

# Teoría de Autómatas y Lenguajes Formales

Laura M. Castro Souto

Primer Cuatrimestre

Curso 2000/2001



# Índice de cuadros



# Capítulo 0

## Introducción

En la asignatura de **Teoría de Autómatas y lenguajes formales** vamos a tratar los “cimientos”, los fundamentos teóricos de todo lo que tiene que ver con la informática y los computadores. No en vano a este campo se le denomina en ocasiones *Informática Teórica*.

Estudiaremos los **lenguajes formales**, lenguajes con una estructura formal, matemática, con unas reglas que nos permitirán usarlos de forma correcta. Veremos esas directrices y posteriormente qué resultaría correcto y qué no ateniéndonos a ellas.

Construiremos **autómatas** para poner en práctica las posibilidades de los lenguajes.

Enlazando ambos, las **gramáticas**, asociadas tanto a los lenguajes como a los autómatas.

LENGUAJES FORMALES	AUTÓMATAS	GRAMÁTICAS
L. REGULARES	A. FINITOS	G. REGULARES
L. INDEP. DEL CONTEXTO	A. DE PILA	G. INDEP. DEL CONTEXTO
L. RECURSIVOS	MÁQUINAS DE TURING	FUNCIONES MI-RECURSIVAS
L. RECURSIVAMENTE		$\lambda$ -CÁLCULO
ENUMERABLES		PROBLEMAS

Las *gramáticas regulares* son fundamentales para el estudio de los *compiladores*, puesto que permiten hacer de forma cómoda el análisis léxico. Las *gramáticas independientes del contexto*, por su parte, nos facilitan el análisis sintáctico.

En la segunda parte de la asignatura veremos el concepto fundamental que constituyen las *Máquinas de Turing*, un modelo matemático de toda la teoría de la computación. En su contexto, las *funciones mi-recursivas*, en las que se basan todos los lenguajes imperativos, y su equivalente base de los lenguajes de programación funcional, el  *$\lambda$ -cálculo*.

Por último, nos ocuparemos de los muchos problemas que se pueden resolver y también de los que no se pueden resolver, que son aún muchos más.



# Capítulo 1

## Alfabetos y lenguajes

### 1.1. Alfabetos, palabras y lenguajes

Llamamos **alfabeto**, y lo representamos  $\Sigma$ , a un *conjunto finito de símbolos* que utilizamos para construir frases, programas, representaciones de números...

Una **palabra** (o **cadena**) es una *secuencia finita de símbolos de un alfabeto*. A las palabras se les puede dar significado e interpretación como símbolos. Cada símbolo puede a su vez ser interpretado como una palabra. Son palabras: **pan**, **pppban**,...

Ejemplo:

$$\Sigma = \{a, b, 1, 2\} \quad \text{Palabras: } ab, ba, b12, ba2abbb$$

Representaremos por  $\varepsilon$  (a veces por  $\lambda$ ) el concepto de **palabra vacía**, que es imprescindible y simboliza la *palabra que no tiene ningún símbolo*.

Se denomina **cierre de sigma** ó **sigma estrella**,  $\Sigma^*$ , al *conjunto de todas las palabras* que se pueden construir a partir de un alfabeto.

Ejemplo:

$$\Sigma = \{1, 2\} \quad \rightsquigarrow \quad \Sigma^* = \{\varepsilon, 1, 2, 11, 12, 21, 22, 111, \dots\}$$

La **longitud**  $|\omega|^1$  de una palabra es el *número de símbolos* del alfabeto que contiene. Por ejemplo,  $|21221| = 5$ .

**Concatenar** dos palabras es poner *una a continuación de la otra*:

$$\begin{array}{ccc} \Sigma^* \times \Sigma^* & \xrightarrow{\cdot} & \Sigma^* \\ (w, z) & \rightsquigarrow & wz \\ \varepsilon & \text{(neutro)} & \end{array}$$

Se definen las **potencias** de una palabra:

$$\omega^n = \begin{cases} \varepsilon & \text{si } n = 0 \\ \omega\omega^{n-1} & \text{si } n > 0 \end{cases}$$

Usaremos también otros conceptos simples y sencillos ampliamente conocidos, como **prefijo**, **sufijo**, **subcadena**,...

---

<sup>1</sup>Por convenio, utilizaremos las letras  $a, b, c, \dots$  y los números  $1, 2, 3, \dots$  para representar símbolos y las letras  $u, v, w, \dots$  para representar palabras.

La **inversa** ó **traspuesta** de una palabra es:

$$\omega^I = \begin{cases} \omega & \text{si } \omega = \varepsilon \\ y^I a & \text{si } \omega = ay, \text{ con } \begin{matrix} a \in \Sigma \\ y \in \Sigma^* \end{matrix} \end{cases}$$

## Lenguajes

Dado un alfabeto cualquiera  $\Sigma$ , hemos definido  $\Sigma^*$ . A partir de esto, decimos que un **lenguaje**  $\mathcal{A}$  es un *subconjunto* de  $\Sigma^*$ ,  $\mathcal{A} \subseteq \Sigma^*$ .

Tenemos  $\aleph_1$  lenguajes. Sobre un mismo alfabeto se pueden tener diferentes lenguajes. Un lenguaje importante es el **lenguaje vacío**,  $\emptyset$ . También es un lenguaje, y diferente, el lenguaje  $\{\varepsilon\}$ .

**Concatenación** de lenguajes:

$$\mathcal{A} \cdot \mathcal{B} = \left\{ \omega x \mid \begin{matrix} \omega \in \mathcal{A} \\ x \in \mathcal{B} \end{matrix} \right\}$$

**Potencia** de un lenguaje:

$$\mathcal{A}^n = \begin{cases} \{\varepsilon\} & n = 0 \text{ (potencia cero)} \\ \mathcal{A}\mathcal{A}^{n-1} & n > 0 \end{cases}$$

Otras propiedades:

- $\mathcal{A} \cdot \{\varepsilon\} = \mathcal{A}$ .
- $\mathcal{A} \cdot \emptyset = \emptyset$ .
- $\mathcal{A} \cup \mathcal{B}, \mathcal{A} \cap \mathcal{B}$ .
- $\bar{\mathcal{A}} = \Sigma^* - \mathcal{A}$ .
- $\mathcal{A} \cdot (\mathcal{B} \cup \mathcal{C}) = \mathcal{A}\mathcal{B} \cup \mathcal{A}\mathcal{C}$ .
- $(\mathcal{B} \cup \mathcal{C}) \cdot \mathcal{A} = \mathcal{B}\mathcal{A} \cup \mathcal{C}\mathcal{A}$ .
- Dos lenguajes son iguales cuando  $\mathcal{A} \subset \mathcal{B}$  y  $\mathcal{B} \subset \mathcal{A}$ .

Se llama **cierre de Kleene** ó **cierre estrella** de  $\mathcal{A}$ ,  $\mathcal{A}^*$ , a la *unión* de todos los posibles *subconjuntos de palabras* que se pueden formar con las palabras que están presentes en dicho lenguaje.

$$\mathcal{A}^0, \mathcal{A}^1, \mathcal{A}^2, \dots, \mathcal{A}^n, \dots \quad \bigcup_{n=0} \mathcal{A}^n = \mathcal{A}^*$$

Si establecemos el límite inferior en  $n = 1$ , se obtiene el **cierre positivo** de  $\mathcal{A}$ :

$$\bigcup_{n=1} \mathcal{A}^n = \mathcal{A}^+$$

Puesto que  $\mathcal{A}^0 = \{\varepsilon\}$ ,  $\mathcal{A}^*$  y  $\mathcal{A}^+$  son iguales si  $\mathcal{A}$  contiene la palabra vacía; de lo contrario, ambos se diferencian en  $\{\varepsilon\}$ .

Ejemplo:

$$\begin{aligned}\mathcal{A} &= \{a\} \\ \mathcal{A}^+ &= \{a, aa, aaa, \dots\} \\ \mathcal{A}^* &= \{\varepsilon, a, aa, aaa, \dots\}\end{aligned}$$

Otras propiedades:

- $\mathcal{A} \cdot \mathcal{A}^* = \mathcal{A}^* \cdot \mathcal{A} = \mathcal{A}^+$ .
- $\mathcal{A}^I = \{\omega^I / \omega \in \mathcal{A}\}$ .
- $(\mathcal{A}^I)^I = \mathcal{A}$ .
- $(\mathcal{A} \cdot \mathcal{B})^I = \mathcal{B}^I \cdot \mathcal{A}^I$ .



# Capítulo 2

## Lenguajes Regulares

### 2.1. Lenguajes sobre alfabetos

Sea  $\Sigma = \{a_1, a_2, \dots, a_n\}$  un alfabeto con  $n$  símbolos. Para ver que  $\Sigma^*$  es **enumerable** simplemente podemos ver que es posible “contar” todas las palabras que se pueden construir con él:

$\varepsilon$	0
$a_1$	1
$a_2$	2
$\vdots$	$\vdots$
$a_n$	$n$
$a_1a_1$	$n + 1$
$a_1a_2$	$n + 2$
$\vdots$	$\vdots$
$a_1a_n$	$2n$
$a_2a_1$	$2n + 1$
$a_2a_2$	$2n + 2$
$\vdots$	$\vdots$
$a_na_n$	$n^2$
$a_1a_1a_1$	$n^2 + 1$
$\vdots$	$\vdots$

Sin embargo, los lenguajes construidos sobre  $\Sigma$  podemos asociarlos con  $2^{\Sigma^*}$ , de modo que **no** es un conjunto **enumerable**.

#### Lenguajes regulares

Se definen:

- (a)  $\emptyset$  es un *lenguaje regular*.
- (b)  $\{\varepsilon\}$  es un *lenguaje regular*.
- (c)  $\forall a \in \Sigma, \{a\}$  es un *lenguaje regular*.
- (d) Si  $\mathcal{A}, \mathcal{B}$  son *lenguajes regulares*  $\Rightarrow \mathcal{A} \cup \mathcal{B}, \mathcal{A} \cdot \mathcal{B}, \mathcal{A}^*, \mathcal{B}^*$  son *lenguajes regulares*.

Y sólo son **lenguajes regulares** los que podemos obtener a partir de los básicos aplicando un número finito de veces los operadores indicados.

Así,

$\{a^i / i \geq 0\}$  es un lenguaje regular porque es el cierre estrella de  $\{a\}$   
 $\{a^i b^j / i \geq 0 \text{ y } j \geq 0\}$  es un lenguaje regular porque es la concatenación de los cierres estrella de  $\{a\}$  y  $\{b\}$   
 $\{a^i b^i / i \geq 0\}$  no podemos afirmar que sea un lenguaje regular ateniéndonos a esta definición

## 2.2. Lenguajes y expresiones regulares

Se considera que:

- (a)  $\emptyset$  y  $\varepsilon$  son *expresiones regulares*.
- (b)  $\forall a \in \Sigma, a$ , entendiéndose que  $a$  denota  $\{a\}$ , es una *expresión regular*.
- (c) Si  $r, s$  son *expresiones regulares*  $\Rightarrow r \cup s, r \cdot s$  y  $r^*, s^*$  son *expresiones regulares*.

Dos **expresiones regulares** se dicen **equivalentes** si el *lenguaje que definen* es el mismo. Representaremos por  $r$  a la expresión y por  $\mathcal{L}(r)$  al lenguaje que define.

## 2.3. Autómatas finitos deterministas

Un **Autómata Finito Determinista (A.F.D.)**<sup>1</sup> es una *colección de 5 elementos*:

$$\mathcal{M} = (\Sigma, Q, s, \mathcal{F}, \delta)$$

donde:

$\Sigma$  es un alfabeto (denominado *alfabeto de entrada*)  
 $Q$  es un conjunto finito de estados<sup>2</sup>  
 $s \in Q$  es un elemento especial que llamamos *estado inicial*  
 $\mathcal{F} \subseteq Q$ , conjunto que llamamos *estados finales* ó de *aceptación*  
 $\delta$  una *función de transición*  $\delta : Q \times \Sigma \rightarrow Q$   
 $(q_i, \sigma) \rightsquigarrow \delta(q_i, \sigma)$

La *función de transición* debe estar definida para todos los *estados* posibles, de no ser así el autómata no se considera finito determinista.

### 2.3.1. Representaciones

Los A.F.D. se pueden representar tanto usando un **grafo**, donde cada *nodo* es un *estado*, que si es final se representa con un doble círculo y si es inicial tiene una flecha apuntándole, y cada *transición* está indicada por una *arista*.

También puede hacerse uso de una **tabla de transiciones**, donde se colocan estados frente a símbolos del alfabeto y se rellenan las casillas según corresponde.

<sup>1</sup>Así es como lo abreviaremos.

<sup>2</sup>A los elementos de  $Q$  los llamamos *estados*.

## 2.4. Autómatas finitos deterministas y lenguajes

Extendemos el concepto de la función  $\delta$  de modo que:

$$\delta(q_1, abab) = \delta(\delta(\delta(\delta(q_1, a), b), a), b), a), b) = \dots$$

Una cadena se dice que es *aceptada por un autómata* si éste, empezando en el estado inicial y analizando toda la cadena, termina en un estado final ó de aceptación. Es decir:

$$\text{Sea } \omega \in \Sigma^*, \text{ es aceptada por el autómata si } \delta(q_0, \omega) \in \mathcal{F}$$

Dado un autómata finito cualquiera  $\mathcal{M}$ , denotaremos por  $\mathcal{L}(\mathcal{M})$  el **lenguaje** que acepta el autómata, y lo definimos como las cadenas/palabras construidas en nuestro alfabeto tales que son aceptadas por el autómata:

$$\mathcal{L}(\mathcal{M}) = \{\omega \in \Sigma^* \mid \delta(q_0, \omega) \in \mathcal{F}\}$$

► Dado cualquier lenguaje regular, existe un autómata finito que lo acepta, y viceversa.

► Dos autómatas son *equivalentes* si el lenguaje que aceptan es el mismo:

$$\mathcal{L}(\mathcal{M}_1) = \mathcal{L}(\mathcal{M}_\epsilon)$$

► Para que un A.F. acepte la cadena vacía sólo hay que hacer que el estado inicial sea también un estado final.

## 2.5. Autómatas finitos no deterministas

Nos encontramos con un **A.F. no determinista** cuando la *función de transición* no es tal, cuando hay estados en los que, ante la llegada de un símbolo, se puede ir a un estado o a otro, por ejemplo. Se tiene, pues, que  $\delta$  ya no es una función; hablaremos de *relaciones* o de *funciones en otro dominio*.

Un **Autómata Finito No Determinista** es una *colección de 5 elementos*:

$$\mathcal{M} = (\Sigma, \mathcal{Q}, \mathcal{F}, s, \Delta)$$

donde:

$\Sigma$  es el *alfabeto de entrada*

$\mathcal{Q}$  es un *conjunto finito de estados*

$\mathcal{F} \subseteq \mathcal{Q}$  es también un conjunto finito de *estados finales* ó de *aceptación*

$s \in \mathcal{Q}$  es el *estado inicial*

$\Delta$  es una *relación de transición*, un subconjunto de:  $\Delta \subseteq (\mathcal{Q} \times \Sigma) \times \mathcal{Q}$

Equivalentemente, podemos decir que la *función de transición*  $\Delta$  es una función cuyo *codominio* es *partes de*  $\mathcal{Q}$ :

$$\Delta : \mathcal{Q} \times \Sigma \rightarrow 2^{\mathcal{Q}}$$

$$(q, \sigma) \rightsquigarrow \Delta(q, \sigma) \subseteq \mathcal{Q}$$

La representación es idéntica al caso de los A.F.D., bien mediante grafos, bien mediante tablas de transiciones, donde en este caso en cada casilla puede ir un conjunto de estados.

También es posible extender la interpretación de la aplicación de  $\Delta$  a un sólo símbolo a la aplicación de  $\Delta$  a una cadena, salvo que, esta vez, como cada símbolo puede llevarnos a más de un estado, lo definimos de la siguiente manera:

$$\Delta(q_1, q_2, \dots, q_k, \sigma) = \bigcup_{i=1}^k \Delta(q_i, \sigma)$$

$\Delta(q_0, \omega)$ ,  $\omega \in \Sigma^*$  es un conjunto de estados si el autómata es AFN. Una cadena será aceptada cuando en dicho conjunto de estados haya algún estado final:  $\Delta(q_0, \omega) \cap \mathcal{F} \neq \emptyset$ .

Dado un AFN, el *lenguaje que acepta* es, evidentemente, el formado por las cadenas que acepta:

$$\mathcal{L}(\mathcal{M}) = \{\omega \in \Sigma^* \mid \Delta(q_0, \omega) \cap \mathcal{F} \neq \emptyset\}$$

Otro tipo de indeterminación es la falta de especificación.

## 2.6. Equivalencia entre AFD y AFN

¿Se pueden simular unos autómatas con otros? Es decir, dado un AFD, ¿podemos construir un AFN que acepte el mismo lenguaje? La respuesta es afirmativa, ya que todo AFD se puede pensar como AFN, donde la  $\Delta$  lleva siempre a un y sólo un estado. Aunque se vea con menos facilidad, al revés también es posible hacerlo.

El procedimiento es siempre finito, porque de un AFN con conjunto de estados  $\mathcal{Q}$  se obtendrá como mucho un AFD con conjunto de estados  $P(\mathcal{Q})$ , de cardinal  $2^{|\mathcal{Q}|}$ .

Los estados finales del AFD equivalente a un AFN dado serán todos aquellos donde haya alguno que era estado final en el AFN de partida.

¿Cuáles son, pues, los motivos para que estudiemos ambos tipos de autómatas si son completamente equivalentes? En ocasiones un problema es más fácil de formular en detalle con AFN.

Lo que acabamos de ver se expresa matemáticamente:

### Teorema.-

Sea  $\mathcal{M} = (\mathcal{Q}, \Sigma, s, \mathcal{F}, \Delta)$  un AFN. Existe un AFD  $\mathcal{M}' = (\mathcal{Q}', \Sigma, s', \mathcal{F}', \delta)$  tal que el *lenguaje que aceptan es el mismo*<sup>3</sup>:

$$\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$$

## 2.7. $\varepsilon$ -transiciones

Las definimos para permitir al autómata “hacer cosas” analizando una cadena *sin consumir nada* de ella.

<sup>3</sup>Ver la demostración por inducción en el libro.

Dado un autómata  $\mathcal{M} = (\Sigma, \mathcal{Q}, \mathcal{F}, s, \Delta)$ :

- sin  $\varepsilon$ -transiciones teníamos que  $\Delta : \mathcal{Q} \times \Sigma \longrightarrow 2^{\mathcal{Q}}$
- ahora:  $\Delta : \mathcal{Q} \times (\Sigma \cup \{\varepsilon\}) \longrightarrow 2^{\mathcal{Q}}$ .

A la hora de representar, tanto en diagramas como en tablas,  $\varepsilon$  se trata como un símbolo más. Las ventajas que aportan se traducen en una mayor potencia a la hora del diseño del autómata.

### 2.7.1. Eliminación de $\varepsilon$ -transiciones

Cuando pasamos de un AFN a un AFD, también hemos de *censurar* las  $\varepsilon$ -transiciones. Para ver cómo se hace esto, definiremos primero:

Dado un estado  $q \in \mathcal{Q}$  de un autómata, llamamos  $\varepsilon$ -**clausura** o  $\varepsilon$ -**cerradura**, y se denota  $\varepsilon - c(q) = \{ \text{conjunto de estados } \mathcal{P}/\mathcal{P} \text{ es accesible desde } \mathcal{Q} \text{ sin consumir nada de la cadena (con } \varepsilon\text{-transiciones}^4) \}$ .

La  $\varepsilon$ -**clausura** de un conjunto de estados es:

$$\varepsilon - c\{p_1, p_2, \dots, p_k\} = \bigcup_{i=1}^k \varepsilon - c(p_i)$$

También se define  $d(q, \sigma) = \{p/ \text{tenemos una conexión de } q \text{ a } p \text{ etiquetada con } \sigma\}$  es decir, consiste en quedarnos sólo con los símbolos.

$$d(\{p_1, p_2, \dots, p_k\}, \sigma) = \bigcup_{i=1}^k d(p_i, \sigma)$$

Lo que se hace entonces para eliminar las  $\varepsilon$ -transiciones es, pues, dado un estado  $q$ , calculamos  $\varepsilon - c(q)$ , después a dónde vamos con un símbolo y de nuevo con  $\varepsilon$ :

$$\varepsilon - c[d(\varepsilon - c(q), \sigma)]$$

Con lo que hemos visto en los apartados anteriores y las  $\varepsilon$ -transiciones, podemos construir un autómata que acepte *cualquier* cadena.

## 2.8. Autómatas finitos y expresiones regulares

Comenzaremos por lo más básico, que no es sino analizar cómo han de ser los **autómatas** que **sólo** aceptan el **vacío**, o la **palabra vacía**, o un **único símbolo**:

Un autómata que acepte la **unión de dos lenguajes**, es decir, dados:

$$\mathcal{L}(\mathcal{M}_1) \text{ asociado a } \mathcal{M}_1 = (\Sigma_1, \mathcal{M}_1, \mathcal{Q}_1, s_1, \mathcal{F}_1, \delta_1)$$

$$\mathcal{L}(\mathcal{M}_2) \text{ asociado a } \mathcal{M}_2 = (\Sigma_2, \mathcal{M}_2, \mathcal{Q}_2, s_2, \mathcal{F}_2, \delta_2)$$

<sup>4</sup>Por ello siempre contiene el propio elemento.

Se representaría:

Y tendría como características:

$$\begin{aligned} \mathcal{L}(\mathcal{M}_1) \cup \mathcal{L}(\mathcal{M}_2) \\ \Sigma = \Sigma_1 \cup \Sigma_2 \\ \mathcal{Q} = \mathcal{Q}_1 \cup \mathcal{Q}_2 \cup s \\ s \text{ (nuevo estado inicial)} \\ \mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \end{aligned} \quad \delta(q, a) \begin{cases} \delta_1(q, a) & \text{si } q \in \mathcal{Q}_1, a \in \Sigma_1 \\ \delta_2(q, a) & \text{si } q \in \mathcal{Q}_2, a \in \Sigma_2 \end{cases}$$

De manera más general incluso, podría escribirse:

$$\mathcal{L}(\mathcal{M}_1) \text{ asociado a } \mathcal{M}_1 = (\Sigma_1, \mathcal{M}_1, \mathcal{Q}_1, s_1, \mathcal{F}_1, \Delta_1)$$

$$\mathcal{L}(\mathcal{M}_2) \text{ asociado a } \mathcal{M}_2 = (\Sigma_2, \mathcal{M}_2, \mathcal{Q}_2, s_2, \mathcal{F}_2, \Delta_2)$$

Con las mismas características y función de transición:

$$\Delta(q, a) \begin{cases} \Delta_1(q, a) & \text{si } q \in \mathcal{Q}_1, a \in \Sigma_1 \\ \Delta_2(q, a) & \text{si } q \in \mathcal{Q}_2, a \in \Sigma_2 \end{cases}$$

Añadiendo:

$$\Delta(s, \varepsilon) = \{s_1, s_2\}$$

Un autómata que aceptase la **concatenación** de dos lenguajes  $\mathcal{L}(\mathcal{M}_1) \cdot \mathcal{L}(\mathcal{M}_2)$  tendría que los estados finales de  $\mathcal{M}_1$  dejarían de serlo porque se irían al inicial de  $\mathcal{M}_2$ :

Con características:

$$\begin{aligned} \Sigma &= \Sigma_1 \cup \Sigma_2 \\ \mathcal{Q} &= \mathcal{Q}_1 \cup \mathcal{Q}_2 \cup s \\ s &= s_1 \\ \mathcal{F} &= \mathcal{F}_2 \end{aligned} \quad \Delta(q, a) \begin{cases} \delta_1(q, a) & \text{si } q \in \mathcal{Q}_1, a \in \Sigma_1 \\ \delta_2(q, a) & \text{si } q \in \mathcal{Q}_2, a \in \Sigma_2 \end{cases}$$

$$\text{Añadiendo } \Delta(p, \varepsilon) = \{s_2\} \quad \forall p \in \mathcal{F}_1$$

En cuanto al **cierre estrella** (un lenguaje concatenado consigo mismo de 0 a todas las veces que queramos):

Aquí también  $Q = Q \cup s$ , estado que se añade para que acepte la palabra vacía, porque no sabemos si el autómata original lo hace; no se podría modificar el estado inicial  $s_1$  y ponerlo como final porque corremos el riesgo de modificar el lenguaje que acepta el autómata.

### ¿Pueden los AF aceptar algo que no sean LR?

La respuesta es *negativa*. Dado un AF, siempre acepta un lenguaje, y ese lenguaje es siempre un LR. Veremos cómo calcularlo.

Sea  $\mathcal{M} = (Q, \Sigma, s, \mathcal{F}, \Delta)$   
 con  $s = q_0$   
 y  $A_i = \{\omega \in \Sigma^* / \Delta(q_i, \omega) \cap \mathcal{F} \neq \emptyset\}$  (cadenas que de  $q_i$  nos llevan a un estado final)  
 $A_0$  es el *lenguaje que acepta el autómata*; a partir de los  $A_i$  intentaremos calcular  $A_0$  como una expresión regular.

**Lema de Arden.**- “Una ecuación de la forma  $X = AX \cup B$  donde  $\varepsilon \notin A$ , tiene una solución única  $X = A^*B$ ”.

$$\text{Demostración.} - A^*B = (A^+ \cup \varepsilon)B = A^+B \cup B = (AA^+)B \cup B = A(A^*B) \cup B$$

$$\left. \begin{array}{l} X = A^*B \cup C \\ C \cap A^*B = \emptyset \end{array} \right\} \text{ solución mayor}$$

Sustituimos en  $X = AX \cup B$ :

$$A^*B \cup C = A(A^*B \cup C) \cup B = A^+B \cup AC \cup B = A^+B \cup B \cup AC = (A^+ \cup \varepsilon)B \cup AC = A^*B \cup AC$$

$$(A^*B \cup C) \cap C = (A^*B \cup AC) \cap C$$

$$C = AC \cap C \Rightarrow C \subseteq AC$$

donde  $A$  no contiene a  $\varepsilon$ , de modo que  $AC$  debería ser mayor que  $C$ , lo que se contradice con lo que acabamos de obtener a no ser que  $C = \emptyset$ , caso en el que  $A^*B$  es *solución única*.

Hemos visto que, siempre que no lleve a engaño, podemos identificar  $\mathcal{L} = \mathcal{L}(\mathcal{M})$ , puesto que el *lenguaje regular* que acepta un autómata finito (ya sea determinista, no determinista o no determinista con  $\varepsilon$ -transiciones) se representa con una expresión regular.

Definamos ahora el **lenguaje complementario** de un lenguaje dado  $\mathcal{L}(\mathcal{M})$  aceptado por un autómata  $\mathcal{M}$  construido sobre un alfabeto  $\Sigma^*$ :

$$\Sigma^* - \mathcal{L}(\mathcal{M}) = \overline{\mathcal{L}(\mathcal{M})}$$

¿Es el *lenguaje complementario* un lenguaje regular? Dado un autómata finito determinista<sup>5</sup>, se tiene que  $\delta(s, \omega) = p$  y  $p \in \mathcal{F}$  o bien  $p \notin \mathcal{F}$ . Según la definición,

<sup>5</sup>El uso de esta suposición, la del determinismo, es muy significativa; si no se usase, no se podría hacer el razonamiento.

$\overline{\mathcal{M}} = (\Sigma, Q, s, Q - \mathcal{F}, \delta)$  de modo que ahora una cadena que antes llegaba a un estado de  $\mathcal{F}$  no es aceptada, o, lo que es lo mismo, se aceptan todas las que no se aceptaban en  $\mathcal{M}$ , de modo que **sí**, es un *lenguaje regular*.

Dados dos *lenguajes regulares*  $\mathcal{L}_1, \mathcal{L}_2$ , también la **intersección**  $\mathcal{L}_1 \cap \mathcal{L}_2$  es un *lenguaje regular*. Se puede demostrar:

$$\mathcal{L}_1, \mathcal{L}_2 \text{ L.R.} \Rightarrow \overline{\mathcal{L}_1}, \overline{\mathcal{L}_2} \text{ L.R.}$$

$$\text{por definición } \overline{\mathcal{L}_1} \cup \overline{\mathcal{L}_2} \text{ L.R.}$$

$$\text{de modo que } \overline{(\overline{\mathcal{L}_1} \cup \overline{\mathcal{L}_2})} \Rightarrow \mathcal{L}_1 \cap \mathcal{L}_2 \text{ L.R.}$$

## 2.9. Propiedades de los lenguajes regulares

Veremos a continuación un **lema** que nos servirá para probar que un lenguaje NO es regular, no obstante, debemos tener cuidado porque no vale para verificar que sí lo es.

### Lema

Sea  $\mathcal{L}$  un *lenguaje regular infinito*<sup>6</sup>. Existe una constante  $n$  de forma que si  $\omega \in \mathcal{L}$ ,  $|\omega| \geq n$  se tiene que  $\omega = uvx$  con

$$\begin{aligned} |v| &\geq 1 \\ |uv| &\leq n \\ uv^i x &\in \mathcal{L} \quad \forall i \geq 0 \end{aligned}$$

Es decir, cualquier cadena de longitud suficiente se puede dividir en subcadenas de modo que la primera parte se puede repetir tantas veces como queramos obteniendo siempre cadenas del propio lenguaje.

Gracias al lema anterior, denominado **lema del bombeo**, podemos deducir algunas otras propiedades de los lenguajes regulares:

### Teorema

Sea  $\mathcal{M}$  un AF con  $k$  estados:

1.  $\mathcal{L}(\mathcal{M}) \neq \emptyset \Leftrightarrow \mathcal{M}$  acepta alguna cadena de longitud menor que  $k$ .
2.  $\mathcal{L}(\mathcal{M})$  infinito  $\Leftrightarrow \mathcal{M}$  acepta alguna cadena de longitud  $k \leq n < 2k$ .

### DEMOSTRACIÓN

1. “ $\Leftarrow$ ” es trivial.

“ $\Rightarrow$ ”: sea  $\omega \in \mathcal{L}(\mathcal{M})$ ; si  $|\omega| < k$ , queda demostrado, y si  $|\omega| \geq k$  usamos

---

<sup>6</sup>Un lenguaje finito siempre es regular, porque siempre se puede representar por un autómata finito. La propiedad que enuncia este lema la verifican todos los lenguajes regulares, aunque no sean infinitos. Comprobaremos que un lenguaje no es regular demostrando que no la tiene.

el lema del bombeo que nos dice que 
$$\left. \begin{array}{l} \omega = uvx \\ |uv| \leq k \\ |v| \geq 1 \end{array} \right] \rightarrow uv^i x \in \mathcal{L} \quad \forall i \geq 0.$$

Si hacemos  $i = 0$ ,  $ux \in \mathcal{L}$  y  $|ux| < |\omega|$ ; en este punto, hacemos de nuevo el mismo razonamiento anterior, que se detendrá en algún punto, puesto que la longitud de una cadena es siempre finita.

2. “ $\Leftarrow$ ”: sea  $\omega \in \mathcal{L}$ ,  $|\omega| = n$   $k \leq n < 2k \rightarrow$  condiciones del lema del bombeo  $\rightarrow \left[ \begin{array}{l} \omega = uvx \\ |uv| \leq k \\ |v| \geq 1 \end{array} \right] \rightarrow uv^i x \in \mathcal{L} \quad \forall i \geq 0.$  Si tenemos una cadena

por cada  $i$ , indudablemente tenemos infinitas, de modo que el lenguaje es infinito.

“ $\Rightarrow$ ”: si  $\mathcal{L}(\mathcal{M})$  es infinito, entonces  $\exists n = |\omega| \geq k$ . Si  $k \leq n < 2k$ , queda demostrado; sino, si  $2k \leq n$ , aplicamos el lema del bombeo

$$\left. \begin{array}{l} \omega = uvx \\ |uv| \leq k \\ |v| \geq 1 \end{array} \right] uv^0 x \in \mathcal{L}.$$
 Como  $|v| \leq k$ ,  $|ux| \geq k$ , y aplicamos el mismo razonamiento otra vez hasta encontrar una cadena de longitud entre  $k$  y  $2k - 1$ .

### Propiedad

Si  $\mathcal{L}$  y  $\mathcal{K}$  son dos lenguajes regulares, entonces también lo es  $\mathcal{L} \cap \mathcal{K}$ .

Esta propiedad es muy usada para determinar la regularidad de los lenguajes.



# Capítulo 3

## Lenguajes Independientes del contexto

### 3.1. Gramáticas regulares

#### Definición

Una **gramática** es un *conjunto de 4 elementos*:

$$\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{S}, \mathcal{P})$$

donde:

$\Sigma$  es el *alfabeto* de símbolos o los *símbolos terminales*  
 $\mathcal{N}$  es un conjunto de *símbolos no terminales* o *variables*  
 $\mathcal{S}$  es el *símbolo inicial*  
 $\mathcal{P}$  son *reglas de producción, sustitución o reescritura*

Las *reglas de producción*, a veces se llamadas simplemente *producciones*, tienen la forma  $A \rightarrow \omega$  o bien  $(A, \omega)$  donde  $A$  es un símbolo no terminal y  $\omega$  es una cadena de símbolos del alfabeto o terminales, y no terminales. Hay una serie de reglas para  $\omega$ :

- $\omega$  puede contener como máximo un símbolo no terminal.
- si  $\omega$  tiene un símbolo no terminal, entonces éste es el símbolo que está en el extremo derecho.

Un ejemplo:

$$\begin{aligned}\Sigma &= \{a, b\} \\ \mathcal{N} &= \{S, A\}\end{aligned}$$

$$\begin{aligned}\mathcal{P} : S &\longrightarrow bA \\ A &\longrightarrow aaA \quad | \quad b \quad | \quad \varepsilon\end{aligned}$$

Se intenta obtener a partir del símbolo inicial de la gramática, usando  $\mathcal{P}$  las veces que haga falta con el objetivo de llegar a una cadena de símbolos terminales.

$$\mathcal{S} \Rightarrow bA \Rightarrow \begin{array}{l} baaA \Rightarrow baaaaA \Rightarrow \dots \Rightarrow baaaa \dots aab \\ bb \\ b \end{array}$$

Para simbolizar la aplicación de una regla varias veces se utiliza el símbolo  $\xRightarrow{*}$  :

$$\mathcal{S} \xRightarrow{*} bA \xRightarrow{*} b(aa)^*A \xRightarrow{*} b(aa)^*b$$

$$\mathcal{S} \xRightarrow{*} b(aa)^*b$$

$$\mathcal{S} \xRightarrow{*} b(aa)^*$$

y para la aplicación cero o más veces  $\xRightarrow{+}$  .

No son válidas las producciones:

$$\begin{array}{ll} A \xRightarrow{+} aBC & \text{porque contiene dos símbolos no terminales} \\ A \xRightarrow{+} Ba & \text{porque el símbolo no terminal no está a la derecha} \end{array}$$

### 3.2. Gramáticas y lenguajes regulares

Las gramáticas sirven para *generar lenguajes*. Denotaremos por  $\mathcal{L}(\mathcal{G})$  el lenguaje generado por una gramática  $\mathcal{G}$ . Decimos que una gramática nos genera un lenguaje de *cadena*s tales que *desde el símbolo inicial* podemos llegar a ellas *aplicando las reglas de producción*:

$$\mathcal{L}(\mathcal{G}) = \{\omega \in \Sigma^* \mid \mathcal{S} \xRightarrow{*} \omega\}$$

Sea  $\mathcal{L}$  un lenguaje regular cualquiera. Veamos que existe una gramática  $\mathcal{G}$  que lo genera:

Dado un lenguaje  $\mathcal{L}$ , sabemos que existe un A.F. que lo acepta. Llamemos  $\mathcal{M}$  (el lenguaje será entonces  $\mathcal{L}(\mathcal{M})$ ) a dicho autómata:

$$\mathcal{M} = (\mathcal{Q}, \Sigma, s, \mathcal{F}, \delta)$$

Construyamos  $\mathcal{L}(\mathcal{G})$  y veamos que también es aceptado por  $\mathcal{M}$ .

Sea la gramática:

$$\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{S}, \mathcal{P})$$

con

$$\begin{aligned}\Sigma &= \Sigma && \text{(mismo alfabeto de símbolos)} \\ \mathcal{N} &= \mathcal{Q} \\ \mathcal{S} &= s \\ \mathcal{P} &= \{(q, ap) / \delta(q, a) = p\} \cup \{(q, \varepsilon) / q \in \mathcal{F}\} \\ &&& \text{(asociamos producciones y transiciones)}\end{aligned}$$

Al revés, si tenemos una gramática  $\mathcal{G}$  y queremos construir el AF que acepte su lenguaje:

Sea  $\mathcal{G}$  una gramática regular,

$$\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{S}, \mathcal{P})$$

construyamos esta vez un AFN<sup>1</sup> definido de la siguiente manera:

$$\mathcal{M} = (\mathcal{Q}, \Sigma, s, \mathcal{F}, \Delta)$$

de suerte que

$$\begin{aligned}\mathcal{Q} &= \mathcal{N} \cup \{f\} \text{ y otros, donde } f \text{ es un estado nuevo} \\ s &= \mathcal{S} \\ \mathcal{F} &= \{f\} \\ \Delta &= 2 \text{ posibilidades}\end{aligned}$$

Las dos posibilidades para  $\Delta$  son:

- (a)  $A \implies \sigma_1, \sigma_2, \dots, \sigma_n B$  añadimos  $n - 1$  nuevos estados  $q_1, q_2, \dots, q_{n-1}$ :  
 $\Delta(A, \sigma_1 \sigma_2 \dots \sigma_n) = \Delta(q_1, \sigma_2 \dots \sigma_n) = \Delta(q_2, \sigma_3 \dots \sigma_n) = \dots = \Delta(q_{n-1}, \sigma_n) = B$
- (b)  $A \implies \sigma_1, \sigma_2, \dots, \sigma_n$  añadimos igual que antes  $n - 1$  nuevos estados  $q_1, q_2, \dots, q_{n-1}$ :  
 $\Delta(A, \sigma_1 \sigma_2 \dots \sigma_n) = \Delta(q_1, \sigma_2 \dots \sigma_n) = \Delta(q_2, \sigma_3 \dots \sigma_n) = \dots = \Delta(q_{n-1}, \sigma_n) = f$   
con  $f$  el nuevo estado final

Puede distinguirse entre *producciones lineales por la derecha* ( $A \rightarrow \omega B$ ) y *producciones lineales por la izquierda* ( $A \rightarrow B\omega$ ); ambas son *gramáticas regulares* y *generan el mismo lenguaje*, siempre que no se permita la coexistencia de las dos a un tiempo. No obstante, pueden darse casos de equivalencias en que no se cumpla ésto pero la gramática sea regular, como por ejemplo:

$$\begin{array}{l} S \rightarrow aA \quad | \quad \varepsilon \\ A \rightarrow Sb \end{array}$$

<sup>1</sup>Es más simple y sabemos que son equivalentes.

### 3.3. Gramáticas independientes del contexto

#### Definición

Una **gramática independiente del contexto** es una 4-tupla tal que

$$\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{S}, \mathcal{P})$$

donde:

- $\mathcal{N}$  = conjunto de símbolos no terminales
- $\Sigma$  = alfabeto ó conjunto de símbolos terminales
- $\mathcal{S}$  = símbolo inicial (símbolo no terminal)
- $\mathcal{P}$  = conjunto de producciones

En este caso, al conjunto de producciones simplemente le pedimos que sea:

$$\mathcal{P} \subseteq \mathcal{N} \times (\mathcal{N} \cup \Sigma)^*$$

es decir, no le imponemos las condiciones que dictábamos para las G.R.

El *lenguaje* que genera una G.I.C. se denomina **lenguaje independiente del contexto**, y, por lo que acabamos de reseñar, toda G.R. es una G.I.C. (el recíproco no es cierto), y por tanto todo L.R. es un L.I.C. De hecho, los L.R. son un *subconjunto estricto* de los L.I.C.

Se dicen *independientes del contexto* porque un símbolo no terminal puede sustituirse independientemente de lo que lo acompañe; en otro caso, se llaman *dependientes* y las veremos más adelante.

### 3.4. Árboles de derivación

En ocasiones puede resultar útil representar gráficamente la secuencia de *derivaciones* por producciones que se sigue para llegar a una cadena de símbolos terminales a partir del símbolo inicial. Dicho gráfico tiene forma de árbol y se conoce como **árbol de derivación**.

El *árbol de derivación* se construye creando un nodo raíz con el símbolo inicial como etiqueta. Los hijos de este nodo raíz serán cada uno de los símbolos que aparezcan a la derecha de la producción usada para reemplazar el símbolo inicial. Todo nodo cuya etiqueta sea un símbolo no terminal, tendrá como hijos los símbolos del lado derecho de su producción asociada. Los nodos que no tengan hijos serán siempre de símbolos terminales, y leyendo todos los nodos hoja de izquierda a derecha se obtiene la cadena que deriva el árbol.

Desgraciadamente, todas las derivaciones posibles para una cadena no tienen por qué dar lugar al mismo árbol de derivación. A raíz de este handicap surge el concepto de **gramática ambigua**:

Una **gramática**  $\mathcal{G}$  es **ambigua** si hay dos o más árboles de derivación distintos para una misma cadena.

A veces puede encontrarse una gramática alternativa y equivalente a una gramática ambigua, es decir, que genere el mismo lenguaje. Sin embargo, si todas las G.I.C. para un lenguaje son ambiguas (no hay ninguna gramática no ambigua que lo genere), entonces el lenguaje no sólo es *ambiguo* sino que se dice que es un **lenguaje inherentemente ambiguo**.

Usando el concepto de **derivación por la izquierda**<sup>2</sup>, una gramática será ambigua si existen dos derivaciones por la izquierda distintas para una misma cadena (lo que implica que existen dos árboles de derivación distintos). La definición de **derivación por la izquierda** es:

Derivación en la cual el símbolo no terminal al que le aplicamos una producción o regla de reescritura es siempre el situado más a la izquierda o en el extremo izquierdo de la cadena.

### 3.5. Simplificación de gramáticas independientes del contexto

Dada una G.I.C. ¿cómo podemos saber si todos sus símbolos y reglas de producción no son redundantes, si son todos necesarios,...

A continuación presentamos una serie de algoritmos, denominados *algoritmos de limpieza* que, aplicados sobre G.I.C., nos proporcionan su “simplificación”. Es decir, dada una gramática  $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{S}, \mathcal{P})$ , la transformaremos en otra gramática  $\mathcal{G}' = (\mathcal{N}', \Sigma, \mathcal{S}, \mathcal{P}')$  *equivalente* (es decir,  $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$ ) de forma que

$$\forall A \in \mathcal{N}' \quad A \overset{*}{\Rightarrow} \omega \quad \text{con } \omega \in \Sigma^*$$

El primer objetivo es, pues, quedarnos con los símbolos no terminales que nos llevan a algún sitio, sobre los cuales, aplicando reglas de producción, podemos llegar a una cadena de símbolos no terminales.

#### Algoritmo 1

1. Se toman los símbolos no terminales que dan directamente cadenas de símbolos terminales aplicando alguna regla de reescritura.
2. Se toman los símbolos no terminales que dan símbolos terminales de los que ya hemos escogido en el paso anterior.
3. Se repite hasta que no se incorpore ningún símbolo nuevo.

Una vez hecho esto, nos interesa eliminar los símbolos a los que no se puede llegar desde el estado inicial  $\mathcal{S}$ ; construimos  $\mathcal{G}' = (\mathcal{N}', \Sigma', \mathcal{S}, \mathcal{P}')$  mediante:

#### Algoritmo 2

1. Empezamos con  $\mathcal{N}' = \mathcal{S}$  y con  $\mathcal{P}'$  y  $\Sigma'$  a  $\emptyset$ .

<sup>2</sup>Análogamente se definiría por la derecha.

2. Se escoge  $A \in \mathcal{N}'$   $A \rightarrow \omega \in \mathcal{P}$ , se mete en  $\mathcal{P}'$  y todo símbolo no terminal  $B$  de  $\omega$  se introduce en  $\mathcal{N}'$  y todo símbolo terminal  $\sigma$  de  $\omega$  se mete en  $\Sigma'$ .
3. Se repite hasta que no se puedan añadir nuevas producciones.

Es importante el *orden* en que se aplican estos dos algoritmos a una gramática; debe hacerse en el orden en que los hemos definido.

### Producciones $\varepsilon$

Son producciones de la forma  $A \rightarrow \varepsilon$ .  $A$  se dice **anulable** si desde  $A$  se puede llegar a  $\varepsilon$  tras haber aplicado una serie de reglas o producciones:

$$A \xRightarrow{*} \varepsilon$$

Si  $\varepsilon$  no está en  $\mathcal{L}(\mathcal{G})$ , todas las producciones  $\varepsilon$  pueden ser eliminadas; pero si no, no puede hacerse tan a la ligera. Es esencial identificar los símbolos no terminales anulables. Para ello contamos con:

### Algoritmo 3

1. Identificamos los símbolos anulables directamente mediante una producción,  $A \rightarrow \varepsilon$ , y se introducen en  $\mathcal{N}$ .
2. Si se tiene una producción  $B \rightarrow \omega$  tal que todos los símbolos de  $\omega$  estén en  $\mathcal{N}$ , metemos  $B$  en  $\mathcal{N}$ .
3. Repetir hasta que no se meta nada más en  $\mathcal{N}$ .

Una vez identificados los símbolos anulables, se procede de la siguiente manera:

- Si el lenguaje  $\mathcal{L}(\mathcal{G})$  generado por la gramática  $\mathcal{G}$ ,  $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{S}, \mathcal{P})$ , *no contiene*  $\varepsilon$ , existe la posibilidad de, como ya comentamos, eliminar todas las  $\varepsilon$ -producciones (si contiene  $\varepsilon$  al menos tendrá que quedar una, como veremos). Identificamos  $\mathcal{N}$  con el método anterior y modificamos nuestro conjunto de producciones de la siguiente manera:

Se sustituyen producciones de la forma  $B \rightarrow X_1, X_2, \dots, X_n$  por producciones  $B \rightarrow Y_1, Y_2, \dots, Y_n$  de forma que:

- $Y_i = X_i$  si  $X_i$  no es anulable.
  - $Y_i = X_i$  o  $\varepsilon$  si  $X_i$  es anulable.
  - $Y_i \neq \varepsilon \quad \forall i$  (ya que si no tenemos  $B \rightarrow \varepsilon$ , que es lo que queremos evitar).
- Si el lenguaje contiene  $\varepsilon$ , simplemente se consiente que sea añadida la producción  $\mathcal{S} \rightarrow \varepsilon$ .

Una producción de la forma  $A \rightarrow B$  se denomina **producción unitaria** o **no generativa** y suelen complicar las gramáticas de modo innecesario. El siguiente es un algoritmo para eliminarlas en la medida de lo posible (sólo afecta a  $\mathcal{P}$ ), y para ponerlo en práctica se define:

$$\text{Unitario}(A) = \{B \in \mathcal{N} \quad / \quad A \xrightarrow{*} B \text{ usando sólo producciones unitarias}\}$$

Nótese que  $A \in \text{Unitario}(A)$  porque  $A \rightarrow A$  sin usar producción alguna.

#### Algoritmo 4

1. Inicializamos  $\mathcal{P}'$  para que contenga todo  $\mathcal{P}$  (esta vez, en lugar de añadir, quitaremos).
2.  $\forall A \in \mathcal{N}$  obtenemos  $\text{Unitario}(A)$ .
3.  $\forall A \quad / \quad \text{Unitario}(A) \neq A \Rightarrow \forall B \in \text{Unitario}(A)$  de modo que se tenga  $B \rightarrow \omega \in \mathcal{P}$ , añadimos la producción  $A \rightarrow \omega$  a  $\mathcal{P}'$ .
4. Eliminamos las producciones unitarias de  $\mathcal{P}'$ .

## 3.6. Forma normal de Chomsky

### Definición

Dada un G.I.C., existe una gramática equivalente que tiene una forma especial: *no contiene producciones  $\varepsilon$*  y todas las *producciones* son de la forma  $A \rightarrow a$  o  $A \rightarrow BC$ .

El *árbol de producción* de una *gramática en forma normal de Chomsky* es un árbol **binario**.

Dada una gramática  $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{S}, \mathcal{P})$ , supuesto que  $\varepsilon \notin \mathcal{L}(\mathcal{G})$  (su lenguaje no contiene la palabra vacía), puede transformarse en una gramática en forma normal de Chomsky de la siguiente forma:

- ◇ Se eliminan todas las producciones  $\varepsilon$ , los símbolos inútiles y las producciones unitarias.
- ◇ Una vez realizado el paso anterior, puede asegurarse que  $A \rightarrow \omega \Rightarrow |\omega| \geq 1$ . Si  $|\omega| = 1$ , entonces  $\omega$  es un símbolo terminal ( $\omega \in \Sigma$ ); y si  $|\omega| > 1$  entonces se puede escribir  $\omega = X_1 X_2 \dots X_n$ .
- ◇ Si  $X_i$  es un símbolo terminal  $\sigma$ , lo sustituimos por un nuevo no terminal  $C_\sigma$  y añadimos la producción  $C_\sigma \rightarrow \sigma$ . Tras este paso, todas las producciones son de la forma  $A \rightarrow \omega$  con  $\omega$  un símbolo terminal o cadena de no terminales.
- ◇ Por último, se eliminan las cadenas con más de dos símbolos no terminales en el lado derecho, reemplazando cada producción  $A = B_1 B_2 \dots B_n$  con

$n \geq 2$ , por  $n - 1$  producciones, de la siguiente manera:

$$\begin{aligned} A &\longrightarrow B_1 D_1 \\ D_1 &\longrightarrow B_2 D_2 \\ &\quad \dots \\ D_{n-2} &\longrightarrow B_{n-1} B_n \end{aligned}$$

donde los  $D_i$  son nuevos no terminales.

Todo L.I.C. que no contenga  $\varepsilon$  puede ser generado por medio de una G.I.C. en forma normal de Chomsky. En caso de que la contenga, simplemente permitimos a mayores sólo que tenga la producción  $S \longrightarrow \varepsilon$ .

Si tenemos un árbol correspondiente a una G.I.C. en forma normal de Chomsky de profundidad  $m + 1$ , significa que hay al menos un camino de esa longitud, que tendrá por tanto  $m + 2$  nodos. De ellos,  $m + 1$  serán no terminales, y el último (hoja), terminal (todo camino tendrá como máximo un símbolo terminal). Además, la longitud de la cadena más larga que se podrá derivar con ese árbol será  $2^m$  (ya que en el nivel  $i$  el número máximo de nodos es  $2^{i3}$ ), o, lo que es lo mismo, para derivar una cadena de longitud mayor o igual que  $2^m$  la profundidad del árbol ha de ser por lo menos  $m + 1$ .

### Lema de bombeo para L.I.C.

Sea  $\mathcal{L}$  un L.I.C. que no contiene  $\varepsilon$ ,  $\varepsilon \notin \mathcal{L}$ . Podemos afirmar que existe un entero  $k$  para el cual si  $z$  es una cadena de  $\mathcal{L}$ ,  $z \in \mathcal{L}$ , y la longitud de  $z$  es mayor que  $k$ ,  $|z| > k$ , entonces  $z$  se puede escribir o descomponer como:

$$z = uvwxy$$

con las condiciones

$$\begin{aligned} |uvw| &\leq k \\ |v| + |x| &> 0 \quad (\text{no pueden ser vacíos al mismo tiempo}) \end{aligned}$$

y verificando que las cadenas generadas mediante el siguiente *bombeo* están en el lenguaje  $\mathcal{L}^4$ :

$$uv^iwx^iy \in \mathcal{L} \quad \forall i \geq 0$$

La utilidad fundamental de este lema es encontrar una cadena de la longitud que nos indica de forma que no se pueda descomponer de esa manera  $\longrightarrow$  lenguaje no I.C.

### Resultado

Dada una gramática  $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{S}, \mathcal{P})$  con  $|\mathcal{N}| = n$ , ¿es el lenguaje que genera,  $\mathcal{L}(\mathcal{G})$  *finito* o *infinito*?

$\mathcal{L}(\mathcal{G})$  es **infinito**  $\iff$  existe una cadena  $z \in \mathcal{L}(\mathcal{G})$  tal que su longitud verifica  $2^{n-1} < |z| \leq 2^{n-1} + 2^n$ .

<sup>3</sup>El nivel del nodo raíz es el nivel 0.

<sup>4</sup>Ver demostración en el libro.

Un lenguaje se dice **decidible** si, dada cualquier cadena  $z$ , se puede examinar si pertenece o no a dicho lenguaje (es decir, si se puede derivar de su símbolo inicial aplicando reglas de su gramática). Para ver si un lenguaje es *decidible* o no tenemos el siguiente lema:

Sea  $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{S}, \mathcal{P})$  una G.I.C. sin  $\varepsilon$ -producciones y en forma normal de Chomsky. Sea  $x$  una subcadena de  $\Sigma^*$ . Se puede determinar, para cada  $A \in \mathcal{N}$  y para cada subcadena  $w$  de  $x$  si  $A \xrightarrow{*} w^5$ .

Uno de los algoritmos que nos permite seguir las producciones correctas para llegar del símbolo inicial a aceptar la cadena (ahora que sabemos que se puede hacer) es el **algoritmo CYK**, que fue construido por Cooke, Younger y Kasami.

### 3.7. Propiedades de los lenguajes independientes del contexto

Como ya podíamos anticipar, la unión, concatenación y el cierre estrella de lenguajes independientes del contexto dan como resultado lenguajes independientes del contexto. Sin embargo, la intersección no tiene por qué serlo. Lo que sí ocurrirá será que la intersección entre un L.I.C. y un L.R. (que, como sabemos, es a su vez un L.I.C. pero más restrictivo) sí da como resultado un L.I.C.<sup>6</sup>

Tampoco el complementario de un L.I.C. es independiente del contexto.

### 3.8. Autómatas de pila

Construiremos ahora un nuevo tipo de autómatas, resultantes de proveer a nuestros AF de una *pila*, con las características habituales de esta estructura.

#### Definición

Un **autómata de pila no determinista** (ADPND) es una 7-tupla:

$$\mathcal{M} = (\mathcal{Q}, \Sigma, \Gamma, \Delta, s, \mathcal{F}, z)$$

donde

$\mathcal{Q}$  = conjunto finito de estados

$\Sigma$  = alfabeto de entrada

$\Gamma$  = alfabeto de pila<sup>7</sup>

$\Delta$  = regla de transición<sup>8</sup>

$s \in \mathcal{Q}$  = estado inicial (de partida)

$z \in \Gamma$  = símbolo inicial de la pila<sup>9</sup>

$\mathcal{F} \subseteq \mathcal{Q}$  = conjunto de estados finales (de aceptación)

<sup>5</sup>Ver demostración en el libro.

<sup>6</sup>Esto se usa también para probar que algunos no son L.I.C. Véase la demostración en el libro.

<sup>7</sup>Puede ser distinto de  $\Sigma$ , son los símbolos que se manejan en la pila.

<sup>8</sup>No función, ya que es no determinista.

<sup>9</sup>A veces se representa por #, significa el fondo de la pila y suponemos siempre que ya está marcado al empezar; sino, se colocaría en el primer estado.

La *regla de transición*  $\Delta$  se define por medio de ternas del tipo  $(q, \sigma, \gamma)$  con  $q \in \mathcal{Q}$ ,  $\sigma \in \sum \cup \{\varepsilon\}$  y  $\gamma \in \Gamma$ , y tras aplicarla se obtiene  $(p, w)$  donde  $p$  es el estado siguiente y  $w$  una cadena de  $\Gamma^*$  que es la que se introduce en la pila.

#### CONSIDERACIONES

- Si se da  $\Delta(p, \sigma, \gamma) = \emptyset$  entonces la máquina se para.
- Pedimos que en la pila necesariamente haya un símbolo (la marca de inicio o fin de pila):

$$\Delta \subseteq \mathcal{Q} \times \left( \sum \cup \{\varepsilon\} \right) \times \underbrace{\Gamma}_{\text{no } \Gamma \cup \{\varepsilon\}} \times \mathcal{Q} \times \Gamma^*$$

Se llama **descripción instantánea** de un ADPND a la terna  $(q, w, u)$ , donde  $q$  es el estado actual,  $w$  la cadena de entrada que queda por procesar y  $u$  el contenido de la pila, con el símbolo de la cima en el extremo de la izquierda.

### 3.9. Autómatas de pila y lenguajes independientes del contexto

Los autómatas de pila no deterministas aceptan los L.I.C:

Sea  $\mathcal{G} = (\mathcal{N}, \sum, \mathcal{S}, \mathcal{P})$  una G.I.C. y  $\mathcal{L} = \mathcal{L}(\mathcal{G})$  el lenguaje que genera. Buscamos el ADPND que la representa; dicho autómatata cumple:

$$\begin{aligned} \mathcal{Q} &= \{q_1, q_2, q_3\} \\ \Gamma &= \mathcal{N} \cup \sum \cup \{z\} \\ \mathcal{F} &= \{q_3\} \\ s &= q_1 \end{aligned}$$

y su regla  $\Delta$  está compuesta por cuatro tipos de transiciones:

1. Introducción del símbolo inicial en la pila:  $\Delta(q_1, \varepsilon, z) = \{(q_2, Sz)\}$ .
2. Cambio de no terminales:  $\Delta(q_2, \varepsilon, A) = \{(q_2, w)\}$  de modo que  $A \rightarrow w$  es una producción de  $\mathcal{P}$  y  $A$  es un símbolo terminal de  $\mathcal{N}$ .
3. Lectura de terminales:  $\Delta(q_2, a, a) = \{(q_2, \varepsilon)\}$  con  $a \in \sum$  no terminal.
4. Lectura del símbolo de fin de pila:  $\Delta(q_2, \varepsilon, z) = \{(q_3, z)\}$

Podemos afirmar, pues, que si  $\mathcal{L}$  es un L.I.C., entonces existirá un ADPND  $\mathcal{M}$  que lo acepta,  $\mathcal{L} = \mathcal{L}(\mathcal{M})$ .

Es más, si  $\mathcal{L}$  es aceptado por un ADPND, entonces  $\mathcal{L}$  es un L.I.C.

### 3.10. Autómatas de pila deterministas

Igual que pasaba cuando comparábamos AFD y AFND, si tenemos un ADPND en el cual en cada movimiento (transición) se hace una sola cosa, podemos construir un ADPD que acepte exactamente el mismo lenguaje.

No obstante, si necesitamos hacer más de un movimiento, por ejemplo en unas ocasiones (según lo que nos llegue y el estado en que estemos) almacenar un símbolo y en otras compararlo con la cima de la pila y borrar el primer elemento de ésta, no podríamos hacerlo.

Por tanto, y al revés que ocurría con los lenguajes regulares, con los lenguajes independientes del contexto los ADPD y los ADPND *no generan los mismos lenguajes*. Es más, hay mucha mayor cantidad de lenguajes para ADPND:

### 3.11. Forma normal de Greibach

Ya para terminar este tema, veremos otra forma normal que se usa para las gramáticas independientes del contexto que es de gran utilidad (sobre todo en algoritmos) por el modo en que restringe la posición en que pueden aparecer los símbolos terminales y los no terminales.

Antes, veremos una serie de resultados y definiciones, para cuya demostración y ejemplos remitimos al libro de texto:

#### Resultado

Si  $A \rightarrow \alpha B \gamma$  es una producción de una GIC y  $B \rightarrow \beta_1 | \beta_2 | \dots | \beta_m$  son todas las producciones que tienen a  $B$  en su lado izquierdo, entonces  $A \rightarrow \alpha B \gamma$  se puede reemplazar por  $A \rightarrow \alpha \beta_1 \gamma | \alpha \beta_2 \gamma | \dots | \alpha \beta_m \gamma$ .

#### Definición

Una producción de la forma  $A \rightarrow \alpha A$ , donde  $\alpha$  es una cadena de terminales y no terminales, se dice que es **recursiva por la derecha**. Análogamente se define la *recursividad por la izquierda*<sup>10</sup> cuando  $A \rightarrow A \alpha$ .

#### Definición

Una producción de la forma  $A \rightarrow \alpha A$ , donde  $\alpha$  es una cadena de terminales y no terminales, se dice que es **recursiva por la izquierda**. Análogamente se define la *recursividad por la derecha* cuando  $A \rightarrow A \alpha$ .

Intentaremos evitar la recursividad por la izquierda, y para ello:

<sup>10</sup>Este tipo de recursividad no es deseable.

### Resultado

Si tenemos una GIC  $\mathcal{G}$  y  $A$  un no terminal cuyas producciones son:

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

puede construirse una gramática equivalente añadiendo un nuevo *no terminal*  $Z$  y reemplazando todas las producciones anteriores por:

$$\begin{aligned} A &\rightarrow \beta_1|\beta_2|\dots|\beta_m|\beta_1Z|\beta_2Z|\dots|\beta_mZ \\ Z &\rightarrow \alpha_1|\alpha_2|\dots|\alpha_n|\alpha_1Z|\alpha_2Z|\dots|\alpha_nZ \end{aligned}$$

Ahora sí, llegamos a la definición que da nombre a este apartado:

### Definición

Una GIC está en **forma normal de Greibach (FNG)** si todas las producciones son de la forma  $A \rightarrow a\alpha$ , donde  $a$  es un símbolo terminal y  $\alpha$  una cadena de terminales y no terminales<sup>11</sup>.

Todo lenguaje independiente del contexto puede generarse mediante una gramática independiente del contexto que esté en forma normal de Greibach.

---

<sup>11</sup>Una definición equivalente afirma que  $\alpha$  puede ser una cadena de no terminales.

# Capítulo 4

## Máquinas de Turing

En este tema estudiaremos un dispositivo matemático en el que se basan las máquinas (computadoras) actuales<sup>1</sup>, fruto de la inventiva del matemático A. Turing (1936), y que en honor a él recibe el nombre de **Máquina de Turing**.

Entre otras cosas, con una Máquina de Turing podremos, además de aceptar lenguajes más generales que los regulares y los independientes del contexto, decidir si un problema es resoluble o no, e incluso analizar si algo es un algoritmo.

### 4.1. Definición de Máquina de Turing

Una **Máquina de Turing** es algo muy sencillo: es una colección de *celdillas* que se extienden infinitamente. Cada celdilla almacena un símbolo y se tiene una *cabeza lectora/escritora* que puede moverse sobre ellas y leer o escribir en ellas. Formalmente:

#### Definición

Una **Máquina de Turing** es una colección de 7 elementos:

$$\mathcal{M} = (\mathcal{Q}, \Sigma, \Gamma, s, \hbar, \mathcal{F}, \delta)$$

donde

$\mathcal{Q}$  es un conjunto finito de estados

$\Sigma$  es un alfabeto de entrada

$\Gamma$  es otro alfabeto, *alfabeto de la cinta*

$s \in \mathcal{Q}$  es el estado inicial

$\hbar \in \Gamma$  es el *símbolo blanco*

$\mathcal{F} \subseteq \mathcal{Q}$  es el conjunto de estados finales o de aceptación

$\delta : \mathcal{Q} \times \Gamma \rightarrow \mathcal{Q} \times \Gamma \times \{L, R\}$  es la función de transición

La función de transición  $\delta$  transforma pares  $(q, \sigma)$  de estado actual y símbolo de la cinta a ternas  $(p, t, X)$  de nuevo estado (estado siguiente), símbolo escrito en la cinta y movimiento hacia la izquierda o la derecha de la cabeza lectora/escritora. Como se puede

---

<sup>1</sup>La *Tesis de Church-Turing* afirma que cualquier cosa que haga una máquina puede hacerlo una Máquina de Turing. Esta aseveración no ha sido ni probada ni refutada, pero rige como admitida.

observar, dentro de su indeterminismo (habrá pares para los que no esté definida), cuando está definida,  $\delta$  es *determinista*.

#### NOTACIÓN

Representaremos las **descripciones instantáneas** de una Máquina de Turing de dos maneras:

- $(q_i, w_1\sigma w_2)$ , con  $q_i$  el estado actual,  $w_1$  la parte de la cinta que nos queda a la izquierda (suponemos que leemos de izquierda a derecha normalmente),  $\sigma$  el símbolo sobre el que está la cabeza lectora/escritora y  $w_2$  el trozo de cadena que queda a la derecha.
- $a_1a_2 \dots a_{k_1}q_ia_{k_2} \dots a_n$ , donde la cabeza lectora/escritora está sobre  $a_k$ , el estado actual es  $q_i$ , etc<sup>2</sup>.

Entre las particularidades que harán especiales a estos dispositivos tenemos que en las Máquinas de Turing no hay transiciones en estados finales y no se necesita consumir enteramente las cadenas para aceptarlas o rechazarlas.

## 4.2. Máquinas de Turing como aceptadoras de lenguajes

Una Máquina de Turing se puede comportar como un aceptador de lenguajes. Se dirá que una cadena  $w$  es aceptada si después de una secuencia de movimientos la Máquina de Turing lleva a un estado final y para. Formalmente:

#### Definición

Sea  $\mathcal{M} = (\mathcal{Q}, \Sigma, \Gamma, s = q_1, \hbar, \mathcal{F}, \delta)$  una Máquina de Turing. El **lenguaje aceptado** por  $\mathcal{M}$  es:

$$\mathcal{L}(\mathcal{M}) = \{\omega \in \Sigma^* \mid q_1\omega \vdash^* \omega_1p\omega_2 \text{ para } p \in \mathcal{F} \text{ y } \omega_i \in \Gamma^*\}$$

Rechazar una cadena supone que la máquina se queda parada en un estado que no es final o bien entra en un bucle infinito.

Un lenguaje que es aceptado por una Máquina de Turing se conoce como lenguaje **recursivamente enumerable**, término que incluye los L.I.C. y representa aquéllos cuyas cadenas pueden ser enumeradas. El subconjunto de los lenguajes *recursivamente enumerables* que se paran en un estado no final cuando la cadena es aceptada se denominan **lenguajes recursivos**.

#### Definición

Se dice que una función de cadena  $f$  es **Turing computable** si existe una Máquina de Turing  $\mathcal{M} = (\mathcal{Q}, \Sigma, \Gamma, q_1, \hbar, \mathcal{F}, \delta)$  para la cual  $q_1\omega \vdash^* q_f u$  para algún estado final  $q_f \in \mathcal{F}$ , cuando  $f(\omega) = u$ .

<sup>2</sup>Remitimos a las anotaciones para estudiar su representación gráfica.

### 4.3. Construcción de Máquinas de Turing

Construiremos una serie de Máquinas de Turing sencillas de modo que al combinarlas podamos crear otras más complejas. Para ello definiremos en primer lugar la **combinación** o **composición** de máquinas de Turing.

Sean  $\mathcal{M}_1 = (\mathcal{Q}_1, \Sigma, \Gamma, s_1, \hbar, \mathcal{F}_1, \delta_1)$  y  $\mathcal{M}_2 = (\mathcal{Q}_2, \Sigma, \Gamma, s_2, \hbar, \mathcal{F}_2, \delta_2)$  dos Máquinas de Turing sobre los mismos alfabetos de entrada  $\Sigma$  y de cinta  $\Gamma$ . Se supone que  $\mathcal{Q}_1 \cap \mathcal{Q}_2 = \emptyset$ .

La **composición** de  $\mathcal{M}_1$  y  $\mathcal{M}_2$  es la Máquina de Turing  $\mathcal{M} = (\mathcal{Q}, \Sigma, \Gamma, s, \hbar, \mathcal{F}, \delta)$  denotada por  $\mathcal{M}_1\mathcal{M}_2$ , donde

$$\mathcal{Q} = \mathcal{Q}_1 \cup \mathcal{Q}_2$$

$$s = s_1$$

$$\mathcal{F} = \mathcal{F}_2$$

$$\delta(q, \sigma) = \begin{cases} \delta_1(q, \sigma) & \text{si } q \in \mathcal{Q}_1 \text{ y } \delta_1(q, \sigma) \neq (p, \tau, \mathcal{X}) \quad \forall p \in \mathcal{F}_1 \\ \delta_2(q, \sigma) & \text{si } q \in \mathcal{Q}_2 \\ (s_2, \tau, \mathcal{X}) & \text{si } q \in \mathcal{Q}_1 \text{ y } \delta_1(p, \sigma) = (p, \tau, \mathcal{X}) \text{ para algún } p \in \mathcal{F}_1 \end{cases}$$

La composición de  $\mathcal{M}_1$  y  $\mathcal{M}_2$  se comporta como  $\mathcal{M}_1$  hasta que ésta llega a un estado final, momento en que cambia al estado inicial de  $\mathcal{M}_2$  y se comporta como tal hasta que termina.

#### 4.3.1. Algunas Máquinas de Turing especiales

Denotaremos por:

$\mathbf{R}_{\hbar}$  a la Máquina de Turing que busca el primer blanco que haya a la derecha de la posición actual<sup>3</sup>.

$\mathbf{L}_{\hbar}$  aquélla que busca hacia la izquierda el primer símbolo de la cinta que no sea blanco.

a la Máquina de Turing que escribe como salida un único símbolo,  $a$ , y permanece sobre la misma celda.

R la Máquina de Turing que se mueve una celda a la derecha.

Todas estas Máquinas de Turing tienen *simétricas* que se denotan de modo análogo.

### 4.4. Modificaciones de las Máquinas de Turing

Cualquier modificación que se pueda hacer de la Máquina de Turing que hemos definido puede añadirle mayor flexibilidad, pero en cualquier caso será siempre simulable con ésta, ya que todas tendrán la misma potencia computacional (de cálculo).

Veremos, no obstante, algunas pequeñas variaciones:

<sup>3</sup>Remitimos al libro para ver su descripción y función delta.

- Nuestra Máquina de Turing requiere siempre que la cabeza lectora-escritora se mueva en cada transición, ya que la función de transición es  $\delta : \mathcal{Q} \times \Gamma \rightarrow \mathcal{Q} \times \Gamma \times \{R, L\}$ . Si la modificamos de forma que quede:

$$\delta : \mathcal{Q} \times \Gamma \rightarrow \mathcal{Q} \times \Gamma \times \{R, L, \mathbf{S}\}$$

donde  $\mathbf{S}$  significa *no mover* la cabeza lectora-escritora, habilitaremos la posibilidad de transiciones en las que leamos y escribamos y permanezcamos sobre la misma celda. Siempre podría simularse leyendo, escribiendo, moviéndose y regresando con un movimiento inverso.

- Otra modificación sencilla es aquélla en la que cada celda de la cinta de nuestra máquina se divide en subceldas. Se dice que la cinta tiene *múltiples pistas*. Se puede interpretar como una nueva función de transición:

$$\delta : \mathcal{Q} \times \Gamma^n \rightarrow \mathcal{Q} \times \Gamma^n \times \{R, L\}$$

Se puede simular, si se tienen  $k$  pistas, con una Máquina de Turing cuyo alfabeto de cinta está formado por todas las  $k$ -tuplas sobre  $\Gamma$ .

- También podríamos pensar en una Máquina de Turing cuya cinta se extendiese infinitamente pero *en una sola dirección* (normalmente hacia la derecha). Se simularía simplemente con marcar con un símbolo una celda a partir de la cual no se podía ir hacia la izquierda. Al revés, la Máquina de Turing con una cinta infinita en un sólo sentido puede simular una Máquina de Turing con una cinta infinita en ambos sentidos si cuenta con dos pistas, cada una de las cuales representa una rama de la cinta doblemente infinita a partir de un punto de referencia o control.
- Llamaremos Máquina de Turing *multicinta* a la variación para la cual se tienen varias cintas, cada una de ellas con su propia cabeza de lectura-escritura controlada independientemente. Sin duda, es la más potente, pero también puede ser simulada con una cinta con  $2k + 1$  pistas (y ya hemos visto cómo se simula este tipo de modificación).
- La eliminación del requerimiento de que la regla de transición  $\delta$  sea una función producirá la construcción de Máquinas de Turing *no deterministas*. Simplemente, la función se define sobre *Partes de* el conjunto destino de la función original. Su simulación se hará con una Máquina de Turing de 3 cintas, donde la primera tendrá la entrada, la segunda generará una “copia” sobre la que se ensayarán sucesivamente las posibilidades que se almacenan en la tercera hasta que se dé (o no) con la aceptación de la cadena.

## 4.5. Máquina de Turing Universal

La **Máquina de Turing Universal** es una Máquina de Turing que a partir de una máquina de Turing  $\mathcal{M}$  y una cadena de entrada  $w$  simula el comportamiento de  $\mathcal{M}$  sobre la cadena  $w$ .

Si codificamos una Máquina de Turing  $\mathcal{M} = (\mathcal{Q}, \Sigma, \Gamma, s, h, \mathcal{F}, \delta)$  que tenga un único estado final o de aceptación<sup>4</sup>  $q_2$  perteneciente a  $\mathcal{Q} = (q_1, q_2, \dots, q_n)$ , y donde  $\Gamma =$

<sup>4</sup>Para forzarlo basta con definirlo y añadir las transiciones que lleven de los otros a él.

$(\sigma_1, \sigma_2, \dots, \sigma_m)$  de forma que  $q_i, \sigma_i$  se representen por cadenas de  $i$  unos y las directivas  $L$  y  $R$  mediante 1 y 11 respectivamente, usando los ceros como separadores, una *Máquina de Turing Universal*  $\mathcal{M}_u$  se puede implementar como una Máquina de Turing de tres cintas para la que  $\Sigma = \{0, 1\}$ . La primera cinta contiene la codificación de ceros y unos de  $\mathcal{M}$ . La segunda, la codificación del contenido de la cinta de  $\mathcal{M}$  y la tercera guarda el estado actual de  $\mathcal{M}$ .  $\mathcal{M}_u$  analiza y compara el contenido de las cintas segunda y tercera con el de la primera hasta que encuentra una transición para la configuración codificada o hasta que agota las posibilidades. Se cumple, además, que la cadena final que quede en la cinta de  $\mathcal{M}_u$  será la codificación de la cadena que hubiera quedado en  $\mathcal{M}$ .



# Capítulo 5

## Máquinas de Turing y Lenguajes

### 5.1. Lenguajes aceptados por Máquinas de Turing

Ya vimos en el tema anterior las definiciones de *lenguajes recursivamente enumerables* y *lenguajes recursivos*. De su definición (ver sección 4.2, página 34) se deduce que todo lenguaje recursivo es recursivamente enumerable, pero el recíproco no siempre es cierto porque el contenido es estricto<sup>1</sup>.

### 5.2. Lenguajes Recursivos y Recursivamente Enumerables

Dado un lenguaje regular o, lo que es lo mismo, dado un autómata finito determinista, puede construirse una Máquina de Turing que lo simule haciendo:

$$\begin{aligned} \mathcal{Q}' &= \mathcal{Q} \cup \{q'\} \\ \Sigma' &= \Sigma \\ \Gamma &= \Sigma \cup \{\hbar\} \\ \mathcal{F}' &= \{q'\} \\ \delta'(q, \sigma) &= (\delta(q, \sigma), \sigma, R) && \text{para } q \in \mathcal{Q}, \sigma \in \Sigma' \\ \delta'(q, \hbar) &= (q', \hbar, S) && \forall q \in \mathcal{F} \text{ donde } S \text{ significa no moverse} \end{aligned}$$

Es decir, que si  $\mathcal{L}$  es **regular** entonces también es **recursivo**<sup>2</sup>.

Del mismo modo, un autómata de pila (lenguajes independientes del contexto) no determinista se puede emular con una Máquina de Turing de dos cintas, una que haga las transiciones y otra que controle el apilamiento y desapilamiento. De modo que también podemos afirmar que si  $\mathcal{L}$  es **independiente del contexto**, entonces es también **recursivo**.

#### Teorema

Si  $\mathcal{L}_1$  y  $\mathcal{L}_2$  son *lenguajes recursivos*, entonces  $\mathcal{L}_1 \cup \mathcal{L}_2$  también lo es.

#### Lema

---

<sup>1</sup>Es más, hay una cantidad bastante mayor de lenguajes recursivamente enumerables que de recursivos.

<sup>2</sup>Para todo AFND se puede construir un AFD equivalente, y viceversa.

Sea  $\mathcal{L}$  un *lenguaje recursivo*; entonces existe una Máquina de Turing  $\mathcal{M} = (\mathcal{Q}, \Sigma, \Gamma, s, \hbar, \mathcal{F}, \delta)$  que lo acepta, parando para cualquier entrada.

Sea  $\mathcal{M}' = (\mathcal{Q}, \Sigma, \Gamma, s, \hbar, \mathcal{Q} - \mathcal{F}, \delta)$ . Esta Máquina de Turing para en un estado de  $\mathcal{F}$  cuando se ejecuta sobre una cadena de  $\mathcal{L}$ . Cuando la cadena no pertenezca al lenguaje, parará sobre algún estado que no sea de  $\mathcal{F}$ , es decir, parará en algún estado de  $\mathcal{Q} - \mathcal{F}$ .

En conclusión, si  $\mathcal{L}$  es un *lenguaje recursivo*, entonces  $\Sigma^* - \mathcal{L}$  también.

Esta es una propiedad que en general no se cumple para los lenguajes recursivamente enumerables.

Existen lenguajes recursivamente enumerables para los cuales sus inversos no son recursivamente enumerables:

Para una cadena arbitraria de un lenguaje recursivo, la Máquina de Turing que la acepta para siempre. Un **algoritmo** se puede considerar, pues, una *Máquina de Turing que para siempre*, esto es, se asocia con un lenguaje recursivo.

Siguiendo el mismo razonamiento, los lenguajes recursivamente enumerables representan *problemas irresolubles*, para los que existen ejemplos a los que no se puede decir ni SI ni NO.

Por último, los lenguajes que ni siquiera son recursivamente enumerables, encarnan *problemas no abordables*. Puesto que nuestra capacidad de computación no los alcanza, concluimos con la certeza de que este nuestro mundo de la computación es limitado.

### Teorema

Si  $\mathcal{L}_1$  y  $\mathcal{L}_2$  son *lenguajes recursivamente enumerables*, entonces  $\mathcal{L}_1 \cup \mathcal{L}_2$  también lo es<sup>3</sup>.

### Teorema

Si  $\mathcal{L}$  es un lenguaje recursivamente enumerable para el cual  $\Sigma^* - \mathcal{L}$  también es recursivamente enumerable, entonces  $\mathcal{L}$  es un lenguaje recursivo.

Se demuestra partiendo del supuesto que tanto  $\mathcal{L}$  como su opuesto son recursivamente enumerables. Entonces, analizando cualquier cadena de  $\Sigma^*$  tendremos que o bien pertenece a uno o bien al otro, esto es, forzosamente uno de los dos la aceptará. Si tenemos una máquina que nos simule su unión, forzosamente parará siempre. Para conseguir una modificación de la misma que acepte  $\mathcal{L}$  sólo hay que sustituir los estados de aceptación de  $\tilde{\mathcal{L}}$  por estados de rechazo, de modo que seguirá deteniéndose siempre y aceptará  $\mathcal{L}$ , esto es,  $\mathcal{L}$  habrá de ser un lenguaje recursivo.

### Teorema

Un lenguaje  $\mathcal{L}$  es recursivamente enumerable si y sólo si  $\mathcal{L}$  es enumerado por alguna máquina de Turing.

---

<sup>3</sup>Ver demostración en libro.

### 5.3. Gramáticas no restringidas y Lenguajes RE

Definiremos en esta sección un último tipo de gramáticas, el más general y menos restrictivo de todos los que hemos visto, resultante de “liberalizar” el lado izquierdo de las producciones de una gramática independiente del contexto.

#### Definición

Una **gramática no restringida** o **gramática estructurada por frases** es una 4-tupla

$$\mathcal{G} = (\mathcal{N}, \Sigma, s, P)$$

donde:

$\mathcal{N}$  es un alfabeto de símbolos *no terminales*

$\Sigma$  es un alfabeto de símbolos *terminales* tal que  $\mathcal{N} \cap \Sigma = \emptyset$

$s \in \mathcal{N}$  es el *símbolo inicial* (no terminal)

$P$  es un *conjunto de producciones* de la forma  $\alpha \rightarrow \beta$  con  $\begin{cases} \alpha \in (\mathcal{N} \cup \Sigma)^+ \\ \beta \in (\mathcal{N} \cup \Sigma)^* \end{cases}$

#### Teorema

Si  $\mathcal{G}$  es una gramática no restringida, entonces  $\mathcal{L}$  es un lenguaje recursivamente enumerable.

#### DEMOSTRACIÓN

Para una gramática no restringida  $\mathcal{G}$  se pueden listar todas sus cadenas del tipo  $S \Rightarrow w$  ya que como el número de producciones es finito, también lo serán ellas. De igual modo se puede proceder para las cadenas derivadas en dos, tres, . . . pasos. Puesto que todo el  $\mathcal{L}(\mathcal{G})$  podrá ser listado de esta manera, podrá ser enumerado por una Máquina de Turing, es decir, es recursivamente enumerable.

A la inversa, a partir de los movimientos de una Máquina de Turing (transiciones), se puede generar un gramática no restringida. Esta afirmación nos demuestra que lenguajes recursivamente enumerables se *asocian* con gramáticas no restringidas:

#### Teorema

Un lenguaje  $\mathcal{L}$  es recursivamente enumerable si y sólo si  $\mathcal{L} = \mathcal{L}(\mathcal{G})$  para alguna gramática no restringida  $\mathcal{G}$ .

## 5.4. Lenguajes sensibles al contexto y jerarquía de Chomsky

### Teorema

Una gramática  $\mathcal{G} = (\mathcal{N}, \Sigma, s, P)$  es una **gramática sensible al contexto**<sup>4</sup> si todas las producciones son de la forma  $\alpha \rightarrow \beta$  con  $\alpha, \beta \in (\mathcal{N} \cup \Sigma)^+$  y  $|\alpha| \leq |\beta|$ .

Según esta definición, en cada paso que aplicamos una derivación la cadena o bien permanece igual (de la misma longitud) o bien aumenta, nunca decrece (esto no lo teníamos en las gramáticas sin restricciones). Debido a esto reciben el nombre de *gramáticas no contráctiles*. Podemos ver que además supone un problema para la representación de  $\varepsilon$ , la cadena vacía; permitimos que la regla  $s \rightarrow \varepsilon$  pertenezca a  $P$ , como única excepción.

Las gramáticas sensibles al contexto producen una clase de lenguajes, los *lenguajes sensibles al contexto*, que está estrictamente situada entre los lenguajes independientes del contexto y los lenguajes recursivos, aunque en principio pudiésemos pensar que son más generales las I.C. porque permiten sustituir los símbolos sin restricciones. No permiten, por el contrario, cambios de este tipo (con terminales y no terminales a la izquierda). Además, ya que cualquier G.I.C. se puede poner en Forma Normal de Chomsky, las producciones serán de la forma  $A \rightarrow a$  o  $A \rightarrow BC$ , que cumplen las especificaciones para ser G.S.C. Así, tenemos que los lenguajes sensibles al contexto son algunos más que los lenguajes independientes del contexto.

### Lema

Sea  $\mathcal{G} = (\mathcal{N}, \Sigma, s, P)$  una gramática sensible al contexto. Entonces existe una gramática sensible al contexto  $\mathcal{G}_1 = (\mathcal{N}_1, \Sigma, s_1, P_1)$  que genera el mismo lenguaje y en la que  $s_1$  nunca aparece al lado derecho de una producción de  $P_1$ .

### Lema

Sea  $\mathcal{G} = (\mathcal{N}, \Sigma, s, P)$  una gramática sensible al contexto. Entonces existe una Máquina de Turing que para sobre toda entrada y acepta  $\mathcal{L}(\mathcal{G})$ <sup>5</sup>.

### Lema

Si  $\mathcal{L}$  es un lenguaje sensible al contexto, entonces es recursivo.

DEMOSTRACIÓN

La demostración consiste en probar que se puede generar una representación del lenguaje utilizando sólo ceros y unos<sup>6</sup>.

### Lema

Hay lenguajes recursivos que no son lenguajes sensibles al contexto.

L. Sensibles al Contexto  $\subsetneq$  L. Rec.  $\subsetneq$  L. Rec. Enumerables

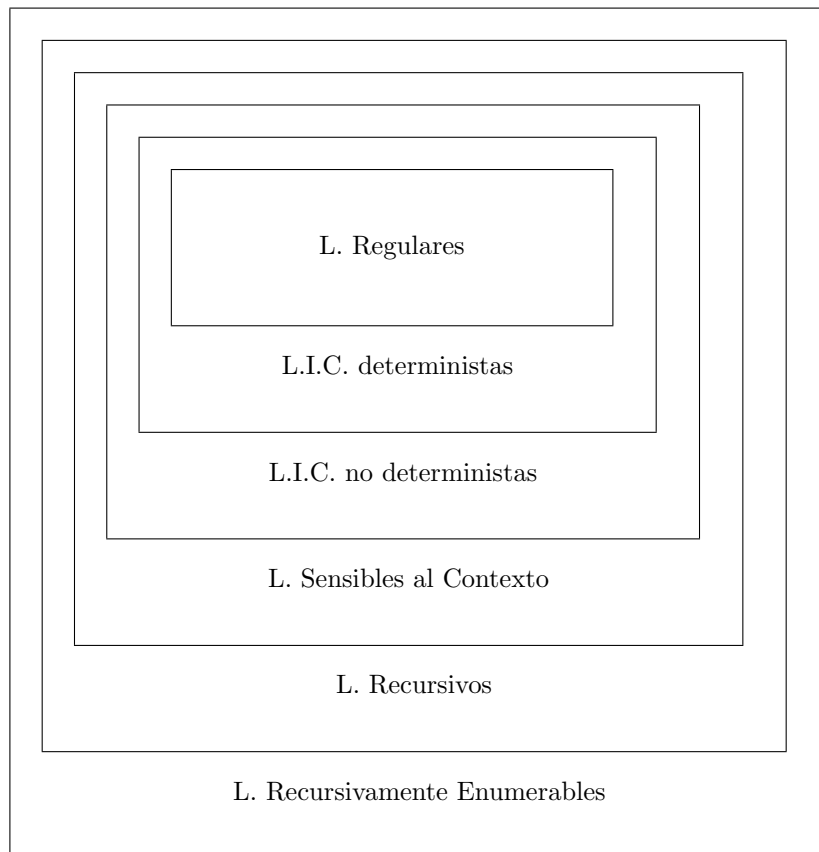
<sup>4</sup>O gramática dependiente del contexto.

<sup>5</sup>Remitimos al libro para su demostración.

<sup>6</sup>Remitimos a la descripción detallada del proceso en el libro.

En 1959, Noam Chomsky clasificó las gramáticas en cuatro familias: gramáticas no restringidas, sensibles al contexto, independientes del contexto y regulares, conocidas como gramáticas de tipo 0, 1, 2 y 3 respectivamente. Los lenguajes que resultan de dichas gramáticas también se identifican como lenguajes de tipo 0, 1, 2 y 3.

El siguiente diagrama de lenguajes se conoce como la **jerarquía de Chomsky**:





# Capítulo 6

## Resolubilidad

Como ya dijimos en su momento (sección 4.2, página 34) una función *Turing computable* es aquella para la que existe una Máquina de Turing que computa  $f(\omega)$  para toda  $\omega$  del dominio de  $f$ .

Por ejemplo, se tiene como función computable la *función característica* de un lenguaje recursivo  $\mathcal{X}_L$ :

$$\mathcal{X}_L(\omega) = \begin{cases} 1 & \text{si } \omega \in L \\ 0 & \text{si } \omega \notin L \end{cases}$$

Sin embargo, la función característica de un lenguaje recursivamente enumerable no es computable ya que hay cadenas para las que la Máquina de Turing que lo acepta no para.

En este tema nos ocuparemos de los llamados **problemas de decisión**, como son por ejemplo decir si, dada una gramática independiente del contexto  $\mathcal{G}$ ,  $\mathcal{L}(\mathcal{G})$  es vacío (ver sección 2.9, página 18), o, dada una gramática sensible al contexto, si una cadena  $\omega$  pertenece al lenguaje que genera,...

### Definición

Se dice que los problemas de decisión son **resolubles** si existe un algoritmo que es capaz de responder *sí* o *no* a cada uno de sus casos. Si el algoritmo no existe, el problema se dice **irresoluble**.

## 6.1. El problema de la parada

El *problema irresoluble* más conocido es el **problema de la parada** para Máquinas de Turing, que se enuncia como sigue:

Sea  $\mathcal{M}$  una Máquina de Turing arbitraria con un alfabeto de entrada  $\Sigma$ , y sea  $\omega \in \Sigma^*$ . ¿Parará  $\mathcal{M}$  con la cadena  $\omega$  como cadena de entrada?

El *problema de la parada* para las Máquinas de Turing es *irresoluble*.

La irresolubilidad del problema de la parada se usa para demostrar que otros problemas también son irresolubles. La estrategia consiste en demostrar que si un determinado problema se puede resolver ello implicaría que el problema de la parada también es resoluble.

### 6.1.1. El problema de la cinta en blanco

Este problema consiste en decidir si una Máquina de Turing parará cuando comience su ejecución con una cinta en blanco. Este problema también es irresoluble.

#### DEMOSTRACIÓN

Sea una Máquina de Turing  $\mathcal{M}$  y una cadena cualquiera  $\omega \in \Sigma$ . Sea  $\mathcal{M}'$  la Máquina de Turing que comienza con una cinta en blanco, escribe en ella  $\omega$  y posteriormente se reposiciona en la primera celda de  $\omega$  en el estado  $q_1$  (primer estado de  $\mathcal{M}$ ) y la emula a partir de ahí. Si tuviésemos un algoritmo que determinase si una Máquina de Turing arbitraria que comienza con una cinta en blanco para, podríamos determinar si  $\mathcal{M}'$  para. Claro que  $\mathcal{M}'$  para si y sólo si la Máquina de Turing  $\mathcal{M}$  original para con la cadena  $\omega$  como entrada, de modo que tendríamos una solución para el problema de la parada, algo que sabemos que no es cierto y que nos lleva a afirmar, consecuentemente, que el problema de la cinta en blanco es irresoluble.

Esta técnica de demostración (reducir un problema al problema de la parada) se denomina *reducción*.

## 6.2. El problema de correspondencia de Post

Un caso del **PCP**<sup>1</sup> se llama *sistema de correspondencia de Post* y está compuesto por tres elementos:

- Un alfabeto  $\Sigma$ .
- Dos conjuntos  $A$  y  $B$  con el mismo número de elementos, que serán cadenas no vacías de  $\Sigma$ :  $A = \{u_1, u_2, \dots, u_k\}$  y  $B = \{v_1, v_2, \dots, v_k\}$ .

Una **solución** para un caso del problema de correspondencia de Post es una secuencia de índices  $i_1, i_2, \dots, i_n$  para los que se cumple:

$$u_{i_1} u_{i_2} \dots u_{i_n} = v_{i_1} v_{i_2} \dots v_{i_n}$$

El problema de determinar si un sistema de correspondencia de Post arbitrario tiene solución es el *problema de correspondencia de Post*, y no hay ningún algoritmo que lo lleve a cabo, de modo que, como veremos, es irresoluble.

Definiremos el **PCP modificado** (*PCPM*) como el PCP para el que buscamos una solución en la que la secuencia de índices debe comenzar por  $i_1 = 1$ , es decir,  $u_{i_1} = u_1$  y  $v_{i_1} = v_1$ .

#### Lema

Si el PCP es resoluble, entonces el PCPM también es resoluble.

---

<sup>1</sup>Abreviaremos así el *Problema de correspondencia de Post*.

## DEMOSTRACIÓN

Sean  $A = \{u_1, u_2, \dots, u_k\}$  y  $B = \{v_1, v_2, \dots, v_k\}$  una muestra del PCPM y supongamos que cada  $u_i = a_{i_1} a_{i_2} \dots a_{i_{m_i}}$  y cada  $v_i = b_{i_1} b_{i_2} \dots b_{i_{n_i}}$ .

Definimos para cada  $i$ :

$$y_i = a_{i_1} \$ a_{i_2} \$ \dots \$ a_{i_{m_i}} \$ \quad y \quad z_i = \$ b_{i_1} \$ b_{i_2} \$ \dots \$ b_{i_{n_i}}$$

donde  $\$$  es un nuevo símbolo que no está en  $\Sigma$ . Sea  $\%$  otro símbolo que no pertenece a  $\Sigma$ ; construimos también:

$$\begin{aligned} y_0 &= \$y_1 & z_0 &= z_1 \\ y_{k+1} &= \% & z_{k+1} &= \$\% \end{aligned}$$

Tenemos, pues el nuevo “juego de fichas”,

que es un PCP, problema que hemos supuesto que tiene solución. Claro que este PCP tiene solución *si y sólo si* el original PCPM tiene solución, ya que cualquier solución debe comenzar por  $y_0$ , ya que es el único  $y_i$  que empieza por  $\$$ , que es el primer carácter de todos los  $z_i$ , así que  $i_1 = 0$ . Además, el último  $i$  debe ser  $k + 1$ , pues sólo  $z_{k+1}$  e  $y_{k+1}$  coinciden en su último símbolo.

Por tanto, las fichas se ordenarán,

es decir,

$$\underbrace{\$a_{1_1} \$a_{1_2} \dots \$a_{1_{m_1}} \$ \dots \$}_{\text{ficha 0}} \underbrace{\%}_{\text{ficha k+1}} = \underbrace{\$b_{1_1} \$b_{1_2} \$ \dots \$b_{1_{n_1}} \$ \dots \$}_{\text{ficha 0}} \underbrace{\%}_{\text{ficha k+1}}$$

Eliminando los  $\$$  se tiene

$$u_1 u_{i_2} \dots u_{i_r} = v_1 v_{i_2} \dots v_{i_r}$$

de donde se deduce que si el PCP es resoluble, lo es el PCPM.

Gracias a este lema podremos demostrar que el PCP es irresoluble si demostramos que el PCPM, que es más estructurado y por tanto simplificará el proceso, es irresoluble. Para ello, demostraremos que si el PCPM tuviese solución, el problema de la parada también la tendría.

### Demostración de la irresolubilidad del PCPM

Supongamos que el PCPM es resoluble, que existe un algoritmo que determina si tiene solución un PCPM cualquiera. Pretendemos demostrar que hay un algoritmo que puede determinar si una Máquina de Turing arbitraria  $\mathcal{M}$  parará con una entrada  $\omega$ .

Cualquier Máquina de Turing puede convertirse en una máquina que sólo para en un estado de aceptación, haciendo que si para en algún estado de no aceptación se introduzca en un bucle o algo así. Esta modificación no afecta al lenguaje que acepta la máquina.

Sea  $\mathcal{M} = (\mathcal{Q}, \Sigma, \Gamma, s, \hbar, \mathcal{F}, \delta)$  y una cadena  $\omega$  sobre  $\Sigma$ . Construiremos una muestra del PCPM para el que la determinación de si tiene solución (que suponemos que podemos hacer) sea equivalente a determinar si  $\mathcal{M}$  para sobre la entrada  $\omega$ .

(Ver el resto de la demostración en el libro, página 248 en adelante.)

El razonamiento realizado, pues, tiene el siguiente esquema:

1. Demostramos que el *problema de la parada* es irresoluble.
2. Argumentamos que si el *PCP* es resoluble  $\Rightarrow$  el *PCPM* es resoluble.
3. Asimismo, si el *PCPM* es resoluble  $\Rightarrow$  el *problema de la parada* también lo será.
4. Pero el *problema de la parada* es irresoluble, de modo que el *PCPM* se deduce irresoluble.
5. Y puesto que el *PCPM* es irresoluble  $\Rightarrow$  el *PCP* es irresoluble.

### 6.3. Irresolubilidad en lenguajes independientes del contexto

Como ya vimos en los temas 2 y 3 muchas de las preguntas que se pueden responder para lenguajes regulares se convierten en preguntas irresolubles para gramáticas independientes del contexto. La irresolubilidad del PCP proporciona una herramienta útil para demostrar la irresolubilidad de algunos problemas.

El planteamiento será el de costumbre: sobre una pregunta acerca de gramáticas o lenguajes independientes del contexto, si es decidible, el PCP lo sería también.

Antes de nada veremos, pues, cómo se construye una gramática independiente del contexto a partir de un sistema de correspondencia de Post arbitrario:

Sean  $\Sigma$ ,  $A = \{u_1, u_2, \dots, u_k\}$  y  $B = \{v_1, v_2, \dots, v_k\}$ , así como  $I = \{1, 2, \dots, k\}$ . Sean entonces  $\mathcal{G}_A = (\{S_A\}, \Sigma \cup \{1, 2, \dots, k\}, S_A, P_A)$  y  $\mathcal{G}_B = (\{S_B\}, \Sigma \cup \{1, 2, \dots, k\}, S_B, P_B)$  donde  $S_A$  y  $S_B$  son símbolos nuevos,  $P_A$  contiene producciones de la forma  $S_A \rightarrow u_i S_A a_i | u_i a_i$  y  $P_B$  de la forma  $S_B \rightarrow v_i S_B i | u_i i$ .

Por la forma de estas dos gramáticas se tienen dos resultados:

- Si este PCP tiene solución, será de la forma:

$$u_{i_1} u_{i_2} \dots u_{i_n} = v_{i_1} v_{i_2} \dots v_{i_n}$$

que se generan por las gramáticas definidas en el siguiente formato:

$$u_{i_1} u_{i_2} \dots u_{i_n} i_n i_{n-1} \dots i_2 i_1 = v_{i_1} v_{i_2} \dots v_{i_n} i_n i_{n-1} \dots i_2 i_1$$

y por tanto ambos lenguajes tienen algún elemento en común,

$$\mathcal{L}(\mathcal{G}_A) \cap \mathcal{L}(\mathcal{G}_B) \neq \emptyset$$

- Por otro lado, si tenemos alguna cadena

$$\omega \in \mathcal{L}(\mathcal{G}_A) \cap \mathcal{L}(\mathcal{G}_B)$$

quiere decir que

$$\omega = u_{i_1} u_{i_2} \dots u_{i_n} i_n i_{n-1} \dots i_2 i_1 = v_{i_1} v_{i_2} \dots v_{i_n} i_n i_{n-1} \dots i_2 i_1$$

es decir, el PCP tiene una solución:

$$u_{i_1} u_{i_2} \dots u_{i_n} = v_{i_1} v_{i_2} \dots v_{i_n}$$

Por tanto, podemos deducir el lema que se expone a continuación.

### Lema

El *problema de la intersección vacía* para dos gramáticas independientes del contexto arbitrarias es *irresoluble*.

También se tienen los siguientes resultados:

### Teorema

Para una gramática independiente del contexto arbitraria, la pregunta  $\mathcal{L}(\mathcal{G}) = \Sigma^*$  es *irresoluble*.

No tenemos un algoritmo que resuelva esta cuestión. Lo demostraremos:

#### DEMOSTRACIÓN

Poder decir si  $\mathcal{L}(\mathcal{G}) = \Sigma^*$  es equivalente a poder responder si  $\mathcal{L}(\bar{\mathcal{G}}) = \emptyset$ . Sea un PCP cualquiera y construyamos  $\mathcal{G}_A$  y  $\mathcal{G}_B$ . Llamemos  $\mathcal{L} = \mathcal{L}(\bar{\mathcal{G}}_A) \cup \mathcal{L}(\bar{\mathcal{G}}_B)$ <sup>(2)</sup>. Entonces,  $\bar{\mathcal{L}} = \mathcal{L}(\mathcal{G}_A) \cap \mathcal{L}(\mathcal{G}_B)$  y como esto último no lo podemos saber, se deduce que no podemos saber si  $\mathcal{L}(\mathcal{G}) = \Sigma^*$ .

### Teorema

El *problema de la ambigüedad* para gramáticas independientes del contexto es *irresoluble*.

---

<sup>2</sup>En general, no tiene por qué cumplirse que el complementario de un lenguaje independiente del contexto sea independiente del contexto (así como tampoco se cumple para la intersección de dos). Se cumple sin embargo en los regulares y también para  $\mathcal{L}(\mathcal{G}_A)$  y  $\mathcal{L}(\mathcal{G}_B)$ .

## DEMOSTRACIÓN

Sea un PCP y  $\mathcal{G}_A$  y  $\mathcal{G}_B$  las gramáticas que generamos a partir de él, que no son ambiguas. Construyamos otra gramática  $\mathcal{G}$  para la que:

$$\begin{aligned}\mathcal{N} &= \{S, S_A, S_B\} \\ \mathcal{P} &= P_A \cup P_B \cup \{S \rightarrow S_A | S_B\} \\ S & \\ T & \text{ el mismo}\end{aligned}$$

Si pudiésemos decir *si* es o *no* ambigua, podrían pasar dos cosas:

1. Si es *ambigua* es que hay una cadena que se puede generar de estas dos formas:

$$\begin{aligned}S &\Rightarrow S_A \Rightarrow \dots \\ S &\Rightarrow S_B \Rightarrow \dots\end{aligned}$$

Claro que una cadena así estaría en la intersección de los lenguajes generados por las dos gramáticas relativas al PCP, es decir, daría una solución al PCP, lo cual sabemos que no es posible.

2. Recíprocamente, si *no* es *ambigua*, no hay ninguna cadena en la intersección, pero esto hace que sepamos con certeza que el PCP no tiene solución, y sabemos que para dicho problema *no sabemos decir* si la tiene o no.

De modo que se sigue que el problema de la ambigüedad no puede ser resoluble.

# Capítulo 7

## Funciones $\mu$ -recursivas

En este tema analizaremos *qué funciones necesitamos* y *cómo las construiremos* para sacar toda la potencia de una Máquina de Turing, esto es, aquéllas necesarias para la construcción de lenguajes de programación imperativos (qué es lo mínimo que ha de tenerse)<sup>1</sup>.

Así pues, ¿qué clase de funciones estudiaremos? Pues aquéllas de la forma:

$$f : \mathbb{N}^n \longrightarrow \mathbb{N}^m$$

Distinguiremos entre **funciones totales** (definidas para cualquier entrada, para cualquier valor del dominio =  $\mathbb{N}^n$ ) y **funciones parciales** (definidas en  $X$ , su dominio es  $\text{dom}(f) \subseteq X$ ). Estas definiciones suponen que las *funciones totales* se pueden tratar como un subconjunto de las *parciales*.

Ejemplo:

$$\text{div}(x, y) = \begin{cases} \text{parte entera } x/y \\ \text{si } y \neq 0 \end{cases}$$
$$\mathbb{N}^2 \longrightarrow \mathbb{N}$$

### 7.1. Tipos de funciones

#### 7.1.1. Función $z$

Establece una correspondencia entre la cadena vacía y 0:

$$z() = 0 \quad \text{o bien} \quad \begin{array}{l} z : \mathbb{N} \longrightarrow \mathbb{N} \\ x \rightsquigarrow z(x) = 0 \end{array}$$

Se puede ampliar el concepto para que asigne a cualquier cadena el 0.

#### 7.1.2. Función siguiente

Establece el *sucesor* de un elemento:

$$\begin{array}{l} x : \mathbb{N} \longrightarrow \mathbb{N} \\ x \rightsquigarrow s(x) = x + 1 \end{array}$$

---

<sup>1</sup>Para los lenguajes funcionales, esto lo analiza el  $\lambda$ -cálculo.

### 7.1.3. Función proyección

Devuelve una posición concreta de un vector de elementos:

$$\begin{aligned} P_i^{(n)} : \mathbb{N} \times \mathbb{N} \dots \times \mathbb{N} &\longrightarrow \mathbb{N} \\ (x_1, x_2, \dots, x_i, \dots, x_n) &\rightsquigarrow x_i \end{aligned}$$

Ejemplos:

$$P_2^3(7, 5, 2) = 5$$

$$P_1^4(1, 2, 4, 2) = 1$$

### 7.1.4. Composición y Recursión primitiva

Con las funciones básicas que acabamos de ver construiremos muchas otras usando la **composición** y la **recursión primitiva**.

#### Composición

Podemos definir:

COMBINACIÓN

$$\begin{aligned} f : \mathbb{N}^k &\longrightarrow \mathbb{N}^m \\ g : \mathbb{N}^k &\longrightarrow \mathbb{N}^n \\ f \times g : \mathbb{N}^k &\longrightarrow \mathbb{N}^{m+n} \\ f \times g(\bar{x}) &\rightsquigarrow (f(\bar{x}), g(\bar{x})) \end{aligned}$$

donde  $\bar{x}$  es un vector de  $k$  componentes.

También hacemos:

COMPOSICIÓN

$$\begin{aligned} f : \mathbb{N}^k &\longrightarrow \mathbb{N}^m \\ g : \mathbb{N}^k &\longrightarrow \mathbb{N}^n \\ f \circ g : \mathbb{N}^k &\longrightarrow \mathbb{N}^m \\ \bar{x} &\rightsquigarrow g[f(\bar{x})] \end{aligned}$$

#### Recursión primitiva

Veámoslo primero con un ejemplo y luego generalizaremos.

Supongamos que queremos una función que nos de el número de nodos en un árbol completo y equilibrado si cada nodo que no es hoja tiene  $x$  hijos y la profundidad del árbol es  $y$ :

$$\begin{aligned} f : \mathbb{N}^2 &\longrightarrow \mathbb{N} \\ f(x, y) &\rightsquigarrow \text{num. nodos} \end{aligned}$$

Podría enunciarse:

$$\begin{aligned} f(x, 0) &= x^0 \\ f(x, y + 1) &= \underbrace{x^y \cdot x}_{\text{este nivel}} + \underbrace{f(x, y)}_{\text{niveles anteriores}} \end{aligned}$$

Usando...

$$\begin{aligned} g : \mathbb{N} &\longrightarrow \mathbb{N} \\ x &\rightsquigarrow g(x) = 1 \\ \\ h : \mathbb{N}^3 &\longrightarrow \mathbb{N} \\ h(x, y, z) &\rightsquigarrow z + x^y \cdot x \end{aligned}$$

podemos definir  $f$  a partir de  $g$  y  $h$  por recursión primitiva de la siguiente manera:

$$\begin{aligned} f(x, 0) &= g(x) \\ f(x, y + 1) &= h(x, y, f(x, y)) \end{aligned}$$

### Definición general

$$\begin{aligned} g : \mathbb{N}^k &\longrightarrow \mathbb{N}^m \\ h : \mathbb{N}^{k+m+1} &\longrightarrow \mathbb{N}^m \\ \\ f(\bar{x}, 0) &= g(\bar{x}) \\ f(\bar{x}, y + 1) &= h(\bar{x}, y, f(\bar{x}, y)) \\ \\ f : \mathbb{N}^{k+1} &\longrightarrow \mathbb{N}^m \end{aligned}$$

#### 7.1.5. Funciones recursivas primitivas

¿Nos llega ya con lo que tenemos? Pues podemos ver que aún no si nos damos cuenta de que hemos estado trabajando todo el rato sólo con funciones totales.

A partir de las funciones básicas (*cero*, *siguiente* y *proyección*) y aplicando un número finito de veces los operadores (*composición* y *recursión primitiva*) obtendremos un nuevo grupo de funciones que denominaremos **funciones recursivas primitivas**.

#### Función suma

$$\begin{aligned} \text{mas}(x, 0) &= x = P_1^1(x) \\ \text{mas}(x, y + 1) &= (x + y) + 1 = h(x, y, \text{mas}(x, y)) = s(P_3^3(x, y, \text{mas}(x, y))) \end{aligned}$$

#### Funciones constantes

$$\begin{aligned} K_m^3 : \mathbb{N} &\longrightarrow \mathbb{N} \\ \bar{x} &\rightsquigarrow 3 = s(s(s(z(P_1^m(x_1, \dots, x_m)))))) \end{aligned}$$

**Función producto**

$$\begin{aligned} \text{mult}(x, 0) &= 0 = z(x) \\ \text{mult}(x, y + 1) &= x + \text{mult}(x, y) = h(x, y, \text{mult}(x, y)) = \text{mas}(P_1^3, P_3^3)[x, y, \text{mult}(x, y)] \\ &= \text{mas}(x, \text{mult}(x, y)) \end{aligned}$$

**Función exponenciación**

$$\begin{aligned} \text{exp}(x, 0) &= 1 \quad (\text{función constante } 1) \\ \text{exp}(x, y + 1) &= h(x, y, \text{exp}(x, y)) = \text{mult}(x, \text{exp}(x, y)) \end{aligned}$$

**Función predecesor**

$$\begin{aligned} \text{pred}(0) &= z() = 0 \\ \text{pred}(y + 1) &= h(y, \text{pred}(y)) = P_1^2(y, \text{pred}(y)) \end{aligned}$$

**Función *monus* (-)**

Es la *sustracción propia*, que devuelve la diferencia  $x - y$  si  $x > y$  y 0 en caso contrario:

$$\begin{aligned} \text{monus}(x, 0) &= P_1^1(x) \\ \text{monus}(x, y + 1) &= h(x, y, \text{monus}(x, y)) = \text{pred}(\text{monus}(x, y)) \end{aligned}$$

**7.1.6. Predicados**

Podemos considerarlos como otro tipo de funciones, que devuelven siempre 0 ó 1. Por ejemplo,

$$\text{eq}(x, y) = \begin{cases} 1 & \text{si } x = y \\ 0 & \text{si } x \neq y \end{cases}$$

¿Cómo lo ponemos en forma recursiva primitiva? Sencillo:

$$\begin{aligned} \text{eq}(x, y) &= 1 - ((y - x) + (x - y)) \\ \text{eq}(x, y) &= \text{monus}(1, \text{mas}(\text{monus}(y, x), \text{monus}(x, y))) \end{aligned}$$

Otro ejemplo:

$$\text{neq}(x, y) = \begin{cases} 1 & \text{si } x \neq y \\ 0 & \text{si } x = y \end{cases}$$

En forma recursiva primitiva:

$$\text{neq}(x, y) = \text{monus}(1, \text{eq}(x, y))$$

Si tenemos una función que coincide con una recursiva primitiva salvo en un número finito de puntos, también es recursiva primitiva:

$$f(x) = \begin{cases} 7 & x = 0 \\ 1 & x = 1 \\ 3 & x = 5 \\ g(x) \text{ recursiva primitiva} & \text{en otro caso} \end{cases}$$

Se puede escribir:

$$\text{eq}(x, 0) \cdot 7 + \text{eq}(x, 1) \cdot 1 + \text{eq}(x, 5) \cdot 3 + \text{neq}(x, 0) \cdot \text{neq}(x, 1) \cdot \text{neq}(x, 5) \cdot g(x)$$

### Función cociente

$$\text{coc}(x, y) = \begin{cases} \text{parte entera de } x \div y & \text{si } y \neq 0 \\ 0 & \text{si } y = 0 \end{cases}$$

Equivalentemente:

$$\begin{aligned} \text{coc}(x, 0) &= 0 \\ \text{coc}(x, y) &= \end{aligned}$$

Sin embargo, hay funciones computables que no podemos construir de esta manera (expresándolas como recursivas primitivas). Un ejemplo es la *función de Ackerman*, que se define:

$$\begin{aligned} A(0, y) &= y + 1 \\ A(x + 1, 0) &= A(x, 1) \\ A(x + 1, y + 1) &= A(x, A(x + 1, y)) \end{aligned}$$

Esta función permite enumerar todas las funciones recursivas primitivas que hemos visto hasta ahora.

$$\begin{aligned} A(1, 1) &= A(0, A(1, 0)) = A(0, A(0, 1)) = A(0, 2) = 3 \\ A(2, 1) &= A(1, A(2, 0)) = A(1, A(1, 1)) = A(1, A(0, A(1, 0))) \\ &= A(1, A(0, A(0, 1))) = A(1, A(0, 2)) = A(1, 3) = A(0, A(1, 2)) \\ &= A(0, A(0, A(1, 1))) = A(0, A(0, A(0, A(1, 0)))) = A(0, A(0, A(0, A(0, 1)))) = \\ &= A(0, A(0, A(0, 2))) = A(0, A(0, 3)) = A(0, 4) = 5 \end{aligned}$$

En general,

$$\begin{aligned} A(1, y) &= y + 2 \\ A(2, y) &= 2y + 3 \\ A(3, y) &= 2^{y+3} - 3 \\ A(4, y) &= 2^{2^y \text{ veces } 2^{16}} - 3 \end{aligned}$$

$$i \in \mathbb{N} \quad f(i) < A(i, i)$$

### 7.1.7. Funciones $\mu$ -recursivas ó recursivas parciales

Hemos trabajado constantemente con funciones totales, ¿acaso no pueden definirse funciones parciales sobre otras funciones parciales? Por supuesto que es posible, y se hace:

$$\begin{aligned} f : \mathbb{N}^n &\longrightarrow \mathbb{N} \\ g : \mathbb{N}^{n+1} &\longrightarrow \mathbb{N} \end{aligned}$$

¿Cómo definir  $f(\bar{x})$  a partir de  $g$ ? Pues lo haremos asignándole  $g(\bar{x}, y) = 0$  con la menor  $y$  posible y con la condición de que  $g(\bar{x}, z)$  esté definida para todo valor de  $z$  anterior a  $y$ . Se representa:

$$f(\bar{x}) = \mu y [g(\bar{x}, y) = 0]$$

Un ejemplo:

$$\begin{array}{lll} g(0, 0) = 2 & g(1, 0) = 3 & g(2, 0) = 8 \\ g(0, 1) = 3 & g(1, 1) = 4 & g(2, 1) = 3 \\ g(0, 2) = 1 & g(1, 2) = 0 & g(2, 2) = \text{no definida} \\ g(0, 3) = 5 & & g(2, 3) = 6 \\ g(0, 4) = 0 & & g(2, 4) = 0 \end{array}$$

En este caso:

$$\begin{aligned} f(0) &= 4 \\ f(1) &= 2 \\ f(2) &\text{ no está definida porque } g \text{ no lo está para todos los anteriores} \end{aligned}$$

De modo que así podemos definir unas funciones parciales a partir de otras, por ejemplo:

$$\text{div}(x, y) = \mu t [(x + 1) - (\text{mult}(t, y) + y) = 0]$$

Con lo que hemos visto, sí tenemos ahora cubiertas todas las *funciones computables* posibles. De hecho, la **tesis de Church-Turing** afirma que para definir un lenguaje sólo se necesita hacer:

```
enteros no negativos, no definición tipos;
```

```
    incr nombre;  
    decr nombre;
```

```
    while nombre <> 0 do
```

```
        .
```

```
        .
```

```
        .
```

```
    end;
```



# Capítulo 8

## Estudio de la Complejidad Computacional

En uno de los temas anteriores (capítulo 6, página 45) vimos que hay problemas que carecen manifiestamente de solución alcanzable. Otra situación que se da en ocasiones es la de tener un problema cuya solución sabemos cómo calcular pero que no puede ser alcanzada en un tiempo asequible. En torno a esta idea se definen los conceptos de **problemas tratables** y **problemas intratables**.

Se definen:

$$\text{Time } D_M(x) = \begin{cases} \text{número de pasos dados si } M \text{ (M.T.) termina con entrada } x \\ \text{no definida en otro caso} \end{cases}$$

$$\text{Space } D_M(x) = \begin{cases} \text{número de celdas usadas si } M \text{ (M.T.) termina con entrada } x \\ \text{no definida en otro caso} \end{cases}$$

Dada una función  $t(n)$ , se dice que  $\text{DTM}^1$  *está en un tiempo*  $t(n)$  si el tiempo que tarda en acabar está acotado por dicha función (análogamente para el espacio).

Si una  $\text{NDTM}^2$  está en  $t(n)$ , entonces su  $\text{DTM}$  asociada está en  $c \cdot [t(n)]$ .

La complejidad de la verificación de un problema suele equivaler a la complejidad no determinista en términos de MT.

### 8.1. Clasificación de problemas

Se definen diferentes clases de problemas:

**Problemas P** Reciben este nombre los problemas tratables (que podemos resolver) con una MTD en un *tiempo polinómico*, es decir, cuando  $t(n)$  es un polinomio (complejidad  $n^i$ ).

**Problemas NP** Se llama así a los problemas no tratables, aquéllos que requerirían un tiempo polinómico en una MTND.

---

<sup>1</sup>Máquina de Turing Determinista.

<sup>2</sup>Máquina de Turing No Determinista.

**Problemas PSPACE y NPSPACE** Son problemas P y NP pero con respecto al espacio que requieren. Resultan ser equivalentes (no así los P y los NP, algo que todavía no está probado).

**Problemas Exponenciales** Son los que consumen un tiempo exponencial en una MTD.

Normalmente, siempre se necesita un poco más de espacio que de tiempo, pero siempre éste es el parámetro más importante.

Dentro de los **problemas NP** se tiene un tipo especial, los problemas **NP-completos**, **NP-duros** o **NP-difíciles**, que, como su propio nombre indica, son de los más complicados (un ejemplo es el problema del grafo hamiltoniano en el problema del viajante). Cualquier problema NP se puede reducir a uno de estos problemas NP-completos en tiempo polinómico. Asimismo, dentro de los problemas NP-completos se puede “pasar” de uno a otro.

# Índice general

<b>0. Introducción</b>	<b>5</b>
<b>1. Alfabetos y lenguajes</b>	<b>7</b>
1.1. Alfabetos, palabras y lenguajes . . . . .	7
<b>2. Lenguajes Regulares</b>	<b>11</b>
2.1. Lenguajes sobre alfabetos . . . . .	11
2.2. Lenguajes y expresiones regulares . . . . .	12
2.3. Autómatas finitos deterministas . . . . .	12
2.3.1. Representaciones . . . . .	12
2.4. Autómatas finitos deterministas y lenguajes . . . . .	13
2.5. Autómatas finitos no deterministas . . . . .	13
2.6. Equivalencia entre AFD y AFN . . . . .	14
2.7. $\varepsilon$ -transiciones . . . . .	14
2.7.1. Eliminación de $\varepsilon$ -transiciones . . . . .	15
2.8. Autómatas finitos y expresiones regulares . . . . .	15
2.9. Propiedades de los lenguajes regulares . . . . .	18
<b>3. Lenguajes Independientes del contexto</b>	<b>21</b>
3.1. Gramáticas regulares . . . . .	21
3.2. Gramáticas y lenguajes regulares . . . . .	22
3.3. Gramáticas independientes del contexto . . . . .	24
3.4. Árboles de derivación . . . . .	24
3.5. Simplificación de gramáticas independientes del contexto . . . . .	25
3.6. Forma normal de Chomsky . . . . .	27
3.7. Propiedades de los lenguajes independientes del contexto . . . . .	29
3.8. Autómatas de pila . . . . .	29
3.9. Autómatas de pila y lenguajes independientes del contexto . . . . .	30
3.10. Autómatas de pila deterministas . . . . .	31
3.11. Forma normal de Greibach . . . . .	31
<b>4. Máquinas de Turing</b>	<b>33</b>
4.1. Definición de Máquina de Turing . . . . .	33
4.2. Máquinas de Turing como aceptadoras de lenguajes . . . . .	34
4.3. Construcción de Máquinas de Turing . . . . .	35
4.3.1. Algunas Máquinas de Turing especiales . . . . .	35
4.4. Modificaciones de las Máquinas de Turing . . . . .	35
4.5. Máquina de Turing Universal . . . . .	36

<b>5. Máquinas de Turing y Lenguajes</b>	<b>39</b>
5.1. Lenguajes aceptados por Máquinas de Turing . . . . .	39
5.2. Lenguajes Recursivos y Recursivamente Enumerables . . . . .	39
5.3. Gramáticas no restringidas y Lenguajes RE . . . . .	41
5.4. Lenguajes sensibles al contexto y jerarquía de Chomsky . . . . .	42
<b>6. Resolubilidad</b>	<b>45</b>
6.1. El problema de la parada . . . . .	45
6.1.1. El problema de la cinta en blanco . . . . .	46
6.2. El problema de correspondencia de Post . . . . .	46
6.3. Irresolubilidad en lenguajes independientes del contexto . . . . .	48
<b>7. Funciones <math>\mu</math>-recursivas</b>	<b>51</b>
7.1. Tipos de funciones . . . . .	51
7.1.1. Función $z$ . . . . .	51
7.1.2. Función siguiente . . . . .	51
7.1.3. Función proyección . . . . .	52
7.1.4. Composición y Recursión primitiva . . . . .	52
7.1.5. Funciones recursivas primitivas . . . . .	53
7.1.6. Predicados . . . . .	54
7.1.7. Funciones $\mu$ -recursivas ó recursivas parciales . . . . .	56
<b>8. Estudio de la Complejidad Computacional</b>	<b>59</b>
8.1. Clasificación de problemas . . . . .	59