

Borrador de unos apuntes preliminares¹ para Lenguajes, Gramáticas y Autómatas

Rafael C. Carrasco Jiménez
Jorge Calera Rubio
Mikel L. Forcada Zubizarreta

14 de septiembre de 1998

¹De estas notas no se hacen responsables ni siquiera sus autores. Se consultan bajo la responsabilidad del consumidor, que deberá pagarse todos los gastos médicos que su lectura ocasione.

Índice General

Presentación	iii
Estructura del libro	iv
1 Lenguajes y computadores	1
1.1 Distintos tipos de programas	3
1.2 Modelos computacionales y modelos de inteligencia	7
1.3 Representaciones de un AFD	8
1.4 Alfabetos y lenguajes	10
1.5 Autómata finito determinista como clasificador	12
1.6 El autómata finito determinista como traductor	16
1.7 Deterministas, indeterministas y estocásticos	18
1.7.1 Autómatas finitos indeterministas	18
1.7.2 Equivalencia AFL-AFD	20
1.7.3 Autómatas finitos estocásticos	23
2 Lenguajes regulares	27
2.1 Expresiones regulares	27
2.1.1 Algunas propiedades de las ER	30
2.2 Equivalencia ER-AF	31
2.3 Operaciones con conjuntos regulares	39
3 Construcción de autómatas finitos	45
3.1 Lema de bombeo	45
3.2 Algoritmos y decidibilidad	47
3.3 Minimización de autómatas finitos	49
3.3.1 Teorema de Myhill y Nerode	51
3.3.2 Algoritmos de minimización	53
4 Gramáticas	57
4.1 Introducción	57
4.2 Definición y clasificación de las gramáticas	59
4.3 Gramáticas regulares	61
4.3.1 Gramática regular asociada a un autómata	61
4.3.2 Autómata asociado a una gramática regular	62

4.4	Gramáticas independientes del contexto (GIC)	63
4.4.1	Derivación y árbol de derivación	64
4.4.2	Ambigüedad	65
4.4.3	Recursividad	68
4.4.4	El análisis sintáctico	71
4.4.5	Simplificación de una GIC	76
4.4.6	Formas normalizadas	82
5	Lenguajes independientes del contexto	89
5.1	Operaciones con LIC	89
5.2	Lema de bombeo para LIC	91
5.3	Algoritmos	94
6	Autómatas con pila y analizadores sintácticos	97
6.1	Definiciones	97
6.2	Autómata de pila para una GIC	100
6.3	Gramática correspondiente a un AP	102
6.4	Analizadores sintácticos	103
A	Nociones básicas de teoría de conjuntos	107
A.1	Correspondencias y relaciones	108
A.2	Cardinal. Conjuntos infinitos	111
A.2.1	Demostración del teorema de Cantor	113
Bibliografía		115

Presentación

La mayoría de los textos dedicados a la *teoría matemática de lenguajes*¹ presentan los elementos básicos de dicha teoría desde un punto de vista fundamentalmente matemático. Esto es, su estructura lógica se basa en una serie de definiciones y teoremas desarrollados con gran rigor y consistencia, dejando en un segundo plano la conexión de estos conceptos con los problemas prácticos y con su evolución histórica. La opción de presentar una materia en la forma elaborada a la que se ha llegado después de décadas de investigación, revisión de los conceptos fundamentales y estructuración lógica de la teoría resulta conveniente desde el punto de vista del rigor científico, pero dificulta el acercamiento a la materia de las personas cuyo interés se centra en los aspectos prácticos de la ciencia. Este es, con frecuencia, el caso de los ingenieros, que perciben su estudio como un ejercicio muy abstracto y de escasa relevancia para los problemas tecnológicos.

Por otra parte, la ingeniería informática es uno de los ámbitos en los que mayor importancia y repercusión tiene la teoría matemática de los lenguajes. Por un lado, gran parte de las tareas que se realizan en este campo, como la implementación de compiladores y analizadores sintácticos, el desarrollo de lenguajes de programación, etc., requieren un conocimiento profundo de la teoría de lenguajes. Por otro, muchos de los problemas computacionales que concitan un mayor interés en la actualidad pueden plantearse también en términos lingüísticos. Algunos de forma inmediata, como la traducción entre idiomas. Otros, de forma indirecta: por ejemplo, el reconocimiento de objetos mediante una cámara puede entenderse también como la clasificación de las cadenas de símbolos que codifican la imagen, un problema análogo al de identificar palabras de un lenguaje.

Por todo ello, hemos considerado conveniente presentar los elementos básicos de la teoría de lenguajes de una forma en parte diferente a la tradicional: a partir de situaciones habituales para las personas que trabajan en el ámbito de la informática. Así, hemos intentado que situaciones que se presentan frecuentemente en las tareas de programación motiven la introducción de los conceptos y que la formalización de éstos se produzca sólo a poste-

¹Hemos optado por esta denominación frente a la más frecuente, pero también algo más oscura de *teoría formal de lenguajes*. La palabra formal sugiere un tratamiento centrado en el aspecto y no en la esencia.

riori. Este método requiere encontrar ejemplos adecuados que conduzcan con la suficiente rapidez a los conceptos, lo que no siempre resulta sencillo. Por ello, consideramos esta obra como un trabajo de investigación abierto a aportaciones y mejoras en el futuro. Desde el punto de vista pedagógico, este método, que podríamos calificar como “más inductivo que deductivo”, requiere generalmente una inversión de tiempo mayor en la presentación de los temas en el aula. No obstante, nuestra experiencia docente nos ha demostrado que el esfuerzo se ve recompensado con una mayor persistencia del aprendizaje.

Estructura del libro

El esquema básico que hemos utilizado es el mostrado en la figura 1. El objetivo que buscamos es introducir, en la medida de lo posible, los conceptos de la teoría de lenguajes a medida que van siendo necesarios para resolver problemas cada vez más complicados.

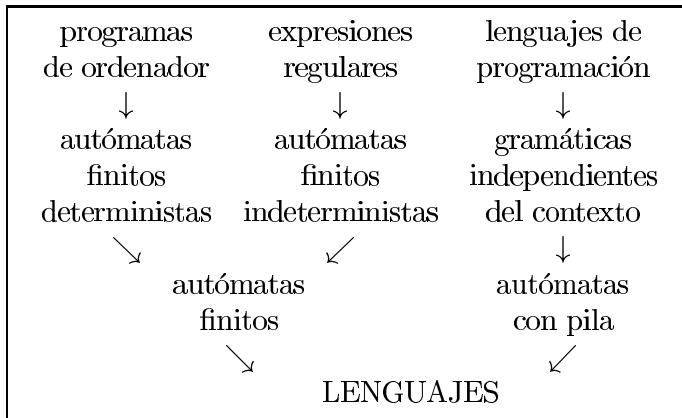


Figura 1: Esquema básico del libro.

Capítulo 1

Lenguajes y computadores

Los programas de ordenador que estamos acostumbrados a utilizar pueden describirse como secuencias de instrucciones (lo que técnicamente se denomina *algoritmo*) que operan sobre los datos de *entrada* para producir la *salida* deseada. Durante su ejecución, el programa utiliza normalmente distintos registros o *unidades de memoria* para almacenar resultados intermedios (véase la figura 1.1). De hecho, una de las características fundamentales de un programa es la cantidad de memoria que requiere durante su ejecución. Dado que la capacidad de almacenamiento de los ordenadores actuales, aunque inmensa, es siempre limitada, un programa diseñado en cualquier lenguaje de programación puede siempre clasificarse en uno de los dos grupos siguientes:

1. algoritmos que durante su ejecución requieren una cantidad de memoria que puede ser establecida *a priori*, es decir, sin conocer la entrada del programa;
2. algoritmos para los que no es posible establecer un límite a la memoria utilizada pues, sea cual sea la cantidad de registros disponible, existe alguna entrada para la que el funcionamiento correcto del algoritmo requiere un número aún mayor de unidades de memoria.

Los programas del primer grupo nos servirán para ilustrar el concepto de autómata finito. De forma intuitiva, los autómatas finitos pueden definirse como sistemas cuyo funcionamiento se puede describir en términos de un número de estados finito. Por otra parte, los algoritmos del segundo tipo aparecen con relativa frecuencia en programación. Entre los ejemplos típicos, se encuentran:

- programas que utilizan memoria dinámica, esto es, memoria que se crea (o se define) según las necesidades del programa;
- programas que utilizan llamadas recursivas, lo que obliga a almacenar en una pila las direcciones de retorno de las llamadas.

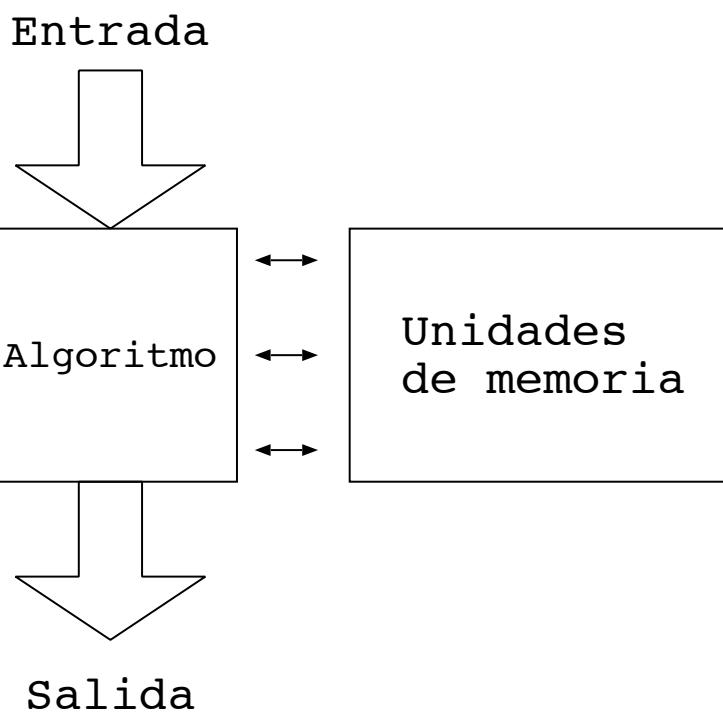


Figura 1.1: Esquema del funcionamiento de un programa.

En muchos casos, los programas que utilizan memoria dinámica o llamadas recursivas no son autómatas finitos: no está garantizado su funcionamiento correcto en todos los casos ya que pueden provocar, durante su ejecución, mensajes de error del tipo “memory exhausted”, “stack overflow”, u otros más crípticos como “segmentation fault” que apuntan a un agotamiento de los recursos de la máquina.

Por contra, los programas que son un autómata finito pueden implementarse utilizando memoria estática, esto es, memoria cuya capacidad se determina en el momento de la compilación y por tanto, no varía en las distintas ejecuciones del programa. El concepto de autómata finito es uno de los más importantes en teoría de la computación y de su estudio nos ocuparemos a lo largo del presente capítulo.

1.1 Distintos tipos de programas

Con el fin de ilustrar el concepto de autómata finito determinista, se pide al lector o lectora que resuelva los siguientes

Ejercicios

1.1 Construye un programa que calcula el cociente de una cadena numérica binaria por un número natural dado n . Por ejemplo, si $n = 3$ y la entrada es 1101, la salida será 0100.

1.2 Elabora un programa que invierte el sentido de la cadena de entrada. Por ejemplo, si la entrada es LAMINA la salida es ANIMAL.

```
programa divide_n
comienza algoritmo
  r = 0      (resto)
  haz
    s = valor(siguiente-símbolo)
    escribe (n * r + s)/n      (cociente)
    r = (n * r + s) mod n     (nuevo resto)
  repite haz
termina algoritmo
```

Figura 1.2: Programa que escribe la cadena numérica de entrada dividida por n . Nótese que sólo utiliza un registro r que permite almacenar un número entre 0 y n .

Una solución al primer ejercicio se presenta en forma de pseudo-código en la figura 1.2. Con el procedimiento allí descrito, el programa puede implementarse de forma sencilla sin que se necesite almacenar toda la cadena

de entrada. Basta con ir procesando ésta símbolo a símbolo, calculando el cociente y guardando el resto para sumárselo (multiplicado por n) a la siguiente lectura. Por tanto, un programa que disponga de un sistema de almacenamiento para guardar al menos $\log_2(n)$ dígitos binarios¹, que es el número mínimo necesario para almacenar una cifra entre 0 y n , es suficiente para realizar la tarea de dividir una cadena numérica por n . Por muy larga que sea la cadena de entrada, bastará con esperar el tiempo suficiente para obtener la salida, pero jamás se producirá el desbordamiento del programa. Todas las posibilidades han sido implícitamente tenidas en cuenta y el programador no debe preocuparse de incluir un aviso o mensaje de advertencia.

Resuelve ahora el siguiente

Ejercicio

1.3 Representa gráficamente los distintos estados de la memoria del programa `divide_3` para distintas entradas, así como los cambios que se producen entre ellos en función del símbolo que es procesado.

El resultado del ejercicio anterior puede representarse mediante un diagrama como el de la figura 1.3. En la próxima sección veremos que los

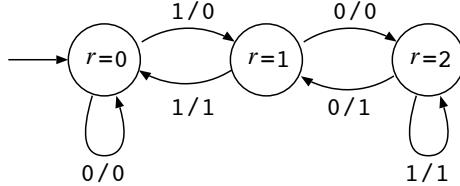


Figura 1.3: Representación gráfica del programa `divide_3`. Los valores sobre cada arco representan la cifra leída/escrita. El arco sin etiqueta distingue el estado inicial.

estados de la memoria utilizada pueden entenderse como los *estados* de un autómata y los cambios que se producen con la entrada pueden interpretarse como la llamada *función de transición* del autómata. En particular, la representación mediante un grafo etiquetado es una de las más habituales para un autómata finito.

Retomamos ahora el ejercicio 1.2, cuya resolución permite optar entre:

1. leer la cadena completa y después escribir el resultado (la cadena invertida), con lo que es de esperar que para una cadena lo suficientemente larga se agoten las reservas de memoria, o
2. utilizar una función recursiva como en el código de la figura 1.4

```

programa función invierte_entrada
comienza algoritmo
    s = siguiente_símbolo
    si s ≠ cadena_vacía entonces
        invierte_entrada
        escribe s
    fin si
termina algoritmo

```

Figura 1.4: Programa que invierte la cadena de entrada utilizando una formulación recursiva.

En esta segunda opción, el desbordamiento se producirá cuando se agote la pila de almacenamiento de llamadas recursivas, lo cual es seguro que ocurrirá si la cadena es lo suficientemente larga. Por tanto, un programa correctamente escrito debe advertir antes de la ejecución de que existe un tamaño máximo del problema que puede resolver. Sin embargo, dicho tamaño dependerá del ordenador en el que se está ejecutando el programa. Por ello, si podemos suponer que no se presentarán cadenas tan largas que provoquen el desbordamiento, la imagen de un autómata finito con una pila de almacenamiento será un buen modelo para describir el comportamiento del ordenador.

En cualquier caso, podemos afirmar que la tarea del ejercicio 1.2 pertenece a un grupo de problemas más difíciles de resolver que los del tipo del ejercicio 1.1. Mientras que los últimos pueden ser tratados mediante una máquina de estados finitos, los primeros requieren una pila de almacenamiento virtualmente ilimitada.

Esta clasificación de problemas en función de su complejidad (medida ésta en términos de qué mecanismos son necesarios para resolverlo), tiene un análogo en la teoría matemática de lenguajes (véase la figura 1.5). Noam

tipo de lenguaje	denominación habitual	mecanismo de análisis
3	regular	autómata finito
2	independiente del contexto	autómata con pila
1	dependiente del contexto	sutómata lineal acotado
0	recursivo	máquina de Turing

Figura 1.5: Jerarquía de Chomsky para los lenguajes.

Chomsky estableció cuatro categorías diferentes de lenguajes (desde el tipo

¹En inglés, *bits*, denominación habitual de la unidad mínima de información.

0 al tipo 3) basándose en la complejidad de las gramáticas que definían estos lenguajes. El significado del término “complejidad de una gramática” será tratado más adelante. Sin embargo, podemos adelantar que cada categoría requiere un tipo de programas distinto para realizar el análisis sintáctico del lenguaje. En efecto, el proceso de análisis sintáctico, esto es, el procedimiento que permite establecer si una frase es correcta dada la definición gramatical del lenguaje, puede realizarse mecánicamente, por ejemplo, mediante un ordenador adecuadamente programado. El programa que es necesario para dicha tarea depende de la complejidad de la gramática, y a cada nivel de la jerarquía de Chomsky se le puede asociar un tipo de programa distinto. En particular, los lenguajes que pueden ser analizados mediante autómatas finitos forman la clase 3, mientras que los que pueden ser analizados mediante un autómata con pila constituyen la clase 2 y serán objeto de estudio en el capítulo 4. Los de tipo 1 no serán estudiados aquí con detalle. El análisis de los lenguajes de tipo 0 requiere el uso de una máquina de Turing. Una máquina de Turing consiste esencialmente en un autómata finito determinista al que se le ha incorporado una cinta como sistema de registro. Esta cinta está formada por un conjunto numerable² de casillas que pueden utilizarse durante la ejecución para escribir en ellas y para leer su contenido, de forma que los cambios de estado del autómata pueden depender del contenido de la casilla consultada en ese instante. En la práctica, una máquina de Turing es equivalente a un ordenador al que se le puede incorporar tantas unidades de memoria como haga falta durante la ejecución del programa.

Existe una relación de inclusión entre las clases definidas de este modo, tal y como queda reflejado en la figura 1.6. La relación de inclusión es además propia, es decir, cada clase es estrictamente más pequeña que la siguiente o, dicho de otra forma, existen lenguajes de cada clase que no pertenecen a las clases más pequeñas. Por ejemplo, no todos los lenguajes independientes del contexto son regulares. En este libro nos limitaremos al estudio de los lenguajes más sencillos (los de tipo 3 y tipo 2) que son los que mayor aplicación práctica tienen en la informática. Veremos más adelante que un simple argumento de conteo demuestra que deben existir lenguajes no recursivos. Tampoco estudiaremos la relevancia de este hecho. Los lectores interesados en las repercusiones filosóficas y prácticas de ésto encontrarán interesante el libro de R. Penrose, “The new Emperor’s mind”.

Los problemas que hemos considerado en los ejercicios de esta sección pueden entenderse como tareas de traducción: una cadena de símbolos de entrada debe traducirse en otra cadena de salida. Más adelante plantearemos tareas más sencillas en las que la salida se reduce a una respuesta binaria (cadena aceptada o rechazada) a las que llamaremos tareas de *clasificación* y tareas en las que no se proporciona ninguna entrada a las que llamaremos tareas de *generación de lenguajes* (véase la figura 1.7).

²véase 112 para una definición de conjunto numerable.

1.2. MODELOS COMPUTACIONALES Y MODELOS DE INTELIGENCIA

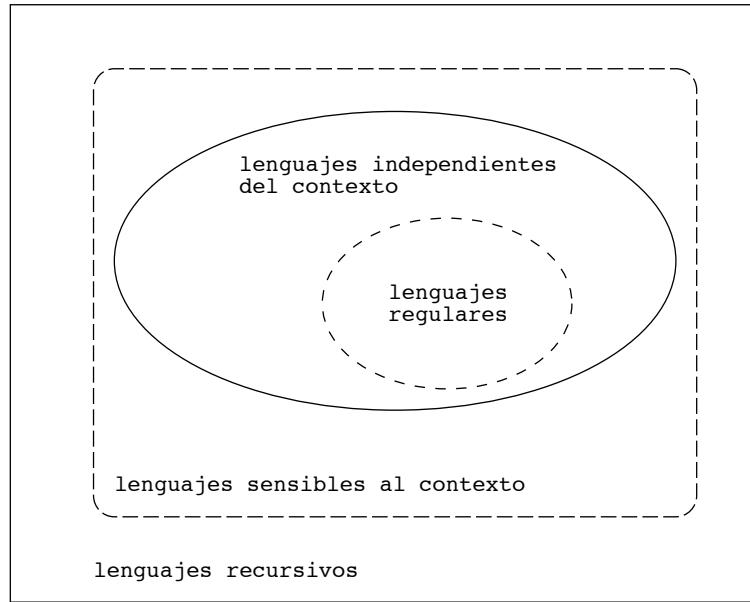


Figura 1.6: Relaciones de inclusión entre las clases de la jerarquía de Chomsky.

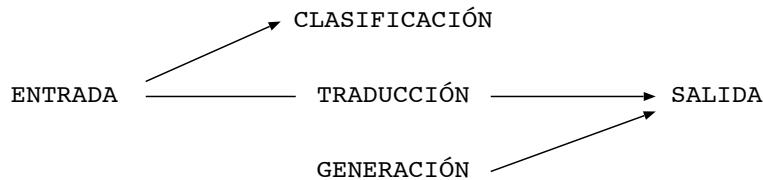


Figura 1.7: Distintos tipos de tarea según la relevancia de la entrada/salida

1.2 Modelos computacionales y modelos de inteligencia

Es importante destacar que sólo los problemas cuya resolución se puede plantear como un autómata finito determinista pueden ser resueltos completamente utilizando un ordenador, dado que su número de estados es enorme pero limitado. Esta limitación no es accidental, sino que se debe a la finalidad con que han sido diseñados: los ordenadores son las máquinas más deterministas construidas por el hombre y su fiabilidad reside, precisamente, en esta característica. Si bien las magnitudes físicas (como, por ejemplo, los potenciales eléctricos a los que se encuentra un elemento del circuito) pueden considerarse continuas, la aplicación de umbrales de activación reduce los resultados posibles a una lógica binaria: activado (valor uno) o desactivado

(valor cero). De esta forma, la respuesta a una determinada entrada es siempre la misma, sin que se vea afectada por la presencia de ruido o pequeñas imprecisiones en el funcionamiento de la circuitería. ¡No sería agradable que el resultado de un cálculo dependiese del momento del día en el que este se realiza! Recientemente se ha sugerido la posibilidad de que ordenadores diseñados con técnicas muy distintas (dando, por ejemplo, cabida a procesos cuánticos que aparecen cuando el tamaño de los circuitos se acerca al tamaño de los átomos que los componen) permitan realizar cálculos con un coste menor. Sin embargo, esta posibilidad es, hoy en día, meramente especulativa.

Si sólo los problemas cuya resolución se puede plantear como un autómata finito determinista pueden ser resueltos completamente utilizando un ordenador, ¿por qué se suele considerar a las máquinas de Turing como el modelo comúnmente aceptado para describir los procesos computables?

Una posible respuesta es que las máquinas de Turing son, al igual que otros modelos matemáticos, dependientes de las circunstancias.

Un modelo matemático, por ejemplo la geometría euclíadiana es correcto para realizar mediciones de parcelas agrícolas, y considerablemente más sencillo que un modelo que tenga en cuenta la curvatura de la Tierra. En cambio, este último modelo es inevitable si se quiere realizar navegación aérea.

Es evidente que las máquinas de Turing ponen un límite a la capacidad de computación. Por otro lado, definiciones basadas en otros momentos históricos y con otras motivaciones y notación han resultado equivalentes a esta definición de computabilidad, lo que proporciona un apoyo considerable al modelo de Turing.

De forma análoga, la descripción como autómata finito determinista puede ser sencilla y realista en muchos casos. En cambio, con frecuencia, las consideraciones sobre los límites de capacidad del ordenador se realizan con más elegancia y sencillez utilizando el modelo de Turing.

Otra cuestión que se plantea con frecuencia es la de si estos modelos pueden ser útiles para comprender y reproducir la inteligencia humana. De hecho, el ánimo de Turing fue definir un modelo para la mente. Esta cuestión es aún controvertida y volvemos a recomendar al lector interesado el libro anteriormente citado de R. Penrose.

1.3 Representaciones de un AFD

En esta sección estudiaremos las tres formas más habituales de representar un autómata finito:

1. gráficamente;
2. como una tabla de transiciones
3. mediante una función de transición.

El diagrama de la figura 1.3 es un *grafo dirigido etiquetado*. Un grafo dirigido etiquetado consta de:

1. un conjunto de etiquetas o alfabeto Σ ;
2. un conjunto Q de nodos;
3. para cada símbolo de trabajo $a \in \Sigma$, un subconjunto $E_a \subset Q \times Q$ de pares de estados llamados arcos.

En el caso que nos ocupa, los nodos del grafo son los estados del autómata y los arcos son los cambios de estado. Por ejemplo, $(q_1, q_2) \in E_1$ indica que existe un arco etiquetado con 1 que se dirige desde q_1 hasta q_2 .

La información que contiene este diagrama puede darse también de otra forma: a través de una tabla como la siguiente.

	0	1
q_1	q_1	q_2
q_2	q_3	q_1
q_3	q_2	q_3

La presentación en forma de tabla es también una forma válida, y utilizada con frecuencia, de describir un autómata finito determinista. A este tipo de tablas se les denomina *tabla de transiciones* y contienen los cambio de estado que se producen cuando se procesa un símbolo de la cadena de entrada. Para un símbolo dado, cada estado de partida está relacionado con un estado destino. Por ello, la tabla anterior puede entenderse también como un conjunto de funciones $\{\delta_0, \delta_1\}$ (una por cada columna) que asocian a cada estado de partida q del conjunto de estados posibles Q , otro de llegada q' también de Q . Matemáticamente, esto se escribe como $\delta_k : Q \rightarrow Q$ y el resultado de aplicar la función δ_k al estado q se representa como $q' = \delta_k(q)$. Aún es posible utilizar una notación más compacta y definir una única función $\delta : Q \times \Sigma \rightarrow Q$ que a cada estado q y a cada símbolo de trabajo $a \in \Sigma$ le asocia un estado de llegada $q' \in Q$, de forma que $\delta(q, a) = \delta_a(q)$. Esta función δ es conocida habitualmente como *función de transición*.

La función de transición del ejemplo es

$$\begin{aligned}
 \delta(q_1, 0) &= q_1 \\
 \delta(q_1, 1) &= q_2 \\
 \delta(q_2, 0) &= q_3 \\
 \delta(q_2, 1) &= q_1 \\
 \delta(q_3, 0) &= q_2 \\
 \delta(q_3, 1) &= q_3
 \end{aligned}$$

Un *autómata finito determinista* funciona efectuando transiciones entre sus estados de forma controlada por el símbolo que se está leyendo en cada momento de la entrada, de forma que un estado de llegada es, a su vez, estado de partida para la siguiente transición. Para nuestro ejemplo, esto puede expresarse de forma matemática como $\delta(q_1, 101) = \delta(q_2, 01) = \delta(q_3, 1) = q_3$. Observemos que ésta propiedad permite considerar la función de transición, que fue definida para símbolos, como una función sobre cadenas de entrada. En la sección 1.5 veremos ésto formalmente.

Una última consideración hace referencia al adjetivo determinista que ha acompañado hasta este punto al nombre de los autómatas finitos. Los autómatas que hemos considerado hasta aquí son deterministas en el sentido habitual de que sólo existe una opción en cada momento: dada un cadena de entrada sólo existe un posible estado en el que se puede encontrar el autómata tras la lectura, exactamente igual que un ordenador sólo puede encontrarse en una configuración después de ejecutar un programa (secuencia de instrucciones). Más adelante (sección 1.7.1) estudiaremos en qué sentido puede un autómata ser indeterminista (es decir, que éste no se encuentre en un estado definido), algo que, en principio, choca con la idea que tenemos de lo que debe ser un autómata o un programa de ordenador.

1.4 Alfabetos y lenguajes

Antes de proseguir, debemos definir formalmente algunos conceptos que nos acompañarán a lo largo de todo el libro.

Un *alfabeto* Σ es un conjunto finito y ordenado de símbolos. Por ejemplo: $\Sigma = \{a, b, c, d, e\}$ o el alfabeto binario $\Sigma = \{0, 1\}$. El orden de sus elementos se denomina *ordenación alfabética*. Una secuencia de un número arbitrario (finito) de símbolos del alfabeto forma una *cadena*, *palabra* o *frase*. Por ejemplo, $w = aca$ significa que la cadena w está formada por una secuencia de tres símbolos del alfabeto: a , c y a . Dos secuencias compuestas de los mismos símbolos pero en diferente orden son distintas: $aca \neq caa$. El número de símbolos que componen la cadena es su longitud, y se representa por $|w|$.

Al conjunto de todas las cadenas generables a partir del alfabeto Σ lo llamaremos *lenguaje universal* Σ^* y a la operación por la que se genera una cadena a partir de los símbolos que la componen, *concatenación*. La operación de concatenación se define como una operación interna y asociativa en Σ^* , de forma que si x , y y z son cadenas de Σ^* , entonces:

$$x(yz) = (xy)z = xyz$$

Es posible formar cadenas de longitud arbitraria, y en particular, cabe la posibilidad de que la cadena contenga 0 símbolos. En ese caso, se trata de una *cadena vacía*, que se representará de forma especial mediante el símbolo

ϵ y que satisface

$$\forall w \in \Sigma^* : w\epsilon = \epsilon w = w.$$

Con la incorporación de ϵ , cuyas características son las de un elemento neutro, la concatenación dota de estructura de *monoide*³ a Σ^* , pues se trata de una operación interna asociativa, con elemento neutro pero no conmutativa. En algunos casos es conveniente considerar ϵ como un símbolo especial del alfabeto (extendido) más que como una cadena. Al fin y al cabo se trata de un nuevo símbolo que representa algo difícil de escribir: nada. Cuando los lenguajes de programación representan cadenas de caracteres, utilizan comillas para marcar su comienzo y su final. En esa notación, la cadena vacía sería “”.

Por último, un *lenguaje* es un subconjunto cualquiera de cadenas de Σ^* . Por ejemplo, las frases del castellano son un subconjunto de todas las cadenas que se pueden formar a partir del alfabeto habitual, las letras especiales (ñ, á, ...), y los signos de puntuación. Pero también llamaremos, por analogía, lenguajes a conjuntos como por ejemplo el de las cadenas binarias que expresan un número par.

Un posible lenguaje es aquél que no contiene ninguna cadena o *conjunto vacío*, que se representa por \emptyset . Importa distinguir entre este y ϵ que es una cadena y como tal puede estar en un conjunto. Por ejemplo, $\{\epsilon\}$ representa a un conjunto con un solo elemento (en este caso ϵ), y por tanto, distinto del conjunto vacío. Los lenguajes que se pueden generar a partir de alfabetos finitos sólo pueden ser finitos o de cardinal infinito *numerable*⁴. En efecto, si un lenguaje es infinito, siempre es posible establecer una ordenación efectiva en él, de forma que a cada cadena se le asigne un número de orden siguiendo la llamada *ordenación lexicográfica* o *canónica*. Ésta consiste en recorrer las palabras del lenguaje de menor a mayor longitud de las cadenas y dentro de cada longitud por el orden alfabético definido para el alfabeto Σ . Por ejemplo, en el caso del alfabeto binario seguiría la sucesión $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$.

Un característica muy importante de los lenguajes es que el número de subconjuntos posibles de Σ^* tiene la *potencia*⁵ de $\mathbb{R} \simeq P(\mathbb{N})$ y por tanto, el número de lenguajes diferentes que existen es no numerable. Como consecuencia de esto se deduce que a partir de sistemas finitos (como lo son los autómatas y las gramáticas o cualquier otra notación matemática) que sólo pueden generar combinaciones distintas de elementos en una cantidad numerable, no es posible representar todos los lenguajes, y que existe un número infinito de ellos que no pueden ser expresados en la notación elegida. Se da, por tanto, la paradoja de que se sabe que existen dichos conjuntos, pero que no se puede hacer referencia explícita a cada uno de ellos por sep-

³Véase su definición en 109.

⁴véase la definición 112.

⁵Véase 112.

arado. Este tipo de cuestiones son objeto de estudio dentro de la teoría de la computabilidad[12]. Aquí sólo se pretende destacar el aspecto de que los lenguajes generados por autómatas (con esquemas de funcionamiento descritos de manera finita) y gramáticas finitas (sistemas con un número finito de reglas) son sólo un pequeño subconjunto, aunque infinito, de todos los posibles.

Ejercicios

1.4 Justifíquese que el elemento neutro ϵ de Σ^* es único.

1.5 Justifíquese que el “orden alfabético”, tal como se usa en los diccionarios, es incapaz de establecer una numeración o un orden en Σ^* .

1.6 Encuéntrese una función que dé el orden lexicográfico (canónico) de una palabra $w \in \Sigma^* = \{0,1\}^*$

1.5 Autómata finito determinista como clasificador

Un autómata finito puede servir para discriminar palabras, observando el estado final en el cuál se encuentra. Por ejemplo, el autómata de la figura 1.8 permite distinguir las cadenas de $\Sigma^* = \{0,1\}^*$ que empiezan por 0 de

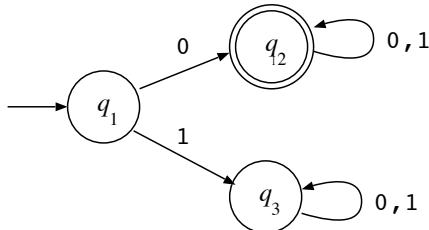


Figura 1.8: Ejemplo de autómata finito determinista como clasificador

aquellas que empiezan por 1. El estado inicial (aquel en el que se encuentra el autómata antes de analizar la cadena) ha sido señalado con una flecha. De ese estado se pasa a otro —bien q_2 , o q_3 — al recibir el primer símbolo y permanece en el mismo nodo independientemente de los símbolos que sean leídos a continuación. Se ha señalado el estado q_2 con doble círculo indicando que el autómata seleccionará como correctas (aceptadas) las cadenas que conduzcan a dicho estado, que denominaremos *estado de aceptación*. Esta función como clasificadores de palabras permite que los autómatas finitos sean utilizados con frecuencia para el análisis léxico de programas o textos, es decir, para determinar qué palabras están correctamente escritas y

pertenecen al lenguaje en cuestión, o también para buscar determinados patrones en el texto. Por ejemplo, de entre las palabras que se pueden formar con $\Sigma = \{0, 1, \dots, 9, E, +, -, .\}$ sólo algunas son números reales válidos en C. De la misma manera, de entre las palabras que se pueden formar con $\Sigma = \{0, 1, \dots, 9, a, b, \dots, y, z, -\}$ sólo unas cuantas son identificadores válidos en C. Para esta tarea, es posible utilizar autómatas finitos y de hecho así se hace en los analizadores léxicos de intérpretes y compiladores.

Siguiendo este planteamiento, los elementos de un autómata finito determinista (AFD) son $M = (Q, \Sigma, \delta, q_1, F)$, siendo:

- $Q = \{q_i\}$, el conjunto finito de estados posibles del autómata;
- Σ , su alfabeto finito de entrada;
- $q_1 \in Q$, su estado inicial;
- $F \subset Q$, el subconjunto de estados de aceptación;
- δ , una *función de transición* entre estados de la forma $\delta : Q \times \Sigma \rightarrow Q$ que dicta el comportamiento del autómata.

Un autómata finito opera ejecutando secuencias de movimientos o pasos según va leyendo los símbolos de la cadena de entrada. Cada paso viene determinado por el estado actual del autómata y por el símbolo que se lee de la entrada. El movimiento consiste en leer ese símbolo (de manera que el siguiente símbolo de la cadena será leído en el siguiente movimiento) y cambiar al estado determinado por la función de transición δ (que en algún caso puede ser el mismo estado).

Para describir formalmente el comportamiento del autómata sobre una cadena, debemos extender la función de transición $\delta : Q \times \Sigma \rightarrow Q$ para que pueda actuar sobre un estado y una cadena y no sólo sobre un estado y un símbolo. Así, la nueva función es $\delta : Q \times \Sigma^* \rightarrow Q$, de manera que $\delta(q, w)$ es el estado en el que se encontrará el autómata después de haber leído la cadena w empezando en el estado q .

La definición formal de la nueva función δ (que incluye a la vieja en el caso de que la cadena leída sea un sólo símbolo) es:

- $\delta(q, \epsilon) = q$ (si no se lee ninguna cadena, es decir, se lee la cadena vacía, el autómata permanece en el mismo estado); y
- $\forall w \in \Sigma^*, \forall a \in \Sigma$ se cumple $\delta(q, wa) = \delta(\delta(q, w), a)$ (es decir, el estado al que llega el autómata al leer la cadena no vacía wa puede obtenerse por pasos: primero se calcula el estado $p = \delta(q, w)$ al que se llega con w desde q , y después se obtiene el estado al que se llega con el símbolo a desde p : $\delta(p, a)$).

Conviene insistir en algunas de las propiedades de δ :

1. El autómata es *determinista*, en el sentido de que siempre está únicamente definido cuál es el estado en que se encuentra el autómata después de haber leído una parte de la cadena de entrada, pues δ es una *aplicación* (véase la definición 109) y la imagen de un estado q_i y un símbolo a es un único estado de destino $q_j = \delta(q_i, a)$. En lo que sigue, se utilizará la notación más compacta⁶ $\delta_a(q) := \delta(q, a)$.
2. El autómata respeta la propiedad de *concatenación*, es decir, el efecto sobre M de dos cadenas consecutivas u y v es el mismo que el que se produce al leer una sola cadena uv formada por u seguida de v . Matemáticamente, esto se expresa como:

$$\delta_{uv} = \delta_v \delta_u.$$

La inversión de u y v en la fórmula se debe al diferente sentido de lectura de la composición de funciones y de la escritura occidental: actúa primero la función escrita más a la derecha, mientras que en la escritura occidental una cadena empieza por el símbolo situado más a la izquierda.

3. La función δ está completamente definida si se da un grafo o una *tabla de transiciones* : $Q \times \Sigma \longrightarrow Q$. Aunque en principio δ proporciona el nodo final para cualquier cadena de Σ^* (y por tanto, para un conjunto infinito de cadenas)m, es suficiente con describir el resultado de la función para el subconjunto finito de cadenas de longitud 1 —los símbolos de Σ . Ésto es lo que se conoce como una tabla de transiciones, ya que suele representarse en forma de tabla. Por ejemplo, el autómata de la figura 1.8 queda perfectamente definido por la tabla de la figura 1.9.

δ	0	1
q_1	q_2	q_3
q_2	q_2	q_2
q_3	q_3	q_3

Figura 1.9: Tabla de transiciones para el autómata de la figura 1.8.

Con estas propiedades, se establece un *homomorfismo*⁷ entre la concatenación de cadenas del monoide Σ^* y la composición de las funciones $\delta_w : Q \rightarrow Q$ que corresponden a cada cadena w . Puede comprobarse que con sólo haber exigido esta naturaleza de homomorfismo se hubiesen cumplido todas las propiedades que se han indicado.

⁶Aunque no estándar.

⁷véase 109.

Como cabe esperar, se define el *lenguaje aceptado* por un AFD como:

$$L(M) = \{w \in \Sigma^* : \delta_w(q_1) \in F\}$$

La definición de lenguaje aceptado viene motivada por la utilización del AFD como reconocedor de lenguajes: el AFD nos permite determinar si una cierta cadena w pertenece o no a $L(M)$ simplemente observando si el estado final al que se llega mediante w es uno de los estados del subconjunto F . Todos los lenguajes reconocibles mediante autómatas finitos deterministas reciben el nombre de *lenguajes regulares*. Por tanto, dado un lenguaje L , si existe un AFD tal que $L = L(M)$, entonces L es un lenguaje regular. Como ya se argumentó, los lenguajes regulares son un pequeño subconjunto de todos los lenguajes posibles, pero de gran importancia en informática teórica y en teoría de la computación.

Para terminar esta sección, definiremos un concepto que aparece frecuentemente, que es el de *nodo útil* en un AFD: un nodo q_k es *útil* si es utilizado en el proceso de reconocimiento de alguna cadena. Para ello, es necesario que el nodo sea accesible⁸ desde el estado inicial q_1 por alguna cadena u y además que algún estado de F sea accesible desde q_k por alguna cadena v . Formalmente, q_k es útil $\iff \exists u, v \in \Sigma^* : \delta_u(q_1) = q_k \wedge \delta_v(q_k) \in F$.

Un *nodo inútil* en un AFD puede ser eliminado sin que por ello $L(M)$ cambie, pues no desempeña ningún papel en la tarea de reconocimiento. Frecuentemente, los nodos inútiles no aparecen representados en los grafos o tablas con el fin de aligerar la notación y no sobrecargar los dibujos, pues esto haría más difícil su interpretación, sobre todo cuando se trata con autómatas cuyo número de nodos no es muy pequeño. Por contra, esta simplificación puede originar que desde algunos nodos no salgan transiciones con alguno (o varios) de los símbolos del alfabeto ($\text{card}(\delta_a) \leq 1$ en lugar de $\text{card}(\delta_a) = 1$). Con ello parece perderse la característica del determinismo, que significa que tras cada paso está perfectamente definido el siguiente. Sin embargo, el autómata puede interpretarse aún como determinista, si se sobreentiende que las cadenas que intenten efectuar esa transición no pertenecen al lenguaje reconocido por el autómata. Ello equivale a decir que los arcos no dibujados se dirigen todos a un estado $q_{\text{abs}} \notin F$ llamado *estado de absorción*. Dicho estado no pertenece a F y además no es posible salir de él porque $\delta_a(q_{\text{abs}}) = q_{\text{abs}}$ para cualquier símbolo a del alfabeto. Por tratarse de un caso de nodo inútil, con frecuencia q_{abs} no aparece representado, pero su existencia debe tenerse en cuenta, sobre todo cuando se apliquen algoritmos cuyo funcionamiento sólo está garantizado sobre autómatas estrictamente deterministas ($\text{card}(\delta_x) = 1 \forall x \in \Sigma$). Un ejemplo de nodo de absorción aparece en la figura 1.8.

Ejercicios

⁸Es decir, que exista algún camino que lo conecte.

1.7 Prueba que, por consistencia, δ_e ha de ser la aplicación identidad en un AFD (recuerda que δ es un homomorfismo entre monoides).

1.8 Dibuja autómatas finitos deterministas que reconozcan los siguientes conjuntos de cadenas construidas sobre el alfabeto $\Sigma = \{0,1\}$:

- cadenas acabadas en 00;
- cadenas con dos unos consecutivos;
- cadenas que no contengan dos unos consecutivos;
- cadenas con dos ceros consecutivos o dos unos consecutivos;
- cadenas con dos ceros consecutivos y dos unos consecutivos;
- cadenas acabadas en 00 o 11;
- cadenas con un 1 en la antepenúltima posición;
- cadenas de longitud 4.

1.9 ¿Qué características debe tener el grafo de un AFD para que el lenguaje regular que reconoce sea infinito?

1.6 El autómata finito determinista como traductor

Los AFD definidos hasta ahora permiten la clasificación de las cadenas en dos clases (aceptadas y no aceptadas). En muchos casos, sin embargo, es necesario realizar una clasificación en multitud de grupos. Esto ocurre, por ejemplo, en los problemas de traducción entre lenguajes. Para este fin se introducen los *traductores secuenciales*. Un traductor secuencial es un AFD al que se le ha incorporado un alfabeto Λ y una función de salida λ . Además, dado que no estamos interesados en la función de aceptor, no es necesario hacer referencia al subconjunto de estados de aceptación. Por tanto, el traductor queda definido por $M = (Q, \Sigma, \Lambda, \delta, \lambda, q_1)$. En lo que sigue supondremos que una cadena de entrada $w_{\text{in}} = a_1 a_2 a_3 \dots$ produce en M una secuencia de transiciones $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} q_3 \dots$. Dependiendo de cuál sea la forma de la función λ , los traductores se clasifican en distintos grupos, entre los cuales se encuentran:

- *Máquinas de Moore* ($\lambda : Q \longrightarrow \Lambda$)

En este caso, se genera una cadena $w_{\text{out}} = \lambda(q_1)\lambda(q_2)\lambda(q_3)\dots$, pues el símbolo de salida es función únicamente del estado en el que se encuentra M . Inmediatamente *después* de que M cambie de estado, se detecta un nuevo símbolo como salida.

- *Máquina de Mealy* ($\lambda : Q \times \Sigma \longrightarrow \Lambda$)

El resultado es una cadena $w_{\text{out}} = \lambda_{a_1}(q_1)\lambda_{a_2}(q_2)\lambda_{a_3}(q_3)\dots$, donde el símbolo de salida depende del estado y de la transición que se efectúa y varía *simultáneamente* con M cuando este cambia de estado. Para que M funcione de una forma razonable, es necesario suponer que $\lambda_\epsilon(q) = \epsilon$.

Aunque aparentemente distintos, existe una equivalencia aproximada entre ambos tipos de autómatas: toda máquina de Moore puede ser implementada como una máquina de Mealy equivalente y viceversa. Por equivalentes entendemos que generan la misma cadena de salida w_{out} para cada w_{in} . Esta afirmación debe tomarse con cierta cautela, pues existirá un desfase temporal entre el momento de aparición de w_{out} en cada máquina. Por este motivo, en algunos casos conviene suponer que el final de la cadena w_{in} está marcado por algún símbolo que no pertenece al alfabeto, por ejemplo $\#$ o $\$$.

- Dada una máquina de Moore M , es sencillo transformarla en una máquina M' de Mealy que simule M . Para ello basta con redefinir la función de salida de M para cada símbolo del alfabeto a de la siguiente forma: $\lambda'_a(q_i) = \lambda(\delta_a(q_i))$.
- En el caso inverso, dada M , una máquina de Mealy, entonces se puede construir una máquina de Moore $M' = (Q', \Sigma, \Lambda, \delta', \lambda', q'_1)$ de la siguiente forma:

- $Q' = Q \times \Lambda = \{q' = (q, \lambda) : q \in Q \wedge \lambda \in \Lambda\}$
- $q'_1 = (q_1, \lambda_1)$, con λ_1 arbitrario
- $\delta'_a(q') = (\delta_a(q), \lambda_a(q))$
- $\lambda'(q') = \lambda'(q, \lambda) = \lambda$

Con estas definiciones, M' y M son equivalentes.

Ejercicios

1.10 Constrúyase una máquina de Mealy con alfabeto de entrada $\{0, 1\}$ y alfabeto de salida $\{\text{sí}, \text{no}\}$ que produzca como última salida **sí** si los dos últimos dígitos de la cadena binaria son iguales y **no** si son diferentes. Represéntese la máquina de Moore equivalente.

1.11 La salida w_{out} de una máquina de Mealy para w_{in} se utiliza como entrada de un AFD. Demuestra que el conjunto de cadenas aceptadas por el autómata compuesto es también un lenguaje regular (w_{in} es aceptada si el AFD se detiene sobre un estado de aceptación).

1.12 Construyase una máquina de Moore con alfabeto de entrada $\{0, 1\}$ y alfabeto de salida $\{a, b\}$, que genere una a cada vez que entren dos unos consecutivos y una b cada vez que entren dos ceros consecutivos. Por ejemplo, la traducción de la cadena “0010111” será “ baa ”. (Obsérvese que algunos estados del traductor finito pueden producir la cadena vacía ϵ).

1.13 Construye una máquina de Mealy que proporcione el cociente entre 3 para cada cadena binaria de entrada. Transforma ésta en una máquina de Moore equivalente.

1.14 Escribe una máquina de Moore que traduzca cadenas de $\Sigma^* = \{0, 1\}^*$ a cadenas de $\Lambda^* = \{0, 1\}^*$ de manera que el símbolo que se saque en cada paso sea 1 si el número de unos leído es impar, y cero si es par. Por ejemplo, la traducción de 100101 sería 111001.

1.15 ¿Es equivalente una máquina de Mealy a un homomorfismo $f : \Sigma \rightarrow \Lambda$? ¿Por qué? Ayuda: fíjate en el ejercicio anterior.

1.7 Deterministas, indeterministas y estocásticos

Un criterio de clasificación alternativo para los autómatas finitos es su forma de funcionamiento. En un AFD todas las transiciones y los estados que va ocupando el autómata están completamente determinados por la cadena de entrada. Sin embargo, es posible construir autómatas en los que existan varias transiciones posibles para un mismo símbolo, a los que se denomina *autómatas finitos indeterministas* (AFI). También se puede considerar la posibilidad de asignar probabilidades a las transiciones, lo cual conduce al concepto de *autómata finito estocástico* (AFE).

1.7.1 Autómatas finitos indeterministas

Un AF *indeterminista* (AFI) es una generalización de los AFD en la que se permite que de un nodo parta más de una transición etiquetada con el mismo símbolo, o que exista más de un estado inicial. Esta flexibilidad podría aumentar, aparentemente, la potencia de los autómatas y permitir el reconocimiento de conjuntos que no son analizables mediante un AFD. Sin embargo, veremos inmediatamente que esto no es así, y que la capacidad de análisis de los AFI es exactamente la misma que la de los AFD.

Dado que varios caminos en el AFI pueden corresponder a una única cadena de entrada, serán varios, en general, los estados alcanzables a través de dicha cadena. El criterio que se toma para decidir qué cadenas son aceptadas es el de tomar aquellas para las que existe al menos un camino que conduce a un estado de aceptación desde alguno de los estados iniciales. Esta definición presenta la ventaja de que el AFD es un caso particular en el cuál

sólo existe un camino posible. Además, el conjunto de lenguajes generables por AFI resulta ser precisamente el conjunto de lenguajes regulares. De hecho, dado M_I , un AFI, existe siempre un AFD equivalente M_D que genera el mismo lenguaje: $L(M_D) = L(M_I)$.

La definición de un AFI es semejante a la de un AFD:

$$M = (Q, \Sigma, \delta, Q_1, F)$$

, con la diferencia de que el estado inicial ha sido sustituido por un subconjunto de posibles estados iniciales $Q_1 \subseteq Q$. Además, la función de transición δ ya no conduce de un estado a otro único estado. Las consecuencias del indeterminismo son:

- La imagen de un estado y un símbolo es, en general, un subconjunto de estados de Q : $\delta_a(q) = R \subset Q$
- La propiedad de concatenación se sigue satisfaciendo si definimos cómo actúa δ sobre un conjunto de estados:

$$\delta_{uv}(q) = \delta_v[\delta_u(q)] = \bigcup_{r \in \delta_u(q)} \delta_v(r).$$

Esta definición se toma porque estamos interesados en ver si existe algún camino que conduce desde el estado inicial a un estado de aceptación. Si un símbolo nos permite llegar a varios estados R , el siguiente símbolo nos permite llegar a todos los alcanzables desde cualquiera de los estados anteriores de R . De esta manera a cada paso se abre un abanico de caminos alternativos y el conjunto que se obtiene en cada momento es el subconjunto de estados alcanzable mediante la cadena analizada.

- En un AFI, δ_ϵ ya no es necesariamente la aplicación identidad, y se llama *clausura nula* o *clausura- ϵ* (C_ϵ) del estado q al conjunto de estados accesibles directamente mediante la cadena ϵ , es decir, $C_\epsilon(q) := \delta_\epsilon(q)$. Es necesario, además, imponer que $q \in C_\epsilon(q)$, dada la posibilidad obvia de que el autómata no cambie de estado cuando no lee ningún símbolo. Si se diese el caso de que $C_\epsilon(q) \neq \{q\}$ para algún nodo q , se dice que el autómata M_I contiene *transiciones nulas* o de tipo ϵ . El concepto de clausura nula puede extenderse para subconjuntos de estados: dado $T \subseteq Q$, su clausura nula $C_\epsilon(T)$ es la unión de las clausuras nulas de todos los estados de T :

$$C_\epsilon(T) = \bigcup_{q \in T} C_\epsilon(q).$$

De acuerdo con todo lo anterior, el lenguaje aceptado por el AFI puede definirse como el conjunto de cadenas w tales que existe algún camino etiquetado con w que lleva hasta un nodo de F partiendo desde alguno de los nodos del subconjunto Q_1 :

$$L(M) = \{w \in \Sigma^* : \delta_w(Q_1) \cap F \neq \emptyset\}$$

Como veremos, para describir completamente un AFI basta con especificar la tabla de una función más restringida, $\tilde{\delta} : Q \times (\Sigma \cup \epsilon) \rightarrow Q$, que relaciona estados y símbolos (o la cadena vacía ϵ) con nuevos estados. Estas transiciones son las que habitualmente figuran en el grafo dirigido que representa al autómata.

1.7.2 Equivalencia AFI-AFD

Vamos a demostrar que los AFI no aumentan la potencia de los AFD, ya que dado un AFI siempre es posible construir un AFD capaz de desempeñar el mismo papel. Para ello procederemos en dos pasos: primero estudiaremos cómo eliminar las transiciones nulas, para después proceder a obtener el AFD equivalente a un AFI sin transiciones nulas.

Para ello, lo primero que hay que hacer es asegurarse de que la función de transición satisface $\delta_{uv} = \delta_v \delta_u$. Aunque esto debería cumplirse por definición, muchas veces se parte de un grafo o tabla de transiciones en el que faltan transiciones compuestas, y se da sólo una función de transición restringida $\tilde{\delta}$. Por ejemplo, la siguiente tabla de transiciones:

$\tilde{\delta}$	a	b	ϵ
q_1	q_1		q_2
q_2		q_2	

corresponde al grafo de la figura 1.10. En él no aparece ninguna transición

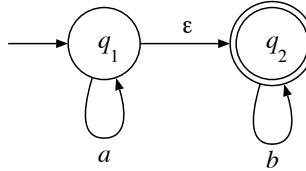


Figura 1.10: Ejemplo de autómata finito indeterminista.

dirigida por a desde q_1 hasta q_2 . Sin embargo, dado que $a = a\epsilon$, dicha transición puede producirse: $q_1 \xrightarrow{a} q_1 \xrightarrow{\epsilon} q_2$. Para completar sistemáticamente la tabla, basta con tomar $\delta_a(q) = C_\epsilon \tilde{\delta}_a C_\epsilon(q)$, que incorpora al conjunto imagen todos los nodos accesibles, aunque para ello se requieran una o varias transiciones vacías. En el ejemplo, $C_\epsilon(q_1) = \{q_1, q_2\}$ y $C_\epsilon(q_2) = \{q_2\}$, y la tabla de transiciones para la función δ queda:

δ	a	b	ϵ
q_1	$\{q_1, q_2\}$	$\{q_2\}$	$C_\epsilon(q_1) = \{q_1, q_2\}$
q_2		$\{q_2\}$	$C_\epsilon(q_2) = \{q_2\}$

Una vez que la correspondencia entre la concatenación de cadenas w en Σ^* y la composición de funciones de transición δ_w tiene las propiedades correctas de un homomorfismo, es posible eliminar todas las transiciones vacías, a excepción de aquellas que vayan desde un estado inicial directamente a algún estado de aceptación. Ello es debido a que para cualquier camino que incluya transiciones vacías existe otro camino alternativo que conduce al mismo estado con la misma cadena pero sin símbolos ϵ . Dado que en un AFI basta con que exista un camino, esta redundancia puede ser eliminada. Por ejemplo, si desde q_1 existe un camino w hasta q_f , uno de cuyos arcos es vacío y lo representamos explícitamente ($w = uev$), entonces se puede reescribir el camino de la siguiente forma: $\delta_w = \delta_v \delta_\epsilon \delta_u = \delta_v \delta_u$; es decir, el conjunto de nodos al que se puede acceder por medio de u seguido de ϵ es exactamente el mismo que el conjunto accesible con sólo u , y la transición vacía no desempeña ningún papel relevante y puede ser suprimida. El único caso en que esto no puede hacerse, debido precisamente a la definición de δ_ϵ es cuando $w = \epsilon$. Para solucionar esto, basta con modificar Q_1 , de forma que incluya a $q_f \in F$ si $q_f \in C_\epsilon(Q_1)$. Todo lo anterior puede resumirse en que la nueva función de transición es:

$$\delta'_w(q) = \begin{cases} C_\epsilon(\delta_w(C_\epsilon(q))) & \text{si } w \neq \epsilon \\ \{q\} & \text{en caso contrario} \end{cases}$$

y el nuevo subconjunto Q'_1 está definido por:

$$Q'_1 = Q_1 \cup (C_\epsilon(Q_1) \cap F)$$

Evidentemente, el autómata así modificado ya no contiene transiciones vacías y por tanto $\forall q \in Q : C_\epsilon(q) = \{q\}$.

Una vez que M no contiene transiciones vacías, queremos construir otro autómata M' que desempeñe las mismas funciones que M pero cuyo funcionamiento sea determinista. Para ello, conviene observar que en un AFI las transiciones informan sobre los nodos accesibles a través de un símbolo particular, de forma que de un nodo pueden salir varias transiciones con la misma etiqueta. Si lo que pretendemos es concentrar toda esta información en un solo arco, necesariamente dicha información ha de estar contenida en el nodo de llegada. Por tanto el nodo de llegada debe decirnos qué subconjunto de estados podemos acceder con ese símbolo. Si el autómata original consta de n nodos, se pueden formar 2^n subconjuntos distintos en Q . Este número, aunque grande, es finito y por tanto podemos construir un nuevo autómata M' en el que los estados representen a subconjuntos de nodos de M y los arcos etiquetados con un símbolo a indiquen qué subconjunto de estados son accesibles mediante a desde el subconjunto de partida. De

esta forma la misma información que está contenida en M , aparece en M' . Rigurosamente $M' = (Q', \Sigma, \delta', q'_1, F')$ está definido por:

- $Q' = P(Q)$
- $q'_1 = Q_1$
- $F' = \{R \in Q' : R \cap F \neq \emptyset\}$
- $\delta'_w(R) = \delta_w(R)$

Trivialmente, M' es un AFD pues δ' tiene las propiedades de un homomorfismo:

- $\delta'_w(R) = \delta_w(R) = S \in Q'$
- $\delta'_\epsilon(R) = R$ (gracias a que M ya no incluye transiciones de tipo ϵ)
- $\delta'_{uv} = \delta_{uw} = \delta_v \delta_u = \delta'_u \delta'_u$

Por otro lado, es necesario demostrar que $L(M) = L(M')$. Para ello recordamos las definiciones de lenguaje aceptado en el caso indeterminista y determinista respectivamente:

$$\begin{aligned} L(M') &= \{w \in \Sigma^* : \delta'_w(q'_1) \in F'\} \\ &= \{w \in \Sigma^* : \delta'_w(Q_1) \in F'\} \\ &= \{w \in \Sigma^* : \delta'_w(Q_1) \cap F \neq \emptyset\} \\ &= \{w \in \Sigma^* : \delta_w(Q_1) \cap F \neq \emptyset\} = L(M) \end{aligned}$$

De forma que queda demostrada la equivalencia de M' y M . Hemos llegado pues a la conclusión de que los AFI no aumentan la capacidad de análisis de los AFD. Por un lado, esto es una buena noticia, ya que podemos evitar los problemas del tratamiento del indeterminismo. Por otro, nos obliga en algunos casos a buscar modelos más generales de análisis de lenguajes. Veremos por ejemplo, que un lenguaje matemático que incluya la posibilidad de anidar un número finito pero arbitrario de paréntesis no puede ser tratado mediante un AFD. Por tanto, un AFI tampoco nos solucionaría el problema, y será necesario recurrir a las gramáticas independientes del contexto.

Adelantaremos aquí, que la construcción del AFD equivalente a un AFI dado tal como se acaba de describir será una operación rutinaria durante la traducción de expresiones regulares (una notación compacta para los lenguajes regulares) a AFD, dado que un paso intermedio en dicho proceso es la construcción de un AFI.

Ejercicios

1.16 Constrúyanse autómatas finitos indeterministas para los siguientes lenguajes sobre $\{0, 1\}$:

- Cadenas con dos ceros consecutivos o dos unos consecutivos.
- Cadenas con un 1 en la antepenúltima posición.

Desárróllense sus equivalentes deterministas.

1.17 Transfórmese en un AFD el autómata dado por $Q_1 = \{q_1\}$, $F = \{q_4, q_7\}$ y la siguiente tabla de transiciones. ¿Qué conjunto de cadenas son aceptadas?

	a	b	ϵ
q_1			q_2, q_5
q_2	q_2, q_3	q_2	
q_3	q_4		
q_4			
q_5	q_5	q_5, q_6	
q_6		q_7	
q_7			

1.18 Determina cuál es el conjunto de cadenas aceptadas por el autómata M dado por la siguiente función de transición: $\delta_a(q_1) = \{q_1, q_2\}$, $\delta_b(q_1) = \{q_2\}$, $\delta_a(q_2) = \emptyset$, $\delta_b(q_2) = \{q_1, q_2\}$, donde $Q_1 = F = \{q_1\}$

1.19 Enuncia ventajas y desventajas del uso de AFI y de AFD. Analiza la complejidad espacial y temporal en ambos casos. En vista del ejercicio anterior, comenta en qué caso es más sencillo identificar visualmente cuál es lenguaje aceptado.

1.7.3 Autómatas finitos estocásticos

Un autómata finito estocástico asocia a cada cadena de Σ^* una cierta probabilidad de aparición, con lo que el lenguaje es generado en un orden aleatorio, pudiendo aparecer las cadenas repetidas veces. Este tipo de comportamiento es importante si se quiere reproducir algún proceso real. Con frecuencia las cadenas con las que se trabaja son generadas por fuentes con comportamiento aleatorio, o sufren efectos de ruido, etc. Si no se da un tratamiento estadístico a estos lenguajes, su reconocimiento y análisis se vuelve muy difícil. Por ello conviene introducir en el modelo la probabilidad de que aparezca cada una de las cadenas. En el caso de los AFE⁹, la probabilidad se asigna a cada una de las transiciones y a los estados de aceptación.

⁹Aquí nos limitaremos a autómatas deterministas.

Formalmente $M = (Q, \Sigma, P, q_1, P_f)$, donde en vez de una función δ aparece P , una tabla de probabilidades de transición $p_{ij}(a)$ desde el nodo q_i hasta el nodo q_j mediante el símbolo a . Además, el conjunto de estados de aceptación ha sido reemplazado por una lista P_f de las probabilidades p_{if} de terminación en cada nodo q_i . Si el autómata se encuentra en un estado q_i , el siguiente paso se determina aleatoriamente según las probabilidades $p_{ij}(a)$ y p_{if} . Evidentemente, las probabilidades de los sucesos que pueden ocurrir a continuación han de sumar uno:

$$\sum_{q_j \in Q, a \in \Sigma} p_{ij}(a) + p_{if} = 1$$

Por tanto, no es necesario conocer los valores de p_{if} , y basta con presentar la tabla P .

La probabilidad $p(w)$ de generar una cadena $w \in \Sigma^*$, vendrá dada por el producto de las probabilidades de las transiciones que ejerce el camino, incluida la probabilidad de terminación en el último nodo. Esto puede hacerse de forma recursiva. Si definimos $p_{ij}(w)$ como la probabilidad de que se vaya del nodo q_i al q_j al leer la cadena w , podemos escribir:

$$p_{ij}(\epsilon) = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{en caso contrario} \end{cases}$$

$$p_{ij}(wa) = \sum_{k=1}^{\text{card}(Q)} p_{ik}(a)p_{kj}(w).$$

La probabilidad de que una palabra w sea generada depende de las probabilidades de aceptación p_{if} :

$$p(w) = \sum_{k=1}^{\text{card}(Q)} p_{1k}p_{kf}.$$

Es posible demostrar que si el AFE no contiene nodos inútiles, entonces la suma de las probabilidades de todas las cadenas de Σ^* es la unidad:

$$\sum_{w \in \Sigma^*} p(w) = 1$$

El lenguaje generado por el AFE consta de todas aquellas cadenas cuya probabilidad de aparición es no nula:

$$L(M) = \{w \in \Sigma^* : p(w) \neq 0\}$$

Es posible introducir autómatas finitos estocásticos indeterministas (AFEI), en los que puede haber más de un camino asociado a una única cadena y la probabilidad global será la suma de las probabilidades de cada camino. Sin embargo, éstos no son equivalentes a los deterministas, pues pueden generar más distribuciones de probabilidad que los AFE deterministas.

Ejercicios

1.20 Demuestra que la suma de las probabilidades de todas las cadenas de Σ^* proporcionadas por un AFE sin nodos inútiles es la unidad.

1.21 Busca un ejemplo para el que no exista un AFE determinista que reproduzca las probabilidades generadas por un AFE indeterminista.

Capítulo 2

Lenguajes regulares

2.1 Expresiones regulares

Como se comentó anteriormente, dado un alfabeto Σ existe un número infinito de lenguajes (subconjuntos) posibles en Σ^* , siendo además este infinito de tipo no numerable. Por otro lado, la cantidad de frases de longitud finita que se pueden formar en cualquier lenguaje (incluidos los lenguajes naturales) es infinito pero numerable. Como consecuencia, sólo podemos describir con frases finitas un número infinito pero numerable de entidades matemáticas. La consecuencia inmediata es que cualquier intento de definir una representación para todos los subconjuntos de Σ^* está condenado al fracaso, entendiendo por representación cualquier método para asignar a cada lenguaje una expresión distinta que lo identifique únicamente.

Una forma de representar algunos subconjuntos de Σ^* de forma precisa consiste en el uso de expresiones regulares. Una *expresión regular* no es más que un tipo de notación matemática para representar lenguajes. Los conjuntos representables mediante una expresión regular reciben el nombre de *conjuntos regulares*. Aunque la definición de estos conjuntos no parece guardar ninguna relación con la forma en que se introdujeron los lenguajes regulares, llegaremos a la interesante conclusión de que estamos hablando exactamente de los mismos objetos¹.

Antes de introducir las expresiones regulares, vamos a definir algunas operaciones con lenguajes:

- La *unión* de dos lenguajes L_1 y L_2 es:

$$L_1 \cup L_2 = \{x : x \in L_1 \vee x \in L_2\}$$

- La *concatenación* de dos lenguajes L_1 y L_2 es::

$$L_1 L_2 = \{xy : x \in L_1 \wedge y \in L_2\}$$

¹Por una vez, un nombre semejante no se ha elegido sólo para confundir al lector desorientado.

La concatenación de un lenguaje consigo mismo un número i de veces se abrevia L^i , de forma que $L^i = LL^{i-1}$ si $i > 0$, y $L^0 = \{\epsilon\}$.

- La *clausura de Kleene* de un lenguaje L se escribe L^* , y se define como:

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

y significa *cero o más concatenaciones de L* .

- La *clausura positiva* de L se escribe L^+ y se define:

$$L^+ = \bigcup_{i=1}^{\infty} L^i,$$

es decir, $L^+ = L(L^*)$. Nótese que L^+ coincide con L^* si y sólo si $\epsilon \in L$.

Las expresiones regulares, que utilizaremos para caracterizar los conjuntos regulares, pueden ser definidas recursivamente. Si escribimos $L(r)$ para el conjunto denotado por una expresión r , entonces:

- \emptyset es una expresión regular que denota el lenguaje vacío ($L(\emptyset) = \emptyset$);
- ϵ es una expresión regular que denota el lenguaje que sólo contiene la cadena vacía ($L(\epsilon) = \{\epsilon\}$);
- Si a es un símbolo de Σ , entonces a es una expresión regular que denota el lenguaje que contiene la cadena a ($L(a) = \{a\}$).

Estas tres primeras reglas (cláusulas iniciales de la definición recursiva) definen las expresiones regulares básicas.

- Dadas r y s , dos expresiones regulares entonces:

- $(r) + (s)$ es una expresión regular² que denota la unión de $L(r)$ y $L(s)$, es decir, $L(r) \cup L(s)$.
- $(r)(s)$ es una expresión regular que denota denota al conjunto concatenación de $L(r)$ y $L(s)$, es decir, $L(r)L(s)$.
- $(r)^*$ es una expresión regular que denota $(L(r))^*$, la clausura de Kleene de $L(r)$.

Con esta última serie de reglas (cláusulas inductivas de la definición recursiva), podemos definir cualquier expresión regular basándonos en las expresiones básicas.

- Ningún otro objeto es una expresión regular (cláusula maximal de la definición recursiva).

²En otra notación $(r)|(s)$.

Las expresiones obtenidas por los mecanismos anteriores pueden ser simplificadas mediante la eliminación de paréntesis innecesarios si se dota de una regla de precedencia a las operaciones de unión, concatenación y clausura. El criterio que se toma es el mismo que el de las operaciones aritméticas de suma, producto y potencia de números respectivamente.

Podemos decir que los *conjuntos regulares* son los subconjuntos de Σ^* que pueden ser definidos con expresiones regulares. Es decir, un subconjunto de Σ^* es regular si y sólo si es \emptyset , $\{\epsilon\}$, $\{a\}$ para algún símbolo $a \in \Sigma$, o puede ser obtenido a partir de estos conjuntos mediante un número finito de aplicaciones de las operaciones de unión, concatenación o clausura de Kleene.

Algunos ejemplos de expresiones regulares sobre $\Sigma = \{0, 1\}$ son:

- 0^* (conjunto de todas las cadenas de ceros)
- $(0 + 1)^*$ (conjunto de cadenas formadas por ceros y unos, es decir, Σ^*)
- $(10)^*$ (cadenas formadas por secuencias de 10, incluyendo ϵ)
- $(1 + 0)^*10^*$ (cadenas que contienen al menos un 1)
- $(0 + 1)^*00(0 + 1)^*$ (cadenas con dos ceros consecutivos)
- $(1 + 10)^*$ (cadenas que no tienen dos ceros consecutivos y empiezan por uno)

Ejercicios

2.1 Describanse los conjuntos que denotan las siguientes expresiones regulares:

- $1(0 + 1)^*1$
- $(0^*(\epsilon + 1))^*$
- $1^*01^*01^*$

2.2 Escríbanse expresiones regulares que denoten los siguientes conjuntos sobre $\Sigma = \{0, 1\}$:

- El conjunto de las cadenas que acaban en 0.
- El conjunto de las cadenas que tienen un sólo 0
- El conjunto de las cadenas que no contienen la secuencia 000.
- Las cadenas que si contienen un 1, éste va precedido y seguido de 0.

2.3 Analizador léxico. Construye un programa que localice la posición (fila y columna del primer símbolo de la cadena) en un fichero que contiene código fuente en C, en la que aparezca alguno de los siguientes elementos del lenguaje:

- Palabras reservadas `while`, `for`, `do` o `case`.
- Identificadores.
- Constantes numéricas enteras.
- Constantes numéricas en coma flotante.

2.4 Las expresiones regulares en Unix: estudia la ayuda que el sistema operativo Unix proporciona para el comando `grep`. A la vista de esta ayuda escribe los comandos necesarios para realizar la misma búsqueda que se pide en el ejercicio anterior.

Conviene destacar que una expresión regular denota con precisión un único subconjunto de Σ^* , mientras que el inverso no es cierto: un mismo conjunto admite representaciones diferentes, pudiendo incluso ser difícil a simple vista determinar si dos expresiones regulares identifican al mismo conjunto. No obstante, como veremos en la sección 3.2, disponemos de algoritmos para determinar si dos expresiones regulares son equivalentes.

2.1.1 Algunas propiedades de las ER

Se deja como ejercicio la justificación de las propiedades siguientes que lo requieran:

- el operador $+$ es asociativo y commutativo
- el operador concatenación es asociativo
- $r(s+t) = rs+rt$ (distributividad por la izquierda de la concatenación sobre la unión)
- $(r+s)t = rt+st$ (distributividad por la derecha de la concatenación sobre la unión)
- $(r^*)^* = r^*$ (idempotencia)
- $r+r=r$
- $r+\emptyset=r$
- $r\epsilon=\epsilon r=r$
- $r\emptyset=\emptyset r=\emptyset$

- $\epsilon^* = \epsilon$
- $\emptyset^* = \epsilon$
- $r^*r^* = r^*$
- $r^*r = rr^* = r^+$
- $r^+ + \epsilon = r^*$
- $r^* + \epsilon = r^*$
- $r \subset s \implies r^* \subset s^* \implies r^* + s^* = (r + s)^* = s^*$
- $(r + \epsilon)^* = r^*$
- $s(rs)^* = (sr)^*s$
- $s(rs)^+ = (sr)^+s$
- $(r + s)^* = (r^*s^*)^*$
- $(r + rs)^*r = r(r + sr)^*$

2.2 Equivalencia ER-AF

Dada una expresión regular r que denota un cierto conjunto regular $L(r)$, siempre es posible construir un autómata finito M que acepte $L(M) = L(r)$. Igualmente, dado un autómata M , siempre es posible describir el conjunto $L(M)$ mediante una³ expresión regular. Por ello, el conjunto de lenguajes regulares es idéntico al conjunto de conjuntos regulares. Los dos sentidos de la equivalencia anterior se pueden demostrar dando procedimientos constructivos, que proceden de forma recursiva.

Comenzaremos construyendo un autómata finito para cualquier expresión regular dada. Si hemos sido estudiantes aplicados y hemos realizado el ejercicio 2.3, nos habremos dado cuenta de que necesitamos implementar 4 funciones básicas para construir un programa que busque cadenas que casan con la expresión regular. Intuitivamente, podemos obtener el algoritmo básico, que es:

- Por cada símbolo de la cadena que no es un metacaracter (paréntesis, barra o estrella) añade dos estados conectados por ese símbolo.
- Por cada concatenación de expresiones añade una transición vacía entre el último estado de la primera y el primero de la segunda, de forma análoga a como se realiza la conexión en serie de dos circuitos eléctricos.

³En realidad, por muchas.

- Por cada unión de expresiones añade un nuevo estado de entrada y otro de salida. Conecta el primero con dos transiciones vacías a los primeros de cada expresión. A su vez, los dos últimos de cada expresión deben conectarse mediante transiciones vacías con el nuevo estado de salida, en analogía con una conexión en paralelo de dos circuitos.
- Por cada clausura de Kleene añade un nuevo estado de entrada que también será el de salida. Conecta este estado con el inicial del autómata correspondiente a la expresión y el de salida de éste con el nuevo estado mediante transiciones vacías.

Importa distinguir entre el algoritmo y el meta-algoritmo (programa) que para una expresión regular (cadena) construye un AFD (programa). El AFD a su vez procesa cadenas de entrada, pero su salida es 0/1 (sí/no).

Las transiciones vacías se han introducido para representar las frases sin contenido. Por ejemplo, en el lenguaje C una sentencia posible es `if(x==1){;}` que indica que tras el condicional se ejecuta una sentencia sin ningún contenido. Esto conduce a una reflexión sobre el significado del indeterminismo: ¿puede un ordenador ser indeterminista? Muchas veces se confunde el concepto de indeterminismo con el de (pseudo)aleatoriedad: si el pasado determina completamente el futuro, aunque sea aleatoriamente, el sistema es determinista. En un proceso indeterminista, varios futuros coexisten para el mismo pasado. Como los ordenadores actuales no admiten indeterminismo (y de momento sólo pseudo-aleatoriedad, es decir, quien conozca el funcionamiento del programa puede ganar al ordenador), éste se debe simular: el programa guardará una lista de los estados que están siendo visitados en ese momento.

Formalicemos estas ideas intuitivas recordando la definición que dimos de las expresiones regulares en la página 28, comenzamos diseñando autómatas que aceptan \emptyset , $\{\epsilon\}$ y $\{a\}$ ($a \in \Sigma$), tal y como aparecen representados en la fig.[2.1].



Figura 2.1: Autómatas finitos para ϵ , \emptyset y $a \in \Sigma$

De acuerdo con su definición, también son expresiones regulares aquellas que denotan los conjuntos obtenidos por unión, concatenación o clausura de conjuntos denotados por expresiones regulares. Si r y s son las expresiones regulares que representan los conjuntos aceptados por los autómatas $M^{[r]}$ y $M^{[s]}$ respectivamente, entonces es posible construir nuevos autómatas para aceptar los conjuntos $L(r+s)$, $L(rs)$ y $L(r^*)$, tal y como aparece representado simbólicamente en las figuras 2.2-2.2.

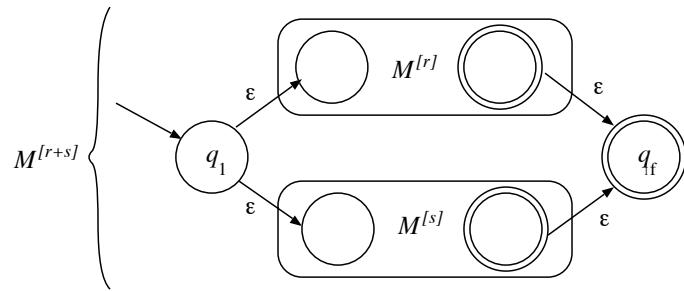


Figura 2.2: Representación simbólica del proceso para obtener el autómata $M^{[r+s]}$ a partir de los autómatas $M^{[r]}$ y $M^{[s]}$.

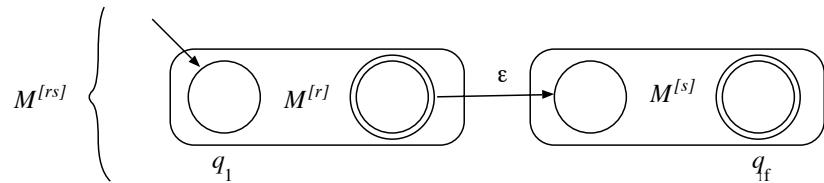


Figura 2.3: Representación simbólica del proceso para obtener el autómata $M^{[rs]}$ a partir de los autómatas $M^{[r]}$ y $M^{[s]}$.

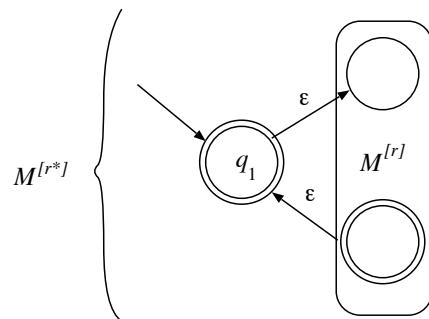


Figura 2.4: Representación simbólica del proceso para obtener el autómata $M^{[r^*]}$ a partir del autómata $M^{[r]}$.

Es claro, que los autómatas allí dibujados aceptan las cadenas de $L(r + s) = L(r) \cup L(s)$ (y sólo esas) mediante su colocación “en paralelo”, las de $L(rs) = L(r)L(s)$ cuando se colocan “en serie” y las de $L(r^*) = (L(r))^*$ cuando se permiten iteraciones a través de un bucle que requiere una subcadena perteneciente a $L(r)$ para ser atravesado. El uso de transiciones vacías garantiza que no se producen interferencias entre los autómatas. Por ejemplo, el diagrama de la figura 2.5 es incorrecto, pues puede ocurrir en

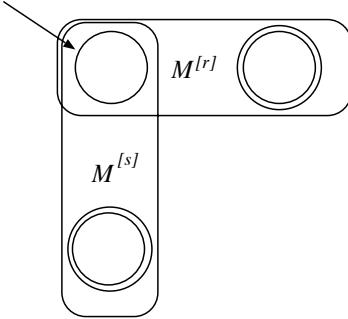


Figura 2.5: Conexión incorrecta de dos autómatas en paralelo.

algún caso que se acepte una cadena $w = xy$ donde x es una cadena tal que $\delta_x(q_1^r) = q_1$ mientras que $y \in s$, sin que la concatenación $w = xy$ pertenezca a ninguno de los dos conjuntos de aceptación.

Con este método constructivo, queda justificado que siempre es posible construir un AFI tal que $L(M)$ sea el dado por una expresión regular. Para el proceso inverso —dado un autómata finito M , obtener una expresión regular para $L(M)$ — también es posible dar un método constructivo. Para ello, basta con observar que los subconjuntos iniciales y de aceptación, Q_1 y F , son arbitrarios, por lo que es posible definir una matriz de lenguajes regulares $L(r_{ij})$ en el autómata M :

$$L(r_{ij}) = \{w \in \Sigma^* : q_j \in \delta_w(q_i)\}$$

Cada cadena incluida en el conjunto anterior permite llegar desde el nodo q_i hasta el q_j , de forma que:

$$L(M) = \bigcup_{q_i \in Q_1} \bigcup_{q_j \in F} L(r_{ij}) = L\left(\sum_{i,j} r_{ij}\right)$$

Además, también denota un lenguaje regular la expresión $r_{ij}^{(k)}$, definido como el conjunto de cadenas que llevan desde q_i hasta q_j sin pasar por nodos⁴ con subíndice superior a k . Efectivamente, $r_{ij}^{(k)}$ denota el conjunto $L'(r_{ij})$

⁴Cuando se dice que el camino pasa por q_m , se entiende que es un nodo interior al camino, es decir, un camino no pasa por sus nodos primero o último.

generado por un autómata M' construido a partir de M , pero donde se han suprimido algunos arcos. Con esta definición:

1. $r_{ij} = r_{ij}^{(n)}$
2. $r_{ij}^{(0)} = \sum_{a \in \Sigma \cup \{\epsilon\}, \delta_a(q_i) = q_j} a$
3. $r_{ij}^{(k)} = r_{ij}^{(k-1)} + r_{ik}^{(k-1)}(r_{kk}^{(k-1)})^* r_{kj}^{(k-1)}$

La ecuación (1) se deduce del hecho de que no hay nodos con índice superior a n , y por tanto $r_{ij}^{(n)}$ no excluye ningún arco y coincide con r_{ij} . El segundo punto (2) afirma que $r_{ij}^{(0)}$ corresponde a los arcos de transición, pues sólo estos no requieren ejercer otras transiciones que no sean las que salen de q_i . Es importante no olvidar que en el caso $i = j$ debe incluirse al menos ϵ en $r_{ij}^{(0)}$. Por último, la ecuación (3) es la que nos indica cómo construir una expresión regular para $L(M)$, ya que permite expresar recursivamente $L(M)$ en función de expresiones con índice k cada vez menor hasta llegar a los arcos de transición. Como los arcos se pueden describir trivialmente mediante expresiones regulares y el proceso sólo involucra operadores de unión, concatenación y clausura —como queda explícito en (3)—, se concluye que $L(M)$ es una expresión regular.

El significado de la ecuación (3) es que para ir desde q_i hasta q_j , o bien el camino no pasa por q_k (y pertenece a $r_{ij}^{(k-1)}$), o bien pasa por q_k . En este último caso, el camino puede descomponerse en una subcadena que lleva hasta q_k por primera vez (y que está contenida en $r_{ik}^{(k-1)}$), seguida de cero o más subcadenas que llevan de q_k hasta q_j pero sin pasar a través de q_k (contenidas en $(r_{kk}^{(k-1)})^*$) terminada por una subcadena que lleva desde q_k hasta q_j (y que pertenece a $r_{kj}^{(k-1)}$).

El algoritmo anterior puede ser programado con relativa facilidad. En casos sencillos, y sobre todo cuando $\text{card}(Q_1) = \text{card}(F) = 1$, el procedimiento puede aplicarse de forma intuitiva.

Ejemplo: Sea el autómata $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \delta, q_1, \{q_1\})$, con δ definido por la tabla

δ	0	1
q_1	q_1	q_2
q_2	q_3	q_1
q_3	q_2	q_3

La expresión regular buscada es $r_{11}^{(3)}$, que se ha de calcular recursivamente:

$$r_{11}^{(3)} = r_{13}^{(2)}(r_{33}^{(2)})^* r_{31}^{(2)} + r_{11}^{(2)};$$

cada componente se calcula usando de nuevo la ecuación (3):

$$r_{11}^{(2)} = r_{12}^{(1)}(r_{22}^{(1)})^*r_{21}^{(1)} + r_{11}^{(1)}$$

$$r_{13}^{(2)} = r_{12}^{(1)}(r_{22}^{(1)})^*r_{23}^{(1)} + r_{13}^{(1)}$$

$$r_{31}^{(2)} = r_{32}^{(1)}(r_{22}^{(1)})^*r_{21}^{(1)} + r_{31}^{(1)}$$

$$r_{33}^{(2)} = r_{32}^{(1)}(r_{22}^{(1)})^*r_{23}^{(1)} + r_{33}^{(1)}$$

que de la misma manera se ha de poner en término de $r_{ij}^{(0)}$, calculados de acuerdo con la ecuación (2), y simplificadas convenientemente:

$$\begin{aligned} r_{11}^{(1)} &= r_{11}^0(r_{11}^{(0)})^*r_{11}^{(0)} + r_{11}^{(0)} &= (0+\epsilon)(0+\epsilon)^*(0+\epsilon) + (0+\epsilon) \\ &= 0^* \\ r_{12}^{(1)} &= r_{11}^0(r_{11}^{(0)})^*r_{12}^{(0)} + r_{12}^{(0)} &= (0+\epsilon)(0+\epsilon)^*1 + 1 \\ &= 0^*1 \\ r_{13}^{(1)} &= r_{11}^0(r_{11}^{(0)})^*r_{13}^{(0)} + r_{13}^{(0)} &= (0+\epsilon)(0+\epsilon)^*\emptyset + \emptyset \\ &= \emptyset \\ r_{21}^{(1)} &= r_{21}^0(r_{11}^{(0)})^*r_{11}^{(0)} + r_{21}^{(0)} &= 1(0+\epsilon)^*(0+\epsilon) + 1 \\ &= 10^* \\ r_{22}^{(1)} &= r_{21}^0(r_{11}^{(0)})^*r_{12}^{(0)} + r_{22}^{(0)} &= 1(0+\epsilon)^*1 + \epsilon \\ &= 10^*1 + \epsilon \\ r_{23}^{(1)} &= r_{21}^0(r_{11}^{(0)})^*r_{13}^{(0)} + r_{23}^{(0)} &= 1(0+\epsilon)^*\emptyset + 0 \\ &= 0 \\ r_{31}^{(1)} &= r_{31}^0(r_{11}^{(0)})^*r_{11}^{(0)} + r_{31}^{(0)} &= \emptyset(0+\epsilon)^*(0+\epsilon) + \emptyset \\ &= \emptyset \\ r_{32}^{(1)} &= r_{31}^0(r_{11}^{(0)})^*r_{12}^{(0)} + r_{32}^{(0)} &= \emptyset(0+\epsilon)^*1 + 0 \\ &= 0 \\ r_{33}^{(1)} &= r_{31}^0(r_{11}^{(0)})^*r_{13}^{(0)} + r_{33}^{(0)} &= \emptyset(0+\epsilon)^*\emptyset + (1+\epsilon) \\ &= 1 + \epsilon \end{aligned}$$

Reuniendo todos los términos, la expresión $r_{11}^{(3)}$ resulta ser bastante compleja:

$$r_{11}^3 = 0^*1(10^*1)^*0(0(10^*1)^*0)^*0(10^*1)10^* + 0^*1(10^*1)^*10^* + 0^*.$$

El lector puede comprobar que si en este método se renumeran los estados de manera que el estado inicial y los de aceptación tengan los números más altos la expresión regular resultante es bastante sencilla.

Hay otra manera de construir la expresión regular correspondiente a un autómata finito (determinista o indeterminista) dado M [13, 14]; esta construcción se puede hacer de manera que las expresiones regulares obtenidas

sean muy sencillas. Para cada nodo del autómata q_i , definimos $r(q_i)$ como la expresión regular que representa el lenguaje

$$L(q_i) = \{w \in \Sigma^* : \delta_w(q_i) \cap F \neq \emptyset\},$$

es decir, el lenguaje de las cadenas que, partiendo de q_i , determinan al menos un camino que llega a alguno de los estados de aceptación. El lenguaje aceptado por el autómata $L(M)$, es la unión de los lenguajes correspondientes a los nodos del conjunto inicial Q_1 :

$$L(M) = \bigcup_{q_i \in Q_1} L(q_i)$$

es posible poner el lenguaje $L(q_i)$ para cualquier nodo q_i en función de los lenguajes $L(q_j)$ correspondientes a los nodos q_j hacia ninguno de los cuales hay transiciones directas (es decir, definidas en la tabla de transiciones $\tilde{\delta}$) desde q_i . En términos de expresiones regulares:

$$r(q_i) = \sum_{a \in \Sigma \cup \{\epsilon\}} \sum_{q_j \in \tilde{\delta}_a(q_i)} ar(q_j) \quad (2.1)$$

si $q_i \notin F$ y

$$r(q_i) = \epsilon + \sum_{a \in \Sigma \cup \{\epsilon\}} \sum_{q_j \in \tilde{\delta}_a(q_i)} ar(q_j) \quad (2.2)$$

si $q_i \in F$. El significado de estas ecuaciones es que cada transición que parte de q_i contribuye al lenguaje $L(q_i)$ de la manera siguiente: si con el símbolo a se puede pasar de q_i a q_j (es decir, $q_j \in \tilde{\delta}_a(q_i)$), las cadenas que son concatenación de a con las cadenas de $L(q_j)$ pertenecen al lenguaje $L(q_i)$. Además, si q_i es un estado de aceptación, la cadena vacía ϵ también pertenece al lenguaje $L(q_i)$.

Si se escribe una ecuación como las anteriores para cada nodo, se obtiene un sistema de *ecuaciones regulares* (en total, $\text{card}(Q)$ ecuaciones) que se pueden resolver para obtener las expresiones regulares $r(q_i)$ correspondientes a los nodos de Q_1 : la unión de estos representa al lenguaje aceptado por el autómata. El sistema de ecuaciones resultante sólo se puede resolver *por sustitución*⁵, intentando hallar los términos $r(q_i)$. Durante este proceso aparecen ecuaciones de la forma

$$r(q_j) = s + tr(q_j) \quad (2.3)$$

donde s y t son dos expresiones regulares cualesquiera. Es posible hallar $r(q_j)$ de estas ecuaciones si tenemos en cuenta que la ecuación anterior es equivalente a

$$r(q_j) = t^*s, \quad (2.4)$$

⁵Hay que notas, que, pesar de que las ecuaciones tienen la apariencia de sumas y productos, representan uniones y concatenaciones y, por tanto, no se las pueden aplicar los mismos métodos que a las ecuaciones lineales.

donde la expresión regular $r(q_j)$ ya está despejada. La equivalencia de (2.3) y (2.4) se puede demostrar rigurosamente por inducción. Intuitivamente, resulta claro que todas las cadenas representadas por s están también en el lenguaje representado por $r(q_j)$. Por tanto, si sustituimos s en la parte derecha de la ecuación (2.3), se ve que también las cadenas de $s + ts$ pertenecen al lenguaje representado por $r(q_j)$. Sucesivamente, se ve que las cadenas de $s + t^n s$, para cualquier n , también pertenecen al lenguaje $r(q_j)$.

El método puede quedar más claro con un ejemplo. Consideremos el autómata finito

$$M = (\{q_1, q_2, q_3, q_4\}, \{a, b\}, \delta, \{q_1\}, \{q_4\})$$

con la tabla de transiciones:

δ	a	b	ϵ
q_1	$\{q_2\}$	\emptyset	$\{q_3\}$
q_2	\emptyset	$\{q_2\}$	$\{q_4\}$
q_3	$\{q_4\}$	$\{q_3\}$	\emptyset
q_4	\emptyset	\emptyset	\emptyset

Las ecuaciones correspondientes, obtenidas según las reglas (2.1) y (2.2) son:

$$\begin{aligned} r(q_1) &= ar(q_2) + \epsilon r(q_3) \\ r(q_2) &= br(q_2) + \epsilon r(q_4) \\ r(q_3) &= ar(q_4) + br(q_3) \\ r(q_4) &= \epsilon \end{aligned}$$

y nos interesa el valor de $r(q_1)$. El valor de $r(q_4)$ se puede sustituir trivialmente en las ecuaciones para $r(q_2)$ y $r(q_3)$; después, se les aplica la regla de equivalencia de las ecuaciones (2.3) y (2.4) para hallar $r(q_2)$ y $r(q_3)$. Las nuevas ecuaciones son:

$$\begin{aligned} r(q_1) &= ar(q_2) + \epsilon r(q_3) \\ r(q_2) &= b^* \\ r(q_3) &= b^*a; \end{aligned}$$

donde se ve que $r(q_1) = ab^* + b^*a$.

Ejercicios

2.5 Encuentra una expresión regular para el conjunto de cadenas binarias cuyo valor es múltiplo de tres a partir del AFD que acepta dichas cadenas.

2.6 Explíquese el significado de las siguientes expresiones regulares y constrúyanse autómatas finitos deterministas para cada uno de los lenguajes denotados:

- 0^*
- $(0 + 1)^*$
- $(10)^*$
- $(0 + 1)^*10^*$
- $(1 + 10)^*$
- $(0 + 1)^*00$
- $(0 + 1)^*11(0 + 1)^*$
- $(0 + 1)^*(00 + 11)(0 + 1)^*$
- $(0 + 1)^*(00 + 11)$
- $(0 + 1)^4$

2.7 A partir del autómata finito que los acepta, calcúlense las expresiones regulares para cada uno de los siguientes lenguajes regulares de $\Sigma^* = \{0, 1\}^*$:

- cadenas con un número impar de unos;
- cadenas que no contienen la subcadena 101 ;
- cadenas que contienen dos dígitos consecutivos iguales;
- cadenas que no contienen más de dos ceros consecutivos.

2.8 Representa gráficamente y como expresión regular el autómata finito definido por $Q_1 = \{q_1\}$, $F = \{q_2, q_3\}$ y la siguiente tabla de transiciones:

Estado	0	1
q_1	q_2	q_3
q_2	q_1	q_3
q_3	q_2	q_2

2.3 Operaciones con conjuntos regulares

Existen operaciones bajo cuya aplicación los conjuntos regulares son estables, es decir, el conjunto que se obtiene como resultado es también regular. Ejemplo de ellas son las siguientes:

- unión, concatenación y clausura
- complementación
- intersección

- sustitución
- cociente
- inversión

La unión, concatenación y clausura de conjuntos regulares es regular por la misma definición de conjunto regular. Que el *complementario* ($L = \Sigma^* - L$) de un conjunto regular L es regular, se puede demostrar construyendo a partir de M —el AFD que acepta L —, un autómata M' que reconoce \bar{L} . Para ello, basta con tomar M' idéntico a M , con la salvedad de que el nuevo subconjunto de estados aceptables es $F' = Q - F$. Es evidente que $w \in L(M') \leftrightarrow w \notin L(M)$.

Como consecuencia inmediata de lo anterior, la *intersección* de dos conjuntos regulares también es regular, pues puede expresarse en términos de unión y complementación: $L_1 \cap L_2 = \overline{\overline{L}_1 \cup \overline{L}_2}$. Una forma, en general más sencilla, de construir el autómata que acepta $L_1 \cap L_2$ a partir de M_1 y M_2 los AFD que reconocen L_1 y L_2 respectivamente, es tomar

$$M = (Q_1 \times Q_2, \Sigma, \delta, (q_1^{[1]}, q_1^{[2]}), F_1 \times F_2)$$

donde

$$\delta_a(q^{[1]}, q^{[2]}) = (\delta_a^{[1]}(q^{[1]}), \delta_a^{[2]}(q^{[2]})).$$

Una *sustitución* es un homomorfismo $f : \Sigma^* \rightarrow P(\Lambda^*)$, tal que la imagen $f(a)$ de cualquier elemento $a \in \Sigma$ es un conjunto regular sobre otro alfabeto Λ . En el monoide final la operación definida es la concatenación de lenguajes, y en el inicial la concatenación de cadenas. Sabemos que:

- $f(xy) = f(x)f(y)$ (por definición de homomorfismo);
- $f(\epsilon) = \{\epsilon\}$ (la imagen del elemento neutro a través de un homomorfismo es el elemento neutro del monoide de llegada).

Dadas estas propiedades, se concluye trivialmente que la imagen $f(w)$ de una cadena $w \in \Sigma^*$ es un conjunto regular. En particular, $f(w)$ es concatenación de los conjuntos regulares que se obtienen al aplicar f a cada uno de los símbolos que componen la cadena. Si $w = a_1a_2a_3\dots$ entonces $f(w) = f(a_1)f(a_2)f(a_3)\dots$ No sólo $f(w)$ es regular, también la imagen de un conjunto R de cadenas lo es, siempre que R sea regular, dado que:

1. $f(a+b) = f(a) + f(b)$
2. $f(ab) = f(a)f(b)$
3. $f(a^*) = f^*(a)$

y por tanto, los conjuntos regulares son estables bajo sustituciones. Un caso particularmente interesante de sustitución se da cuando f es de la forma: $f : \Sigma^* \rightarrow \Lambda^*$ y cada cadena es sustituida por otra.

El *cociente* de dos cadenas se define como la operación inversa de la concatenación. Dado que la concatenación no es comutativa, debe distinguirse entre el cociente por la derecha y el cociente por la izquierda:

$$w = xa \iff x = wa^{-1}$$

$$w = ax \iff x = a^{-1}w$$

El cociente entre conjuntos se define entonces:

$$AB^{-1} = \{x \in \Sigma^* : \exists y \in B : xy \in A\}$$

$$B^{-1}A = \{x \in \Sigma^* : \exists y \in B : yx \in A\},$$

Otro concepto utilizado con frecuencia es el de *conjunto de colas* de la cadena $w \in \Sigma^*$ en el lenguaje L , definido como $w^{-1}L$. Si el conjunto de colas de w es no vacío ($w^{-1}L \neq \emptyset$), entonces diremos que w es un *prefijo* de L . A su vez, diremos que w es un *sufijo* del lenguaje L si $Lw^{-1} \neq \emptyset$.

En general, el cociente de un lenguaje regular R con un lenguaje cualquiera (no necesariamente regular) L es regular. En efecto, dado R existe un AFD que reconoce a R , por ejemplo $M = (Q, \Sigma, \delta, q_1, F)$. El AFD que reconoce a $C = RL^{-1}$ es $M' = (Q, \Sigma, \delta, q_1, F')$, con $F' = \{q \in Q : \exists y \in L : \delta_y(q) \in F\}$. Trivialmente:

$$\begin{aligned} x \in L(M') &\iff \delta_x(q_1) \in F' \\ &\iff \delta_y \delta_x(q_1) \in F, y \in L \\ &\iff xy \in R, y \in L \iff x \in C \end{aligned}$$

Luego $C = L(M')$, lo que demuestra que RL^{-1} es regular. Si el cociente se realizase por la izquierda, el proceso sería semejante, pero redefiniendo el estado inicial q_1 . Este pasaría en general a ser un subconjunto de estados iniciales Q_1 , que pueden reducirse a uno único mediante las técnicas de conversión a AFD explicadas en la sección 1.7.1.

Conviene destacar que el procedimiento anterior, no constituye por sí solo un procedimiento efectivo para encontrar el autómata M' , pues que la cadena y debe buscarse entre un conjunto eventualmente infinito de cadenas L , y el tiempo requerido para ello puede no estar acotado. Esto no ocurre si L está caracterizado como lenguaje regular o por una gramática independiente del contexto. Volveremos sobre esta cuestión de la resolución algorítmica de problemas más adelante en la sección 3.2.

Por último, se define w^R , la *cadena inversa* de la cadena w , de forma recursiva:

- $\epsilon^R = \epsilon$

- $a^R = a \quad \forall a \in \Sigma$
- $(xy)^R = y^R x^R \quad \forall x, y \in \Sigma^*$

También se define *lenguaje inverso* de un lenguaje L a

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

Obviamente, $\emptyset^R = \emptyset$. Con estas definiciones, el conjunto L^R es regular si y sólo si L es regular. Para probarlo, basta con observar que:

- $(L_1 \cup L_2)^R = L_1^R \cup L_2^R$
- $(L_1 L_2)^R = L_1^R L_2^R$
- $(L^*)^R = (L^R)^*$.

Esto permite obtener una expresión regular para L^R conocida una expresión para L , y viceversa, con el convenio de que r^R representa a la expresión regular que denota $(L(r))^R$.

Ejercicios

2.9 ¿Es la unión de infinitos conjuntos regulares también regular? ¿Y cualquier subconjunto de un conjunto regular? Justifica la respuesta.

2.10 Demuestra que si $A \oplus B$ (la diferencia simétrica de A y B) y A son regulares entonces B es regular.

2.11 Demuestra que si M_k es el AFD que acepta cadenas binarias cuyo valor es múltiplo de k , entonces $M_6 = M_2 \cap M_3$.

2.12 Razona la validez de las siguientes identidades:

1. $AA^{-1} = \emptyset$
2. $AA^{-1} = \{\epsilon\}$
3. $(AB)B^{-1} = A$
4. $(AB^{-1})B = A$
5. $(AB^{-1})C = A(B^{-1}C)$

2.13 Demuestra que el conjunto de prefijos $\text{Pr}(L)$ de un lenguaje regular L es regular: $\text{Pr}(L) = \{x \in \Sigma^* : xw \in L \text{ para alguna } w \in \Sigma^*\}$. Haz lo mismo con el conjunto $C(L) = \{y \in \Sigma^* : xyz \in L \text{ para algunas } x, z \in \Sigma^*\}$.

2.14 ¿Cómo se construiría a partir de M el AFD que acepta $f(L(M))$ siendo f una sustitución? Aplica el razonamiento para demostrar que $C = \{w \in \Sigma^* : \exists v \in L : |w| = |v|\}$ es regular.

2.15 Justifica que, para cualquier lenguaje L , $(L^*)^R = (L^R)^*$.

2.16 Demuestra que si $f : \Sigma^* \longrightarrow \Lambda^*$ es un homomorfismo y $R \subset \Lambda^*$ es regular entonces $f^{-1}(R) = \{x \in \Sigma^* : f(x) \in R\}$ es regular (estabilidad para la inversa del homomorfismo).

2.17 Demuestra que $A(BC)^{-1} = AC^{-1}B^{-1}$.

2.18 Sea la operación $C(L) = \{y : \exists x, z \in \Sigma^* : xyz \in L\}$. ¿Es la clase de los lenguajes regulares estable bajo esta operación?

2.19 Sea la operación P definida sobre una cadena w como sigue: si w contiene dos a consecutivas, cada b se convierte en una c ; si w contiene dos b consecutivas cada a se convierte en una c . ¿Es la clase de los lenguajes regulares sobre $\{a, b\}$ cerrada bajo P ?

Capítulo 3

Construcción de autómatas finitos

3.1 Lema de bombeo

Un problema que se plantea con frecuencia es el de diseñar un autómata finito (o alternativamente encontrar una expresión regular) para el reconocimiento de un lenguaje dado. Para ello, conviene preguntarse previamente si el conjunto en el que estamos interesados es efectivamente regular. Aunque no disponemos de un método general que responda a esta pregunta en todos los casos, existen al menos algunas propiedades que debe cumplir cualquier lenguaje regular y que, en caso de no ser satisfechas, permiten descartar la posibilidad de que el conjunto dado sea regular. Una de ellas es el *lema de bombeo*. Esencialmente, lo que recoge su enunciado es que los lenguajes regulares son reconocibles mediante autómatas con un número finito de estados, y por tanto, la cantidad de información que en ellos se puede almacenar es finita. En particular, al proceder al reconocimiento de una cadena lo suficientemente larga, el autómata debe pasar más de una vez por alguno de los estados. En ese caso, el camino recorrido presenta un bucle, y el fragmento de cadena que lo origina podría repetirse (“bombearse”) tantas veces como e quiera. Veamos el enunciado del lema de bombeo:

Lema 3.1 *Para todo conjunto regular L existe un número $n \in \mathbb{N}$ tal que*

$$w \in L : |w| \geq n \implies \text{para algún } x, y, z : \begin{cases} w = xyz \\ |xy| \leq n \\ |y| \geq 1 \\ xy^k z \in L \quad \forall k \in \mathbb{N} \end{cases}$$

Demostración. Si L es regular, existe M , un AFD que acepta L . Sea $n = |M|$ el número de nodos de M y $w = a_1 a_2 \dots a_m$ una cadena de Σ^* con $m \geq n$. Cada símbolo de w induce una transición a otro estado en

M , por lo que al cabo de m transiciones al menos algún nodo q_i ha sido visitado dos veces¹. Supongamos que la sucesión de estados inducida por w en M es $q_1 q_2 \dots q_i \dots q_j \dots q_{m+1}$ y que $q_j = q_i$ es el primer nodo que se repite en la sucesión. Si llamamos $x = a_1 a_2 \dots a_{i-1}$, $y = a_i a_{i+1} \dots a_{j-1}$, $z = a_j a_{j+1} \dots a_{m+1}$, resulta que $\delta_y(q_i) = q_i$ y por tanto $q_{m+1} = \delta_z(\delta_y)^k \delta_x(q_1)$ para cualquier $k \in \mathbb{N}$. Se concluye pues, que $xy^k z \in L$ para todos los valores de k . Además, $y \neq \epsilon$ por ser el autómata determinista, y $|xy| \leq n$, pues el primer nodo repetido puede aparecer como muy tarde en la posición $n + 1$ del camino. ■

Es importante notar que en ningún caso el lema de bombeo demuestra que un conjunto C dado sea regular. Se trata simplemente de un filtro que permite rechazar algunos conjuntos como candidatos a conjuntos regulares. Esto ocurre cuando, dado el conjunto C , no existe el número n al que se refiere el lema, lo cual se demuestra generalmente por reducción al absurdo: independientemente del valor de n , no hay forma de satisfacer las condiciones del lema. Ello requiere justificar que *existe* alguna cadena de longitud superior o igual a n que no puede ser descompuesta en tres fragmentos que satisfagan todas las condiciones del lema. La dificultad, pues, estriba en encontrar dicha cadena (en realidad, conjunto de cadenas, ya que al ser n desconocido la cadena elegida dependerá del valor de n).

Veámos un ejemplo de aplicación del lema. Dado el conjunto $C = \{w \in (0+1)^* : w \text{ contiene igual número de ceros que de unos}\}$, supongamos que existe n . Entonces podemos tomar, por ejemplo, la cadena $w = 0^{2n}1^{2n}$. Es evidente que no existe una fragmentación $w = xyz$ que satisfaga a un tiempo que $|xy| \leq n$, $|y| \geq 1$, $xy^k z \in L \forall k$, ya que la subcadena y debe estar compuesta de uno o más ceros, y al variar k se varía el contenido de ceros de la cadena sin variar simultáneamente el de unos. Por tanto, si $k \neq 1$ entonces $xy^k z \notin L$, y el conjunto C no es regular.

Ejercicios

3.1 Demuestra que $D = \{w \in \Sigma^* : w = 0^{n^2}\}$ no es regular.

3.2 Razónese qué subconjuntos de $\{0, 1\}^*$ de entre los siguientes son regulares:

- $\{0^{2n} : n \in \mathbb{N}\};$
- el conjunto de las cadenas que representan un número primo;
- el conjunto de las cadenas que representan múltiplos de 3;
- el conjunto de las cadenas cuya longitud es un cuadrado perfecto;

¹Basta con que $m = n$ ya que una cadena de m símbolos recorre $m+1$ nodos, incluidos los extremos.

- el conjunto de las cadenas que tienen el mismo número de ceros que de unos;
- el conjunto de las cadenas palíndromas (capicúa);
- el conjunto de los prefijos de $1/3$;
- el conjunto de los prefijos de $\sqrt{2}$;
- el conjunto de las cadenas cuyo contenido en ceros y unos difiere en un número par

3.3 Dado un lenguaje regular L , ¿son regulares los siguientes conjuntos?

- $\{xx^R : x \in L\}$;
- $\{x : xx^R \in L\}$;
- $\text{mitad}(L) = \{x : \exists y : |x| = |y|, xy \in L\}$, donde $|x|$ es la longitud de la cadena x .

3.4 Dada la cadena $s = abaababaababaab\dots$, ¿es $\text{Pr}(s) = \{x : x \text{ es un prefijo de } s\}$ regular?

Una consecuencia relacionada con el lema de bombeo es que no existen autómatas finitos que reconozcan cadenas prefijo de un número irracional, por ejemplo $\sqrt{2}$. Esto equivale a afirmar que $\sqrt{2}$ no es computable: cualquier ordenador tiene un número finito de estados (y la memoria accesible en la práctica está acotada) y por tanto el resultado obtenido sería periódico a partir de cierta cifra, y por tanto incorrecto.

3.2 Algoritmos y decidibilidad

Un *algoritmo* es una secuencia finita de instrucciones inambiguas y efectivas que constituyen un método general para obtener la respuesta a una pregunta determinada. Por efectivo debemos entender que éste proporciona la respuesta siempre en un tiempo finito. A las preguntas para las que existe dicho método se las denomina *decidibles*. No todas las cuestiones admiten una solución algorítmica². En este apartado estudiaremos la decidibilidad de preguntas como: ¿son equivalentes dos autómatas?; ¿genera un autómata dado M un conjunto vacío, finito o infinito de cadenas? Veremos que estas preguntas son decidibles, en el sentido de que disponemos de un método general que proporciona una respuesta en un tiempo finito en cualquier caso que se presente. Para ello enunciaremos los siguientes teoremas:

²Por ejemplo, la existencia o no de un cardinal intermedio entre el de los números naturales y el de los reales (ver página 112)

Teorema 3.1 *El lenguaje aceptado por un AFD es distinto del vacío si y sólo si existe una palabra w en este lenguaje cuya longitud es más pequeña que el número de estados del autómata:*

$$L(M) \neq \emptyset \iff \exists w \in L(M) : |w| < |M|$$

Demostración. La implicación hacia la izquierda es trivial, así que demostraremos la implicación hacia la derecha. Sea $n = |M|$. Si $L(M) \neq \emptyset$ entonces $L(M)$ contiene al menos una cadena w . Si $|w| < n$ ya está demostrado. Si $|w| \geq n$ entonces por el lema de bombeo $w = xyz$ con $|y| \geq 1$ y además $xz \in L(M)$, cuya longitud es estrictamente menor que la de w . Si $|xz| < n$ ya hemos terminado. Si no, se procede sucesivamente, aplicando el lema de bombeo a la nueva cadena obtenida. En cada paso la longitud de la cadena se acorta al menos en un símbolo, por lo que en algún momento debe obtenerse una subcadena cuya longitud es menor que n . ■

Teorema 3.2 *El lenguaje aceptado por un autómata es infinito si y sólo si existe alguna palabra en este lenguaje cuya longitud se encuentra entre dos cotas: el número de estados del autómata y dos veces el número de estados del autómata:*

$$L(M)\text{infinito} \iff \exists w \in L(M) : |M| \leq |w| < 2|M|$$

Demostración.

- (\Leftarrow) Es consecuencia del lema de bombeo y de que $|w| \geq |M|$. En efecto, si $w = xyz$ es la partición de la cadena que satisface el lema de bombeo, entonces xy^kz genera un conjunto infinito de cadenas al variar k , todas ellas de $L(M)$.
- (\Rightarrow) Si $L(M)$ es infinito, entonces existen en él cadenas de longitud superior a cualquiera dada, puesto que en caso contrario el conjunto sería finito. Tomemos $w : |w| \geq n = |M|$. Si $|w|$ está comprendida entre n y $2n$ ya está demostrado. Si $|w| \geq 2n$ entonces se puede aplicar el lema de bombeo con $k = 0$, obteniendo una cadena más corta. Como $w = xyz : |y| \geq 1, |xy| \leq n$, necesariamente $1 \leq |y| \leq n$, y la cadena se va acortando en fragmentos de longitud menor o igual que n . Es necesario, por tanto, que en algún momento la cadena que obtenemos tenga una longitud l comprendida entre n y $2n$ ($n \leq l < 2n$). ■

Cuestión : Si $L(M)$ es finito y $n = |M|$, ¿cuál es como máximo la longitud de las cadenas aceptadas por M ?

Teorema 3.3 *Existe un algoritmo que determina si dos autómatas son equivalentes (por tanto, también si dos expresiones regulares representan al mismo conjunto).*

Demostración. Para determinar si $L(M_1) = L(M_2)$ basta con construir —utilizando las técnicas de la sección 2.3— un autómata M_3 que acepte $L(M_3) = L(M_1) \oplus L(M_2)$ y aplicar el algoritmo del teorema 3.1 para determinar si $L(M_3) = \emptyset$. Trivialmente $L(M_1) = L(M_2) \iff L(M_3) = \emptyset$

■

Los teoremas anteriores proporcionan procedimientos cuya conclusión está garantizada. Por ejemplo, para determinar si el conjunto de cadenas que acepta M es vacío, finito o infinito basta con buscar entre las cadenas de longitud comprendida entre 0 y $2|M| - 1$. Dado que para un alfabeto finito Σ , el número de cadenas con longitud menor que $2|M|$ es finito y está perfectamente determinado, es seguro que obtendremos una respuesta.

Ejercicios

3.5 *Describábase un algoritmo para obtener el cociente $L_1L_2^{-1}$ de dos lenguajes regulares L_1 y L_2 .*

3.6 *Dése una cota para el número de cadenas de Σ^* con longitud menor que una dada l , si Σ contiene p símbolos.*

3.3 Minimización de autómatas finitos

Una propiedad muy interesante de los autómatas finitos deterministas es que si se establece un relación de orden de acuerdo con su tamaño —según el número de nodos que contiene—, el subconjunto de los autómatas finitos que generan un lenguaje dado L tiene un mínimo, es decir, existe un único autómata que es menor (estrictamente) que todos los demás. Esto no ocurre, por ejemplo, en el caso de los autómatas finitos indeterministas, donde podemos encontrar varios autómatas equivalentes tales que no existe otro equivalente de tamaño menor. Tampoco es posible si trabajamos con expresiones regulares o gramáticas. En cambio, si caracterizamos cada conjunto regular por el mínimo AFD que lo genera, esta correspondencia es biunívoca. Las consecuencias prácticas de esta propiedad son enormes. Por ejemplo, el disponer de una representación inambigua proporciona una gran potencia a los métodos de aprendizaje o inferencia de lenguajes regulares[11], característica que está lejos de ser conseguida en otros tipos de lenguajes.

La existencia de un mínimo hace que el problema de encontrar el AFD más pequeño que genera un lenguaje regular L dado esté bien definido. Es posible demostrar que cualquier otro AFD que acepte L es en realidad una copia del mínimo, en la que algunos nodos se han multiplicado de forma

innecesaria en forma de nodos equivalentes. Las transiciones que se originan en estos nodos equivalentes son las mismas que las del nodo original en el AFD mínimo, es decir, conducen al mismo estado o a estados equivalentes entre sí.

Existen diversos algoritmos para encontrar el autómata mínimo M que acepta L cuya justificación es posible a partir del teorema de Myhill y Nerode. Sin embargo, en muchos casos se conoce ya un autómata determinista A tal que $L = L(A)$ y lo que se desea es obtener la partición de los estados de A en las clases de nodos equivalentes (que son copia del nodo original del autómata mínimo M). Ambas cuestiones serán tratadas a continuación, pero antes introduciremos algunas definiciones³.

Definiciones:

- Se llama *orden* de una relación de equivalencia al número de sus clases. Si este es finito se dice que la relación es de *orden finito*.
- Dadas P_1 y P_2 , dos particiones del mismo conjunto, P_1 es un *refinamiento* de P_2 si cada clase de P_1 está totalmente incluida en alguna de P_2 .
- Sea R es una relación en Σ^* . Se dice que R es *invariante por la derecha* si $\forall z \in \Sigma^* : xRy \implies xzRyz$
- Dado un lenguaje $L \subset \Sigma^*$, se define R_L como la relación de equivalencia entre cadenas: $xR_L y \iff x^{-1}L = y^{-1}L$. Es decir, $xR_L y$ si los conjuntos de las colas de x y de y en L coinciden.
- Dado un autómata finito determinista M , se define la relación de equivalencia R_M entre cadenas de Σ^* mediante: $xR_M y \iff \delta_x(q_1) = \delta_y(q_1)$

Ejercicios

3.7 Comprueba que R_M es una relación de equivalencia invariante por la derecha y de orden finito, cuyo orden es menor o igual que el tamaño $|M|$ del autómata.

3.8 Comprueba que R_L es una relación de equivalencia invariante por la derecha.

3.9 Justifica que la relación R_L definida anteriormente se puede expresar también de la siguiente manera: $xR_L y \iff \forall z \in \Sigma^* (xz \in L \iff yz \in L)$

³Véanse también los conceptos de partición y relación de equivalencia en la página 110.

3.3.1 Teorema de Myhill y Nerode

Teorema 3.4 (*de Myhill y Nerode*) Dado M , un autómata finito determinista:

- (a) La partición de Σ^* inducida por R_M es un refinamiento de la partición inducida por $R_{L(M)}$
- (b) $L \subset \Sigma^*$ es regular $\iff R_L$ es de orden finito.

Demostración.

- (a) Esta proposición es equivalente a la implicación:

$$xR_My \implies xR_{L(M)}y.$$

Dada una cadena w , vamos a denotar como q_w el nodo tal que $q_w = \delta_w(q_1)$ y por $L(q_w)$ el lenguaje generado por M cuando se toma q_w como estado inicial. Por la definición de cociente:

$$\begin{aligned} w^{-1}L(M) &= \{z \in \Sigma^* : wz \in L(M)\} = \\ &= \{z \in \Sigma^* : \delta_z \delta_w(q_1) \in F\} = L(q_w) \end{aligned}$$

Esto expresa que las colas en lenguaje definido por el autómata de una cadena w dada son las cadenas que conducen a estados de aceptación partiendo del nodo q_w al que se llega desde q_1 mediante w . Como consecuencia de lo anterior, si x e y conducen al mismo estado, sus conjuntos respectivos de colas coinciden:

$$\begin{aligned} xR_My &\iff q_x = q_y \implies L(q_x) = L(q_y) \iff \\ &\iff x^{-1}L(M) = y^{-1}L(M) \iff xR_{L(M)}y, \end{aligned}$$

es decir, $xR_My \implies xR_{L(M)}y$ (aunque no necesariamente al revés), lo que implica que la partición establecida por R_M es un refinamiento de la establecida por $R_{L(M)}$.

- (b) (\implies) Si L es regular, existe un autómata finito M de tamaño n tal que $L = L(M)$. Como R_M es un refinamiento de R_L , y además es de orden finito menor que n se satisface:

$$n \geq \text{orden de } R_M \geq \text{orden de } R_L$$

En otras palabras, como mucho existen n conjuntos de colas distintos en Σ^* (uno por cada nodo del AFD, pudiendo además algunos de ellos coincidir).

(b) (\Leftarrow) Para probar que L es regular basta con encontrar M , un AFD tal que $L = L(M)$. Para ello aprovechamos el hecho de que R_L establece una partición de Σ^* en un número finito de clases. Interpretaremos que cada uno de dichos subconjuntos es un estado, y que se pueden efectuar transiciones entre ellos de la siguiente forma: si la cadena w pertenece a un subconjunto, el símbolo a produce una transición al estado en el que se encuentra wa . Rigurosamente, se definen:

- $Q = \{\zeta[x]\}$: clases de equivalencia de R_L , el conjunto (finito) de las clases de equivalencia de la partición definida en Σ^* por la relación R_L ;
- $q_1 = \zeta[\epsilon]$, la clase de equivalencia a la que pertenece ϵ ;
- $F = \{\zeta[x] : x \in L\}$, el conjunto de las clases de equivalencia a las que pertenecen las palabras del lenguaje;
- $\delta_a(\zeta[x]) = \zeta[xa]$, es decir, si el autómata está en el estado asociado a la clase de equivalencia a la que pertenece x , el símbolo $a \in \Sigma$ produce una transición al estado que representa a la clase de equivalencia donde se encuentra xa .

Con estas definiciones $\delta_x(q_1) = \delta_x(\zeta[\epsilon]) = \zeta[x]$, y como hemos definido $\zeta[x] \in F \iff x \in L$, el autómata sólo acepta cadenas pertenecientes a L . La demostración se completa probando que tanto la definición de la función δ como la del subconjunto F son consistentes, es decir, que no dependen del elemento x que se elija como representante de la clase $\zeta[x]$.

■

Ejercicios

3.10 Demuestra, a partir de que R_L es invariante por la derecha, que la definición de la función δ es consistente: $xR_Ly \implies \delta_a(\zeta[x]) = \delta_a(\zeta[y])$. Demuestra también la consistencia de la definición de F .

3.11 Dado $\Sigma = \{a, b\}$, se define el conjunto de lenguajes finitos: $L_n = \{wcw' : w, w' \in \Sigma \wedge w \neq w'\}$. Demuestra que $L = \bigcup_{n \geq 0} L_n$ no es regular.

Corolario 3.1 Dado L regular existe un único (salvo reordenación de las etiquetas de los nodos) autómata finito determinista M tal que $L = L(M)$ y cuyo número de nodos sea mínimo.

Demostración. El autómata M es el que se ha definido en el teorema anterior a partir de R_L , que tiene tantos estados como clases presenta R_L . En efecto, cualquier autómata M' equivalente a M genera también el lenguaje

L y define una relación $R_{M'}$ —cuyo índice es como mucho $|M'|$ — que origina un refinamiento de la partición inducida por R_L . Por tanto:

$$|M'| \geq \text{índice de } R_{M'} \geq \text{índice de } R_{L(M)} = |M|$$

Luego cualquier otro autómata determinista equivalente contiene un número mayor o igual de estados. Además, en el caso particular de que $|M| = |M'|$, entonces M y M' son idénticos, salvo reordenaciones triviales de los nodos, pues el refinamiento tiene el mismo número de clases que la partición más gruesa y, por tanto, sólo pueden ser iguales. ■

3.3.2 Algoritmos de minimización

En el apartado anterior hemos visto cómo es posible construir el AFD mínimo que acepta un cierto lenguaje regular L a partir de las clases de equivalencia de R_L . Sin embargo, no se proporcionaba un método efectivo para obtener dichas clases. El procedimiento habitual es partir de A , un AFD ya conocido⁴ que acepta $L = L(A)$, y se busca el mínimo autómata equivalente a A . Para ello demostraremos que es equivalente hablar de partición en L y de partición en los estados de A . Si los nodos del autómata inducen una partición en Σ^* definida por R_A , la partición R_L de Σ^* induce una partición de los estados de A que conduce al autómata minimizado. Esta partición viene dada por los nodos que son accesibles mediante cadenas de una misma clase de R_L .

Dado $A = (Q, \Sigma, \delta, q_1, F)$, llamaremos $L(q_k)$ al lenguaje regular generado cuando se toma q_k como nodo inicial en A , de forma que $L(q_k) = \{w \in \Sigma^* : \delta_w(q_k) \in F\}$. En particular: $L(A) = L(q_1)$. Esto nos permite definir la siguiente relación de equivalencia entre nodos:

$$q_i \equiv q_j \iff L(q_i) = L(q_j)$$

El autómata minimizado se construye como $M = (Q', \Sigma, \delta', [q_1], F')$ con

- $Q' = \{[q] : q \text{ accesible desde } q_1\}$
- $F' = \{[q] : q \in F\}$
- $\delta'_a([q]) = [\delta_a(q)]$

Trivialmente $L(M) = L(A)$, ya que:

$$\begin{aligned} w \in L(M) &\iff \delta'_w([q_1]) \in F' \\ &\iff [\delta_w(q_1)] \in F' \\ &\iff \delta_w(q_1) \in F \iff w \in L(A) \end{aligned}$$

⁴Muchas veces A ha sido construido a partir de una expresión regular siguiendo las técnicas del apartado 2.2.

Por tanto M es un autómata equivalente a A cuyo tamaño es igual o menor que el de A . Además $R_M = R_{L(A)}$, lo cual sólo ocurre para el AFD mínimo (siempre y cuando no haya nodos inútiles), tal y como vimos en el apartado [3.1]. En efecto, si definimos q_x y q_y como $q_x = \delta_x(q_1)$ y $q_y = \delta_y(q_1)$:

$$\begin{aligned} xR_Ly &\iff x^{-1}L = y^{-1}L \\ &\implies L(q_x) = L(q_y) \\ &\implies q_x \equiv q_y \implies xR_My \end{aligned}$$

Y por tanto, $R_L = R_M$. La ecuación anterior muestra además que la partición de estados inducida por \equiv es la misma que la inducida por R_L al agrupar los nodos accesibles por cadenas de una misma clase, si se excluyen los nodos inútiles.

Ejercicios

3.12 Demuestra que la definición de δ' es consistente, es decir que no depende del elemento representante de la clase elegido: $q_i \equiv q_j \implies \delta'([q_i]) = \delta'([q_j])$.

3.13 Demuestra que si M es mínimo, entonces contiene como máximo un nodo inútil.

En la práctica, para determinar la partición en los nodos de A se puede proceder de forma recursiva empezando desde la partición trivial (todos los nodos son equivalentes). Sabemos que $L(q_i) \neq L(q_j) \iff q_i \not\equiv q_j$. La condición $L(q_i) \neq L(q_j)$ significa que existe alguna cadena w que pertenece a $L(q_i)$ pero no a $L(q_j)$ o viceversa. Para $w = \epsilon$ esto significa que:

$$q_i \in F \wedge q_j \notin F \implies q_i \not\equiv q_j$$

Lo cual nos permite establecer una partición más fina que la trivial: las dos clases de equivalencia son F y $Q - F$.

Para cadenas w de longitud 1 se obtiene:

$$\delta_a(q_i) \in F \wedge \delta_a(q_j) \notin F \implies q_i \not\equiv q_j$$

Y para cadenas w de longitud arbitraria:

$$\delta_a(q_i) \not\equiv \delta_a(q_j) \implies q_i \not\equiv q_j$$

Por tanto, un posible algoritmo para obtener la partición P sería el siguiente:

Algoritmo 3.1 (Minimización del número de estados de un AFD)

Entrada: Un AFD $A = (Q, \Sigma, \delta, q_1, F)$ con transiciones definidas para todos los estados y símbolos de entrada.

Salida: Un AFD $M = (Q', \Sigma, \delta', q'_1, F')$ que acepta el mismo lenguaje y que tiene el mínimo número de estados posible.

Método:

1. Establézcase una partición inicial P de Q que contiene dos clases de equivalencia: los estados de aceptación, F , y los estados de no aceptación $Q - F$.

2. Para cada clase G de P :

- Divídase G en subclases tales que dos estados q_i y q_j están en la misma subclase si y sólo si para todos los símbolos $a \in \Sigma$, $\delta(q_i, a)$ y $\delta(q_j, a)$ pertenecen la misma clase de P .

- Substitúyase G en la nueva partición P' por el conjunto de todas las subclases creadas dentro de él.

3. Si $P = P'$, tómese $P_{\text{final}} = P$; si no, tómese $P = P'$ y ejecútese el paso 2.

4. Elíjase uno de los estados en cada clase de la partición P_{final} como estado representativo de la clase. Las transiciones que existan entre los estados representativos de cada clase serán las transiciones del nuevo autómata M . El estado inicial de M , q'_1 , será el estado representativo de la clase que contenía a q_1 en A . El conjunto F' de estados de aceptación de M' será el de los estados representativos que estuviesen originalmente en F .

5. (opcional) si M tiene un estado de absorción, es decir, un estado q_{abs} que no es de aceptación y tiene transiciones hacia sí mismo con todos los símbolos de Σ , elimínese este estado.

6. Elimínense todos los estados que no son accesibles desde el estado inicial.

Ejercicios

3.14 Minimiza el autómata M_6 obtenido en el ejercicio 2.11.

3.15 Minimícese el autómata M definido por $F = \{q_1, q_3\}$ y la tabla de transición

	a	b
q_1	q_2	q_1
q_2	q_5	q_4
q_3	q_5	q_1
q_4	q_1	q_6
q_5	q_2	q_6
q_6	q_3	q_6

y calcula la expresión regular para el lenguaje aceptado.

3.16 Constrúyase un AFI que acepte cadenas que contengan la subcadena 111 y desarrollese el AFD equivalente. Minimícese el AFD obtenido.

3.17 Construye el AFD mínimo que reconoce la expresión regular $(a+ab)^*a$. Haz lo mismo para la expresión $a(a+ba)^*$ y comprueba que se llega al mismo resultado. Como consecuencia, describe un algoritmo que permita decidir la equivalencia de dos expresiones regulares.

3.18 Describe un algoritmo que permita minimizar máquinas de Moore.

Capítulo 4

Gramáticas

4.1 Introducción

En la segunda mitad de la década de los 50, Noam Chomsky idea una teoría matemática para especificar lenguajes basada en el concepto de *gramática generativa*. El modelo que propone introduce los conceptos de gramaticalidad y creatividad: “todo hablante nativo posee una cierta intuición de la estructura de su lengua que le permite, por una parte, distinguir las frases gramaticales de las frases agramaticales y, por otra, comprender y emitir infinitud de frases inéditas. Del mismo modo, una gramática deberá rendir cuenta explícitamente de todas las frases gramaticales de la lengua considerada” (Chomsky, 1957).

La definición de gramática debe dar reglas con las que generar cadenas o frases pertenecientes al lenguaje que caracteriza, permitiendo, asimismo, el reconocimiento de una cadena arbitrariamente dada como perteneciente o no a éste¹.

Es útil, para comprender el concepto de gramática, volver por un momento a la clase de Lengua en la escuela secundaria, en la que el profesor nos pide que analicemos sintácticamente la frase *Juan estudia informática en la Universidad*. Como estudiantes aventajados, no tenemos ningún problema en realizar este ejercicio (cuyo resultado se muestra en la figura 4.1). Observamos que el árbol del análisis parte de la unidad de información *oración*, que en el siguiente paso es reescrita como un par de sintagmas (nominal y verbal). Cada uno de estos sintagmas vuelve a ser reescrito como otra cadena de elementos, y así sucesivamente hasta llegar a los constituyentes de la frase. Las diferentes reescrituras pueden ser descritas mediante la aplicación sucesiva de las siguientes reglas:

¹En materia de lenguajes de programación, este segundo aspecto es el más importante, un programa no es más que una cadena de caracteres que el compilador debe analizar y traducir a la luz de las reglas enunciadas en la gramática, detectando las incorrecciones sintácticas que encuentre y que le impidan compilarlo.

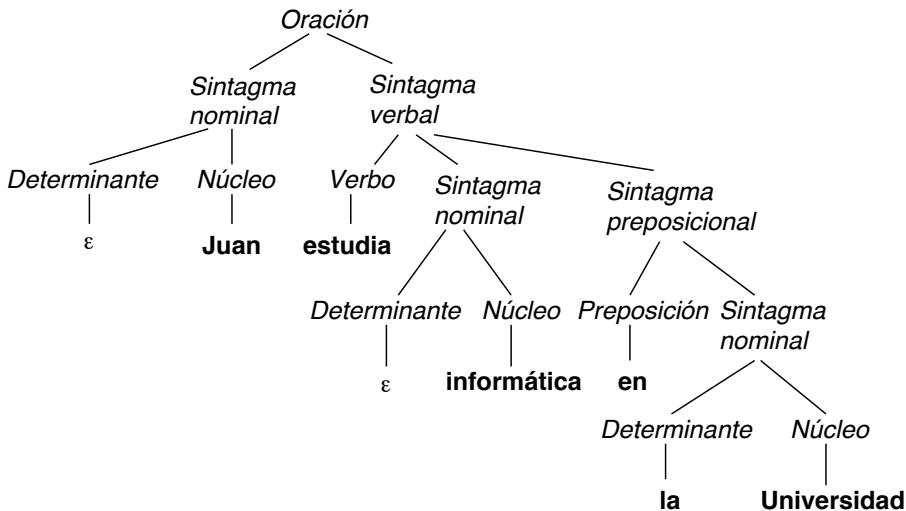


Figura 4.1: Árbol de análisis sintáctico de una frase de la lengua castellana.

- $\langle oración \rangle \rightarrow \langle sintagma nominal \rangle \langle sintagma verbal \rangle$
- $\langle sintagma nominal \rangle \rightarrow \langle determinante \rangle \langle núcleo \rangle$
- $\langle sintagma verbal \rangle \rightarrow \langle verbo \rangle \langle sintagma nominal \rangle \langle sintagma preposicional \rangle$
- $\langle sintagma preposicional \rangle \rightarrow \langle preposición \rangle \langle sintagma nominal \rangle$
- $\langle determinante \rangle \rightarrow \epsilon \mid la$
- $\langle núcleo \rangle \rightarrow Juan \mid informática \mid universidad$
- $\langle verbo \rangle \rightarrow estudia$
- $\langle preposición \rangle \rightarrow en$

El símbolo $|$ debe leerse “o”. Observemos en la anterior lista de reglas que existen dos tipos de elementos; por un lado, aparecen las categorías sintácticas y que en nuestro contexto denominaremos como *variables* o *no terminales* (encerradas entre los símbolos \langle y \rangle); y por otro, los llamados *símbolos terminales* (en negrita). Aunque la frase está compuesta sólo por terminales, la corrección sintáctica de la misma está dada por las reglas de reescritura de las variables.

Ejercicio

4.1 Repetir el análisis anterior para la cadena $\omega = \mathbf{a} * (\mathbf{a} + \mathbf{a})$ cuando la lista de reglas de reescritura es la siguiente:

1. $E \rightarrow E + E$
2. $E \rightarrow E * E$
3. $E \rightarrow (E)$
4. $E \rightarrow \mathbf{a}$

Aquí, E es una variable y $\{+, *, (,), \mathbf{a}\}$ es el conjunto de símbolos terminales. Señalemos, por otra parte, que con esta gramática podemos generar cadenas de longitud arbitrariamente grande, esto es, el conjunto de cadenas generables es infinito (*¿numerable o no numerable?*).

Ahora estamos en condiciones de dar una definición formal de gramática.

4.2 Definición y clasificación de las gramáticas

Definición: Una gramática es una cuádrupla

$$G = (V_N, V_T, S, P),$$

donde V_N es un conjunto finito de símbolos variables (no terminales); $V_T = \Sigma$, un conjunto finito de símbolos terminales; $S \in V_N$, el símbolo o variable inicial; y P , un conjunto finito de reglas de derivación.

Lógicamente se debe verificar que V_N y V_T sean disjuntos: $V_N \cap V_T = \emptyset$. Usaremos V para denotar al conjunto de todos los símbolos, tanto variables como terminales: $V = V_N \cup V_T$. Las reglas de derivación $r \in P$ indican como sustituir variables y pueden expresarse como aplicaciones

$$r : V^* V_N V^* \longrightarrow V^*,$$

donde V^* denota al conjunto de todas las cadenas que se pueden formar con los símbolos de V . Es decir, las reglas transforman cadenas de V^* que contienen al menos una variable en cadenas de V^* . A lo largo de este libro representaremos las variables por letras latinas mayúsculas, $V_N = \{A, B, C, \dots\}$, los terminales por letras latinas minúsculas, $V_T = \{a, b, c, \dots\}$ y las cadenas de V^* por letras griegas minúsculas ($\alpha, \beta, \gamma, \dots$).

Las cadenas generadas por la gramática son aquellas que se pueden obtener partiendo del símbolo inicial S mediante la aplicación sucesiva de reglas de P . Este conjunto de cadenas derivables desde S se denomina *lenguaje generado por la gramática G*. Simbólicamente,

$$L(G) = \{\omega \in V_T^* : S \xrightarrow{*} \omega\},$$

donde el asterisco indica que la derivación se puede realizar en un número arbitrario de pasos, y se podría leer como *produce en cero o más pasos*. Más concretamente, el símbolo $\xrightarrow{*}$ representa a la relación reflexiva y transitiva más pequeña que contiene a \longrightarrow , esto es, $\xrightarrow{*}$ satisface, para cualesquiera cadenas $\alpha, \beta, \gamma \in V^*$:

$$\begin{aligned} \alpha \longrightarrow \beta &\implies \alpha \xrightarrow{*} \beta \\ \alpha \xrightarrow{*} \alpha \\ \alpha \xrightarrow{*} \beta \wedge \beta \longrightarrow \gamma &\implies \alpha \xrightarrow{*} \gamma \end{aligned}$$

Diremos que dos gramáticas son equivalentes si generan el mismo lenguaje, es decir,

$$G \equiv G' \iff L(G) = L(G')$$

Según la complejidad de las reglas que definen una gramática, ésta puede ser de uno de los cuatro tipos ($\mathcal{G}_0, \mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$) que constituyen la llamada *jerarquía de Chomsky*:

Gramáticas estructuradas por frases o de tipo 0: son las más generales.

Sus reglas no tienen ninguna restricción. Generan los llamados *lenguajes recursivamente numerables* o *lenguajes sin restricciones*, aunque a pesar de este nombre no son capaces de representar a los lenguajes naturales.

Gramáticas sensibles al contexto o de tipo 1: sus reglas son de la forma $\alpha \rightarrow \beta$ ($\alpha, \beta \in V^*, |\alpha| \leq |\beta|$, α contiene al menos una variable), es decir, las derivaciones sucesivas no pueden disminuir la longitud de la cadenas de símbolos anteriores. Se puede demostrar que este tipo de reglas son equivalentes a las de la forma $\alpha A \beta \rightarrow \alpha \gamma \beta$, ($\gamma \in V^+$), de ahí el nombre de este tipo de gramática (la sustitución de A por γ depende del contexto en el que aparece A).

Gramáticas independientes del contexto o de tipo 2: sus reglas tienen la forma $A \rightarrow \alpha$, ($A \in V_N$, $\alpha \in V^*$). A diferencia del tipo anterior, la sustitución de A por α se realiza independientemente del lugar en el que aparezca A . La mayor parte de los lenguajes de programación están generados por gramáticas de este tipo (aumentadas con algunos elementos contextuales necesarios para la semántica del lenguaje); de ahí la importancia de su estudio.

Gramáticas regulares o de tipo 3: son las más simples, sus reglas son de uno de los siguientes tipos: $A \rightarrow aB$, $A \rightarrow a$, $A \rightarrow \epsilon$, ($A, B \in V_N$, $a \in V_T$), y, como veremos a continuación, generan los mismos lenguajes que son reconocidos por un autómata finito.

Los lenguajes generados por los últimos tres tipos son llamados por el mismo nombre que la gramática respectiva. Obsérvese que cada tipo de gramática está incluido en el tipo anterior, esto es:

$$\mathcal{G}_3 \subset \mathcal{G}_2 \subset \mathcal{G}_1 \subset \mathcal{G}_0.$$

NOTA: El estudio detallado de los tipos 0 y 1 se sale del marco de este libro y no serán tratados.

4.3 Gramáticas regulares

Las gramáticas regulares, o de tipo 3, tienen una relación muy estrecha tanto con los lenguajes del mismo nombre, como con los autómatas finitos que los reconocen; de hecho, según veremos en esta sección, los lenguajes generados por las gramáticas regulares son precisamente los regulares, lo que explica, por otra parte, el abuso del calificativo *regular*. Recordemos que una gramática $G = (V_N, V_T, P, S)$ es regular si todas las reglas $r \in P$ son de uno de los tipos siguientes:

$$\begin{aligned} r : V_N &\longrightarrow V_T V_N \\ r : V_N &\longrightarrow V_T \\ r : V_N &\longrightarrow \epsilon \end{aligned}$$

4.3.1 Gramática regular asociada a un autómata

Comencemos observando el autómata finito de la figura 4.2. El lector puede comprobar que el lenguaje reconocido por este autómata es $(0+1)^*10$, el conjunto de cadenas sobre el alfabeto 0, 1 que acaban en 10. Supongamos

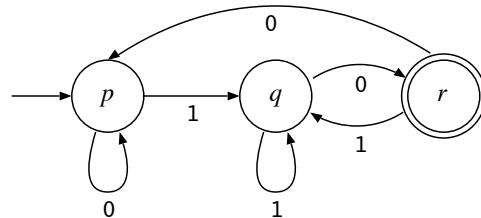


Figura 4.2: Autómata que reconoce el lenguaje $(0+1)^*10$.

que tomamos una cadena $\omega = 1001110$ de este lenguaje. Veremos su recorrido a través del autómata escribiendo en cada instante, a continuación de la subcadena leída, el estado en el que se encuentra el autómata. Obtenemos la secuencia

$$p \longrightarrow 1q \longrightarrow 10r \longrightarrow 100p \longrightarrow 1001q \longrightarrow 10011q \longrightarrow 100111q \longrightarrow 1001110$$

que se puede interpretar como una generación de la cadena en una gramática. Para obtener la gramática especificamos las variables como los estados del AFD, y a partir de los arcos obtenemos las reglas de reescritura:

$$\begin{aligned} p &\longrightarrow 0p \mid 1q \\ q &\longrightarrow 0r \mid 1q \\ r &\longrightarrow 0p \mid 1q \end{aligned}$$

El símbolo inicial de la gramática representa al estado inicial del autómata. Para completar el proceso hay que conseguir que la derivación acabe cuando

en el autómata se alcanza un estado de aceptación; esto se logra en nuestro ejemplo añadiendo la producción

$$q \longrightarrow 0,$$

o, alternativamente,

$$r \longrightarrow \epsilon.$$

Observemos que todas las producciones que hemos necesitado son de una de las formas que corresponden a una gramática regular.

Resumiendo, dado un autómata $M = (Q, \Sigma, \delta, q_1, F)$, obtenemos una gramática $G = (V_N, V_T, P, S)$ tal que $L(M) = L(G)$, tomando

- $V_N = Q$;
- $V_T = \Sigma$;
- $S = q_1$;
- para cada transición del autómata $\delta_a(p) = q$, añadimos a P la regla $p \longrightarrow aq$;
- para cada estado $q \in F$ añadimos a P la regla $q \longrightarrow \epsilon$, o, alternativamente, tantas reglas de la forma $p \longrightarrow a$ como transiciones del tipo $\delta_a(p) = q$ con $q \in F$ tenga el AFD.

4.3.2 Autómata asociado a una gramática regular

El proceso inverso también es muy sencillo de realizar, si bien, en el caso más general, el autómata que se obtiene es indeterminista. Consideremos la gramática regular $G = (V_N, V_T, P, S)$; el autómata $M = (Q, \Sigma, \delta, q_1, F)$, tal que $L(M) = L(G)$, se puede encontrar mediante:

- $Q = V_N \cup \{Z\}$ con Z un símbolo nuevo;
- $\Sigma = V_T$;
- $q_1 = S$;
- $F = \{Z\}$
- por cada regla $r \in P$ de la forma $A \longrightarrow aB$ definimos la transición $\delta_a(A) = B$;
- por cada regla $r \in P$ de la forma $A \longrightarrow a$ definimos la transición $\delta_a(A) = Z$;
- por cada regla $r \in P$ de la forma $A \longrightarrow \epsilon$ definimos la transición $\delta_\epsilon(A) = Z$

Ejemplo: Para la gramática siguiente encontramos el autómata de la figura 4.3.

$$\begin{aligned} A &\longrightarrow 0A \mid 0B \mid \epsilon \\ B &\longrightarrow 1C \mid 0D \mid 0 \\ C &\longrightarrow 0C \mid 1D \mid \epsilon \\ D &\longrightarrow 0D \end{aligned}$$

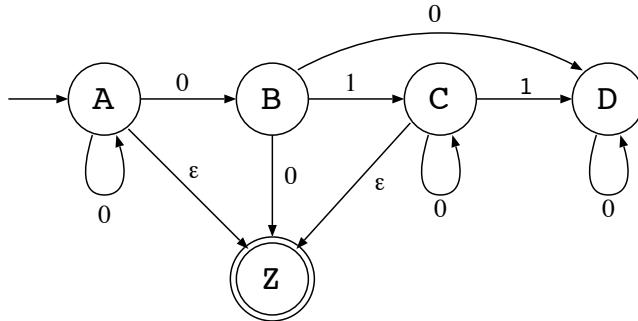


Figura 4.3: Autómata equivalente a la gramática del ejemplo.

Ejercicios

4.2 Constrúyase una gramática regular para los lenguajes siguientes:

- cadenas acabadas en 00;
- cadenas con dos unos consecutivos;
- cadenas con un 1 en la antepenúltima posición;
- cadenas de longitud 4.

Dibuja en algún caso el arbol de derivación de alguna cadena particular del lenguaje.

4.3 Demuéstrese que el conjunto de lenguajes regulares es idéntico al conjunto de lenguajes generables por gramáticas regulares.

4.4 Gramáticas independientes del contexto (GIC)

La gran importancia de las *gramáticas independientes del contexto* o *gramáticas incontextuales* reside en el hecho de que la mayor parte de los lenguajes de programación se representan mediante una gramática de este tipo. Estas gramáticas suponen una ampliación sobre las gramáticas regulares. Se caracterizan porque sus reglas son de la forma

$$r : V_N \longrightarrow V^*,$$

es decir, la parte derecha de la regla puede ser cualquier cadena de símbolos terminales y no terminales, si bien la parte izquierda está restringida a una sola variable. Un ejemplo de GIC estaría dado por

$$\begin{aligned} S &\rightarrow 0B \mid 1A \\ A &\rightarrow 0 \mid 0S \mid 1AA \\ B &\rightarrow 1 \mid 1S \mid 0BB \end{aligned}$$

que, como el lector puede intentar probar, representa al lenguaje formado por las cadenas que contienen igual número de ceros que de unos. Recorremos que en la sección 3.1 demostramos que este lenguaje no era regular, con lo que por fin hemos aumentado el número de lenguajes que podemos estudiar. A los lenguajes generados por GIC les llamaremos *lenguajes independientes del contexto* o, abreviadamente, LIC.

Ejercicio

4.4 Escribe gramáticas independientes del contexto para los conjuntos:

- cadenas capicúa;
- expresiones regulares sobre el alfabeto $\{0, 1\}$;
- cadenas que no son capicúa;
- cadenas del tipo $a^{n-1}b^n$.

4.4.1 Derivación y árbol de derivación

A la secuencia de reglas que nos permite obtener una cadena del lenguaje generado por la gramática a partir de su símbolo inicial, se la conoce con el nombre de *derivación de la cadena*. Por ejemplo, para la gramática del ejercicio 4.1:

$$E \longrightarrow E + E \mid E * E \mid (E) \mid a,$$

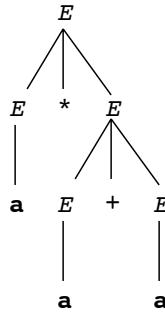
la secuencia

$$E \longrightarrow E * E \longrightarrow E * E + E \longrightarrow E * E + a \longrightarrow E * a + a \longrightarrow a * a + a$$

constituye una derivación de la cadena **a * a + a**. La cadena de V^* obtenida en cada paso de la derivación recibe el nombre de *forma sentencial*.

Un *árbol de derivación* es una representación gráfica de la secuencia de reglas aplicada (es fácil comprobar que las gramáticas tipo 0 no admiten esta representación). Por ejemplo, la derivación anterior se podría representar mediante el árbol de la fig. 4.4. Más formalmente, un árbol de derivación se define como un *árbol ordenado*² que cumple lo siguiente:

²Se dice que una árbol está *ordenado* si se ha definido una relación de orden entre los hijos de cada nodo. Se suele entonces tomar como criterio de representación que los nodos dibujados a la izquierda preceden a los que están más a la derecha.

Figura 4.4: Árbol de derivación de la cadena $a * a + a$.

- cada nodo interior (con descendencia) está etiquetado con un símbolo de V_N ;
- el símbolo inicial S de la gramática ocupa el nodo raíz;
- los nodos terminales (sin descendencia) u *hojas* sólo contienen símbolos de V_T o, tal vez, la cadena vacía ϵ ;
- si un nodo contiene el símbolo $A \in V_N$ y presenta como descendientes, en este orden, a los nodos $X_1, X_2 \dots X_n$, entonces existe una regla en la gramática G en la que la variable A es sustituida por $X_1, X_2 \dots X_n$.

A la hora de reconstruir la derivación a partir del árbol, se nos presentan varias alternativas según el orden en el que efectuemos las sustituciones de las diversas variables; al objeto de eliminar esta ambigüedad es habitual utilizar dos de ellas, llamadas *derivación por la izquierda* y *derivación por la derecha*. La diferencia entre ambas reside en decidir qué variable es sustituida en cada paso de la derivación, la que está más a la izquierda o la que está más a la derecha en la forma sentencial. Para el árbol anterior estas derivaciones serían:

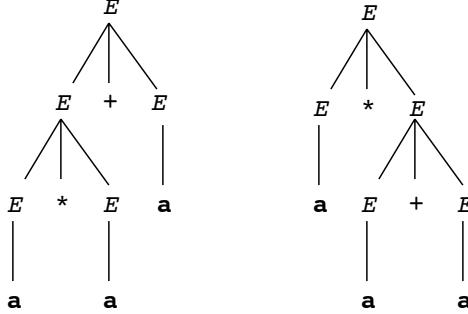
1. Por la izquierda: $E \rightarrow E * E \rightarrow a * E \rightarrow a * E + E \rightarrow a * a + E \rightarrow a * a + a$
2. Por la derecha: $E \rightarrow E * E \rightarrow E * E + E \rightarrow E * E + a \rightarrow E * a + a \rightarrow a * a + a$.

Tomando uno de estos criterios como fijo, la correspondencia entre los árboles de derivación y las derivaciones es biunívoca.

4.4.2 Ambigüedad

Siguiendo con el ejemplo de esta sección, observemos que la cadena $a * a + a$ puede ser generada por dos árboles de derivación (ver fig. 4.5).

Diremos que una gramática G es *ambigua* si hay al menos una cadena en $L(G)$ que tenga dos o más árboles de derivación distintos (o, equivalentemente, dos o más derivaciones por la izquierda distintas).

Figura 4.5: Dos árboles de derivación de la cadena **a * a + a**.

Se podría pensar que esta ambigüedad es consecuencia de cómo se ha elegido la gramática, y que siempre es posible encontrar una gramática equivalente a la dada que no sea ambigua (y en el caso anterior eso es sencillo de conseguir); pero esta idea es incorrecta, ya que hay lenguajes *intrínsecamente ambiguos*, para los cuales se ha demostrado que todas las gramáticas que los generan son ambiguas. La propiedad de ambigüedad es indecidible: no hay ningún algoritmo que nos dé la respuesta para cualquier gramática en un tiempo finito. Aunque existen algoritmos que para determinar si una cierta cadena $\omega \in L(G)$ admite dos o más árboles de derivación diferentes, para demostrar que G no es ambigua por este método, tendríamos que probar todas las cadenas de V_T^* . Como este conjunto es infinito, el procedimiento requeriría un tiempo potencialmente infinito.

Ejemplo : Un ejemplo estándar de este problema en los lenguajes de programación, es el fenómeno del “else ambiguo”. Consideremos las producciones

$$\begin{aligned}
 <\text{instr}> &\longrightarrow \text{if}(<\text{expr}>) <\text{instr}> \\
 &\quad | \quad \text{if}(<\text{expr}>) <\text{instr}> \text{else } <\text{instr}> \\
 &\quad | \quad <\text{ostrainstr}>
 \end{aligned}$$

que describen la sentencia *if-else* del lenguaje C. Y consideremos ahora la cadena

```
if (expr1) if (expr2) instr1; else instr2;
```

Esta cadena puede ser derivada por los dos caminos que se muestran en la figura 4.6; en (a) el **else** va emparejado con el primer **if**; en (b), va con el segundo. El compilador de C debe interpretar la sentencia por el segundo camino, pero la regla sintáctica no es suficiente para indicarlo y, por tanto, es necesaria información adicional.

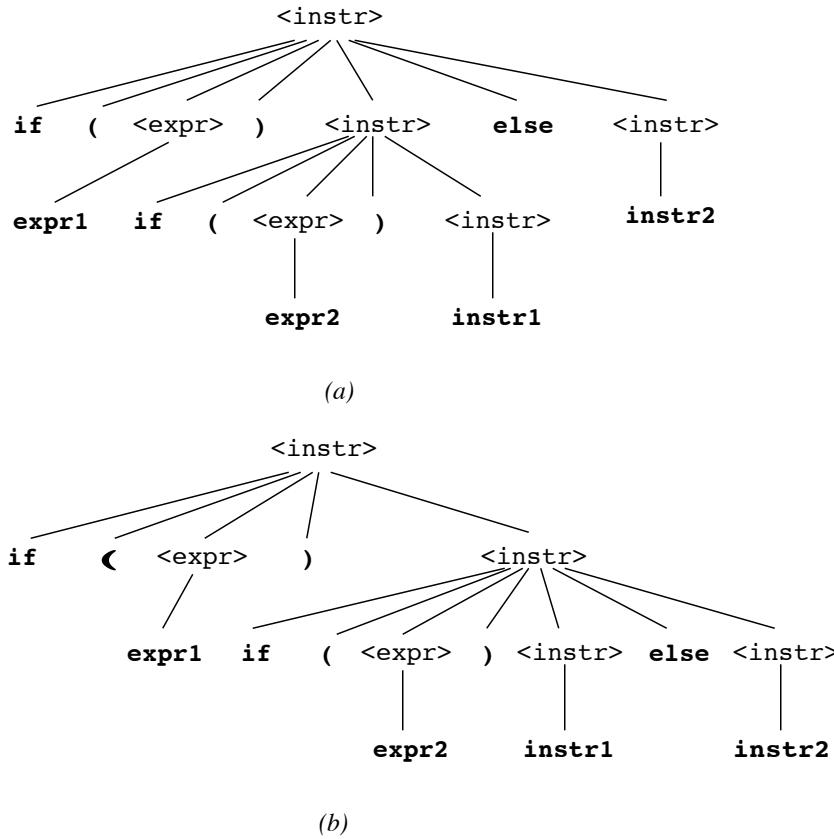


Figura 4.6: Dos posibles interpretaciones para el “else ambiguo”.

Generalmente, en las gramáticas de los lenguajes de programación se elimina la ambigüedad atendiendo a la *precedencia* deseada para los operadores. Una gramática equivalente a la del ejercicio 4.1 que no es ambigua y conserva la mayor precedencia esperada de la multiplicación respecto a la suma, es:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow a \mid (E) \end{aligned}$$

Ejercicio

4.5 Las gramáticas regulares son un subconjunto de las gramáticas independientes del contexto. ¿Puede ser intrínsecamente ambiguo un lenguaje regular? Justifíquese la respuesta.

4.4.3 Recursividad

Si nos fijamos detenidamente en la gramática anterior, observaremos que podemos tener derivaciones de la forma $E \xrightarrow{+} \alpha E \beta$ (con $|\alpha|, |\beta| \geq 0$), donde el símbolo $\xrightarrow{+}$ debe leerse “produce en uno o más pasos”, con lo que se pueden generar cadenas pertenecientes al lenguaje de longitud arbitraria y, por tanto, podemos afirmar que el lenguaje generado por la gramática es infinito. A las gramáticas que poseen esta propiedad se las llama *recursivas*. Si $\alpha = \epsilon$, se dice que la gramática es *recursiva por la izquierda*; mientras que si $\beta = \epsilon$ se dice que la gramática es *recursiva por la derecha*.

La recursividad de las gramáticas de los lenguajes de programación está estrechamente ligada a la asociatividad de los operadores. Para entender esto, consideremos la cadena $a * a * a$. Notemos que existen dos posibles modos de evaluar esta expresión: $(a * a) * a$ y $a * (a * a)$; en el primer caso, decimos que ** asocia a la izquierda*; y en el segundo, que *asocia a la derecha*. Veamos los árboles de derivación de esta cadena para dos gramáticas, una recursiva por la izquierda y la otra por la derecha (fig. 4.7):

$$(a) \quad T \xrightarrow{} T * F \mid F \quad F \xrightarrow{} a \quad (b) \quad T \xrightarrow{} F * T \mid F \quad F \xrightarrow{} a$$

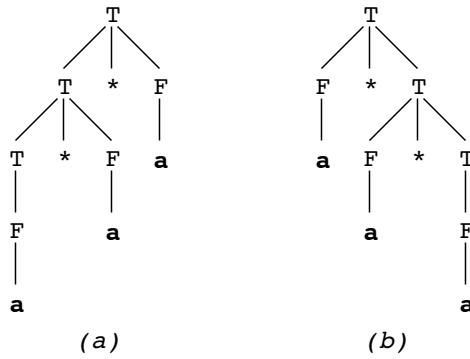


Figura 4.7: Árboles de derivación de $a * a * a$ para dos gramáticas equivalentes. En (a), el operador $*$ asocia a la izquierda; en (b) lo hace a la derecha.

Si deseamos que un operador asocie a la izquierda, deberemos escribir una gramática recursiva por la izquierda. Si deseamos que asocie a la derecha, deberemos escribirla recursiva por la derecha. Observemos que el árbol (a), correspondiente a la gramática recursiva por la izquierda, desciende hacia a la izquierda; mientras que el (b), lo hace hacia la derecha.

Por otra parte, la recursividad por la izquierda impide utilizar algunos algoritmos de análisis sintáctico (los llamados descendentes). Se plantea entonces el problema de encontrar una gramática equivalente a la dada que no

presente este tipo de recursión. La eliminación de la recursión por la izquierda es también necesaria para transformar una gramática en otra equivalente que tenga la llamada *forma normalizada de Greibach* (ver más abajo).

Eliminación de la recursión por la izquierda

Si tenemos una gramática con reglas del tipo

$$A \rightarrow A\alpha \mid \beta,$$

que presenta recursión por la izquierda inmediata, las producciones que puede generar son del tipo

$$A \rightarrow A\alpha \rightarrow A\alpha\alpha \rightarrow \dots \rightarrow A\alpha\alpha\dots\alpha \rightarrow \beta\alpha\alpha\dots\alpha.$$

Observemos que estas producciones también pueden ser generadas mediante las reglas

$$\begin{aligned} A &\rightarrow \beta \mid \beta B \\ B &\rightarrow \alpha \mid \alpha B, \end{aligned}$$

que presentan recursión por la derecha.

Este esquema se puede generalizar fácilmente en el caso de tener reglas de la forma

$$\begin{aligned} A &\rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \\ A &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_m, \end{aligned}$$

sustituyéndolas por las reglas

$$\begin{aligned} A &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_m \mid \beta_1 B \mid \beta_2 B \mid \dots \mid \beta_m B \\ B &\rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n \mid \alpha_1 B \mid \alpha_2 B \mid \dots \mid \alpha_n B \end{aligned}$$

Ejemplo: Volvamos a considerar la gramática de expresiones aritméticas

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow \mathbf{a} \mid (E) \end{aligned}$$

Una gramática equivalente que no presenta recursión por la izquierda es

$$\begin{aligned} E &\rightarrow T \mid TE' \\ E' &\rightarrow +T \mid +TE' \\ T &\rightarrow F \mid FT' \\ T' &\rightarrow *F \mid *FT' \\ F &\rightarrow \mathbf{a} \mid (E) \end{aligned}$$

Un problema añadido lo presentan las gramáticas que contienen recursión por la izquierda no inmediata. Una gramática de este tipo es

$$\begin{aligned} A &\rightarrow B0 \mid 1 \\ B &\rightarrow A1 \mid 0 \mid \epsilon, \end{aligned}$$

que presenta recursión por la izquierda porque son posibles derivaciones del tipo

$$A \longrightarrow B_0 \longrightarrow A_{10}.$$

Para eliminar sistemáticamente la posibilidad de tales derivaciones podemos utilizar el algoritmo siguiente, que siempre funciona si la gramática no tiene ciclos (producciones de la forma $A \xrightarrow{+} A$) o producciones vacías (de la forma $A \longrightarrow \epsilon$). Este tipo de producciones se pueden eliminar fácilmente (ver más abajo).

Algoritmo 4.1 *Eliminación de la recursión por la izquierda*

Entrada: Una gramática $G = (V_N, V_T, P, S)$ sin ciclos ni producciones vacías.

Salida: Una gramática equivalente sin recursión por la izquierda.

Método:

1. Numérense las variables de la gramática arbitrariamente $V_N = \{A_1, A_2 \dots A_r\}$. (El algoritmo conseguirá que todas las producciones de la nueva gramática comiencen por terminal o por una variable con subíndice superior al de la variable de la parte izquierda).
2. Tomar $i = 1$.
3. Considerar todas las producciones que comienzan por A_i :

$$A_i \longrightarrow A_i\alpha_1 \mid \dots \mid A_i\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

(dada la forma en que procede este algoritmo, siempre se cumple que ningún β_j comienza por A_k si $k \leq i$). Sustituir todas estas producciones por

$$\begin{aligned} A_i &\longrightarrow \beta_1 \mid \dots \mid \beta_m \mid \beta_1B_i \mid \dots \mid \beta_mB_i \\ B_i &\longrightarrow \alpha_1 \mid \dots \mid \alpha_n \mid \alpha_1B_i \mid \dots \mid \alpha_nB_i, \end{aligned}$$

donde B_i es una nueva variable. Eso asegura que todas las producciones cuya parte izquierda es A_i empiezan por terminal o por una variable A_k tal que $k > i$.

4. Si $i = n$, devolver la gramática resultante y parar el algoritmo.
5. Tomar $i = i + 1$ y $j = 1$.
6. Sustituir todas las reglas de la forma $A_i \longrightarrow A_j\alpha$ por las reglas $A_i \longrightarrow \beta_1\alpha \mid \dots \mid \beta_m\alpha$ donde $A_j \longrightarrow \beta_1 \mid \dots \mid \beta_m$ son todas las reglas de derivación que tienen como parte izquierda A_j . Como ya se habrá conseguido que las partes derechas de las reglas que tienen A_j como parte izquierda comiencen por un terminal o con A_k con $k \geq j$, este paso garantiza que todas las reglas con parte izquierda A_i también cumplen la propiedad.

7. Si $j = i - 1$, saltar al paso 3. Si no, tomar $j = j + 1$ y saltar al paso 6.

El algoritmo anterior, además de eliminar la recursión por la izquierda, conduce a una gramática donde todas las reglas satisfacen que si $A_i \rightarrow A_j\alpha \in P$, entonces $i < j$.

Ejercicio

4.6 Elimina la recursión por la izquierda en las gramáticas:

- $S \rightarrow SAB \mid AB \mid 0 \mid 1; A \rightarrow SB \mid 1; B \rightarrow AB \mid 0.$
- $S \rightarrow AB \mid c; A \rightarrow BS \mid a; B \rightarrow SA \mid b.$

4.4.4 El análisis sintáctico

Como ya hemos mencionado varias veces, la mayor parte de los lenguajes de programación son LIC. Un compilador debe reconocer si un determinado programa (cadena de caracteres en definitiva) se adapta a las especificaciones del lenguaje. Para ello, resulta imprescindible que dispongamos de algoritmos que nos permitan decidir la corrección o incorrección del programa antes de traducirlo a instrucciones de máquina. Para decidir si una cadena dada pertenece al lenguaje generado por una GIC particular, debemos ser capaces de reconstruir el árbol de derivación de la cadena. A este proceso se le llama *análisis sintáctico*, y su dominio es, por otra parte, el principal objetivo que debemos plantearnos en el estudio de las GIC. Los métodos de análisis sintáctico son múltiples, y su estudio en profundidad se realiza en la asignatura de *Diseño de compiladores*. En este curso nos centraremos en un algoritmo de uso general, válido tanto para gramáticas ambiguas como no ambiguas, llamado *Algoritmo de Cocke, Younger y Kasami* (CYK). Introduciremos este método de análisis sintáctico de manera constructiva; para ello vamos a partir de un ejemplo en el que todas las reglas de la gramática tienen una forma especial (veremos más adelante que esta gramática está en *forma normalizada de Chomsky*).

Ejemplo: Consideremos la gramática G siguiente,

$$\begin{aligned} S &\rightarrow EB \mid FA \\ A &\rightarrow 0 \mid ES \mid FC \\ B &\rightarrow 1 \mid FS \mid ED \\ C &\rightarrow AA \\ D &\rightarrow BB \\ E &\rightarrow 0 \\ F &\rightarrow 1 \end{aligned}$$

y tratemos de reconstruir el árbol o árboles de derivación de la cadena $\omega = 001101$.

La gramática del ejemplo tiene dos tipos de reglas; uno produce sólo un terminal, el otro produce cadenas de dos variables. Las cadenas de terminales de longitud mayor que 1 se generan duplicando las variables hasta alcanzar la longitud deseada. Para reconstruir, si es posible, la derivación de ω , deberemos partirla en dos trozos, cada uno derivado desde una variable. Las posibles formas de trocear ω son

0	01101
00	1101
001	101
0011	01
00110	1

Con esto conseguimos descomponer el problema de analizar una cadena de longitud 6 en 5 problemas de longitud menor que 6. Siguiendo con este proceso, podemos descomponer las subcadenas de longitud 5 en subcadenas de longitud menor que 5, y así sucesivamente, hasta conseguir que todas las subcadenas obtenidas tengan longitud 1.

Construyamos una función

$$f : \Sigma^* \longrightarrow 2^P$$

que, para cada subcadena de ω devuelva el conjunto de reglas que le permitan derivarla desde cualquier variable. Observando la gramática, obtenemos para las subcadenas de longitud 1

$$f(0) = \{A \rightarrow 0, E \rightarrow 0\} \quad f(1) = \{B \rightarrow 1, F \rightarrow 1\}$$

La subcadena 00 sólo puede ser generada partiendo de una regla que produzca alguna de las cadenas contenidas en el conjunto $\{AA, AE, EA, EE\}$. La primera variable de estas cadenas se obtiene de la parte izquierda de las reglas que producen el primer símbolo; el segundo, de las reglas que producen el segundo (en este caso particular ambos símbolos coinciden). Observando las reglas de la gramática, la única variable que produce alguna de las anteriores cadenas es C , es decir, $f(00) = \{C \rightarrow AA\}$. Procediendo análogamente con la subcadena 01, se obtiene $f(01) = \{S \rightarrow EB\}$. Puesto que este conjunto contiene una regla cuya parte izquierda es el símbolo inicial de la gramática, podemos afirmar que la subcadena 01 pertenece al lenguaje generado por G .

La subcadena 001 puede ser troceada de dos formas: 0-01 y 00-1. Esta subcadena podrá ser generada sólo mediante la regla $A \rightarrow ES$, puesto que es la única regla de G cuya parte derecha está formada por la concatenación de las variables que están en la parte izquierda de las reglas que producen 0

con las variables de la parte izquierda de las reglas que producen 01; o bien, de las que producen 00 con las que producen 1. Con ello,

$$f(001) = \{A \rightarrow ES\}.$$

Análogamente, la subcadena 101 se puede descomponer en 1-01 o en 10-1. En ambos casos nos aparece una subcadena de longitud 2 que no habíamos analizado previamente, por lo que es necesario calcular $f(10)$ y $f(01)$. De hecho, necesitamos encontrar el valor de la función f para todas las subcadenas de longitud menor o igual que 5 contenidas en ω .

	1 (0)	2 (0)	3 (1)	4 (1)	5 (0)	6 (1)
1	$A \rightarrow 0$ $E \rightarrow 0$	$A \rightarrow 0$ $E \rightarrow 0$	$B \rightarrow 1$ $F \rightarrow 1$	$B \rightarrow 1$ $F \rightarrow 1$	$A \rightarrow 0$ $E \rightarrow 0$	$B \rightarrow 1$ $F \rightarrow 1$
2	$C \rightarrow AA$	$S \rightarrow EB$	$D \rightarrow BB$	$S \rightarrow FA$	$S \rightarrow EB$	
3	$A \rightarrow ES$	$B \rightarrow ED$	$B \rightarrow FS$	$B \rightarrow FS$		
4	$S \rightarrow EB$	$S \rightarrow EB$	$D \rightarrow BB (2)$			
5	$A \rightarrow ES$	$B \rightarrow ED$				
6	$S \rightarrow EB$					

Figura 4.8: Tabla de análisis sintáctico CYK para la cadena $\omega = 001101$.

Para evitar llamadas recursivas a la función f , conviene almacenar los resultados previos. Organizaremos esta información en una tabla como la de la figura 4.8, en la que cada celda contendrá la regla o reglas de la gramática que se ha de aplicar en primer lugar para generar la subcadena que comienza en su número de columna y tiene una longitud igual a su número de fila. Por ejemplo, la celda (1,1) debe contener las reglas $A \rightarrow 0$ y $E \rightarrow 0$; y la celda (1,3), la regla $A \rightarrow ES$. Esta tabla puede irse llenando por filas, ya que el valor de la función f para una subcadena de longitud n sólo depende de cómo se han generado las subcadenas de longitud $n - 1$. La celda situada en la última fila representará a toda la cadena ω , por lo que si contiene una regla cuya parte izquierda es el símbolo inicial S de la gramática —como es el caso en nuestro ejemplo—, podemos afirmar que $\omega \in L(G)$; si no es así, entonces $\omega \notin L(G)$. En la figura 4.9 se observa el esquema de dependencias de la celda (4,2); para llenar una celda debemos tener en cuenta las reglas cuya parte derecha esté formada por las partes izquierdas de las celdas unidas por una flecha. Por ejemplo, dicha celda, que corresponde a la subcadena 0110, debe obtenerse mirando las partes izquierdas de las reglas contenidas en los

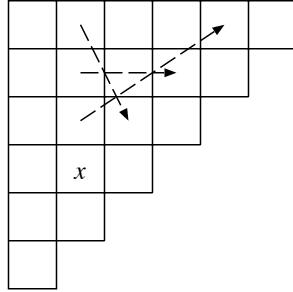


Figura 4.9: Esquema de dependencias en la tabla de análisis sintáctico mediante el algoritmo CYK.

pares de celdas $[(1, 2), (3, 3)]$, $[(2, 2), (2, 4)]$ y $[(3, 2), (1, 5)]$. Para construir el árbol o árboles de derivación de la cadena analizada, leeremos la tabla de abajo hacia arriba. El (2) que aparece en la celda $(4, 3)$ indica que la regla $D \rightarrow BB$ se puede obtener de dos maneras: $[(1, 3), (3, 4)]$ y $[(3, 3), (1, 6)]$ (ver figura 4.10), lo que permite estudiar la ambigüedad de la gramática, ya que podemos obtener todos los árboles de derivación de la cadena analizada. Pasemos ahora a enunciar formalmente el algoritmo de Cocke, Younger y Kasami.

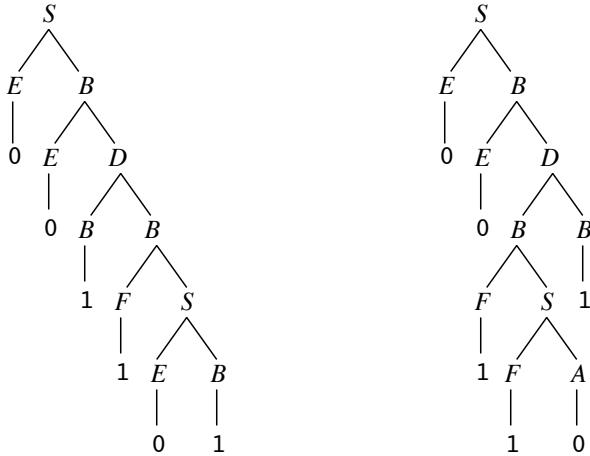


Figura 4.10: Dos árboles de derivación para la cadena $\omega = 001101$.

Sea la cadena $\omega = a_1a_2\dots a_l$. Podemos extraer $l(l+1)/2$ subcadenas (l de longitud 1, $l-1$ de longitud 2, etc.), que denotaremos x_i^j , donde el subíndice indica la posición del primer símbolo de la subcadena; y el superíndice, su longitud. Por tanto, $|x_i^j| = j$ y $x_i^0 = \epsilon$. Además $x = x_1^l = x_1^1x_2^1\dots x_l^1$. Llamaremos P_i^j al subconjunto de reglas de P , tales que $r \equiv A \xrightarrow{+} \alpha \in P_i^j$ si r es la primera regla en una derivación $A \xrightarrow{+} x_i^j$. Obviamente, si $i + j > l + 1$, se tendrá $P_i^j = \emptyset$. Estos conjuntos se construyen recursivamente, y el

problema se reduce a comprobar si alguna regla de la forma $S \rightarrow \alpha \in P_1^l$. El algoritmo es:

Algoritmo 4.2 Algoritmo de Cocke, Younger y Kasami

Entrada: Una gramática $G = (V_N, V_T, P, S)$ en forma de Chomsky, sin producciones vacías, y una cadena de entrada $x = a_1a_2\dots a_l$ de V_T^+ .

Salida: La tabla de análisis sintáctico $\mathcal{V}(G, x)$ tal que cada elemento P_i^j contiene la regla $r \equiv A \rightarrow \alpha$ sí y sólo sí r es la primera regla en una derivación $A \xrightarrow{+} x_i^j$, donde $x_i^j = a_i a_{i+1} \dots a_{i+j-1}$.

Método:

1. ($j = 1$) Construir los conjuntos P_i^1 para $i = 1, 2, \dots, l$. Si después de este paso $r \equiv A \rightarrow \alpha \in P_i^1$, entonces $A \xrightarrow{+} a_i$.
2. ($j > 1$) Si se conocen $P_{i'}^{j'}$ para todos los $j' < j$ ($i' \leq l - j' + 1$), construir

$$P_i^j = \{A \rightarrow BC \in P : B \rightarrow \alpha \in P_i^k \wedge C \rightarrow \beta \in P_{i+k}^{j-k} \text{ para algún } k\},$$

donde k y $j - k$ son ambos menores que j , y por tanto, P_i^j y P_{i+k}^{j-k} han sido calculados previamente. Este paso asegura que si $A \rightarrow BC \in P_i^j$, entonces

$$A \rightarrow BC \xrightarrow{+} a_i \dots a_{i+k-1} C \xrightarrow{+} a_i \dots a_{i+j-1}.$$

3. El paso anterior se repite para obtener todos los conjuntos P_i^j (variando j desde 1 hasta $l - j + 1$).

Ejercicio

4.7 Por medio del algoritmo de Cocke, Younger y Kasami, determina si la cadena $\omega = abaa$ pertenece al lenguaje generado por la gramática

$$\begin{aligned} S &\rightarrow AB \mid AA \\ A &\rightarrow SB \mid a \\ B &\rightarrow AB \mid b \end{aligned}$$

4.4.5 Simplificación de una GIC

A diferencia del caso de los AFD y los lenguajes regulares, no disponemos de un método para construir la gramática equivalente más sencilla que genere un lenguaje independiente del contexto dado. A pesar de eso, sí podemos encontrar gramáticas equivalentes a una dada que no contengan ciertos tipos de producciones que dificulten el trabajo con ella y que nos permitan estandarizar las producciones de la gramática de modo que todas tengan una cierta “forma normalizada”. Hay tres tipos de reglas no deseadas:

1. Derivaciones vacías (de la forma $A \rightarrow \epsilon$).
2. Derivaciones unitarias (de la forma $A \rightarrow B$).
3. Símbolos inútiles (símbolos no utilizados en la derivación de ninguna cadena de $L(G)$).

La gramática que resulta de estas operaciones de eliminación la denominaremos *gramática simplificada*. Este nombre puede ser equívoco, ya que en algunos casos la versión no simplificada puede resultar más fácil de entender para el lector.

Eliminación de las producciones vacías

Diremos que una regla de derivación es *vacía* si es del tipo $A \rightarrow \epsilon$. En general, si A produce ϵ en un número arbitrario de pasos ($A \xrightarrow{*} \epsilon$), se dice que la variable A es *anulable*. Si $\epsilon \in L(G)$ no es posible eliminar todas las reglas de derivación vacías, ya que es preciso conservar la posibilidad de generar ϵ . Salvo por esta excepción, es posible encontrar una gramática equivalente a la dada que no contenga ninguna producción vacía. El objetivo es que en la derivación de cualquier cadena $x \neq \epsilon$ no se aplique ninguna regla de producción vacía.

Comencemos con un ejemplo. Consideremos la gramática G

$$\begin{aligned} S &\rightarrow ABA \\ A &\rightarrow aA \mid \epsilon \\ B &\rightarrow bB \mid \epsilon, \end{aligned}$$

en la que tenemos dos reglas de derivación vacías. Observemos que desde la variable A y B podemos generar cadenas de a^* y b^* respectivamente. El objetivo es encontrar una gramática equivalente a G que no contenga las reglas $A \rightarrow \epsilon$ y $B \rightarrow \epsilon$. Para ello comenzamos buscando el conjunto de variables anulables. Evidentemente A y B pertenecen a este conjunto. Además, ya que la regla $S \rightarrow ABA$ produce una cadena formada sólo por variables anulables, también se tiene $S \xrightarrow{*} \epsilon$, con lo que S también es anulable (obviamente $\epsilon \in L(g)$). Podemos construir las nuevas producciones

haciendo explícita la posibilidad de que estas variables produzcan ϵ , es decir, sustituiremos las reglas $A \rightarrow aA \mid \epsilon$ por las reglas

$$A \rightarrow aA \mid a$$

y las reglas $B \rightarrow bB \mid \epsilon$ por las reglas

$$B \rightarrow bB \mid b.$$

En la regla $S \rightarrow ABA$ las variables de la parte derecha se pueden anular conjuntamente o por separado, de modo que las posibilidades son

$$S \rightarrow ABA \mid AB \mid AA \mid BA \mid A \mid B \mid \epsilon.$$

Puesto que S es anulable, es necesario mantener esta última regla, aunque no sería válido en el caso de que S apareciera en alguna parte derecha. Por ello, si $\epsilon \in L(G)$, resulta más general sustituir las reglas para S por el conjunto de reglas siguiente

$$\begin{aligned} S' &\rightarrow S \mid \epsilon \\ S &\rightarrow ABA \mid AB \mid AA \mid BA \mid A \mid B, \end{aligned}$$

tomando S' como nuevo símbolo inicial de la gramática.

Ahora estamos en condiciones de dar un algoritmo formal que elimina sistemáticamente las producciones vacías. Previamente enunciaremos un algoritmo que nos permite encontrar las variables anulables de la gramática.

Algoritmo 4.3 Busqueda de variables anulables

Entrada: Una gramática $G = (V_N, V_T, P, S)$.

Salida: El conjunto V_ϵ de variables anulables de G .

Método:

1. Tomar $i = 0$ y $V_0 = \{A \in V_N : A \rightarrow \epsilon \in P\}$.
2. Hacer

$$\begin{aligned} i &= i + 1 \\ V_i &= V_{i-1} \cup \{A : A \rightarrow \alpha \in P \text{ para algún } \alpha \in V_{i-1}^*\} \end{aligned}$$

mientras $V_i \neq V_{i-1}$.

3. Tomar $V_\epsilon = V_i$.

Algoritmo 4.4 Eliminación de las derivaciones vacías

Entrada: Una gramática $G = (V_N, V_T, P, S)$.

Salida: Una gramática equivalente $G' = (V'_N, V_T, P', S')$ sin producciones vacías.

Método:

1. Tomar $P' = P$.
2. Encontrar V_ϵ aplicando el algoritmo anterior.
3. Para cada producción $A \rightarrow \alpha \in P$, añadir a P' todas las producciones que puedan ser obtenidas de ella borrando de α una o más de las variables anulables que contenga.
4. Eliminar de P' todas las producciones vacías, duplicadas y de la forma $A \rightarrow A$ que contenga.
5. Si $S \in V_\epsilon$, tomar $V'_N = V_N \cup \{S'\}$ y añadir las reglas $S' \rightarrow S \mid \epsilon$ a P' ; en otro caso, tomar $V'_N = V_N$ y $S' = S$.

Ejercicio

4.8 Elimínense las producciones vacías de la gramática:

$$\begin{aligned} E &\longrightarrow TR \\ R &\longrightarrow +TR \\ R &\longrightarrow \epsilon \\ T &\longrightarrow FA \\ A &\longrightarrow *FA \\ A &\longrightarrow \epsilon \\ F &\longrightarrow (E) \\ F &\longrightarrow a \end{aligned}$$

Eliminación de las producciones unitarias

Una producción es *unitaria* si su estructura es del tipo $A \rightarrow B$. Es siempre posible encontrar una gramática equivalente a una dada que no contenga derivaciones de este tipo. Ilustraremos el método que hay que seguir en el proceso de eliminación de esta clase de reglas con la siguiente gramática de expresiones aritméticas:

$$\begin{aligned} E &\longrightarrow E + T \mid T \\ T &\longrightarrow T * F \mid F \\ F &\longrightarrow \mathbf{a} \mid (E), \end{aligned}$$

en la que aparecen las reglas de derivación unitarias $E \rightarrow T$ y $T \rightarrow F$.

Comencemos con la variable E . Un primer paso para eliminar la producción unitaria de esta variable puede consistir en sustituirla por todas las producciones que se pueden obtener a partir de las reglas de T , con lo que las reglas para E quedarían,

$$E \longrightarrow E + T \mid T * F \mid F.$$

Observamos que no hemos conseguido eliminar totalmente las producciones unitarias de E , por lo que repetimos el proceso con F

$$E \longrightarrow E + T \mid T * F \mid \mathbf{a} \mid (E).$$

Realizando el proceso análogo con la variable T obtenemos, finalmente, una gramática equivalente que no presenta derivaciones unitarias

$$\begin{aligned} E &\longrightarrow E + T \mid T * F \mid \mathbf{a} \mid (E) \\ T &\longrightarrow T * F \mid \mathbf{a} \mid (E) \\ F &\longrightarrow \mathbf{a} \mid (E). \end{aligned}$$

Este proceso se puede realizar sistemáticamente utilizando el siguiente algoritmo.

Algoritmo 4.5 Eliminación de las derivaciones unitarias

Entrada: Una gramática $G = (V_N, V_T, P, S)$ sin derivaciones vacías.

Salida: Una gramática equivalente $G' = (V_N, V_T, P', S)$ sin producciones unitarias ni vacías.

Método:

1. Construir, para cada variable $A \in V_N$, el conjunto $V_A = \{B : A \xrightarrow{*} B\}$ de las variables accesibles desde A por derivaciones unitarias como sigue:
 - (a) Tomar $V_1 = \{A\}$, $i = 1$.
 - (b) Hacer

$$\begin{aligned} i &= i + 1 \\ V_i &= V_{i-1} \cup \{C : B \longrightarrow C \in P \wedge B \in V_{i-1}\} \end{aligned}$$

mientras $V_i \neq V_{i-1}$.

- (c) Tomar $V_A = V_i$.

2. Construir el conjunto de producciones P' : si $B \longrightarrow \alpha \in P$ y no es una producción unitaria, añadir la regla $A \longrightarrow \alpha$ a P' para toda A que tal que $B \in V_A$.

Eliminación de los símbolos inútiles

Dada una GIC $G = (V_N, V_T, P, S)$, diremos que un símbolo $X \in V$ es útil si se utiliza en el proceso de generación de alguna cadena de $L(G)$, esto es, si existe alguna derivación $S \xrightarrow{*} \alpha X \beta \xrightarrow{*} \omega$ con $\alpha, \beta \in V^*$ y $\omega \in V_T^*$. En otro caso diremos que X es inútil. Observemos que para que un símbolo sea útil deben ocurrir dos cosas: debe ser accesible desde el símbolo inicial y desde él se debe poder generar alguna cadena de terminales. Veamos con un ejemplo cómo se pueden eliminar los símbolos que no cumplen estas dos condiciones. Consideremos la gramática

$$\begin{aligned} S &\longrightarrow aAB \mid ASB \mid cCc \\ A &\longrightarrow aAa \mid aa \\ B &\longrightarrow bBb \mid bb \\ C &\longrightarrow cCc \\ D &\longrightarrow ASB \\ E &\longrightarrow ABA \end{aligned}$$

y busquemos en primer lugar los símbolos que no son capaces de generar cadenas de terminales. Para ello construiremos el conjunto V_{NE} de símbolos que sí lo hacen de manera recursiva. Sea V_i el conjunto de símbolos que son capaces de generar cadenas de sólo terminales en i o menos pasos. En cero pasos los símbolos que pueden generar cadenas de sólo terminales son los propios terminales, es decir, $V_0 = V_T = \{a, b, c\}$. Para construir V_1 observemos que las únicas variables que generan en un paso cadenas de sólo terminales, o, lo que es lo mismo, que son parte izquierda de alguna regla cuya parte derecha es una cadena de V_0^* , son A y B , con lo que $V_1 = \{a, b, c, A, B\}$. Los símbolos en V_2 serán aquellos que son parte izquierda de alguna regla cuya parte derecha es una cadena de V_1^* , $V_2 = \{a, b, c, A, B, S, E\}$. Y continuando con este proceso obtenemos finalmente que $V_{NE} = V_3 = \{a, b, c, A, B, S, E, D\}$. El resto de símbolos de la gramática, $V - V_{NE}$, no son capaces de producir cadenas de sólo terminales y, por tanto, pueden ser eliminados sin que perdamos ninguna cadena de $L(G)$. La gramática quedará como

$$\begin{aligned} S &\longrightarrow aAB \mid ASB \\ A &\longrightarrow aAa \mid aa \\ B &\longrightarrow bBb \mid bb \\ D &\longrightarrow ASB \\ E &\longrightarrow ABA \end{aligned}$$

Ahora construiremos el conjunto V_A de símbolos accesibles desde el símbolo inicial. De nuevo procederemos recursivamente. Sea V_i el conjunto de símbolos accesibles desde S en i o menos pasos. Evidentemente, $V_0 = \{S\}$ y $V_1 = \{S, a, A, B\}$. Para construir V_2 tendremos que añadir a V_1 los símbolos accesibles desde A y desde B , con lo que $V_2 = \{S, a, A, B, b\}$. Puesto que

ya no hay nuevos símbolos accesibles podemos concluir que $V_A = V_2$. Las variables D , E y c (de hecho este símbolo ya no aparecía en ninguna regla) no son accesibles desde S y pueden ser eliminadas. La gramática resultante queda finalmente

$$\begin{aligned} S &\longrightarrow aAB \mid ASB \\ A &\longrightarrow aAa \mid aa \\ B &\longrightarrow bBb \mid bb \end{aligned}$$

El siguiente algoritmo permite realizar este proceso sistemáticamente para una gramática arbitraria.

Algoritmo 4.6 Eliminación de símbolos inútiles

Entrada: Una gramática $G = (V_N, V_T, P, S)$.

Salida: Una gramática $G' = (V'_N, V'_T, P', S)$ equivalente a G sin símbolos inútiles.

Método:

1. Construir el conjunto V_{NE} de símbolos de V que generan cadenas de V_T^* :

- (a) Tomar $i = 0$ y $V_0 = V_T$.
- (b) Hacer

$$\begin{aligned} i &= i + 1 \\ V_i &= V_{i-1} \cup \{A : A \longrightarrow \alpha \in P \wedge \alpha \in V_{i-1}^*\} \end{aligned}$$

mientras $V_i \neq V_{i-1}$.

- (c) Tomar $V_{NE} = V_i$.

2. Construir la gramática G'' eliminando de G todos los símbolos, junto con sus reglas, que no aparezcan en V_{NE} .

3. Construir el conjunto V_A de símbolos de G'' accesibles desde S :

- (a) Tomar $i = 0$ y $V_0 = \{S\}$.
- (b) Hacer

$$\begin{aligned} i &= i + 1 \\ V_i &= V_{i-1} \cup \{X : A \longrightarrow \alpha X \beta \in P \wedge \alpha, \beta \in V_{i-1}^*\} \end{aligned}$$

mientras $V_i \neq V_{i-1}$.

- (c) Tomar $V_A = V_i$

4. Construir la gramática final G' sin símbolos inútiles eliminando de G'' todos los símbolos, junto con sus reglas, que no estén en V_A .

Conviene recordar que el algoritmo de eliminación de producciones unitarias puede generar, en algunos casos, símbolos inútiles. Por tanto, para garantizar que la gramática queda simplificada al final del proceso, el orden de aplicaciones de las transformaciones debería de ser el seguido en esta sección:

1. Eliminar las derivaciones vacías.
2. Eliminar las producciones unitarias.
3. Eliminar los símbolos inútiles.

No obstante, una aplicación preliminar del último algoritmo puede mejorar la eficiencia del proceso.

Ejercicio

4.9 Simplifíquese la gramática:

$$\begin{aligned}
 S &\longrightarrow aB|aBC|F \\
 A &\longrightarrow a|\epsilon \\
 B &\longrightarrow b|CA|DF \\
 C &\longrightarrow c|F|\epsilon \\
 D &\longrightarrow EF \\
 E &\longrightarrow e \\
 F &\longrightarrow f|S
 \end{aligned}$$

4.4.6 Formas normalizadas

Una vez que hemos simplificado la gramática de acuerdo con los criterios mencionados, las GIC pueden adoptar dos formas, llamadas formas normalizadas, que resultan útiles en aplicaciones específicas. La primera de ellas es la *forma normalizada de Chomsky*, que nos va a permitir aplicar el algoritmo de análisis sintáctico de Cocke, Younger y Kasami visto más arriba, la otra es la llamada *forma normalizada de Greibach*, útil principalmente para obtener un autómata con pila asociado a la gramática menos indeterminista que en el caso general. En ambos casos, proporcionaremos argumentos de tipo constructivo para justificar que cualquier GIC se puede escribir en forma normalizada. Es importante destacar que, incluso con estas restricciones, no queda únicamente determinada la forma de una gramática que genera un lenguaje dado. En otras palabras: es posible la existencia de diferentes GIC simplificadas y normalizadas equivalentes.

Forma normalizada de Chomsky

Se dice que una gramática $G = (V_N, V_T, P, S)$ está en *forma normalizada de Chomsky* cuando todas las reglas de derivación de P tienen alguna de las formas siguientes:

1. $A \rightarrow BC$ (con $A, B, C \in V_N$).
2. $A \rightarrow a$ (con $a \in V_T$).
3. $S \rightarrow \epsilon$. En este caso, en el cual $\epsilon \in L(G)$, S no puede aparecer en la parte derecha de ninguna regla de P .

Es sencillo transformar cualquier GIC dada en una equivalente en forma de Chomsky, siempre que se halla simplificado previamente. La idea consiste en que las producciones que tengan longitud mayor que 2 se descompongan en una secuencia de reglas de longitud 2. Veamos como se ha obtenido la gramática en forma de Chomsky de la página 71 a partir de la gramática

$$\begin{aligned} S &\rightarrow 0B \mid 1A \\ A &\rightarrow 0 \mid 0S \mid 1AA \\ B &\rightarrow 1 \mid 1S \mid 0BB \end{aligned}$$

Para ello, observamos que las reglas $A \rightarrow 0$ y $B \rightarrow 1$ ya tienen una de las formas admitidas. Las reglas $S \rightarrow 0B \mid 1A$, $A \rightarrow 0S$ y $B \rightarrow 1S$ tienen longitud 2 pero aparecen terminales en el lado derecho, por lo que sustituimos estos terminales por nuevas variables que sólo producen dicho terminal, $Z_0 \rightarrow 0$ y $Z_1 \rightarrow 1$. Con estos cambios la gramática queda

$$\begin{aligned} S &\rightarrow Z_0B \mid Z_1A \\ A &\rightarrow 0 \mid Z_0S \mid Z_1AA \\ B &\rightarrow 1 \mid Z_1S \mid Z_0BB \\ Z_0 &\rightarrow 0 \\ Z_1 &\rightarrow 1 \end{aligned}$$

cuyas reglas de longitud 1 y 2 ya tienen la forma deseada. La regla $A \rightarrow Z_1AA$ la descomponemos como el par de reglas $A \rightarrow Z_1Y_{AA}$, $Y_{AA} \rightarrow AA$ y procediendo análogamente con la regla que falta obtenemos finalmente la gramática en forma normalizada de Chomsky

$$\begin{aligned} S &\rightarrow Z_0B \mid Z_1A \\ A &\rightarrow 0 \mid Z_0S \mid Z_1Y_{AA} \\ B &\rightarrow 1 \mid Z_1S \mid Z_0Y_{BB} \\ Z_0 &\rightarrow 0 \\ Z_1 &\rightarrow 1 \\ Y_{AA} &\rightarrow AA \\ Y_{BB} &\rightarrow BB \end{aligned}$$

El algoritmo siguiente permite realizar este proceso con cualquier gramática simplificada.

Algoritmo 4.7 Conversión de una gramática a la forma normalizada de Chomsky

Entrada: Una gramática $G = (V_N, V_T, P, S)$ simplificada.

Salida: Una gramática $G' = (V'_N, V_T, P', S)$ en forma normalizada de Chomsky.

Método:

1. Incluir en P' todas las reglas de P que ya tengan una de las formas permitidas.
2. Por cada regla de producción de la forma $A \rightarrow X_1X_2$, donde al menos uno de los dos símbolos de la parte derecha es un terminal, añadir la regla $A \rightarrow Z_1Z_2$, donde Z_i representa X_i si $X_i \in V_N$, o es una nueva variable $Z_i = Z_a$ si $X_i = a \in V_T$.
3. Para cada regla de la forma $A \rightarrow X_1 \dots X_n \in P$ con $n > 2$, añadir a P' el conjunto de reglas siguiente:

$$\begin{array}{rcl} A & \longrightarrow & Z_1Y_1 \\ Y_1 & \longrightarrow & Z_2Y_2 \\ & \vdots & \\ Y_{n-2} & \longrightarrow & Z_{n-1}Z_n \end{array}$$

donde Z_k es X_k si $X_k \in V_N$, o una nueva variable Z_a si $X_k = a \in V_T$, e Y_1, \dots, Y_{n-2} son nuevas variables.

4. Para cada nueva variable Z_a introducida en los dos pasos anteriores en sustitución de un terminal, añadir a P' la regla $Z_a \rightarrow a$.
5. Formar V'_N añadiendo todas las nuevas variables creadas en los pasos anteriores.

Ejercicios

4.10 Dada la gramática G

$$\begin{aligned} S &\rightarrow aAa \mid CA \mid BaB \\ A &\rightarrow aaB \mid CD \mid aa \\ B &\rightarrow bB \mid bAB \mid aE \\ C &\rightarrow Ca \mid D \\ D &\rightarrow bD \mid \epsilon \\ E &\rightarrow aB \end{aligned}$$

encuentra una gramática en forma normal de Chomsky para el lenguaje $L(G) - \{\epsilon\}$

4.11 Escríbese en forma de Chomsky la siguiente gramática:

$$\begin{aligned} S &\longrightarrow aB|bA \\ S &\longrightarrow a|aS|bAA \\ S &\longrightarrow b|bS|aBB \end{aligned}$$

Describase $L(G)$.

4.12 Proporciona cotas superior e inferior para la longitud de una cadena $w \in L(G)$ en función del número de pasos de la derivación si G está en forma de Chomsky.

4.13 Escribe una gramática en la forma normalizada de Chomsky equivalente a la expresión regular $(a + b)^*b(a + b)^*a$.

4.14 A la vista del ejercicio anterior, ¿es eficiente el algoritmo CYK para determinar si $x \in L(G)$ para el caso en que G sea regular? Razona la respuesta.

4.15 Dada la gramática de la sección 4.4, comprueba si la cadena $w = 001011$ pertenece a $L(G)$. Dibuja todos los árboles de derivación de w . ¿Qué se puede concluir sobre la ambigüedad de la gramática?

Forma normalizada de Greibach

Ahora mostraremos que, para toda GIC, existe una forma equivalente en la cual la parte derecha de cada regla de derivación comienza siempre por un terminal. Las nociones de *recursión por la izquierda* y su eliminación son muy importantes en la construcción de esta forma normalizada, llamada de Greibach. La forma normalizada de Greibach puede ser especialmente útil para diseñar autómatas de pila capaces de reconocer una cadena de la gramática. En este caso, las derivaciones de P son de forma tal que la cadena de terminales se va construyendo de izquierda a derecha a lo largo de la derivación.

Se dice que una gramática está en *forma normalizada de Greibach* cuando todas sus producciones son de la forma $A \longrightarrow a\alpha$, con $a \in V_T$ y $\alpha \in V_N^*$ (a excepción, si existe, de la regla $S \longrightarrow \epsilon$).

El algoritmo que permite construir la forma normalizada de Greibach de una gramática dada requiere que en ésta se haya eliminado previamente la *recursión por la izquierda*, tal y como se vió en el algoritmo de la página 69. Dicho algoritmo, además de eliminar la recursión por la izquierda, conduce a una gramática en la que todas las reglas satisfacen que si $A_i \longrightarrow A_j\alpha \in P$, entonces $i < j$. Esta propiedad puede ser utilizada para escribir la gramática en la forma normalizada de Greibach. Veamoslo primeramente con un ejemplo. Consideremos la gramática recursiva por la izquierda

$$\begin{aligned} A &\longrightarrow BB \mid a \\ B &\longrightarrow AA \mid b \end{aligned}$$

Para eliminar la recursión por la izquierda renombramos las variables, $A_1 \equiv A$, $A_2 \equiv B$, y aplicamos el algoritmo 4.4.3. El resultado es

$$\begin{aligned} A_1 &\longrightarrow A_2A_2 \mid a \\ A_2 &\longrightarrow aA_1 \mid b \mid aA_1A'_2 \mid bA'_2 \\ A'_2 &\longrightarrow A_2A_1 \mid A_2A_1A'_2 \end{aligned}$$

Observemos que las reglas de A_2 ya tienen la forma de Greibach (aunque no se puede garantizar que esto ocurra siempre con la variable que tiene mayor subíndice, sí podemos garantizar que cuando se aplica el algoritmo 4.4.3 todas las producciones de dicha variable comenzarán por terminal).

El siguiente paso consistirá en conseguir que todas las producciones de las restantes variables, A_1 y A'_2 , también comiencen por terminal. Para ello, puesto que tenemos garantizado que todas las producciones de una variable comienzan por terminal o por una variable con subíndice mayor, podemos sustituir estas últimas por todas sus producciones, es decir, sustituiremos las reglas

$$A_1 \longrightarrow A_2A_2 \mid a$$

por las reglas

$$A_1 \longrightarrow aA_1A_2 \mid bA_2 \mid aA_1A'_2A_2 \mid bA'_2A_2 \mid a$$

y las reglas

$$A'_2 \longrightarrow A_2A_1 \mid A_2A_1A'_2$$

por las reglas

$$\begin{aligned} A'_2 \longrightarrow & aA_1A_1 \mid bA_1 \mid aA_1A'_2A_1 \mid bA'_2A_1 \mid \\ & \mid aA_1A_1A'_2 \mid bA_1A'_2 \mid aA_1A'_2A_1A'_2 \mid bA'_2A_1A'_2 \end{aligned}$$

Con lo que observamos que todas las reglas de producción de la gramática están en forma de Greibach. Si en este paso ocurriera que alguna regla tuviera la forma $A \longrightarrow a\alpha b\beta$, con $a, b \in V_T$ y $\alpha, \beta \in V_N^*$, simplemente tendríamos que sustituir dicha regla por las reglas, $A \longrightarrow a\alpha B\beta$ y $B \longrightarrow b$, con B una nueva variable.

Este proceso queda recogido en el siguiente

Algoritmo 4.8 Conversión a la forma normalizada de Greibach

Entrada: Una gramática simplificada $G = (V_N, V_T, P, S)$ sin recursión por la izquierda.

Salida: Una gramática $G = (V'_N, V_T, P', S)$ en forma normalizada de Greibach.

Método:

1. Ordenar $V_N = \{A_1, A_2, \dots, A_r\}$ de manera que toda producción $A_i \rightarrow \alpha \in P$ satisfaga que el primer símbolo de α sea un terminal o una variable A_j tal que $i < j$ (este orden se consigue naturalmente en el algoritmo de la página 69 de eliminación de la recursión por la izquierda).
2. Tomar $i = r - 1$.
3. Mientras $i \neq 0$,
 - (a) Sustituir todas las reglas de la forma $A_i \rightarrow A_j\alpha$, donde $i < j$, por $A_i \rightarrow \beta_1\alpha | \dots | \beta_m\alpha$ donde $A_j \rightarrow \beta_1 | \dots | \beta_m$ son todas las reglas que tienen como parte izquierda a A_j . Las cadenas β_1, \dots, β_m comienzan todas por terminal.
 - (b) $i = i - 1$.
4. En este punto, todas las reglas (excepto, si existe, $S \rightarrow \epsilon$) tienen partes derechas que comienzan por un terminal. Por cada regla $A \rightarrow aX_1X_2\dots X_k$, sustituir todos los X_j que sean terminales por nuevos no terminales Z_j y añadir la producción $Z_j \rightarrow X_j$.

Ejercicios

4.16 Escribe en forma de Greibach la gramática definida por las siguientes reglas de derivación:

$$\begin{aligned} A_1 &\rightarrow A_2A_3 \\ A_2 &\rightarrow A_3A_1, b \\ A_3 &\rightarrow A_1A_2, a \end{aligned}$$

4.17 Sea la gramática G :

$$\begin{aligned} A &\rightarrow AB|B|C \\ B &\rightarrow bB|aDX|\epsilon \\ C &\rightarrow bC|abE|\epsilon \\ D &\rightarrow bB|cD \\ E &\rightarrow bCX|BCV \\ X &\rightarrow aXC|bBYZ \\ Y &\rightarrow aYb|aE \\ Z &\rightarrow XC|aZY \end{aligned}$$

Escríbase una gramática G' en forma normalizada de Greibach para $L(G) - \{\epsilon\}$.

Capítulo 5

Lenguajes independientes del contexto

5.1 Operaciones con LIC

Algunas operaciones con lenguajes independientes del contexto producen siempre como resultado lenguajes que son también independientes del contexto. Decimos entonces que los LIC son estables bajo dicha operación. Ejemplos de operaciones bajo las cuales los LIC son estables son:

- unión, concatenación y clausura;
- sustitución;
- inversión.

Vamos a justificar estas propiedades, mostrando que se puede escribir una GIC para los lenguajes resultantes de la operación. En cambio, los LIC no son estables bajo:

- intersección;
- complementación;

Unión: Supongamos que tenemos dos gramáticas no contextuales $G_1 = (V_{N1}, V_{T1}, P_1, S_1)$ y $G_2 = (V_{N2}, V_{T2}, P_2, S_2)$. Es sencillo contruir $G = (V_N, V_T, P, S)$ tal que $L(G) = L(G_1) \cup L(G_2)$ tomando:

- $V_N = V_{N1} \cup V_{N2} \cup \{S\}$
- $V_T = V_{T1} \cup V_{T2}$
- $P = P_1 + P_2 + \{S \rightarrow S_1, S \rightarrow S_2\}$

Evidentemente, una cadena de terminales es generada por G si y sólo si es generada por G_1 o por G_2 .

Concatenación: La gramática G para el lenguaje resultante de la concatenación de $L(G_1)$ y $L(G_2)$, $L(G_1)L(G_2)$, se puede obtener de forma similar con

- $V_N = V_{N1} \cup V_{N2} \cup \{S\}$
- $V_T = V_{T1} \cup V_{T2}$
- $P = P_1 \cup P_2 \cup \{S \rightarrow S_1S_2\}.$

La nueva regla $S \rightarrow S_1S_2$ es la que concatena las cadenas de los dos lenguajes.

Clausura: En este caso, para que G sea tal que $L(G) = L(G_1)^*$ podemos definir:

- $V_N = V_{N1} \cup \{S\}$
- $T = T_1$
- $R = R_1 \cup R_2 \cup \{S \rightarrow SS_1, S \rightarrow \epsilon\},$

donde la última regla (una recursión por la izquierda) es la responsable de generar cero o más concatenaciones.

Sustitución: El concepto de sustitución f de terminales por lenguajes independientes del contexto es análogo al que se introdujo de sustitución por conjuntos regulares: f es un homomorfismo tal que la imagen $f(a)$ de cualquier terminal $a \in V_T$ es un cierto LIC, supongamos $f(a) = L_a$, y la imagen de una cadena de terminales es la concatenación de los conjuntos imágenes de cada símbolo de la cadena: $f(a_1a_2\dots) = L_{a_1}L_{a_2}\dots$ Si $G_a = (V_{Na}, V_{Ta}, P_a, S_a)$ es la gramática que genera L_a , $G_b = (V_{Nb}, V_{Tb}, P_b, S_b)$ la que genera L_b , etc., el lenguaje $f(L(G))$ puede generarse mediante una nueva gramática G' tal que $L(G') = f(L(G))$, construida a partir de $G = (V, T, P, S)$: P' se obtiene sustituyendo en P cada aparición de un símbolo terminal a por S_a e incorporando las reglas de derivación de $P_a, P_b, \text{etc.}$ a P' . Además $V_T = V_{Ta} \cup V_{Tb} + \dots$

Inversión: La gramática G' que genera el lenguaje inverso $L(G)^R = \{x \in \Sigma^* : x^R \in L(G)\}$ puede construirse simplemente invirtiendo el orden de escritura de los símbolos de la parte derecha de cada regla de derivación de P : $X \rightarrow \alpha \in P' \iff X \rightarrow \alpha^R \in P$.

Ejercicios

5.1 Busca contraejemplos para justificar que los LIC no son estables bajo intersección(y/o complementación).

5.2 Demuestra que si $f : \Sigma^* \rightarrow \Lambda^*$ es un homomorfismo y $L \in \Lambda^*$ es un LIC entonces $f^{-1}(L)$ es un LIC (estabilidad para la función inversa de un homomorfismo).

5.3 Considérese la gramática de expresiones aritméticas infijas:

$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid a \end{array}$$

Demuéstrese, modificando la gramática convenientemente, que el lenguaje de las expresiones aritméticas postfijas (donde la traducción de $(a+a)*(a+a)$ es $aa + aa + *$) es también un LIC.

5.2 Lema de bombeo para LIC

De forma análoga a como se hizo para los conjuntos regulares, es también posible enunciar un lema de bombeo para LIC. Como cabe esperar, el lema del apartado 3.1 es más restrictivo, de forma que cualquier conjunto que satisfaga la versión para conjuntos regulares satisface automáticamente el nuevo enunciado, pero no a la inversa. En este caso, el hecho de que la gramática sólo pueda usar un número finito de variables permite afirmar que si una derivación es lo suficientemente larga, entonces alguna de las variables ha de utilizarse más de una vez. En ese caso, la gramática ha de ser recursiva, y permite generar un conjunto infinito de cadenas.

Lema 5.1 (*de bombeo*) Para cada gramática independiente del contexto $G = (V_N, V_T, P, S)$, existe un número $n(G) \in \mathbb{N}$ tal que

$$w \in L : |w| \geq n \implies \text{para algún } u, v, x, y, z : \left\{ \begin{array}{l} w = uvxyz \\ |vxy| \leq n \\ |vy| \geq 1 \\ uv^kxy^kz \in L(G) \quad \forall k \in \mathbb{N} \end{array} \right.$$

Demostración. Supondremos que G ha sido simplificada y reescrita en forma normalizada de Chomsky. Es sencillo comprobar —utilizando el método de inducción— que entonces:

$$S \xrightarrow{k} w \implies |w| < 2^k$$

ya que el número de símbolos de la cadena generada no puede más que duplicarse en cada paso y algunos pasos más son necesarios para que las variables sean sustituidas por terminales. Como consecuencia, una elección adecuada para n es $n = 2^{|V_N|}$, siendo $|V_N|$ el número de variables en V_N . Dada una cadena w de longitud $|w| \geq n$, cualquier derivación de w debe contener más de $|V_N|$ pasos, por lo que alguna de las variables $X \in V_N$ debe aparecer en más de un paso de la derivación, o lo que es lo mismo, en más de un nodo del árbol de derivación.

Es posible concretar aún más pues en realidad, alguna repetición de variable debe darse en el camino desde el nodo raíz hasta alguno de los terminales (basta con elegir un camino con longitud mayor que $|V_N|$). Por tanto, si nos fijamos en la variable X que primero se repite en el camino (comenzando desde el nodo terminal), ésta aparece en dos nodos del camino, que llamaremos superior e inferior respectivamente. El subárbol inferior, con raíz en $X^{(i)}$, genera una cadena de terminales x , que aparecen en los nodos terminales del subárbol:

$$X^{(i)} \xrightarrow{*} x$$

Por otro lado, el subárbol superior, con raíz en $X^{(s)}$, contiene en su interior al inferior y por tanto, genera una cadena ψ que contiene a x : $\psi = vxy$, es decir:

$$X^{(s)} \xrightarrow{*} \alpha_1 X^{(i)} \alpha_2 \xrightarrow{*} vxy$$

donde

$$\begin{array}{rcl} \alpha_1 & \xrightarrow{*} & v \\ \alpha_2 & \xrightarrow{*} & y \end{array}$$

Estas subcadenas v, x, y son precisamente las que satisfacen las condiciones del lema:

- $|vxy| \leq n$ ya que hemos tomado la variable X que primero se repite en el camino (comenzando desde el nodo terminal). Disponemos entonces, como máximo, de $|V| + 1$ variables (la X se utiliza dos veces), y es sencillo probar que $2^{|V|} = n$ es la longitud máxima de la subcadena generada a partir de X .
- $|vy| \geq 1$, ya que el vértice superior, por estar G en forma de Chomsky simplificada, sólo puede ser del tipo $X \rightarrow AB$ y ni A ni B son anulables.
- $uv^kxy^kz \in L(G)$ ya que

$$\left. \begin{array}{rcl} X & \xrightarrow{*} & vXy \\ X & \xrightarrow{*} & x \end{array} \right\} \implies X \xrightarrow{*} v^k X y^k \xrightarrow{*} v^k x y^k$$

luego $S \xrightarrow{*} uXz \xrightarrow{*} uv^kxy^kz$ para cualquier $k \in \mathbb{N}$.

■

Como en el caso de los LR el lema de bombeo permite únicamente descartar algunos conjuntos como LIC, pero no demuestra que un conjunto dado lo sea. Por ejemplo, sabemos que $\{a^i b^i : i \in \mathbb{N}\}$ es un LIC, pero no por el hecho de que no sepamos hallar el n que figura en el lema. En cambio, $\{a^i b^i c^i : i \in \mathbb{N}\}$ no lo es. De nuevo, la demostración procede por reducción al absurdo. Supongamos que existe ese n y tomemos, por ejemplo, la cadena $w = a^n b^n c^n$. No es difícil comprobar que ninguna partición de w es compatible con el lema de bombeo, ya que vxy puede contener como mucho dos tipos de letra diferentes, y al bombear la subcadena, se deshace el empate entre los tres tipos de letra.

Ejercicios

5.4 Comprueba que si un lenguaje satisface el lema de bombeo para conjuntos regulares, también satisface el de LIC.

5.5 Comprueba si los siguientes lenguajes son LIC:

- $\{a^i b^j : j \geq i\}$
- $\{a^i b^i c^j : j \geq i\}$
- $\{ww : w \in \Sigma^*\}$
- cadenas con igual número de apariciones de a , de b y de c
- $\{w = a^i b^j c^i d^j\}$

Una vez más nos encontramos ante la cuestión de las limitaciones de los lenguajes teóricos. Si vimos que los lenguajes regulares no eran capaces de describir adecuadamente situaciones de aparición frecuente en el campo de la computación (por ejemplo, al determinar si los paréntesis de una expresión algebraica están correctamente anidados), observamos ahora que tampoco los lenguajes independientes del contexto van a resolver todos los problemas. Por ejemplo, el ejercicio 5.5 está relacionado con el problema de comprobar si el número y tipo de argumentos en una llamada a una función concuerda con los que aparecen en la declaración de la función. Aunque existen lenguajes de rango superior a los aquí estudiados, la complejidad de su tratamiento constituye un obstáculo de importante para su utilización. Veremos en breve, que incluso el paso desde los lenguajes regulares a los LIC supone un significativo aumento en el coste de los algoritmos que se aplican (en algunos casos, la misma cuestión se transforma en indecidible o su complejidad puede resultar intratable comparada con el caso de los LR). Como siempre, es necesario buscar un compromiso entre exactitud y complejidad. En otros campos de trabajo, como la inferencia gramatical o

aprendizaje, suele recurrirse a los modelos más sencillos de lenguaje, aunque estos sean sólo aproximaciones a los correctos¹, si esto permite una mejor comprensión y mayor rapidez de las predicciones del modelo. También es posible la situación opuesta, en la que un lenguaje presumiblemente finito resulta descrito de forma mucho más compacta por su aproximación regular o como LIC.

5.3 Algoritmos

En esta sección vamos a considerar la cuestión de determinar si dada una gramática G entonces es $L(G)$ vacío, finito o infinito, así como la determinación de si $x \in \Sigma^*$ pertenece o no al lenguaje.

¿Es $L(G) = \emptyset$? Para determinar si $L(G)$ es vacío basta con simplificar G . Trivialmente: $L(G) = \emptyset \iff S$ es inútil. Para determinar si S es útil, basta con usar la primera parte del algoritmo de la página 81, donde se determina el conjunto N_e de los no terminales capaces de generar cadenas de terminales. Si $S \in N_e$, entonces S no es inútil, y por tanto $L(G)$ no es vacío.

¿Es $L(G)$ infinito? Para determinar si $L(G)$ es finito o infinito basta con comprobar si G , en su versión simplificada, es o no recursiva. La simplificación es necesaria para que no aparezcan símbolos inútiles. Por otro lado, el carácter recursivo de G se puede comprobar de forma sencilla si se dibuja un grafo en el que cada nodo está etiquetado con una de las variables y se conecta a su vez mediante arcos dirigidos a todas aquellas variables que aparecen a la derecha de alguna de sus derivaciones. Es decir, $A \rightsquigarrow B$ si existe alguna derivación en R tal que $A \longrightarrow \alpha B \beta$. Entonces $L(G)$ es finito si y sólo si el grafo dibujado es acíclico, es decir, no contiene bucles o ciclos (caminos que partiendo de alguna de las variables conduzca de nuevo hasta la misma variable: $A_0 \rightsquigarrow A_1 \rightsquigarrow \dots \rightsquigarrow A_n \rightsquigarrow A_0$). Esto es equivalente a afirmar que no existe una secuencia de derivaciones del tipo: $A_0 \longrightarrow \alpha_1 A_1 \beta_1 \longrightarrow \dots \alpha_n A_n \beta_n \longrightarrow \alpha_{n+1} A_0 \beta_{n+1}$.

Demostración. Para simplificar, supongamos que G está en forma de Chomsky y por tanto $|\alpha_i \beta_i| = i$. Como además G no contiene símbolos inútiles, existen $y, x, v : \alpha_{n+1} \xrightarrow{*} v, \beta_{n+1} \xrightarrow{*} y, A_0 \xrightarrow{*} x$. Por tanto:

$$S \xrightarrow{*} uA_0z \xrightarrow{*} uv^kxy^kz, |vy| \geq 1$$

luego existen cadenas de longitud superior a cualquiera dada y el conjunto no puede ser finito. ■

¹En algunos casos, ni siquiera sabemos con certeza cuál es el modelo correcto.

5.6 Determinese si $L(G)$ es vacío, finito, o infinito para la gramática G definida por las siguientes reglas

$$\begin{aligned} S &\longrightarrow AB \\ A &\longrightarrow AS|S|C \\ B &\longrightarrow b \\ C &\longrightarrow c|\epsilon \end{aligned}$$

¿Genera G la cadena x ? Pasamos ahora a considerar la cuestión de si una cadena dada $x \in \Sigma^*$ es generada o no por la gramática G . Esta cuestión constituye el problema central de la tarea denominada *análisis* o *reconocimiento sintáctico*. Si la longitud de x es $l = |x|$ y G está en forma de Chomsky, cualquier derivación de x no puede contener más de k pasos, siendo k tal que $l < 2^k$. Por tanto, siempre es posible comprobar en un tiempo finito—pues el número de derivaciones con menos de k pasos está acotado—si se da el caso de que $x \in L(G)$. Sin embargo, el tiempo que requiere esta comprobación crece exponencialmente con $|x|$ (el número de cadenas de longitud l crece exponencialmente con l). El algoritmo 4.2 descubierto independientemente por Cocke, Younger y Kasami[8]) es un poco más elaborado, y permite que el tiempo de ejecución crezca menos que $|x|^3$. Recordemos que este algoritmo, además de permitirnos determinar si $x \in L(G)$, proporciona la tabla de análisis sintáctico que contiene la información suficiente para construir todos los árboles de derivación de x .

El método de Cocke-Younger-Kasami no es el mejor algoritmo general para determinar si una cadena pertenece al lenguaje generado por una gramática. Existen algoritmos generales como el de Earley[8], cuya complejidad temporal es con frecuencia menor (por ejemplo si la gramática es no ambigua), pero cuya descripción escapa al alcance temático de este libro.

Capítulo 6

Autómatas con pila y analizadores sintácticos

Ya ha sido mencionado (pgs. 5 y 60) que los autómatas con pila (AP) ocupan una posición intermedia entre los autómatas finitos y las máquinas de Turing, pues la estructuración de los datos en forma de pila supone una gran rigidez a la hora de almacenar y acceder a la información allí depositada. Este acceso se realiza de forma muy restrictiva, porque en cada instante sólo el elemento superior de la pila es accesible¹ (aquel que fue introducido en último lugar). Es posible demostrar que una máquina de Turing es equivalente a un autómata con dos pilas de almacenamiento, y también que un AP con una pila cuya profundidad está limitada no es más que un AFD.

La equivalencia entre los lenguajes independientes del contexto y los autómatas de pila es menos satisfactoria que la existente entre los lenguajes regulares y los autómatas finitos deterministas, ya que en este caso no existe necesariamente un autómata de pila determinista equivalente a un AP indeterminista dado, y algunos LIC sólo pueden ser reconocidos por AP indeterministas. Afortunadamente, los AP deterministas pueden describir la sintaxis de la mayoría de los lenguajes de programación. Para reconocer las estructuras válidas de los lenguajes de programación se utilizan habitualmente analizadores sintácticos LL y LR, que son subclases (propias) de los autómatas con pila deterministas.

6.1 Definiciones

Un *autómata con pila* es una séptupla $M = (Q, \Sigma, \Gamma, \delta, Q_1, Z_1, F)$ donde:

- Q es el conjunto finito de estados del autómata y $q_1 \in Q$ el estado inicial.

¹Siguiendo el procedimiento LIFO, siglas inglesas de *last-in first-out*, “el último que entró es el primero que sale”.

- Σ es el alfabeto finito para el lenguaje.
- Γ es un alfabeto finito o conjunto de símbolos que pueden ser almacenados en la pila, la cual no se encuentra vacía inicialmente² sino que contiene $Z_1 \in \Gamma$.
- F es el subconjunto de los estados de aceptación.
- δ es una función de transición de $Q \times \Sigma^* \times \Gamma$ en subconjuntos finitos de $Q \times \Gamma^*$.

Las transiciones del AP vienen determinadas por la función δ , que en este caso depende de tres argumentos: el estado previo q del AP, el símbolo de entrada a y el símbolo superior de la pila Z . El resultado (o resultados en el caso indeterminista) es un nuevo estado del autómata (p) y un nuevo estado de la pila que resulta de sustituir el símbolo superior Z por una cadena finita de símbolos de Γ ($\gamma \in \Gamma^*$).

Adoptaremos el convenio de que los símbolos de γ se apilan de derecha a izquierda, es decir si $\gamma = XYZ$ entonces X es el nuevo símbolo superior de la pila³. En general, el AP será indeterminista y varias transiciones son posibles:

$$\delta_a(q, Z) = \{(p_k, \gamma_k)\}_{k=1}^m$$

A la terna formada por un estado q del AP, una cierta subcadena $w \in \Sigma^*$ de entrada (el sufijo que resta por leer) y la cadena que describe el contenido de la pila ($\gamma \in \Gamma^*$) se le conoce con el nombre de *descripción instantánea* del AP, pues nos proporciona toda la información que necesitamos para describir el estado del AP y su evolución posterior. Definiremos una relación de derivación entre descripciones instantáneas cuando estas corresponden a dos posibles estados consecutivos del autómata⁴:

$$(q, aw, Z\alpha) \longrightarrow (p, w, \beta\alpha) \iff (p, \beta) \in \delta_a(q, Z)$$

El lenguaje aceptado por un AP se define entonces como

$$L(M) = \{w \in \Sigma^* : (q_1, w, Z_1) \xrightarrow{*} (p, \epsilon, \gamma), p \in F, \gamma \in \Gamma^*\}$$

es decir, el conjunto de aquellas cadenas que llevan al autómata desde el estado inicial a un estado del subconjunto F . Una definición alternativa de

²La inclusión de un símbolo inicial en la pila no es esencial en la definición, pero resulta conveniente para la analogía con las GIC, al desempeñar Z_1 el papel del símbolo inicial S de las gramáticas.

³El hecho de que la función de transición pueda sustituir un símbolo Z por una cadena γ tampoco es esencial a la definición. De hecho, podría definirse el AP de forma que no pudiese eliminar o añadir más de un símbolo de la pila en cada paso sin variar la potencia de los AP, pero la definición adoptada resulta más cómoda. Tampoco es esencial que en la definición la función δ esté definida sobre $Q \times \Sigma^* \times \Gamma$, ya que bastaría con $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$.

⁴Otros autores usan el símbolo \vdash para esta relación.

lenguaje aceptado es:

$$N(M) = \{w \in \Sigma^* : (q_1, w, Z_1) \xrightarrow{*} (p, \epsilon, \epsilon), p \in Q\}$$

En esta última definición, una palabra w es aceptada cuando la pila queda vacía en cualquier estado del autómata y la palabra ha sido leída completamente (no es necesario, por tanto, distinguir estados finales). Más adelante veremos que estas dos definiciones son equivalentes.

Si dada cualquier descripción instantánea (DI) está unívocamente definida la siguiente DI, diremos que el AP es determinista. En caso contrario, diremos que el AP es indeterminista. A diferencia con el caso de los AFD, no basta con que se cumpla que $\text{card}(\delta_a(q, Z)) = 1$. Por ejemplo, si $\delta_a(q, \epsilon) = (p_1, Z_1)$ y $\delta_a(q, X) = (p_2, Z_2)$ no está fijado cuál es la descripción instantánea que sigue a $(q, aw, X\gamma)$. Por tanto, se dice que una autómata es determinista si, para todos los estados $q \in Q$ y símbolos de la pila $Z \in \Gamma$:

- o bien $\text{card}(\delta_a(q, Z)) \leq 1$ para todo $a \in \Sigma$ y $\text{card}(\delta_\epsilon(q, Z)) = 0$,
- o bien $\text{card}(\delta_a(q, Z)) = 0$ para todo $a \in \Sigma$ y $\text{card}(\delta_\epsilon(q, Z)) \leq 1$.

De esta forma se asegura que sólo existe un movimiento posible (como mucho) tras cada descripción instantánea.

Desafortunadamente, existen lenguajes independientes del contexto que sólo pueden ser reconocidos por AP indeterministas, de modo que los lenguajes generados por AP deterministas es un subconjunto propio de los LIC.

Ejercicios

6.1 Construye la función de transición de un AP que reconozca $C = \{wcw^R : w \in (a + b)^*\}$

6.2 Construye un AP que reconozca $D = \{ww^R : w \in (a + b)^*\}$. Justifica que D no puede ser generado por un AP determinista.

Equivalencia entre criterios de aceptación: La equivalencia entre las definiciones de aceptación dadas en esta sección, $L(M)$ y $N(M)$ se justifica por el hecho de que dado un autómata M tal que reconoce un cierto LIC por vaciado de pila $L = N(M)$ siempre es posible construir otro AP que acepta L cuando el autómata entra en un estado de aceptación: $L = L(M')$ y viceversa, dado $L = L(M)$ se puede construir M' tal que $L = N(M')$.

Supongamos $L = L(M)$, y construyamos el autómata M' tal que $N(M') = L$, a partir de M . Los estados de aceptación de M tienen ahora, además de las transiciones que pudiera tener M , la posibilidad de pasar a un nuevo estado q_e que vacía la pila cuando el autómata accede a un estado de F . El fondo de la pila es marcado con un símbolo inicial diferente X_1 , que sólo puede ser sacado de ella por q_e , para evitar que sean aceptadas cadenas que

vacían la pila en M (y que no están en $L(M)$, es decir, no llegan a un estado de aceptación de M). Para ello se cambia el estado inicial a q'_1 con contenido en la pila X_1 . De esta forma $Q' = Q \cup \{q'_1, q_e\}$ y $\Gamma' = \Gamma \cup \{X_1\}$. La función de transición δ' difiere de δ en los siguientes puntos:

- $\delta'_\epsilon(q'_1, X_1) = (q_1, Z_1 X_1)$
- $q \in F \implies (q_e, \epsilon) \in \delta'_\epsilon(q, Z)$
- $(q_e, \epsilon) \in \delta'_\epsilon(q_e, Z)$

Con las definiciones anteriores tenemos $N(M') = L(M)$.

En el caso contrario de que tengamos $L = N(M)$, definimos M' tal que accede a un estado final q_f en el momento en que la pila de M se vacía. Podemos tomar: $M' = (Q \cup \{q'_1, q_f\}, \Sigma, \Gamma \cup \{X_1\}, \delta', q'_1, X_1, \{q_f\})$. Las diferencias entre δ' y δ son:

- $\delta'_\epsilon(q'_1, X_1) = (q_1, Z_1 X_1)$
- $(q_f, \epsilon) \in \delta'_\epsilon(q, X_1)$

Con estas definiciones $N(M) = L(M')$.

6.2 Autómata de pila para una GIC

Dada $G = (V_N, V_T, P, S)$, vamos a construir un AP que acepte precisamente las cadenas de $L(G)$. Para ello mostraremos dos procedimientos diferentes, dependiendo de si partimos de la gramática en una forma arbitraria o de su transformada en la forma normalizada de Greibach. En el primer caso, el funcionamiento del autómata se realiza de acuerdo con alguna forma de gramática que tiene una interpretación sencilla. En este procedimiento se basará la construcción de analizadores sintácticos de la sección 6.4. En el segundo caso se obtiene un autómata menos indeterminista, y más conveniente para el estudio teórico de los LIC.

La construcción inmediata [8] define, dada la gramática G , un autómata $M = (Q, \Sigma, \Lambda, \delta, q_1, Z_1, F)$ que tiene:

- un único estado que es, obviamente, también el inicial ($Q = \{q\}$, $q_1 = q$);
- un alfabeto de entrada igual al alfabeto terminal de la gramática ($\Sigma = V_T$);
- un alfabeto de pila ($\Lambda = V$) idéntico al conjunto de todos los símbolos de la gramática; y
- el símbolo inicial de la gramática como símbolo inicial de pila ($Z_1 = S$);

y no tiene estados de aceptación ($F = \emptyset$). La función de transición δ se define como:

$$\delta_\epsilon(q, A) = \bigcup_{A \xrightarrow{\alpha} \alpha \in P} \{(q, \alpha)\}, \forall A \in V_N$$

$$\delta_a(q, a) = \{(q, \epsilon)\}, \forall a \in V_T.$$

Conviene recordar que el autómata así construido es indeterminista siempre que haya más de una regla de derivación con la misma parte izquierda. Se puede demostrar que $L(G) = N(M)$; es decir, el autómata acepta por vaciado de pila el lenguaje generado por G .

En el segundo procedimiento se reescribe G en forma normalizada de Greibach. Se define entonces $M = (Q, \Sigma, \Gamma, \delta, \{q\}, Z_1, F)$ donde $Q = \{q\}$ contiene un solo estado, los símbolos de la pila son las variables de la gramática $\Gamma = V_N$, el alfabeto del autómata es $\Sigma = V_T$, el contenido inicial de la pila es el símbolo inicial de G ($Z_1 = S$) y $F = \emptyset$. Por último, la función de transición se obtiene directamente a partir las reglas de derivación:

$$\delta_a(q, A) = (q, \gamma) \iff A \xrightarrow{} a\gamma \in P$$

Con estas definiciones $S \xrightarrow{*} w \iff w \in N(M)$, pues en la pila se encuentra almacenado en cada instante la subcadena de variables que resta por sustituir.

Ejercicios:

6.3 Construye un autómata de pila que permita reconocer $L(G)$ para la gramática del ejercicio 4.4

6.4 Constrúyase un autómata de pila equivalente a la gramática:

$$\begin{array}{lcl} S & \longrightarrow & a|AaS|BB \\ A & \longrightarrow & SB|b|CC \\ B & \longrightarrow & aB|bD \\ C & \longrightarrow & bS|\epsilon \\ D & \longrightarrow & aD|bB \\ E & \longrightarrow & a|aS|CA \end{array}$$

6.5 Constrúyase un autómata de pila para la gramática:

$$\begin{array}{lcl} S & \longrightarrow & AA|0 \\ A & \longrightarrow & SS|1 \end{array}$$

6.3 Gramática correspondiente a un AP

Dado el autómata $M = (Q, \Sigma, \Gamma, \delta, q_1, Z_1, F)$ queremos construir una gramática independiente del contexto $G = (V, \Sigma, S, R)$ tal que $L(G) = N(M)$. Para ello tomamos:

$$V_N = \{S\} \cup \{\langle q, A, p \rangle : q, p \in Q, A \in \Gamma\}$$

Los objetos $\langle q, A, p \rangle$ son los símbolos de la nueva gramática y representan el objetivo del autómata M de pasar del estado q al estado p eliminando el símbolo A de la parte superior de la pila, ya sea directamente o a través de una secuencia de pasos intermedios. Como el objetivo de M para que acepte una cierta cadena es eliminar Z_1 de la pila, pasando al mismo tiempo desde q_1 hasta un estado cualquiera q , escribiremos como reglas de derivación de G :

$$S \longrightarrow \langle q_1, Z_1, p \rangle \quad \forall p \in Q$$

Si un símbolo a (o quizás ϵ) de la cadena de entrada w produce una transición desde un nodo q hasta otro p_0 y reemplaza A por $\gamma = B_1B_2\dots B_m$, es decir, si $(p_0, \gamma) \in \delta_a(q, A)$, el resultado es que la lectura de a produce un cambio en el objetivo del autómata, que es ahora pasar de p_0 hasta el nodo dado $p = p_m$ descargando la cadena γ :

$$\langle p_0, \gamma, p_m \rangle = \langle p_0, B_1, p_1 \rangle \langle p_1, B_2, p_2 \rangle \dots \langle p_{m-1}, B_m, p_m \rangle$$

Por ello, por cada transición $(p_0, \gamma) \in \delta_a(q, A)$ con $\gamma = B_1B_2\dots B_n$, añadimos un conjunto de reglas de derivación a R :

$$\langle q, A, p_m \rangle \longrightarrow a \langle p_0, B_1, p_1 \rangle \langle p_1, B_2, p_2 \rangle \dots \langle p_{m-1}, B_m, p_m \rangle,$$

para cada $p_1, p_2, \dots, p_m \in Q$. Si $m = 0$, entonces la regla es simplemente

$$\langle q, A, p_m \rangle \longrightarrow a.$$

Es posible demostrar [9] que de esta forma se obtiene una gramática G tal que $L(G) = N(M)$. La gramática obtenida puede contener símbolos inútiles, que se pueden de eliminar usando el algoritmo 4.6.

Ejercicios

6.6 Construye la gramática equivalente al siguiente autómata de pila $M = \{q_1, q_2\}, \{a, b\}, \{X, Z\}, \delta, q_1, Z, \emptyset\}$ cuya función de transición está definida por:

$$\begin{array}{ll} \delta_a(q_1, Z) = (q_1, XZ) & \delta_b(q_2, X) = (q_1, \epsilon) \\ \delta_a(q_1, X) = (q_1, XX) & \delta_\epsilon(q_2, \epsilon) \\ \delta_b(q_1, X) = (q_2, \epsilon) & \delta_\epsilon(q_2, \epsilon) \end{array}$$

6.7 Reconstruye la gramática del ejercicio 6.3 a partir de su autómata de pila.

6.8 Demuestra que dados L (LIC) y R (regular), entonces: $L \cap R$ es un LIC; RL^{-1} es regular; LR^{-1} es LIC.

6.4 Analizadores sintácticos

Los autómatas de pila se utilizan en los intérpretes y compiladores de los lenguajes de programación para realizar el denominado *análisis sintáctico*, es decir, para determinar si una determinada construcción (sentencia, bucle, etc.) está bien formada. De hecho, en los compiladores, se acoplan al autómata de pila mecanismos paralelos de generación de código, de manera que la compilación consiste básicamente en una tarea de *traducción dirigida por la sintaxis*, realizada por un AP. En los intérpretes, se dota al autómata de pila de la capacidad de ejecutar las acciones que solicita el programa.

Los autómatas de pila que se utilizan son normalmente deterministas, o, en todo caso, con un indeterminismo muy limitado, que puede ser resuelto mediante técnicas de *retroceso limitado* en caso de haber elegido una opción incorrecta.

Los lenguajes independientes del contexto deterministas son un subconjunto de los LIC que pueden ser reconocidos por AP deterministas. Dentro de ellos existen subclases, determinadas por la naturaleza de los autómatas que los reconocen. Existen esencialmente dos tipos de análisis sintáctico: el análisis *descendente* (en inglés *top-down*) y el análisis *ascendente* (en inglés *bottom-up*). El análisis descendente se basa en comenzar con el símbolo inicial e ir estableciendo derivaciones por la izquierda que vayan casando con la cadena de entrada. El análisis ascendente consiste en simular la inversa de una derivación por la derecha, desplazando la entrada a la pila y reduciéndola de acuerdo con las reglas de la gramática cuando en la pila aparece la parte derecha de una producción, hasta que se acaba por tener el símbolo inicial de la gramática.

En esta sección sólo nos fijaremos en el análisis descendente, que resulta no ser en general tan potente como el ascendente (existen lenguajes para los cuales es imposible construir un analizador descendente determinista). Los lenguajes y gramáticas que trataremos son los denominados LL(k).

Una gramática no ambigua es LL(k) si el autómata de pila correspondiente, construido de acuerdo con las prescripciones de este capítulo, que realiza el análisis descendente, de derecha a izquierda (*Left-to-right*, la primera L del acrónimo) según una derivación por la izquierda (*Leftmost derivation*, la segunda L), es tal que su indeterminismo puede ser eliminado si se le dota de la capacidad de *espiar* k símbolos de la parte de la cadena de entrada que queda por leer antes de leerlos *oficialmente*. A esta última capacidad se le llama *anticipación* o *preanálisis*, en inglés *lookahead*.

Un ejemplo de gramática LL(1) servirá para ilustrarlo. Considérese la típica gramática de expresiones aritméticas rudimentarias:

$$\begin{array}{lcl} E & \longrightarrow & E + T | T \\ T & \longrightarrow & T * F | F \\ F & \longrightarrow & (E) | a \end{array}$$

Tal como está escrita, esta gramática es recursiva por la izquierda. El autómata de pila correspondiente, que sería indeterminista, podría caer en un bucle infinito en algún caso: por ejemplo, podría decidir repetir indefinidamente la acción siguiente: sacar E de la pila, leyendo ϵ de la entrada, y meter en la pila $E + T$. Si se elimina la recursión por la izquierda de esta gramática, el autómata sigue siendo indeterminista, aunque sin este tipo de posibles bucles infinitos:

$$\begin{array}{lcl} E & \longrightarrow & TE' \\ E' & \longrightarrow & +TE' | \epsilon \\ T & \longrightarrow & FT' \\ T' & \longrightarrow & *FT' | \epsilon \\ F & \longrightarrow & (E) | a \end{array}$$

La tabla de transiciones del autómata sería indeterminista:

$$\begin{array}{ll} \delta_\epsilon(q_1, E) & = \{(q_1, TE')\} \\ \delta_+(q_1, E') & = \{(q_1, TE')\} * \\ \delta_\epsilon(q_1, E') & = \{(q_1, \epsilon)\} * \\ \delta_\epsilon(q_1, T) & = \{(q_1, FT')\} \\ \delta_*(q_1, T') & = \{(q_1, FT')\} * \\ \delta_\epsilon(q_1, T') & = \{(q_1, \epsilon)\} * \\ \delta_((q_1, F) & = \{(q_1, E)\} \\ \delta_a(q_1, F) & = \{(q_1, \epsilon)\} \end{array}$$

(las transiciones marcadas con un asterisco hacen que el autómata sea indeterminista). Sin embargo, si dotamos al autómata de la capacidad de mirar el primer símbolo de la parte de la cadena que queda por leer antes de decidir qué acción ha de tomar, es posible realizar un analizador completamente determinista para esta gramática. Denominaremos “pre” al conjunto de símbolos de preanálisis de cada transición, que además puede ser utilizado para determinar si una transición determinista dada sería correcta a la vista de la cadena que queda por leer. La nueva tabla de transiciones, con los símbolos de preanálisis correspondientes se muestra en la figura 6.1. Nótese que en el nuevo autómata, las únicas transiciones que consumen símbolos de la entrada son las que ocurren cuando hay un terminal en la cima de la pila, y en este caso no es necesario el preanálisis. El autómata así construido es un analizador LL(1), dado que sólo necesita mirar un símbolo de la entrada para decidir la acción. Así, el conjunto de descripciones instantáneas

$\delta_\epsilon(q_1, E)$	$\{(q_1, TE')\}$	pre = $\{(), a\}$
$\delta_\epsilon(q_1, E')$	$\{(q_1, +TE')\}$	pre = $\{+\}$
$\delta_\epsilon(q_1, E')$	$\{(q_1, \epsilon)\}$	pre = $\{\}, \epsilon\}$
$\delta_\epsilon(q_1, T)$	$\{(q_1, FT')\}$	pre = $\{(), a\}$
$\delta_\epsilon(q_1, T')$	$\{(q_1, *FT')\}$	pre = $\{*\}$
$\delta_\epsilon(q_1, T')$	$\{(q_1, \epsilon)\}$	pre = $\{+, \), \epsilon\}$
$\delta_\epsilon(q_1, F)$	$\{(q_1, (E))\}$	pre = $\{()\}$
$\delta_\epsilon(q_1, F)$	$\{(q_1, a\}$	pre = $\{a\}$
$\delta_a(q_1, a)$	$\{(q_1, \epsilon)\}$	
$\delta_+(q_1, +)$	$\{(q_1, \epsilon)\}$	
$\delta_*(q_1, *)$	$\{(q_1, \epsilon)\}$	
$\delta_((q_1, ()$	$\{(q_1, \epsilon)\}$	
$\delta_((q_1,))$	$\{(q_1, \epsilon)\}$	

Figura 6.1: Tabla de transiciones para la gramática LL(1) descrita en el texto.

correspondientes al análisis de la cadena $(a + a) * a$ sería el mostrado en la figura 6.2.

Cuando una entrada no está definida o el símbolo de anticipación no es del conjunto que determina una transición, se hace que el analizador indique que la cadena no pertenece al lenguaje. La construcción de la tabla de transición, con los símbolos de preanálisis, a partir de la gramática, es algorítmica, aunque el algoritmo excede el alcance de este libro. De hecho, cuando la tabla resultante de aplicar el algoritmo es indeterminista incluso considerando el preanálisis, se dice que la gramática no es LL(1).

Se puede demostrar que para cada autómata de pila con preanálisis existe un autómata de pila determinista sin preanálisis. La demostración, que se deja como ejercicio, se basa en construir un nuevo autómata donde el símbolo de preanálisis forma parte del estado, y donde se establece un esquema diferente de lectura de la cadena de entrada, lo que permite clasificar los lenguajes reconocidos por analizadores LL(1) (y, en general, LL(k)) como LIC deterministas.

$(q_1,$	$(a + a) * a,$	$E)$	\longrightarrow
$(q_1,$	$(a + a) * a,$	$TE')$	\longrightarrow
$(q_1,$	$(a + a) * a,$	$(E)E')$	\longrightarrow
$(q_1,$	$a + a) * a,$	$E)E')$	\longrightarrow
$(q_1,$	$a + a) * a,$	$TE')E')$	\longrightarrow
$(q_1,$	$a + a) * a,$	$FT'E')E')$	\longrightarrow
$(q_1,$	$a + a) * a,$	$aT'E')E')$	\longrightarrow
$(q_1,$	$+a) * a,$	$T'E')E')$	\longrightarrow
$(q_1,$	$+a) * a,$	$+T'E'E')E')$	\longrightarrow
$(q_1,$	$a) * a,$	$T'E'E')E')$	\longrightarrow
$(q_1,$	$a) + a,$	$FT'E'E')E')$	\longrightarrow
$(q_1,$	$a) + a,$	$aT'E'E')E')$	\longrightarrow
$(q_1,$	$) + a,$	$T'E'E')E')$	\longrightarrow
$(q_1,$	$) + a,$	$E'E')E')$	\longrightarrow
$(q_1,$	$) + a,$	$E')E')$	\longrightarrow
$(q_1,$	$) + a,$	$)E')$	\longrightarrow
$(q_1,$	$+a,$	$E')$	\longrightarrow
$(q_1,$	$+a,$	$+TE')$	\longrightarrow
$(q_1,$	$a,$	$TE')$	\longrightarrow
$(q_1,$	$a,$	$FT'E')$	\longrightarrow
$(q_1,$	$a,$	$aT'E')$	\longrightarrow
$(q_1,$	$\epsilon,$	$T'E')$	\longrightarrow
$(q_1,$	$\epsilon,$	$E')$	\longrightarrow
$(q_1,$	$\epsilon,$	$\epsilon)$	

Figura 6.2: Conjunto de descripciones instantáneas correspondientes al análisis de la cadena $(a + a) * a$ (ver texto).

Apéndice A

Nociones básicas de teoría de conjuntos

Partiremos, de forma acrítica, de la idea intuitiva de *conjunto* como reunión de objetos o elementos, y supondremos que dado cualquier objeto siempre es posible determinar si este pertenece o no al conjunto¹.

Se suele denotar con letras minúsculas a los elementos y con mayúsculas a los conjuntos. Por ejemplo, $A = \{a, e\}$ denota un conjunto A formado por dos elementos a y e . Diremos que a pertenece a A y se escribe $a \in A$. En este punto, aceptaremos que los conjuntos pueden quedar definidos bien por *extensión* (enumeración de todos sus elementos) o por *comprensión* (enunciación de una propiedad característica). A partir de aquí, utilizaremos las siguientes definiciones y propiedades[2, 3]:

- A es *parte* o *subconjunto* de B o *está incluido en* B ($A \subset B$) $\iff \forall x : x \in A \Rightarrow x \in B$. La relación entre A y B se denomina relación de inclusión² y diremos también que B contiene a A ($B \supset A$).
- A es *idéntico* a B ($A = B$) $\iff A \subset B \wedge B \subset A$
- A es *parte propia* o *subconjunto propio* de B , o *está propiamente incluido en* B $\iff A \subset B \wedge A \neq B$
- Se llama *conjunto vacío* (\emptyset) al conjunto que no contiene ningún elemento y que satisface $\forall A : \emptyset \subset A$
- El *conjunto de las partes* de un conjunto ($P(U)$) es el conjunto formado por todos los subconjuntos de U : $A \in P(U) \iff A \subset U$
- El conjunto *unión* de A y B : $A \cup B = \{x : x \in A \vee x \in B\}$

¹Esta restricción prohíbe definiciones como la célebre *paradoja del barbero* de B.Russell[1], y en general, la existencia de conjuntos que se contengan a sí mismos, como el conjunto de todos los conjuntos.

²La inclusión de conjuntos es un ejemplo de relación de orden. Véase la página 108.

- El conjunto *intersección* de A y B : $A \cap B = \{x : x \in A \wedge x \in B\}$
- El conjunto *complementario* de A respecto a U : $\bar{A} = \{x \in U : x \notin A\}$
- $A + B = A \cup B$
- $A - B = A \cap \bar{B}$ (*diferencia*)
- $A \oplus B = (A - B) + (B - A)$ (*diferencia simétrica*)
- $A \times B = \{(x, y) : x \in A \wedge y \in B\}$ (*producto cartesiano*)

Ejercicios

A.1 Comprueba que la complementación satisface:

- $\bar{\bar{A}} = A$
- $\bar{\emptyset} = U, \bar{U} = \emptyset$
- $A \bar{\cup} B = \bar{A} \cap \bar{B}, A \bar{\cap} B = \bar{A} \cup \bar{B}$ (leyes de De Morgan)
- $A \subset B \iff \bar{B} \subset \bar{A}$
- $A \cup \bar{A} = U, A \cap \bar{A} = \emptyset$

A.2 Comprueba que el producto cartesiano de conjuntos satisface:

- $(C \subset A) \wedge (D \subset B) \Rightarrow (C \times D) \subset (A \times B)$
- $A \times (B \cup C) = (A \times B) \cup (A \times C)$
- $A \times (B \cap C) = (A \times B) \cap (A \times C)$

A.3 Comprueba que $A = B \iff A \oplus B = \emptyset$

A.1 Correspondencias y relaciones

Dados dos conjuntos A y B , se llama *correspondencia* f de A en B ($f : A \longrightarrow B$) a un subconjunto dado F del producto cartesiano $A \times B$, y se denota:

$$f(x) = \{y \in B : (x, y) \in F \subset A \times B\}$$

Al conjunto A se le llama *conjunto inicial* ($A = \text{in}(f)$) y al B *conjunto final* ($B = \text{fin}(f)$). También se definen:

- $\text{or}(f) = \{x \in A : (x, y) \in F \text{ para algún } y \in B\}$ (*conjunto original*).
- $\text{im}(f) = \{y \in B : (x, y) \in F \text{ para algún } x \in A\} = \bigcup_{x \in A} f(x)$ (*conjunto imagen*).

Una correspondencia se denomina:

- *unívoca* $\iff \forall x \in A : f(x)$ contiene como máximo un elemento
- *aplicación* $\iff \forall x \in A : f(x)$ contiene un único elemento³. Si y es el único elemento de $f(x)$, escribiremos $y = f(x)$.

A su vez, una aplicación puede calificarse como:

- *inyectiva* $\iff \forall x, y \in A : f(x) = f(y) \implies x = y$
- *suprayectiva* $\iff \text{im}(f) = \text{fin}(f)$
- *biyección* $\iff f$ es aplicación inyectiva y suprayectiva. Una biyección establece un “emparejamiento” de cada elemento del conjunto inicial con uno del final y viceversa, y se denota $f : A \longleftrightarrow B$

Una *operación interna* en A es una aplicación $c : A \times A \rightarrow A$, de forma que $z = c(x, y) \in A$. Si c es asociativa se dice que A es un *semigrupo*. Si además existe un elemento neutro, entonces A es un *monoide*.

- *homomorfismo* $\iff f : A \rightarrow B$ es una aplicación entre semigrupos o monoídes y conserva las operaciones internas c, c' definidas en los conjuntos inicial y final respectivamente

$$\forall x, y \in A : f(c(x, y)) = c'(f(x), f(y))$$

Una correspondencia de A en A recibe el nombre de *relación*:

$$xRy \iff (x, y) \in R \subset A \times A$$

Una *relación* es *de equivalencia* si es:

- reflexiva: $xRx \quad \forall x \in A$
- simétrica: $xRy \implies yRx, \quad x, y \in A$
- transitiva: $xRy \wedge yRz \implies xRz, \quad x, y, z \in A$

Una *relación de orden* tiene las propiedades:

- reflexiva
- antisimétrica: $xRy \wedge yRx \implies x = y \quad x, y \in A$.
- transitiva

³Alternativamente, f es unívoca y $\text{or}(f) = \text{in}(f)$

110 APÉNDICE A. NOCIONES BÁSICAS DE TEORÍA DE CONJUNTOS

Dada una relación R de equivalencia en A , se definen sus *clases de equivalencia* como:

$$C(x) = \{y \in A : xRy\}$$

Una *partición* de A es una colección de subconjuntos $\{C_i\}$ tales que:

- $C_i \neq \emptyset$
- $\bigcup_i C_i = A$
- $C_i \cap C_j = \emptyset \iff i \neq j$

Dada R de orden en A , se denota:

- $a \leq b \iff (a, b) \in R \subset A \times A$
- $a < b \iff a \leq b \wedge a \neq b$
- $[a, b] = \{x \in A : a \leq x \leq b\}$

Se dice que $C \subset A$ es *convexo* $\iff \forall x, y \in C : [x, y] \subset C$

Se define \hat{C} , *clausura* o *cierre convexo* de C , como el menor conjunto convexo que contiene a C . El concepto de clausura es un concepto bastante general, y siempre se entiende como una extensión del conjunto de partida, hasta que este adquiere una cierta propiedad. Aquí hemos hablado de la clausura convexa (menor conjunto convexo), pero de forma análoga se definen la clausura reflexiva-transitiva de una relación (menor relación \hat{R} que incluye a R y que posee dichas propiedades) o la clausura de Kleene de un lenguaje (menor lenguaje en el que la concatenación de dos cadenas es otra cadena del lenguaje), etc.

Ejercicios

A.4 Prueba que el elemento neutro de un monoide es único.

A.5 Demuestra que dado un homomorfismo entre monoïdes $f : A \longrightarrow B$ entonces $f(\lambda) = \epsilon$, siendo λ y ϵ los elementos neutros de A y B respectivamente.

A.6 Comprueba que la definición de una clase de equivalencia $C(x)$ no depende del elemento representante x elegido.

A.7 Demuestra que una relación de equivalencia en A define una partición en A y viceversa.

A.8 Comprueba que la inclusión de conjuntos es una relación de orden.

A.9 Demuestra las propiedades de la clausura o cierre convexo:

- $C \subset \widehat{C}$
- C convexo $\iff C = \widehat{C}$
- $\widehat{\widehat{C}} = \widehat{C}$
- A, B convexos $\implies A \cap B$ convexo
- $\widehat{A} \cup \widehat{B} \subset \widehat{A \cup B}$
- $\widehat{A} \cap \widehat{B} \supset \widehat{A \cap B}$

A.2 Cardinal. Conjuntos infinitos

La noción intuitiva de cardinal de un conjunto (el número de elementos que contiene) no está exenta de problemas, sobre todo cuando se considera la existencia de conjuntos infinitos⁴. Por otro lado, en la moderna teoría matemática este concepto resulta clave para poder introducir de forma rigurosa los números naturales. Por ello se debe definir de forma precisa lo que se entiende por cardinal o potencia de un conjunto.

Diremos que dos conjuntos A y B son *equipotentes* ($A \simeq B$) si existe una biyección entre ellos $f : A \longleftrightarrow B$. Esta definición genera una partición entre los conjuntos, ya que la relación que se establece es de equivalencia.

Si $A \simeq B$ diremos que A y B tienen el mismo cardinal, y escribiremos: $\text{card}(A) = \text{card}(B)$.

Interesa definir una relación de orden entre los cardinales. Ello puede hacerse de la siguiente forma:

$$\text{card}(A) \leq \text{card}(B) \iff A \simeq C \subset B$$

Las propiedades reflexiva y transitiva de esta relación son inmediatas. La propiedad antisimétrica requiere una demostración más elaborada (véase [4]).

Los números naturales se introducen como símbolos para denotar cardinales, y se definen de forma constructiva, a partir de la siguiente sucesión de conjuntos:

$$\begin{aligned} C_0 &= \emptyset \\ C_1 &= \{\emptyset\} \\ C_2 &= \{\emptyset, \{\emptyset\}\} \\ C_3 &= \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\ &\vdots \end{aligned}$$

⁴Recuérdese, por ejemplo, la famosa *paradoja de Galileo*.

Cada conjunto se define como reunión de todos los objetos (en este caso conjuntos) previamente definidos. De esta forma, los números naturales no son más que etiquetas para denotar los cardinales de los conjuntos de la sucesión anterior. Por ejemplo, $3 = \text{card}(C_3)$. Si un conjunto A es equipotente con alguno de los C_n anteriores, entonces su cardinal es el número natural $n \in \mathbb{N}$ y se dice que A es *finito*.

Un conjunto *infinito* no puede ponerse en biyección con ninguno de los C_n anteriormente definidos, y posee además la sorprendente propiedad (que suele tomarse como definición) de que puede ponerse en biyección con una parte propia de sí mismo. Por ejemplo:

$$f(n) = 2n, f : \mathbb{N} \longleftrightarrow \{n \in \mathbb{N}, n \text{ es par}\}$$

$$f(x) = \frac{x}{1-x^2}, f : (-1, 1) \longleftrightarrow \mathbb{R}$$

Otra propiedad importante de los conjuntos infinitos es que no todos tienen el mismo cardinal, sino que existe una sucesión infinita de potencias posibles diferentes. Esto es consecuencia del teorema de Cantor, que afirma que el cardinal del conjunto de las partes $P(A)$ es estrictamente mayor que el cardinal del conjunto de partida A . Por ello, siempre es posible construir conjuntos de mayor cardinalidad que cualquiera dada:

$$\text{card}(A) < \text{card}(P(A)) < \text{card}(P(P(A))) < \dots$$

El menor cardinal infinito es el del conjunto \mathbb{N} de los números naturales. A todos los conjuntos equipotentes a \mathbb{N} se les denomina infinitos numerables (ya que sus elementos se pueden contar). Por tanto, A se dice *numerable* si $A \simeq \mathbb{N}$. El siguiente cardinal conocido es el de \mathbb{R} , conjunto de los números reales. Si $A \simeq \mathbb{R}$ se dice que A tiene la *potencia del continuo* (por ser los números reales representables como puntos sobre una recta, la recta real).

La cuestión de si existe algún conjunto cuyo cardinal c sea tal que $\text{card}(\mathbb{N}) < c < \text{card}(\mathbb{R})$ es un clásico ejemplo de teorema indecidible dentro de la aritmética. La existencia de proposiciones indecidibles en cualquier sistema axiomático que incluya la aritmética fue demostrada por Gödel en 1931 y constituye el célebre *teorema de incompletitud de Gödel*[5]. En este caso, es posible tomar como axioma una u otra de las opciones (es decir, que existe c o que no existe), lo que conduce a las llamadas teorías cantorianas o a las no-cantorianas respectivamente (algo semejante a lo que ocurrió con el desarrollo de las geometrías no-euclidianas, que rechazaban el llamado quinto postulado de Euclides).

Ejercicios

Demuestra las siguientes propiedades:

A.10 *La relación de equipotencia es una relación de equivalencia.*

A.11 A, B numerables $\Rightarrow A \cup B$ es numerable.

A.12 A, B numerables $\Rightarrow A \times B$ es numerable.

A.13 El conjunto de los números racionales \mathbb{Q} es numerable.

A.14 $\text{card}(A) = n \in \mathbb{N} \Rightarrow \text{card}(P(A)) = 2^n$.

A.15 $\text{card}(\mathbb{R}) = \text{card}(P(\mathbb{N}))$, y por tanto $\text{card}(\mathbb{R}) > \text{card}(\mathbb{N})$.

A.16 $\text{card}(\mathbb{C}) = \text{card}(\mathbb{R})$, siendo \mathbb{C} el conjunto de los números complejos.

A.2.1 Demostración del teorema de Cantor

El teorema afirma que $\text{card}(A) < \text{card}(P(A))$, por lo que hemos de demostrar:

- $\text{card}(A) \leq \text{card}(P(A))$
- $\text{card}(A) \neq \text{card}(P(A))$

Lo primero es trivial al tomar $f(a) = \{a\}$, biyección de A con un subconjunto de $P(A)$. La demostración de la segunda proposición se realiza por reducción al absurdo. Supongamos que existe una biyección $f : A \longleftrightarrow P(A)$. Se define entonces el conjunto siguiente:

$$D = \{e \in A : e \notin f(e)\} \subset A$$

Como f es una biyección, cada elemento de $P(A)$ tiene su original en A , en particular $D \in P(A) \Rightarrow \exists d \in A : D = f(d)$.

Es fácil comprobar que la existencia de este elemento d es contradictoria en sí misma, ya que

$$d \in D \iff d \notin f(d) \iff d \notin D !!$$

Por tanto la existencia de tal biyección debe ser rechazada.

Bibliografía

- [1] M.Gardner, *Paradojas*. Labor, Barcelona (1981)
- [2] J. de Burgos,*Curso de álgebra y geometría*. Alhambra, Madrid (1880)
- [3] J.J.Climent, V.Migallón, J.Penadés,*Matemática discreta. Teoría de conjuntos y análisis combinatorio*. Babel Editores, Alacant (1991)
- [4] Y.Ershov, *Lógica Matemática*. MIR, Moscú (1990)
- [5] Varios autores, *¿Qué es la lógica matemática?* Tecnos, Madrid (1983)
- [6] N.Chomsky, *El Lenguaje y el Entendimiento*. Seix Barral, Barcelona (1971).
- [7] N.Chomsky, *Reglas y Representaciones*. FCE, México.
- [8] A.V. Aho y J.D. Ullman *The Theory of Parsing, Translation, and Compiling - Volume 1: Parsing*. Prentice-Hall, Englewood Cliffs, NJ, EE.UU. (1972).
- [9] J.E.Hopcroft and J.D.Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley (1979)
- [10] J.G. Brookshear, *Teoría de la Computación*. Addison-Wesley Iberoamericana, Delaware (1993).
- [11] Natarajan, B.K. Machine learning: a theoretical approach. Morgan Kaufmann 1991, San Mateo CA(USA).
- [12] H.Hermes, *Enumerability, Decidability, Compatability*, Springer-Verlag, Berlin (Alemania)(1969).
- [13] J-L.A. Van de Snepscheut, *What computing is all about*. Springer-Verlag, New York, EUA (1993).
- [14] M. Alfonseca, J. Sancho, M. Martínez Orga, *Teoría de lenguajes, gramáticas y autómatas*. Universidad y Cultura, Madrid (1990).