TEORÍA DE LENGUAJES FORMALES

Una Introducción para Lingüistas

Sergio Balari
Universitat Autònoma de Barcelona
&
Centre de Lingüística Teòrica

2014

Índice general

Índice de figuras

Índice de cuadros

Prefacio

Desde finales de la década de 1950, la teoría de lenguajes formales se ha convertido en compañero de viaje permanente de la lingüística teórica. Compañero a veces imprescindible, otras molesto, otras invisible, no ha dejado, sin embargo, de estar siempre ahí. No en vano fue Noam Chomsky una de las principales figuras que impulsó su desarrollo, aunque también uno de los que ha mantenido una postura más ambigua sobre su verdadera relevancia y utilidad para la lingüística. Hoy en día, sin embargo, parece que su relevancia está fuera de toda duda, en un momento en que el concepto de computación ocupa un papel central en la lingüística y en las ciencias cognitivas.

No suele ser habitual, sin embargo, que la teoría de lenguajes formales tenga un espacio reservado en los cursos de lingüística que se imparten en nuestras universidades y todo aquel que quiera conocerla mejor siempre tiene que acudir a manuales o textos pensados para audiencias interesadas en las ciencias de la computación teóricas y, normalmente, con una base matemática que pocos (o ninguno) estudiantes de lingüística suelen tener.

Cuando Ángel Gallego (en buena medida el responsable de que este texto vea la luz) me propuso impartir un curso sobre lenguajes formales para los becarios de doctorado del Centre de Lingüística Teòrica de la Universitat Autònoma de Barcelona, acepté sin dudarlo. Casi de inmediato descubrí, sin embargo, que las fuentes a que podía recurrir para prepararlo no eran precisamente accesibles para cualquiera que no estuviera, por ejemplo, familiarizado con el desarrollo de una demostración matemática por inducción o por contradicción. Estaba claro, incluso, que al estudiante de lingüística poco le iban a interesar esas demostraciones, si su interés se centraba simplemente en tener unos conocimientos básicos de la teoría, los imprescindibles para comprender algunos resultados y aplicaciones que otros autores han hecho de ella. Para ello hacía falta hacer una presentación de la teoría libre de esas complejidades o, en todo caso, que las expusiera en formato «semidigerido». Así que me puse a escribir. Y me salió esto. Confío no haber digerido mal algunas cosas o, en todo caso, haber sido capaz de trasladarlas de una manera mínimamente comprensible.

El texto se estructura en cuatro partes, con una primera parte dedicada a nocio-

nes matemáticas básicas, una segunda dedicada a la teoría de lenguajes formales propiamente dicha y con una tercera parte, más breve, dedicada a complejidad computacional. Me ha parecido necesario introducir esta tercera parte, porque no parece hoy en día posible ocuparse de la complejidad de ciertos problemas apelando simplemente a las propiedades estructurales de estos. En mi opinión, la teoría de la complejidad computacional es el complemento necesario de toda investigación en este campo. Mi presentación es aun más informal si cabe y claramente incompleta, pero espero que así resulte accesible. La cuarta y última parte desarrolla algunas conclusiones y reflexiones finales. Gracias a Ángel por meterme en este lío, a Guillermo Lorenzo por leerlo y hacer algunas sugerencias útiles y a LaTeX por facilitarme la tarea de escribirlo.

Terrassa, invierno de 2014.

Parte I

Introducción

Capítulo 1

Nociones básicas

La teoría de los lenguajes formales estudia unas entidades matemáticas abstractas denominadas *lenguajes* que en ningún momento debemos confundir o equiparar con las lenguas naturales. Sin embargo, como veremos, los lenguajes formales pueden, en determinadas circunstancias, servirnos como modelos abstractos de determinadas propiedades de las lenguas naturales, de ahí la importancia de conocer los fundamentos de la teoría.

Un lenguaje formal (en adelante, simplemente «lenguaje») es un conjunto, finito o infinito, de cadenas definidas sobre un alfabeto finito.

Definición 1.1 (Alfabeto)**.** Un *alfabeto* es un conjunto finito de símbolos. El símbolo es un primitivo de la teoría de los lenguajes formales y para representarlos se suelen utilizar o bien las primeras letras del alfabeto latino o bien dígitos. Por tanto, cualquiera de los conjuntos siguientes es un alfabeto:

$$\Sigma_1 = \{a, b, c\} \qquad \Sigma_2 = \{0, 1\}.$$

Definición 1.2 (Cadena)**.** Una *cadena* o *palabra* es una serie arbitrariamente larga de símbolos unidos por concatenación que representamos disponiendo los diferentes símbolos que la componen en el orden deseado; por ejemplo: *aaabbbccc*, es una cadena. Como la definición de cadena es recursiva, podemos referirnos, si es necesario, a una cadena completa o a una subcadena que forme parte de una cadena mayor con las letras finales del alfabeto u, v, w, x, y y z. Por tanto, *aaaw* o, simplemente, w también son cadenas. Denotamos la cadena vacía con el símbolo ε. La cadena vacía es el elemento de identidad de la operación de concatenación,

de tal modo que $\varepsilon \oplus aaabbb = aaabbb$. La operación de concatenación de cadenas no es conmutativa (pero sí asociativa); por tanto $aaabbb \oplus ab \neq ab \oplus aaabbb$.

Definición 1.3 (Lenguaje)**.** Un *lenguaje* es un conjunto finito o infinito de cadenas. Los conjuntos siguientes son, por tanto, lenguajes:

$$L_1 = \{\varepsilon, ab, aabb, aaabbb\} \qquad L_2 = \{001, 011, 111\}.$$

De la Definición 1.3 se sigue que existe un espacio o universo infinito de lenguajes (Figura 1.1). Sin embargo, no todos los lenguajes que pueblan este universo son del interés de la teoría de los lenguajes formales, cuyo objetivo es investigar si hay un orden dentro de ese universo y estudiar las propiedades de aquellos lenguajes que podríamos calificar de «interesantes».

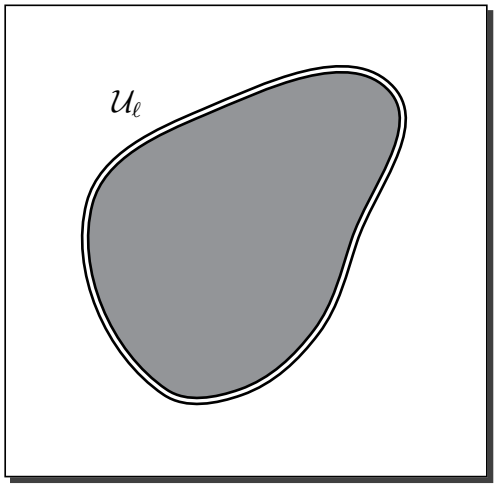


Figura 1.1: El universo $\mathcal{U}_\ell$ de los lenguajes formales.

Hay dos puntos de vista desde los cuales podemos determinar si un lenguaje es interesante o no. Por un lado, son interesantes aquellos lenguajes en los que se observa que se sigue alguna pauta regular en la construcción de las cadenas. Desde este punto de vista, el lenguaje L_3 es digno de estudio, pero no L_4:

$$L_3 = \{abc, aabbccc, aaabbbccc, ...\} \qquad L_4 = \{a, cab, bdac, ...\}$$

Para explicar el segundo punto de vista, tenemos que adelantarnos un poco a los acontecimientos, pero, de momento, basta con imaginar que disponemos de

un dispositivo mecánico capaz de determinar si una cadena dada pertenece o no a un lenguaje determinado. Este proceso, que denominaremos procedimiento de decisión, puede ejecutarse de manera más o menos eficiente, utilizando más o menos recursos, y no todos los lenguajes son iguales a este respecto.

Estos dos puntos de vista son los que definen los objetivos principales de las dos grandes disciplinas matemáticas ocupadas de poner orden en el universo de los lenguajes formales: la Teoría de los Lenguajes Formales (*sensu strictu*) y la Teoría de la Complejidad Computacional.

Definición 1.4 (Teoría de lenguajes formales)**.** La *Teoría de los lenguajes formales* estudia los lenguajes prestando atención únicamente a sus propiedades estructurales, definiendo clases de complejidad estructural y estableciendo relaciones entre las diferentes clases.

Definición 1.5 (Teoría de la complejidad computacional)**.** La *Teoría de la complejidad computacional* estudia los lenguajes prestando atención a los recursos que utilizaría un dispositivo mecánico para completar un procedimiento de decisión, definiendo así diferentes clases de complejidad computacional y las relaciones que existen entre ellas.

Es importante dejar claro desde el principio que los lenguajes que resultan interesantes desde el punto de vista de la Teoría de los lenguajes formales son los mismos que lo son para la Teoría de la complejidad. El motivo es claro, la condición *sine qua non* para que exista un procedimiento de decisión de un lenguaje dado es que las cadenas que lo componen sigan alguna pauta regular de construcción,[1] que es la que determina, en última instancia, su complejidad estructural. Por tanto, una y otra teoría centran su atención en una galaxia dentro del universo de los lenguajes (Figura 1.2), aunque la estructura que una y otra perciben en ella es distinta.

La existencia de una pauta regular para la construcción de las cadenas de un lenguaje es una propiedad fundamental, ya que, en la medida que seamos capaces de determinar esa pauta, podremos ofrecer una caracterización precisa de todo el lenguaje. Esto es particularmente relevante para la teoría, porque, como veremos más adelante, la inmensa mayoría de los lenguajes que tienen algún interés son

[1] Esta afirmación habrá que matizarla más adelante, pero de momento podemos asumir que es cierta.

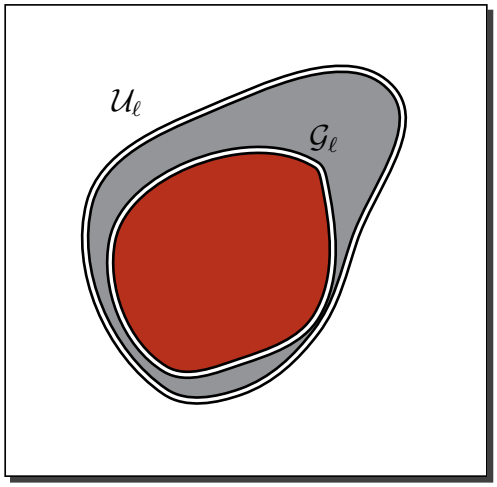


Figura 1.2: La galaxia $\mathcal{G}_\ell$ de los lenguajes formales interesantes.

infinitos, lo que nos impide caracterizarlos por simple enumeración como sería el caso de lenguajes finitos con un número relativamente pequeño de elementos. En el primer caso, y dada la existencia de una pauta, podremos describirlos utilizando lo que denominaremos *constructor de conjuntos*. Un constructor de conjuntos es una fórmula que nos permite derivar cualquier cadena que pertenezca al lenguaje que hemos descrito mediante el constructor. Antes de ver algunos ejemplos, tenemos que introducir algunas nociones auxiliares.

Definición 1.6. Sea $\mathbb{N}$ el conjunto de los números naturales. Entonces, dado un $n \in \mathbb{N}$, diremos que w^n es la cadena resultante de concatenar w consigo misma n veces. Por tanto, para cualquier cadena w,

$$\begin{aligned} w^0 &= \varepsilon \\ w^1 &= w \\ w^2 &= ww \\ w^3 &= www \end{aligned}$$

y, en general,

$$w^n = \underbrace{ww...w}_{n \text{ veces}}$$

A este respecto, conviene no olvidar que w puede ser una cadena que contenga un único símbolo, de tal modo que tanto a^5 como $(ab)^n$ son descripciones legales de cadenas. La única diferencia, claro está, es que la primera descripción denota una única cadena, mientras que la segunda denota un conjunto infinito de cadenas.

Definición 1.7 (Clausura transitiva o de Kleene)**.** La *clausura transitiva* o, también, *clausura de Kleene* de un alfabeto Σ, escrito Σ^*, es el conjunto de todas las cadenas sobre Σ. Por tanto:

$$\{a\}^* = \{\varepsilon, a, aa, aaa, ...\}$$
$$\{a, b\}^* = \{\varepsilon, a, b, aa, bb, ab, ba, aaa, ...\}$$

Con estas herramientas ya podemos especificar algunos lenguajes utilizando constructores de conjuntos:

$$L_1 = \{x \in \{a, b\}^* \mid |x| \leq 2\}$$
$$L_2 = \{xy \mid x \in \{a, aa\} \text{ e } y \in \{b, bb\}\}$$
$$L_3 = \{a^n b^n \mid n \geq 1\}$$
$$L_4 = \{(ab)^n \mid n \geq 1\}$$

Como vemos, un constructor de conjuntos es una fórmula donde se especifica cómo debemos concatenar los símbolos del alfabeto, más alguna restricción que el procedimiento de concatenación debe obedecer. Así, por ejemplo, L_1, a pesar de definirse el procedimiento como la clausura transitiva sobre el alfabeto $\{a, b\}$, al imponerse la restricción de que la longitud de las cadenas resultantes no puede ser superior a 2, es un lenguaje finito. L_4, en cambio, es un lenguaje infinito sin la cadena vacía, ya que n puede ser cualquier número natural igual o mayor que 1.

Parte II

Complejidad estructural

Capítulo 2

Introducción

Desde finales de los años cincuenta y principios de los sesenta del siglo pasado, gracias sobre todo a los trabajos de Noam Chomsky, se conoce con cierta precisión la estructura del espacio de lenguajes formales definido por la galaxia $\mathcal{G}_\ell$. En la Figura 2.1 tenemos la Jerarquía de Chomsky que, como vemos, define una serie de relaciones de inclusión entre diferentes clases de lenguajes. Dichas relaciones de inclusión se definen en función de una complejidad estructural creciente, siendo T3 la clase que contiene los lenguajes más simples y $\mathrm{T0}_{re}$ la que contiene los más complejos; en la leyenda de la figura aparecen los nombres con los que Chomsky bautizó a los lenguajes incluidos en cada clase. Con el tiempo, la Jerarquía ha ido ganando en detalle, a medida que se iban descubriendo nuevas clases de lenguajes que definían con mayor precisión el espacio delimitado por las clases originalmente propuestas por Chomsky, pero sin modificar radicalmente su estructura original. De especial interés para nosotros aquí será toda una colección de lenguajes que definen una o más subclases dentro del espacio T1. Estos lenguajes comparten una serie de propiedades que los hacen *moderadamente sensibles al contexto* y que resultan de especial interés para el estudio de las lenguas naturales. Sin embargo, para comprender mejor en qué consiste la sensibilidad al contexto moderada, primero tendremos que estudiar con detalle aquellos lenguajes que se hallan en la zona baja de la Jerarquía.

Uno de los principales obstáculos a los que se enfrenta cualquiera que desee conocer la complejidad estructural de un lenguaje es el hecho de que esta difícilmente puede deducirse a partir de la simple observación de las cadenas que lo componen. Tampoco resultan particularmente informativos los constructores de conjuntos que podamos utilizar para definir los lenguajes, ya que estos quizá nos permitan hacernos una idea intuitiva de su complejidad, pero en ningún caso podremos establecerla de forma precisa; peor aún, con las herramientas de que disponemos en este momento, nos es totalmente imposible determinar si dos lenguajes L_i y L_j tienen la misma complejidad estructural o no.

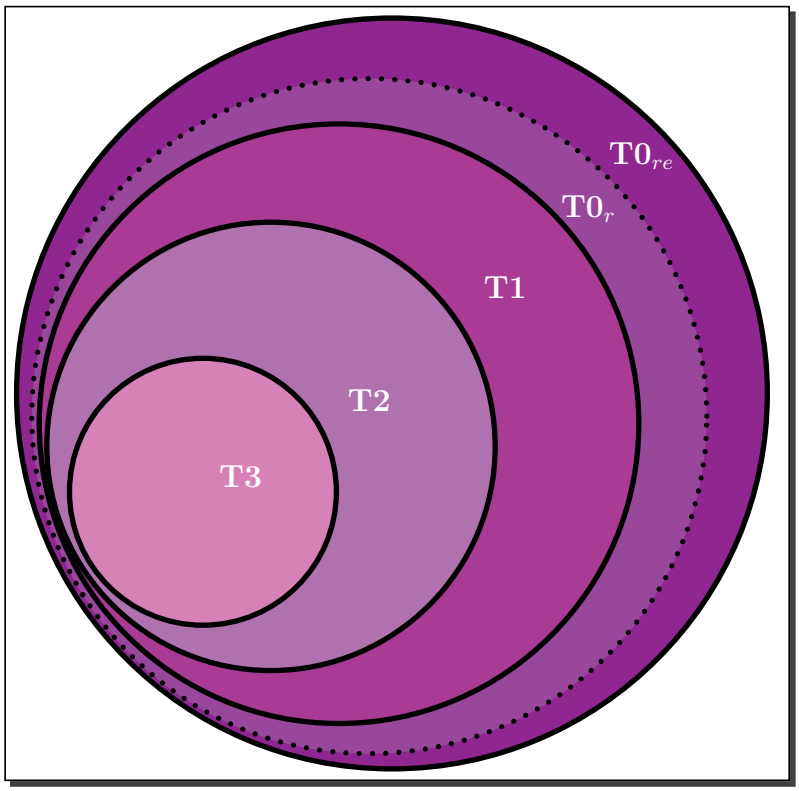


Figura 2.1: La Jerarquía de Chomsky. T3 = Lenguajes regulares. T2 = Lenguajes independientes del contexto. T1 = Lenguajes sensibles al contexto. T0$_r$ = Lenguajes recursivos. T0$_{re}$ = Lenguajes recursivamente enumerables.

Uno de los principales hallazgos de Chomsky fue la demostración de que podemos construir modelos matemáticos cuyas propiedades son un reflejo directo del grado de complejidad estructural de los lenguajes que se ajustan a dichos modelos. Por tanto, en la medida que dos lenguajes pueden caracterizarse a través de un modelo matemático del mismo tipo, podremos decir que tienen la misma complejidad estructural y que pertenecen a la misma clase; y viceversa, claro. Los modelos matemáticos mediante los cuales podemos caracterizar lenguajes son dos: los *autómatas* y las *gramáticas*. Ambos, aunque parten de principios distintos, se puede demostrar que son equivalentes. Por tanto, si un lenguaje se puede caracterizar mediante un autómata propio de una determinada clase de complejidad, existe un procedimiento automático para derivar su caracterización mediante la gramática correspondiente, y viceversa.

La mejor manera de comprender qué son exactamente los autómatas y las gramáticas es ver ejemplos de los unos y de las otras, pero, aunque sea de forma intuitiva, podemos hacer una caracterización general de cada tipo de modelo. Para un lingüista, no resulta muy difícil comprender qué es una gramática, ya que las gra-

máticas son conjuntos finitos de reglas que especifican de forma exhaustiva todas y cada una de las cadenas que pertenecen al lenguaje. En este caso, la complejidad estructural del lenguaje estará en función de ciertas restricciones que impondremos sobre el formato de las reglas. Los autómatas, por su parte, son dispositivos dinámicos, máquinas abstractas, que, siguiendo una serie de instrucciones, son capaces de decidir si una cadena determinada pertenece a un lenguaje dado o no. Como vemos, los autómatas son precisamente aquellos dispositivos mecánicos capaces de completar un procedimiento de decisión a los que hacíamos referencia en relación con la Teoría de la complejidad computacional. Esta es la manera más común de concebir un autómata, pero, como veremos, existen otras. Lo importante, por el momento, es que la complejidad estructural de un lenguaje desde el punto de vista de un autómata vendrá determinada a partir de determinadas propiedades estructurales o capacidades del propio autómata.

Llegados a este punto, resulta ya inevitable ocuparse con detalle de las diferentes clases que componen la Jerarquía de Chomsky y de las propiedades características de los autómatas y las gramáticas asociados a cada clase.

Capítulo 3

Lenguajes regulares

La clase de los lenguajes regulares contiene los lenguajes estructuralmente más simples dentro de la Jerarquía de Chomsky. A día de hoy, se han descrito lenguajes todavía más simples de los descritos originalmente por Chomsky, los que conforman lo que se conoce como *Jerarquía Subregular* y que no alcanzan ni siquiera el nivel de complejidad de los lenguajes de los que nos ocuparemos aquí. Aunque no es totalmente irrelevante para la lingüística, en este texto dejaremos de lado la Jerarquía Subregular.

Los lenguajes regulares propiamente dichos, en cambio, sí tienen interés lingüístico, ya que, como veremos más adelante, ciertas propiedades de la fonología y quizá la morfología de las lenguas naturales son modelables con lenguajes de esta complejidad estructural. De forma intuitiva, un lenguaje regular puede caracterizarse como aquel cuyas cadenas contienen dependencias lineales, de tal modo que la presencia de un símbolo u otro en una posición determinada de la cadena depende exclusivamente del símbolo que lo precede inmediatamente.[1]

3.1. Autómatas de estados finitos

El Autómata de estados finitos (AEF) es el modelo computacional asociado a los sistemas regulares. Es un modelo computacional muy simple que podemos imaginar como una unidad de control asociada a un cabezal de lectura, el cual, a su vez, está conectado a una cinta dividida en celdillas. En cada celdilla de la cinta hay un símbolo de la cadena que queremos analizar, más el símbolo «#» que indica el final de la cadena. El cabezal, que solo puede desplazarse hacia la derecha, va leyendo esos símbolos sucesivamente hasta que se detiene definitivamente; en

[1] Lo cual explica por qué la sintaxis de las lenguas naturales no es regular, ya que en sintaxis las dependencias no siempre son lineales en este sentido.

la Figura 3.1 tenemos una representación de cómo podría ser un AEF, donde la flecha doble indica la dirección en que se mueve el cabezal.

La unidad de control, en función del símbolo que haya leído el cabezal, entrará en un nuevo estado de los estados en los que se puede hallar o permanecerá en el que está (q_0 a q_3 en la Figura 3.1). Dentro del conjunto de estados, destacamos el estado inicial y el final, que son estados especiales, en el sentido de que el primero es aquel en el que se halla la unidad de control en el momento de iniciar la computación, mientras que el estado final es aquel que esta debe alcanzar al terminarla. Un autómata de estados finitos puede tener más de un estado final, pero el número total de estados, incluidos el inicial y los finales, siempre debe ser finito. Si, cuando el cabezal ha leído todos los símbolos de la cinta, la unidad de control no se halla en un estado final o si la unidad de control se detiene antes de que el cabezal haya leído todos los símbolos de la cinta, diremos que el autómata no ha aceptado la cadena y, por tanto, que esta no pertenece al lenguaje que el autómata es capaz de aceptar. En cambio, si el autómata se detiene en un estado final una vez leídos todos los símbolos de la cadena, entonces este la habrá aceptado y pertenecerá al lenguaje en cuestión.

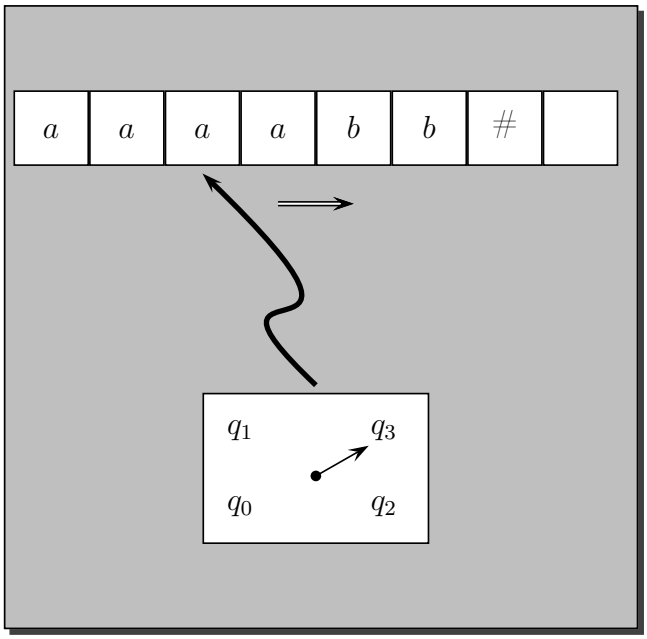


Figura 3.1: Un autómata de estados finitos.

Antes de presentar la definición formal de un AEF, podemos ofrecer una caracterización intuitiva de su complejidad estructural. Como hemos visto, este consta de una unidad de control que puede hallarse en un número finito de estados, un cabezal de lectura capaz de desplazarse únicamente de izquierda a derecha y de una cinta en la que se introducen los símbolos de la cadena que queremos recono-

cer. Nada más. Como veremos, a partir de este modelo básico, podemos introducir pequeñas mejoras, como permitir que el cabezal pueda desplazarse en ambas direcciones, que pueda no solo leer, sino también borrar o escribir símbolos en la cinta, que la unidad de control pueda guardar símbolos leídos en una pila de memoria, etc. Algunas de estas mejoras harán que el autómata resultante sea capaz de reconocer lenguajes estructuralmente más complejos. Lo importante, por ahora, es que cualquier autómata construido de acuerdo con estas especificaciones solo será capaz de reconocer lenguajes regulares. Otro punto a tener en cuenta es que, si bien es posible construir más de un autómata para un mismo lenguaje, un mismo autómata solo es capaz de reconocer un único lenguaje, ya que su capacidad para hacerlo depende de cómo se hayan definido los diferentes estados y las posibles transiciones entre ellos, que hemos de suponer que se hallan directamente especificados en la unidad de control. Vamos, pues, a la definición formal.

Definición 3.1 (Autómata de Estados Finitos)**.** Un *autómata de estados finitos* es una quíntupla $(Q, \Sigma, \delta, q_0, F)$, tal que Q es un conjunto finito de *estados*, Σ es un *alfabeto de entrada*, $q_0 \in Q$ es el *estado inicial*, $F \subseteq Q$ es el conjunto de *estados finales* y δ es la *función de transición* que proyecta $Q \times \Sigma$ en Q. Es decir, $\delta(q, a)$ es un estado para cada estado q y cada símbolo de entrada a.

Como puede verse, aquí todo el trabajo lo hace la función δ, que es la que debemos asumir que implementa directamente la unidad de control del autómata. Esta función pone en relación pares ordenados de estados y símbolos obtenidos a partir del producto cartesiano del conjunto Q de estados y el conjunto Σ de símbolos del alfabeto con estados del conjunto Q, de tal modo que $\delta(q_i, a) = q_j$, por ejemplo, debe leerse: si me hallo en el estado q_i y estoy leyendo el símbolo a, entonces debo desplazarme hacia la derecha y entrar en el estado q_j. Es decir, la función δ define de forma exhaustiva la conducta de la unidad de control en función del símbolo que el cabezal esté leyendo en ese momento.

Con esto, ya podemos construir un autómata simple, capaz de reconocer, por ejemplo, el lenguaje regular $L_1 = \{a^*b^*\}$. Nótese que L_1 contiene cualquier cadena que contenga una secuencia de cero o más as, seguida de una secuencia de cero o más bs, lo que incluye, evidentemente, la cadena vacía. Por tanto, la única restricción que deben obedecer las cadenas de este lenguaje es que si hay as y bs, las segundas deben aparecer todas siempre después de las primeras. Estos son, pues, algunos de los elementos del conjunto L_1:

$$L_1 = \{\varepsilon, a, b, aa, bb, ab, aabb, aab, abb, ...\}.$$

En el ejemplo 3.1.1 tenemos la construcción del autómata que reconoce este lenguaje.

Ejemplo 3.1.1. Dado que q_0 designa siempre el estado inicial, los diferentes elementos de la quíntupla seran

$$Q = \{q_0, q_1\}.$$
$$\Sigma = \{a, b\}.$$
$$F = \{q_0, q_1\}.$$

La función δ la definimos haciendo una tabla de transiciones, como la que tenemos en el cuadro 3.1, que relacione pares de estados y símbolos con estados.

	a	b
q_0	q_0	q_1
q_1	—	q_1

Cuadro 3.1: La función δ de un autómata para el lenguaje L_1.

Comentemos brevemente la tabla del cuadro 3.1 para comprender mejor cómo se comporta exactamente el autómata. Hay tres condiciones previas sobre las situaciones en que el autómata debe detenerse que es importante tener en cuenta antes de interpretar la tabla. En primer lugar, si el primer símbolo que lee el autómata es la cadena vacía ε, se detendrá en este punto, en el supuesto de que al autómata «nunca se le hacen trampas», es decir, que el primer símbolo de la cadena siempre se halla en la primera casilla de la cinta y no en cualquier otro lugar. El motivo es simple, cualquier cadena puede ir precedida por un número indeterminado de cadenas vacías, de tal modo que, si permitiéramos este tipo de situaciones, el autómata debería ir recorriendo la cinta durante un tiempo indefinido hasta localizar la cadena que debe computar. En segundo lugar, el autómata se detendrá siempre cuando el último símbolo que lee es la cadena vacía. Este supuesto también es lógico: si después de leer un símbolo, el autómata se desplaza hacia la derecha y lee la cadena vacía, debemos asumir que ha llegado al final y que ya no le quedan más símbolos por leer. Con este supuesto, nos ahorramos la necesidad de incluir el símbolo especial de final de cadena y de definir transiciones especiales para cuando el autómata se tope con la cadena vacía. Finalmente, el autómata se detendrá, se halle donde se halle, si no hay una transición definida para la configuración en la que se encuentra en ese momento. Cuando se da este caso y el autómata no ha llegado al final de la cadena, diremos que esta no ha sido aceptada. Veamos ahora cómo se traducen estos tres supuestos en el momento de interpretar la tabla. Al

inicio de la computación, cuando el autómata se halla en el estado inicial q_0, hay tres configuraciones posibles, a saber: (a) que la primera casilla esté vacía; (b) que en la primera casilla haya una a; y (c) que en la primera casilla haya una b. En el primer caso, el autómata se detendrá sin cambiar de estado; como q_0 es también un estado final, habrá aceptado la cadena, la cadena ε en este caso, que es un elemento legítimo del lenguaje L_1. En el segundo caso, el autómata se desplazará hacia la derecha, pero sin cambiar de estado y lo que ocurra luego dependerá de lo que haya en la segunda casilla. En el tercer caso, el autómata se desplazará hacia la derecha y pasará al estado q_1. Consideremos ahora el contenido de la segunda casilla. Fuese cual fuese la configuración anterior, si el autómata lee ε permanecerá en el estado en que se encontraba y se detendrá. Como tanto q_0 como q_1 son estados finales, la cadena (a o b) será una cadena legítima del lenguaje. Si en la segunda casilla hay una a y el autómata se halla en q_0, se desplazará hacia la derecha y permanecerá en el mismo estado; si, en cambio, se halla en q_1 se detendrá inmediatamente, ya que no hay transición definida para esta configuración y, dado que el cabezal no está leyendo una casilla vacía, diremos que la cadena, ba, no ha sido aceptada, lo cual es correcto. Si en la segunda casilla hay una b y el estado es q_0, el autómata se desplazará a la derecha y pasará a q_1; si ya se hallaba en q_1, se desplazará pero sin cambiar de estado. Como vemos, hay dos transiciones de la función δ, $\delta(q_0, a) = q_0$ y $\delta(q_1, b) = q_1$, que garantizan que el autómata siga operando mientras haya as o bs en la cinta. La transición $\delta(q_0, b) = q_1$ es la que garantiza que las subcadenas de bs siempre vayan solas o precedidas por secuencias de as.

Una manera gráfica de visualizar mejor el funcionamiento de un autómata consiste en dibujar un diagrama de transiciones como el que tenemos en la Figura 3.2. En el diagrama, los estados se representan mediante círculos; los estados finales aparecen como círculos dobles y el estado inicial con una flecha que apunta hacia él. Las transiciones se representan mediante flechas que conectan estados y van etiquetadas con un símbolo; como se aprecia en el dibujo, las transiciones pueden conectar un estado consigo mismo. Nótese que el diagrama recoge todos los casos de la función δ del cuadro 3.1: el autómata permanecerá en el estado q_0 mientras vaya leyendo as, pasará a q_1 tan pronto como halle una b y permanecerá en q_1 mientras vaya leyendo bs; como la lectura de la cadena vacía es una de las condiciones para que el autómata se detenga se encuentre en el estado en que se encuentre, no es preciso definir transiciones para esta situación.

Evidentemente, el diseño del autómata de 3.2 no es el único posible, aunque sí el más compacto, para una máquina capaz de reconocer el lenguaje L_1. Por ejemplo, podríamos haber fijado q_0 como único estado final, en cuyo caso deberíamos definir una transición $\delta(q_1, \varepsilon) = q_0$ para asegurarnos de que el autómata se halle en un estado final al llegar al final de la cadena. Y, como esta, existen muchas otras variantes, ya con un número mayor de estados (el mínimo es dos), pero todas serán equivalentes; la única restricción es que el número de estados sea finito.

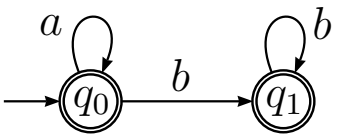


Figura 3.2: Diagrama de transiciones del autómata del ejemplo 3.1.1.

Veamos ahora un nuevo ejemplo que, con el que acabamos de tratar, nos permitirá hacer explícitas algunas propiedades relevantes de los lenguajes regulares. En este caso, tenemos el lenguaje $L_2 = \{x \in \{a, b\}^*\}$, es decir, la clausura de Kleene del alfabeto $\Sigma = \{a, b\}$. El lenguaje es, por tanto, infinito. Estos son algunos de los elementos que pertenecen al conjunto:

$$L_2 = \{\varepsilon, a, b, aa, bb, ab, ba, aaa, ...\}.$$

Ahora, y a diferencia de L_1, en las cadenas de este lenguaje el orden relativo de las secuencias de as y bs no está fijado, cualquier secuencia as o bs o de uno y otro símbolo en cualquier orden es válida. En el ejemplo 3.1.2 definimos su autómata.

Ejemplo 3.1.2. Dado que q_0 designa siempre el estado inicial, los diferentes elementos de la quíntupla seran

$$Q = \{q_0, q_1\}.$$
$$\Sigma = \{a, b\}.$$
$$F = \{q_0, q_1\}.$$

La función δ también la definimos haciendo una tabla de transiciones que relacione pares de estados y símbolos con estados; cuadro 3.2.

	a	b
q_0	q_0	q_1
q_1	q_0	q_1

Cuadro 3.2: La función δ de un autómata para el lenguaje L_2.

Nótese que, como en este caso sí es posible que después de una secuencia de bs haya una secuencia de as, tenemos la transición $\delta(q_1, a) = q_0$ que se ocupa precisamente de esta posibilidad. En la Figura 3.3 tenemos el diagrama de transiciones correspondiente.

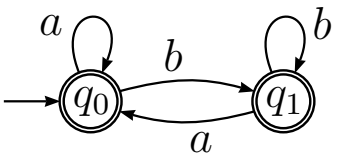


Figura 3.3: Diagrama de transiciones del autómata del ejemplo 3.1.2.

Los diagramas de transiciones de las Figuras 3.2 y 3.3 nos permiten hacer explícita, aunque de manera intuitiva, una propiedad muy importante de todos los lenguajes regulares, que no se observa en ningún otro tipo de lenguaje de mayor complejidad estructural. Todos los lenguajes regulares tienen subestructuras cíclicas o, dicho de otro modo, dado que el número de estados es siempre finito y los lenguajes pueden ser infinitos, la única manera de reconciliar una cosa con la otra es que el autómata regrese a un estado en el que ya estuvo anteriormente en un momento posterior de la computación. Esta propiedad se cumple de forma trivial en el autómata de la Figura 3.2, ya que los bucles que apuntan hacia los estados q_0 y q_1, respectivamente, representan claramente los dos ciclos que definen el lenguaje en cuestión. El caso de la Figura 3.3 es más sutil, pero igual de transparente, ya que la transición $\delta(q_1, a) = q_0$ permite la posibilidad de que el ciclo de las as vuelva a repetirse una vez que se ha completado un ciclo de bs que, a su vez (por $\delta(q_0, b) = q_1$), puede volver a repetirse y, así, sucesivamente. Esta propiedad se traduce, en última instancia, en el hecho de que en cualquier cadena arbitrariamente larga que pertenezca a un lenguaje regular L_k, podemos siempre identificar una cadena w de una longitud dada que es una cadena de L_k. Este es, de hecho, el «secreto» de la regularidad: la posibilidad de definir un patrón básico para las cadenas que puede repetirse un número indeterminado de veces dentro de las mismas cadenas.[2] Como veremos, esto se debe al hecho de que cualquier lenguaje regular puede entenderse como el producto de combinar dos o más lenguajes regulares a través de alguna operación de composición específica. De este punto nos ocuparemos más adelante con detalle, antes debemos conocer a una clase ligeramente distinta de autómatas de estados finitos. En este caso, empezaremos por su definición formal.

Definición 3.2 (Autómata de Estados Finitos Nodeterminista)**.** Un *autómata de estados finitos nodeterminista* (AEFN) es una quíntupla $(Q, \Sigma, \delta, q_0, F)$, donde Q, Σ, q_0 y F se definen del mismo modo que para un AEF, pero δ es una función que proyecta $Q \times \Sigma$ en 2^Q, es decir, en el *conjunto potencia* de Q (el conjunto

[2]Esta viene a ser la caracterización informal de lo que se conoce como *Lema del Bombeo*, cuya definición formal obviaremos aquí, y que es una herramienta muy potente para detectar lenguajes que no son regulares, ya que la propiedad que esbozamos en el texto solo se da en los lenguajes regulares.

de todos los subconjuntos de Q). Por tanto, ahora $\delta(q, a)$ designa el *conjunto* de estados p tal que existe una transición con la etiqueta a de q a p.

De la comparación entre la definición 3.1 del AEF determinista (en adelante AEFD) y la definición del AEFN se desprende que el segundo, a diferencia del primero, cuando se halla en un estado específico leyendo un símbolo dado, tiene la opción de elegir entre dos o más estados en los que entrar dada la misma configuración. Como debemos entender que la elección es «libre» o «aleatoria», de ahí la calificación de nodeterminista. En este punto, conviene ya adelantar, dado que no lo vamos a demostrar formalmente, que dentro del ámbito estricto de los lenguajes regulares, la opción entre un modelo computacional nodeterminista frente a uno determinista no supone ningún incremento en la complejidad del sistema y, por tanto, ambos tipos de autómata son equivalentes. En áreas superiores de la Jerarquía de Chomsky, sin embargo, los modelos nodeterministas de un mismo tipo de autómata sí pueden definir espacios de mayor complejidad que los deterministas. Antes de ver algunos ejemplos de AEFN, introduciremos una pequeña precisión en la definición de 3.2 para permitir lo que denominaremos *transiciones espontáneas*.

Definición 3.3 (AEFN con transiciones espontáneas)**.** Un *AEFN con tansiciones espontáneas* es una quíntupla $(Q, \Sigma, \delta, q_0, F)$, donde Q, Σ, q_0 y F se definen del mismo modo que para un AEFN, pero δ es una función que proyecta $(Q \times (\Sigma \cup \{\epsilon\}))$ en 2^Q. Por tanto, ahora $\delta(q, a)$ designa el conjunto de estados p tal que existe una transición con la etiqueta a de q a p y $\delta(q, \epsilon)$ designa el conjunto de estados p' tal que existe una transición espontánea, es decir sin consumir un símbolo de entrada, de q a p'.

Nótese que las definiciones 3.2 y 3.3 son casi idénticas con la única diferencia de cómo se construye la función δ. En la primera, δ es una función que relaciona pares de estados y símbolos con conjuntos de estados, mientras que en la segunda relaciona pares de estados y símbolos o de estados y el elemento ϵ con conjuntos de estados. El elemento ϵ, que debemos distinguir de la cadena vacía ε, viene a ser un «no símbolo», ya que las transiciones espontáneas tienen lugar sin que se produzca el desplazamiento correspondiente del cabezal. Con esto, vamos ahora a construir un AEFN con transiciones espontáneas para lenguaje $L_1 = \{a^*b^*\}$.

Ejemplo 3.1.3. Dado que q_0 designa siempre el estado inicial, los diferentes ele-

mentos de la quíntupla seran

$$Q = \{q_0, q_1\}.$$
$$\Sigma = \{a, b\} \cup \{\epsilon\}.$$
$$F = \{q_0, q_1\}.$$

La función δ la definimos haciendo una tabla de transiciones, como la que tenemos en el cuadro 3.3, que relacione pares de estados y símbolos con conjuntos estados.

	a	b	ϵ
q_0	$\{q_0\}$	$\emptyset$	$\{q_1\}$
q_1	$\emptyset$	$\{q_1\}$	$\emptyset$

Cuadro 3.3: La función δ de un AEFN para el lenguaje L_1.

Como se ve, ahora las configuraciones del tipo $\delta(q, a)$ remiten a conjuntos de estados, aunque, en este caso, todas las configuraciones remiten a conjuntos que tienen un único elemento; no hay ninguna transición definida que remita al conjunto $\{q_0, q_1\} \subset 2^Q$. Las configuraciones que no están definidas remiten al conjunto vacío. El diagrama de transiciones de este autómata es el que tenemos en la Figura 3.4.

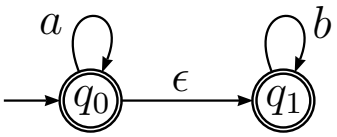


Figura 3.4: Diagrama de transiciones del AEFN del ejemplo 3.1.3.

El diagrama de la Figura 3.4 nos permite poner de relieve uno de los aspectos más interesantes (e intrigantes) del nodeterminismo. Veamos en primer lugar cómo se comporta el AEFD de la Figura 3.2 cuando tiene que procesar una cadena de L_1 como, por ejemplo, *aaabb*. El autómata empieza en el estado inicial q_0 e irá leyendo *a*s sin cambiar de estado hasta que se tope con la primera *b*, momento en el que pasará al estado q_1. En q_1 irá leyendo *b*s hasta llegar al final de la cadena y se detendrá. Como q_1 es un estado final, podremos decir que ha aceptado la cadena. Como vemos, un AEFD nos garantiza que si la cadena pertenece al lenguaje la aceptará siempre y, además, que hay un único camino que lleva del estado inicial al final. Veamos ahora qué ocurre con el AEFN de 3.4. Supongamos que tiene que procesar la misma cadena *aaabb*. Supongamos también que ya ha

iniciado el proceso y que se halla en el estado q_0 con el cabezal sobre la tercera a. En este punto, como en cualquier otro del proceso mientras se encuentre en q_0, tiene la opción de hacer una transición espontánea hacia q_1. Supongamos que «toma esa decisión». Después de la transición espontánea, el autómata se hallará en la siguiente configuración: $\delta(q_1, a)$, cuya imagen es $\emptyset$; es decir, no hay transición definida y el autómata se detendrá sin haber llegado al final de la cadena. Sabemos que la cadena sí pertenece al lenguaje (nos lo ha confirmado el AEFD) y, sin embargo, el AEFN no la ha aceptado. ¿Cómo hemos de intepretar esta situación? Bien, pues esta es la propiedad intrigante de los dispositivos nodeterministas, en general, no solo de los AEFN: solo podemos «fiarnos» de ellos cuando aceptan, pero nunca cuando no aceptan. El *quid* de la cuestión es que, a diferencia de los dispositivos deterministas que garantizan la existencia de un único camino hacia la aceptación, los nodeterministas solo nos garantizan que haya *algún* camino hacia la aceptación, no necesariamente todos los posibles. Por tanto, y dado el elemento de aleatoriedad que incorpora la posibilidad de que haya más de una transición a partir de una misma configuración y de que haya, además, transiciones espontáneas, siempre es posible que el autómata «opte» por elegir un camino erróneo que conduzca a una configuración que lo bloquee antes de llegar al final. Los dispositivos nodeterministas, por tanto, y a diferencia de los deterministas, no definen procedimientos computacionales completos; pero, aun así, como veremos, su importancia es crucial, porque determinados procedimientos de decisión solo se pueden definir mediante dispositivos nodeterministas, ya que los deterministas se muestran demasiado restrictivos.

Además del «misterio» del nodeterminismo, el diagrama de la Figura 3.4 nos sirve para ilustrar una propiedad muy importante de los lenguajes regulares y a la que ya aludimos un poco más arriba: el hecho de que todo lenguaje regular se puede definir como la combinación de dos o más lenguajes también regulares a partir de alguna operación de composición. No olvidemos que los lenguajes son conjuntos y, por tanto, cualquier operación que podamos realizar con conjuntos (unión, intersección, etc.) podremos también realizarla con lenguajes. Pero, además, los lenguajes son conjuntos de cadenas, lo que hace que sea posible definir la concatenación de dos lenguajes.

Definición 3.4 (Concatenación de Lenguajes)**.** La concatenación de dos lenguajes regulares L_i y L_j es el lenguaje regular $L_k = L_i L_j$, cuyas cadenas son la concatenación de cualquier cadena de L_i con cualquier cadena de L_j, en este orden. Evidentemente, si la cadena vacía ε pertenece a alguno de los lenguajes que combinamos, esta se comportará como cualquier otra cadena de los lenguajes en cuestión.

De acuerdo con esta definición, vemos que, efectivamente, podemos definir el lenguaje $L_1 = \{a^*b^*\}$, como la concatenación de dos lenguajes L_a y L_b:

$$L_a = \{a^*\}$$
$$L_b = \{b^*\}$$
$$L_1 = L_a L_b$$

En el diagrama de la Figura 3.4 esto se ve perfectamente, ya que el primer ciclo, en el estado q_0, se corresponde con lo que sería (y, de hecho es) un autómata para el lenguaje L_a, mientras que el segundo ciclo, en el estado q_1, se corresponde con el autómata para el lenguaje L_b. La transición espontánea del uno al otro es la responsable de concatenar ambos autómatas en uno mayor y capaz de reconocer el lenguaje L_1. Nótese, finalmente, que, como tanto L_a como L_b contienen la cadena vacía, L_1 contiene todas las cadenas que pertenecen a L_a, así como todas las cadenas que pertenecen a L_b, además, claro está, de las respectivas concatenaciones. Por tanto,

$$L_a \subset L_1$$
$$L_b \subset L_1$$

de donde se sigue que

$$L_a \cap L_1 = L_a$$
$$L_b \cap L_1 = L_b$$

y, por tanto, podemos deducir que la intersección de dos lenguajes regulares es siempre un lenguaje regular. Evidentemente, si tanto L_a como L_b son subconjuntos de L_1, entonces

$$L_{ab} = (L_a \cup L_b) \subset L_1$$

Por tanto, si

$$L_1 - L_{ab} = L_c$$

entonces

$$L_1 = L_{ab} \cup L_c$$

es decir, que la unión de dos lenguajes regulares es siempre un lenguaje regular.

Los enunciados precedentes constituyen una demostración semiformal de que el conjunto de los lenguajes regulares está cerrado bajo las operaciones de concatenación, intersección, diferencia y unión; dicho de otro modo, la concatenación, la intersección, la diferencia y la unión de dos lenguajes regulares cualesquiera es un lenguaje regular. De hecho, el conjunto de los lenguajes regulares está cerrado bajo cualquier tipo de operación; nos faltan dos: la complementación y la clausura, y en ambos casos también se cumple la afirmación. Por tanto, el complemento $\overline{L_i}$ de un lenguaje regular es regular y la clausura transitiva L_i^* de un lenguaje regular es también regular. Ambas afirmaciones son relativamente fáciles de demostrar intuitivamente. Consideremos primeramente el caso del complemento y tomemos el lenguaje $L_1 = \{a^*b^*\}$ como referencia. Por la definición de complemento, sabemos que el complemento $\overline{L_1}$ de L_1 es el conjunto que contiene todos aquellos elementos que no están en L_1, de tal modo que $L_1 \cap \overline{L_1} = \emptyset$. Llamemos $\mathcal{L}_r$ al conjunto de todos los lenguajes regulares (incluido L_1) y tomemos la unión de todos los conjuntos que lo componen, $\bigcup \mathcal{L}_r$, que es el conjunto de todas las cadenas de todos los lenguajes de $\mathcal{L}_r$ y, por tanto, un lenguaje regular. Si ahora calculamos $\bigcup \mathcal{L}_r - L_1$, la diferencia, el resultado es el conjunto de todas las cadenas que no están en L_1, es decir su complemento $\overline{L_1}$, que será también regular. El caso de la clausura es todavía más sencillo. Considérese el conjunto $L_0 = \{a, b\}$. Demostrar que L_0 es un lenguaje regular es trivial (de hecho todos los conjuntos finitos son regulares), construyendo, por ejemplo, un AEFD con tres estados q_0, q_1 y q_2, todos ellos finales, tal que $\delta(q_0, a) = q_1$ y $\delta(q_0, b) = q_2$. Si L_0 es regular, su clausura transitiva $L_0^* = \{a, b\}^* = L_2$, y L_2, como ya sabemos, es regular.

3.2. Expresiones y gramáticas regulares

En la introducción del presente capítulo, observamos que la principal contribución de Noam Chomsky a la Teoría de los lenguajes formales fue demostrar la existencia de modelos matemáticos capaces de determinar la complejidad estructural de un lenguaje dado. También señalamos que estos modelos son de dos tipos: los autómatas y las gramáticas. Solo hay una excepción a esta generalización y la hallamos en el ámbito de los lenguajes regulares, para los que existe un tercer tipo de modelo además de los autómatas y las gramáticas: *las expresiones regulares*. En la sección 3.1 nos hemos ocupado de describir con detalle los AEF en sus variantes determinista y nodeterminista y hemos visto que ambas son modelos adecuados y equivalentes para caracterizar los lenguajes regulares. Aquí, nos ocuparemos de las expresiones regulares y de las gramáticas regulares que son, también, modelos matemáticos capaces de caracterizar solamente lenguajes dentro de esta clase. Aunque no nos ocuparemos de demostrarlo formalmente, existe una equivalencia absoluta entre los tres tipos de modelos y es, por tanto, posible construir cualquiera

de ellos a partir de otro de forma totalmente automática.

3.2.1. Expresiones regulares

Una expresión regular es una fórmula expresada en un lenguaje para el cual podemos definir una interpretación en un modelo. El modelo dentro del cual podemos definir una interpretación (o denotación) para cualquier expresión regular es, precisamente, el conjunto $\mathcal{L}_r$ de todos los lenguajes regulares. Como veremos, la definición de expresión regular, que es recursiva, explota la propiedad única de los lenguajes regulares según la cual cualquier lenguaje regular puede expresarse como la concatenación, unión o clausura de dos o más lenguajes regulares. Como paso previo a la definición de expresión regular, vamos a dar una definición un poco más precisa de concatenación, unión y clausura de lenguajes.

Definición 3.5 (Concatenación de lenguajes, II)**.** Si L_1 y L_2 son lenguajes, la concatenación de L_1 y L_2 es $L_1L_2 = \{xy \mid x \in L_1 \text{ e } y \in L_2\}$.

Lo importante aquí es ver que el conjunto resultante de la concatenación de dos lenguajes es aquel que contiene aquellas cadenas construidas a partir de la concatenación de todas las cadenas de L_1 con las de L_2 *en este orden*, más todas las cadenas de L_1 y todas las de L_2, que son el resultado de concatenar cada una de ellas con la cadena vacía. Evidentemente, dado que la concatenación no es conmutativa, la concatenación de L_1 y L_2 no define el mismo conjunto que la concatenación de L_2 y L_1.

Definición 3.6 (Unión de lenguajes)**.** Si L_1 y L_2 son lenguajes, la unión de L_1 y L_2 es $L_1 \cup L_2 = \{x, y \mid x \in L_1 \text{ e } y \in L_2\}$

En este caso, el punto relevante es que el conjunto unión resultante no es más que aquel conjunto que contiene las cadenas de L_1 y las cadenas de L_2. Por tanto, si $L_1 = \{ad, cd\}$ y $L_2 = \{ef, gh\}$, entonces $L_1 \cup L_2 = \{ab, cd, ef, gh\}$. Finalmente, y antes de pasar a la definición de clausura de un conjunto, conviene hacer hincapié en el hecho de que, a menos que uno de los conjuntos con los que operamos sea infinito, ni la concatenación ni la unión pueden dar lugar a lenguajes infinitos. Esto solo puede conseguirlo la operación de clausura.

Definición 3.7 (Clausura de Kleene de un lenguaje)**.** Si L es un lenguaje, la clasura transitiva o de Kleene de L es $L^* = \{x_1 x_2 \dots x_n \mid n \geq 0 \text{ y todo } x_i \in L\}$.

Conviene detenerse en esta definición, que viene a ser una generalización de la definición 1.7 de la página 13. Por si acaso, fijémonos primero en lo que la definición *no* dice. Lo que no dice es que L^* sea el conjunto de cadenas construidas a partir de la concatenación de cero o más copias de alguna cadena de L; eso sería $\{x^n \mid n \geq 0 \text{ y } x \in L\}$. Si así fuera, entonces a partir de $L = \{ab, cd\}$, por ejemplo, obtendríamos $\{ab, abab, cd, cdcd, ...\}$. De lo que se trata, en cambio, es de construir cadenas cuyas subcadenas x_i están en L. Por tanto, si $L = \{ab, cd\}$, entonces $L^* = \{\varepsilon, ab, cd, abcdab, cdcdab, abab, ...\}$. Nótese que, dado que $n \geq 0$, ε siempre está en L^*. Como apúntabamos más arriba, la operación de clausura de un conjunto siempre da lugar a conjuntos infinitos. Ahora ya podemos dar la definición de expresión regular.

Definición 3.8 (Expresión regular)**.** Una expresión regular es una fórmula $\mathbf{r}$ cuya denotación es un lenguaje $L(\mathbf{r})$ definido sobre un alfabeto Σ. Hay dos tipos de expresiones regulares: la expresiones regulares *atómicas* y las expresiones regulares *compuestas*. Cada tipo posee tres subtipos. La sintaxis y la denotación de las expresiones regulares atómicas son:

1. Cualquier literal $\mathbf{a}$, tal que $a \in \Sigma$ es una expresión regular cuya denotación es $L(\mathbf{a}) = \{a\}$.
2. El símbolo especial ε es una expresión regular cuya denotación es $L(\varepsilon) = \{\varepsilon\}$.
3. El símbolo especial $\emptyset$ es una expresión regular cuya denotación es $L(\emptyset) = \{\}$.

La sintaxis y la denotación de las expresiones regulares compuestas son:

4. $(\mathbf{r_1 r_2})$ es una expresión regular cuya denotación es $L(\mathbf{r_1 r_2}) = L(\mathbf{r_1})L(\mathbf{r_2})$.
5. $(\mathbf{r_1} + \mathbf{r_2})$ es una expresión regular cuya denotación es $L(\mathbf{r_1} + \mathbf{r_2}) = L(\mathbf{r_1}) \cup L(\mathbf{r_2})$.
6. $(\mathbf{r})^*$ es una expresión regular cuya denotación es $L((\mathbf{r})^*) = (L(\mathbf{r}))^*$.

Nada más puede ser una expresión regular.

En la definición, utilizamos la letra negrita para diferenciar las fórmulas de las cadenas que pueden hallarse en su denotación. Por lo tanto, $(\mathbf{a})^*$ denota, entre

otras cadenas, la cadena aaa. Al escribir una expresión regular, debemos tener en cuenta que el operador $*$ tiene mayor precedencia que el operador $+$, que tiene menor precedencia que la concatenación. Por lo tanto, en vez de escribir $((\mathbf{ab}) + \mathbf{c})$, podemos escribir $\mathbf{ab} + \mathbf{c}$, teniendo muy en cuenta, sin embargo, que $\mathbf{ab} + \mathbf{c}^*$ no es lo mismo que $(\mathbf{ab} + \mathbf{c})^*$. Veamos ahora algunos ejemplos.

Ejemplo 3.2.1. Con las reglas de la definición 3.8 podemos definir expresiones regulares tan complejas como queramos. Aquí vamos a ver solo algunos casos simples, pero relevantes.

Expresión: $\mathbf{aa}^*$

Denotación: $\{a, aa, aaa, ...\}$

En este primer ejemplo, tenemos una expresión regular compuesta que podemos descomponer en las expresiones regulares $\mathbf{a}$ y $\mathbf{a}^*$, esta última a su vez una expresión regular compuesta. La primera denota el lenguaje finito $\{a\}$, mientras que la segunda denota el lenguaje infinito $\{a^n \mid n \geq 0\}$. La composición de ambas, denota la concatenación de estos dos lenguajes que, como vemos, es el lenguaje formado por cadenas con una o más as; nótese que este lenguaje no contiene la cadena vacía.

Expresión: $(\mathbf{a} + \mathbf{b})^*$

Denotación: $\{a, b\}^*$

La expresión regular de este segundo ejemplo nos permite describir a un viejo conocido. Analicemos cada una de sus partes, empezando por el interior del paréntesis. Aquí tenemos la unión de dos conjuntos con un único elemento cada uno: $\{a\} \cup \{b\} = \{a, b\}$. Luego, calculamos la clausura de Kleene del conjunto resultante. El siguiente caso es parecido, pero el resultado muy diferente.

Expresión: $\mathbf{a}^* + \mathbf{b}^*$

Denotación: $\{a^n, b^n \mid n \geq 0\}$

Ahora, primero obtenemos el conjunto formado por cadenas con cero o más as y el conjunto formado por cadenas con cero o más bs; luego, hacemos la unión de ambos, que es el conjunto con cadenas con solo as o solo bs, más la cadena vacía. Para acabar, otro viejo conocido.

Expresión: $\mathbf{a}^*\mathbf{b}^*$

Denotación: $\{a^n b^m \mid n, m \geq 0\}$

Aquí concatenamos el conjunto formado por cadenas con cero o más as y el conjunto formado por cadenas con cero o más bs.

Al principio de esta sección ya adelantamos que no íbamos a desarrollar una prueba formal de la equivalencia entre expresiones regulares y autómatas de estados finitos. Sin embargo, dicha equivalencia es fácilmente perceptible de forma intuitiva. El «truco» consiste en descomponer la expresión regular en sus partes más básicas, construir un AEFD para cada una de ellas y conectar luego estos autómatas utilizando transiciones espontáneas para obtener un AEFN que los englobe a todos ellos. Podemos ejemplificar este procedimiento con el primer caso del ejemplo 3.2.1, que es el más simple.

Escribimos, primeramente, la expresión regular sin simplificar los paréntesis a fin de identificar mejor sus partes propias. Partimos, por tanto, de la fórmula $(\mathbf{a}(\mathbf{a}^*))$. Si eliminamos los paréntesis más exteriores, obtenemos $\mathbf{a}(\mathbf{a}^*)$, con el literal $\mathbf{a}$ cuya denotación es el conjunto $\{a\}$. Construir un AEFD para este lenguaje es elemental, basta con definir un estado inicial, no final, con una transición con la etiqueta a hacia un estado final. Hecho esto, nos queda transformar la parte $(\mathbf{a}^*)$ y conectarla al autómata que ya hemos construido. Para ello, definimos una transición espontánea desde el estado final del primer autómata hacia un nuevo estado final, para el cual definimos una transición en bucle también con la etiqueta a para dar cuenta de la clausura transitiva y, con ello, hemos completado el proceso; en la Figura 3.5 ilustramos gráficamente la secuencia de pasos de esta transformación. En los demás casos procederíamos de una manera similar.

3.2.2. Gramáticas regulares

Finalmente, el último modelo matemático para caracterizar lenguajes que nos queda por tratar es el de las gramáticas. En este caso, buena parte de lo que digamos aquí no resultará totalmente ajeno a nadie con unos mínimos conocimientos de lingüística, ya que el concepto de gramática propio de la teoría de los lenguajes formales no es realmente muy distinto del que se suele utilizar en lingüística, de hecho, el segundo procede históricamente del primero.

Una gramática no es más que un conjunto de reglas cuya aplicación recursiva siempre permitirá derivar una cadena del lenguaje para el cual se construyó la gramática. En este punto, nos resultará conveniente dar una definición genérica

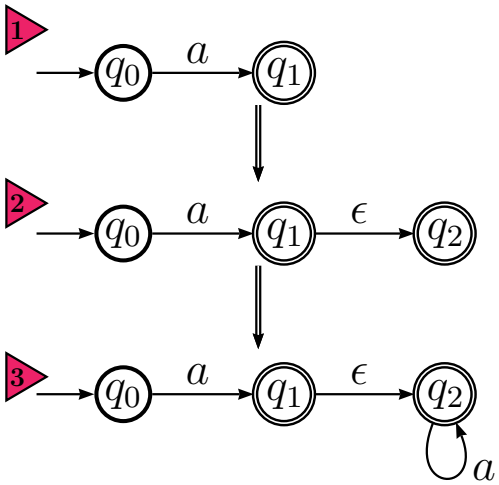


Figura 3.5: Los pasos para transformar una expresión regular en un AEFN.

de gramática para, luego, definir un tipo de gramática específico para caracterizar lenguajes regulares.

Definición 3.9 (Gramática)**.** Una *gramática* G es una cuádrupla $G = (V, \Sigma, S, P)$, tal que V es un alfabeto de *símbolos no terminales*, Σ es un alfabeto de *símbolos terminales* y $V \cap \Sigma = \emptyset$, $S \in V$ es el *símbolo inicial*, y P es un conjunto finito de *producciones* o *reglas*. Las reglas de P son siempre de la forma $x \to y$, tal que x e y son cadenas sobre $\Sigma \cup V$ y $x \neq \varepsilon$.

A partir de esta definición genérica de gramática, podemos definir la relación de *derivación* como una relación entre cadenas, dado un conjunto de producciones determinado y, finalmente, podremos dar una definición de lenguaje *generado* por una gramática. Todo ello nos permitirá poner de relieve una diferencia muy importante entre la noción de gramática formal y lo que suele tener en mente un lingüista cuando piensa en una gramática.

Definición 3.10 (Derivación)**.** Simbolizaremos como $\Rightarrow$ la relación entre dos cadenas w y z, tal que $w \Rightarrow z$, si, y solo si, existen las cadenas u, x, y y v sobre $\Sigma \cup V$, donde $w = uxv$ y $z = uyv$ y $(x \to y) \in P$. En este caso, llamaremos *derivación* a la secuencia, $x_0 \Rightarrow x_1 \Rightarrow ... \Rightarrow x_n$, de cadenas relacionadas por $\Rightarrow$, de tal modo que, para cualquier $n \geq 0$, la secuencia de cadenas de x_0 a x_n, tal que para todo $0 \leq i < n, x_i \Rightarrow x_{i+1}$, será una *derivación en n pasos*. Finalmente, simbolizaremos como $\overset{*}{\Rightarrow}$ la relación entre dos cadenas w y z, tal que $w \overset{*}{\Rightarrow} z$, si, y solo si, existe una derivación de cero o más pasos que empieza con w y termina con z.

Dada la definición de $\overset{*}{\Rightarrow}$, debemos asumir que la relación de derivación es reflexiva, ya que, para cualquier cadena x, podemos tener $x \overset{*}{\Rightarrow} x$ por una derivación de cero pasos.

A partir de la definición de derivación, podemos definir el lenguaje generado por una gramática.

Definición 3.11 (Lenguaje generado por G)**.** El lenguaje generado por una gramática G es $L(G) = \{x \in \Sigma^* \mid S \overset{*}{\Rightarrow} x\}$.

Es en este punto donde conviene hacer una pausa y poner de relieve las sutiles, pero muy importantes, diferencias entre el concepto de gramática formal y la idea de gramática que suelen manejar los lingüistas. La clave es la definición 3.11. Las gramáticas formales generan *conjuntos de cadenas*, lo cual es lógico ya que los lenguajes formales *son* conjuntos de cadenas. Lo que, crucialmente, una gramática formal *no hace ni hará nunca* es asignar una *descripción estructural* a las cadenas del lenguaje que genera y, como todos sabemos, para un lingüista no solo es importante que una gramática genere todas las frases de una lengua y solo esas, sino que también les asigne una descripción estructural adecuada. Esto es precisamente lo que motivó la distinción entre *capacidad generativa débil* y *capacidad generativa fuerte* que Chomsky introdujo en 1965, tal que una gramática genera débilmente un conjunto de cadenas y genera fuertemente un conjunto de descripciones estructurales, donde, solo lo segundo es relevante para una teoría lingüística. ¿Significa esto que, finalmente, la teoría de los lenguajes formales no tiene nada que aportar a las ciencias del lenguaje? No, por supuesto, pero es importante conocer sus limitaciones en el momento de hacer uso de los resultados que podamos obtener de su aplicación a casos concretos. La principal limitación, claro está, tiene que ver con la capacidad generativa fuerte que sería, en principio, aquella que nos podría proporcionar datos realmente fidedignos sobre la complejidad real de las estructuras lingüísticas. Desgraciadamente, nadie comprende muy bien o sabe realmente lo que es la capacidad generativa fuerte y, por tanto, nadie sabe como medir la complejidad desde este punto de vista. En este sentido, por tanto, una gramática formal nos proporciona una medida ciertamente indirecta del grado de complejidad de las estructuras lingüísticas que debemos interpretar, en todo caso, como un umbral mínimo. Es decir, si a través del análisis formal de, pongamos por caso, una determinada estructura fonológica somos capaces de determinar que se halla dentro de la escala de complejidad de los lenguajes regulares, sabemos que, como mínimo, esa es la complejidad del sistema y, por tanto, los recursos mínimos necesarios para computar la estructura en cuestión son los mismos que necesitaríamos para reconocer un lenguaje regular. Evidentemente, puede ser que

sean necesarios más recursos y, eso, solo podremos determinarlo por otros medios, pero lo que es seguro es que no será posible realizar la computación con menos; en la sección 3.3 volveremos sobre este asunto con algunos ejemplos concretos.

Veamos, pues, cuáles deben ser los recursos mínimos que necesita una gramática para generar un lenguaje regular. La definición 3.9 es una definición genérica de gramática, cualquier gramática de cualquier tipo se ajustará a ella, pero es posible hallar definiciones más restrictivas para caracterizar gramáticas capaces de generar solo los lenguajes de una determinada clase estructural, incluidos todos los lenguajes que pertenezcan a clases que sean subconjuntos propios de la clase en cuestión. Así, por tanto, aquellas gramáticas que por sus características solo sean adecuadas para generar lenguajes regulares, podran, claro está, generar lenguajes dentro de la Jerarquía Subregular, pero ningún lenguaje que pertenezca a una clase de orden superior dentro de la Jerarquía de Chomsky. Las gramáticas apropiadas para generar lenguajes regulares son aquellas que se conocen con el nombre de *gramáticas lineales* (*por la derecha* o *por la izquierda*).

Definición 3.12 (Gramática Lineal)**.** Una gramática $G = (V, \Sigma, S, P)$ es *lineal por la derecha* (resp. *lineal por la izquierda*), si, y solo si, para toda producción en P esta tiene o bien una de las dos formas siguientes (lineal por la derecha):

$$X \to zY$$
$$X \to z$$

o bien una de las dos siguientes (lineal por la izquierda):

$$X \to Yz$$
$$X \to z$$

donde $X \in V, Y \in V$ y $z \in \Sigma^*$. Una gramática que es o lineal por la derecha o lineal por la izquierda es una *gramática regular*.

Nótese, en primer lugar, que en las producciones de una gramática regular, a la izquierda de la flecha solamente puede haber un símbolo no terminal, mientras que a la derecha cómo máximo puede haber un no terminal y, como mínimo, una cadena posiblemente vacía de terminales. Si por un momento volvemos a la definición genérica de gramática de 3.9, vemos que ahí simplemente se nos dice que los elementos que pueden aparecer a izquierda y derecha de la flecha pueden ser cadenas construidas sobre la unión del conjunto de terminales con el de no terminales. Por tanto, en una gramática regular se impone una restricción muy fuerte sobre el formato de las reglas y es precisamente esta restricción la que

determina su capacidad generativa débil. Un formato habitual (y legal) en el que se suelen presentar las reglas de una gramática regular es el siguiente:

$$X \to wY$$
$$X \to \varepsilon$$

donde $|w| \leq 1$, es decir, la cadena de terminales w contiene un único símbolo o es vacía. Las gramáticas con este formato se denominan *gramáticas regulares de paso único*. Veamos ahora cómo serían las gramáticas regulares de los lenguajes L_1 y L_2 que nos han venido siriviendo de ejemplo a lo largo de toda esta presentación de los sistemas regulares. Utilizaremos el formato de paso único y, desde ahí, veremos que la transformación desde la gramática al autómata correspondiente es casi directa.

Ejemplo 3.2.2. La gramática regular de paso único para el lenguaje $L_1 = \{a^*b^*\}$ contiene las producciones siguientes:

$$S \to aS \quad (1)$$
$$S \to bR \quad (2)$$
$$S \to T \quad (3)$$
$$R \to bR \quad (4)$$
$$R \to T \quad (5)$$
$$T \to \varepsilon \quad (6)$$

La regla (1) es la que debemos aplicar para obtener cadenas que contengan alguna a, incluidas aquellas que solo contengan as, ya que una derivación $S \Rightarrow aS \Rightarrow ... \ S \Rightarrow T \Rightarrow \varepsilon$ sin aplicar nunca la regla (2) producirá siempre cadenas de estas características. La regla (2) es la que debemos aplicar para producir cadenas que contengan as seguidas de bs o también cadenas que contengan solo bs, ya que S es el símbolo inicial y es por tanto legal empezar la derivación en la regla (2); la regla (4) es la que nos permite generar subcadenas con un número arbitrario de bs. La cadena vacía la obtenemos aplicando las reglas (3) y (6).

La gramática regular de paso único para el lenguaje $L_2 = \{a, b\}^*$ contiene las producciones siguientes:

$$S \to aS \quad (1)$$
$$S \to bR \quad (2)$$
$$S \to T \quad (3)$$
$$R \to bR \quad (4)$$
$$R \to aS \quad (5)$$
$$R \to T \quad (6)$$
$$T \to \varepsilon \quad (7)$$

En este caso, solo merece comentario la regla (5) que es la que permite iniciar o reiniciar el ciclo de generación de subcadenas de *a*s después del ciclo de generación de subcadenas de *b*s. Las demás reglas funcionan exactamente igual que en el caso anterior.

Como apuntábamos más arriba, el formato de paso único nos permite hacer una traducción directa desde la gramática regular de un lenguaje a su autómata. Efectivamente, para ello basta interpretar los símbolos no terminales como estados, de tal modo que el de la izquierda de la flecha es el estado en que se encuentra el autómata cuando lee el terminal a la derecha de la flecha y el no terminal después de la flecha es el estado en el que tiene que entrar. Por ejemplo, la regla (1) en uno y otro caso del ejemplo 3.2.2, $S \to aS$, se convierte en la transición $\delta(S, a) = S$, donde $S = q_0$, el estado inicial del autómata. La transformación de las demás reglas es igual de simple. Las reglas (3) y (5) del primer caso—(3) y (6) en el segundo—, simplemente nos indican que tanto S como R son estados finales, ya que ambos conducen a la cadena vacía.

Para acabar, aunque hemos venido insistiendo en el hecho de que tanto autómatas como gramáticas (y expresiones regulares) son modelos equivalentes, unos y otras parecen comportarse de modos muy diferentes. En efecto, mientras las gramáticas se perciben como sistemas capaces de *generar* lenguajes, los autómatas parecen más bien sistemas capaces solo de *reconocerlos*. De hecho, la idea original es esta, aunque podríamos modificar ligeramente la definición de AEF, añadiéndole un alfabeto de salida, por ejemplo $\Sigma = \{0, 1\}$, para que nos respondiera, a través de un cabezal de escritura y una cinta auxiliares, si ha aceptado el lenguaje (1) o no (0); es lo que se conoce como una Máquina de Moore. En un AEF el cabezal principal es, por tanto, solo de lectura y siempre se supone que la cinta contiene una cadena que el autómata debe computar a fin de completar la tarea de reconocimiento. En teoría de la computación se da mucha importancia a la tarea de reconocer un lenguaje, porque, como veremos más adelante, a partir de ella y de determinados lenguajes con propiedades muy específicas se derivan resultados cruciales para lo que es el concepto global de computación. Sin embargo, dada la traducción directa entre una gramática de paso único y el conjunto de transiciones de un AEF, es perfectamente lícito entender la gramática como el «programa» que implementa el autómata que, con una mínima transformación podemos convertir en un dispositivo generador de lenguajes, de tal modo que el cabezal, en vez de ir leyendo símbolos de una cinta, los vaya escribiendo sobre las celdillas en blanco. Es solo cuestión de perspectiva.

3.3. Relevancia para la lingüística

El formato de paso único de las gramáticas regulares nos permite destacar la que probablemente es la propiedad más relevante de todo sistema regular: la presencia de un símbolo en un lugar determinado de la cadena depende exclusivamente del símbolo que lo precede inmediatamente. Dicho de otro modo, en un sistema regular las dependencias son únicamente de tipo lineal, pero con el añadido de que la relación lineal relevante es la de dependencia inmediata. Traducido a una jerga semilingüística, un sistema regular es incapaz de gestionar relaciones de dependencia a larga distancia. En términos computacionales esta propiedad también tiene una traducción muy precisa: los AEF son modelos computacionales sin memoria, ya que el sistema en ningún momento necesita «recordar» otros pasos que haya realizado anteriormente durante la computación más allá del inmediatamente anterior.

A priori, todo ello parece sugerir que los sistemas regulares no poseen ninguna relevancia para la lingüística, porque, como todo lingüista sabe, en las lenguas existen numerosas dependencias que no siguen pautas estrictamente lineales. Sin embargo, ello no significa que no existan dependencias lineales y su estudio puede resulta revelador. Por ejemplo, parece que la fonología de las lenguas naturales no sobrepasa en ningún momento los límites de la capacidad de los lenguajes regulares. Esto podemos verlo, por ejemplo, en el ámbito de las restricciones fonotácticas que determinan lo que es una sílaba posible en una lengua determinada. Si pensamos en el caso del español, veremos que es posible expresar dichas restricciones sobre ataques, núcleos y codas posibles mediante AEF. Construyamos los autómatas en cuestión. Definamos, primero, dos alfabetos $\mathcal{V}$ y $\mathcal{C}$ de vocales y consonantes, respectivamente:[3]

$$\mathcal{V} = \{a, e, i, o, u\}$$
$$\mathcal{C} = \{b, ch, d, f, g, j, k, l, ll, m, n, \tilde{n}, p, r, rr, s, t, w, y, z\}$$

Ahora, podemos definir los AEF respectivos para ataques, núcleos y codas posibles en español. El de los núcleos y el de las codas son triviales, si asumimos que en español los núcleos y las codas siempre son simples y contienen, respectivamente, un único símbolo del alfabeto $\mathcal{V}$ y un único símbolo del alfabeto $\mathcal{CODA} = \{d, j, l, n, r, s, z\}$, un subconjunto de $\mathcal{C}$.[4] El autómata para los ataques es un poco más interesante y lo tenemos representado en la Figura 3.6.

[3] Como en el ejemplo no buscamos la precisión lingüística, utilizamos un alfabeto a medio camino entre la ortografía y la fonética, de ahí, por ejemplo, la ausencia de la letras $< c >$ y $< q >$, cuyos valores fonéticos quedan cubiertos por los símbolos k y z, por ejemplo.

[4] Seguimos sin buscar la adecuación descriptiva y obviamos casos complicados como, por ejemplo, los diptongos, que podrían tratarse como núcleos complejos o, alternativamente, como

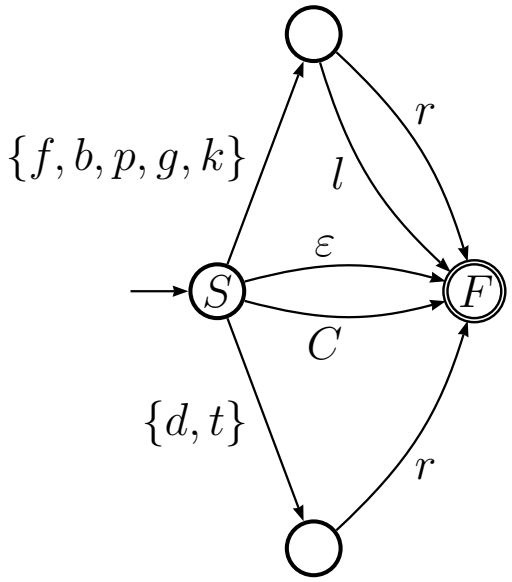


Figura 3.6: AEF para el lenguaje de los ataques silábicos del español; el símbolo C representa una variable sobre los elementos del conjunto $\mathcal{C}$.

En la figura hemos utilizado algunos trucos para minimizar un poco las dimensiones del autómata. Por un lado, hemos usado la variable C en la transición del estado inicial hacia el estado final para representar todas las transiciones con una única consonante para los ataques simples. Por el otro lado, para los ataques complejos, hemos definido dos transiciones hacia estados intermedios etiquetadas con subconjuntos de $\mathcal{C}$ para representar el primer segmento de los ataques complejos posibles. Finalmente, desde esos estados intermedios parten sendas transiciones etiquetadas r y l para representar el segundo segmento de los ataques complejos. Como vemos, el lenguaje ATAQUE de ataques silábicos posibles en español es un lenguaje finito y, por tanto, regular, como también lo son los lenguajes NÚCLEO y CODA. Ahora, podríamos concatenar estos tres lenguajes para construir el autómata para el lenguaje SÍLABA; Figura 3.7.

Similarmente, podríamos definir el lenguaje PALABRA como la iteración de una serie de sílabas, como mínimo una, lo que se puede expresar con una expresión regular que ya nos hemos encontrado anteriormente: $\sigma\sigma^*$, donde σ representa una sílaba, un lenguaje para el cual construimos un autómata en la sección dedicada

aproximantes en ataque (diptongos crecientes) o coda (diptongos decrecientes). En casos como los de «piano» y «puerta», podríamos quizá hablar de ataques con consonantes palatalizadas y velarizadas, respectivamente, como se ha defendido para el caso del catalán. Tampoco ofrecemos un tratamiento de los grupos consonánticos cultos que dan lugar a codas complejas como «ns» o «bs».

Figura 3.7: AEFN para el lenguaje SÍLABA de sílabas posibles en español construido como la concatenación de tres autómatas.

a las expresiones regulares.

En definitiva, parece que todo lo que está relacionado con el sistema de externalización del lenguaje no supera el límite de los sistemas regulares o, si se prefiere, una de las restricciones que impone la interfaz Senso-Motora es el sometimiento a un régimen estrictamente regular.[5] El resultado es interesante, porque, este parece ser el régimen que también siguen los sistemas de externalización en otras especies a la vista de lo que se observa en sus vocalizaciones. Por ejemplo, Kazuo Okanoya, en su estudio del canto de los pinzones de Bengala (*Lonchura striata domestica*), nos demuestra que las tonadas siguen combinaciones de notas estructuradas de acuerdo con el sistema regular caracterizable con el AEFN de la Figura 3.8.[6]

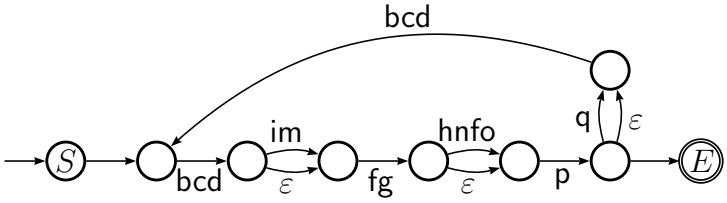


Figura 3.8: Estructura del canto de los pinzones de Bengala, donde cada símbolo representa una nota. Como se observa, algunas notas siempre aparecen agrupadas.

Similarmente, Karim Ouattara y colaboradores,[7] han descrito el sistema de llamadas de la mona de Campbell (*Cercopithecus campbelli*) cuya estructura también puede describirse con un AEFN; Figura 3.9.[8]

[5]Parece existir cierto consenso sobre la regularidad del componente fonológico del lenguaje humano; véase, por ejemplo, Jeffrey Heinz & William Idsardi. 2013. "What complexity differences reveal about domains in language". *Topics in Cognitive Science* 5:111–131.

[6]Kazuo Okanoya. 2002. "Sexual display as a syntactic vehicle: The evolution of syntax in birdsong and human language through sexual selection". En A. Wray (Ed.), *The Transition to Language*. Oxford: Oxford University Press, pp. 46–64.

[7]Karim Ouattara, Alban Lemasson & Klaus Zuberbühler. 2009. "Campbell's monkeys concatenate vocalizations into context-specific call sequences". *Proceedings of the National Academy of Sciences USA* 106(51):22026–22031.

[8]La formalización del autómata está tomada de Guillermo Lorenzo. 2013. *Biolingüística. La nueva síntesis*. Disponible en http://www.unioviedo.es/biolang/la-nueva-sintesis/.

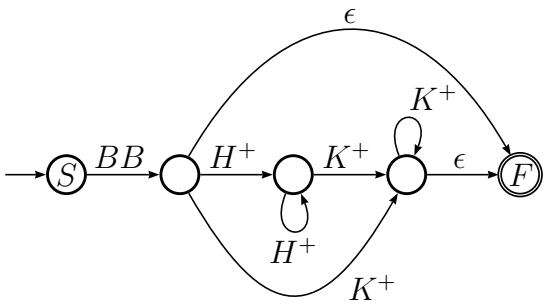


Figura 3.9: AEFN para caracterizar un subconjunto del repertorio de llamadas de una mona de Campbell. Los símbolos B, H y K representan subllamadas; el superíndice «+» indica que la subllamada tiene una duración mayor.

Estas observaciones pueden tener, por tanto, cierta relevancia desde el punto de vista evolutivo, por ejemplo, en la medida de que al menos uno de los sistemas externos asociados a la Facultad de Lenguaje impone unos principios de linealización de los objetos lingüísticos en los que la única operación permitida es la concatenación y donde las dependencias son estrictamente lineales y basadas en la relación de precedencia inmediata. Si esto es así, los demás subcomponentes de la gramática pertenecen a un ámbito totalmente distinto, ya que en todos ellos se observan propiedades estructurales que se hallan por encima de las capacidades de los sistemas regulares. La morfología, por ejemplo, y en la medida de que estemos dispuestos a aceptar que esta comporta la existencia de determinadas operaciones de combinación de morfos, no es siempre concatenativa. En efecto, operaciones como la infijación, cuyo producto es una cadena, no son sin embargo operaciones concatenativas y quizá nos remitan a lenguajes como $\{a^n b^i c^m \mid n \leq m > 1, i \geq 1\}$, que no es regular. Similarmente, la reduplicación, que podemos describir de forma abstracta con el lenguaje $\{ww\}$, no es, como veremos, ni siquiera independiente del contexto.[9]

[9]El caso de la reduplicación es claro, ya que los procesos de este tipo que observamos en lenguas malayo-polinesias como, por ejemplo, la formación del plural en tagalo o en indonesio, comportan claramente la generación de dos copias exactas de la cadena de entrada (reduplicación total). El caso de la infijación es quizá algo más problemático y depende, en parte, de cómo queramos describirla, ya que dichos procesos suelen ser sensibles a unidades estructurales fonológicas, como el ataque silábico, la sílaba, el pie métrico, etc. De todos modos, si nos remitimos de nuevo a las lenguas malayo-polinesias, la mayoría de procesos de este tipo suelen ser sensibles a la posición de ataque de la primera sílaba de la palabra y, por tanto, su caracterización formal no diferiría mucho de la que ofrecemos en el texto.

Nota sobre los lenguajes finitos

En alguna ocasión a lo largo de este texto ya hemos apuntado que los lenguajes finitos siempre son regulares. No lo hemos demostrado ni hemos hecho ninguna observación sobre qué relevancia lingüística puede tener este dato o si la tiene. Antes de pasar a hablar de lenguajes independientes del contexto, quisiéramos decir algo a este respecto.

Primeramente, veamos que cualquier lenguaje finito es regular. La prueba es intuitivamente bastante simple si consideramos, por ejemplo, el lenguaje finito como el siguiente:

$$L_3 = \{a^n b^n \mid 0 < n \leq 5\}$$

Como vemos, este lenguaje impone una restricción my fuerte sobre la constitución de sus cadenas, ya que el número de *a*s de la primera subcadena debe siempre ser igual al número de *b*s de la segunda. Solo que pensemos por un momento en qué comporta ser un lenguaje regular, veremos inmediatamente que ningún sistema regular es capaz de satisfacer esta condición, ya que no dispone de los recursos necesarios para «contar» los símbolos de la primera subcadena y determinar después si ese número coincide con el de símbolos de la segunda: el sistema regular solo «sabe» qué símbolo acaba de leer, nunca cuantos ha leído antes que él; por eso $L_1 = \{a^*b^*\}$ sí es regular y L_3 es, de hecho, un subconjunto propio de L_1. Desde este punto de vista, podría parecer que L_3 no es regular, pero resulta que a la restricción de que las dos subcadenas tengan el mismo número de símbolos hay que añadir la de que cada subcadena tenga como mínimo un símbolo y como máximo cinco, lo que significa que el lenguaje tiene solo cinco elementos:

$$\{ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb\}$$

Que esto está perfectamente dentro de las capacidades de un sistema regular es fácil de ver si pensamos que siempre podemos construir un AEF en el que utilicemos los estados del autómata para contar las *a*s y las *b*s. Mientras este número sea finito, no hay problema: el número de estados del autómata será finito. El problema, claro, surge cuando eliminamos la restricción sobre la longitud de las cadenas, porque, entonces, el lenguaje es infinito y, por tanto, necesitaríamos un número infinito de estados para contar *a*s y *b*s.

¿Tiene esto alguna relevancia para la teoría lingüística? Tiene, sí, pero en este caso por cuestiones ligadas a la aprendibilidad de lenguajes infinitos. La historia se remonta a un trabajo de Mark Gold publicado en 1967 en el que se presentan algunos resultados al respecto.[10] En esencia, Gold argumenta que un dispositivo mecánico (el aprendiz) expuesto a una muestra azarosa de cadenas pertenecientes

[10] E. Mark Gold. 1967. "Language identification in the limit". *Information and Control* 10(5):447–474.

a un lenguaje infinito dado jamás podrá completar la tarea a menos que exista algún tipo de supervisión que ayude al aprendiz a hacerlo. Este resultado se ha extrapolado al caso de la adquisición lingüística con interpretaciones que van desde la apelación a la necesidad de ofrecer datos negativos al aprendiz y no solo positivos a la idea de que el dispositivo de adquisición debe de incorporar algún tipo de sesgo que le permita efectuar algunas inferencias y no otras sobre las gramáticas. El caso es que el sistema lingüístico humano (y los sistemas de llamadas de los pinzones y los de las monas de Campbell) son sistemas potencialmente infinitos, lo que plantea algunas preguntas sobre su aprendibilidad a la luz de las conclusiones de Gold. La cuestión puede exponerse con cierta claridad si volvemos a los lenguajes L_1 y L_3 y añadimos el lenguaje L_4 a la ecuación:

$$\begin{aligned} L_1 &= \{a^*b^*\} \\ L_3 &= \{a^nb^n \mid 0 < n \leq 5\} \\ L_4 &= \{a^nb^n\} \end{aligned}$$

Supongamos ahora que un aprendiz va recibiendo estímulos que podrían pertenece a cualquiera de los tres lenguajes; por ejemplo, cadenas dentro del conjunto $\{ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb\} = L_3$. Como el tiempo de instrucción es necesariamente finito y los datos que recibe el aprendiz también son necesariamente finitos, si el corpus de datos se mantiene estable durante un tiempo dentro del ámbito definido por L_3, podemos concluir sin temor a equivocarnos que, dado que este lenguaje es relativamente pequeño, el aprendiz terminará por inferir que el sistema es, efectivamente, L_3. Supongamos ahora que el corpus de datos es mucho más amplio, las cadenas siguen conteniendo exactamente el mismo número de *a*s que de *b*s, pero observamos que este es claramente superior a 5, de tal modo que, a medida que avanza el proceso de instrucción, lo único que a priori podemos determinar es que el número de *a*s y *b*s puede ser superior a 5. ¿Qué hemos de concluir en este caso? ¿Estamos ante el lenguaje L_1, ante L_4 o ante un lenguaje finito pero muy grande (e.g. con $n \leq 10^9$)? Nótese que cualquiera de las hipótesis es, a priori, plausible, ya que tanto L_4 como ese hipotético lenguaje finito pero muy grande (llamémosle L_{10}) son subconjuntos propios de L_1. Por tanto, una posible generalización es inferir la gramática de L_1 que engloba todos los casos presentes en el corpus de instrucción más otros que no lo están, pero como es imposible que estén todos, a alguna conclusión tiene que llegar el aprendiz. La primera moraleja de este pequeño experimento mental es que, a partir de un determinado umbral (no sabemos cuál, claro) y dada una situación mínimamente realista de aprendizaje, *no hay diferencia alguna entre un lenguaje finito y otro infinito*, ya que la finitud no es necesariamente garantía de poder acceder a todos los datos. En situaciones como esta, por tanto, parece que la hipótesis más plausible es inferir que estamos ante un sistema infinito. Este podría ser el sesgo a que nos referíamos anteriormente en relación con el dispositivo de adquisición. Dado este sesgo, por

tanto, podemos asumir que el aprendiz pronto descartará la hipótesis de L_{10}; lo más probable es que ni siquiera la contemple y, superado ese indeterminado umbral a que aludíamos, descarte por completo la hipótesis de la finitud. La cosa, por tanto, está entre L_1 o L_4. ¿Qué comporta optar por uno u otro lenguaje? Mucho, porque L_1 es regular, pero L_4 no, de tal modo que la opción L_4, la más plausible a ojos de cualquiera que lea esto, comporta inclinarse por un sistema estructuralmente más complejo. Es decir, que simplicidad no siempre rima con plausibilidad. ¿Puede ser esta una explicación de por qué el lenguaje humano parece poseer una notable inflación de recursos computacionales? ¿Acaso algún sesgo en el dispositivo de adquisición nos hace inferir un sistema de gran complejidad a partir de datos aparentemente simples? ¿Cuál puede ser ese sesgo?

Capítulo 4

Lenguajes independientes del contexto

Al final del capítulo anterior hemos visto que existen lenguajes cuya complejidad estructural supera los límites de la regularidad. Una clase particularmente importante de este tipo de lenguajes, por sus aplicaciones informáticas y por su relevancia para la lingüística, es la clase de los lenguajes independientes del contexto.

También en el capítulo anterior vimos que es relativamente simple determinar que un lenguaje no es regular en relación con el lenguaje $L_4 = \{a^n b^n\}$. La demostración semi-formal de que L_4 efectivamente no es regular es relativamente sencilla y comporta intentar construir un AEF capaz de reconocer el lenguaje. Se trata de hallar un autómata capaz de contar as para, después, garantizar que la subcadena de bs contiene exactamente el mismo número de símbolos que la subcadena de as. Basta pensar un poco para ver que es imposible construir un AEF capaz de hacer esto. En primer lugar, L_4 es infinito y, por tanto, la cardinalidad de los lenguajes $L_{4_1} = \{a^n\}$ y $L_{4_2} = \{b^n\}$ que debemos concatenar para obtener L_4 es también infinito. Por tanto, necesitaríamos un número infinito de estados para contar todas las posibles as que pueden formar la primera subcadena del lenguaje y un número infinito de estados para contar el número de bs de la segunda subcadena, a fin de determinar si el número de símbolos es el mismo en ambos casos. Evidentemente, el autómata resultante no será en ningún caso un AEF. Por consiguiente, L_4 no es regular. Como veremos más adelante, para reconocer un lenguaje de estas características necesitamos un autómata que incorpore un recurso que no hemos necesitado hasta ahora: *memoria*, para almacenar un registro del número de as y, así, cotejar ese dato con el número de bs. Con una cantidad relativamente limitada de este recurso podremos, sin embargo, tratar lenguajes cuyas propiedades estructurales van más allá de las simples dependencias lineales que se observan en los sistemas regulares. Podremos, de hecho, establecer *dependencias estructu-*

rales a larga distancia, aunque no de un modo totalmente irrestricto, ya que las limitaciones de memoria solo permiten dependencias de este tipo que obedezcan ciertas restricciones. Para comprender bien cuáles son esas restricciones, conviene ocuparse primero de las gramáticas capaces de generar este tipo de lenguajes y, a continuación, de los autómatas correspondientes.

4.1. Gramáticas independientes del contexto

Las gramáticas independientes del contexto relajan algunas restricciones sobre el formato de las reglas en relación con las de las gramáticas regulares. Veamos, primero, su definición formal y, a continuación, comentaremos las principales consecuencias estructurales de los lenguajes que estas gramáticas generan gracias a esta relajación de restricciones.

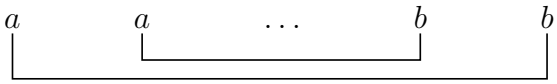
Definición 4.1 (Gramática independiente del contexto)**.** Una *gramática independiente del contexto* es una cuádrupla $G = (V, T, P, S)$, tal que V y T son conjuntos finitos de *variables* y *terminales*, respectivamente y $V \cap T = \emptyset$. P es un conjunto finito de *producciones* o *reglas* y todos los miembros del conjunto son de la forma $A \to \alpha$, donde A es una variable y α es una cadena de símbolos en $(V \cup T)^*$. S es una variable especial que denominaremos *símbolo inicial*.

Basta comparar esta definición con la de gramática lineal en 3.12 para ver qué propiedad estructural distingue a los lenguajes regulares de los independientes del contexto: la linealidad. Recordemos que las reglas de las gramáticas lineales se caracterizan por permitir, a la derecha de la flecha, un único terminal seguido o precedido por una cadena de símbolos terminales. Ahora, a la derecha de la flecha podemos tener cadenas que combinen en cualquier orden terminales y no terminales, de tal modo que las dependencias entre terminales ya no son estrictamente lineales. Para comprenderlo mejor, veamos un ejemplo muy simple de gramática independiente del contexto para el lenguaje L_4.

Ejemplo 4.1.1. Tenemos una gramática $G = (V, T, P, S)$, donde $V = \{S\}$, $T = \{a, b\}$ y $P = \{S \to aSb, S \to ab\}$. Tenemos, por tanto, una única variable, que es también el símbolo inicial, y dos símbolos terminales. De las dos producciones, la que nos interesa es $S \to aSb$, que es la que contiene variables a uno y otro lado de la flecha; la producción $S \to ab$ es necesaria para obtener la cadena ab y, como veremos, para cerrar la aplicación recursiva de la primera producción. Si iniciamos ahora una derivación con la primera regla tal y como la definimos en 3.10 tenemos

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow a^3Sb^3 \Rightarrow \ldots \Rightarrow a^{n-1}Sb^{n-1} \Rightarrow a^nb^n.$$

Es decir, mientras realicemos aplicaciones sucesivas de la regla $S \to aSb$, siempre obtenemos una cadena que incluye la variable S que, además, separa las subcadenas de as y bs que vamos generando incrementalmente. Nótese que, gracias al formato de la regla $S \to aSb$, la dependencia entre la primera a y la primera b que introducimos se conserva a medida que vamos introduciendo nuevas dependencias en aplicaciones sucesivas de la misma regla. Dichas dependencias, además, son claramente no lineales en el sentido de que los símbolos terminales relacionados no son adyacentes en la cadena como ocurriría con un sistema regular. Es así que conseguimos «contar» el número de as y bs de la cadena. Finalmente, la aplicación recursiva de $S \to aSb$, se cierra con la regla $S \to ab$, que no contiene ninguna variable a la derecha de la flecha. Un último detalle, nótese que la relación que existe entre los símbolos terminales se caracteriza por el hecho de que la primera a de la cadena está relacionada con la última b, la segunda a con la penúltima b y, así, sucesivamente. Algo así:



Este punto es relevante para comprender algunas propiedades importantes de los lenguajes independientes del contexto, tal como veremos a medida que avancemos en este capítulo.

Esta propiedad que hemos desvelado en el ejemplo 4.1.1 merece un poco más de atención. Veamos, en primer lugar, algunos lenguajes independientes del contexto más:

$$L_5 = \{a^n b^i c^m \mid n \leq m > 1, i \geq 1\}.$$
$$L_6 = \{ww^R \mid x \in \{a, b\}^*\}.$$
$$L_7 = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\}.$$
$$L_8 = \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}.$$

Empecemos por L_5. Este lenguaje establece una dependencia entre la subcadena inicial de a y la subcadena final de cs, tal que la primera nunca puede ser más larga que la segunda; el lenguaje exige, además, que entre ambas subcadenas se intercale una subcadena que contenga como mínimo una b. Es muy probable que el tipo de condiciones que un morfólogo establecería para los procesos de infijación hagan que estos resulten de hecho más complejos de lo que puede tratar una gramática independiente del contexto. Sin embargo, L_5 nos muestra un lenguaje cuya estructura, si no es la de la infijación, se aproxima mucho a ella, en el sentido de que, en la práctica, tenemos una cadena discontinua de as y cs entre las que existe una dependencia mutua en el interior de la cual se intercala una cadena arbitrariamente larga de bs (el supuesto infijo). Por ejemplo, si consideramos el caso de la

formación de las formas pasivas e imperativas de los verbos en tagalo—mediante infijación de *-in-* y *-um-*, respectivamente—, las cadenas resultantes producto de la combinación de la raíz con el afijo tienen exactamente esta estructura:[1]

(1) a. basa → b**in**asa (es leído)

b. basa → b**um**asa (lee tú)

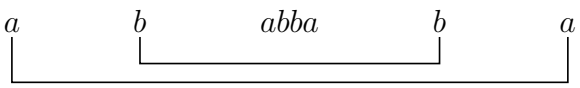
c. tawag → t**in**awag (es gritado)

d. tawag → t**um**awag (grita tú)

Conviene insistir en el hecho de que, al ser la infijación una operación no concatenativa que, además, quizá obedezca restricciones adicionales, esta podría superar los límites de la independencia del contexto. Sin embargo, el ejemplo no deja de ser ilustrativo del tipo de relaciones estructurales que sí se hallan dentro de los límites de la independencia del contexto.

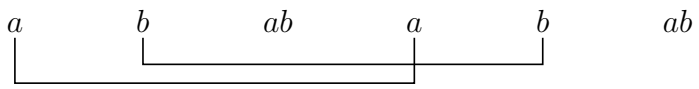
El lenguaje L_6 no tiene relevancia lingüística alguna, al menos que se sepa, ya que sus cadenas están siempre formadas por una cadena arbitrariamente larga de as y de bs (w) seguida por una cadena que es la imagen especular de la primera (w^R). L_6 contiene, por tanto, cadenas tales como *ababbaba*, *aabbbbaa*, *abbaabba*, etc. Este lenguaje es interesante porque nos ilustra de manera perfecta algo que *no* está dentro de la capacidad de un sistema independiente del contexto: la reduplicación. Si nos fijamos, las dependencias entre símbolos terminales en el lenguaje L_6 siguen la pauta anidada que ya observamos en el ejemplo 4.1.1 y que reproducimos abajo como (2) para la cadena *ababbaba*:

(2)



Sin embargo, la reduplicación se correspondería con el lenguaje $L_9 = \{ww \mid x \in \{a, b\}^*\}$ donde la pauta es otra, ya que el primer símbolo de la segunda cadena debe corresponderse con el primero de la primera cadena, no con el último como ocurre en L_6. El patrón ahora es cruzado, no anidado. Lo ilustramos en (3) para la cadena *abababab*, que podría ser el equivalente de una forma del plural formada por reduplicación total de la raíz como la que observamos en indonesio en casos como *rumah* (casa) vs. *rumahrumah* (casas).

[1] Las traducciones son solo aproximadas, ya que en tagalo los verbos no muestran marcas de flexión de número ni de persona. Los infijos aparecen destacados en negrita.

(3) 

Estructuralmente, algo como lo que tenemos en (3) está totalmente fuera del alcance de un sistema independiente del contexto y, como sabemos, en las lenguas naturales este tipo de dependencias cruzadas no son raras.

Finalmente, los lenguajes L_7 y L_8 son interesantes sobre todo por dos motivos. El primero es que las dependencias múltiples a larga distancia *son independientes del contexto*; el segundo es que estas solo son posibles si *no están relacionadas entre sí o están anidadas*. L_7 es un ejemplo de lo primero y L_8 un ejemplo de lo segundo. Construcciones que sigan la pauta definida por L_7 no son raras; por ejemplo:

(4) ¿A quién$_i$ instó el juez t_i a confesar quién$_j$ t_j había robado las joyas?

Sin embargo, la pauta de L_8 no siempre es posible, aunque a veces parece ser la preferida. ¿Cómo os suenan los ejemplos siguientes?:

(5) a. ¿Quién$_i$ no sabes qué libro$_j$ t_i le ha regalado t_j a María? (cruzada)

b. ¿A quién$_i$ no sabes quién$_j$ t_j le ha regalado un libro t_i? (anidada)

c. ¿A quién$_i$ no sabes qué libro$_j$ le ha regalado Juan t_j t_i? (anidada)

d. ¿Qué libro$_i$ no sabes a quién$_j$ le ha regalado Juan t_i t_j? (cruzada)

Es evidente que los famosos efectos de superioridad con toda seguridad van más allá de un mero contraste entre dependencias cruzadas o anidadas, pero, al menos a juicio de quien esto escribe, las construcciones con dependencias anidadas son preferibles a aquellas con las dependencias cruzadas. Sea como sea, las dependencias cruzadas no son independientes del contexto y, como vamos viendo poco a poco, ello tiene que ver con la manera que tiene de gestionar la memoria un sistema de este tipo. Esto lo veremos claro cuando nos ocupemos de los autómatas de pila, pero intuitivamente debería ser posible ver que el problema está en la manera que el sistema tiene de «contar» o de «recordar» los símbolos terminales que ya ha procesado y de los cuales dependen otros que deben aparecer más adelante.

4.1.1. Árboles de derivación y 'center-embedding'

Para profundizar un poco más en las propiedades estructurales de los lenguajes independientes del contexto, conviene regresar por un momento a la definición de derivación de 3.10. Como se deduce de la definición y como ya apuntamos

en su momento, las gramáticas y los autómatas son modelos matemáticos que nos permiten definir conjuntos de cadenas, es decir lenguajes. De ahí que una derivación, entendida como la aplicación sucesiva de las reglas de una gramática dada sea una relación entre cadenas. Las gramáticas formales generan cadenas, no descripciones estructurales. Sin embargo, también es cierto que una derivación no deja de ser un proceso, una secuencia de pasos ordenados en el tiempo y que culmina con la generación de una cadena completa. Vista así, tiene sentido trazar la historia de la derivación con la forma de un árbol que crece a medida que vamos aplicando reglas y expandiendo los símbolos no terminales y que termina cuando ya no queda ninguna variable por sustituir. Al hacer esto, está claro que superponemos una estructura jerárquica sobre la cadena del lenguaje que ni la gramática ni el autómata correspondiente son capaces de generar. Es muy importante entender esto porque suele dar lugar a muchas confusiones: el árbol no surge naturalmente a partir de la aplicación sucesiva de las reglas de la gramática, sino que es *un artefacto externo que nosotros mismos construimos a partir de una determinada interpretación del proceso de derivación de una cadena*. Más importante aún, sobre todo en el contexto en que nos encontramos, *un árbol de derivación no tiene por qué ser una representación de la estructura de constituyentes de una cadena*, si es que tiene sentido hablar de «estructura de constituyentes» de una cadena en teoría de lenguajes formales. Más adelante volveremos sobre este asunto, de momento, demos una definición formal de árbol de derivación.

Definición 4.2 (Árbol de derivación)**.** Sea $G = (V, T, P, S)$ una gramática independiente del contexto. Un árbol será un *árbol de derivación* para G si:

1. Todo vértice tiene una *etiqueta* y esta es un símbolo en $V \cup T \cup \{\epsilon\}$.
2. La etiqueta de la raíz es S.
3. Si un vértice es interior y lleva la etiqueta A, entonces $A \in V$.
4. Si n lleva la etiqueta A y los vértices $n_1, n_2, \ldots, n_k$ son hijos del vértice n, en ese orden y empezando por la izquierda, y llevan las etiquetas $X_1, X_2, \ldots, X_k$, respectivamente, entonces

   $$A \to X_1 X_2 \ldots X_k$$

   es una regla en P.
5. Si el vértice n lleva la etiqueta ϵ, entonces n es una hoja y es la única hija de su vértice madre.

Nótese que, tal como está definido, el concepto de árbol de derivación solo tiene sentido para una gramática independiente del contexto, no para una gramática

regular. El motivo es obvio: en una gramática regular nunca habrá símbolos no terminales en el interior de la cadena, ya que estos siempre se hallan en uno u otro extremo. Por tanto, y dada la definición de 4.2, el árbol de derivación que obtendríamos a partir de la que aparece en el ejemplo 4.1.1 es el que tenemos representado en la figura 4.1.

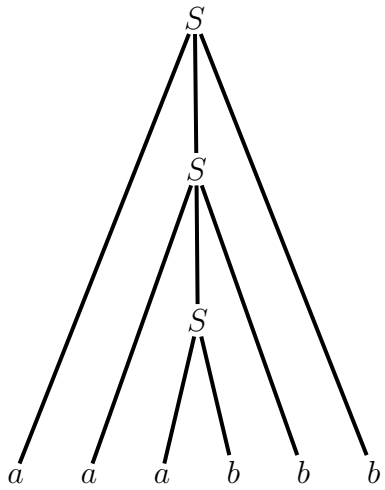


Figura 4.1: Árbol de derivación para la cadena *aaabbb* del lenguaje L_4.

Es importante insistir en el hecho de que lo que genera la gramática es la cadena símbolos terminales que aparecen en las hojas del árbol, no el árbol en sí. Sin embargo, este nos proporciona un dato particularmente relevante sobre una propiedad estructural básica de los lenguajes independientes del contexto. Si nos fijamos en cómo ha procedido la derivación, vemos que en el primer paso, que es el que marca el vértice raíz, se introducen dos terminales separados por una variable. Eso, en la práctica, significa que introducimos una dependencia discontinua que, además, va ganando distancia a medida que el espacio que separa ambos terminales se va ensanchando con la introducción de nuevos terminales. Si pensamos en cómo proceden las derivaciones en una gramática regular, en seguida veremos que, en este caso, la cadena va creciendo de forma continua, por la derecha o por la izquierda, a medida que vamos añadiendo terminales que siempre son adyacentes. En el caso de 4.1, la cadena también crece de manera continua, pero por ambos extremos a la vez y hacia el interior de la cadena. Cómo veremos más adelante, el número «mágico» de los lenguajes independientes del contexto es precisamente el 2 (por las dos subcadenas en los extremos que crecen simultáneamente durante la derivación), una propiedad a la cual inmediatamente dotaremos de mayor contenido formal y que nos servirá para identificar lenguajes que no sean independientes del contexto.

Gracias a los árboles de derivación es posible establecer una propiedad genérica que se cumple para todo lenguaje independiente del contexto. Esta viene a ser

una variante del lema del bombeo para los lenguajes regulares, pero de aplicación exclusiva para los lenguajes independientes del contexto y discernible solamente a partir de los árboles de derivación. El lema del bombeo para los lenguajes regulares nos dice que siempre podemos hallar una subcadena en el interior de la cadena que podemos repetir tantas veces como queramos y que pertenece al lenguaje. En el caso de los lenguajes independientes del contexto, veremos que siempre podemos hallar *dos* subcadenas de igual longitud, separadas entre sí, pero lo suficientemente próximas, que se pueden repetir el mismo número de veces y tantas veces como queramos. Antes de demostrar esta propiedad con algunos árboles de derivación, tenemos que definir un requisito previo que estos deben cumplir.

Definición 4.3 (Árbol de bombeo)**.** Un *árbol de bombeo* para una gramática independiente del contexto $G = (V, T, S, P)$ debe ser un árbol de derivación que cumpla las dos propiedades siguientes:

1. Existe un vértice etiquetado con un símbolo no terminal A que tiene al mismo símbolo no terminal A como uno de sus descendientes.

2. La cadena de terminales generada por el ancestro A es más larga que la cadena de terminales generada por el descendiente A.

Considérese ahora una variante genérica del árbol de derivación de la figura 4.1, como la que tenemos en la figura 4.2. Este árbol cumple los requisitos para ser un árbol de bombeo para una gramática independiente del contexto G. En primer lugar, el árbol da lugar a la cadena $uvwxy$.[2] En segundo lugar, se cumple la primera condición para ser un árbol de bombeo, ya que tenemos un variable A que tiene a otra variable A como una de sus descendientes. También se cumple la segunda condición, ya que la cadena generada por la variable A superior (más próxima a la raíz) es necesariamente más larga que la generada por la variable A inferior ($|vwx| > |w|$).

Satisfechas estas condiciones, podemos afirmar que si una gramática G genera un árbol de bombeo como el de 4.3, entonces el lenguaje $L(G)$ generado por esa gramática incluye las cadenas uv^iwx^iy para todo i. La demostración es muy simple, si tenemos en cuenta que el árbol contiene dos subárboles definidos por la misma variable A. Ello significa que podemos por tanto substituir cualquiera de ellos por el otro y que siempre obtendremos una cadena legal. En la figura 4.3 tenemos el resultado de sustituir el árbol superior por el inferior; en 4.4 tenemos el árbol resultante al sustituir el árbol inferior por el superior.

[2]Conviene recordar que cuando usamos las últimas letras del alfabeto denotamos cadenas, no necesariamente símbolos terminales: Por tanto, $uvwxy$ es la concatenación de cinco subcadenas que pueden contener más de un símbolo terminal.

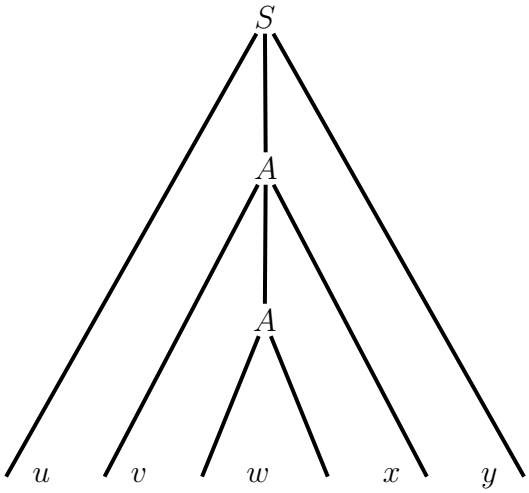


Figura 4.2: Árbol de bombeo para la cadena $uvwxy$.

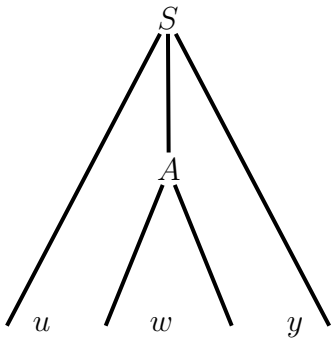


Figura 4.3: Árbol de bombeo para la cadena uwy, donde v y x son ε.

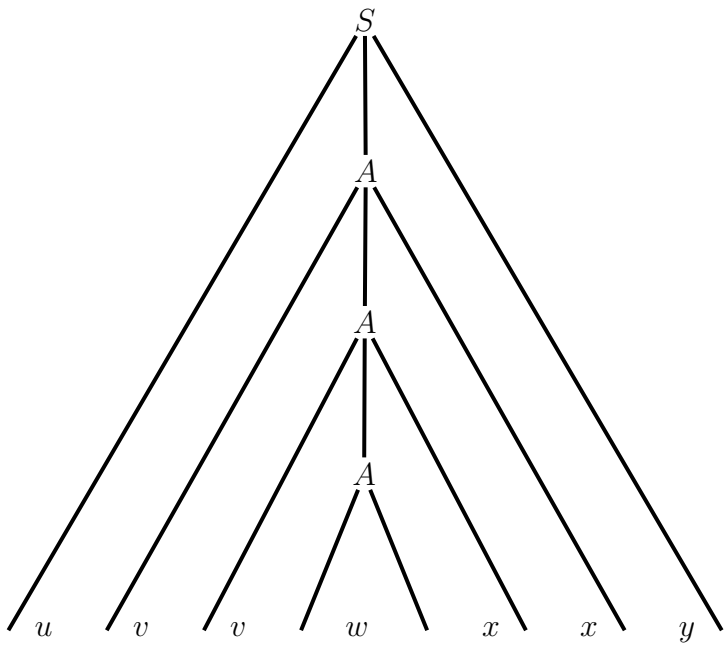


Figura 4.4: Árbol de bombeo para la cadena uv^2wx^2y.

Podemos repetir esta operación todas las veces que queramos y siempre obtendremos una cadena de la forma uv^iwx^iy para cualquier valor de i. Como anticipábamos un poco más arriba, el número «mágico» es 2, correspondiente a las dos cadenas v^i y x^i que crecen a la vez a medida que procede la derivación: por eso hablamos de dependencia. En un lenguaje independiente del contexto, pueden existir tantas dependencias de este tipo como queramos, lo importante es que siempre pongan en relación pares de subcadenas (o columnas, como a veces se dice). Todo ello se sigue, de hecho, de la propia naturaleza de las reglas de una gramática independiente del contexto que, al permitir una única variable a la izquierda de la flecha, solo puede poner en relación pares de columnas. Efectivamente, nótese que aunque tuviésemos una regla como $A \to aAbBc$, mientras es posible mantener el «control» sobre el crecimiento de las subcadenas de as y de bs, es imposible hacer que las subcadenas de cs crezcan al mismo ritmo. Por ejemplo, si asumimos que tenemos también las reglas $B \to c$ y $A \to ab$, entonces tendríamos derivaciones como

$$A \Rightarrow aAbBc \Rightarrow aaAbBcbBc \Rightarrow aaabbBcbBc \Rightarrow aaabbccbBc \Rightarrow aaabbccbcc.$$

De ahí se sigue que los lenguajes con tres o más columnas (con dependencias que vayan más allá de pares de cadenas, como $a^nb^nc^n$ o $a^nb^nc^nd^n$) no sean independientes del contexto y tengan un complejidad estructural mayor.

Para terminar esta sección, una consideración final sobre una cuestión que ha dado mucho de que hablar últimamente: *la recursividad*. Si nos fijamos en los pro-

cesos de derivación en los sistemas regulares y en los independientes del contexto, observamos que en ambos casos las cadenas se generan a través de la aplicación *recursiva* de las reglas. Desde este punto de vista, ambos tipos de sistemas son idénticos, ya que en ambos se observa lo que podríamos llamar *recursividad de proceso*. Sin embargo, los sistemas independientes del contexto, como nos demuestra la existencia de árboles de bombeo, possen una propiedad estructural especial, que podríamos denominar *recursividad de estructura*, que es la que permite que podamos sustituir un subárbol por otro definido por el mismo no terminal y obtener siempre una cadena que pertenezca al lenguaje. Sospecho que, desde el punto de vista de la lingüística, la recursividad que nos interesa es la segunda.

4.2. Autómatas de pila

Para terminar este capítulo, cerraremos el círculo dedicando un breve apartado a los autómatas de pila, la clase de autómatas equivalente a las gramáticas independientes del contexto. Aquí podremos establecer la conexión entre el tipo y la naturaleza de las dependencias que permite un sistema independiente del contexto con el recurso computacional de la *memoria* al que ya aludimos al principio del capítulo.

Si tomamos como referencia un AEF, el autómata de pila posee los mismos tres elementos básicos: una unidad de control, un cabezal de lectura con desplazamiento hacia la derecha y una cinta donde se disponen en celdillas los símbolos de la cadena que el autómata debe reconocer. El autómata de pila, sin embargo, incorpora dos elementos más: una cinta auxiliar o *pila* y un cabezal adicional capaz de leer, escribir y borrar símbolos de la pila.

Para comprender su funcionamiento, pensemos en un autómata de pila como el que tenemos en la figura 4.5 durante el proceso de reconocer la cadena *aaabbb* del lenguaje independiente del contexto L_4.

En la imagen, vemos que el cabezal de lectura apunta a la última a de la cadena, mientras que el cabezal adicional acaba de escribir a en la celdilla superior de la pila. Como se observa, las dos celdillas inferiores de la pila contienen sendas as que el cabezal correspondiente introdujo cuando el cabezal principal leyó la segunda y la primera a, respectivamente. La pila, por tanto, se va llenando de abajo a arriba de tal modo que el símbolo de la pila que está por debajo de todos los demás se corresponde con el primer símbolo de la cadena de la cinta.

Cuando el cabezal principal lee la primera b de la cadena (figura 4.6), el cabezal auxiliar borra la primera a de la pila y la pila automáticamente se desplaza hacia arriba de tal modo que el cabezal, que permanece fijo, apunte a la siguiente a. De este modo, el autómata es capaz de ir contando las as y las bs, ya que la pila le

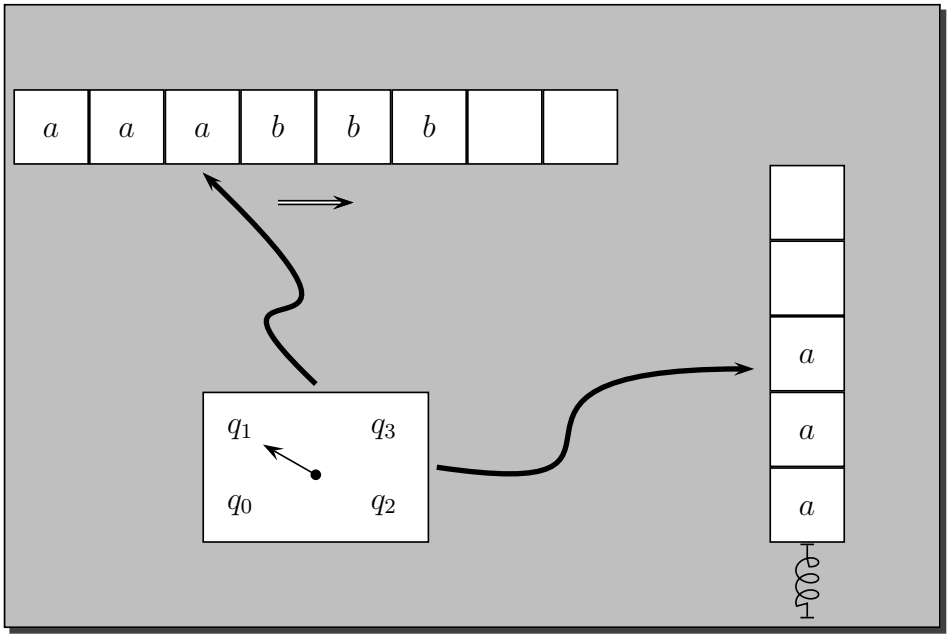


Figura 4.5: Un autómata de pila tras leer la subcadena *aaa*.

permite guardar las *a*s que ha leído a fin de cotejar posteriormente cada símbolo con la *b* que le corresponde. Vemos, sin embargo, que la estructura de la pila impone un régimen muy estricto sobre cómo debe realizarse ese cotejo, ya que la primera *a* de la cinta siempre será la última en la pila y, por tanto, solo se borrará cuando aparezca la última *b*. Y viceversa, la última *a* de la cinta, al ser la primera en la pila, se borrará tan pronto como aparezca la primera *b*.

Si observamos ahora las cosas desde esta perspectiva, podríamos decir que esta manera que tienen los sistemas independientes del contexto de gestionar las dependencias, anidándolas, es una consecuencia directa de la manera como está estructurado su sistema de memoria, que impone un régimen según el cual «el primero que entra es el último que sale». Más importante aun, dado que el cabezal de la pila permanece fijo y es la propia pila la que se desplaza hacia arriba o hacia abajo, como si tuviera debajo un muelle que se contrae o se expande en función del «peso» de los símbolos que contiene, el cabezal solo tiene acceso al símbolo que ocupa la posición más alta de la pila. Nunca puede «ver» qué hay por debajo y, sobre todo, nunca puede utilizar las celdillas que quedan libres para guardar otros símbolos. Si así fuera, el autómata podría utilizar regiones diferentes de la pila para guardar más datos, como, por ejemplo, el número de *b*s, en cuyo caso podría cotejar estas con, por ejemplo, el número de *c*s detrás de las *b*s en un lenguaje como $a^n b^n c^n$. Pero, entonces, ya no tendríamos un autómata de pila simple, sino algo mucho más potente. Precisamente, el interés de concebir el funcionamiento de la pila de la manera en que lo hemos descrito reside en el hecho de que así

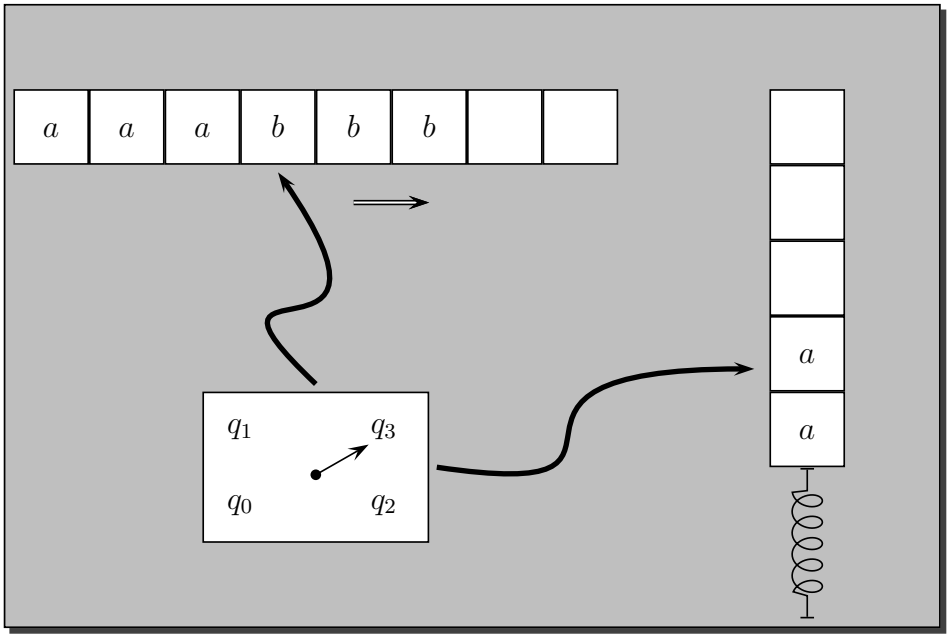


Figura 4.6: Un autómata de pila tras empezar a leer la subcadena *bbb*.

podemos delimitar perfectamente las capacidades de cómputo de los autómatas: si no tienen pila, son estrictamente equivalentes a los sistemas regulares; si tienen una, son estrictamente equivalentes a los sistemas independientes del contexto. En el próximo capítulo veremos qué ocurre si añadimos más pilas. De momento, nos limitaremos a observar que con una sola pila es imposible reconocer el lenguaje $a^n b^n c^n$, ya que, cuando el autómata ha borrado todas las as de la pila, también ha leído ya todas las bs y, por tanto, con la pila vacía, no es capaz de contar también las cs; evidentemente, por tanto, el lenguaje $a^n b^n c^m$ sí es independiente del contexto.

Ahora ya podemos dar la definición formal de autómata de pila, aunque antes debemos hacer una consideración previa de la que depende la forma de la definición formal. Existen dos maneras de definir el lenguaje aceptado por un autómata de pila. La primera consiste en definir el lenguaje aceptado como el conjunto de todas aquellas cadenas para las cuales una secuencia de operaciones del autómata lleva a un estado en el que la pila está vacía. La segunda manera de definir el lenguaje aceptado es parecida a la que utilizamos para definir el lenguaje aceptado por AEF, es decir, fijamos un número finito de estados finales y definimos el lenguaje aceptado como el conjunto de aquellas cadenas que conducen al autómata a un estado final. Ambas son equivalentes, pero en la definición formal asumiremos que el autómata posee estados finales (que serían innecesarios si asumiéramos la primera definición de aceptación).

Definición 4.4 (Autómata de pila)**.** Un *autómata de pila* M es una séptupla $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ tal que

1. Q es un conjunto finito de *estados*.
2. Σ es un alfabeto, que designaremos como *alfabeto de entrada*.
3. Γ es un alfabeto, que designaremos como *alfabeto de la pila*.
4. $q_0 \in Q$ es el *estado inicial*.
5. $Z_0 \in \Gamma$ es un símbolo especial de la pila que denominamos *símbolo inicial*.
6. $F \subseteq Q$ es el conjunto de *estados finales*.
7. δ es una función que proyecta $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ en subconjuntos finitos de $Q \times \Gamma^*$.

Dada esta definición, el autómata puede ejecutar dos tipos de acciones. Las del primer tipo tienen la forma

$$\delta(q, a, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_m, \gamma_m)\}$$

donde q y p_i, $1 \leq i \leq m$, son estados, $a \in \Sigma$, Z es un símbolo de la pila y $\gamma_i \in \Gamma^*$, $1 \leq i \leq m$. Este tipo de acciones debemos interpretarlo en los siguientes términos: si el autómata se halla en el estado q, con a en la cinta y Z es el primer símbolo de la pila, entonces, para cualquier i, puede entrar en el estado p_i, sustituir el símbolo Z por la cadena γ_i y desplazar el cabezal una celdilla hacia la derecha.

Las del segundo tipo tienen la forma

$$\delta(q, \epsilon, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_m, \gamma_m)\}$$

cuya interpretación es: en el estado q, independientemente de cuál sea el símbolo de la cinta cuando Z es el primer símbolo de la pila, el autómata puede entrar en el estado p_i y reemplazar Z por γ_i. En este caso, el cabezal no se mueve.

El autómata es, por tanto, nodeterminista en la medida en que en las acciones del primer tipo, $\delta(q, a, Z)$ tiene como valor un conjunto con más de un elemento y en las acciones del segundo tipo, $\delta(q, \epsilon, Z)$ tiene como valor un conjunto que no es vacío. Tan pronto como estas dos situaciones no se dan, es decir, en las acciones del primer tipo el valor de la función es un conjunto con como máximo un elemento y, en las del segundo tipo, la función tiene como valor el conjunto vacío, entonces el autómata será determinista. Lo interesante del caso es que, a diferencia de lo que

ocurre con los AEF, la versión determinista y la nodeterminista del autómata de pila *no son equivalentes*. Efectivamente, los lenguajes *independientes del contexto deterministas* son una familia de lenguajes que se sitúa justo entre la clase de los lenguajes regulares y los lenguajes independientes del contexto propiamente dichos; nuestro viejo conocido, L_4 es uno de ellos. En cambio $L_6 = \{ww^R \mid x \in \{a, b\}^*\}$, por ejemplo, no lo es, porque el autómata nunca puede saber donde se halla la frontera entre las dos subcadenas y, por tanto, no puede tomar una decisión determinista sobre cuándo hay que dejar de introducir símbolos en la pila y hay que empezar a borrarlos; la variante $L_{6'} = \{wcw^R \mid x \in \{a, b\}^*\}$ vuelve a ser determinista, ya que tenemos perfectamente identificado el centro de la cadena con el símbolo c.

Y con esto concluimos nuestra presentación de las dos familias más importantes, por conocidas y estudiadas, de lenguajes formales. Ambas familias definen, además, un espacio que, desde el punto de vista de la complejidad computacional, marca la frontera entre los procedimientos computacionales eficientes y aquellos que no lo son necesariamente. Este es el motivo por el cual, por ejemplo, todos los lenguajes de programación están construidos de tal modo que su complejidad estructural nunca exceda la de un sistema independiente del contexto (y, si puede ser determinista, mejor). A partir de aquí, penetramos en un territorio peor explorado, pero no por ello menos relevante, especialmente para la lingüística.

Capítulo 5

Más allá de la independencia del contexto

Cuando Chomsky estableció la jerarquía de lenguajes formales que lleva su nombre, estableció también la bien conocida relación entre los lenguajes de Tipo 3 (regulares), los lenguajes de Tipo 2 (independientes del contexto), los lenguajes de Tipo 1 (sensibles al contexto) y los lenguajes de Tipo 0 (recursivamente enumerables), más la clase de los lenguajes recursivos, como una clase dentro del Tipo 0, pero solo un poco por encima del Tipo 1 (figura 2.1).

Evidentemente, para cada clase Chomsky definió también un tipo específico de gramática capaz de generar solo lenguajes dentro de esa clase y demostró que bastaba con relajar ciertas restricciones sobre el formato de las reglas para ir subiendo peldaños dentro de la jerarquía; en el cuadro 5.1 tenemos un resumen de esa tipología de reglas.

Chomsky también definió los correspondientes autómatas para cada clase. Ya hemos tenido oportunidad de conocer los AEF y los autómatas de pila, como modelos computacionales para los sistemas regulares y los sistemas independientes del contexto, respectivamente. El modelo computacional equivalente a los sistemas de Tipo 0 es la Máquina de Turing, un autómata formalizado a mediados de la década de los treinta del siglo pasado por el matemático británico Alan Turing, mientras que el modelo para los sistemas de Tipo 1 fue formalizado por Chomsky como una Máquina de Turing a la cual se imponían ciertas restricciones de operatividad en cuanto a espacio de trabajo, de ahí que se conozca con el nombre de *autómata linealmente acotado*. De las máquinas de Turing nos ocuparemos en la parte dedicada a la complejidad computacional, pero del autómata linealmente acotado no tendremos nada que decir aquí. El motivo, como veremos en la próxima sección, es que este modelo computacional ha quedado obsoleto y ha sido sustituido por modelos basados en el autómata de pila simple que describimos en

Tipo de lenguaje	Restricciones sobre las reglas
Recursivamente enumerable	$x \to y$, donde x e y son cadenas de terminales y no terminales.
Sensible al contexto	$x \to y$, $\|x\| \leq \|y\|$, donde x e y son cadenas de terminales y no terminales.[a]
Independiente del contexto	$A \to y$, donde A es un terminal e y es una cadena de terminales y no terminales.[b]
Regular	$A \to wB, A \to w$, donde A y B son no terminales y w es una cadena de terminales.

[a] Esta restricción define, de hecho, las reglas para gramáticas capaces de generar cualquier lenguaje recursivo. Las gramáticas sensibles al contexto añadirían la restricción adicional de tener la forma $xAy \to xwy$, es decir, que el no terminal A debe reescribirse como w en el contexto definido por las subcadenas x e y, que deben permanecer constantes, de ahí el término gramática sensible al contexto.

[b] Es decir, una regla independiente del contexto es el resultado de imponer que las subcadenas que definen el contexto en una regla sensible al contexto sean nulas. Además, toda gramática independiente del contexto puede normalizarse de manera que todas las reglas sean de la forma $A \to BC$ o $A \to a$, donde A, B y C son no terminales y a es un único terminal. Las gramáticas con este formato se dice que están en la Forma Normal de Chomsky.

Cuadro 5.1: Tipología de reglas de la gramática en la Jerarquía de Chomsky.

el capítulo dedicado a los lenguajes independientes del contexto. Buena parte de estos desarrollos tienen mucho que ver con las investigaciones en el campo de la complejidad estructural del lenguaje humano llevadas a cabo al hilo de los trabajos del propio Chomsky y que eventualmente culminaron con el desarrollo de las gramáticas transformacionales. En esencia, el debate se centró sobre la cuestión de en qué medida el lenguaje natural sobrepasaba los límites de complejidad definidos por la clase de los sistemas independientes al contexto y hasta qué punto las gramáticas transformacionales fijaban ese límite mucho más allá de donde realmente se encuentra. El debate culminó a mediados de la década de 1980 con la definición de un nuevo espacio dentro de la Jerarquía de Chomsky caracterizado por lo que hoy se conoce como *sensibilidad al contexto moderada*.

5.1. Sensibilidad al contexto moderada

En 1957, Chomsky concluyó que los sistemas independientes del contexto no eran adecuados para dar cuenta de toda la complejidad estructural de las lenguas naturales.[1] Concretamente, la conclusión de Chomsky apuntaba a la idea de que, si bien una gramática independiente del contexto sería suficiente para generar lo que se conocía con el nombre de *estructuras de la base*, otras estructuras, más complejas, como las dependencias a larga distancia solo podían generarse aplicando unas reglas especiales, las reglas transformacionales, sobre las estructuras de la base a fin de derivar las primeras a partir de las segundas. No tardaron en surgir voces contrarias a esta idea y favorables a explorar los límites verdaderos de la independencia del contexto antes de dar el salto cualitativo y cuantitativo que suponía la adopción del componente transformacional.[2] Sin embargo, estas críticas tuvieron una repercusión mínima o nula y no fue hasta principios de la década de 1980 en que podemos decir que se reabrió el caso gracias a los trabajos de Gerald Gazdar.[3] Gazdar siempre tuvo la precaución de afirmar que de los fenómenos gramaticales descritos hasta la fecha, ninguno superaba los límites de la independencia del contexto, lo cual no significaba que estos fenómenos no existieran. Quizá espoleados por las observaciones de Gazdar, algunos lingüistas se embarcaron en la empresa

[1] Noam Chomsky. 1957. *Syntactic Structures*. Mouton: La Haya; especialmente el capítulo 5.

[2] Por ejemplo: Gilbert H. Harman. 1963. "Generative grammars without trnaformation rules. A defense of phrase structure". *Language* 39(4):597–616.

[3] En particular, son dos los trabajos más relevantes:

- Gerald Gazdar. 1981. "Unbounded dependencies and coordinate structure". *Linguistic Inquiry* 12:155–184.
- Gerald Gazdar. 1982. "Phrase structure grammar". En P. Jacobson & G. K. Pullum (eds), *The Nature of Syntactic Representation*. Reidel: Dordrecht, pp. 131–186.

de localizar esos fenómenos gramaticales que pusieran a prueba los límites de la independencia del contexto y, hacia finales de los ochenta, ya se habían descrito las famosas dependencias cruzadas del suizo-alemán[4] y del neerlandés,[5] así como ciertas complejidades morfológicas de la lengua bambara.[6] Hoy en día ya sabemos, por tanto, que Chomsky estaba en lo cierto, pero solo parcialmente, ya que el límite de complejidad estructural del lenguaje parece hallarse solo un poco por encima de la independencia del contexto, aunque ya dentro del espacio de los sistemas sensibles al contexto. Este espacio donde se hallaría el lenguaje es lo que Aravind Joshi ha caracterizado como sensibilidad al contexto moderada.[7] Según Joshi, el espacio definido por la sensibilidad al contexto moderada contiene una clase $\mathfrak{L}$ de lenguajes que se caracterizan por poseer las propiedades siguientes:

1. $\mathfrak{L}$ contiene a todos los lenguajes independientes del contexto.
2. $\mathfrak{L}$ contiene lenguajes que toleran una cantidad limitada de dependencias cruzadas.
3. Los lenguajes en $\mathfrak{L}$ se pueden procesar en tiempo polinómico, es decir, $\mathfrak{L} \subset$ **TIEMPOP**.
4. Los lenguajes en $\mathfrak{L}$ poseen *la propiedad del crecimiento constante.*

Las dos primeras propiedades se explican por sí mismas. La tercera se comprenderá mejor cuando nos ocupemos de complejidad computacional, pero viene a decirnos que el tiempo máximo que se tardaría en procesar una cadena del lenguaje se puede definir mediante un polinomio expresado en función de la longitud n de la cadena, como por ejemplo n^3, es decir, que el tiempo máximo necesario para procesar una cadena será igual al cubo de su longitud. Finalmente, la cuarta propiedad nos viene a decir que si ordenáramos todas las cadenas en función de su longitud, observaríamos que la longitud va creciendo de forma lineal.

En este capítulo no entraremos en detalle en cuestiones formales como hemos hecho en los anteriores e intentaremos ofrecer una caracterización intuitiva los suficientemente comprensible de las diferentes cuestiones y problemas que se plantean.

[4] Stuart M. Shieber. 1985. "Evidence against context-freeness of natural language". *Linguistics & Philosophy* 8:333–343.

[5] Joan W. Bresnan, Ronald M. Kaplan, P. Stanley Peters & Annie Zaenen. 1982. "Cross-serial dependencies in Dutch". *Linguistic Inquiry* 13:613–635.

[6] Christopher Culy. 1985. "The complexity of the vocabulary of Bambara". *Linguistics & Philosophy* 8:345–351.

[7] Aravind K. Joshi. 1985. "Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions?". En D. R. Dowty, L. Karttunen & A. M. Zwicky (eds), *Natural Language Parsing: Psychological, Computational, and Theoretical Perspectives.* Cambridge University Press: Cambridge, pp. 206–250.

El principal de ellos es que en la investigación en este campo con frecuencia se han tratado simultáneamente dos cuestiones cruciales para la lingüística como son el problema de la capacidad generativa fuerte y el de la capacidad generativa débil. Como consecuencia de ello, se han desarrollado numerosos formalismos gramaticales diferentes desde el punto de vista de las descripciones estructurales que estos son capaces de asignar a una frase, pero equivalentes desde el punto de vista de la capacidad generativa débil.[8] Como el asunto principal de este trabajo no es el de la adecuación descriptiva de las gramáticas sino el de los requisitos mínimos que una gramática debe tener para poder tratar la complejidad estructural de las lenguas naturales, aquí nos limitaremos a dotar de mayor contenido a las propiedades que enumerábamos arriba, con especial atención en la segunda, que es la que mayores repercusiones tiene para la cuestión de la adecuación descriptiva. Lo importante es comprender que, independientemente de la descripción estructural que queramos asignarle a una construcción con dependencias cruzadas del neerlandés, el formalismo que utilicemos deberá, como mínimo, ser capaz de generar débilmente ese tipo de dependencias y, para ello, deberá incorporar algunas propiedades básicas de las gramáticas formales capaces de generar lenguajes moderadamente sensibles al contexto. Lo que nos lleva a hablar de los Linear Context-Free Rewriting Systems.

5.1.1. Linear Context-Free Rewriting Systems

En 1974, el matemático Nabil Khabbaz tuvo la intuición de que las transiciones de un espacio a otro dentro de la Jearquía de Chomsky podían expresarse de forma más generalizada de lo que lo hizo Chomsky en su momento, al menos en todo lo que a los lenguajes de Tipo 1 (y los que esta clase incluye) se refiere.[9] Dicha intuición halló su formalización definitiva en la década de 1990 gracias al trabajo de David Weir, quien desarrolló las propuestas de Khabbaz sobre lenguajes ampliándolas tanto en el campo de las gramáticas[10] como en el de los autómatas.[11]

Los detalles formales son complejos, pero la intuición es muy simple y pasa por recordar algo que señalamos en relación con los lenguajes independientes del contexto y que denominamos su «número mágico», en este caso el 2. Recordemos que 2, en un lenguaje independiente del contexto, es el número de subcadenas que estan relacionadas entre sí pero de forma discontinua; alternativamente, podemos

[8] Véase al respecto: Aravind K. Joshi, K. Vijay-Shanker & David Weir. 1991. "The convergence of mildly context-sensitive grammar formalisms". En P. Sells, S. M. Shieber & T. Wasow (eds), *Foundational Issues in Natural Language Processing*. MIT Press: Cambridge, MA, pp. 31-81.

[9] Nabil A. Khabbaz. 1974. "A geometric hierarchy of languages". *Journal of Computer and System Sciences* 8:142–157.

[10] David Weir. 1992. "A geometric hierarchy beyond context-free languages". *Theoretical Computer Science* 104:235–261.

[11] David Weir. 1994. "Linear iterated pushdowns". *Computational Intelligence* 10:431–439.

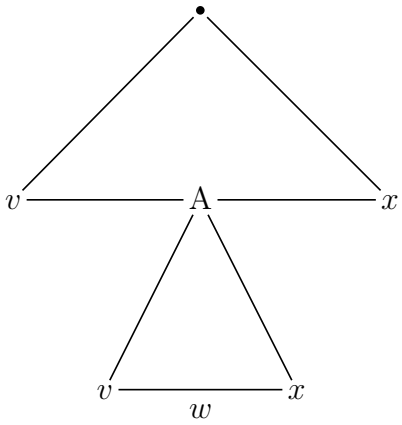


Figura 5.1: Abanico de terminales igual a 2.

decir que 2 es el número de «columnas» que mantienen una dependencia mutua, como ocurre en L_4 y en otros lenguajes de la familia que hemos encontrado. En una gramática, esto se traduce en el hecho de que hay reglas con el mismo no terminal a izquierda y derecha de la flecha, tal que el segundo no terminal se interpone entre dos terminales. Es decir, que lo que denominaremos *abanico de terminales* en una gramática independiente del contexto nunca es superior a 2 (figura 5.1).

¿Es posible formalizar un tipo de reglas cuyo abanico de terminales sea superior a 2? La respuesta es sí y en este punto radica el interés del trabajo de Weir, quien demostró que es posible tener un formato de regla no muy distinta de la típica regla de gramática independiente del contexto con un único no terminal a la izquierda de la flecha, pero con abanicos de terminales superiores a 2. La clave, por tanto, está en que la parte derecha de la regla introduzca un terminal que cree una discontinuidad entre, por ejemplo, tres subcadenas de terminales de tal modo que, entonces, podamos establecer una dependencia de tres columnas como la que observamos en el lenguaje $L_{10} = \{a^n b^n c^n\}$, que es el que se correspondería con las dependencias cruzadas del neerlandés, por ejemplo. En este caso, por tanto, tendremos un abanico de terminales igual a 3, de tal modo que, con la aplicación recursiva de la misma regla conseguiremos poner en relación tres cadenas que crecerán al unísono y siempre tendrán la misma longitud (figura 5.2).

Las gramáticas formales que amplían el formato típico de las gramáticas independientes del contexto para generar abanicos de terminales superiores a 2 se denominan *Linear Context-Free Rewriting Systems* (LCFRS; que podríamos traducir como «Sistemas de Reescritura Independientes del Contexto Lineales»). Son independientes del contexto en el sentido de que las reglas solo contienen un no terminal a su izquierda y lineales porque su abanico de terminales crece linealmente a medida que progresa la derivación de la cadena. Evidentemente, los lenguajes que estos sistemas generan *no* son independientes del contexto en cuanto a su complejidad estructural se refiere (ya hemos visto que L_{10} no lo es), pero se pueden

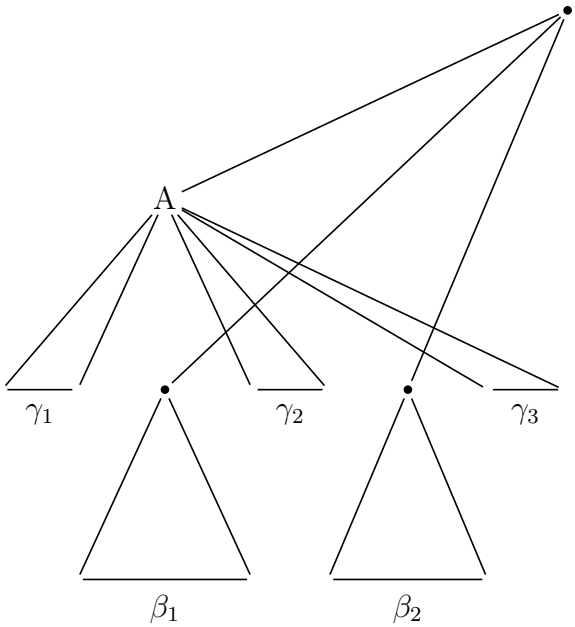


Figura 5.2: Abanico de terminales igual a 3.

generar con unas reglas independientes del contexto ampliadas. La idea es, por tanto, que el formato de regla independiente del contexto es generalizable para dar cuenta de familias diferentes de lenguajes con una complejidad incluso mayor que la de los lenguajes independientes del contexto propiamente dichos. En este sentido una gramática independiente del contexto no es más que un 2-LCFRS, es decir, un LCFRS con abanico de terminales igual a 2. Técnicamente, por tanto, un sistema regular es un 1-LCFRS, ya que sus reglas solo gestionan una única columna de terminales, mientras que un sistema capaz de dar cuenta de dependencias cruzadas es un 3-LCFRS. Nótese, por tanto, que el número que prefijamos determina el abanico de terminales *máximo* que puede gestionar la gramática de tal modo que se puede establecer la relación de inclusión de lenguajes siguiente:

$$\text{1-LCFRL} \subset \text{2-LCFRL} \subset \text{3-LCFRL} \subset \ldots \subset k\text{-LCFRL}, k \in \mathbb{N}.$$

De donde se deduce que un 2-LCFRS no puede generar un 3-LCFRL, pero sí al revés. Vemos, pues, que la generalización del formato de regla independiente del contexto es suficiente para definir una estructura mucho más precisa dentro de la Jerarquía de Chomsky, al menos, como ya apuntábamos, dentro del espacio de los lenguajes de Tipo 1. Existen lenguajes cuya complejidad estructural supera con creces las capacidades de un LCFRS, pero está por ver que esos lenguajes incorporen alguna propiedad hasta ahora desconocida que también se observe en las lenguas naturales. Es por este motivo que hoy en día se perciban los LCFRS como

la clase que caracteriza de forma precisa la sensibilidad al contexto moderada.[12]

¿Tiene esta genaralización de las gramáticas independientes del contexto una traducción al campo de los autómatas? Sí, también. Basta con preguntarse, ¿Cuántas pilas se necesitan para reconocer un lenguaje independiente del contexto? Una. ¿Y para reconocer un lenguaje regular? Ninguna. Por tanto, si el abanico de terminales del lenguaje es k, el número de pilas necesarias para reconocerlo es $k-1$. Es decir, que podemos generalizar el autómata de pila a los diferentes casos que nos interesan, incrementando el número de pilas de memoria que este gestiona, *sin necesidad de introducir ninguna modificación estructural más*. Para ver que esto es efectivamente así, basta con asumir que la única restricción sobre el funcionamiento de un autómata de pila ampliado con un número de pilas igual a k es que no podrá usar la pila k hasta que haya vaciado la pila $k-1$, momento en el cual esta queda inutilizada para lo que queda de computación.[13] Con esto, podemos ver que un autómata con dos pilas podrá reconocer una cadena de L_{10} sin problemas. Efectivamente, cuando el autómata haya vaciado la primera pila de *a*s, señal de que ya habrá localizado la subcadena de *b*s, podrá empezar a llenar la segunda pila con *b*s, que quedará vacía cuando haya procesado todas las *c*s. Si tuviéramos un lenguaje con una columna más, es decir con abanico de terminales igual a 4, necesitaríamos un autómata con tres pilas y, así, sucesivamente.

5.2. Coda

Con este breve repaso de algunas de las principales propiedades de los LCFRS concluimos nuestro examen de las principales familias de lenguajes formales con alguna relevancia lingüística. Evidentemente, la galaxia de lenguajes interesantes (figura 1.2, página 12) contiene muchas más clases de lenguajes. Algunas de ellas han merecido la atención de los investigadores por su posible relevancia para la lingüística, como, por ejemplo, los lenguajes indexados lineales[14] o los lenguajes indexados globales.[15] Sin embargo, aunque no podemos descartar que la complejidad estructural del lenguaje humano vaya más allá de la sensibilidad al contexto moderada, los argumentos presentados hasta el momento en favor de esta posibilidad

[12]Por ejemplo, un lenguaje que escaparía al poder expresivo de la sensibilidad al contexto moderada (y, por tanto, al de los LCFRS) sería el lenguaje $L_{mix} = \{w \mid w \in \{a,b,c\}^* \text{ y } |a|_w = |b|_w = |c|_w \geq 1\}$. Es decir, un lenguaje que contenga el mismo número de *a*s, *b*s y *c*s pero mezcladas dentro de la cadena w.

[13]El motivo de esta restricción es el mismo por el cual no permitimos que el autómata de pila simple acceda a otras celdillas de la pila aparte de la que contiene el primer símbolo de la pila.

[14]Gerald Gazdar. 1988. "Applicability of indexed grammars to natural language". En U. Reyle & C. Rohrer (eds), *Natural Language Parsing and Linguistic Theories*. Reidel: Dordrecht, pp. 69–94.

[15]José M. Castaño. 2004. *Global Indexed Languages*. Tesis doctoral, Brandeis University.

descansan en evidencias puramente circunstanciales y relativamente poco sólidas, como el sistema numérico del mandarín[16] o la absorción de caso (*Suffixaufnahme*) en georgiano antiguo.[17]

Sea como sea, lo que todos estos trabajos nos demuestran es que la teoría de lenguajes formales puede ser una herramienta muy útil en el campo de la investigación lingüística. No la definitiva ni la única, desde luego, porque, como ya hemos señalado en algún momento, los lenguajes formales solo nos pueden servir de modelos abstractos para caracterizar determinadas propiedades estructurales de las lenguas. En ningún momento, por ejemplo, serán el sustituto del trabajo descriptivo y teórico encaminado a buscar la adecuación descriptiva de esos fenómenos ni, tampoco, de la inevitable tarea de diseñar formalismos adecuados para llevar a cabo esas tareas de descripción. En este sentido, es muy importante tener en cuenta una cuestión sobre la cual nunca está de más repetirse: una cosa es la capacidad generativa débil y otra, muy distinta, la capacidad generativa fuerte. La teoría de lenguajes formales nos ofrece, principalmente, resultados sobre la primera, lo cual nos permite establecer un umbral mínimo de complejidad para el fenómeno en cuestión (por ejemplo, las dependencias cruzadas). Ello no significa que esa sea la complejidad real del fenómeno, ya que, dependiendo de la representación estructural que queramos asignar en cada caso, puede darse la circunstancia de que necesitemos un sistema con una complejidad mayor. Esto se entiende perfectamente si pensamos en el árbol de derivación para una cadena de L_4 como el que tenemos en la figura 4.1 de la página 52, con una dependencia anidada. Podría darse el caso, sin embargo, que la descripción estructural que quisiéramos asignar a esa misma cadena fuera la que tenemos en la figura 5.3.

Ninguna gramática independiente del contexto es capaz de efectuar una derivación de este tipo; nótese que el árbol de 5.3 en ningún caso es un árbol de bombeo legítimo. De hecho, tampoco una gramática moderadamente sensible al contexto podría hacerlo, ya que la derivación no sigue la pauta que daría lugar a un abanico de terminales como los que caracterizan las derivaciones en un LCFRS. Aquí tenemos algo muy distinto, quizá ya dentro del espacio de los lenguajes indexados o, en todo caso, si existe algún sistema moderadamente sensible al contexto capaz de generar un árbol de este tipo, sus propiedades son muy distintas de las de los LCFRS.

La delimitación de un umbral mínimo de complejidad es, sin embargo, un dato muy útil, porque automáticamente excluye todos aquellos sistemas que se hallan por debajo de ese umbral; en el caso del lenguaje, los sistemas regulares y los siste-

[16]Daniel Radzinski. 1991. "Chinese number-names, tree-adjoining languages, and mild context-sensitivity". *Computational Linguistics* 17:277–299.

[17]Jens Michaelis & Marcus Kracht. 1997. "Semilinearity as a syntactic invariant". En C. Retoré (ed), *Logical Aspects of Computational Linguistics*. Springer: Berlín & Heidelberg, pp. 329–345.

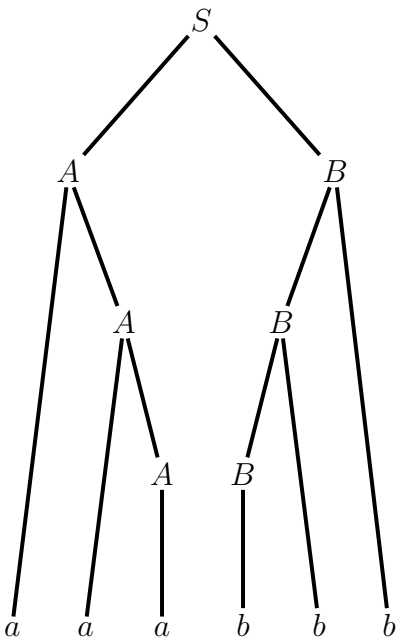


Figura 5.3: Árbol de derivación en dos ramas para la cadena *aaabbb*.

mas independientes del contexto. Todo lo que puede hacer un sistema regular (la fonología, por ejemplo) lo puede hacer un sistema independiente del contexto (buena parte de la sintaxis); y todo lo que puede hacer un sistema independiente del contexto lo puede hacer también un sistema moderadamente sensible al contexto (lo que queda de la sintaxis). Por tanto, si pensamos en el sistema computacional responsable de efectuar las computaciones del lenguaje, ese sistema tiene las capacidades, como mínimo, de un sistema moderadamente sensible al contexto. Lo que no significa, como hemos visto, que pueda ser más potente. En este sentido, como ya señalamos en el capítulo dedicado a los sistemas regulares, la teoría de los lenguajes formales también puede ser un instrumento muy útil en el campo de lo que podríamos denominar «cognición comparada», ya que el análisis comparativo de ciertas conductas animales nos puede revelar muchas cosas sobre nuestra propia cognición, especialmente si tomamos como referencia el lenguaje, sobre el cual disponemos ya de mucha información.

En este campo existen numerosos estudios como los que comentamos en su momento sobre el canto de ciertas aves o los sistemas de llamadas de ciertos primates. Estos estudios siguen un procedimiento similar al que se sigue en el caso del lenguaje, identificando patrones estructurales y estableciendo su grado de complejidad en relación con algún lenguaje formal. Hasta el momento, nadie ha sido capaz de hallar conductas cuya complejidad estructural excediera la complejidad de un sistema regular, probablemente porque la mayoría de estos estudios han centrado su atención en las conductas comunicativas, partiendo del supuesto, pro-

bablemente erróneo, que es ahí donde hay que buscar si queremos saber algo sobre el lenguaje humano. Es evidente que la cognición comparada debe ocuparse de las conductas comunicativas, pero es importante que esta amplíe sus horizontes a otras conductas que, ¡quién sabe!, a la postre resultan más reveladoras. No siempre es posible realizar un análisis formal de una conducta a fin de desvelar la complejidad estructural que esta encierra, pero no por ello debemos descartar a priori la potencial relevancia de este tipo de estudios. El sesgo que pueden adquirir este tipo de investigaciones depende en buena medida de hasta qué punto uno esté dispuesto a aceptar la especificidad (humana o lingüística) del sistema computacional del lenguaje. Existen dudas más que razonables de que dicha especificidad sea real,[18] lo que refuerza la idea de que el estudio de la cognición comparada resulte realmente fructífero. Especialmente si complementamos nuestra colección de instrumentos de trabajo con algunos conocimientos de la teoría de la complejidad computacional.

[18]Sergio Balari & Guillermo Lorenzo. 2013. *Computational Phenotypes. Towards an Evolutionary Developmental Biolinguistics*. Oxford University Press: Oxford.

Parte III

Complejidad computacional

Capítulo 6

Máquinas de Turing y decidibilidad

En 1928, el célebre matemático David Hilbert, en el marco del VIII Congreso Internacional de Matemáticos, celebrado en Bolonia, planteó tres preguntas cuyas respuestas en pocos años removerían por completo los cimientos de la disciplina. La tres preguntas eran, a saber:

1. ¿Son *completas* las matemáticas?
2. ¿Son *coherentes* las matemáticas?
3. ¿Son *decidibles* las matemáticas?

Las tres preguntas venían a ser una reformulación del Segundo Problema de Hilbert, que el propio Hilbert formuló junto con una lista de veintidós más durante el II Congreso Internacional del Matemáticos, celebrado en París en 1900, y que se preguntaba sobre la consistencia de los axiomas de la aritmética. Hasta la fecha de la formulación de las tres preguntas, nadie había sabido dar respuesta al Segundo Problema de Hilbert, pero no deberían pasar muchos años hasta que la primera de las preguntas recibiera una respuesta negativa. Efectivamente, en 1931, Kurt Gödel publicó la demostración de sus dos teoremas de la incompletitud, propinando así el primer golpe mortal al Programa de Hilbert de desarrollar una matemática basada en un sistema finito y consistente de axiomas. Entre 1935 y 1937, de forma casi simultánea e independiente, Alonzo Church en los EE. UU. y Alan Turing en el Reino Unido, publican sus trabajos donde demuestran que la tercera pregunta de Hilbert, el *Entscheidungsproblem* o problema de la decisión, no tiene solución.

Esta breve introducción histórica viene a cuento porque tanto Church como Turing, basándose en el trabajo previo de Gödel, se vieron en la necesidad de definir formalmente el concepto de *algoritmo*. Church desarrolló la explicación de qué significa que una función sea *efectivamente calculable* sobre la base del cálculo

lambda, que él mismo había desarrollado. Turing, por su parte, ideó un modelo distinto basado en la idea de *computación por una máquina*, lo que, con el tiempo, vendría a denominarse *Máquina de Turing*. En su trabajo, Church demostró que no existe ningún algoritmo capaz de demostrar la equivalencia de dos expresiones en cálculo lambda; Turing demostró que dada una Máquina de Turing M y una cadena w es imposible determinar si M se detendrá en un número finito de pasos usando w como entrada, lo que se conoce como *el Problema de la Parada*. Ambas demostraciones resultaron ser equivalentes y se resumen en la afirmación de que existen funciones que no son computables, cuyo corolario, tomando como referencia el modelo de Turing, es la conjetura de que la Máquina de Turing define los límites de aquello que es computable y que no existe ningún dispositivo más potente que este: superar los límites que impone la Máquina de Turing es superar los límites de la computabilidad. Esto es lo que se conoce con el nombre de *Tesis de Church-Turing* que conjetura que una función es algorítmicamente computable si, y solo si, es computable por una Máquina de Turing. Nótese que una cosa es la demostración efectiva de que existen funciones no computables y otra cosa muy distinta es la conjetura de que el límite de la computabilidad es el que impone la Máquina de Turing. Lo segundo no se ha demostrado nunca, aunque la mayoría de los matemáticos considera que la conjetura es cierta y la aceptación de la veracidad de dicha conjetura es uno de los supuestos fundacionales de la teoría de la complejidad computacional.

6.1. Computabilidad (y también lo contrario)

Se podría decir que Turing, cuando inventó su máquina, inventó también los lenguajes formales, ya que esta desde el principio fue concebida como un dispositivo que operaba sobre cadenas de símbolos dispuestos sobre una cinta. Pero Turing hizo algo más que «inventar» los lenguajes formales, descubrió que dentro del universo que estos ocupan (figura 1.1, página 1.1), hay algunos lenguajes muy especiales que quedan fuera de la galaxia de lenguajes interesantes (figura 1.2, página 12) definida por las capacidades computacionales de la Máquina de Turing. Efectivamente, la demostración de que existen funciones no computables nos dice, por un lado, que existe todo un espacio de lenguajes que quedan dentro de las capacidades de la máquina: los lenguajes recursivamente enumerables y todas las subclases incluidas dentro de esta superclase, que representan todo aquello que es efectivamente computable; pero, por otro lado, la demostración también conlleva la existencia de lenguajes que no son ni recursivamente enumerables, cuya complejidad estructural es tal que ni siquiera un dispositivo tan potente como la Máquina de Turing puede con ellos. En esta sección, nos ocuparemos de dar mayor contenido formal a estos conceptos, lo que nos servirá como punto de partida para

hablar de complejidad.

Lo primero, será presentar una descripción básica de la Máquina de Turing (en adelante, MT); en la figura 6.1 tenemos una.

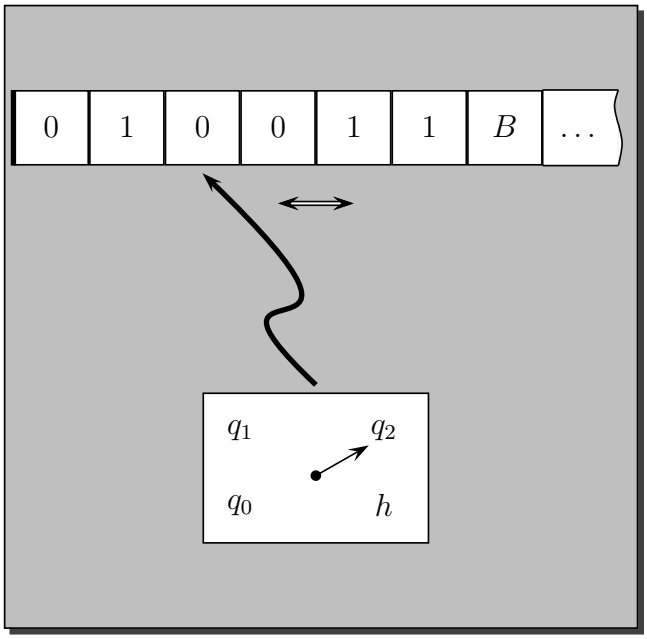


Figura 6.1: Una Máquina de Turing.

A primera vista, una MT no difiere mucho de un AEF: tenemos una unidad finita de control, un cabezal y una cinta con símbolos. Usamos ceros y unos en vez de letras, porque la convención suele ser que las MT leen siempre código binario; el símbolo B es un símbolo especial que indica que la celdilla está en blanco. Las diferencias son pocas, pero significativas. En primer lugar, la cinta está acotada por la izquierda, pero se extiende hasta el infinito por la derecha, de ahí los puntos suspensivos; en segundo lugar, el cabezal, que puede leer un símbolo de una celdilla o sobreescribir un nuevo símbolo sobre el símbolo original (incluido B), puede desplazarse a derecha e izquierda, aunque nunca sobrepasar la cota a la izquierda de la cinta (línea de trazo grueso en 6.1) y, si lo intenta, se quedará «colgada» y se detendrá.[1] Distinguimos, también, uno (o más) estados de parada o finales (h en la figura) en los que entra la MT cuando ha completado la computación, momento en el cual esta se detiene. Este último punto es importante, porque la

[1]El acotamiento de la cinta por la izquierda no es una característica esencial de la definición de MT y, de hecho, existen definiciones que consideran que la cinta se extiende infinitamente por ambos lados, sin que ello afecte las capacidades del sistema. Veremos que en el caso de las MT existen numerosas modificaciones que podemos hacer sobre la configuración básica que, sin embargo, resultan siempre en dispositivos equivalentes.

MT puede detenerse en cualquier momento, independientemente de si ha leído todos los símbolos de la cinta o no.

Pasemos ahora a la definición formal y, a partir de ella, nos ocuparemos con un poco más de detalle del funcionamiento de la MT.

Definición 6.1 (Máquina de Turing)**.** Una Máquina de Turing M es una séptupla $M = (Q, \Sigma, \Gamma, \delta, B, q_0, F)$, donde

1. Q es un conjunto finito de *estados*;
2. Σ es el *alfabeto de entrada*;
3. Γ es el *alfabeto de la cinta*, donde $\Sigma \subset \Gamma$ y $Q \cap \Gamma = \emptyset$;
4. $\delta \in (Q \times \Gamma \to Q \times \Gamma \times \{\mathbb{I}, \mathbb{D}\})$ es la *función de transición*;
5. B es el *símbolo blanco*, tal que $B \in \Gamma$ y $B \notin \Sigma$;
6. $q_0 \in Q$ es el *estado inicial*;
7. $F \subseteq Q$ es el conjunto de *estados finales*.

Nótese que el alfabeto de la cinta Γ incluye el alfabeto de entrada más el símbolo blanco B; el requisito de que $Q \cap \Gamma = \emptyset$ impide que un estado pueda ser también interpretado como un símbolo. Las transiciones de la MT se definen a través de la función de transición δ que toma como valores pares ordenados formados por un estado $q \in Q$, que es el estado en que se encuentra la máquina en ese momento, y un símbolo $X \in \Gamma$, que es el símbolo que está leyendo el cabezal. La imagen de δ es una tripleta (p, Y, D), donde $p \in Q$ es el nuevo estado, $Y \in \Gamma$ es el símbolo que la máquina debe escribir en la celdilla que está leyendo el cabezal y D es una de las dos direcciones $\mathbb{I}$ o $\mathbb{D}$, que indican si el cabezal debe desplazarse a la izquierda o a la derecha. Esta MT es determinista, ya que a cada par ordenado de estados y símbolos, δ le asigna como máximo una tripleta; sin embargo, δ no tiene por qué estar definida para todo su dominio, de tal modo que puede haber pares de estados y símbolos para los cuales no exista una acción definida.

En una MT es muy importante definir con precisión qué es exactamente una *acción* de la máquina. Para ello necesitamos, primero, una definición auxiliar.

Definición 6.2 (Descripción instantánea)**.** La *descripción instantánea* (DI) de una Máquina de Turing M es una cadena de la forma $\alpha_1 q \alpha_2$, donde $q \in Q$ es el estado en que se encuentra M y $\alpha_1 \in \Gamma^*$ es el contenido de la cinta *a la izquierda*

del cabezal y $\alpha_2 \in \Gamma^*$ es el contenido de la cinta *a la derecha del cabezal*. Asumimos que el cabezal apunta al primer símbolo de α_2 o, si $\alpha_2 = \varepsilon$, que el cabezal apunta a un blanco.

Nótese que es esta definición la que nos obliga a evitar que los estados puedan confundirse con símbolos, dada la naturaleza «mixta» de las DI. Con la definición de DI, ya podemos definir las acciones de una MT.

Hay dos tipos de acciones: las que desplazan el cabezal a la izquierda y las que lo desplazan a la derecha. Supongamos que tenemos la DI

$$X_1X_2 \dots X_{i-1}qX_i \dots X_n.$$

Para una acción hacia la izquierda $\delta(q, X_i) = (p, Y, \mathbb{I})$, donde si $i - 1 = n$ entonces $X_i = B$, pueden darse dos circunstancias posibles. La primera, es que $i = 1$, en cuyo caso no hay DI siguiente, ya que el cabezal nunca puede traspasar la cota izquierda de la cinta. La segunda es que $i > 1$, en cuyo caso

$$X_1X_2 \dots X_{i-1}qX_i \dots X_n \underset{M}{\longmapsto} X_1X_2 \dots X_{i-2}pX_{i-1}YX_{i+1} \dots X_n,$$

donde vemos que el símbolo X_i ha sido sustituido por el símbolo Y y, como indica la posición del nuevo estado p, por la definición 6.2, el cabezal se ha desplazado hacia la izquierda y ahora apunta a X_{i-1}.

Para una acción hacia la derecha $\delta(q, X_i) = (p, Y, \mathbb{D})$, tenemos

$$X_1X_2 \dots X_{i-1}qX_i \dots X_n \underset{M}{\longmapsto} X_1X_2 \dots X_{i-1}YpX_{i+1} \dots X_n.$$

Nótese que aquí, en el caso en que $i - 1 = n$, la cadena $X_i \dots X_n$ es vacía y la acción comporta escribir el símbolo Y sobre un blanco y, como consecuencia de ello, la DI de la derecha es más larga que la de la izquierda.

Si una DI J es el resultado de otra DI I tras un número finito de pasos, entonces escribiremos $I \overset{*}{\underset{M}{\longmapsto}} J$. Y, por fin, llegamos a la definición de lenguaje aceptado por una MT.

Definición 6.3. El *lenguaje aceptado* por una MT M, $L(M)$, es el conjunto de cadenas sobre Σ^* que causa la entrada de M en un estado final.[2] Formalmente

$$L(M) = \{w \mid w \in \Sigma^* \text{ y } q_0w \overset{*}{\underset{M}{\longmapsto}} \alpha_1p\alpha_2, \text{tal que } p \in F \text{ y } \alpha_1, \alpha_2 \in \Gamma^*\}.$$

---

[2]Siempre asumimos que la cadena se dispone de modo que su primer símbolo ocupe la primera celdilla empezando por la izquierda de la cinta de M y que M se halla en q_0 con el cabezal sobre la primera celdilla.

De esta definición se desprende que dada una MT M con una función de transición δ_i, si al procesar una cadena w en algún momento se detiene en un estado final, entonces decimos que M ha *aceptado* w; si por el contrario, con la misma función de transición, M se «cuelga» en un estado que no es final, entonces diremos que M ha *rechazado* w. Existe, sin embargo, una tercera posibilidad, concretamente, que M entre en un «bucle» y no llegue nunca a detenerse. De hecho, esta posibilidad también existe en otros autómatas de rango inferior que la MT, especialmente en sus variantes nodeterministas con transiciones espontáneas, pero en el caso de la MT esta situación se corresponde con un caso particularmente interesante. Evidentemente, podemos intentar corregir la función de transición para evitar el bucle, pero es posible demostrar (cosa que no haremos) que hay ciertos lenguajes para los cuales esto no es posible y la MT siempre, tarde o temprano, entrará en un bucle.

Consideremos, de momento, el caso en que tenemos una MT M que siempre se detiene, es decir que, dada una cadena arbitraria w, M siempre acepta o rechaza. En este caso diremos que M es un *algoritmo* o, también, que M es una *MT total* y diremos que el lenguaje $L(M)$ que M acepta es un lenguaje *recursivo*. Supongamos ahora que construimos una MT M' que funciona al revés que M, es decir, que cuando M acepta M' rechaza y viceversa. Nótese que M' es también un algoritmo. ¿Cuál es el lenguaje $L(M')$ que acepta M'? Evidentemente, el complemento $\overline{L(M)}$ de $L(M)$. Por tanto, el complemento de un lenguaje recursivo *es también recursivo*. O, dicho de otro modo, *un lenguaje recursivo es aquel cuyo complemento es también recursivo*.

Supongamos ahora que tenemos una MT M que solo se detiene cuando $w \in L(M)$, pero si $w \in \overline{L(M)}$ en vez de detenerse, entra en un bucle. Por lo tanto, $L(M)$ no es recursivo, solo recursivamente enumerable. Nótese que, en este caso, es imposible construir una MT M' que acepte $\overline{L(M)}$, por lo tanto, este lenguaje no solo no es recursivo, sino que *ni siquiera es recursivamente enumerable*. Es decir, que si el complemento de un lenguaje recursivamente enumerable no es recursivo, este lenguaje tampoco es recursivo, pero su complemento ni siquiera es recursivamente enumerable, es un lenguaje *no computable*.

Para poner un poco de orden en la terminología, diremos que un algoritmo *decide* un lenguaje o, alternativamente, que los lenguajes recursivos son siempre *decidibles*. Si un lenguaje no es decidible, a lo sumo, puede darse la circunstancia de que haya una MT que lo *acepte* y diremos que es *semi-decidible*, en cuyo caso, su complemento no será recursivamente enumerable y, por tanto, tampoco será computable. Evidentemente, existen lenguajes no computables cuyo complemento tampoco lo es.[3]

[3]Quizá no esté de más una breve nota aclaratoria sobre el término *recursivamente enumerable*. De lo dicho hasta el momento, dado un lenguaje recursivamente enumerable, por muy complejo

que este sea, siempre habrá al menos una MT que lo acepte. Esta idea podemos expresarla desde un punto de vista diferente: en vez de diseñar una MT que acepte el lenguaje, podemos diseñar una MT que lo *enumere*, es decir que vaya escribiendo en una cinta todas y cada una de las cadenas que lo componen. Evidentemente, esto solo es concebible desde el punto de vista teòrico, ya que solo las MT disponen de tiempo y espacio infinitos para operar. Nótese que la idea de enumerabilidad guarda una relación muy estrecha con la *Hipótesis del Continuo* formulada por Georg Cantor en 1878 sobre la cardinalidad de los conjuntos infinitos. Como es bien sabido, los conjuntos $\mathbb{N}$, de los naturales, $\mathbb{Z}$, de los enteros, y $\mathbb{Q}$, de los racionales, son conjuntos infinitos, pero los tres tienen el mismo cardinal ya que es posible ponerlos en relación con el conjunto de los números naturales. Estos conjuntos se pueden, por tanto, contar o enumerar: son recursivamente enumerables. Sin embargo, el conjunto $\mathbb{R}$ de los números reales tiene un cardinal mayor que el cardinal de los enteros, por ejemplo. El conjunto de los reales (como el $\mathbb{C}$ de los complejos), no se puede contar, no es recursivamente enumerable. Si, siguiendo la notación de Cantor, designamos como $\aleph_0$ el cardinal de naturales, enteros y racionales y como $\aleph_1$ el cardinal de reales y complejos, la Hipótesis del Continuo afirma que no hay ningún cardinal C tal que $\aleph_0 < C < \aleph_1$.

Capítulo 7

La MT como modelo computacional

La Tesis de Church-Turing comporta que no hay ningún dispositivo computacional más potente que la MT. Por tanto, ninguna modificación que introduzcamos en la construcción básica de la MT de la figura 6.1 y la definición de 6.1 mejorará las capacidades computacionales del nuevo dispositivo. Podemos añadir más cintas, más cabezales, permitir que el dispositivo genere una cadena de respuesta a partir de la cadena de entrada, podemos incluso combinar dos o más MT para que trabajen juntas. Hagamos lo que hagamos, el dispositivo resultante siempre será equivalente a una MT básica. Nada que no pueda computar una MT es computable, por tanto, no hay dispositivo imaginable capaz de computar lo no computable.[1] Ello convierte a la MT no solo en el epítome de la computabilidad, sino también en una potencial medida de la eficiencia con que se puede resolver un problema.

Uno de los aspectos más importantes de la respuesta que desarrolló Turing al Entscheidungsproblem es que cualquier problema de interés puede codificarse en forma de lenguaje y, por tanto, que el proceso de evaluación de su complejidad puede reducirse a un simple problema de decisión que podemos plantear a una MT. Evidentemente, una de las cosas que nos dirá la MT es si el problema es soluble o no, pero, cuando se trata de un problema soluble, lo que entonces nos interesa es cuán difícil de resolver es el problema en cuestión. Este es el objetivo principal de la teoría de la complejidad computacional y la MT es el modelo ideal para evaluar la complejidad de los problemas solubles.

Si nos centramos, pues, en aquellos problemas que son solubles, entonces, dadas las definiciones de decidibilidad y de computabilidad, se sigue que lo que nos interesa es exactamente la familia de lenguajes que son recursivamente enumerables, los mismos que componen la Jerarquía de Chomsky. Este es el punto de

[1] A menos, claro está, que uno crea en la *hipercomputación*; cf. B. Jack Copeland. 2002. "Hypercomputation". *Minds and Machines* 12:461–502.

conexión entre la complejidad estructural y la complejidad computacional, ya que la complejidad estructural de un lenguaje determinará su posición en la jerarquía de complejidad computacional. Como veremos, hay problemas que son solubles, pero son tan difíciles de resolver que resultan ser literalmene *intratables*. La teoría de la complejidad computacional nos sirve para definir donde se halla la frontera que separa los problemas tratables de aquellos que no lo son.

Hay muchas maneras de evaluar la dificultad de un problema, su complejidad, pero la más común en teoría de la complejidad es determinar la cantidad máxima de recursos de espacio y de tiempo que ha necesitado una MT para resolverlo. En seguida nos ocuparemos de ofrecer una caracterización más precisa de qué queremos decir con espacio y con tiempo; de momento, nos limitaremos ha hacer hincapié en el hecho de que la teoría de la complejidad computacional, a fin de fijar una referencia para calcular los recursos utilizados, asume un *modelo* de computación fijo, la MT, aunque no necesariamente un *modo* de computación fijo. Efectivamente, como tendremos oportunidad de comprobar más adelante, el mismo modelo (la MT) operando en modos diferentes (por ejemplo, determinista vs. nodeterminista) puede dar lugar a resultados de complejidad diferentes, ya que los dispositivos nodeterministas son, en este caso, más potentes que los deterministas; de hecho, ciertos problemas para los cuales no existe una solución eficiente en modo determinista, pueden tenerla en modo nodeterminista.

Para definir con precisión los conceptos de *tiempo* y *espacio* en tanto que recursos consumidos por una MT durante una computación, primero necesitamos definir nuestro modelo computacional con precisión. En la definición de 6.1 tenemos la definición básica de MT, pero la que se suele utilizar en teoría de la complejidad es ligeramente distinta, fundamentalmente porque facilita el cálculo de los recursos de tiempo y, sobre todo, de espacio. La principal diferencia con el modelo básico es que la MT que utilizaremos ahora es una MT con múltiples cintas y cabezales, lo cual no supone ninguna diferencia en cuanto a poder de cómputo, pero sí en cuanto a comodidad en el momento de evaluar ciertos resultados.

Definición 7.1 (Máquina de Turing con k-cintas)**.** Aquí, asumiremos que una MT M con k-cintas, $k \geq 1$, es una quíntupla, $M = (Q, \Sigma, \delta, q_0, F)$, donde

1. Q es un conjunto finito de *estados*;
2. Σ es un *alfabeto* que contiene, además, los símbolos B (blanco) y $\triangleright$ (símbolo inicial);
3. δ es la *función de transición*;
4. q_0 es el *estado inicial*;
5. $F = \{sí, no, h\}$ es el conjunto de los *estados finales* de *aceptación*, *rechazo* y *parada*, respectivamente.

Vamos a comentar algunas diferencias entre esta definición y la de 6.1. En primer lugar, ahora tenemos un único alfabeto Σ y no distinguimos entre el de entrada y el de la cinta, aunque seguiremos asumiento que $Q \cap \Sigma = \emptyset$. Los principales motivos son dos: por un lado, ahora vamos a permitir a la MT escribir blancos en las celdillas, es decir, que en determinadas circunstancias, la MT podrá borrar el contenido de una celdilla en alguna de las cintas; por el otro, en vez de asumir que las diferentes cintas están acotadas por la izquierda, simplemente incluimos el símbolo especial $\triangleright$ que marca el inicio de la cadena en cualquiera de las cintas y a la izquierda del cual el cabezal nunca puede pasar; de este modo, en la práctica, siempre tenemos todas las cintas *alineadas* si asumimos además que $\triangleright$ nunca se puede borrar y que en el estado inicial todos los cabezales apuntan hacia él en su cinta correspondiente. Tenemos, además, tres estados finales diferentes: aquel en el que entra la MT cuando acepta el lenguaje (*sí*); aquel en el que entra cuando lo rechaza (*no*); y aquel en el que entra cuando se detiene (h). Este último estado es necesario, porque en determinadas ocasiones, querremos que, más que aceptar o rechazar, la MT nos responda algo en una de las cintas (una cadena definida sobre el alfabeto), de tal modo que cuando haya terminado de escribir la respuesta se detenga sin más. Finalmente, tenemos que considerar una situación especial: que la MT no se detenga y entre en un bucle, lo cual denotaremos con el símbolo $\nearrow$; nótese que este no es un estado, solo una convención gráfica para denotar esta situación especial en que la MT no se detiene. Finalmente, la función δ tenemos que adaptarla al caso de las cintas múltiples y, por tanto, ahora es una función que proyecta $K \times \Sigma^k$ en $(K \cup F) \times (\Sigma \times \{\mathbb{I}, \mathbb{D}, \mathbb{S}\})$, donde $\mathbb{I}$ (izquierda), $\mathbb{D}$ (derecha) y $\mathbb{S}$ (quedarse en su sitio). En general, $\delta(q, \sigma_1, \ldots, \sigma_k) = (p, \rho_1, D_1, \ldots, \rho_k, D_k)$ debe interpretarse así: cuando la MT se halla en el estado q, el cabezal de la primera cinta está leyendo σ_1, el de la segunda cinta σ_2, y así sucesivamente, debe entrar en el estado p escribir ρ_1 en la primera cinta y desplazar su cabezal en la dirección D_1 y, así, para todas las cintas. Como en el estado inicial q_0 todos los cabezales de todas las cintas están leyendo $\sigma_i = \triangleright$, entonces $\rho_i = \triangleright$ (no se puede borrar este símbolo) y $D_i = \mathbb{D}$ (no es posible desplazar el cabezal a la izquierda del símbolo inicial de la cadena). Por tanto, la DI inicial de toda MT con k-cintas es siempre $(q_0, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon)$, donde $\triangleright$ es el símbolo a la izquierda del cabezal y x es la cadena a la derecha del cabezal; es decir, que, al principio, solo la primera cinta incluye la cadena de entrada x, mientras que todas las demás cintas están vacías (ϵ). Para el resto, todo es igual que en la MT básica, con la excepción del caso de aquellas MT de k-cintas que responden algo tras alcanzar el estado h: en este caso, dada una cadena de entrada v, la respuesta será la cadena w que la MT ha escrito en la *última* cinta.

Con la MT de k-cintas como modelo básico, ya podemos definir el tiempo consumido por la máquina durante una computación.

Definición 7.2 (Tiempo consumido por una MT)**.** Si para una MT de k-cintas M y cadena de entrada x, tenemos que

$$(q_0, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \xmapsto[M]{t} (H, w_1, u_1, \ldots, w_k, u_k)$$

donde $H \in F$, entonces el *tiempo que M ha necesitado* para computar x es t; es decir, el tiempo utilizado es simplemente el número de pasos hasta el momento de la parada. Si $M(x) = \nearrow$, entonces el tiempo que M necesita para computar x es ∞.

Con esto, tenemos una definición de tiempo para cada caso particular. Ahora debemos generalizar la definición a *todos los casos posibles* con que se puede encontrar M. Para ello, tomaremos como referencia la longitud de las cadenas que reconoce M y expresaremos una medida generalizada de tiempo en función de esta. Sea pues f una función del conjunto de los enteros positivos en el conjunto de los enteros positivos. Diremos que M *opera en tiempo* $f(n)$ si, por cada cadena de entrada x, el tiempo necesario para procesarla es, a lo sumo, $f(|x|)$, donde $|x|$ es la longitud de la cadena x. La función $f(n)$ es la *cota de tiempo* de M.

Y, ahora, ya podemos definir la noción de *clase de complejidad*.

Definición 7.3 (Clase de complejidad)**.** Supongamos ahora que el lenguaje $L \subset (\Sigma - \{B\})^*$ es decidible por una MT con múltiples cintas que opera en tiempo $f(n)$. Diremos entonces que $L \in$ **TIEMPO**$(f(n))$. **TIEMPO**$(f(n))$ es un conjunto de lenguajes, el conjunto de todos los lenguajes decidibles por una MT de múltiples cintas operando con la cota de tiempo $f(n)$. **TIEMPO**$(f(n))$ es una *clase de complejidad* y los lenguajes que contiene poseen la misma complejidad computacional en relación con el recurso del tiempo requerido por una MT.

Calcular los recursos de espacio utilizados por una MT es un poco más complicado, pero intuitivamente lo que queremos es que *espacio* sea igual a la longitud de la cantidad de cinta visitada por la MT. Para ello, utilizaremos una variante de la MT de k-cintas que denominaremos *MT de k-cintas con entrada y salida*. Esta MT es muy parecida a la de la definición de 7.1, pero introduciremos unas restricciones cuyo efecto será el de convertirla en un dispositivo cuya primera cinta sea de solo lectura y su última cinta sea de solo escritura; es decir, de la primera cinta solo se podrán leer símbolos sin borrar nunca ninguno y en la segunda solo se podrán escribir símbolos pero tampoco borrarlos. De este modo evitamos el engorro de tener que hacer un seguimiento de aquellas celdillas que la MT puede

haber visitado más de una vez en la primera y última cinta (y que deberíamos contar para tener una medida de espacio exacta), ya que, con estas restricciones, esto no ocurrirá nunca. De este modo, las únicas cintas que la MT habrá utilizado efectivamente como *memoria* (el espacio) serán aquellas que no son ni la primera ni la última, que podemos excluir de nuestro cálculo.

Definición 7.4 (Máquina de Turing de entrada y salida)**.** Sea $k > 2$ un entero positivo. Una *MT de entrada y salida* es como una MT con k-cintas con las siguientes restricciones sobre la función δ: siempre que

$$\delta(q, \sigma_1, \ldots, \sigma_k) = (p, \rho_1, D_1, \ldots, \rho_k, D_k),$$

entonces

(a) $\rho_1 = \sigma_1$ y

(b) $D_k \neq \mathbb{I}$. Además,

(c) si $\sigma_1 = B$, entonces $D_1 = \mathbb{I}$.

La restricción (a) tiene el efecto de que el símbolo que se «escribe» al cambiar de estado sea el mismo que había antes de hacerlo y, por tanto, la primera cinta es, a todos los efectos, solo de lectura; la restricción (b) impide que el cabezal de la última cinta se desplace hacia la izquierda, lo que le impide volver a visitar celdillas y, así, es, a todos los efectos, una cinta de solo escritura; finalmente, (c), aunque no imprescindible, nos garantiza que el cabezal de la primera cinta no se «despiste» leyendo los blancos que hay al final de la cadena de entrada. Nótese, además, que la MT tendrá, como mínimo, tres cintas: la de entrada, la de salida y una o más cintas auxiliares intermedias.

Ahora que sabemos que basta con fijarnos en el contenido de las cintas intermedias para calcular el espacio (la memoria) que ha consumido una MT de k-cintas en reconocer un lenguajes, podemos definir *espacio* como la suma de las longitudes de todas las cadenas que han quedado inscritas en las cintas intermedias de la MT al terminar la computación. Formalmente:

Definición 7.5 (Espacio consumido por una MT)**.** Sea M una MT de entrada y salida con k-cintas. El *espacio* consumido por M al procesar la cadena de entrada x cuando

$$(q_0, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \overset{*}{\underset{M}{\longmapsto}} (H, w_1, u_1, \ldots, w_k, u_k),$$

donde $H \in F$, es igual a $\sum_{i=2}^{k-1} |w_i u_i|$. Supongamos ahora que f es una función de $\mathbb{N}$ en $\mathbb{N}$. Diremos que la MT M opera con una cota de espacio $f(n)$ si, para toda cadena de entrada x, M necesita como máximo el espacio $f(|x|)$.

A partir de esta definición, podemos definir a su vez una nueva clase de complejidad, la clase **ESPACIO**$(f(n))$. A esta clase pertenecerán aquellos lenguajes decidibles por una MT de entrada y salida que opere dentro de la cota de espacio $f(n)$.

Una última observación antes de continuar. De lo dicho hasta ahora sobre el recurso de espacio se deduce que este nunca es nulo. Esta conclusión parece, en principio, chocar con la afirmación que hicimos en su momento de que los AEF, a diferencia de todos los demás, no tienen pila de memoria, de lo que parece seguirse que para reconocer un lenguaje regular no se necesita de este recurso. Sin embargo, es muy importante en este punto poner de relieve un matiz entre el concepto de *espacio* tal y como lo estamos usando aquí y el de *pila de memoria*. Nótese que el segundo es a todos los efectos un dispositivo auxiliar al que puede recurrir el autómata para almacenar datos y es, por tanto, equivalente, por ejemplo, a una cinta auxiliar en una MT o, de hecho, a la capacidad que tienen las MT, si solo tienen una cinta, de escribir y borrar símbolos de las celdillas. El espacio, en cambio, viene a ser la cantidad de celdillas que recorre la MT antes de detenerse y, en el caso de que una MT esté decidiendo un lenguaje regular, necesariamente tendrá que inspeccionar algún símbolo de la cadena antes de poder decidir si esta pertenece o no al lenguaje. Lo interesante del caso es que los lenguajes regulares pertenecen a la clase de complejidad **ESPACIO**(k), donde k es un entero. Es decir, que el espacio necesario para reconocer un lenguaje regular dado es siempre una constante y es, por tanto, *independiente de la longitud de la cadena*. Este es el dato que correlaciona realmente con el hecho de que un AEF no posea pila de memoria y que tiene que ver, a su vez, con lo que calificamos como el «secreto» de la regularidad: la posibilidad de identificar siempre una subcadena w dentro de una cadena mayor tal que w también pertenece al lenguaje en cuestión.

7.1. Normalización de funciones

Las funciones que definen las cotas de espacio o tiempo con que opera una MT para decidir un lenguaje que pertenece a una clase de complejidad determinada son a menudo muy complejas. Esto dificulta lo cálculos en el momento de determinar con exactitud valores concretos de espacio o tiempo para instancias específicas de un problema. La manera más habitual de sortear este obstáculo suele ser lo que aquí denominaremos «normalizar» la función. Para ello basta con hallar una función $g(n)$ más simple que podamos poner en relación con la $f(n)$ original.

En seguida veremos qué queremos decir con «poner en relación», de momento es suficiente con saber que $g(n)$ debe ser una función cuyo comportamiento sea bien conocido. Hay numerosas funciones de variable entera cuyo comportamiento es perfectamente predecible incluso para valores muy altos de la variable, desde las más sencillas, como las constantes ($g(n) = c$) o las lineales ($g(n) = n$), hasta otras ligeramente más complejas, como las logarítmicas ($g(n) = \log n$)), las polinómicas ($g(n) = n^c$), c una constante), las exponenciales ($g(n) = c^n$), c una constante) o incluso las factoriales ($g(n) = n!$)).

Evidentemente, para saber qué tipo de función $g(n)$ nos interesa debemos conocer mínimamente el comportamiento de $f(n)$. Para ello, es importante tener claro qué nos dice exactamente $f(n)$ sobre el recurso de espacio o de tiempo. Sabemos que la variable n, la longitud de la cadena, crece siempre de forma lineal y, por tanto, al expresar la cota de espacio o de tiempo en función de n, lo que obtenemos es información sobre el ritmo al que crecen los recursos necesarios a medida que procesamos cadenas de longitud mayor. Por tanto, $f(n)$ suele ser una función creciente y lo que nos interesa saber sobre ella con la mayor precisión posible es su *tasa de crecimiento*. Cuando normalizamos una función $f(n)$ poniéndola en relación con otra función más simple $g(n)$ estamos simplemente diciendo que la tasa de crecimiento de $g(n)$ es igual o muy parecida a la de $f(n)$, de tal modo que basta con fijarse en la primera para tener una idea bastante fiable de cómo se comporta realmente la segunda. Cuando hallamos la $g(n)$ apropiada, pueden ocurrir tres cosas, a saber:

1. Que $f(n)$ crezca a una tasa *igual o inferior* que $g(n)$, en cuyo caso diremos que $f(n)$ es $\mathcal{O}(g(n))$.
2. Que $f(n)$ crezca a una tasa *igual o superior* que $g(n)$, en cuyo caso diremos que $f(n)$ es $\Omega(g(n))$.
3. Que $f(n)$ crezca a una tasa *igual* que $g(n)$, en cuyo caso diremos que $f(n)$ es $\Theta(g(n))$.

Analicemos los tres casos con un poco más de detalle. Nótese que cuando decimos que $f(n)$ es $\mathcal{O}(g(n))$ en cierto modo estamos diciendo que $g(n)$ define una *cota superior* para $f(n)$, ya que por muy rápido que crezca la segunda, nunca será más rápido que la primera. La función $g(n)$ puede interpretarse como un límite, cuando n (la longitud de la cadena) tiende a ∞, en este caso superior, del ritmo al que crecen los recursos de espacio o de tiempo. Similarmente, cuando decimos que $f(n)$ es $\Omega(g(n))$ estamos estableciendo una *cota inferior* para $f(n)$, ya que por muy despacio que esta crezca nunca será más despacio que $g(n)$. Finalmente, decir que $f(n)$ es $\Theta(g(n))$ equivale a establecer una *cota ajustada* para $f(n)$ y que el ritmo de crecimiento de ambas funciones es el mismo. Nótese que hablamos de

tasa o de *ritmo* de crecimiento, lo cual no significa que para determinados valores de n el valor de $f(n)$ nunca sea superior al de $g(n)$, simplemente que, a medida que vamos asignando valores a n, el ritmo al que crecen los valores de $f(n)$ nunca es mayor que el ritmo al que crecen los valores de $g(n)$. Esto se ve perfectamente si comparamos el ritmo de crecimiento de una función polinómica con el de una función exponencial; cuadro 7.1.

n	n^2	2^n
0	0	1
1	1	2
2	4	4
3	9	8
4	16	16
5	25	32
6	36	64
7	49	128
8	64	256
9	81	512
10	100	1024

Cuadro 7.1: Comparación del ritmo de crecimiento de una función polinómica y una función exponencial.

Como se observa en el cuadro, para valores pequeños de n las diferencias entre una y otra función son despreciables; sin embargo, a medida que los valores de n se van haciendo más altos, la función exponencial empieza a crecer de manera mucho más rápida que la polinómica, lo cual sugiere que los problemas que tengan cotas de crecimiento exponencial serán más complejos que aquellos que tengan cotas de crecimiento polinómico.

Conseguir establecer una cota ajustada para una función no suele ser fácil y comporta, de hecho, haber conseguido previamente establecer cuáles son sus cotas superior e inferior. Esto, aunque resulta enormemente útil para conocer la complejidad de un determinado problema de decisión, no siempre es factible, de tal modo que en teoría de la complejidad se suele trabajar solo con cotas superiores. El motivo es claro, ya que la cota superior nos determina la complejidad de un problema *en el peor de los casos*, es decir, que por muy complejo que sea un problema nunca será más complejo de lo que determina su cota superior. Es por este motivo que, cuando definimos clases de complejidad como **TIEMPO**$(f(n))$ o **ESPACIO**$(f(n))$, de hecho $f(n)$ hace referencia a la cota superior de la función que define la cota de tiempo o espacio de los problemas dentro de esa clase.

7.2. Clases de complejidad

En la sección anterior hemos visto que podemos normalizar una función poniéndola en relación con otra función cuyo comportamiento es conocido. También vimos que lo que nos interesan son las tasas de crecimiento de las funciones, ya que son ellas las que determinan realmente la complejidad de un problema. Diferentes tipos de funciones muestran tasas de crecimiento diferentes de tal modo que, por ejemplo, una función polinómica siempre crecerá más lentamente que una función exponencial por muy elevado que sea el exponente de la primera. De todo ello se deduce, por un lado, que, por ejemplo, la clase $\mathbf{TIEMPO}(n^2)$ incluye problemas menos complejos que la clase $\mathbf{TIEMPO}(n^3)$, pero, por el otro lado, que cualquier clase del tipo $\mathbf{TIEMPO}(n^c)$ siempre incluirá problemas menos complejos que los que podamos hallar en la clase $\mathbf{TIEMPO}(c^{n^k})$.[2] Por tanto, si centramos nuestra atención en *familias de funciones*, como las polinómicas, las exponenciales, etc., observamos que existe una relación de inclusión entre las clases con funciones de la misma familia; por ejemplo

$$\mathbf{TIEMPO}(n) \subset \mathbf{TIEMPO}(n^2) \subset \mathbf{TIEMPO}(n^3) \subset \mathbf{TIEMPO}(n^4) \ldots$$

Lo que conforma una jerarquía infinita de clases de complejidad polinómica. Si ahora tomamos la unión de todas las clases de tiempo de complejidad polinómica obtenemos una de las clases de complejidad más importante, la clase $\mathscr{P}$, que incluye todos los lenguajes decidibles en tiempo polinómico:

$$\mathscr{P} = \bigcup_k \mathbf{TIEMPO}(n^k).$$

Esta clase incluye algunos lenguajes que son de especial interés; por ejemplo:

$$\{L \mid L \text{ es independiente del contexto}\} \subset \mathscr{P}.$$

Pero no solo los lenguajes independientes del contexto están en $\mathscr{P}$. Si recordamos la caracterización que hicimos en su momento de la sensibilidad al contexto moderada, una de las propiedades que asociamos a los lenguajes de esta clase es la de ser procesables en tiempo polinómico; es decir, que la clase $\mathfrak{L}$ de lenguajes moderadamente sensibles al contexto también está en $\mathscr{P}$. La importancia de esta clase es que se suele asumir que define los límites de lo que es tratable: cualquier problema que no esté en $\mathscr{P}$ se puede decir que es *intratable*.

Existen, evidentemente, clases de tiempo no polinómicas que están en $\mathscr{P}$, como $\mathbf{TIEMPO}(\log n)$, logarítmica, y $\mathbf{TIEMPO}(n \log n)$, lineal-logarítmica. La primera contiene lenguajes regulares como $\{aw \mid w \in \Sigma^*\}$, mientras que la segunda

[2] En las funciones exponenciales, la base de la función suele ser 2 pero puede, de hecho, ser cualquier otra constante; el exponente, asumimos que es un polinomio de base n cuyo exponente k es un entero positivo.

contiene lenguajes independientes del contexto deterministas como $\{a^n b^n\}$. Nótese que las tasas de crecimiento son, respectivamente, sensiblemente inferiores, en el caso de la clase logarítmica, o sensiblemente superiores a los de la clase lineal **TIEMPO**(n), en el caso de la lineal-logarítmica.[3]

Si cruzamos ahora la frontera del tiempo polinómico y nos adentramos en el territorio del tiempo exponencial, podemos definir

$$\textbf{TIEMPOEXP} = \bigcup_k \textbf{TIEMPO}(2^{n^k})$$

como la clase que contiene todos los problemas decidibles en tiempo exponencial. Como vimos en el cuadro comparativo 7.1, para valores pequeños de n, las diferencias entre un problema en $\mathscr{P}$ y un problema en **TIEMPOEXP** no son sustanciales, pero estas pronto se convierten en críticas a medida que los valores de n crecen. Un problema cuyo algoritmo tenga necesariamente que funcionar a tiempo exponencial es intratable.

Pero existen clases de complejidad de tiempo aún mayor, como la de tiempo exponencial doble (2**TIEMPOEXP**) o triple (3**TIEMPOEXP**) y, así, sucesivamente, cuya unión conforma la clase **TIEMPO ELEMENTAL** o simplemente **ELEMENTAL**:

$$\textbf{ELEMENTAL} = \bigcup_n n\textbf{TIEMPOEXP}.$$

Todas estas clases están relacionadas por inclusión, lo que nos permite establecer una primera versión de la jerarquía de complejidad:

$$\mathscr{P} \subset \textbf{TIEMPOEXP} \subset \textbf{ELEMENTAL}.$$

Esta jerarquía incluye, sin embargo, solo clases de complejidad que hacen referencia al tiempo, pero utilizando principios similares a los que hemos usado para construir la jerarquía de clases de tiempo, podemos construir una jerarquía de espacio:

$$\textbf{ESPACIOP} \subset \textbf{ESPACIOEXP}.$$

Ahora la cuestión es: ¿Es posible poner en relación ambas jerarquías? La respuesta es sí, pero con matices, ya que no siempre es posible establecer equivalencias fiables entre los recursos de tiempo y espacio, pero, como mínimo es posible demostrar lo siguiente:

$$\mathscr{P} \subseteq \textbf{ESPACIOP}.$$
$$\textbf{TIEMPOEXP} \subseteq \textbf{ESPACIOEXP}.$$
$$\textbf{ESPACIOP} \subseteq \textbf{TIEMPOEXP}.$$

[3] Se asume normalmente que la base del logaritmo es 2.

Nótese que las inclusiones no son propias, es decir que algunas de estas clases podrían ser iguales. Aunque se suele asumir que $\mathscr{P} \neq$ **ESPACIOP** y que **TIEMPOEXP** $\neq$ **ESPACIOEXP**, nadie ha sido todavía capaz de probarlo. Este tipo de situaciones no son infrecuentes en teoría de la complejidad computacional

Para terminar este capítulo, volvamos por un momento a la cuestión de las cotas superiores e inferiores de un problema dado. Ya hemos visto que para definir clases de complejidad usamos cotas superiores, la cual cosa significa que si un problema pertenece a esa clase, su complejidad es esa, pero no superior. En cierto modo también sabemos que el problema no es menos complejo, es decir que si está en $\mathscr{P}$ no se puede resolver con un algoritmo de complejidad de tiempo logarítmica, pero en este caso la certeza es menor. Es en relación con este punto que cobra importancia la cuestión de las cotas inferiores. Ya hemos dicho que es mucho más complicado determinar una cota inferior para un problema determinado que pertenezca a una clase dada **C**. Sin embargo, si hallamos una cota inferior para ese problema, entonces inmediatamente sabemos que cualquier algoritmo que podamos diseñar para resolverlo siempre será, como mínimo, de esa complejidad o, dicho de otro modo, que el problema en cuestión es como mínimo tan difícil de resolver o *duro* como cualquier problema que esté en **C**; puede ser más duro, pero no menos y en este caso diremos que es un problema **C**-duro. Por ejemplo, si logramos determinar que un problema es **TIEMPOEXP**-duro, entonces tenemos la absoluta seguridad de que no está en $\mathscr{P}$. Evidentemente, conocer la dureza mínima de un problema no nos lo dice todo sobre su dureza real, pero si además de saber que un problema es, pongamos por caso $\mathscr{P}$-duro, conseguimos demostrar que el problema está efectivamente en $\mathscr{P}$, entonces se convierte en una especie de problema «modelo» para la clase en cuestión; es lo que denominamos un problema *completo*. La existencia de lenguajes que podemos calificar de completos es fundamental porque nos permite utilizarlos como referencia en el análisis de otros lenguajes cuya complejidad desconocemos. Supongamos, por ejemplo, que el lenguaje L_c es un lenguaje $\mathscr{P}$-completo y que tenemos un lenguaje L_n cuya complejidad desconocemos. Si conseguimos demostrar que L_c y L_n son equivalentes, entonces ya disponemos de una caracterización exacta y precisa de la complejidad de L_n.[4]

[4]La equivalencia de lenguajes se demuestra a través de la posibilidad de reducir el problema completo al problema que estamos estudiando. Por tanto, una reducción es una conversión de un lenguaje de complejidad conocida a lenguajes cuya complejidad queremos determinar.

Capítulo 8

Nodeterminismo

En más de una ocasión a lo largo de este trabajo hemos señalado que el nodeterminismo es una propiedad misteriosa y poco comprendida. En este capítulo conoceremos mejor alguno de los motivos por los cuales es así. Recordemos, en primer lugar, que con la excepción de los AEF, los dispositivos nodeterministas tienen mayor poder de cómputo, como es el caso de los autómatas de pila. En el caso de las MT ocurre algo parecido, pero desde el punto de vista de la eficiencia, ya que una MT determinista siempre podrá simular a una nodeterminista pero con una pérdida exponencial del grado de eficiencia. A priori, por tanto, una MT nodeterminista, al ser más eficiente debería ser el modelo preferido en todo momento, pero esto no es así por un motivo muy claro: una MT nodeterminista no es un modelo computacional *realista*. No es difícil comprender el porqué; considérese la figura 8.1.

En la figura hemos intentado representar el espacio de opciones posibles que puede seguir una MT nodeterminista cuando realiza una computación. Al ser nodeterminista, en cada paso tiene una o más opciones a seguir y, estas, crecen exponencialmente a medida que avanzamos en el tiempo. Y lo que es peor, no todos los caminos que puede seguir la MT conducen a alguna parte: unos, a ninguna (ramas que no llegan al final en la figura), otros, son bucles infinitos (ramas seguidas de puntos suspensivos). Es decir, que si tenemos la suerte de que la MT elija *siempre* el camino adecuado, aceptará la cadena si esta pertenece al lenguaje y lo hará de forma eficiente; pero no tenemos la garantía de que lo haga, de tal modo que si se detiene, puede ser simplemente porque tomó el camino equivocado y se quedó «colgada» y no porque no haya aceptado la cadena. Similarmente, una opción equivocada (o una serie de ellas) puede hacer que la MT entre en un bucle y no se detenga *aunque la cadena pertenezca al lenguaje*. Es en este sentido que la MT nodeterminista no es un modelo computacional realista: a efectos prácticos, no es un algoritmo y lo que nos interesan son los algoritmos, aunque resulten ineficientes, porque sabemos con certeza que llegará un momento que se detendrán;

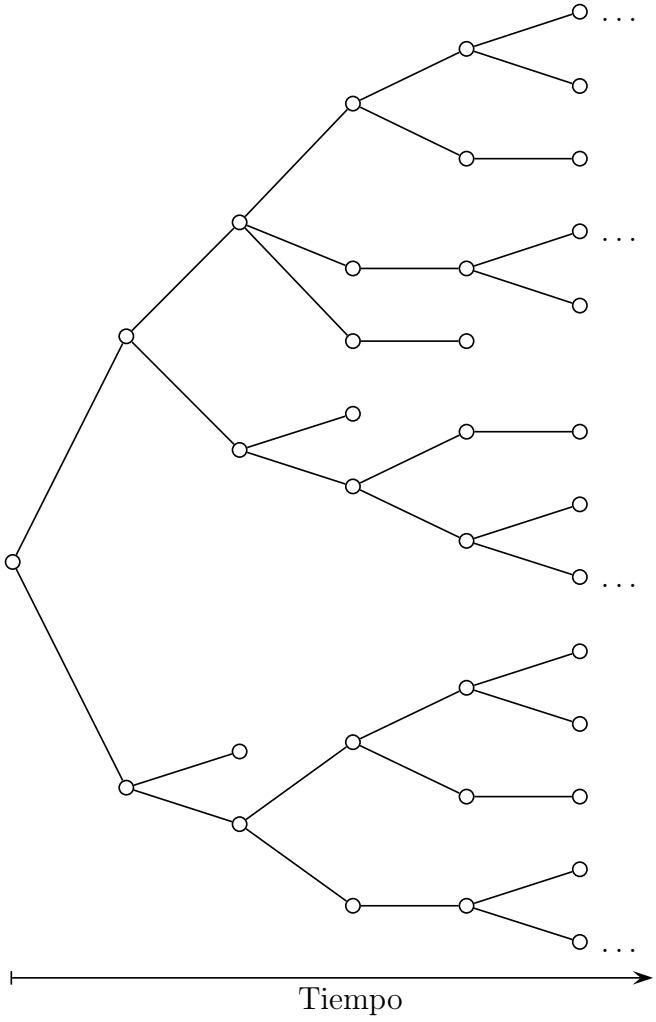


Figura 8.1: Una computación nodeterminista.

a nadie le gusta que se le cuelgue el ordenador. A continuación intentaremos de dotar de mayor contenido formal a lo dicho hasta ahora, luego explicaremos por qué, a pesar de todo, las MT nodeterministas tienen una importancia crucial en teoría de la complejidad.

Definición 8.1 (Máquina de Turing nodeterminista)**.** Una *MT nodeterminista* es una quíntupla $N = (Q, \Sigma, \Delta, q_0, F)$, donde Q, Σ, q_0 y F son, como siempre, un conjunto de estados, un alfabeto, el estado inicial y el conjunto de estados finales, respectivamente. Sin embargo, ahora, dado que el dispositivo es nodeterminista, para cada estado no habrá una única acción posible, sino que tendrá la opción de elegir entre una serie de acciones. Por tanto, Δ no es una función, sino una simple *relación*:

$$\Delta \subset (Q \times \Sigma) \times [(Q \cup F) \times \Sigma \times \{\mathbb{I}, \mathbb{D}, \mathbb{S}\}].$$

Es decir, que para cada combinación estado-símbolo, hay *más de un paso siguiente* apropiado, o ninguno.

Las configuraciones de una MT nodeterminista son exactamente iguales que las de una determinista, pero con la diferencia de que, al ser Δ una relación y no una función, diremos que la configuración (q, w, u) «resulta» en la configuración (q', w', u') en un paso, $(q, w, u) \underset{N}{\longmapsto} (q', w', u')$, si, y solo si, hay una regla que ratifica dicha transición como legal. Similarmente, podemos definir «resulta en k pasos», $\overset{k}{\underset{N}{\longmapsto}}$, y «resulta en cero o más pasos», $\overset{*}{\underset{N}{\longmapsto}}$, de modo similar, siempre teniendo en cuenta que ya no trabajamos con una función sino con una relación.

En una MT nodeterminista, por tanto, la relación entre entrada y salida es muy laxa y, como consecuencia de ello, el significado de «resolver un problema» o «decidir un lenguaje» ya no es el mismo que en el caso de una MT determinista.

Definición 8.2 (Decisión en modo nodeterminista)**.** Diremos que una MT nodeterminista N *decide* L si para cada $x \in \Sigma^*$ se cumple la condición siguiente: $x \in L$ si, y solo si, $(q_0, \triangleright, x) \overset{*}{\underset{N}{\longmapsto}} (\textit{«sí»}, w, u)$ para algunas cadenas w y u.

Como vemos, una cadena será aceptada si hay *alguna* secuencia de opciones nodeterministas que eventualmente conducen al estado «*sí*». Con que haya una basta; todas las demás pueden resultar en el rechazo de la cadena. De hecho, solo diremos que la MT N no acepta el lenguaje L si no hay *ninguna* secuencia que lleve al «*sí*». Existe, por tanto, una profunda asimetría entre el tratamiento que se da al «*sí*» y al «*no*» en una MT nodeterminista, ya que, aunque siempre nos fiaremos de un «*sí*», nunca nos podremos tomar en serio cualquier «*no*» que obtengamos.

Como solo nos interesan los resultados que terminan en «*sí*», entonces, en el momento de determinar los recursos de tiempo, por ejemplo, que utiliza la MT, nos fijaremos solamente en aquellas configuraciones que llevan a la aceptación. Pero no solo eso, sino que solo nos fijaremos en aquellas que no tomen un tiempo superior a una cota de tiempo $f(n)$ dada. Cualquier computación que no se ajuste a esa cota consideraremos que es *infinita*. Formalmente:

Definición 8.3 (Tiempo consumido por una MT nodeterminista)**.** Diremos que una MT nodeterminista N decide el lenguaje L *en tiempo* $f(n)$ si N decide L y, además, para toda $x \in \Sigma^*$, si $(q_0, \triangleright, x) \overset{k}{\underset{N}{\longmapsto}} (q, w, u)$, entonces $k \leq f(|x|)$.

El conjunto de lenguajes decidibles por MT nodeterministas en cotas de tiempo definidas por funciones polinómicas conforma toda una serie de clases de complejidad del tipo **TIEMPON**(n^k), cuya unión es la más fundamental e importante de las clases de complejidad nodeterminista, la clase $\mathscr{NP}$.

Si ahora pensamos que las MT deterministas son, de hecho, un caso particular de las nodeterministas, concretamente aquel en el que Δ es una función y no una relación, inmediatamente podemos ya establecer una relación entre clases deterministas y nodeterministas. La más crucial de todas ellas es que $\mathscr{P} \subseteq \mathscr{NP}$. Nótese que la inclusión no es propia, ambas clases podrían ser iguales, pero no lo sabemos. De hecho, uno de los problemas no resueltos más cruciales de la teoría de la complejidad computacional es si $\mathscr{P} \overset{?}{=} \mathscr{NP}$. En seguida veremos por qué.

8.1. ¿Qué significa estar en $\mathscr{NP}$?

Existen muchas otras clases nodeterministas, de espacio y de tiempo, exponenciales y logarítmicas, pero ninguna tan importante como $\mathscr{NP}$. El motivo es que sabemos que hay muchos problemas interesantes que están en $\mathscr{NP}$, pero lo sabemos por una vía que podríamos calificar de indirecta, porque hay otra manera de definir la clase $\mathscr{NP}$ además de la que hemos usado.

Olvidemos por un momento que lo que queremos es resolver problemas y asumamos que ya tenemos presuntas soluciones para ellos. Lo único que nos queda es verificar si lo son realmente o no. Para ello, ya no necesitamos un procedimiento de decisión sino algo más simple que denominaremos *procedimiento de verificación*.

Un procedimiento de verificación es, de hecho, otra manera de definir un lenguaje solo que ahora, en vez de usar el método tradicional, ponemos en relación dos cadenas x y p, donde la primera es la cadena que queremos verificar y p es lo que llamaremos un *certificado* que confirma la pertenencia de x al lenguaje en

cuestión. No entraremos en cuestiones formales complejas, pero la idea es que si una MT determinista consigue hallar una p para x, entonces x pertenece al lenguaje. Nótese que un procedimiento de verificación simplemente nos informa de los casos positivos en los que existe un certificado para x; es decir, que dada una serie de cadenas sobre un alfabeto, si para todas ellas podemos confirmar la existencia de un certificado, todas ellas pertenecen a un lenguaje dado.

Supongamos ahora que ponemos restricciones sobre el tiempo que debe durar un procedimiento de verificación $v(x, p)$ y sobre la longitud de los certificados, de tal modo que el tiempo que tarda la MT en verificar tenga cota polinómica y que p nunca sea más larga que x. Entonces:

Definición 8.4 (La clase $\mathscr{NP}$)**.** La clase de complejidad $\mathscr{NP}$ es el conjunto de todos los lenguajes L para los cuales existe un procedimiento de verificación v en tiempo polinómico y una constante k, tal que

$$L = \{x \mid \text{para algún certificado } p \text{ tal que } |p| \leq |x|^k, v(x, p)\}.$$

Dicho de otro modo, los lenguajes en $\mathscr{NP}$ son aquellos para los cuales podemos *verificar* una solución de forma eficiente. Los lenguajes en $\mathscr{P}$ son aquellos para los que podemos *hallar* una solución de forma eficiente. De ahí la importancia (incluso filosófica) de la conjetura $\mathscr{P} \stackrel{?}{=} \mathscr{NP}$: ¿es lo mismo resolver un problema que comprobar si una posible solución a este es la correcta? Intuitivamente, uno tiende a creer que no, pero la confirmación matemática de esta intuición todavía no ha sido posible hallarla.

Parte IV

Conclusiones

Capítulo 9

Reflexión final

Unas conclusiones deben ser necesariamente breves. Y estas no serán una excepción. En ellas intentaré destilar algunas de las cuestiones más importantes que hemos visto a lo largo de todo este trabajo e intentaré también resumir algunas consecuencias que de ellas se derivan.

Este trabajo se articula a lo largo de dos ejes principales, la complejidad estructura y la complejidad computacional de los lenguajes, y uno de nuestros principales objetivos ha sido mostrar cómo ambas se complementan, precisamente por el hecho de que cada una nos ofrece una medida de complejidad centrada en parámetros distintos. Por un lado, como su propio nombre indica, la complejidad estructural nos informa sobre la posibilidad/imposibilidad de presentar determinadas pautas de organización estructural que correlacionan con propiedades también estructurales de los modelos computacionales que definen los lenguajes en cuestión. Por el otro lado, la complejidad computacional nos dice algo sobre el grado de dificultad que comporta procesar lenguajes que presenten un tipo u otro de pautas estructurales.

En los tres capítulos dedicados a estudiar la complejidad estructural de los sistemas que conforman la Jerarquía de Chomsky hemos intentado demostrar que un lenguaje formal y sus correspondientes modelos computacionales nos pueden servir de modelo abstracto para caracterizar propiedades estructurales en fenómenos naturales como el lenguaje humano u otras actividades observables y susceptibles de un análisis de este tipo, como el canto de los pájaros. Gracias a las herramientas que nos proporciona la teoría de los lenguajes formales hemos podido, por tanto, observar que el canto de los pájaros parece organizarse de acuerdo con patrones recursivos de dependencia exclusivamente lineal, mientras que en el lenguaje humano los patrones recursivos dan lugar a dependencias más complejas, anidadas e incluso cruzadas.

Pese a la innegable utilidad de esta metodología de análisis, también hemos hecho hincapié en más de una ocasión en el hecho de que la posibilidad de asignar

un determinado grado de complejidad a un sistema es un resultado que debe siempre tomarse con precaución. El motivo, que también hemos hecho explícito en su momento, es que la complejidad estructural, por muy informativa que sea, no deja de ser una medida relativamente poco precisa de la complejidad «real» del sistema. Por un lado, *y en el mejor de los casos*, el método puede resultar en la definición de una cota mínima de complejidad, en el sentido de que si *realmente hemos sido capaces* de determinar que el sistema que estamos estudiando muestra patrones estructurales asociables a la independencia del contexto, entonces los recursos computacionales necesarios para procesarlo nunca podrán ser inferiores a los que tiene a su disposición un autómata de pila. Los subrayados en la frase anterior no son superfluos y nos deberían poner en guardia ante un hecho que es vital no perder nunca de vista en el momento de recurrir a la teoría de los lenguajes formales como herramienta de investigación. La teoría de los lenguajes formales puede ser muy útil, pero lo que nos puede decir (o dejar de decir) nunca es independiente del uso que hagamos de ella en contextos específicos: una aplicación inadecuada o poco precisa de ella puede dar lugar a resultados o interpretaciones no concluyentes o, directamente, falsos. Para comprender mejor este punto, conviene recurrir a un ejemplo concreto.

En el año 2004, Marc Hauser y Tecumseh Fitch publicaron un trabajo en el que seguían un protocolo experimental que con el tiempo ha venido a conocerse como *reconocimiento de patrones auriculares* o, también, de *aprendizaje de lenguajes artificiales*.[1] En esencia, el protocolo consiste en entrenar a individuos de una determinada especie animal a reconocer estímulos acústicos (por ejemplo, cadenas de sílabas) que muestren una pauta estructural determinada. En el experimento de Fitch y Hauser, los sujetos de experimentación eran monos tamarinos (*Saguinus oedipus*) y los resultados que presentaron los experimentadores fue que estos primates, si bien eran capaces de reconocer patrones que siguieran una pauta regular (p. ej., $(ab)^n$), no podían hacerlo si la pauta era independiente del contexto (p. ej., $a^n b^n$). La interpretación de Fitch y Hauser de estos resultados apuntaba a la conclusión de que el sistema computacional de los monos tamarinos no poseería una capacidad mayor a la de un sistema regular.

El mismo protocolo fue aplicado poco después por Timothy Gentner y colaboradores, pero utilizando en este caso sujetos pertenecientes a una especie de ave, el estornino pinto (*Sturnus vulgaris*).[2] En este caso, el resultado apuntaba hacia algo sorprendente, ya que, al parecer, los estorninos *sí* eran capaces de identificar pautas independientes del contexto. El problema de esta interpretación es que los experimentadores nunca utilizaron estímulos más complejos que *aaaabbbb*, es de-

[1] W. Tecumseh Fitch & Marc D. Hauser. 2004. "Computational constraints on syntactic processing in nonhuman primates". *Science* 303:377–380.

[2] Timothy Q. Gentner, Kimberly M. Fenn, Daniel Margoliash & Howard Nusbaum. 2006. "Recursive syntactic pattern learning by songbirds". *Nature* 440:1204–1207.

cir que, en la práctica, lo que los estorninos «aprendieron» es el lenguaje finito a^4b^4, no el lenguaje infinito e independiente del contexto a^nb^n. Recordemos que los lenguajes finitos *siempre son regulares*.

Esta es la historia de un fracaso. Un fracaso tal que pone en tela de juicio la validez del protocolo experimental y, quizá también, la aplicabilidad de la teoría de lenguajes formales al estudio de la cognición. Sin embargo, en mi opinión, este ejemplo nos muestra simplemente lo que, en todo caso, es una cierta torpeza en el momento de aplicar la teoría. Por un lado, torpeza en el momento de interpretar la mera presencia de una pauta de estructuras anidadas automáticamente como evidencia de independencia del contexto cuando esto no es cierto, ya que el sistema, además de mostrar la pauta, debe ser también infinito. Lo cual no siempre es fácil de determinar. Y eso nos lleva al segundo problema, que no es más que una variante del caso de la aprendibilidad de lenguajes que comentamos en la sección del capítulo 3 dedicada a lenguajes finitos (página 43). Recordemos que, entonces, el problema que se nos planteaba era: ¿cómo es posible que un aprendiz, por ejemplo humano, infiera a partir de un conjunto necesariamente finito de estímulos un sistema infinito? La respuesta, en este caso, era que, a partir de un determinado umbral (que no conocemos), la inferencia más simple es apostar por un sistema infinito antes que por un sistema finito pero muy grande. Nótese que el problema en el caso del experimento de Gentner y colaboradores es asumir que basta con exponer al aprendiz a la pauta en cuestión para que este infiera que se halla ante un sistema infinito. La evidencia de que el aprendiz efectivamente ha hecho la inferencia en cuestión nunca vendrá de su capacidad de identificar secuencias que posean esa pauta, sino, como ocurre en el caso del lenguaje humano, de la evidencia de que el aprendiz explota esa pauta y *produce* secuencias que contengan la pauta *aunque nunca haya estado expuesto a ellas durante el período de entrenamiento*. Lo que es, en cierto modo, una variante del argumento de la pobreza del estímulo.

Estas consideraciones restan, sin duda, relevancia al experimento de los estorninos, algo que, por cierto, no han pasado por alto algunos investigadores críticos con el trabajo de Gentner,[3] pero no necesariamente a las conclusiones de Fitch y Hauser que, sin embargo, sí debemos tomarnos con cierta precaución.[4] Un resultado de aprendizaje nunca puede ser definitivo y, para que lo fuese, debería ser posible observar de manera concluyente, pautas complejas en acciones *espontáneas* del sujeto de estudio. Esas acciones, por lo demás, no tienen necesariamente por

[3]Caroline A. A. van Heijningen, Jos de Visser, Willem Zuidema & Carel ten Cate. 2009. "Simple rules can explain discrimination of putative recursive syntactic structures by a songbird species". *Proceedings of the National Academy of Sciences USA* 106(48):20538–20543.

[4]Para una revisión puesta al día de los protocolos experimentales y de los resultados de este tipo de experimentos, puede consultarse el volumen monográfico de las *Philosophical Transactions of the Royal Society*, coordinado por W. Tecumseh Fitch, Angela D. Friederici y Peter Hagoort en http://rstb.royalsocietypublishing.org/content/367/1598.toc.

qué localizarse en el ámbito estricto de las vocalizaciones, cuya estructura, casi con toda seguridad, como ocurre en el lenguaje humano, mostrará un patrón lineal, lo cual guarda, sin duda, una estrecha relación con restricciones de tipo biológico y físico relativas a la naturaleza intrínsecamente serial de los gestos articulatorios y, ¿por qué no?, con la naturaleza del sonido en tanto que fenómeno ondulatorio sujeto a las restricciones inviolables que impone la línea del tiempo; quizá incluso también con la naturaleza del sistema perceptivo de los vertebrados. No ir más allá es un error fruto de la tradicional estrechez de miras enraizada en la idea de que tiene sentido aplicar el método comparativo atendiendo a criterios funcionales y que, por tanto, si hablamos de lenguaje, allí donde tenemos que buscar es en otros sistemas de comunicación, especialmente aquellos que utilizan el canal oral-auditivo. Una pauta de conducta muy compleja en un ámbito totalmente ajeno a la comunicación puede ser tanto o más reveladora que una pauta simple en una conducta comunicativa.[5]

Estas observaciones cuadran, además, con una tesis recientemente expuesta por Chomsky sobre la asimetría de los sistemas periféricos en relación con el sistema computacional humano.[6] De acuerdo con ella, la interfaz con los sistemas de exteriorización serían una incorporación reciente, evolutivamente hablando, a un sistema preexistente formado por el sistema computacional propiamente dicho y el sistema conceptual-intencional. Nótese que, efectivamente, es en este «subsitema» interno donde se observan más claramente aquellas propiedades que sugieren que, como mínimo, la complejidad estructural del lenguaje humano se sitúa en una zona dentro de la sensibilidad al contexto moderada, la cual queda en cierto modo «enmascarada» por la restricciones de linealidad que impone el sistema de exteriorización de la expresiones generadas internamente por la actuación conjunta del sistema computacional y el sistema conceptual-intencional.

No podemos más que insistir, sin embargo en ese «como mínimo» en relación con la complejidad del lenguaje, ya que esta podría ser mayor en función de diversos factores. Uno de ellos tendría una base puramente estructural y empírica y guardaría relación con o bien la posibilidad de que existan evidencias que apunten a un nivel de complejidad más allá de la sensibilidad al contexto moderada o bien relacionadas con la capacidad generativa fuerte y las descripciones estructurales que queramos asignar (figura 5.3, página 70). Pero, ¿y si así fuera? ¿Qué consecuencias tendría un hallazgo que estableciera que la complejidad real del lenguaje va más allá de los límites definidos por la sensibilidad al contexto moderada? Aquí es donde la teoría de la complejidad computacional tiene algo que decirnos y la

[5]Sergio Balari & Guillermo Lorenzo. 2013. *Computational Phenotypes. Towards an Evolutionary Developmental Biolinguistics*. Oxford University Press: Oxford.

[6]Noam Chomsky. 2010. "Some simple evo devo theses: How true might they be for language". En R. K. Larson, V. Déprez & H. Yamakido (eds), *The Evolution of Language. Biolinguistic Perspectives*. Cambridge University Press: Cambridge, pp. 45–62.

respuesta es que, en este caso, el lenguaje sería un problema computacionalmente intratable, no estaría en $\mathscr{P}$.

Los humanos somos capaces de resolver muchos problemas intratables. Por ejemplo, por sorprendente que parezca, la complejidad del Tres en raya, con un tablero de 3×3 y un número medio de 9 turnos, es un problema **ESPACIOP**-completo. Las Damas, el Ajedrez o el Go son problemas **TIEMPOEXP**-completos. Nótese que lo que esto significa no es que no tengan solución, sino que no existe un algoritmo eficiente para resolverlos; significa, de hecho, que cualquier algoritmo que podamos hallar será, siempre, altamente ineficiente (son problemas completos). La paradoja, claro, es que hay humanos capaces de resolverlos con gran eficiencia y destreza: el Tres en raya, casi cualquiera; los demás, depende ya de cada uno. ¿Cuál es la interpretación de esta paradoja? Pues que las estrategias cognitivas que utilizamos para resolver estos problemas no son computacionales o lo son solo parcialmente: de un modo u otro, interviene algún tipo de heurística que permite minimizar el coste computacional real del problema; ¿quizás algo equivalente a los certificados que encontramos en la definición de la clase $\mathscr{NP}$?[7]

A principios de la década de 1990, Eric Sven Ristad publicó un libro en el que afirmaba que la complejidad computacional del lenguaje humano se halla, efectivamente, en $\mathscr{NP}$.[8] Ristad interpretó sus resultados (a día de hoy no rebatidos) como una prueba en contra de la hipótesis de la modularidad de la mente de Fodor,[9] según la cual determinados sistemas cognitivos, el lenguaje entre ellos, poseen la propiedad del encapsulamiento o, en otras palabras, que no interactúan con otros módulos o sistemas mientras ejecutan los procesos que les corresponden. Según Ristad, a la luz de los resultados de complejidad por él obtenidos, el lenguaje no sería modular en el sentido fodoriano del término, sino que, en determinadas circunstancias recurriría a «certificados» que permitirían verificar de forma eficiente ciertas soluciones a las que sería demasiado difícil llegar siguiendo el modo computacional clásico.

Este es solo un ejemplo. No sabemos con certeza hasta qué punto son válidos los resultados de Ristad, pero tanto este caso como el de los juegos nos muestran el tipo de conclusiones a las que podemos llegar complementando la teoría de la complejidad estructural con la teoría de la complejidad computacional. Ambas, aplicadas en su justa medida, nos pueden abrir nuevas vías de investigación en la compleja tarea de desentrañar los misterios de las mentes humanas o de otros animales y, eventualmente, llegar a comprender cómo llegaron a ser lo que son. Evidentemente, siempre y cuando Church y Turing tuvieran razón, porque, si no

[7] Todo esto depende, claro, de la aceptación de la veracidad de la Tesis de Church-Turing y, por tanto, de que los límites de la computabilidad los marca la máquina de Turing, la vara de medir complejidad sobre la que se basa la teoría de la complejidad computacional.

[8] Eric Sven Ristad. 1993. *The Language Complexity Game.* The MIT Press: Cambridge, MA.

[9] Jerry A. Fodor. 1983. *The Modularity of Mind.* The MIT Press: Cambridge, MA.

es así, quizá nos veamos obligados a replantearnos muchas cosas. A lo mejor, resulta que hipercomputamos...

Bibliografía

La bibliografía sobre lenguajes formales y complejidad computacional es inmensa. Abundan los textos más o menos introductorios que se ocupan, también más o menos, de los mismos aspectos, quizá con mayor o menor énfasis en unos que en otros, pero las diferencias no son sustanciales. El problema es que todos estos libros van dirigidos a estudiantes de ciencias, casi siempre de informática, lo que explica que la selección de temas sea siempre muy parecida: lenguajes regulares, lenguajes independientes del contexto, máquinas de Turing, decidibilidad, etc. En ningún caso se presta atención a aspectos que puedan ser relevantes para otras disciplinas que no sean científicas (la física, por ejemplo) o técnicas, de ahí, por ejemplo, la casi total ausencia de capítulos dedicados a lenguajes por encima de los independientes del contexto: no sirven para diseñar lenguajes de programación. Por tanto, no puedo recomendar ningún libro en concreto, así que me limito a citar mis fuentes y a comentarlas.

- Arora, Sanjeev & Boaz Barak (2009). *Computational complexity. A modern approach*. Cambridge University Press: Cambridge.

  Un texto muy puesto al día sobre complejidad computacional. Se ocupa de otros modelos computacionales y de las clases que se pueden definir a partir de ellos.

- Hopcroft, John E. & Jeffrey D. Ullman (1979). *Introduction to automata theory, languages, and computation*. Addison Wesley: Reading, MA.

  Este es un clásico. Existen nuevas ediciones, pero me remito a la primera de 1979. No es el más accesible de los textos, pero sus capítulos sobre sistemas regulares e independientes del contexto son muy completos. Menos claras son las secciones dedicadas a complejidad computacional. Es el único que trata, aunque de pasada, otros lenguajes aparte de los mencionados.

- Kallmeyer, Laura (2013). "Linear Context-Free Rewriting Systems". *Language and Linguistics Compass* 7(1):22–38.

  Un artículo (casi) divulgativo donde se trata la relevancia lingüística de los LCFRS. Con ejemplos de dependencias cruzadas y cosas así.

- Lewis, Harry R. & Christos H. Papadimitriou (1981). *Elements of the theory of computation*. Prentice-Hall: Englewood Cliffs, NJ.

  Apenas se ocupa de lenguajes, pero sus secciones dedicadas a nociones matemáticas básicas son muy accesibles. Su tratamiento de las máquinas de Turing, de la computabilidad y la decidibilidad es excelente y muy claro.

- Papadimitriou, Christos H. (1994). *Computational complexity*. Addison Wesley: Reading, MA.

  La mayor parte del libro está dedicada al análisis y al estudio de problemas concretos desde el punto de vista de la complejidad computacional, pero sus primeros capítulos, donde se presentan las clases principales, son bastante claros.

- Webber, Adam Brooks (2008). *Formal language. A practical introduction*. Franklin, Beedle & Associates: Wilsonville, OR.

  Es probablemente el más accesible de todos los textos tanto para la teoría de lenguajes como para la complejidad computacional. Aunque va dirigido a programadores en Java, es posible seguir la mayoría de las explicaciones y demostraciones sin conocer este lenguaje. Puestos a plantearse la posibilidad de adquirir un texto de consulta, sin duda este es el mejor candidato.