la dirección 103. Pero cuando el ensamblador lee la primera instrucción aún no puede saber qué dirección le va a corresponder a «PRINCIP». Por ello, en una primera lectura del programa fuente *(primer paso)* el ensamblador construye en la memoria una tabla de etiquetas, que en el caso anterior quedaría así:

Etiqueta	*Dirección*
UNO	1
N	2
A	3
PRINCIP	103
...	...
...	...

Cuando ha leído todo el programa y construido la tabla de etiquetas, entra en el *segundo paso,* leyendo de nuevo desde la primera línea. Al encontrarse con «SAI PRINCIP», consulta la tabla de etiquetas, donde ve que a «PRINCIP» le corresponde la dirección relativa 103, con lo que genera ya el código objeto correspondiente a esta instrucción. Desde luego, puede haber etiquetas que no aparezcan en la tabla, que corresponderán a referencias externas, de las cuales se ocupará el montador de enlaces.

Ahora bien, en ciertos casos no conviene que el ensamblador tenga que leer dos veces el programa fuente. Por ejemplo, en pequeños sistemas en los que no se dispone de almacenamiento auxiliar: si el programa fuente no cabe en la memoria, sería preciso que el usuario introdujese manualmente dos veces ese programa fuente. Para tales casos, se puede tener un ensamblador de un solo paso. Basta con ir formando una tabla adicional con los nombres no referenciados aún en la tabla de etiquetas y la dirección relativa de las instrucciones en las que aparecen; conforme se va llenando la tabla de etiquetas, se retrocede para poder obtener el código de las instrucciones pendientes de traducir.

5.1.3. *Un ensamblador de dos pasos*

Vamos a describir, a nivel de organigramas, un posible ensamblador para el lenguaje ensamblador definido en los apartados 2.2.2 y 3.3. Como el proceso es bastante sencillo, no nos preocuparemos por hacer un uso sistemático de la definición formal del lenguaje dada en el apartado 3.3. Los organigramas no son detallados ni completos, ni siquiera estructurados*, pero esperamos que permitirán al lector hacerse una idea concreta del proceso de ensamblaje.

La tarea del primer paso consiste, esencialmente, en formar la tabla de etiquetas, llevando, al mismo tiempo, algunas tareas de detección de errores sintácticos, como etiquetas erróneamente formadas o duplicadas. Para formar la tabla se utiliza una variable llamada *contador de ensamblaje,* que se inicializa en cero y se incrementa, normalmente en una unidad, al leer cada instrucción o pseudoinstrucción. En el caso de la pseudoinstrucción «MEM», se incrementa en el número de palabras a reservar. Así, al leer una línea, primero se detecta si hay algún carácter distinto de «bl» y de «*» en la primera columna; si es así, es que hay una etiqueta, cuyos caracteres se leen y se introducen en la tabla de etiquetas junto con el contador de ensamblaje. Se analiza luego el código de operación, y si éste es «MEM», se mira en cuánto hay que incrementar el contador de ensamblaje, tras lo cual se lee la siguiente línea, y así sucesivamente hasta encontrar la pseudo-instrucción «FIN».

En la figura 5.9 puede verse el organigrama para este primer paso. CONT_ENS es una variable entera que corresponde al contador de ensamblaje. LIN es una variable estructurada como tira de caracteres. Si, por ejemplo, el número de caracteres de cada línea es 80, la declaración de LIN en un programa en Pascal podría ser:

```
var LIN: packed array [1..80] of char.
```

Se supone que el programa fuente está almacenado en un fichero de un disco, de donde se van leyendo sucesivamente las líneas, que se introducen en la variable LIN.

En el organigrama aparecen llamadas a tres procedimientos, «Explorador», «Busca_ETI» y «Act_ETI», cuyas funciones son las siguientes:

«Explorador» recorre caracteres en la variable LIN saltando los espacios en blanco hasta encontrar el primero no blanco. Sigue entonces explorando hasta que de nuevo aparece un blanco. La secuencia de caracteres no blancos la pone en la variable CARAC. Si se trata de un número, lo convierte a binario, lo deja en la variable NUM y pone la variable booleana ES_NUM con el valor «true». Si es una etiqueta correctamente formada pone a «true» la variable booleana ES_ETI. Y si es un código de operación pone a «true» la variable booleana ES_CO y devuelve el código binario (4 bits) en la variable CO. Alguna de estas informaciones, concretamente NUM y CO, no son necesarias en el primer paso, pero sí en el segundo. Aunque no se ha indicado en el organigrama, cada vez que el programa lee una línea nueva y llama a

* Hemos decidido presentarlos así en aras de una mayor concisión. Comprendido el proceso, el lector puede tratar de presentarlo de manera estructurada, siguiendo los principios explicados en el tema «Algoritmos».

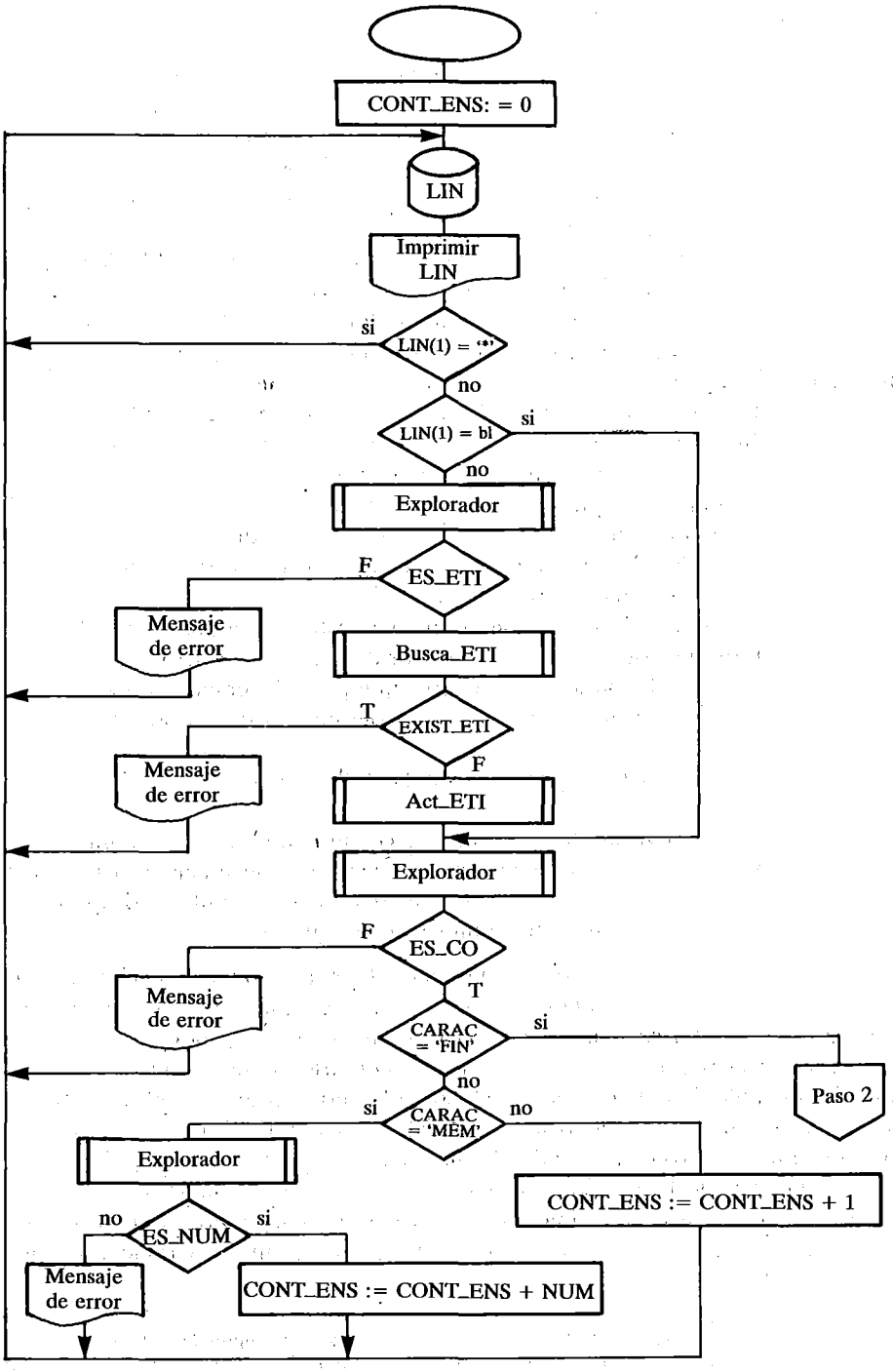

CONT_ENS: = 0
LIN
Imprimir LIN
si
LIN(1) = '*'
no
LIN(1) = bl
si
no
Explorador
F
ES_ETI
Mensaje de error
Busca_ETI
T
EXIST_ETI
F
Mensaje de error
Act_ETI
Explorador
F
ES_CO
T
Mensaje de error
CARAC = 'FIN'
si
Paso 2
no
si
CARAC = 'MEM'
no
Explorador
CONT_ENS := CONT_ENS + 1
no
ES_NUM
si
Mensaje de error
CONT_ENS := CONT_ENS + NUM

FIGURA 5.9.

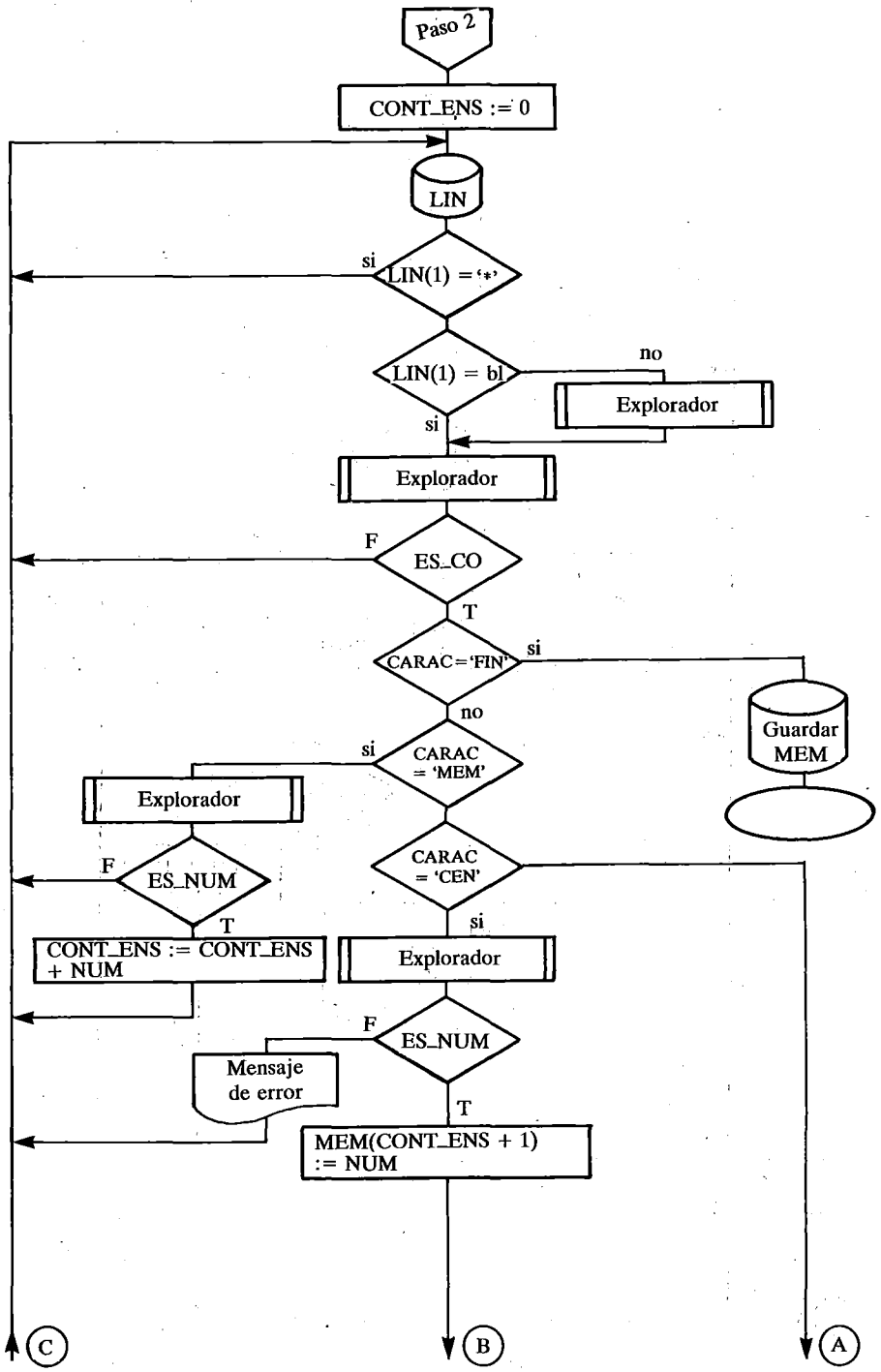

Paso 2
CONT_ENS := 0
LIN
si
LIN(1) = '*'
LIN(1) = bl
no
Explorador
si
Explorador
F
ES_CO
T
CARAC='FIN'
si
Guardar MEM
no
si
CARAC = 'MEM'
Explorador
F
ES_NUM
T
CONT_ENS := CONT_ENS + NUM
CARAC = 'CEN'
si
Explorador
F
ES_NUM
Mensaje de error
T
MEM(CONT_ENS + 1) := NUM
C
B
A

FIGURA 5.10.

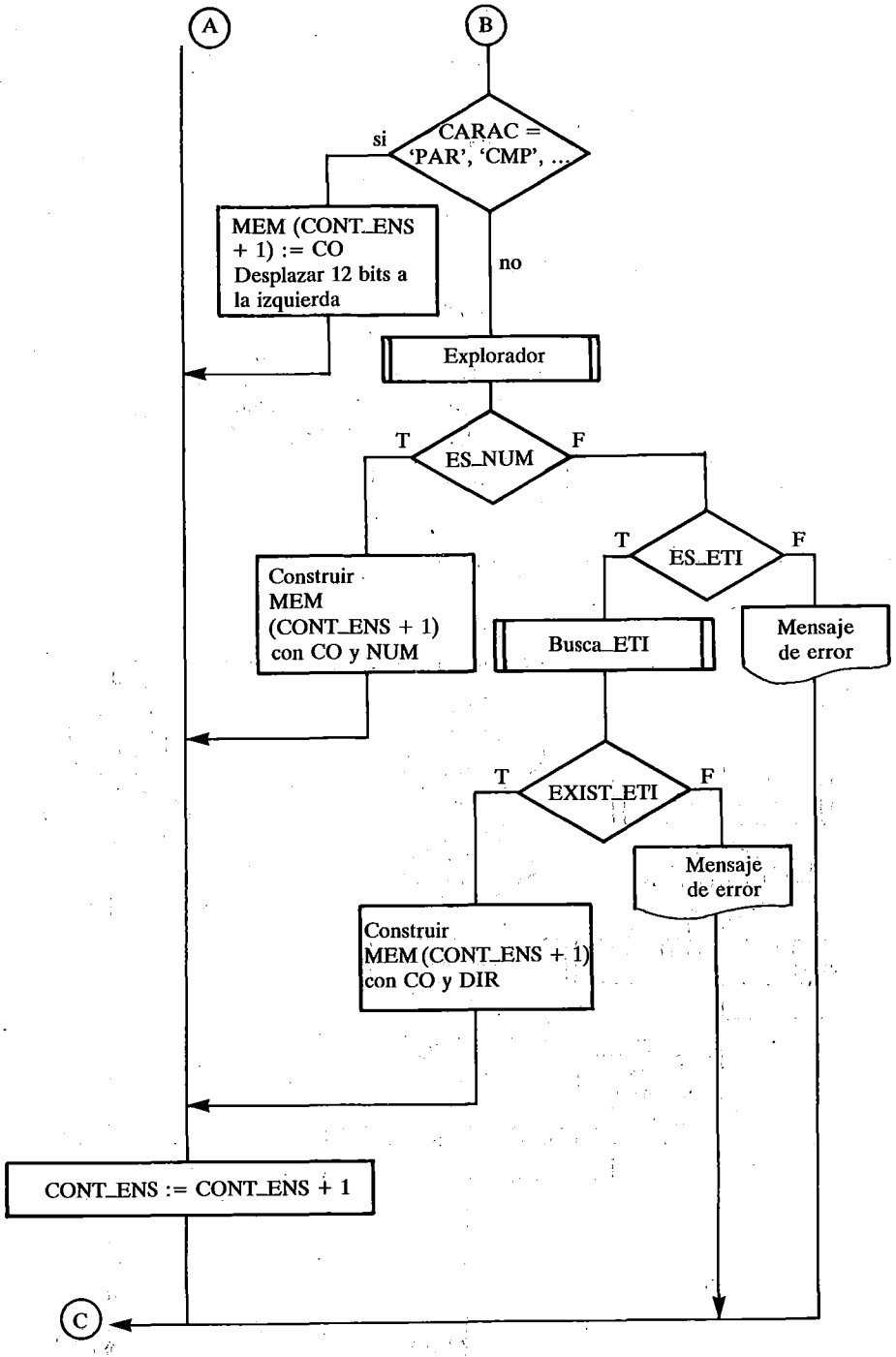


FIGURA 5.10. *(Continuación).*

«Explorador» tiene que inicializarlo para que empiece a explorar desde el primer carácter de la línea; en llamadas sucesivas dentro del análisis de la misma línea, el explorador seguirá desde el carácter en que se detuvo (el primer blanco siguiente a una cadena de caracteres). En el apartado 5.2.2 explicaremos, en un contexto más general, una técnica para diseñar programas exploradores.

«Busca_ETI» es un subprograma que consulta la tabla de etiquetas: compara CARAC con las etiquetas ya introducidas. Si encuentra coincidencia, pone a «true» la variable booleana EXIST_ETI y devuelve en la variable entera DIR la dirección relativa correspondiente (esto último, para el segundo paso). «Act_ETI» construye y actualiza la tabla de etiquetas: cada vez que se le llama introduce CARAC y CONT_ENS en la tabla.

El segundo paso puede verse en la figura 5.10. Los códigos objeto de las distintas instrucciones y pseudoinstrucciones se van guardando sucesivamente en MEM(CONT_ENS). En el organigrama aparece un mensaje de error en el caso de que una dirección simbólica no aparezca en la tabla de etiquetas. Esto es porque suponemos que nuestro ensamblador no permite el uso de referencias externas. Para permitirlo bastaría con que, en lugar de dar un error, se fuese construyendo otra tabla con la etiqueta no encontrada y la dirección relativa de la instrucción en la que aparece, y esta tabla se guardaría al final en el disco y serviría de entrada para el montador de enlaces.

Cuando se identifica un código de operación, se sintetiza la instrucción en lenguaje de máquina superponiendo el código binario en los cuatro primeros bits y a continuación la dirección, salvo si la instrucción no hace referencia a memoria (PAR, CMP, etc.), en cuyo caso basta con poner el código de operación.

El ensamblador para un ordenador real, con diversos registros y modos de direccionamiento, instrucciones de longitud variable, etc., puede ser algo más complicado en sus detalles, pero seguirá teniendo, en esencia, la misma estructura que hemos expuesto para este modelo sencillo.

5.2. Compiladores

5.2.1. *Fases de la compilación*

En la ejecución de un programa compilador podemos distinguir dos etapas: la de análisis, en la que se reconoce el programa fuente y se le traduce a un código o lenguaje intermedio, y la de síntesis, en la que se genera el módulo objeto. En cada etapa tienen lugar varias *fases,* como indica la figura 5.11.

El número de *pasos* del compilador (lecturas sucesivas del programa fuente o de alguna versión intermedia) es muy variable y depende de diversos factores condicionantes del diseño. Si la capacidad de memoria central es reducida, interesa realizar el proceso en muchos pasos, con resultados intermedios en memoria auxiliar. En el caso extremo, que mencionábamos al hablar de los ensambladores, de que no se disponga de memoria auxiliar, es necesario realizar todo el proceso en un solo paso.

En este apartado describiremos a grandes rasgos las funciones de cada fase, y en

los siguientes concretaremos viendo cómo se puede aplicar la definición formal de nuestro «minilenguaje» del apartado 3.4 para el diseño de las dos primeras.

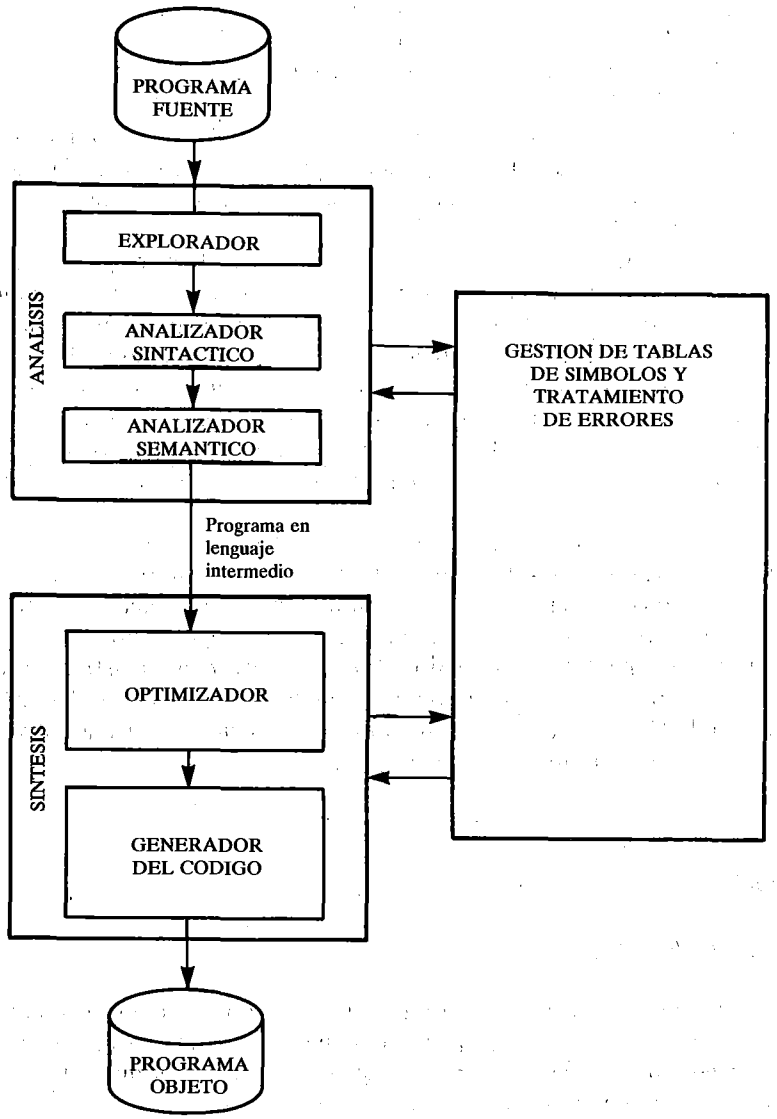


FIGURA 5.11.

La primera fase del análisis es el *análisis léxico*. El analizador léxico, que aquí llamaremos *explorador**, tiene como función principal leer los caracteres del progra-

* En otros libros se restringe el término «explorador» a un subprograma del analizador léxico que realiza las funciones más sencillas, como leer caracteres y eliminar espacios en blanco y cambios de línea.

ma fuente e identificar las *categorías sintácticas* («tokens») del lenguaje. Son categorías sintácticas las palabras clave (begin, if, ...), los identificadores, las constantes, etc. Recuérdese que en nuestro ejemplo de ensamblador del apartado anterior también teníamos un explorador que reconocía las etiquetas, los códigos de operación y los números. Además de esta labor de reconocimiento, el explorador va creando una *tabla de símbolos* en la que se registran los nombres y los atributos de los identificadores (tipo de variable, número y tipos de los argumentos en el caso de que sea un nombre de procedimiento, etc.).

El *analizador sintáctico* es, formalmente, un reconocedor del lenguaje. Como ya hemos dicho, los lenguajes de programación se suelen formalizar con gramáticas libres de contexto, y como sabemos del capítulo 4 (apartado 4), para reconocer lenguajes libres de contexto se precisan autómatas de pila. Y en efecto, durante la fase de análisis sintáctico se construyen estructuras de pila en la memoria central. Pero el analizador sintáctico no sólo reconoce si el programa es correcto o no, también reconstruye el árbol de derivación. Para su diseño resulta de gran utilidad disponer de la definición sintáctica formal.

Estas dos fases, como las siguientes, son fases lógicas que no necesariamente tienen que ejecutarse una tras otra. Así, el explorador es normalmente un subprograma llamado por el analizador sintáctico, como indica la figura 5.12.

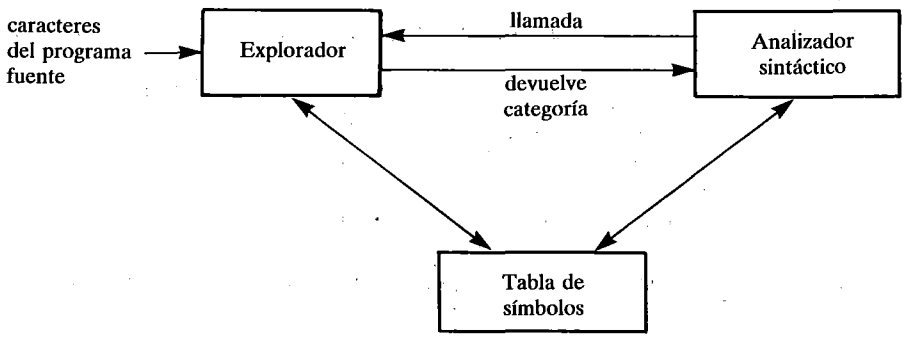


FIGURA 5.12

Se puede pensar que, puesto que la definición formal del lenguaje permite formar programas (y, por tanto, analizarlos) a partir de los caracteres elementales, ambas fases, análisis léxico y sintáctico, podrían fundirse en un solo algoritmo. Tal cosa es, en efecto, posible, pero hay varios motivos para hacer la separación. Aparte de facilitar un desarrollo modular, la tarea del analizador léxico puede modelarse, como veremos con un ejemplo en el apartado 5.2.2, mediante un reconocedor finito, lo que conduce a programas muy eficientes para esa tarea.

Concretemos estas funciones de los analizadores léxico y sintáctico con un sencillo ejemplo: al analizar la sentencia

$$\text{media} := (c1 + c2)/2$$

el explorador, en llamadas sucesivas del analizador sintáctico, reconoce «media», «c1» y «c2» como identificadores, «:=», «+» y «/» como símbolos terminales, con significado especial, del lenguaje, y 2 como una constante entera. Al consultar la tabla de símbolos, si los identificadores se han declarado previamente, los encuentra y devuelve sus punteros al analizador sintáctico (figura 5.13(a)). Este construye el árbol de derivación de la figura 5.13(b). La representación interna de este árbol puede hacerse mediante una estructura de datos como la de la figura 5.13(c), donde los rectángulos representan registros cuyo primer campo es el nombre de la categoría sintáctica, y en los otros campos puede haber punteros. Así, en cada nodo de identificador el puntero contiene la dirección de la tabla de símbolos en la que el explorador ha guardado el nombre y los atributos. En el nodo «cte» el siguiente campo contiene el valor numérico de la constante.

La fase de *análisis semántico** tiene dos funciones. Una es la de interpretación, en el sentido de determinar el significado de las construcciones reconocidas por el analizador sintáctico, y de detectar los errores llamados semánticos (por ejemplo: asignación de un valor real a una variable declarada como booleana o uso de un identificador no declarado). La segunda función consiste en representar ese significado en un *lenguaje intermedio* entre el lenguaje fuente y el de máquina. Esta fase suele llevarse a cabo también en paralelo con el análisis sintáctico: conforme va generando el árbol de derivación, el analizador sintáctico va llamando a las correspondientes *rutinas semánticas*.

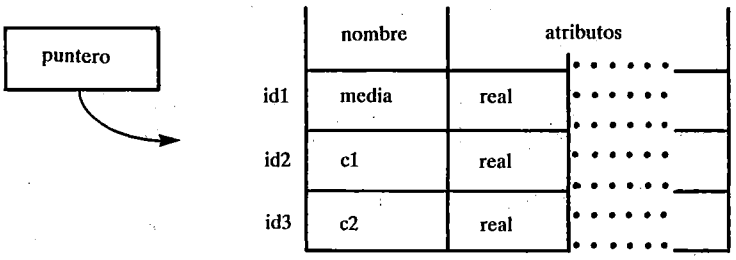
Para el ejemplo anterior, la traducción de la sentencia al lenguaje intermedio podría ser:

sum	id2	id3	m1
asign	2		m2
div	m1	m2	m3
trans	m3		id1

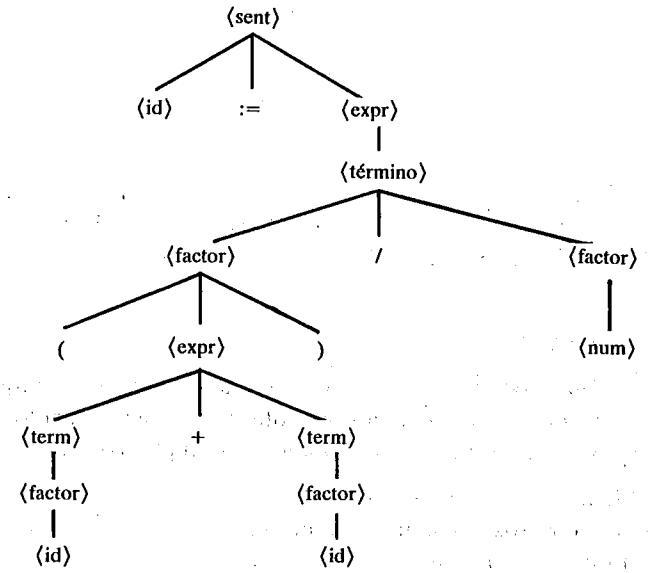
Este lenguaje sería el de una máquina virtual con tres direcciones de memoria. Las instrucciones de máquina (o «cuádruplas») tienen cuatro campos: un código de operación y las direcciones de dos operandos y del resultado. En las instrucciones de transferencia del contenido de una dirección a otra o de asignación de un valor constante a una dirección sólo se utilizan, como vemos, tres de los campos.

En la fase de *optimización* se procura modificar el código intermedio para obtener un programa que se ejecute más eficazmente (más rápido y usando menos recursos; por ejemplo, en el segmento de código anterior las dos instrucciones últimas pueden fundirse en una al tiempo que se ahorra una posición de memoria: div m1 m2 id1). Esta fase puede tener más o menos importancia dependiendo del uso que se vaya a hacer del compilador. Si el programa final compilado va a ejecutarse con mucha frecuencia y rara vez se va a modificar, entonces interesa que esté lo más optimizado posible, a costa de que el tiempo de compilación pueda ser muy grande. Si, por el

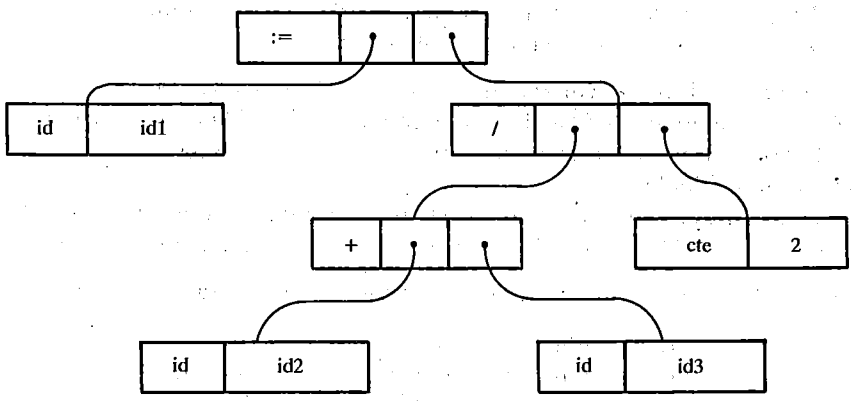
* Se trata aquí de lo que se llama «semántica estática», no de la semántica considerada en el apartado 4. En su mayor parte, este análisis semántico es necesario porque, como ya hemos dicho, en los lenguajes suele haber características dependientes del contexto que no se modelan en la sintaxis.



(a) Tabla de símbolos



(b) Arbol de derivación



(c) Estructura de datos

FIGURA 5.13.

contrario, se está desarrollando y probando un programa, interesa más bien que el tiempo de compilación no sea excesivo aunque el código final no esté muy optimizado.

En la fase final de *generación de código* se obtiene el programa traducido ya al lenguaje objeto. Algunos compiladores terminan con el programa en ensamblador, y otros incluyen las funciones de ensamblaje en la generación de código, de manera que producen un módulo objeto.

Así, el código en ensamblador, optimizado, para el ejemplo que venimos tratando podría ser:

```
M2    CEN    2
      ...
      CAR    ID2
      SUM    ID3
      DIV    M2
      ALM    ID1
```

donde ID1, ID2, ID3 son las direcciones relativas de la tabla de símbolos. Si no existiera la instrucción de dividir, se haría un salto a su subprograma de división.

5.2.2. *Un analizador léxico*

Vamos a ver cómo podría diseñarse un explorador para el lenguaje definido en el apartado 3.4. Consideraremos categorías sintácticas, que el explorador tiene que reconocer e informar de su aparición al analizador sintáctico, a:

a) los identificadores («⟨identif⟩»);
b) los números enteros sin signo («⟨num⟩»);
c) los operadores relacionales («⟨oper⟩»);
d) los operadores aritméticos «+», «−», «*» y «/», y los símbolos terminales «:», «,», «.», «(», «)» y «:=»;
e) las palabras clave «begin», «end», «const», «var», «integer», «if», «then», «else», «while», «do», «read» y «write».

Las palabras clave, así como el símbolo «:=» y los operadores «⟩=», «⟨=» y «⟨ ⟩» se consideran símbolos terminales en la gramática definida en el apartado 3.4, pero para el explorador son cadenas, puesto que para él los símbolos terminales son los caracteres individuales.

Si hemos elegido estos símbolos auxiliares y terminales como categorías sintácticas es porque para cada uno de ellos podemos definir una gramática regular:

a) Para ⟨identif⟩, en lugar de la producción (5) del apartado 3.4 podemos escribir:

$$\begin{aligned}
&\langle \text{id} \rangle ::= a|b| \ldots |z|A|B| \ldots |Z \\
&\langle \text{id} \rangle ::= a\langle \text{restoid} \rangle|b\langle \text{restoid} \rangle| \ldots |Z\langle \text{restoid} \rangle \\
&\langle \text{restoid} \rangle ::= a|b| \ldots |Z|0|1| \ldots |9 \\
&\langle \text{restoid} \rangle ::= a\langle \text{restoid} \rangle|b\langle \text{restoid} \rangle| \ldots |Z\langle \text{restoid} \rangle| \\
&\qquad\qquad 0\langle \text{restoid} \rangle| \ldots |9\langle \text{restoid} \rangle
\end{aligned}$$

Para esta gramática tenemos el reconocedor finito (capítulo 4, apartado 5.3) de la figura 5.14(a), y el lenguaje puede representarse también mediante la expresión regular:

$$(a + b + ... + A + B + ...)(a + b + ... + A + B + ... + Z + 0 + 1 + ... + 9)^*$$

b) Análogamente, para ⟨num⟩ podemos escribir:

⟨num⟩ ::= 0|1| ... |9
⟨num⟩ ::= 0⟨restonum⟩|1⟨restonum⟩| ... |9⟨restonum⟩

El reconocedor es el de la figura 5.14(b), y la expresión regular:

$$(0 + 1 + ... + 9)(0 + 1 + ... + 9)^*$$

c) Para ⟨oper⟩ las reglas regulares serían:

⟨oper⟩ ::= =|⟩|⟨|⟩⟨restop1⟩|⟨⟨restop2⟩
⟨restop1⟩ ::= =
⟨restop2⟩ ::= =|⟩

Y el reconocedor finito es el de la figura 5.14(c). (La expresión regular es, simplemente, la suma de las cadenas =, ⟨, ⟩, ⟨=, ⟩= y ⟨ ⟩ : el lenguaje es finito.)

d) De igual modo, para los otros símbolos tenemos el reconocedor de la figura 5.14(d).

e) Finalmente, para cada una de las palabras clave es trivial encontrar las reglas regulares y el reconocedor (cada una de ellas es un lenguaje con una sola cadena). Por ejemplo, para «begin» podríamos escribir:

⟨begin⟩ ::= b⟨egin⟩
⟨egin⟩ ::= e⟨gin⟩
⟨gin⟩ ::= g⟨in⟩
⟨in⟩ ::= i⟨n⟩
⟨n⟩ ::= n

Y el reconocedor sería el de la figura 5.14(e). No obstante, la consideración de todas las palabras clave implicaría un número muy elevado de estados, y, consecuentemente, muchas instrucciones para el programa explorador. Por ello, suele preferirse otra solución: las palabras clave se tratan, a efectos de reconocimiento, igual que los identificadores, pero inicialmente se introducen en la tabla de símbolos con un atributo que indica su categoría de palabra clave. Cuando el explorador reconoce un identificador tiene que explorar en cualquier caso la tabla de símbolos; si se trata de una variable o de una constante devuelve el puntero, y si está registrado como palabra clave, el código de la palabra.

Para diseñar el explorador con la ayuda de estos reconocedores tenemos que hacer algunas pequeñas modificaciones. Por una parte, como el explorador no puede saber

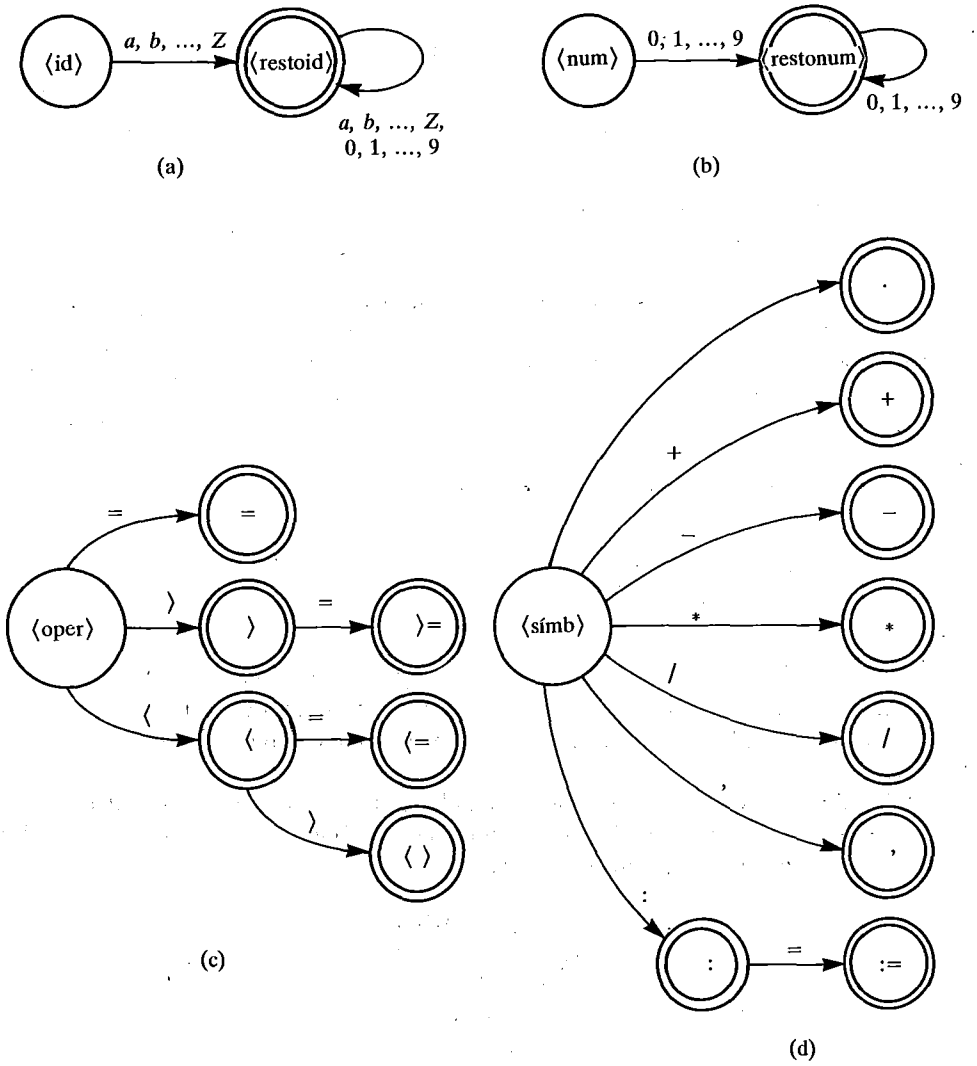

⟨id⟩
a, b, ..., Z
⟨restoid⟩
a, b, ..., Z,
0, 1, ..., 9
(a)
⟨num⟩
0, 1, ..., 9
⟨restonum⟩
0, 1, ..., 9
(b)
⟨oper⟩
=
=
⟩
⟩=
⟨
⟨=
⟨ ⟩
(c)
⟨símb⟩
.
+
−
*
/
,
:
:=
(d)
⟨begin⟩
b
⟨egin⟩
e
⟨gin⟩
g
⟨in⟩
i
⟨n⟩
n
begin
(e)

FIGURA 5.14

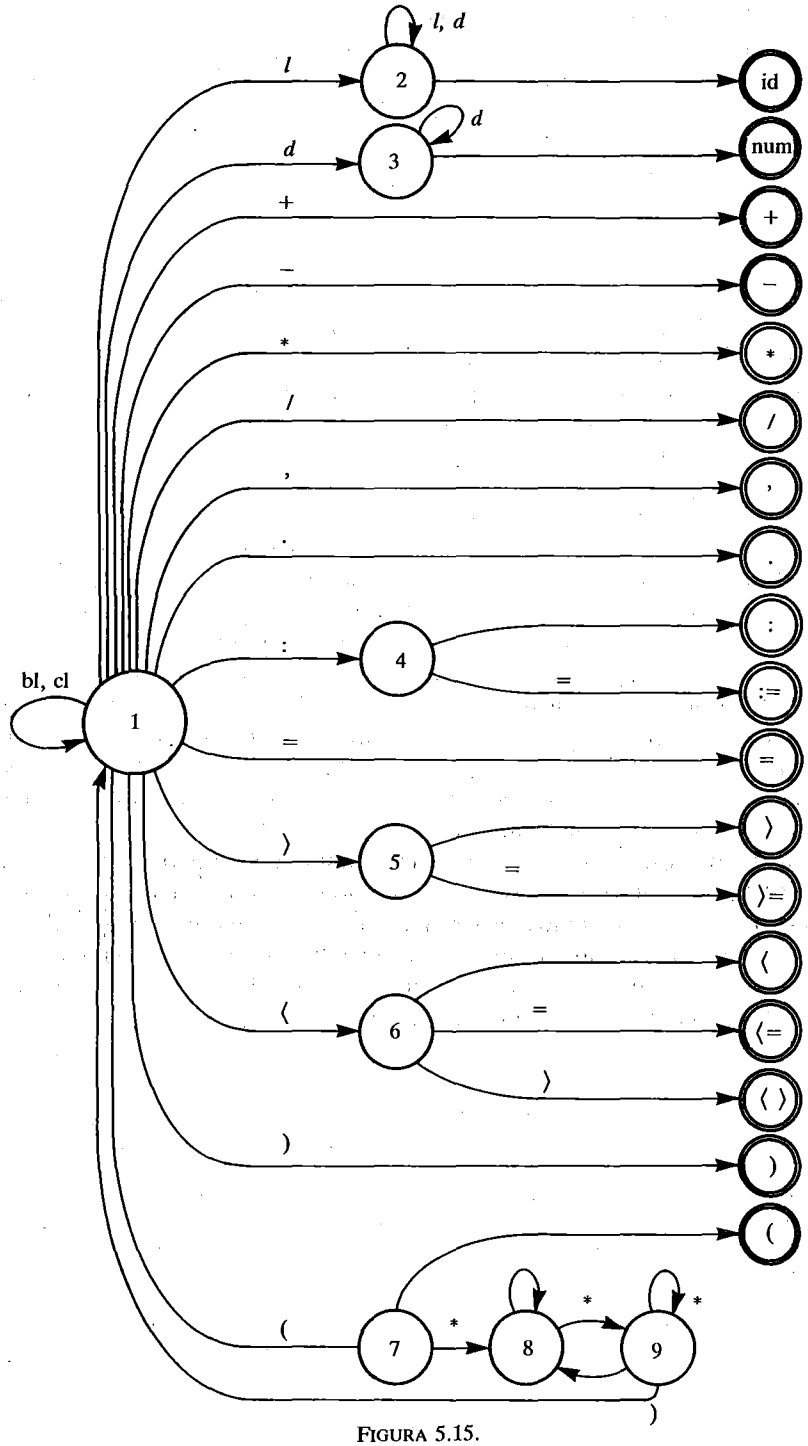


FIGURA 5.15.

a priori cuál es la categoría que va a reconocer en los caracteres que todavía no ha leído, fundiremos todos los diagramas en uno solo con un único estado inicial. Por otra, el explorador irá saltando, a medida que lee caracteres, a un estado u otro, pero sólo puede terminar reconociendo una categoría cuando le llega el primer carácter que sigue a esa categoría (salvo si se trata de un solo símbolo, «*», «/», etc.). Por ejemplo, al explorar de izquierda a derecha la cadena

$$\text{media} := (c1 + c2)/2$$

reconocerá «media» como identificador cuando vea que le sigue un espacio en blanco (o «:», si no se deja el espacio). Teniendo esto en cuenta, los estados finales no pueden ser los que aparecen en la figura 5.14, sino otros a los que se llega a partir de ellos cuando el último carácter leído permite clasificar la cadena de los anteriores en una u otra categoría. De este modo, nuestro explorador siempre irá «adelantado en un carácter».

El diagrama completo del reconocedor (finito y determinista) puede verse en la figura 5.15. Hemos denominado a los estados mediante números, salvo los finales, representados por la categoría que devuelven al analizador sintáctico. Para simplificar, «*l*» representa «cualquier letra» y «*d*», «cualquier dígito». En las transiciones en las que no figura ningún carácter debe entenderse «cualquier otro carácter que no sea uno de los que corresponden a las otras transiciones que salen del mismo estado». Además, hemos incluido la función de ignorar espacios en blanco («bl») y cambios de línea («cl») (transición que entra en y sale del estado 1) y comentarios. Suponemos que los comentarios pueden aparecer en cualquier punto del programa fuente y se caracterizan porque comienzan con «(*» y terminan con «*)». Los estados 7 y 8 identifican el comienzo de un comentario, y cuando se reconoce el final (estado 9 con carácter «)») se vuelve al estado inicial.

A partir del diagrama, podemos escribir la tabla de transiciones del reconocedor (figura 5.16). Esta tabla puede estar en memoria y servir como guía al programa explorador. Para ello, podemos programar un procedimiento, al que llamaremos «transición» que reciba como parámetros de entrada el último carácter leído (car) y el número del estado anterior (est_ant), y que, tras consultar la tabla, devuelva como parámetro de salida el estado resultante (est_act).

	bl	cl	l	d	+	−	*	/	,	.	:	=	⟩	⟨	(	)
1	1	1	2	3	+	−	*	/	,	.	4	=	5	6	7	)
2	id	id	2	2	id	id	id	id	id	id	id	id	id	id	id	id
3	cte	cte	cte	3	cte	cte	cte	cte	cte	cte	cte	cte	cte	cte	cte	cte
4	:	:	:	:	:	:	:	:	:	:	:	:=	:	:	:	:
5	⟩	⟩	⟩	⟩	⟩	⟩	⟩	⟩	⟩	⟩	⟩	⟩=	⟩	⟩	⟩	⟩
6	⟨	⟨	⟨	⟨	⟨	⟨	⟨	⟨	⟨	⟨	⟨	⟨=	⟨⟩	⟨	⟨	⟨
7	(	(	(	(	(	(	8	(	(	(	(	(	(	(	(	(
8	8	8	8	8	8	8	9	8	8	8	8	8	8	8	8	8
9	8	8	8	8	8	8	9	8	8	8	8	8	8	8	8	1

FIGURA 5.16.

Otra posibilidad para programar el procedimiento «transición», sin necesidad de explorar ninguna tabla, es construirlo a base de sentencias condicionales (o de tipo «case») escritas siguiendo el diagrama de la figura 5.15:

```
if est_ant = 1 and car in letra
        then est_act = 2
        else...
```

etc.

En cualquier caso, disponiendo del procedimiento «transición», el explorador puede programarse siguiendo el organigrama de la figura 5.17. El analizador sintáctico deberá leer un carácter antes de llamarlo la primera vez. Se supone que el procedimiento «lee_car» lee el siguiente carácter del programa fuente. «Act_Tab_Simb» consulta, si es necesario (o sea, en el caso de que el estado final sea «id» o «cte») la tabla de símbolos y devuelve los atributos al analizador sintáctico.

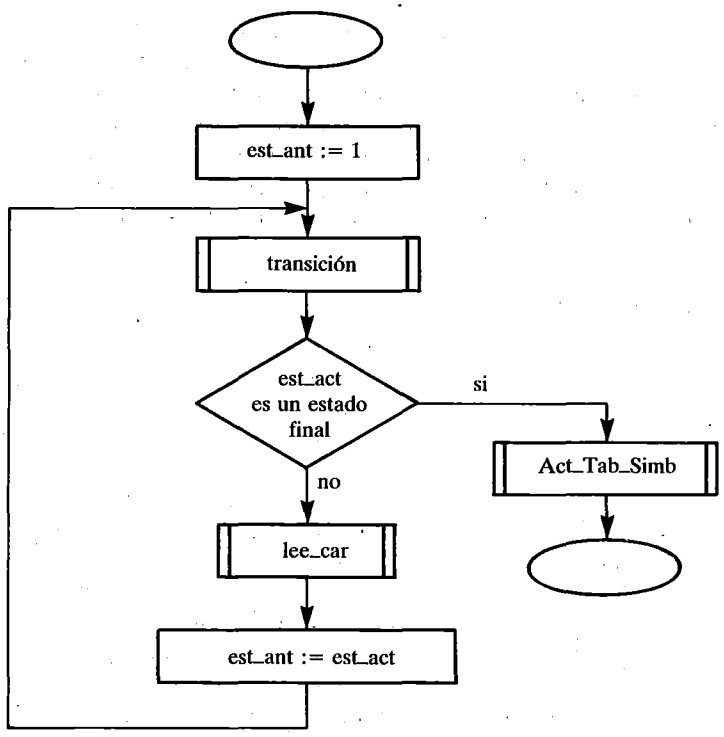


FIGURA 5.17

5.2.3. *Un analizador sintáctico*

Existe una gran variedad de algoritmos de reconocimiento para gramáticas libres de contexto. En general, pueden clasificarse en *descendentes* y *ascendentes*. Los primeros construyen los árboles de derivación desde la raíz (símbolo inicial) hacia abajo (con nuestro convenio de representación, la raíz está arriba, al contrario de lo que ocurre con los árboles de la naturaleza). En los algoritmos ascendentes el análisis procede en sentido contrario: a partir de las categorías sintácticas, buscan el árbol ascendiendo hacia la raíz. Para una gramática determinada, es más fácil diseñar un algoritmo descendente (siempre que la gramática cumpla ciertas condiciones). Pero los algoritmos ascendentes son más adaptables a distintos tipos de gramáticas. Por esta razón, los primeros son preferibles cuando se diseña «a mano» un compilador para un lenguaje, mientras que los segundos resultan más útiles en herramientas que generan algoritmos de análisis a partir de las especificaciones del lenguaje.

Veremos aquí, aplicado al minilenguaje definido en el apartado 3.4, el algoritmo más sencillo y más utilizado. Se trata de un algoritmo descendente recursivo llamado *predictivo,* que está basado en la definición sintáctica formal del lenguaje: para cada símbolo auxiliar se escribe un procedimiento de acuerdo con la producción que le corresponde. Como en el lado derecho de esa producción puede figurar el propio símbolo (u otro auxiliar en cuya producción figure el primero), los procedimientos pueden ser recursivos. Este algoritmo funciona «un símbolo por adelantado y sin retroceso» («one-symbol-lookahead without backtracking»). Esto quiere decir que, conforme se avanza en el análisis del programa fuente, la decisión sobre la acción a seguir se basa exclusivamente en el último símbolo (categoría sintáctica) entregado por el explorador (del mismo modo que éste entrega el resultado tras leer el carácter que le sigue). «Sin retroceso» significa que no vuelve atrás en el proceso de construir el árbol de derivación. Ahora bien, para que esto sea posible es preciso que la gramática cumpla las dos condiciones enunciadas en el apartado 3.2. Veámoslo con dos ejemplos.

a) Consideremos la gramática

$$S ::= A|B$$
$$A ::= aA|b$$
$$B ::= aB|c$$

que no cumple la primera condición, porque a partir de A y B pueden derivarse cadenas que empiezan con el mismo símbolo terminal, «*a*». Si tratamos de reconstruir el árbol de derivación de la cadena «*ac*», primero leeríamos el símbolo «*a*». La aplicación de la primera producción nos llevaría a la segunda, en la cual vemos que el símbolo coincide con el primero de «*aA*». En este momento, el árbol de derivación construido es el de la figura 5.18(a). Avanzamos en la lectura, tratando de emparejar el resto de la cadena de entrada con «*A*». Pero el resto es «*c*», y no existe la producción «$A ::= c$». Por tanto, sería preciso retroceder para elegir «$S ::= B$» y construir el árbol de la figura 5.18(b). Vemos que el problema está en que el símbolo leído, «*a*»,

no determina unívocamente la producción a elegir. El lector puede comprobar que esta otra gramática:

$$S ::= A|aS$$
$$A ::= b|c$$

es equivalente y no presenta ese problema.

FIGURA 5.18

b) La gramática

$$S ::= a|Sb$$

es recursiva por la izquierda, por lo que no cumple la segunda condición. Si tratamos de reconocer la cadena «*abb*», en cuanto se lee la primera «*b*» se produce un bucle infinito, como indica el árbol de la figura 5.19(a), en el que el procedimiento correspondiente a «*S*» se llama siempre a sí mismo sin seguir leyendo más caracteres

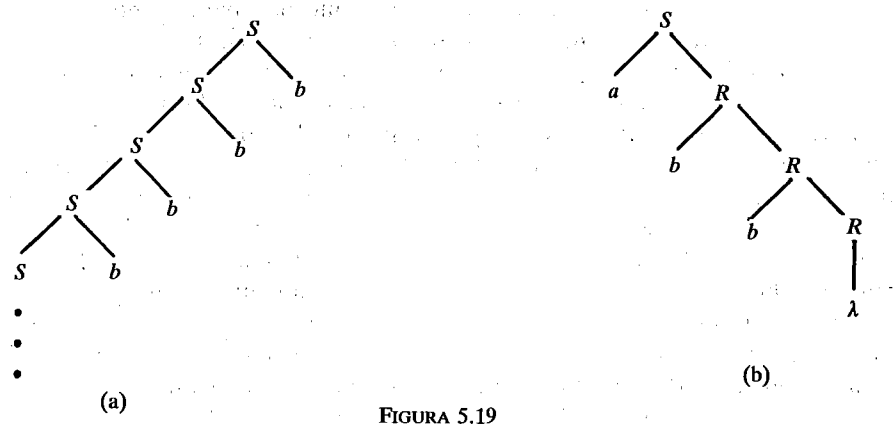


FIGURA 5.19

de la cadena de entrada. El problema está en que habría que seguir leyendo hasta el final para decidir en qué momento aplicar la primera producción. Con la gramática equivalente

$$S ::= aR$$
$$R : = bR|\lambda$$

se puede generar, leyendo los símblos de «*abb*» de izquierda a derecha y sin retroceder, el árbol de la figura 5.19(b).

Es fácil comprobar que la gramática de nuestro lenguaje cumple las dos condiciones. Veamos ahora un método mediante el cual puede obtenerse el algoritmo predictivo a partir de la definición formal (de las producciones, o, equivalentemente, del diagrama sintáctico) para cualquier gramática que cumpla esas condiciones.

El algoritmo en cuestión puede obtenerse a partir de los diagramas sintácticos siguiendo las normas resumidas en la figura 5.20. Para cada subdiagrama sintáctico del tipo que aparece a la izquierda se escribe un procedimiento de acuerdo con el organigrama de la columna central, o con el esquema de codificación en Pascal de la última columna. Explicamos las abreviaturas que se utilizan:

$P(\alpha)$ significa «procedimiento asociado al subdiagrama sintáctico de α».

«categ» es la categoría sintáctica que devuelve el explorador cuando se le llama. Esto sucede, como indica la última parte de la figura, cuando en el diagrama sintáctico aparece un símbolo terminal (recuérdese que, para el analizador sintáctico, los símbolos terminales son las categorías sintácticas). Como el analizador va «un símbolo por adelantado», la última categoría entregada por el explorador tiene que coincidir con el símbolo del diagrama; entonces, se llama al explorador y se sigue adelante.

«prim(α)» representa el conjunto de todos los primeros símbolos terminales que pueden encontrarse en las cadenas derivadas a partir de α. La primera de las condiciones que se han enunciado para la gramática se puede expresar también diciendo que si $A ::= \alpha_1|\alpha_2| \ldots |\alpha_n$ es una producción, entonces $\text{prim}(\alpha_1) \cap \text{prim}(\alpha_2) \cap \ldots \cap \text{prim}(\alpha_n) = \emptyset$. Normalmente, el diagrama sintáctico muestra prim(α) muy claramente, porque α suele empezar por un símbolo terminal, y prim(α) es justamente ese símbolo. Por ejemplo, en la ramificación que hay para «sentencia» en los diagramas de la figura 5.6, $\text{prim}(\alpha_1)$ = begin, $\text{prim}(\alpha_2)$ = identif, $\text{prim}(\alpha_3)$ = if, etc.

La aplicación de estas normas al caso particular de nuestro lenguaje es inmediata. En la figura 5.21 puede verse el organigrama correspondiente al primer subdiagrama de la figura 5.6. Este podría ser el programa principal, a partir del cual se llama a «bloque», y de éste a los demás procedimientos. Antes de llamar la primera vez al explorador se inicializa la variable «car» con un espacio en blanco. Esto es necesario porque, como se recordará, el explorador funciona con «un carácter por adelantdo». El resto de los organigramas no presenta mayores problemas. A título de ejemplo, indicamos en las figuras 5.22, 5.23 y 5.24 los correspondientes a los procedimientos «expresión», «término» y «factor».

En esta descripción nos interesaba sobre todo ver cómo la definición sintáctica formal del lenguaje es una herramienta útil para el diseño sistemático del procesador. Por ello, hemos prescindido de detalles, como el tratamiento de errores y el manejo

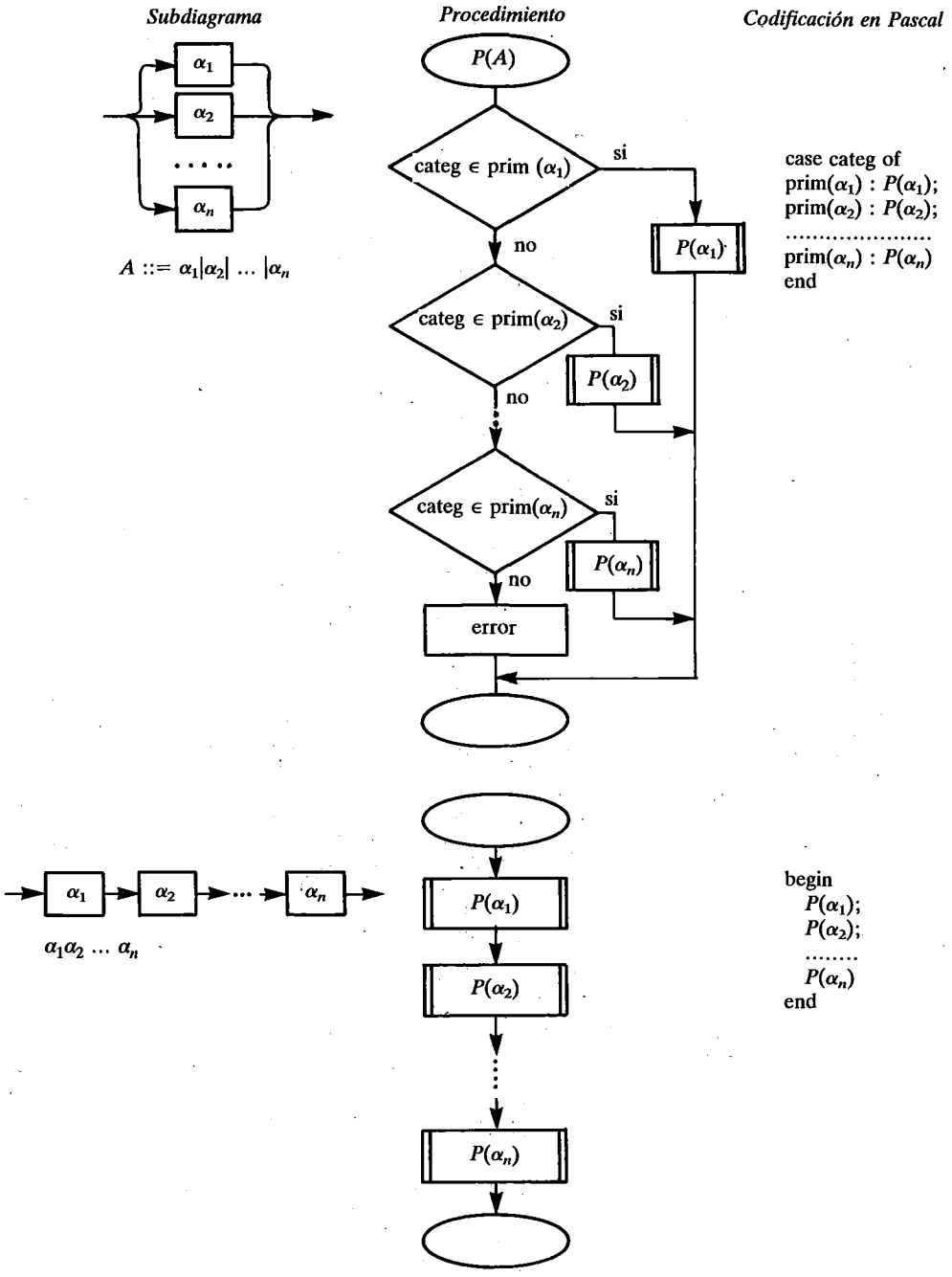

Subdiagrama
Procedimiento
Codificación en Pascal
α_1
α_2
α_n
$A ::= \alpha_1|\alpha_2| \ldots |\alpha_n$
P(A)
categ ∈ prim (α_1)
si
no
$P(\alpha_1)$
categ ∈ prim(α_2)
si
$P(\alpha_2)$
no
categ ∈ prim(α_n)
si
$P(\alpha_n)$
no
error
case categ of
prim(α_1) : $P(\alpha_1)$;
prim(α_2) : $P(\alpha_2)$;
.....
prim(α_n) : $P(\alpha_n)$
end
α_1 α_2 ... α_n
$\alpha_1\alpha_2 \ldots \alpha_n$
$P(\alpha_1)$
$P(\alpha_2)$
$P(\alpha_n)$
begin
$P(\alpha_1)$;
$P(\alpha_2)$;
.....
$P(\alpha_n)$
end

FIGURA 5.20.

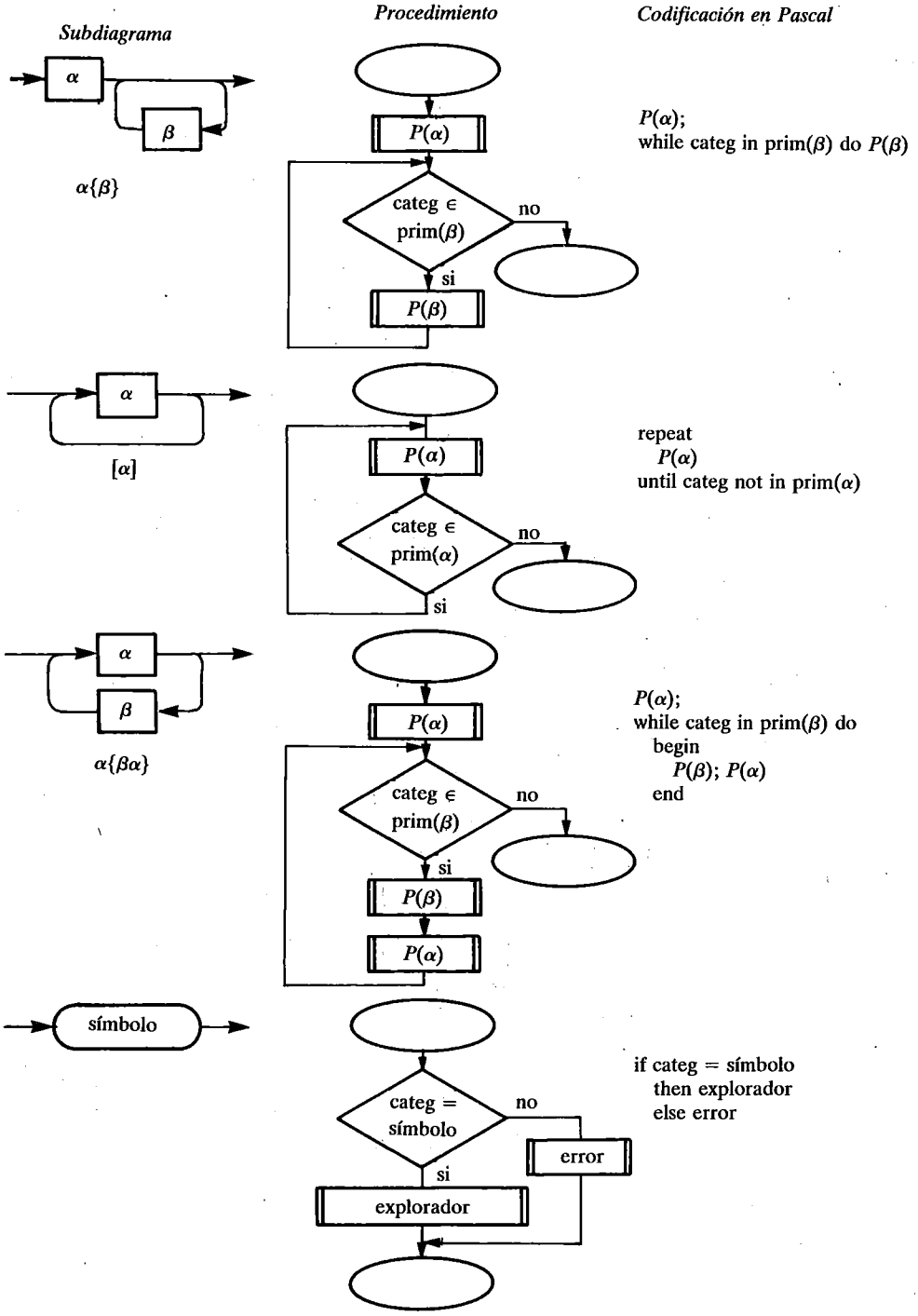


FIGURA 5.20. *(Continuación).*

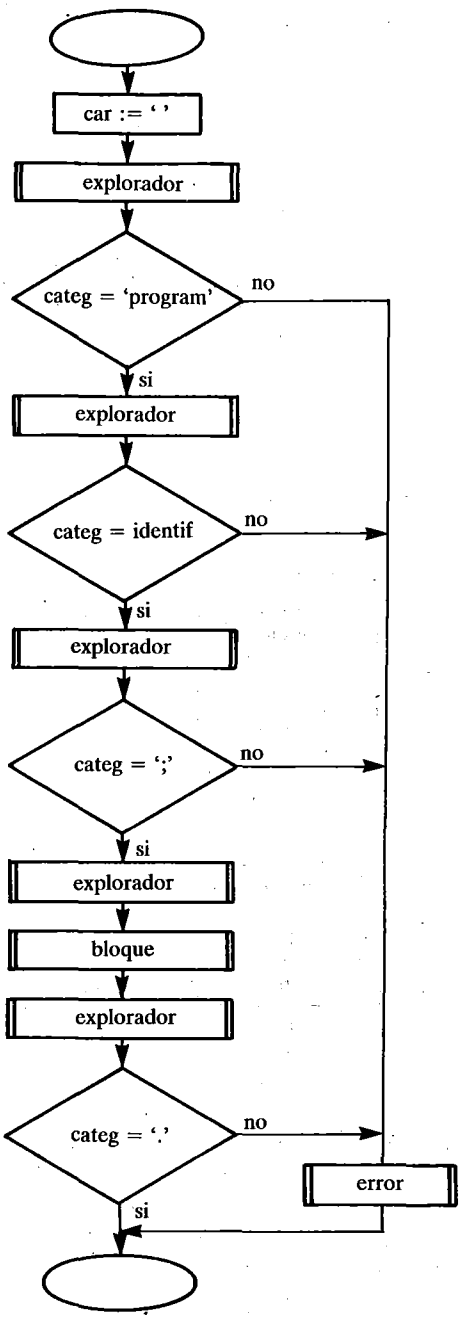

car := ' '
explorador
categ = 'program'
no
si
explorador
categ = identif
no
si
explorador
categ = ';'
no
si
explorador
bloque
explorador
categ = '.'
no
error
si

FIGURA 5.21.

de la tabla de símbolos, que, aun siendo importantes, serían ya propios de un texto sobre compiladores.

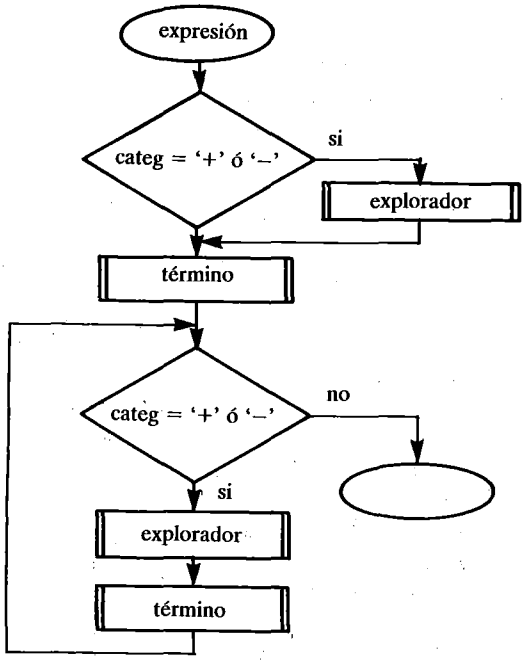


FIGURA 5.22.

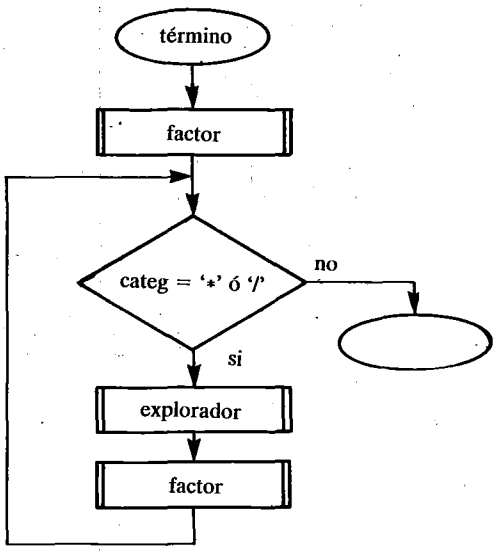


FIGURA 5.23.

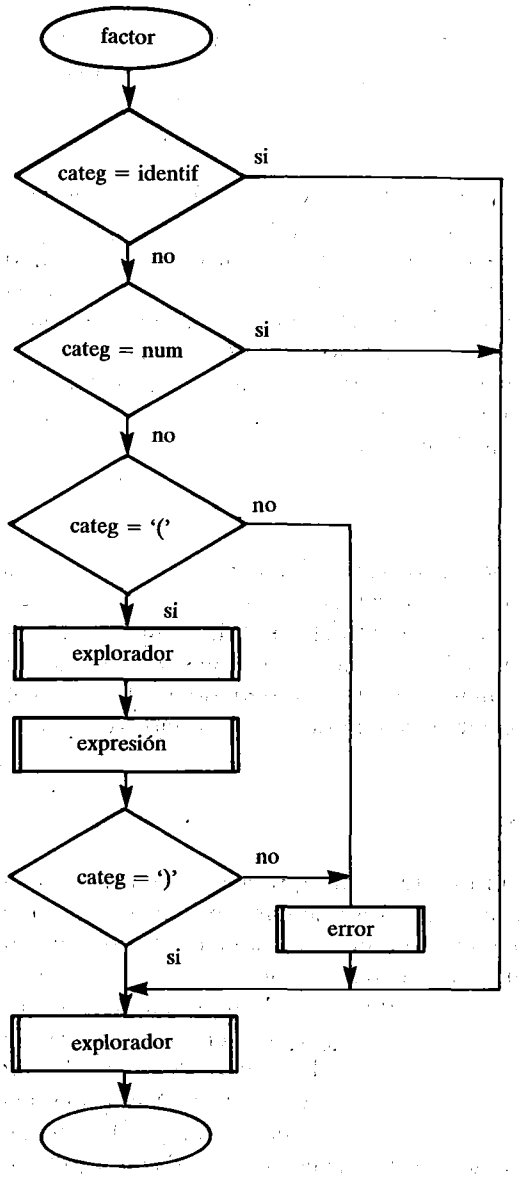


FIGURA 5.24.

Quizás el lector se pregunte por la construcción del árbol del árbol de derivación, que, como decíamos en el apartado 5.2.1, es una tarea del analizador sintáctico. Pues bien, en este algoritmo predictivo el árbol está implícito en la secuencia de procedimientos que se van llamando. Los punteros que aparecen en la figura 5.13 los va devolviendo el explorador conforme va reconociendo identificadores, y los utilizan las

rutinas semánticas para ir generando el código intermedio. Estas rutinas semánticas son procedimientos asociados a cada uno de los procedimientos que realizan el análisis sintáctico. Por ejemplo, cuando se analiza la sentencia

$$\text{media} := (c1 + c2)/2$$

al entrar en el procedimiento «sentencia» se llama al explorador. Este reconoce el identificador «media» y devuelve su puntero, id1, que queda guardado en una variable local de «sentencia». «sentencia» llama de nuevo al explorador, que devuelve «:=», y luego a «expresión»; ésta llama a «término», y «término» llama al explorador y a «factor» (ver figuras 5.23 a 5.25), que, al encontrarse con «(», llama a su vez a «expresión». Dentro de esta ejecución de «expresión» se llama dos veces al explorador y a «término». Pues bien, las rutinas semánticas asociadas comprueban los tipos de *c*1 y *c*2 (cuyos punteros, id2 e id3, ha entregado el explorador) para ver si su suma es posible y construyen el cuarteto

sum id2 id3 m1

(ver apartado 5.2.1). Se sale de este nivel de recursión en la ejecución de «expresión» al devolver el explorador el carácter «)». En este momento se ha construido, implícitamente, la parte más baja del árbol de la figura 5.13. Se completa luego la ejecución de la primera llamada a «expresión», en la que las rutinas semánticas construyen los dos siguientes cuartetos (apartado 5.2.1), y, al volver a «sentencia», la rutina semántica asociada a ésta construye el último cuarteto.

5.3. Interpretadores

La estructura de un programa interpretador es mucho más sencilla que la de un compilador. En esencia, puede representarse por el bucle de la figura 5.25. CONT_SENT es una variable que indica en cada momento la posición en el programa fuente de la siguiente sentencia a ejecutar. Su función es similar a la de «CONT_ENS» del ensamblador descrito en el apartado 5.1.3: en principio, se va incrementando en una unidad, pero puede tomar otro valor, dependiendo del análisis de la sentencia (si ésta es una llamada a procedimiento, o de control de un bucle, etc.). A cada tipo de sentencia posible se le asocia un procedimiento en el programa interpretador para ejecutar en la máquina real la acción especificada por la sentencia.

El análisis de las sentencias puede estar guiado, como en los compiladores, por la definición formal de la sintaxis del lenguaje.

6. RESUMEN

Los lenguajes de *bajo nivel* (ensambladores y macroensambladores) son cercanos a los lenguajes de las máquinas reales, mientras que los de *alto nivel* facilitan la tarea de codificar algoritmos, porque independizan al programador de los detalles concretos

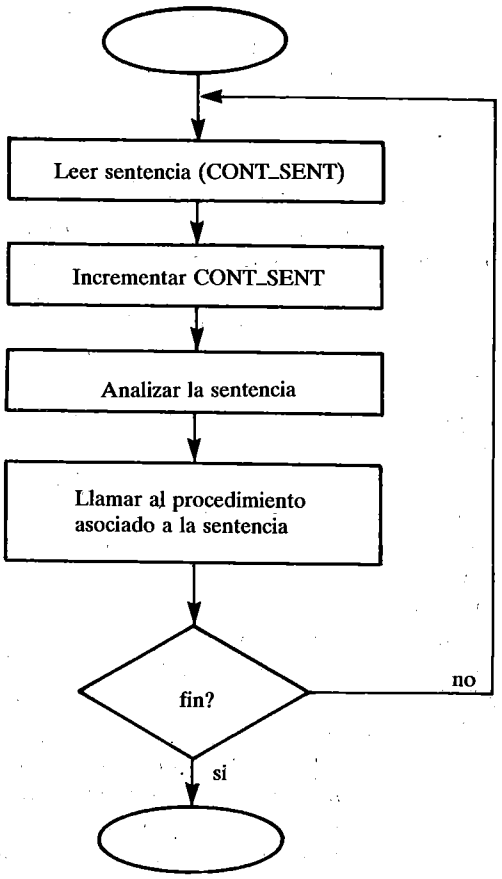


FIGURA 5.25.

del harware y le permiten concentrarse en el problema a resolver. En cualquier caso, siempre se puede considerar que todo lenguaje define a una máquina que lo «entiende», es decir, que reconoce los programas escritos en ese lenguaje y los ejecuta. Si el lenguaje no corresponde al de una máquina real, entonces la máquina definida es una *máquina virtual.*

Los *procesadores de lenguaje* son programas que permiten ejecutar en una máquina real programas escritos para una máquina virtual. El lenguaje de la máquina virtual se llama *lenguaje fuente,* y el de la máquina real, *lenguaje objeto.* Hay dos tipos esencialmente diferentes de procesadores: los *traductores* y los *interpretadores.* Los primeros toman como datos la sucesión de caracteres que constituyen un programa fuente y generan un programa objeto que posteriormente se puede ejecutar en la máquina real. Los traductores para lenguajes ensambladores se llaman *ensambladores,* y los diseñados para lenguajes de alto nivel, *compiladores.* Los interpretadores no generan de una sola vez todo el programa objeto correspondiente a un programa

fuente, sino que van analizando sucesivamente las sentencias de éste y generando sobre la marcha las instrucciones de máquina necesarias para ejecutar esas sentencias.

Los lenguajes tradicionales, sean de bajo o alto nivel, suelen ser *imperativos:* las instrucciones o sentencias expresan órdenes elementales para una máquina, real o virtual. La idea de los lenguajes *declarativos* es que el programa no sea una sucesión de órdenes, sino una especificación del problema. En los lenguajes de *programación funcional* esta idea se plasma en tratar de expresar los problemas a través de aplicaciones de funciones matemáticas, mientras que en los lenguajes de *programación lógica* se trata de formular los problemas por medio de sentencias de la lógica formal.

La *definición formal de la sintaxis* de un lenguaje de programación, sea por notación BNF o por diagramas sintácticos, es de gran utilidad en el diseño de las fases de análisis léxico y sintáctico de los procesadores de lenguaje. Generalmente, la parte que define las *categorías sintácticas* del lenguaje (palabras clave, identificadores, números, etc.), a partir de los caracteres elementales (símbolos terminales) se puede definir mediante una *gramática regular,* a la que corresponde un reconocedor finito, que se materializa en el programa *analizador léxico* o *explorador.* El resto de la definición sintáctica se formaliza como una *gramática libre de contexto,* a la que corresponde un reconocedor de pila, y que da lugar al programa *analizador sintáctico.* La función del explorador es leer caracteres del programa fuente de entrada y reconocer las distintas categorías sintácticas, mientras que la del analizador sintáctico consiste básicamente en reconstruir el árbol de derivación del programa en cuestión según la gramática del lenguaje. Las *rutinas semánticas* se ocupan de la traducción, sea al lenguaje objeto final, o a un código intermedio.

La *definición formal de la semántica* es un asunto más difícil, y existen actualmente varios enfoques (*operacional, denotacional* y *axiomático*) que pueden aplicarse al diseño de nuevos lenguajes, a la verificación y construcción sistemática de programas y al diseño de herramientas y entornos de programación potentes.

7. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

La historia de los lenguajes de programación es paralela a la de los modelos de máquinas computadoras, reales o virtuales. Suele decirse que el primer programador fue Ada Augusta, Condesa de Lovelace, que a mediados del siglo pasado propuso diversos programas para una máquina que jamás llegó a construirse, el «analytical engine» de Babagge.

Pero el comienzo de la historia moderna de los ordenadores está jalonado por el modelo de von Neumann, propuesto en 1946, y la aparición del primer ordenador comercial, el Univac 1, en 1951. En esta época aún se programaba en lenguaje de máquina. En 1952, en el M.I.T., se diseñó un lenguaje llamado SAP (Symbolic Assembly Program), un ensamblador para el IBM 701. Y en el mismo año, Grace Murray Hopper publicaba el primer artículo sobre compiladores, en el que se describía «A-O», un compilador para el Univac 1. Por entonces, estos trabajos se consideraban dentro del campo de la «programación automática». Hay un trabajo de

Knuth y Pardo (1980) donde se expone con detalle la historia de los primeros tiempos de la programación.

Al final de los años 50 aparecen ya las primeras versiones de tres lenguajes de alto nivel que todavía siguen siendo ampliamente utilizados: FORTRAN (Backus *et al.*, 1957), COBOL (cuyo primer borrador data de 1959) y LISP (McCarthy, 1960). Backus propuso su notación en 1959, para describir el «International Algebraic Language» (Backus, 1959), precursor de Algol (Naur, 1963). Y heredero de Algol es Pascal, diseñado por Niklaus Wirth (1971), y que, a su vez, ha sido inspirador de otros lenguajes más modernos, como Ada y Modula-2. En la primera edición del manual de Pascal se introducían los diagramas sintácticos como alternativa a la notación BNF.

Como ya hemos dicho en el apartado 2.4.2, la programación funcional tiene su origen teórico en el cálculo lambda, propuesto por el matemático Alonzo Church (1936). Un artículo ya clásico sobre los lenguajes aplicativos es el de Backus (1978). En cuanto a Prolog, procede de los trabajos de un equipo de investigadores de Marsella sobre lenguaje natural (Colmerauer *et al.*, 1973). Cabe citar también una comunicación de Kowalski (1974) en la que por primera vez se expresan claramente las ideas básicas de la programación lógica.

La semántica operacional está ligada a los estudios sobre relaciones entre lenguajes y máquinas interpretadoras, como los de McCarthy y Painter (1967) y Allen *et al.* (1972). El origen de la semántica denotacional se encuentra en la «teoría de dominios» de Scott y Strachey (1971) y en la «teoría del punto fijo» de Manna (1974). La semántica axiomática está relacionada con la preocupación por demostrar matemáticamente la corrección de los programas. Un precursor en esta línea es el creador de LISP, John McCarthy: «en lugar de comprobar los programas con casos de prueba hasta que están depurados, deberíamos poder demostrar que tienen las propiedades deseadas» (McCarthy, 1962). La teoría fue desarrollada por diversos autores, principalmente Floyd (1967), Hoare (1969) y Dijkstra (1975). El transformador de predicados que aquí hemos llamado «pmd» lo introdujo Dijkstra con el nombre de «wp» («weakest precondition»).

Para ampliar los contenidos de este capítulo, recomendamos los siguientes libros:

Sobre lenguajes de programación en general, Horowitz (1984) describe características comunes (tipos, procedimientos, abstracciones de datos, etc.) y dedica tres capítulos a la programación funcional y a dos tipos de lenguajes que aquí no hemos comentado: los de «flujo de datos» y los «orientados a objetos». El enfoque de Tucker (1986) es diferente: describe, en sucesivos capítulos, diversos lenguajes: Pascal, FORTRAN, COBOL, PL/I, SNOBOL, APL, LISP, Prolog, C, Ada y Modula-2.

De Pascal, la referencia básica es el manual de Jensen y Wirth (1985). Textos didácticos sobre el lenguaje hay muchos. Por ejemplo, Grogono (1980), Schneider y Bruell (1981), o, en español, Sanchís y Morales (1981) y Keller (1982).

Dos libros recomendables sobre LISP son el de Weissman (1976) y el de Winston y Horn (1984). El más conocido sobre Prolog es el de Cloksin y Mellish (1981). El de Bratko (1986) es especialmente interesante, al cubrir tanto Prolog como técnicas y aplicaciones de inteligencia artificial.

Respecto a la semántica operacional, hay un libro de Ollongren (1974) que está dedicado a la definición de lenguajes de programación a través de interpretadores.

Sobre semántica denotacional, un artículo y un libro clásicos son, respectivamente, el de Tennent (1976) y el de Stoy (1977). Un texto resumido y muy claro es el de Gordon (1979). En Bjørner y Jones (1972) se describe VDM (Vienna Development Method), un método sistemático de especificación formal y desarrollo de programas basado en semántica denotacional. En cuanto a la semántica axiomática y sus aplicaciones a la verificación de programas son especialmente recomendables los libros de Gries (1981) y Hehner (1984).

Finalmente, sobre procesamiento de lenguajes, un texto clásico es el de Gries (1971). Más reciente, y muy amplio, detallado y riguroso, es el de Aho *et al.* (1986). También son recomendables, a un nivel más elemental, el de Calingaert (1979) y el de Pyster (1980), y una descripción resumida puede encontrarse en el capítulo 5 de Wirth (1976). Pero si el lector prefiere referencias en español, aparte de los libros citados de Gries y de Wirth, que están traducidos, le señalamos el de Sanchís y Galán (1986).

REFERENCIAS BIBLIOGRAFICAS

AHO, A. V., SETHI, R. y ULLMAN, J. D. *Compilers. Principles, techniques, and tools.* Addison-Wesley, Reading, Mass., 1986.

AIKINS, J. S. Prototypical knowledge for expert systems. *Artificial Intelligence,* 20 (1983), pp. 163-210.

ALABAU, A. y FIGUERAS, J. *Algoritmos y máquinas.* Universidad Politécnica de Barcelona, C.P.D.A., 1975.

ALAGIC, S. y ARBIB, M. A. *The design of well-structured and correct programs.* Springer-Verlag, N. Y., 1978.

ALLEN, C. D., CHAPMAN, D. N. y JONES, C. B. *A formal definition of ALGOL 60.* TR. 12.105, Hursley (UK), agosto 1972.

ALLEN, J. F. An internal-based representation of temporal knowledge. *Proc. 7th Int. Joint Conf. Artif. Int.,* Vancouver, Canadá, 1981, pp. 221-226.

ALTY, J. y COMMBS, M. *Expert systems. Concepts and examples.* John Wiley, 1974. (Traducción de R. Pérez y P. Rubio: *Sistemas expertos: conceptos y ejemplos.* Diaz de Santos, Madrid, 1986).

ARBIB, M. A. *Brains, machines and mathematics.* McGraw-Hill paperbacks, 1965. (Existe versión castellana en la colección Alianza Universidad.)

ARBIB, M. A. *Theories of abstract automata.* Prentice-Hall, Englewood Cliffs, N. J., 1969.

ARBIB, M. A. *Computers and the cybernetic society.* Academic Press. 1977. (Publicado en castellano por Editorial AC, bajo la supervisión de F. Sáez Vacas, 1978).

ASHBY, W. R. *An introduction to cybernetics.* Chapman and Hall, Londres, 1956. (Traducción de J. Santos: *Introducción a la cibernética.* Nueva Visión, Buenos Aires, 1960).

BACKUS, J. W. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. *Proc. Internat. Conf. Information Processing,* UNESCO, París, 1959 (Butterworth's, Londres, 1960, pp. 125-132).

BACKUS, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM,* 21, 8 (1978), pp. 613-641.

BACKUS, J. W., BEEBER, R. J., BEST, S., GOLDBERG, R., HAIBT, L. M., HERRICK, H. L.,

NELSON, R. A., SAYRE, D., SHERIDAN, P. B., STERN, H., ZILLER, I., HUGHES, R. A. y NUTT, R. The FORTRAN automatic coding system. *Proc. Western Joint Computer Conf.*, AIEE (actualmente IEEE), Los Angeles, 1957.

BAR-HILLEL, Y., PERLES, M. y SHAMIR, E. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetic, Sprachwissenschaft und Kommunikationsforschung,* 14 (1961), 143-172.

BARTEE, T. C. *Digital computers fundamentals.* McGraw-Hill, N. Y., 1960 (sexta ed.: 1985).

BARTEE, T. C., LEBOW, I. L. y REED, I. S. *Theory and design of digital machines.* McGraw-Hill, N. Y., 1962.

BJØRNER, D. y JONES, C. B. *Formal specification and software development.* Prentice-Hall, Englewood Cliffs, N. J., 1982.

BOCHMANN, G. V. *Architecture of distributed computer systems.* Springer-Verlag, Berlín, 1979.

BOCHVAR, D. On three-valued logical calculus and its application to the analysis of contradictions. *Matematiceskij Sbornik,* 4 (1939), pp. 353-369.

BÖHM, C. y JACOPINI, G. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM,* 9, 5, mayo 1966.

BOOLE, G. *An investigation of the laws of thought.* Dover Publ. Inc., N. Y., 1854.

BOOTH, T. L. *Sequential machines and automata theory.* Wiley, N. Y., 1967.

BRATKO, I. *Prolog programming for artificial intelligence.* Addison-Wesley, Reading, Mass., 1986.

BROZOZOWSKI, J. A. A survey of regular expressions and their applications. *IRE Trans. Elec. Comp.*, EC-11 (1962), pp. 324-325.

BROZOZOWSKI, J. A. Regular expressions for linear sequential circuits. *IEEE Trans. Elec. Comp.*, EC-14 (1965), pp. 148-156.

BUCHANAN, B. G., SUTHERLAND, G. L. y FEIGENBAUM, E. A. Heuristic DENDRAL: A program for generating explanatory hypotheses in organic chemistry. En B. Meltzer y D. Michie (Eds.): *Machine Intelligence, vol 4,* Edinburgh Univ. Press, 1969, pp. 209-254.

BUCHANAN, B. G. y SHORTLIFFE, E. H. (eds.) *Rule-based expert systems.* Addison-Wesley, Reading, Mass., 1984.

CALINGAERT, P. *Assemblers, compilers and program translation.* Pitman, Londres, 1979.

CANTOR, D. C. On the ambiguity problem of Backus systems. *Journal of ACM,* 9 (1962), 4, 477-479.

CARBONELL, J. G., CULLINGFORD, R. y GERSHMAN, A. Steps towards knowledge-based machine translation. *IEEE Trans. Pattern Anal. and Mach. Int.,* 3, 4 (julio 1981), pp. 376-392.

CHANDRASEKARAN, B., MITTAL, S. y SMITH, J. N. RADEX. Towards a computer-based radiology consultant. En E.S. Gelsema y L. N. Kanal (eds.): *Pattern recognition in practice.* North-Holland, Amsterdam, 1980, pp. 463-474.

CHARNIAK, E. y MCDERMOTT, D. *Introduction to artificial intelligence.* Addison-Wesley, Reading, Mass., 1985.

CHOMSKY, N. Three models for the description of language. *I.R.E. Trans. Inf. Theory,* IT-2 (1956), 113-114.

CHOMSKY, N. On certain formal properties of grammars. *Information and Control,* 2 (1959), 137-167.

CHOMSKY, N. Context-free grammars and pushdown storage. *Quart. Prog. Dept. No. 65* (1962), MIT Res. Lab. Elect., 187-194.

CHOMSKY, N. Prólogo al libro de Gross y Lentin (1967).

CHOMSKY, N. *Language and mind,* Harcourt Braze, N. Y., 1968.

CHOMSKY, N. *The logical structure of linguistic theory.* Plenum Press, N. Y., 1975.

CHOMSKY, N. y MILLER, G. A. Finite state languages. *Information and Control,* 1 (1958), 91-112.

CHOMSKY, N. y SCHUTZENBERGER, M. P. The algebraic theory of context-free languages. *Computer Programming and Formal Systems.* North-Holland, Amsterdam, 1963, 118-161.

CHURCH, A. An unsolvable problem in elementary number theory. *American Journal of Mathematics,* 58 (1936), pp. 345-363.

CLEMENTS, A. *The principles of computer hardware.* Oxford Univ. Press, 1985.

CLOCKSIN, W. F., DARLINGTON, J., KENNAWAY, S. R. y SLEEP, M. R. «Part II. Declarative systems». En Chambers, F. B., Duce, D. A. y Jones, G. P. (eds.): *Distributed computing.* Academic Press, Londres, 1984, pp. 55-138.

CLOKSIN, W. F. y MELLISH, C. S. *Programming in Prolog.* Springer Verlag, Berlín, 1981.

CODD, E. F. A relational model of data for large shared data banks. *Comm. ACM,* 13, 6 (1970), pp. 337-387.

COLMERAUER, A., KANONI, H., PASERO, R. y ROUSSEL, P. *Un système de communication homme-machine en français.* Groupe d'intelligence artificielle. Université d'Aix-Marseille. Luminy, 1973.

CORDIER, M. O. Los sistemas expertos. *Mundo Científico,* 34 (marzo 1984), pp. 236-247.

CORGE, CH. *Eléments d'informatique.* Larousse Université, París, 1975.

COULTER, N. S. Software science and cognitive psychology, *IEEE Trans. Softw. Eng.,* SE-9, 2 (marzo 1983), pp. 166-171.

CUENA, J. Sistemas expertos. En R. Valle, J. Barberá y F. Ros (Eds.): *Inteligencia artificial: introducción y situación en España.* Fundesco, Madrid, 1984, pp. 31-38.

CUENA, J. *Los sistemas expertos: concepto, realización y técnicas de construcción.* Cuadernos CDTI, núm. 17, Madrid, marzo. 1985a.

CUENA, J. Sistemas expertos. *Mundo electrónico,* 149 (marzo 1985b), pp. 73-80.

CUENA, J. *Lógica informática.* Alianza, Madrid, 1986.

CUENA, J., FERNÁNDEZ, G., VERDEJO, F. y LÓPEZ DE MÁNTARAS, R. *Inteligencia artificial: sistemas expertos.* Alianza, Madrid, 1986.

CURTIS, M. W. A Turing machine simulator. *Journal of the ACM,* 12, 1, (enero 1965), pp. 1-13.

DAHL, O. I., Dijkstra, E. W. y HOARE, C. A. R. *Notes on structured programming.* Academic Press, 1972.

DATAMATION. Revolution in programming. *Datamation,* Dic. 1973. Número dedicado preferentemente al tema de la programación estructurada.

DAVIS, M. y PUTNAM, H. A computing procedure for quantification theory. *Journal of the ACM,* 7, 3 (julio 1960), pp. 201-215.

DE MORGAN, A. *Formal logic.* Londres, 1847.

DEAÑO, A. *Introducción a la lógica formal.* Alianza, Madrid, 1974. (3.ª ed., reimpr. 1986).

DELGADO KLOOS, C. *Simulador de Máquina de Turing.* Proyecto Fin de Carrera, E.T.S.I. Telecomunicación, Madrid, Sep. 1978.

DENNING, P. J., DENNIS, J. B. y QUALITZ, J. E. *Machines, languages and computation.* Prentice-Hall, Englewood Cliffs, N. J., 1978.

DÍAZ CORT, J., *Complejidad concreta y complejidad abstracta de algoritmos: Un panorama actual.* IV Escuela de Verano de Informática, Asociación Española de Informática y Automática, Granada, 1982.

DÍEZ MEDRANO, J., *Complejidad.* Trabajo para la asignatura de Fundamentos de Ordenadores I, Escuela Técnica Superior de Ingenieros de Telecomunicación, Madrid, junio 1984.

DIJKSTRA, E. W. Guarded commands, non-determinacy and the formal derivation of programs. *Comm. ACM,* 18 (agosto 1975), pp. 453-458.

DIJKSTRA, E. W., *A discipline of programming.* Prentice-Hall, N. J., 1976.

DUBOIS, P. y PRADE, H. *Théorie des possibilités. Application à la représentation des connaissances en informatique.* Masson, París, 1985.

DUDA, R. O., HART, P. E., NILSSON, N. J. y SUTHERLAND, G. Semantic networtk representation in rule-based inference systems. En D. A. Waterman y F. Hayes-Roth (Eds.): *Pattern-directed inference systems.* Academic Press, N. Y., 1978, pp. 203-221.

EVEY, J. *The theory and application of pushdown store machines.* (Tesis Doctoral). Harvard University, Cambridge, Mass., 1963.

FERRATER, J. y LEBLANC, H. *Lógica matemática.* Fondo de Cultura Económica, México, 1955 (2.ª ed., 1962).

FLOYD, R. W. On ambiguity in phrase structure languages. *Comm. of ACM,* 5 (1962), 10, 526-534.

FLOYD, R. W. The syntax of programming languages-A survey, *I.R.E. Trans. Electronic Computers,* 14, 1964, 4, 346-353.

FLOYD, R. W. Assigning meaning to programs. En J. T. Schwartz (ed.): *Mathematical aspects of computer science.* Proc. Symp. Amer. Math. Soc., 1967, pp. 19-33.

FRENKEL, K. Piecing together complexity, and complexity and parallel processing: an interview with Richard Karp. *Comm. ACM,* 29, 2 (febrero 1986), pp. 110-117.

GAMELLA, M. (ed.) *La tecnología del software. Temática y situación en España.* Fundesco, Madrid, 1985.

GARIJO, M. *et al. Tema 10: Algoritmos y complejidad.* Apuntes complementarios para la asignatura Fundamentos de la Programación, Escuela Técnica Superior de Ingenieros de Telecomunicación, Madrid, 1986.

GILB, T. Laws of unreliability. *Datamation,* vol. 21, N. 3, marzo 1975.

GILBERT, W. J. *Modern algebra with applications.* John Wiley, N. Y., 1976.

GINSBURG, S. *The mathematical theory of context-free languages.* McGraw-Hill, N. Y., 1966.

GINSBURG, S. Lectures on context-free languages. En Arbib, M. A. (ed.): *Algebraic theory of machines, languages, and semigroups.* Academic Press, N. Y., 1968.

GLORIOSO, R. M. y COLÓN, F. C. *Engineering intelligent systems.* Digital Press, Bedford, Mass., 1980.

GÖDEL, K., Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I (sobre proposiciones formalmente indecidibles de los Principia Mathematica y sistemas relacionados). *Monatshefte für Mathematik und Physik,* 38, 1931, pp. 173-98.

GOLDSCHLAGER, L., y LISTER, A. *Computer science: a modern introduction,* Prentice-Hall, N. J., 1982.

GONDRAN, M. *Introduction aux systèmes experts.* Eyrolles, París, 1984.

GORDON, M. J. C. *The denotational description of programming languages.* Springer-Verlag, N.Y., 1979.

GRIES, D. *Compiler construction for digital computers.* Wiley, N.Y., 1971 (Traducción de F. J. Sanchís: *Construcción de compiladores.* Paraninfo, Madrid, 1975).

GRIES, D. *The science of programming.* Springer-Verlag, N. Y., 1981.

GROGONO, P. *Programming in Pascal.* Addison-Wesley, Reading, Mass., 1980.

GROSS, M. y LENTIN, A. *Notions sur les grammaires formelles.* Gautier-Villars, París, 1967. (Traducción: Tecnos, Madrid, 1976).

GUPTA, M. M., SARIDIS, G. N. y GAINES, B. R. (eds.). *Fuzzy automata and decision processes.* North-Holland, Amsterdam, 1977.

HAAK, S. *Deviant logic.* Cambridge University Press, 1974.

HAAK, S. *Philosophy of logics.* Cambrige University Press, 1978.
HALSTEAD, M. H. *Elements of software science,* Elsevier, N. Y., 1977.
HARRISON, M. A. *Introduction to schwitching and automata theory.* McGraw-Hill, N. Y., 1965.
HARRISON, M. A. *Introduction to formal language theory.* Addison Wesley, Reading, Mass. 1978.
HAYES, P. J. In defense of logic. *Proc. 5th Int. Joint Conf. Artif. Intell.* (1977), pp. 559-565.
HAYES, P. J. The logic of frames. En *The frame reader.* De Guyter, Berlín, 1979.
HAYES-ROTH, F. The knowledge-based expert system: a tutorial. *Computer,* 17,9 (septiembre 1984a), pp. 11-28.
HAYES-ROTH, F. Knowledge-based expert systems. *Computer,* 17, 10 (octubre 1984b), pp. 263-273.
HAYES-ROTH, F., WATERMAN, D. A. y LENAT, D. B. *Building expert systems.* Addison-Wesley, Reading, Mass., 1983.
HELMER, E. C. R. *The logic of programming.* Prentice-Hall, Englewood Cliffs, N. J., 1984.
HENNIE, F. *Introduction to computability.* Addison-Wesley, Reading, Mass., 1977.
HILL, F. y PETERSON, E. *Introduction to switching theory and logical design.* Wiley, N. Y., 1968. (Traducción de H. Corona: *Teoría de la conmutación y diseño lógico.* Limusa, México, 1978).
HIRST, G. *Anaphora in natural language understanding: a survey.* Springer-Verlag, Berlín, 1981.
HOARE, C. A. R. An axiomatic approach to computer programming. *Comm. ACM,* 12 (octubre 1969), pp. 576-580, 583.
HOPCROFT, J. E. Máquinas de Turing. *Investigación y Ciencia,* 94, julio 1984, pp. 8-19.
HOPCROFT, J. E. y ULLMAN, J. D. *Formal languages and their relation to automata.* Addison-Wesley, Reading, Mass., 1969.
HOPCROFT, J. E. y ULLMAN, J. E. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Reading, Mass., 1979.
HOROWITZ, E. *Fundamentals of programming languages.* Springer-Verlag, Berlín, 1984.
HUFFMAN, D. A. The synthesis of sequential switching circuits. *Journal Franklin Institute,* 257 (1954), 3, pp. 161-190, 4, pp. 275-303.
HUNT, E. B. *Artificial Intelligence.* Academic Press, 1975.

JACKSON, M. J. *Principles of program design.* Academic Press, Londres, 1975.
JENSEN, K. y WIRTH, N. *Pascal user manual and report. ISO Pascal standard* (3.ª ed.). Springer-Verlag, N. Y., 1985.
JONES, C. F. *Systematic software development using VDM.* Prentice-Hall, Englewood Cliffs, N. J., 1986.

KARNAUGH, M. The map method for synthesis of combinational logic circuits. *Trans. AIEE,* 72, 9 (1953), pp. 593-599.
KARP, R. M. Combinatorics, complexity, and randomness, *Comm. ACM,* 29, 2 (febrero 1986), pp. 98-109.
KAUFMANN, A. *Introduction à la théorie des sous-ensembles flous à l'usage des ingénieurs.* (3 vol.). Masson, París, 1973-75.
KELLER, A. M. *A first course in computer programming using Pascal.* McGraw-Hill, N. Y., 1982 (Traducción de G. León y J. M. Vela: *Programación en Pascal.* McGraw-Hill, Madrid, 1983).
KLEENE, S. *Introduction to metamathematics.* North-Holland, Amsterdam, 1952. (Traducción de M. Garrido: *Introducción a la metamatemática.* Tecnos, Madrid, 1974).
KLEENE, S. C. Representation of events in nerve nets and finite automata. En *Automata*

studies, Princeton Univ. Press, Princeton, N. J., 1956.
KNUTH, D. E. Algoritmos. *Investigación y Ciencia,* núm. 9, junio 1977, pp. 42-53.
KNUTH, D. E. y PARDO, L. T. The early development of programming languages. En N. Metroplis, J. Howlet y G. C. Rota (eds.): *A history of computing in the twentieth century.* Academic Press, N. Y., 1980, pp. 197-213.
KOHAVI, Z. *Switching and finite automata theory.* McGraw-Hill, N. Y., 1970.
KOWALSKI, R. Predicate logic as a programming language. *Proc. IFIP 74.* North Holland, Amsterdam, 1974, pp. 569-574.
KOWALSKI, R. *Logic for problem solving.* North-Holland, New York, 1979. (Traducción de J. A. Calle: *Lógica, programación e inteligencia artificial.* Díaz de Santos, Madrid, 1986).
KURODA, S. Y. Classes of languages and linear-bounded automata. *Information and Control,* 7 (1964), 2, 114-125.

LEWIS, C. I. y LANGFORD, C. H. *Symbolic logic.* The Century Comp., N. Y. 1932 (2.ª ed., Dover Publ., N. Y., 1959).
LINGER, R., MILLS, H. y WITT, B. *Structured programming theory and practice.* Addison Wesley, Reading, Mass., 1979.
LUKASIEWICZ, J. On 3-valued logic. (1920). Reproducido en McCall, S. (ed.): *Polish logic.* Oxford University Press, 1967.
LUKASIEWICZ, J. Many-valued systems of propositional logic. (1930). Reproducido en McCall, S. (ed.): *Polish Logic.* Oxford University Press, 1967.
LURIA, A. R. *Cerebro y lenguaje.* Fontanella, Barcelona, 1974a.
LURIA, A. R. *Cerebro en acción.* Fontanella, Barcelona, 1974b.

MADNICK, S. E. y DONOVAN, J. J. *Operating systems.* McGraw-Hill, N. Y., 1974.
MANDADO, E. *Sistemas electrónicos digitales.* (3.ª ed.). Marcombo, Barcelona, 1977.
MANNA, Z., *Mathematical theory of computation.* McGraw-Hill, N. Y., 1974.
MANNA, Z. y PNUELI, A. The modal logic of programs. *Proc. 6th Int. Colloquium on Automata, Languages and Programming.* Springer Verlag (Lecture Notes in Computer Science, Vol. 71), 1979, pp. 385-411.
MANNA, Z. y PNUELI, A. Verification of concurrent programs: the temporal framework. En Boyer, R. S. y Moore, J. S. (eds.): *The correctness problem in computer science.* Academic Press, New York, 1981, pp. 215-273.
MANNA, Z., y WOLPER, P. Synthesis of communicating processes from temporal logic. *ACM Trans. Progr. Lang. and Syst.,* 6 (1984), pp. 68-93.
MARTIN, M. A. y FATEMAN, R. J. The MACSYMA system. *Proc. 2nd Symp. Symbolics and Algebraic Manipulation.* Los Angeles, Ca., 1971, pp. 59-75.
MCCABE, T. J. A complexity measure, *IEEE Trans. Softw. Eng.,* diciembre 1976, pp. 308-320.
MCCALLA, G. y CERCONE, N. (eds.). *Computer,* 16, 10 (octubre 1983). (Número monográfico con 16 artículos sobre representación del conocimiento.)
MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. *Comm. ACM,* 3, 4 (1960), pp. 184-195.
MCCARTHY, J. A basis for a mathematical theory of computation. *Proc. IFIP Congress 62,* North Holland, Amsterdam, 1963.
MCCARTHY, J. y PAINTER, J. Correctness of a compiler for arithmetic expressions. En J. T. Schwartz (ed.): *Mathematical aspects of computer science.* American Mathematical Society, 1967, pp. 33-41.
MCCULLOGH, W. y PITTS, W. A logical calculus of the ideas inmanent in nervous activity. *Bull. Math. Biophysics,* 5 (1943), pp. 115-133.
MCDERMOTT, D. A temporal logic for reasoning about plans and actions. *Cognitive Science,* 6

(1982), pp. 101-155.

MCDERMOTT, D. y DOYLE, J. Non-monotonic logic. *Artificial Intelligence,* 13 (1980), pp. 27-39.

MCNAUGHTON, R. y YAMADA, H. Regular expressions and graphs for automata. *IRE Trans. Elec. Comp.,* EC-9 (1960), pp. 39-47.

MEAD, C. y CONWAY, L. *Introduction to VLSI systems.* Addison-Wesley, Reading, Mass., 1980.

MEALY, G. H. A method for synthetising sequential circuits. *Bell System Tech. J.,* 34 (1955), pp. 1.045-1.079.

MICHALSKI, R. S., CARBONELL, J. G. y MITCHELL, T. M. *Machine learning: an artificial intelligence approach.* Springer-Verlag, Berlín, 1983.

MICHALSKI, R. S., CARBONELL, J. G. y MITCHELL, T. M. *Machine learning: an artificial intelligence approach, vol. II.* Springer-Verlag, Berlín, 1986.

MILLS, H. D. The new math of computer programming. *Comm. ACM,* 18, 1 (enero 1975).

MINSKY, M. A. framework for representing knowledge. En P. Winston (ed.): *The psychology of computer vision.* McGraw-Hill, N. Y., 1975, pp. 211-277.

MOMPIN, J. (coord.) *Inteligencia artificial. Conceptos, técnicas y aplicaciones.* Marcombo, Barcelona, 1987.

MOORE, E. F. Gendanken-experiments on sequential machines. *Automata studies: Annals of mathematical studies,* No. 34, Princeton Univ. Press, Princeton, N. J., 1956, pp. 129-153.

MOORE, R. C. A formal theory of knowledge and action. En Hobbs, J. R. y Moore, R. C. (eds.): *Formal theories of the common sense world.* Ablex, Norwood, N. J., 1984.

MUÑOZ, E. (Coord.) *Circuitos electrónicos digitales II.* Serv. Public. E.T.S.I.T.M., Madrid, 1983.

MYHILL, J. *Linear bounded automata.* WADD Tech. Note 60-165, Wright Patterson Air Force Base, Ohio, 1960.

NAGLE, H. T., CARROL, B. D. y IRWIN, J. D. *An introduction to computer logic.* Prentice Hall, Englewood Cliffs, N. J., 1975.

NARENDRA, K. S. y THATHACHAR, M. A. L. Learning automata- a survey. *IEEE Trans. Syst, Man, and Cyb.,* SMC-4 (julio 1974), pp. 323-334.

NAU, D. S. Expert computer systems. *Computer,* 16, 2 (febrero 1982), pp. 63-85.

NAUR, P. (ed.). Revised report on the algorithmic language ALGOL 60. *Comm. ACM.,* 6, 1 (1963), pp. 1-17.

NAYLOR, C. *Build your own expert system.* Sigma Technical Press, Cheshire, U.K., 1983. (Traducción de G. Fernández: *Construya su propio sistema experto.* Díaz de Santos, Madrid, 1986).

NILSSON, J. J. *Principles of artificial intelligence.* Springer Verlag, Berlín, 1982. (Traducción de J. Fernández-Biarge: *Principios de inteligencia artificial.* Díaz de Santos, Madrid, 1987).

OBERMAN, R. M. M. *Disciplines in combinational and sequential circuit design.* McGraw-Hill, N. Y., 1970.

OETTINGER, A. G. Automatic syntactic analysis and the pushdown store. *Proc. Symp. Applied Math.,* Amer. Math. Soc., Providence, Rhode Island, 1961.

OLLONGREN, A. *Definition of programming languages by interpreting automata.* Academic Press, N. Y., 1974.

OTT, G. y FEINSTEIN, N. Design of sequential machines from their regular expressions. *Journal ACM,* 8, 4 (1961), pp. 585-600.

PAUKER, S. C., GORRY, G. A., KASSIRER, J. P. y SCHWARTZ, W. B. Towards the simulation

of clinical consultation. Taking a present illness by computer. *American Journal of Medicine,* 60 (1976), pp. 981-996.

PETERSON, J. L. Petri nets. *Computing Surveys,* 9, 3 (septiembre 1977), pp. 223-251.

PETRI, C. A. *Kommunication mit automaten.* Univ. Bonn, 1962. (Traducido en *Supplement I to Technical Report RADC-TR-65-377, vol. I,* Rome Air Development Center, Griffis Air Force Base, New York, 1965).

POPLE, H. R., MYERS, J. D. y MILLER, R. A. DIALOG: A model of diagnostic logic for internal medicine. *Proc. 4th Int. Joint Conf. Artif. Int.* Tbilisi (URSS), 1975, pp. 848-855.

POST, E. Formal reduction of the general combinatorial problem. *American Journal of Mathematics,* 65 (1943), pp. 197-268.

PRADÉ, H. A computational approach to approximate and plausible reasoning whith applications to expert systems. *IEEE Trans. Pattern Analysis and Mach. Intell.,* PAMI-7, 3 (mayo 1985), pp. 260-283.

PYSTER, A. B. *Compiler design and construction.* Van Nostrand Reinhold, N. Y., 1980.

QUILLIAN, M. R. Semantic memory. En M. Minsky (ed.): *Semantic information processing.* M.I.T. Press, Cambridge, Mass., 1968, pp. 354-402.

RABIN, M. O. Probabilistic automata. *Information and control,* 6, 3 (1963), pp. 230-245.

RABIN, M. O. y SCOTT, D. Finite automata and their decision problems. *IBM Journal Res. Dev.,* 3 (1959), 2, 114-125.

RANDELL, B. y RUSSELL, L. J. *Algol 60 implementation.* Academic Press, N. Y., 1964.

ROBINSON, J. A. A machine-oriented logic based on the resolution principle. *Journal of the ACM,* 12, 1 (enero 1965), pp. 23-41.

ROBINSON, J. A. *Logic: form and function.* Edinburg University Press, 1979.

SÁEZ VACAS, F. Guía para un análisis estructurado de la programación estructurada. Inforprim 1975. *Proceso de Datos,* núm. 54, enero-febrero 1976.

SANCHÍS, F. J. y GALÁN. *Compiladores. Teoría y construcción.* Paraninfo, Madrid, 1986.

SANCHÍS, F. J. y MORALES, A. *Programación con el lenguaje Pascal* (2.ª ed.). Paraninfo, Madrid, 1981.

SCALA, J. J. y MINGUET, J. M. *Informática II. Unidades didácticas 2 y 3.* Universidad Nacional de Educación a Distancia, 1974.

SCHANK, R. C. y ABELSON, R. P. *Scripts, plans, goals, and understanding.* Lawrence Erlbaum, Hillsdale, N. J., 1977.

SCHNEIDER, G. M. y BRUELL, S. C. *Advanced programming and problem solving with Pascal.* John Wiley, N. Y., 1981.

SCOTT, D. y STRACHEY, C. Towards a mathematical semantics for computer languages. *Proc. Symp. Computers and Automata,* 1971, y en *Technical Monograph* PRG-6, Oxford Univ. Comp. Lab., pp. 19-46.

SHAFER, G. *A mathematical theory of evidence.* Princeton University Press, Princeton, N. J., 1976.

SHANNON, C. E. The synthesis of two-terminal switching circuits. *Bell System Tech. J.,* vol. 28 (1949), pp. 59-98.

SHANNON, C. E. *A symbolic analysis of relay and switching circuits.* Van Nostrand, N. Y., 1954.

SHEFFER, H. M. A set of five independent postulates for boolean algebras, whith application to logical constants. *Trans. Amer. Math. Soc.,* 14 (1913).

SHOOMAN, M. L. *Software engineering.* McGraw-Hill, International Student Edition, Auckland, 1983.

SHORTLIFFE, E. H. *Computer-based medical consultations.* Elsevier, N. Y., 1976.

SILVA, M. *Las redes de Petri: en la automática y la informática.* AC, Madrid, 1985.
SINGH, J. *Teoría de la información, del lenguaje y de la cibernética* (2.ª ed.). Alianza, Madrid, 1976.
SKOLEM, T. Uber die mathematishe logik. *Norsk matematisk tidsskrift,* 10 (1928), pp. 125-142. Versión traducida al inglés en van Heijenoort, J. (ed.): *From Frege to Gödel.* Harvard University Press, 1967, pp. 508-524.
SOWA, J. F. *Conceptual structures: information processing in mind and machine.* Addison-Wesley, Reading, Mass., 1984.
STEWART, I. *Conceptos de matemática moderna.* Alianza, Madrid, 1977.
STONE, H. S. *Introduction to computer organization and data structures.* McGraw-Hill, N. Y., 1972.
STOY, J. E. *Denotational semantics: the Scott-Strackey approach to programming language theory.* MIT Press, Cambridge, Mass., 1977.

TABOURIER, Y., ROCHFELD, A. y FRANK, C. *La programmation structurée en informatique.* Les Editions d'Organisation, París, 1975.
TENNANT, H. *Natural language processing.* Petrocelli, New York, 1981.
TENNENT, R. D. The denotational semantics of programming languages. *Comm. ACM,* 19, 8 (agosto 1976), pp. 437-453.
TENNY, T. Structured programming in Fortran. *Datamation,* julio 1974.
TRAKHTENBROT, B. A. Algoritmos. En *Perspectivas de la revolución de los computadores.* Alianza Universidad, 1975, pp. 108-130. Edición inglesa: Prentice-Hall, 1970.
TRIGOBOFF, M. y KULIKOWSKI, C. A. IRIS: a system for the propagation of inferences in a semantic net. *Proc. 5th Int. Joint Conf. Artif. Intell.,* Cambridge, 1977, pp. 274-280.
TSETLIN, M. L. Sobre el comportamiento de autómatas finitos en entornos aleatorios (en ruso). *Automctika i telemekhanika,* 22, 10 (octubre 1961), pp. 1.345-1.354.
TUCKER, A. B. *Programming languages* (2.ª ed.) Mc-Graw-Hill, N. Y., 1986. (Traducción de J. M. Troya: *Lenguajes de programación.* McGraw-Hill, Madrid, 1987).
TURNER, R. *Logics for artificial intelligence.* Ellis Horwood, Chichester (UK), 1984.

ULLMAN, J. D. *Principles of database systems.* Pitman, Londres, 1982.

VARSHAVSKII, V. I. VORONTSOVA, I. P. Sobre el comportamiento de autómatas estocásticos con estructura variable (en ruso). *Automatika i telemekhanika,* 24, 3 (marzo 1963), pp. 353-360.
VEITCH, E. W. A chart method for simplifying truth funtions. *Proc. ACM,* mayo 1952, pp. 127-133.
VON NEUMANN, J. Probabilistic logic and the synthesis of reliable organisms from unreliable components. *Automata studies,* Princeton Univ. Press, Princeton, N. J., 1956.

WANG, H. Juegos, lógica y computadores. En *Computadoras y computación.* Ed. R. R. Fenichel y J. Weizenbaum. Blume, 1974, pp. 150-158. Edición inglesa: W. H. Freeman. (El artículo de Wang se publicó en 1965 en el *Scientific American*).
WARNIER, J. D. *Programación lógica.* Tomos I y II, Editores Técnicos Asociados, Barcelona, 1973.
WEISSMAN, C. *LISP 1.5 primer.* Dickenson, Encino, Ca., 1967.
WEISS, S. M. y KULIKOWSKI, C. A. EXPERT: A system for developing consultations models. *Proc. 6th Int. Joint Conf. Artif. Int.,* Tokyo, 1979, pp. 942-947.
WEISS, S. M., KULIKOWSKI, C. A. y SAFIR, A. Glaucoma consultation by computer. *Computers in Biology and Medicine,* 8 (1978), pp. 25-40.

WHITEHEAD, A. N. y RUSSELL, B. *Principia Mathematica.* Cambridge, 1910 (vol. I), 1912 (vol. II), 1913 (vol. III).

WINOGRAD, S. y COWAN, J. D. *Reliable computation in the presence of noise.* M. I. T. Press, Cambridge, Mass., 1963.

WINSTON, P. H. y HORN, B. K. P. *LISP.* Addison-Wesley, Reading, Mass., 1984.

WIRTH, N. The programming langage Pascal. *Acta Informatica,* 1, 1 (1971), 35-36.

WIRTH, N. *Algorithms + data structures = programs.* Prentice-Hall, 1976 (Traducción de A. Alvarez y J. Cuena: *Algoritmos + estructuras de datos = programas.* Ediciones del Castillo, Madrid, 1980, 4.ª impr., 1985).

WIRTH, N. Algoritmos y estructuras de datos. *Investigación y Ciencia,* núm. 98, noviembre, 1984, pp. 24-35.

YOURDON, E., *Techniques of program structure and design.* Prentice-Hall, Englewood Cliffs, N. J., 1975.

ZADEH, L. A. Fuzzy sets. *Information and Control,* 8 (1965), pp. 338-353.

ZADEH, L. A. Fuzzy algorithms. *Information and control,* 12 (1968), pp. 94-102.

ZADEH, L. A. Biological applications of the theory of fuzzy sets and systems. En L. D. Proctor (ed.): *Biocybernetics of the central nervous system.* Little, Brown and Co., Boston, Mass., 1969, pp. 199-212.

ZADEH, L. A. Outline of a new approach to the analysis of complex systems and decision process. *IEEE Trans. Systems, Man, and Cyb.,* SMC-3 (1973), pp. 28-44.

ZADEH, L. A. Fuzzy logic and its applications to approximate reasoning. *Proc. IFIP Congress Information Processing 74,* North-Holland, Amsterdam, 1974, pp. 591-594.

ZADEH, L. A. Fuzzy sets as a basis for a theory of possibility. *Fuzzy sets and systems,* 1 (1978), pp. 3-28.

ZADEH, L. A., Fu, K. S., TANAKA, K. y SHIMURA, M. (eds.). *Fuzzy sets and their applications to cognitive and decision process.* Academic Press, N. Y., 1975.