Índice

Agradecimientos

Varias son las personas que han colaborado, en mayor o menor medida, en la elaboración de este libro. Entre ellas queremos mencionar (por orden alfabético): José del Campo Ávila, Enrique Domínguez Merino, Inmaculada Fortes Ruiz, Ezequiel López Rubio, Blas Carlos Ruiz Jiménez y Francisco José Vico Vela. A todos ellos nuestro más sincero agradecimiento.

Prólogo para el profesor

Este libro pretende ser una herramienta útil para la docencia a nivel universitario de la asignatura *Teoría de Autómatas y Lenguajes Formales*, la cual se engloba dentro de las materias de informática teórica que se imparten en ingenierías (fundamentalmente de Informática). El contenido es el resultado de refundir en un solo texto dos libros, *Teoría de Autómtas y Lenguajes Formales* (Gonzalo Ramos Jiménez, 2005) y *Modelos de Cómputo* (Rafael Morales Bueno y Gonzalo Ramos Jiménez, 2007), acortando el contenido para poder impartirse en sesenta horas (cuatro horas semanales durante quince semanas), que corresponden a una asignatura cuatrimestral de seis créditos en los nuevos planes de grado.

El propósito es dar un texto sobre el que impartir las clases de modo que éstas puedan ser más cómodas, dinámicas, participativas, e incluso más profundas (si se desea). No es un libro pensado para aprender la materia sin una docencia, si bien alguien con una buena formación matemática puede seguirlo sin problemas.

Se supone que el alumno tiene cierta base en matemática discreta y álgebra, y por tanto conoce, por ejemplo, los conceptos de conjunto, pertenencia, inclusión, el principio de buen orden de los naturales, etc..

Este es un libro fundamentalmente teórico. En él se dan ejemplos de prácticamente todos los conceptos definidos, si bien no es un libro de ejercicios.

Entrando más en aspectos técnicos, indicar que respecto a los tipos de gramáticas se han definido aparte las gramáticas regulares y de contexto libre que tienen reglas con la cadena vacía como consecuente (a las que hemos denominado reglas épsilon). El motivo es que queríamos incluirlas en el texto sin que ello cambiara las inclusiones "limpias" de la jerarquía de Chomsky de las gramáticas. Se ha mantenido la denominación *"de contexto libre"* por ser más habitual, a pesar de que quizás sería más correcto *"independiente del contexto"*.

También comentar que hemos introducido una tipología de las reglas de una gramática, que posteriormente hemos utilizado para definir los tipos de las gramáticas propiamente dichas. El motivo de hacerlo así ha sido puramente didáctico, ya que nuestra experiencia docente (de varios años) nos ha mostrado que el alumno asimila mejor de esta forma la tipología de las gramáticas, aparte de que con ello podemos referirnos con mayor propiedad al tipo de una regla concreta.

Por otra parte, en el aspecto notacional, señalar que se ha utilizado la doble barra para cardinal con el objeto de diferenciarla de la barra simple de longitud de una cadena. Creemos que de esta forma el texto queda más claro, ya que aunque habitualmente se utiliza la doble barra para norma, aquí no hay lugar a error puesto que el concepto de norma no aparece en el presente texto.

Al final del texto se han incluido tres anexos. El primero con las definiciones de las máquinas de Mealy y de Moore, con el objeto de que el lector conozca al menos el significado del término traductor finito dentro de la teoría de autómatas. El segundo y el tercero son las codificaciones de Cántor y Gödel, respectivamente.

Para terminar decir que esperamos que el presente libro cumpla su propósito y sea una herramienta útil para la docencia de esta materia, pudiendo gracias a él concentrarse el profesor, más que en la elaboración del material docente, en otras tareas docentes, como son la de ampliar el número de ejemplos y proponer ejercicios, así como explicar y relacionar como parte de un todo la materia dada.

Para cualquier comentario sobre el libro, que estaremos encantados de recibir, nos pueden mandar un correo electrónico a *ramos@lcc.uma.es* (por favor con *subject*: libro).

Prólogo para el alumno

El libro que tienes en tus manos pretende ser una herramienta útil para que aprendas con más facilidad y mayor profundidad la asignatura *Teoría de Autómatas y Lenguajes Formales*, la cual se engloba dentro de las materias de informática teórica que se imparten en ingenierías (fundamentalmente de Informática).

Se supone que tienes cierta base en matemática discreta y álgebra, y por tanto conoces, por ejemplo, los conceptos de conjunto, pertenencia, inclusión, el principio de buen orden de los naturales, etc., así como todos los conceptos que aparecen en el primer apartado (titulado *Recordatorio*) del primer tema. Si al iniciarse las clases de esta asignatura tienes alguna duda sobre los conceptos mencionados, deberías ponerte al día rápidamente si quieres comprender por completo las citadas clases.

Este libro en modo alguno pretende sustituir la asistencia a clase, sino que por el contrario su propósito es incentivar dicha asistencia haciendo que las clases puedan ser más cómodas, comprensibles, dinámicas, y participativas.

El profesor irá marcando la pauta sobre qué contenidos se verán y con que profundidad, la importancia de cada uno de ellos, así como sus interrelaciones. Es una muy buena práctica (aunque muy poco extendida) aprovechar que se dispone de un texto docente para ir leyendo con anterioridad los contenidos que el día siguiente se van a impartir, lo cual hace que uno siga mucho mejor la futura clase.

La asignatura que aquí se expone, que es fundamentalmente teórica, por la forma en que se construye, paso a paso sobre los conceptos que se van introduciendo, es de las asignaturas en que más rendimiento se obtiene del hecho de llevarla al día. Permítenos por tanto, para terminar, darte un consejo; si durante las clases de esta asignatura hay algún concepto que no entiendes, cuanto antes lo aclares mucho mejor para la comprensión de las clases restantes.

Normas de estilo y notación

Dentro de cada tema las definiciones, proposiciones, ejemplos, etc. han sido numerados consecutivamente de forma conjunta (y no independientemente), lo que facilita su localización en el texto. Cada bloque de definiciones se ha señalado con una línea horizontal al inicio y al final del mismo. Cada ejemplo viene señalado por una línea vertical a su izquierda. Las proposiciones, teoremas y lemas aparecen enmarcados y sombreados. También se enmarcan (sin sombrear) las demostraciones. Se ha utilizado el símbolo "□" para señalar fin de demostración (o de una parte de una demostración), o fin de ejemplo.

La notación utilizada es la habitual en matemáticas, si bien conviene especificar el uso en el texto de algunos símbolos:

$A \subseteq B$	El conjunto A está incluido en el B, o es igual al B.
$A \subset B$	El conjunto A está incluido en el B y no es igual al B.
$\| A \|$	Cardinal del conjunto A.
$\| x \|$	Longitud de la cadena x.
$\aleph_i$	Transfinito *alef* i.
➔←	Contradicción.
ε	Cadena vacía.
$\alpha \to \beta$	Es una regla de producción de una gramática, donde α es el antecedente y β es el consecuente.
$x \Rightarrow y$	La cadena x produce directamente la cadena y. Si el símbolo "⇒" aparece aislado en un razonamiento, entonces significa "implica".

$C_1 \vdash C_2$	La configuración C_1 transita directamente a la C_2.
$x \prec y$	La cadena x es subcadena de la cadena y.

CAPÍTULO I

Lenguajes Formales

Tema 1: Preliminares Matemáticos

Tema 2: Lenguajes y Gramáticas

Tema 3: Expresiones Regulares

Tema 4: Autómatas Finitos

Tema 5: Condiciones de Regularidad

Tema 6: Lenguajes de Contexto Libre

1
Preliminares Matemáticos

1.1 Recordatorio

Definición 1.1: *Conjunto potencia de un conjunto A*
Dado un conjunto A, definimos el conjunto potencia de A, notado 2^A, como:

$$2^A = \{ B \mid B \subseteq A \}$$

Nota: Para todo conjunto A, $A \in 2^A$ y $\varnothing \in 2^A$.

Definición 1.2: *Subconjunto propio*
Dado un conjunto A, B es subconjunto propio de A sii $B \subseteq A \ \wedge \ B \neq A \ \wedge \ B \neq \varnothing$.

Definición 1.3: *Partición*
Dado un conjunto A, $\Pi \subseteq 2^A$ es una partición de A sii cumple las siguientes condiciones:

1) $\bigcup_{A_i \in \Pi} A_i = A$
2) $A_i \cap A_j = \varnothing \quad \forall i \neq j$
3) $A_i \neq \varnothing \quad \forall A_i \in \Pi$

Definición 1.4: *Relación*
Dados dos conjuntos A y B, una relación R entre A (conjunto *inicial*) y B (conjunto *final*) es un subconjunto del producto cartesiano de A por B ($R \subseteq A \times B$).

Definición 1.5: *Relación sobre un conjunto (relación binaria)*
Dado un conjunto A, una relación R es sobre A si $R \subseteq A \times A$.

Definición 1.6: *Propiedades de las relaciones binarias*
Sea A un conjunto y sea R una relación sobre A.
Diremos que R es *Reflexiva* sii $(a, a) \in R \ \forall a \in A$.
Diremos que R es *Simétrica* sii $(a, b) \in R \Rightarrow (b, a) \in R$.
Diremos que R es *Antisimétrica* sii $(a, b) \in R \wedge a \neq b \Rightarrow (b, a) \notin R$.
Diremos que R es *Transitiva*: sii $(a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$.

Definición 1.7: *Relación de equivalencia*
Una relación binaria es de equivalencia si es reflexiva, simétrica y transitiva.

Definición 1.8: *Clase de equivalencia*
Sea A un conjunto y sea R una relación de equivalencia sobre A. Sea $a \in A$. Definimos la clase de equivalencia de A a la que pertenece a, notada $[a]$, como:

$$[a] = \{ b \in A \mid (a, b) \in R \}$$

Nota: Una relación de equivalencia sobre un conjunto establece una partición del mismo (conjunto cociente) donde cada elemento de la partición es una clase de equivalencia.

Definición 1.9: *Relación identidad (I)*
Dado un conjunto A definimos la relación identidad como:

$$I = \{ (a, a) \mid a \in A \}$$

Definición 1.10: *Relación inversa (R^{-1})*
Dado un conjunto A y una relación R sobre A definimos la relación inversa de R como:

$$R^{-1} = \{ (a, b) \mid (b, a) \in R \}$$

Definición 1.11: *Potencia de una relación (R^n)*
Dado un conjunto A y una relación R sobre A definimos la potencia de R como:
Para $n = 1$: $R^1 = R$.
Para $n \geq 2$: $(a, b) \in R^n$ sii $\exists x \in A \mid (a, x) \in R^{n-1} \wedge (x, b) \in R$.

Definición 1.12: *Cierre de relaciones*
Dado un conjunto A y una relación R sobre A definimos:

Cierre *reflexivo* de R es $R \cup I$.

Cierre *simétrico* de R es $R \cup R^{-1}$.

Cierre *transitivo* de R es R^{∞}, siendo $R^{\infty} = \bigcup_{n=1}^{\infty} R^n$.

Nota: Si definimos el concepto más general de cierre de una relación para una propiedad, entonces los anteriores se derivan de éste.

Definición 1.13: *Aplicación*
Una aplicación f de un conjunto A a uno B, notado $f: A \to B$, es una relación entre A y B que cumple que $\forall a \in A \ \exists! (a,b) \in f$.

Definición 1.14: *Función*
Una relación $f: A \to B$ es una función sii f es aplicación.

Notas: Es habitual escribir f (a)=b en lugar de (a, b) ∈ f y así lo haremos a partir de ahora.
Aunque función y aplicación aquí son sinónimos, función se suele utilizar cuando la relación se ve como un proceso de transformación de una entrada en una salida.
Este concepto de función (que es total) se puede extender para que englobe también las funciones parciales (aunque en el primer capítulo no las usaremos) de la siguiente forma:
Una relación $f: A \to B$ es una función sii $\exists C \subseteq A \mid f: C \to B$ es aplicación.

Definición 1.15: *Dominio y Rango de una función (... de una aplicación)*
Sea la función $f: A \to B$.
Definimos su dominio ($Dom(f)$) y su rango ($Rg(f)$) como:
$Dom(f) = \{ a \in A \mid f(a) = b \text{ con } b \in B \}$.
$Rg(f) = \{ f(a) \in B \mid a \in A \}$.

Nota: Con nuestra definición de función (que es total) siempre el dominio y el conjunto inicial de una función coinciden.

Definición 1.16: *Función inyectiva (aplicación inyectiva)*
Una función $f: A \to B$ es inyectiva sii $x \neq z \Rightarrow f(x) \neq f(z)$.

Nota: f es inyectiva sii $f(x) = f(z) \Rightarrow x = z$.

Definición 1.17: *Función sobreyectiva (aplicación sobreyectiva)*
Una función $f: A \to B$ es sobreyectiva sii $Rg(f) = B$.

Definición 1.18: *Función biyectiva (aplicación biyectiva, biyección)*
Sea $f: A \to B$ una función, decimos que es biyectiva sii es inyectiva y sobreyectiva.

Según la definición de función biyectiva, para demostrar que una función dada lo es debemos demostrar que es inyectiva y sobreyectiva.

El procedimiento para ver si una función es inyectiva consiste en escribir $f(x) = f(z)$, resolver como ecuación, y si resulta que la única solución es $x = z$, entonces es inyectiva.

Por otra parte, para ver si una función es sobreyectiva escribimos $f(x) = b$, vista como ecuación se despeja x , y si esto siempre es posible entonces es sobreyectiva.

Ejemplo 1.19:
Sea $\mathbf{I}$ el conjunto de los números impares.
Sea $f: \mathsf{N} \to \mathbf{I}$ una función definida como $f(n) = 2n+1$.
Demostraremos que f es biyectiva.

a) Inyectiva:
$f(x) = f(z) \Rightarrow 2x+1 = 2z+1 \Rightarrow 2x = 2z \Rightarrow x = z$ que es la única solución. □ a)

b) Sobreyectiva:
$f(x) = b \Rightarrow 2x+1 = b \Rightarrow 2x = b-1 \Rightarrow x = (b-1)/2$ al ser b impar, $(b-1)$ es par, y por tanto siempre $(b-1)/2 \in \mathsf{N}$. □ b)

Definición 1.20: *Operación interna sobre un conjunto*
Sea A un conjunto.
Toda aplicación $f: A \times A \to A$ es una operación interna sobre A.
También se puede notar $A \times A \xrightarrow{f} A$.

Nota: El concepto de operación interna (binaria) definido se puede extender a operaciones *n*-arias (monaria, ternaria, etc.).

Definición 1.21: *Propiedad asociativa*
Sea A un conjunto, y sea $\bullet$ una operación interna sobre dicho conjunto ($\bullet : A \times A \to A$).
Diremos que $\bullet$ es asociativa sii $(x \bullet y) \bullet z = x \bullet (y \bullet z) \quad \forall x,y,z \in A$.

Definición 1.22: *Semigrupo*
Semigrupo es un par $(A, \bullet)$, donde A es un conjunto y $\bullet$ es una operación interna sobre dicho conjunto que es asociativa.

Definición 1.23: *Elemento neutro (e)*
Dada una operación $\bullet$ sobre un conjunto A, decimos que $e \in A$ es el elemento neutro para dicha operación sii $\forall a \in A \quad a \bullet e = e \bullet a = a$.

Definición 1.24: *Monoide*
Monoide es un semigrupo que posee elemento neutro para la operación.

Definición 1.25: *Conjunto cerrado para una operación*
Sea A un conjunto y $\bullet$ una operación sobre A, y sea $B \subseteq A$.
Diremos que B es cerrado para la operación $\bullet$ sii $x \bullet y \in B \quad \forall x,y \in B$.

Definición 1.26: *Cierre estricto de un conjunto para una operación*
Sea A un conjunto y $\bullet$ una operación sobre A, y sea $B \subseteq A$.
El cierre estricto de B para la operación $\bullet$, notado $B^{\bullet}$, se define como:

1) $x \in B \Rightarrow x \in B^{\bullet}$
2) $x, y \in B^{\bullet} \Rightarrow x \bullet y \in B^{\bullet}$
3) Ningún otro elemento pertenece a $B^{\bullet}$.

Definición 1.27: *Cierre amplio de un conjunto para una operación*
Sea el monoide $(A, \bullet)$, y sea $B \subseteq A$.
El cierre amplio de B para la operación $\bullet$, notado $B^{\bullet e}$, se define como:
$B^{\bullet e} = B^{\bullet} \cup \{e\}$ donde e es el elemento neutro del monoide.

Nota: Obsérvese que mientras que ser cerrado para una operación es una propiedad, que puede o no cumplir un conjunto, el cierre estricto y el cierre amplio son operaciones que realizamos sobre un conjunto, no propiedades del mismo.

1.2 Conjuntos Finitos e Infinitos

Definición 1.28: *Conjuntos equipotenciales (equipotentes)*
Sean A y B dos conjuntos, decimos que son equipotenciales si $\exists f: A \to B \mid f$ es biyectiva.

Definición 1.29: *Conjunto finito*
Un conjunto A es finito si es vacío o $\exists n \in \mathbb{N} \mid \{1, 2, ..., n\}$ y A son equipotenciales.

Definición 1.30: *Cardinal de un conjunto finito no vacío*
El cardinal de un conjunto finito no vacío A es n, y lo notaremos $\| A \| = n$, sii A y $\{1, 2, ..., n\}$ son equipotenciales.

Definición 1.31: *Cardinal del conjunto vacío*
$\| \varnothing \| = 0$.

Definición 1.32: *Conjunto infinito*
Un conjunto A se dice infinito sii no es finito.

Definición 1.33: *Conjunto infinito numerable*
Un conjunto A es infinito numerable sii A y $\mathbb{N}$ son equipotenciales.

Definición 1.34: *Conjunto numerable, Conjunto no numerable*
Sea A un conjunto, decimos que es numerable si es finito o infinito numerable. En caso contrario diremos que es no numerable.

Proposición 1.35:
Todo subconjunto de un conjunto numerable es numerable.

Demostración:
Sabemos que un conjunto numerable puede ser finito o infinito numerable.
Es trivial que cualquier subconjunto de un conjunto finito es también finito y por tanto numerable. Por otro lado, un subconjunto de un conjunto infinito numerable puede ser finito o infinito.
En el primer caso es numerable.
En el segundo caso tenemos un conjunto A que es infinito numerable (por medio de la biyección $g: \mathbb{N} \to A$), y un conjunto infinito $B \subseteq A$.
Dado $D \subseteq A$ definimos el mínimo del conjunto D , notado *mínimo(D)* , como:

$$mínimo(\ D\ ) = g(\ min\{\ i \in \mathbb{N} \mid\ g(i) \in D\ \}\ )$$

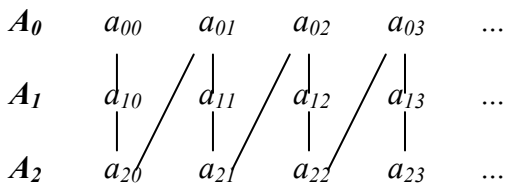
Definimos la función $f: \mathbb{N} \to B$ como:

$$f(n) = mínimo(\ B - \{\ f(m) \mid\ m < n\ \}\ )$$

que por construcción es biyectiva, y por tanto B es numerable. □

Proposición 1.36:
La unión finita de conjuntos numerables es numerable.

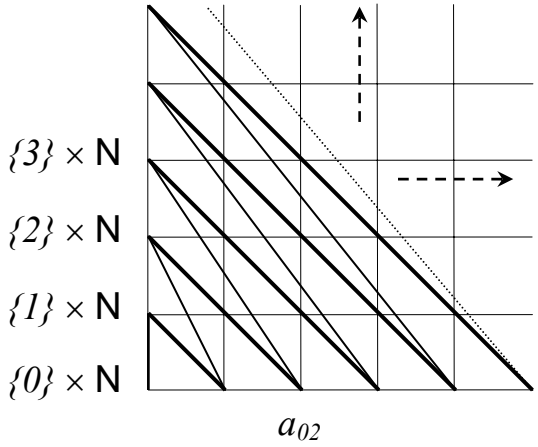
Demostración:
Supondremos que los conjuntos numerables son infinitos y disjuntos, ya que en otro caso la demostración es trivial (finitos) o reducible (no disjuntos). En la demostración usaremos una técnica conocida como *machihembrado*, que consiste en encontrar una función biyectiva entre el conjunto unión y los naturales ($f: A_0 \cup A_1 \cup ... \cup A_k \to \mathbb{N}$) según muestra la figura.



La expresión de esta función, para n conjuntos, es: $f_n(a_{ij}) = nj+i$ □

Proposición 1.37:
La unión infinita numerable de conjuntos numerables es numerable.

Demostración:
Para su demostración emplearemos una variante de la técnica de machihembrado. Realizamos la unión infinita numerable de conjuntos numerables y obtenemos una biyección con los naturales según la figura.



La expresión de esta función es: $f(a_{ij}) = \frac{(i+j)(i+j+1)}{2} + j$ □

Corolario 1.38:
El producto cartesiano de dos conjuntos numerables es numerable.

Proposición 1.39:
La potencia finita de un conjunto numerable es numerable.

Demostración:
Lo demostraremos por inducción:

a) $\mathbb{N}^1 = \mathbb{N}$, que es numerable por definición.
b) Supongamos que $\mathbb{N}^n$ es numerable.
c) $\mathbb{N}^{n+1} = \mathbb{N}^n \times \mathbb{N} \Rightarrow \mathbb{N}^{n+1}$ es numerable.
$\uparrow$
por b) y el corolario **1.38**

Notas: La inducción como técnica válida de demostración la veremos en el próximo apartado.
Para $\mathbb{N}^0$ también se cumple, ya que es finito y por tanto numerable.

□

Ejemplo 1.40:
Los números impares son numerables, ya que son un subconjunto de un conjunto numerable, y por tanto sabemos que existe una biyección entre este conjunto y el de los naturales. Una función para establecer esa biyección es $f(n) = 2n + 1$.

Los números impares mayores que cien son numerables, ya que también son un subconjunto de un conjunto numerable, y por tanto sabemos que existe una biyección entre este conjunto y el de los naturales. Una biyección con los impares es $f(i) = i + 100$.

Por el mismo motivo los números múltiplos de siete mayores que diecisiete también son numerables. Una biyección con los naturales es $f(n) = 7n + 21$.

□

Definición 1.41: *Cardinal del conjunto de los naturales*
$\| \mathbb{N} \| = \aleph_0$ (*alef cero*).

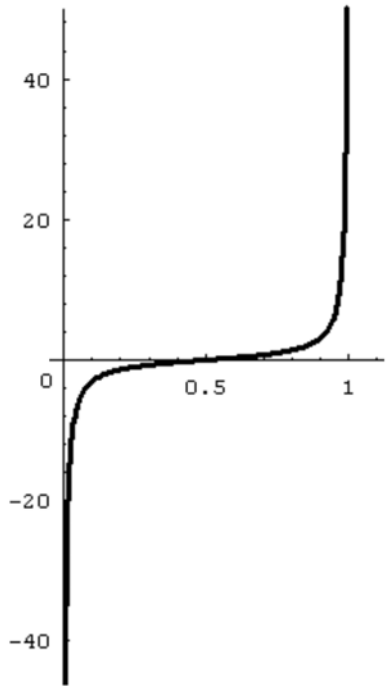
Teorema de Cantor:
$\| \mathbb{N} \| \neq \| \mathbb{R} \|$

Demostración:

Para demostrar que $\| \mathbb{N} \| \neq \| \mathbb{R} \|$, en vez de trabajar con todos los reales trabajaremos sólo con el intervalo $(0, 1)$, que es equipotencial con $\mathbb{R}$. Esto lo podemos comprobar viendo que la función:

$$g(x) = tg\left[\left(x - \frac{1}{2}\right) \cdot \pi\right]$$

establece una biyección entre $(0, 1)$ y $\mathbb{R}$.



La demostración de que $(0, 1)$ no es numerable la realizaremos por reducción al absurdo.

Así, supongamos que el intervalo $(0, 1)$ es equipotencial con $\mathbb{N}$. En ese caso habrá una función $f: \mathbb{N} \rightarrow (0, 1)$ que es biyectiva.

Si denominamos r_i a $f(i)$ tenemos:

$$0 \rightarrow f(0)=r_0$$
$$1 \rightarrow f(1)=r_1$$
$$2 \rightarrow f(2)=r_2$$
$$\vdots$$

Si esto es así, podemos afirmar que todos los reales del intervalo estarán en la siguiente lista:

$$r_0 = 0' \mathbf{d_{00}}\ d_{01}\ d_{02}\ d_{03}\ \ldots$$
$$r_1 = 0' d_{10}\ \mathbf{d_{11}}\ d_{12}\ d_{13}\ \ldots$$
$$r_2 = 0' d_{20}\ d_{21}\ \mathbf{d_{22}}\ d_{23}\ \ldots$$
$$r_3 = 0' d_{30}\ d_{31}\ d_{32}\ \mathbf{d_{33}}\ \ldots$$
$$\vdots$$

donde cada d_{ij} (con $i, j \in \mathbb{N}$) es el j-ésimo dígito de la expresión decimal completa del real r_i (el i-ésimo real de la lista). Para los reales con dos posibles expresiones decimales (infinitos ceros o nueves hacia la derecha) se escogerá la de infinitos nueves hacia la derecha.

Sea $r = 0' d_0\, d_1\, d_2\, d_3 \ldots \in (0, 1)$ donde $\forall i \geq 0$ $\begin{cases} d_i = 4 & \text{si } d_{ii} \neq 4 \\ d_i = 5 & \text{si } d_{ii} = 4 \end{cases}$

(d_{ii} es el i-ésimo dígito de la diagonal de la lista anterior).

Por lo tanto: $d_i \neq d_{ii} \quad \forall i \geq 0 \quad \Rightarrow \quad r \neq r_i \quad \forall i \geq 0 \quad \Rightarrow$ r no está en la anterior lista.
Pero esto contradice la suposición inicial, ya que en esta lista estaban todos los elementos del intervalo $(0, 1)$, de donde concluimos que f no es biyectiva y por tanto $\mathbb{R}$ no es equipotencial con $\mathbb{N}$. □

Definición 1.42: *Cardinal del conjunto de los reales*
$\| \mathbb{R} \| = \aleph_1$ (*alef uno*).

Definición 1.43: *Transfinitos*
Diremos que $\aleph_i$ (*alef i*) es el *i*-ésimo transfinito.
El conjunto $\{ \aleph_i \mid i \in \mathbb{N} \}$ es el conjunto de todos los transfinitos.

Definición 1.44: *Cardinal de un conjunto infinito*
El cardinal de un conjunto infinito A es $\aleph_i$ sii es equipotencial con un conjunto de cardinal conocido $\aleph_i$, y lo notaremos $\| A \| = \aleph_i$, con $i \in \mathbb{N}$.

Ejemplo 1.45:
Los tres conjuntos del ejemplo **1.40** tienen cardinal $\aleph_0$, ya que existe una biyección entre cada uno de ellos y los naturales.

El intervalo $(0, 1)$ tiene cardinal $\aleph_1$, ya que existe una biyección entre este conjunto y el de los reales. Una función para establecer esa biyección es $f(x) = tg[(x-0'5)\cdot\pi]$.

El intervalo $(0, 2)$ tiene cardinal $\aleph_1$, ya que existe una biyección entre este conjunto y el intervalo $(0, 1)$, del cual sabemos que su cardinal es $\aleph_1$. Una función para establecer esa biyección es $f(x) = 2x$. □

Trabajar con conjuntos infinitos es algo bastante contraintuitivo. Quizás lo mejor es imaginarse el cardinal de un conjunto infinito como una medida de su "densidad".

Ya hemos visto que al cardinal de $\mathbb{N}$ se le denomina *Alef Cero* ($\aleph_0$), que es el primer infinito (el infinito "menos denso"). También $\| \mathbb{Q} \| = \aleph_0$ ya que a/b se puede considerar como otra notación para $(a, b) \in \mathbb{N} \times \mathbb{N}$. Al cardinal de $\mathbb{R}$ se le denomina también *potencia del continuum* (c), pero denominarlo así nos llevaría a la *hipótesis del continuum*, la cual queda fuera de los objetivos docentes de este libro.

Señalar por último dos resultados que necesitaremos, $\aleph_1 - \aleph_0 = \aleph_1$ y $2^{\aleph_0} = \aleph_1$.

1.3 Técnicas Fundamentales de Demostración

Vamos a estudiar tres técnicas fundamentales de demostración: el *principio de inducción*, el de los *casilleros*, y el de *diagonalización*. Para demostrar su validez utilizaremos la técnica de reducción al absurdo.

Principio de inducción

Si A es un conjunto de números naturales que cumple:

a) $0 \in A$;

b) $\{ 0, 1, ..., n \} \subseteq A \Rightarrow n + 1 \in A \quad \forall n \in \mathsf{N}$.

entonces $A = \mathsf{N}$.

Demostración :

Sea $A \subseteq \mathsf{N}$ que cumple *a)* y *b)*.

Supongamos que $A \neq \mathsf{N}$.

$$A \neq \mathsf{N} \Rightarrow \mathsf{N} - A \neq \varnothing \Rightarrow \exists m \notin A \mid m = min(\mathsf{N} - A) \Rightarrow m \neq 0$$

(↑ *por a)*)

$$\Rightarrow \{0, 1, ..., m-1\} \subseteq A \Rightarrow m \in A \Rightarrow A = \mathsf{N}$$

(↑ *por b)*; ↑ ➔➔)

□

La inducción se utiliza para probar afirmaciones de la forma:

"Para todos los números naturales, la propiedad P es verdadera".

El principio se aplica al conjunto $A = \{ n \mid P \text{ es verdadera para } n \} = \{ n \mid P(n) \}$ de la siguiente forma:

1º) *Caso Base (C.B.)*:

Demostramos $P(0)$ (a veces el C.B. es $P(n)$ con $n>0$)

2º) *Hipótesis de Inducción (H.I.)*:

Suponemos que $\exists n \geq 0 \mid P(i) \quad \forall i = 0, 1, ... , n$

3º) *Paso de Inducción (P.I.)*:

Demostramos, usando la H.I., $P(n+1)$

por el principio de inducción $A = \mathsf{N}$, es decir, $\forall n \; P(n)$.

Ejemplo 1.46:

Demostrar que $1+2+...+n=\frac{(n^2+n)}{2}$, $\forall n \geq 0$

C.B.: Es trivial para $n = 0$

H.I.: Suponemos que $\exists n \geq 0$ tal que $1+2+...+m=\frac{(m^2+m)}{2}$ con $m \leq n$

P.I.: Demostramos para $n+1$

$$1+2+...+n+(n+1)=(1+2+...+n)+(n+1) \Rightarrow$$

$$\uparrow$$

por H.I.

$$(1+2+...+n)+(n+1)=\frac{n^2+n}{2}+(n+1)=\frac{n^2+n+2n+2}{2}=\frac{n^2+2n+1+n+1}{2}=\frac{(n+1)^2+(n+1)}{2}$$

□

Principio de los casilleros

Si A y B son conjuntos finitos, con $\| A \| > \| B \| > 0$, y f es una función de A a B , entonces f no es inyectiva.

Demostración:

Lo demostraremos por inducción según el cardinal de B .

C.B.: Sean A y B finitos, y $\| A \| > \| B \| = 1$ ($B = \{b\}$) , y una función $f: A \to B$.

$f: A \to B \Rightarrow \exists a_1, a_2 \in A \mid f(a_1) = f(a_2) = b$, con $a_1 \neq a_2 \Rightarrow$

f no es inyectiva.

H.I: Si A y B son finitos, y $\| A \| > \| B \| = n$, con $n \geq 1$, y una función $f: A \to B$,

entonces f no es inyectiva.

P.I.: Sean A y B finitos, y $\| A \| > \| B \| = n+1$, con $n \geq 1$, y una función $f: A \to B$. Sea $b \in B$.

Sabemos que $B - \{b\} \neq \varnothing$ ya que $\| B \| \geq 2$.

Consideremos el conjunto $f^{-1}(b) = \{ a \in A \mid f(a) = b \}$

Tenemos dos casos a estudiar: $\| f^{-1}(b) \| \geq 2$ y $\| f^{-1}(b) \| \leq 1$.

Si $\| f^{-1}(b) \| \geq 2 \Rightarrow \exists a_1, a_2 \in A \mid f(a_1) = f(a_2) = b$, con $a_1 \neq a_2 \Rightarrow$ f no es inyectiva.

Si $\| f^{-1}(b) \| \leq 1$ entonces,

Sea $g: A - f^{-1}(b) \rightarrow B - \{b\} \mid g(a) = f(a) \ \forall a \in A - f^{-1}(b)$.

Como $\| f^{-1}(b) \| \leq 1 \Rightarrow \| A - f^{-1}(b) \| \geq \| A \| - 1$

Como $\| A \| > \| B \| = n+1 \Rightarrow \| A \| - 1 > \| B - \{b\} \| = n$ $\Rightarrow$

$\| A - f^{-1}(b) \| > \| B - \{b\} \| = n \Rightarrow g$ no es inyectiva $\Rightarrow$

$\uparrow$ *por H.I.*

$\exists a_1, a_2 \in A - f^{-1}(b) \mid g(a_1) = g(a_2) = c$, con $a_1 \neq a_2$ y $c \in B - \{b\} \Rightarrow$

$f(a_1) = f(a_2) \Rightarrow f$ no es inyectiva. □

Ejemplo 1.47:

Demostrar que cualquier subconjunto propio de un conjunto finito no es equipotencial con él.

Sea $\| A \| = n$ y sea B un subconjunto propio de $A \Rightarrow \| A \| > \| B \| > 0 \Rightarrow$

por el Principio de los Casilleros $\uparrow$

No existe función inyectiva de A a $B \Rightarrow$ No existe función biyectiva de A a $B \Rightarrow A$ y B no son equipotenciales. □

Principio de diagonalización

Sea R una relación binaria sobre un conjunto A $(R \subseteq A \times A)$.

Sea D el conjunto diagonal de R, definido como: $D = \{ a \in A \mid (a, a) \notin R \}$.

Sea $R_a = \{ b \in A \mid (a, b) \in R \}$, entonces $D \neq R_a \ \forall a \in A$.

Demostración :

$\forall a \in A$ tenemos que:

$(a, a) \in R \Rightarrow (a \in R_a) \wedge (a \notin D) \Rightarrow R_a \neq D$

$(a, a) \notin R \Rightarrow (a \notin R_a) \wedge (a \in D) \Rightarrow R_a \neq D$

$\Big\}$ $R_a \neq D \quad \forall a \in A$

□

Ejemplo 1.48:
Demostrar que $2^{\mathbb{N}}$ es no numerable.
Supongamos que $2^{\mathbb{N}}$ es infinito numerable, esto implica que $\exists f: \mathbb{N} \to 2^{\mathbb{N}} \mid f$ es biyectiva. Entonces $2^{\mathbb{N}}$ se puede numerar como $2^{\mathbb{N}} = \{ S_0, S_1, S_2, \ldots \}$, donde $S_i = f(i) \quad \forall i \in \mathbb{N}$.

a) Demostración con uso implícito del principio:
Sea $D = \{ n \in \mathbb{N} \mid n \notin S_n \}$
Como $D \subseteq \mathbb{N} \Rightarrow D \in 2^{\mathbb{N}} \Rightarrow \exists! k \in \mathbb{N} \mid D = S_k$
Pero, ¿ $k \in S_k$?, veámoslo:
Si $k \in S_k \Rightarrow k \notin D \Rightarrow k \notin S_k$ ➔➔
$\uparrow$
$D = S_k$
$\downarrow$
Si $k \notin S_k \Rightarrow k \in D \Rightarrow k \in S_k$ ➔➔ $\Rightarrow$
$D \neq S_k \quad \forall k \in \mathbb{N} \Rightarrow D \notin Rg(f) \Rightarrow f$ no es biyectiva $\Rightarrow 2^{\mathbb{N}}$ es no numerable.
$\uparrow$
➔➔ □ a)

b) Demostración con uso explícito del principio:
Sea $R = \{(i, j) \mid j \in f(i)\}$ sobre $\mathbb{N}$.
Sea D el conjunto diagonal de R , es decir, $D = \{n \in \mathbb{N} \mid (n, n) \notin R\}$.
Como $S_i = f(i)$ tenemos que $R_i = \{j \in \mathbb{N} \mid (i, j) \in R\}$ es S_i .
Por el principio de diagonalización sabemos que $R_n \neq D \quad \forall n \in \mathbb{N} \Rightarrow D \notin Rg(f)$
$\Rightarrow f$ no es biyectiva $\Rightarrow 2^{\mathbb{N}}$ es no numerable.
$\uparrow$
➔➔ □ b)

El teorema de Cantor visto anteriormente, al que hemos denominado así en su honor, es uno de los muchos enunciados y demostrados por este matemático. En realidad, se conoce con este nombre (*"El Teorema de Cantor"*), por su importancia, a una generalización del ejemplo anterior que enuncia: *"Para todo conjunto, el cardinal de su conjunto potencia es mayor que el cardinal del conjunto"*.

2
Lenguajes y Gramáticas

2.1 Concepto de Lenguaje

Definición 2.1: *Alfabeto*
Un alfabeto es un conjunto finito no vacío de símbolos.
Lo representaremos como Σ (a veces usaremos letras mayúsculas).

Definición 2.2: *Cadena sobre un alfabeto*
Una cadena sobre un alfabeto es una secuencia finita de símbolos del mismo.
Habitualmente, las cadenas suelen denotarse con letras minúsculas a partir de la '*u*', o letras minúsculas del principio del alfabeto griego (α, β, ...).

Definición 2.3: *Cadena vacía*
La cadena vacía es aquella que no tiene ningún símbolo.
La denotaremos como ε.

Definición 2.4: Σ^*, Σ^+
Definimos Σ^* como el conjunto de todas las cadenas, incluyendo la vacía, sobre Σ.
Definimos Σ^+ como el conjunto de todas las cadenas, excluyendo la vacía, sobre Σ.

Definición 2.5: *Longitud de una cadena ($|x|$)*
La longitud de una cadena se define como el número de símbolos que contiene.

Ejemplo 2.6:
Sea $\Sigma = \{a,b\}$
$|\varepsilon| = 0$
$|a| = 1$
$|abab| = 4$
Si $w = aba$, entonces $|w| = 3$ □

Definición 2.7: *Ocurrencia*

El símbolo $a \in \Sigma$ diremos que ocurre en la posición j-ésima de la cadena $w \in \Sigma^+$ sii el j-ésimo símbolo de w es a, y lo notaremos $w(j) = a$, con $0 < j \leq |w|$.

Ningún símbolo ocurre en la cadena vacía.

Al número de ocurrencias de un símbolo a en una cadena $w \in \Sigma^*$ lo notaremos $|\, w \,|_a$.

Nota: $|\, \varepsilon \,|_a = 0 \quad \forall a \in \Sigma$.

Ejemplo 2.8:

Sea $\Sigma = \{a,b,c\}$

Sea $w \in \Sigma^*$ con $w = aba$, entonces:

$w(1) = a$ $\qquad$ $|\, w \,|_a = 2$

$w(2) = b$ $\qquad$ y $\qquad$ $|\, w \,|_b = 1$

$w(3) = a$ $\qquad$ $|\, w \,|_c = 0$ $\qquad$ □

Definición 2.9: *Concatenación de cadenas*

Dadas dos cadenas $x, y \in \Sigma^+$, x concatenada con y, notado $x \cdot y$ o simplemente xy, será la cadena que cumple:

a) $|\, xy \,| = |\, x \,| + |\, y \,|$

b) $xy(j) = \begin{cases} x(j), & \text{si} \quad j \leq |\, x \,| \\ y\,(j - |\, x \,|), & \text{si} \quad j > |\, x \,| \end{cases}$ $\qquad$ con $0 < j \leq |\, xy \,|$

Dada una cadena $x \in \Sigma^*$, $x\,\varepsilon = \varepsilon\, x = x$.

Notas: Dadas tres cadenas $x, y, z \in \Sigma^*$, se cumple que $(xy)\,z = x\,(yz)$.
Por tanto, $(\Sigma^*, \cdot)$ es un monoide, al ser la concatenación una operación interna asociativa y con elemento neutro.

Definición 2.10: *Subcadena*

Una cadena $v \in \Sigma^*$ es subcadena de otra $w \in \Sigma^*$ sii $\exists x, y \in \Sigma^* \mid w = xvy$.

Notas: Toda cadena es subcadena de sí misma ($x = y = \varepsilon$) .
La cadena vacía es subcadena de toda cadena ($x = w;\ v = y = \varepsilon$) .

Definición 2.11: *Sufijo*

Sean $w, v \in \Sigma^*$, decimos que v es un sufijo de w sii $\exists x \in \Sigma^* \mid w = xv$.

Definición 2.12: *Prefijo*

Sean $w, v \in \Sigma^*$, decimos que v es un prefijo de w sii $\exists y \in \Sigma^* \mid w = vy$.

Ejemplo 2.13:

Sea $v = ata$.

Si $w = catar$ entonces

v es subcadena de w ya que $\exists x=c, y=r \in \Sigma^* \mid w = xvy$

Si $w = atalaya$ entonces

v es prefijo de w ya que $\exists x=laya \in \Sigma^* \mid w = vx$

Si $w = rata$ entonces

v es sufijo de w ya que $\exists x=r \in \Sigma^* \mid w = xv$ □

Definición 2.14: *Potencia de una cadena (w^n)*

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

Ejemplo 2.15:

Calcular $(pa)^2$

$$(pa)^2 = \left\{ \begin{array}{l} \text{con } i = 1 : (pa)^2 = (pa)^1 pa \\ \text{con } i = 0 : (pa)^1 = (pa)^0 pa \\ \text{C.B.: } (pa)^0 = \varepsilon \end{array} \right\} \Rightarrow (pa)^2 = ((\varepsilon pa)\, pa) = papa$$

□

Definición 2.16: *Inversión de una cadena (w^R) (cadena inversa)*

$$w^R \begin{cases} \text{si } |w| = 0 \;\Rightarrow\; w^R = w = \varepsilon \\ \text{si } |w| > 0 \;\text{ y }\; w = ua \;\text{ con }\; u \in \Sigma^* \;\text{ y }\; a \in \Sigma \;\Rightarrow\; w^R = au^R \end{cases}$$

Ejemplo 2.17:
Sea $w = casa$. Obtener w^R .
$(casa)^R = a(cas)^R = as(ca)^R = asa(c)^R = asac(\varepsilon)^R = asac\varepsilon = asac$ □

Proposición 2.18:
Para toda cadena $x,w \in \Sigma^*$ tenemos que: $(wx)^R = x^R w^R$

Demostración:
Realizaremos esta demostración empleando la técnica de inducción aplicada sobre la longitud de la cadena x.

1º. C.B.: $|x|=0 \;\Rightarrow\; x=\varepsilon \;\Rightarrow\; (wx)^R = (w\varepsilon)^R = w^R = \varepsilon w^R = \varepsilon^R w^R = x^R w^R$
2º. H.I.: $|x| \le n \;\Rightarrow\; (wx)^R = x^R w^R$
3º. P.I.: Sea $|x| = n+1 \;\Rightarrow\; x = ua$ con $u \in \Sigma^*$, $a \in \Sigma$ y $|u| = n$

$$(wx)^R = (w(ua))^R = ((wu)a)^R = a(wu)^R = au^R w^R = (ua)^R w^R =$$

↑	↑	↑	↑	↑	↑
$x = ua$	*por asociatividad en la concatenación*	*por inversión de una cadena*	*H.I.*	*por inversión de una cadena*	$x = ua$

$x^R w^R$ □

Ejemplo 2.19:
Sean $x, y \in \Sigma^*$, donde $x = si$ e $y = entonces$.
Vemos que $(xy)^R = y^R x^R$ pues $(sientonces)^R = (entonces)^R (si)^R = secnotneis$
□

Definición 2.20: *Lenguaje*
L es un lenguaje sobre Σ sii $L \subseteq \Sigma^*$.

Nota: Un lenguaje es cualquier conjunto de cadenas sobre un alfabeto Σ.

Ejemplo 2.21:
Σ^* es un lenguaje
$\{ \varepsilon \}$ es un lenguaje
$\varnothing$ es un lenguaje
$\{ \{ \varepsilon \} \}$ no es un lenguaje (sobre el alfabeto Σ)
$\{\varepsilon\} \neq \varnothing$; $\| \{\varepsilon\} \| = 1$; $\| \varnothing \| = 0$

Nota: Puesto que no haremos distinción entre los símbolos de un alfabeto y las cadenas de longitud uno sobre dicho alfabeto, tenemos que Σ también es un lenguaje. □

Si un lenguaje es finito podemos definirlo por *extensión* (dando todos sus elementos).

Ejemplo 2.22:
Sea $\Sigma = \{ a, b, c, d, f, r, z \}$
Sea $L \subseteq \Sigma^*$, $L = \{ aba, czr, d, f \}$ □

Si un lenguaje es infinito sólo podemos definirlo por *comprensión*, es decir, $L = \{ w \in \Sigma^* \mid P(w) \}$, donde P es una propiedad que verifican todas las cadenas que pertenecen a L y no verifican las que no pertenecen.

Ejemplo 2.23:
Sea $\Sigma = \{ 0, 1 \}$
Sea $L \subseteq \Sigma^*$, $L = \{ w \in \Sigma^* \mid |w|_0 = 2n+1 \text{ con } n \in \mathbb{N} \}$ □

Proposición 2.24:
Si Σ es un alfabeto, entonces Σ^* es infinito numerable.

Demostración:
Hemos de demostrar que $\| \Sigma^* \| = \aleph_0$.
Necesitamos construir una biyección $f: \Sigma^* \to \mathbb{N}$
Así, fijada una ordenación arbitraria del alfabeto $\Sigma = \{ a_1, a_2, ..., a_n \}$, podemos numerar los elementos de Σ^* de la siguiente forma:

1º) Todas las cadenas de longitud k (de las cuales encontramos un total de n^k) se numeran antes que todas las cadenas de longitud $k+1$, con $k \geq 0$, es decir:
Sea $W_k = \{w \in \Sigma^* \mid |w| = k\}$. $||W_k|| = n^k$
W_k se numera antes que W_{k+1} , $\forall k \geq 0$

2º) Las n^k cadenas de longitud k se numeran (ordenan) lexicográficamente, es decir:
La cadena $a_{i_1} \ldots a_{i_k}$ precede a la cadena $a_{j_1} \ldots a_{j_k}$ si
$\exists m \in \mathbb{N},\ 0 \leq m \leq k\text{-}1 \mid i_p = j_p$ para $p=1, \ldots, m$ y $i_{m+1} < j_{m+1}$

De esta forma obtenemos la función biyectiva que estábamos buscando.
La expresión de esta función, a partir de la ordenación arbitraria ya fijada del alfabeto $\Sigma = \{ a_1, a_2, \ldots, a_n \}$, para cualquier cadena dada $w = a_{i_1} a_{i_2} \ldots a_{i_{|w|}}$ con $i_k \in \{ 1, \ldots, n \}$, es:

$$f : \Sigma^* \rightarrow \mathbb{N}$$

$$f(w) = \sum_{j=1}^{|w|} n^{(|w|-j)}\ i_j$$ □

Ejemplo 2.25:
Sea $\Sigma = \{ a_1, a_2 \}$

$|w| = 0$ ε 0
$|w| = 1$ a_1 1
a_2 2
$|w| = 2$ a_1a_1 3
a_1a_2 4
a_2a_1 5
a_2a_2 6
$|w| = 3$ $a_1a_1a_1$ 7
⋮

□

2.2 Concepto de Representación

El concepto de "*Representación*" es muy amplio. Nosotros vamos a restringirnos aquí al ámbito de la Informática Teórica, dando una definición formal del mismo que es independiente de los conceptos "computable" y "decidible".

Sea un alfabeto Σ, y Σ^* el conjunto de todas las cadena posibles sobre ese alfabeto. Cualquier subconjunto L de Σ^* diremos que es un lenguaje sobre Σ, y llamaremos 2^{Σ^*} al conjunto de todos los lenguajes sobre Σ.

Sabemos que la cardinalidad de estos conjuntos es:

Σ es finito

Σ^* es infinito numerable

L es numerable (infinito numerable o finito)

2^{Σ^*} es infinito no numerable (ya que $2^{\aleph_0} = \aleph_1$)

Sea Σ_M el alfabeto del lenguaje natural más todos los símbolos matemáticos, es decir, nuestro *alfabeto matemático*, que utilizamos en el *lenguaje matemático* (mejor cabría decir *metalenguaje*). Cualquier descripción o definición (finitas) de un lenguaje L sobre Σ en un metalenguaje matemático (el habitual) no es más que una cadena de Σ_M^*.

Diremos que una cadena $r \in \Sigma_M^*$ es una representación de un lenguaje L sobre Σ si existe una relación SR (Sistema de Representación) incluida en $\Sigma_M^* \times 2^{\Sigma^*}$ tal que $(r, L) \in SR$ y se cumple que, para todo $L' \neq L$ $(r, L') \notin SR$.

Es decir:

$r \in \Sigma_M^*$ es una representación de un lenguaje L sobre Σ sii

$$\exists SR \subset \Sigma_M^* \times 2^{\Sigma^*} \mid (r, L) \in SR \ \wedge \ \forall L' \neq L \ \ (r, L') \notin SR$$

Si r es una representación de un lenguaje L entonces diremos que dicho lenguaje es representable. Al conjunto de todos los lenguajes representables lo notaremos $L.REP$.

Dos representaciones serán equivalentes si representan el mismo lenguaje.

Al fijar un sistema de representación (SR) queda definido el conjunto de las representaciones de lenguajes para ese SR . Dicho conjunto de representaciones,

al que notaremos REP, siempre será numerable, pues REP está incluido en Σ_M^* que es numerable.

De aquí en adelante, consideraremos que Σ_M y SR son los habituales.

2.3 Cardinalidad, Representaciones y Lenguajes

Según hemos visto en el apartado precedente, una representación no es más que una cadena sobre un alfabeto Σ_M (y para un SR dado), que cumple que sólo representa un lenguaje, es decir, si dos lenguajes representables son diferentes han de serlo sus representaciones, por tanto: $\| L.REP \| \leq \| REP \|$.

Estudiamos a continuación cuántos lenguajes son representables y cuántos no.

Sabemos que $\| \Sigma_M^* \| = \aleph_0$, y como el conjunto de representaciones está incluido en Σ_M^* tenemos que $\| REP \| \leq \aleph_0$.

Por otra parte, es trivial que hay una cantidad infinita de lenguajes finitos ($L.finitos$), y como todo lenguaje finito es representable (ya que lo podemos definir por extensión), tenemos que $\aleph_0 \leq \| L.REP \| \leq \| REP \|$.

Por tanto:

$$\aleph_0 \leq \| L.REP \| \leq \| REP \| \leq \aleph_0 \quad \Rightarrow \quad \| L.REP \| = \| REP \| = \aleph_0$$

es decir, hay una cantidad infinita numerable ($\aleph_0$) de representaciones y de lenguajes representables.

Como por definición $L \subseteq \Sigma^*$, el conjunto de todos los lenguajes sobre un alfabeto Σ es 2^{Σ^*}. De esta forma, al ser $\| \Sigma^* \| = \aleph_0$ tenemos que $\| 2^{\Sigma^*} \| = \aleph_1$, es decir, hay una cantidad infinita no numerable ($\aleph_1$) de lenguajes.

Si al total de lenguajes (2^{Σ^*}) le restamos los que son representables ($L.REP$) obtenemos los lenguajes no representables ($L.NOREP$), y como $\aleph_1 - \aleph_0 = \aleph_1$ podemos afirmar que hay una cantidad infinita no numerable ($\aleph_1$) de lenguajes no representables.

2.4 Representación de Lenguajes

Recordamos que la forma general de definir un lenguaje infinito (también válida para finitos) es $L = \{ w \in \Sigma^* \mid P(w) \}$.

Además de las formas habituales de expresar P , en el caso de los lenguajes formales hay dos maneras específicas de definirlos, con un *dispositivo reconocedor* y con un *dispositivo generador*.

Definición 2.26: *Algoritmo conclusivo*
Secuencia finita de instrucciones finitas precisas (no ambiguas) que, ante un cierto tipo de entrada/pregunta/problema, nos devuelve (calcula) siempre, en un tiempo finito, una salida/respuesta/solución.

Definición 2.27: *Dispositivo reconocedor de lenguaje*
Un algoritmo conclusivo diseñado para un lenguaje L que contesta correctamente a las preguntas: "*¿Pertenece la cadena w a L ?*" se denomina dispositivo reconocedor de L .

Definición 2.28: *Dispositivo generador de lenguaje*
Un sistema de algoritmos conclusivos diseñado para un lenguaje L que produce todas y sólo las cadenas de L se denomina dispositivo generador de L .

Ejemplo 2.29:
Sea $L_1 = \{ w \in \{a, b\}^* \mid w$ no contiene la subcadena $bbb \}$.

Un dispositivo reconocedor de L_1 podría ser el siguiente:

Vamos leyendo la cadena de izquierda a derecha y (tras inicializar)

1º. mantenemos un contador que es puesto a cero cada vez que leemos una '*a*'
2º. sumamos uno al contador cada vez que leemos una '*b*'
3º. paramos y damos como respuesta "NO" si el contador alcanza tres
4º. paramos y damos como respuesta "SI" si se lee completa la cadena sin que el contador llegue a tres

Un dispositivo generador de L_1 podría ser el siguiente:

" Para producir un elemento de L_1 primero no escribo nada, o escribo '*b*' o bien '*bb*'; a continuación repito, cero o más veces, la operación de escribir '*a*', o bien '*ab*' o bien '*abb*' ".

□

Al contrario que los dispositivos reconocedores, los dispositivos generadores no son algoritmos, ya que estos dispositivos resultan ambiguos (elección libre de opciones). Los dispositivos reconocedores que definiremos se denominan *autómatas*, y los generadores *gramáticas*.

2.5 Concepto de Gramática

A continuación veremos la definición formal de gramática, posteriormente su clasificación, y a partir de ahí introduciremos la de los lenguajes.

Definición 2.30: *Gramática*

Una gramática es una cuádrupla $G = (N, T, P, S)$ donde:

N es un alfabeto, el *alfabeto no terminal*, formado por *símbolos no terminales*

T es un alfabeto, el *alfabeto terminal*, formado por *símbolos terminales*

$N \cap T = \varnothing$

$N \cup T = V$

$S \in N$ se denomina *axioma*

$P \subset V^+ \times V^*$ y es finito

Los elementos de P, $(\alpha, \beta) \in P$, se denominan *reglas de producción*, y las notaremos $\alpha \to \beta$. Así, P es finito con

$$P = \{\alpha \to \beta, \text{ con } \alpha \in V^+ \wedge \beta \in V^*\}.$$

Generalmente los símbolos que pertenecen al alfabeto no terminal (N) se representan con letras mayúsculas del principio del alfabeto, mientras que los que pertenecen al alfabeto terminal (T) se representan con letras minúsculas del principio del alfabeto.

Definición 2.31: *Producir directamente*
Sea una gramática $G = (N, T, P, S)$.
Dadas $x \in V^+$ e $y \in V^*$, se dice que x produce directamente y, notado $x \Rightarrow y$, si $\exists u,v \in V^*$ tales que:

a) $x = uzv$
b) $y = u\beta v$
c) $\exists (z \to \beta) \in P$ con $z \in V^+$ y $\beta \in V^*$

Ejemplo 2.32:
Sea $aBaCaab$ una cadena.
Si $(aB \to b) \in P$, entonces $aBaCaab \Rightarrow baCaab$
Si $(C \to BaD) \in P$, entonces $aBaCaab \Rightarrow aBaBaDaab$ □

Definición 2.33: *Producir en n pasos*
Sea una gramática $G = (N, T, P, S)$, y sean $x,y \in V^*$.

Diremos que x produce en cero pasos y, notado $x \Rightarrow^0 y$, sii $x = y$.

Diremos que x produce en un paso y, notado $x \Rightarrow^1 y$, sii $x \Rightarrow y$.

Diremos que x produce en n pasos y, con $n>1$, notado $x \Rightarrow^n y$, sii:
$\exists z_1, z_2, ..., z_{n-1} \in V^+ \mid x \Rightarrow z_1, \quad z_1 \Rightarrow z_2, \quad ..., \quad z_{n-1} \Rightarrow y$

Definición 2.34: *Producir en al menos un paso*
Diremos que x produce en al menos un paso y, notado $x \Rightarrow^+ y$, sii
$\exists n>0 \mid x \Rightarrow^n y$.

Definición 2.35: *Producir*
Diremos que x produce y, notado $x \Rightarrow^* y$, sii $\exists n \geq 0 \mid x \Rightarrow^n y$.

En resumen:
$\Rightarrow$ es una relación binaria entre cadenas (sobre V^*)
$\Rightarrow^+$ es su cierre transitivo
$\Rightarrow^*$ es su cierre reflexivo y transitivo

Definición 2.36: *Derivación, Longitud de una derivación*
Dada $G = (N, T, P, S)$, denominamos derivación de G a toda secuencia finita de producciones directas de la forma:

$w_0 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$ donde $w_0 = S$ y $w_i \in V^*$ con $0 \le i \le n$

La longitud de dicha derivación es n ($n \in \mathbb{N}$).

Definición 2.37: *Forma sentencial*
Sea $G = (N, T, P, S)$ una gramática.
Diremos que α es una forma sentencial de G sii

$$S \Rightarrow^* \alpha \quad \text{con} \quad \alpha \in V^*$$

Definición 2.38: *Cadena generada por una gramática*
Se dice que y es generada por una gramática $G = (N, T, P, S)$ sii

$$(S \Rightarrow^* y) \wedge (y \in T^*)$$

Definición 2.39: *Lenguaje generado por una gramática*
Sea $G = (N, T, P, S)$, definimos el lenguaje generado por G, notado $L(G)$, como:

$$L(G) = \{ y \in T^* \mid S \Rightarrow^* y \}$$

Ejemplo 2.40:
Sea $G = (N, T, P, S)$ una gramática con:

$N = \{S\}$
$T = \{a, b\}$
$P = \{S \rightarrow aSb,\ S \rightarrow ab\}$

Mediante la aplicación de la primera regla ($S \rightarrow aSb$) $n-1$ veces, seguida de la aplicación de la segunda ($S \rightarrow ab$), tenemos :

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow \ldots \Rightarrow a^{n-1}Sb^{n-1} \Rightarrow a^n b^n$$

En esta expresión (que es una derivación) todas las cadenas son formas sentenciales.
Observamos por tanto que $L(G) = \{ w \in \{a,b\}^* \mid w = a^n b^n \quad \text{con} \quad n \ge 1 \}$. □

Ejemplo 2.41:

Sea $G = (N, T, P, S)$ con:

$N = \{A, B\}$

$T = \{a, b\}$

$P = \{A \to AaA,\ A \to b\}$

$S = A$

$$A \Rightarrow b \in L(G)$$
$$\uparrow$$
$$A \to b$$

$$A \Rightarrow AaA \Rightarrow AaAaA \Rightarrow baAaA \Rightarrow baAab \Rightarrow babab \in L(G)$$
$$\uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow$$
$$A \to AaA \quad A \to AaA \quad A \to b \quad A \to b \quad A \to b$$

$L(G) = \{ b, bab, babab, ... \} = \{ w \in \{a,b\}^* \mid w = b(ab)^n \text{ con } n \geq 0 \}.$ □

Definición 2.42: *Gramáticas equivalentes ($G_1 \equiv G_2$)*

Sean G_1 y G_2 dos gramáticas, decimos que son equivalentes sii $L(G_1) = L(G_2)$

Ejemplo 2.43:

Sean $G_1 = (N_1, T_1, P_1, S_1)$ y $G_2 = (N_2, T_2, P_2, S_2)$ dos gramáticas donde:

$N_1 = \{A\}$	$N_2 = \{A\}$
$T_1 = \{a, b\}$	$T_2 = \{a, b\}$
$P_1 = \{A \to a, A \to Aa\}$	$P_2 = \{A \to a,\ A \to aa,\ A \to Aaa\}$
$S_1 = A$	$S_2 = A$

$L(G_1) = L(G_2) = \{a, aa, aaa, ... \} = \{ w \in \{a,b\}^* \mid w = a^n \text{ con } n \geq 1 \}$, por tanto, G_1 y G_2 son equivalentes. □

2.6 Clasificación de Gramáticas

La clasificación de las gramáticas se realiza en función de sus reglas de producción, es decir, según los elementos de P. Empezaremos por tanto definiendo los distintos tipos de reglas de producción.

Definición 2.44: *Regla de tipo 0 (con estructura de frase)*
Dada una gramática $G = (N, T, P, S)$ diremos que una regla que pertenece a P es de tipo *0* sii es de la forma

$$\alpha \to \beta \quad \text{con} \quad (\alpha \in V^+) \wedge (\beta \in V^*) .$$

Nota: Toda regla de producción es de tipo *0*.

Definición 2.45: *Regla de tipo 1 (sensible al contexto)*
Dada una gramática $G = (N, T, P, S)$ diremos que una regla que pertenece a P es de tipo *1* sii es de la forma

$$\alpha A \beta \to \alpha \gamma \beta \quad \text{con} \quad (\alpha, \beta \in V^*) \wedge (\gamma \in V^+) \wedge (A \in N) .$$

Definición 2.46: *Regla de tipo 2 (de contexto libre)*
Dada una gramática $G = (N, T, P, S)$ diremos que una regla que pertenece a P es de tipo *2* sii es de la forma

$$A \to \alpha \quad \text{con} \quad (A \in N) \wedge (\alpha \in V^+) .$$

Nota: Denominarla *"independiente del contexto"* sería más correcto, pero no es lo usual.

Definición 2.47: *Regla regular izquierda*
Dada una gramática $G = (N, T, P, S)$ diremos que una regla que pertenece a P es regular izquierda sii es de la forma

$$A \to aB \quad \text{con} \quad (A,B \in N) \wedge (a \in T) .$$

Definición 2.48: *Regla regular derecha*
Dada una gramática $G = (N, T, P, S)$ diremos que una regla que pertenece a P es regular derecha sii es de la forma

$$A \to Ba \quad \text{con} \quad (A,B \in N) \wedge (a \in T) .$$

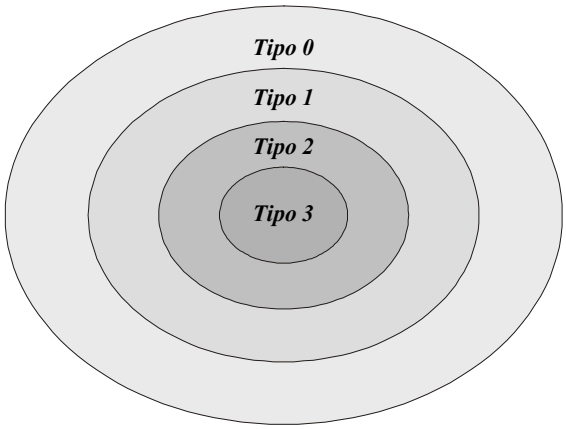
Definición 2.49: *Regla regular terminal*
Dada una gramática $G = (N, T, P, S)$ diremos que una regla que pertenece a P es regular terminal sii es de la forma

$$A \to a \quad \text{con} \quad (A \in N) \wedge (a \in T) .$$

Definición 2.50: *Regla de tipo 3 (regular)*
Dada una gramática $G = (N, T, P, S)$ diremos que una regla que pertenece a P es de tipo *3* sii es regular derecha, o regular izquierda, o regular terminal.

Según estas definiciones se establecen las siguientes inclusiones de los tipos de reglas de producción:

$$Tipo\ 3 \subset Tipo\ 2 \subset Tipo\ 1 \subset Tipo\ 0$$



Ejemplo 2.51:
Sea una gramática $G = (N, T, P, S)$ con $N = \{S,B\}$ y $T = \{c,d\}$.
Veamos el tipo de algunas reglas pertenecientes a P :

$S \to c$ es regular terminal, y por tanto de tipo *3*.
$S \to cB$ es regular izquierda, y por tanto de tipo *3*.
$S \to Sc$ es regular derecha, y por tanto de tipo *3*.
$B \to cc$ es de tipo *2* y no es de tipo *3*.
$B \to SS$ es de tipo *2* y no es de tipo *3*.
$B \to B$ es de tipo *2* y no es de tipo *3*.
$S \to cBSccd$ es de tipo *2* y no es de tipo *3*.
$BB \to cB$ es de tipo *1* y no es de tipo *2*.
$BS \to BdS$ es de tipo *1* y no es de tipo *2*.

$cdBSScB \rightarrow cdBcBBdScB$ es de tipo *1* y no es de tipo *2*.
$BBB \rightarrow SSS$ es de tipo *0* y no es de tipo *1*.
$cB \rightarrow Bc$ es de tipo *0* y no es de tipo *1*.
$SS \rightarrow S$ es de tipo *0* y no es de tipo *1*.
$c \rightarrow d$ es de tipo *0* y no es de tipo *1*.
$B \rightarrow \varepsilon$ es de tipo *0* y no es de tipo *1*.
$c \rightarrow \varepsilon$ es de tipo *0* y no es de tipo *1*. □

Definición 2.52: *Regla lineal terminal*
Dada una gramática $G = (N, T, P, S)$ diremos que una regla que pertenece a P es lineal terminal sii es de la forma

$$A \rightarrow \alpha \quad \text{con} \quad (A \in N) \wedge (\alpha \in T^{+}) .$$

Definición 2.53: *Regla lineal*
Dada una gramática $G = (N, T, P, S)$ diremos que una regla que pertenece a P es lineal sii es lineal terminal o es de la forma

$$A \rightarrow \alpha B \beta \quad \text{con} \quad (A,B \in N) \wedge (\alpha ,\beta \in T^{*})$$

Definición 2.54: *Regla lineal izquierda*
Dada una gramática $G = (N, T, P, S)$ diremos que una regla que pertenece a P es lineal izquierda sii es de la forma

$$A \rightarrow \alpha B \quad \text{con} \quad (A,B \in N) \wedge (\alpha \in T^{*}) .$$

Definición 2.55: *Regla lineal derecha*
Dada una gramática $G = (N, T, P, S)$ diremos que una regla que pertenece a P es lineal derecha sii es de la forma

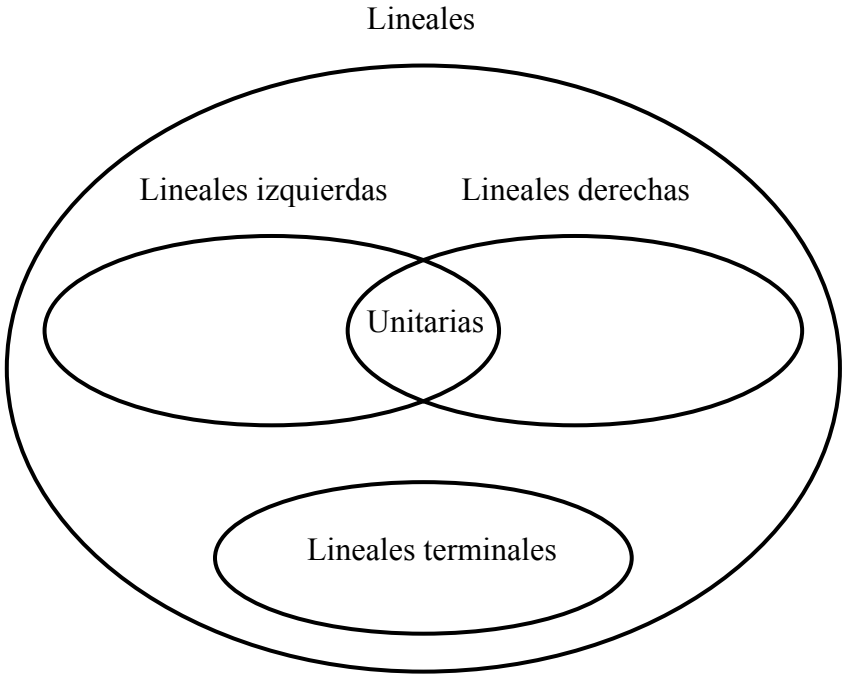
$$A \rightarrow B \alpha \quad \text{con} \quad (A,B \in N) \wedge (\alpha \in T^{*}) .$$

Definición 2.56: *Regla unitaria*
Dada una gramática $G = (N, T, P, S)$ diremos que una regla que pertenece a P es unitaria sii es de la forma

$$A \rightarrow B \quad \text{con} \quad A,B \in N .$$

Nota: Una regla puede ser lineal sin ser lineal izquierda ni lineal derecha.

Estas definiciones establecen las siguientes inclusiones dentro de las reglas lineales:



Ejemplo 2.57:
Sea una gramática $G = (N, T, P, S)$ con $N = \{S,B\}$ y $T = \{c,d\}$.
Veamos el tipo de algunas reglas pertenecientes a P :

$S \to B$ es unitaria, y por tanto lineal.

$S \to S$ es unitaria, y por tanto lineal.

$B \to c$ es lineal terminal.

$S \to cdd$ es lineal terminal.

$B \to cS$ es lineal izquierda.

$B \to dddB$ es lineal izquierda.

$S \to Sc$ es lineal derecha.

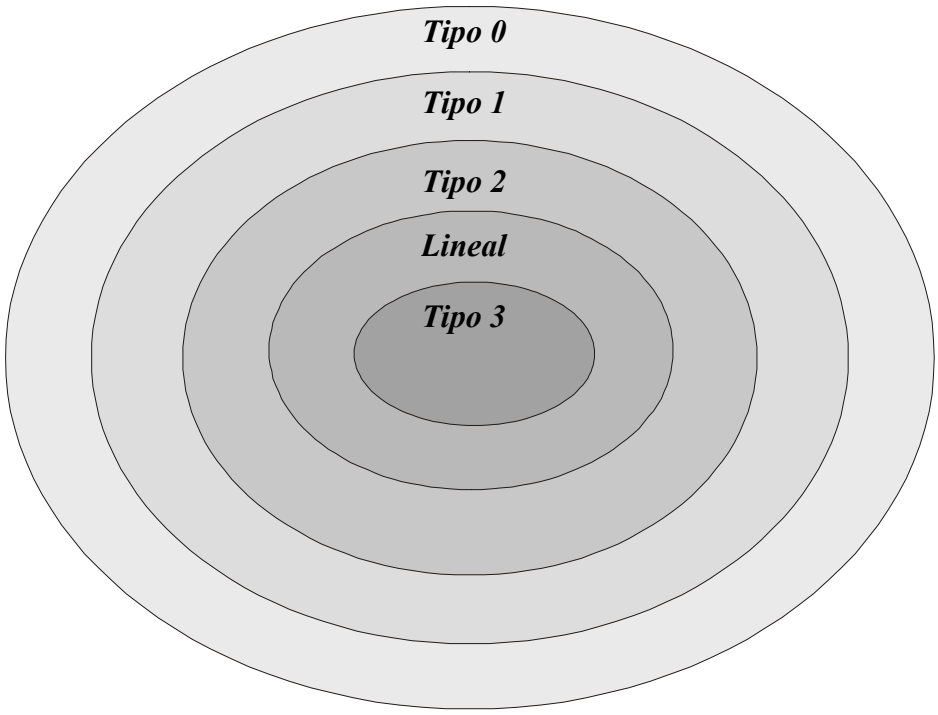
$S \to Bcdcc$ es lineal derecha.

$B \to cBd$ es lineal, y no es ni lineal izquierda ni lineal derecha.

$S \to cdcBc$ es lineal, y no es ni lineal izquierda ni lineal derecha. □

Si añadimos las reglas lineales a los tipos de reglas anteriores tenemos las siguientes inclusiones:

$$Tipo\ 3 \subset Lineales \subset Tipo\ 2 \subset Tipo\ 1 \subset Tipo\ 0$$



Veamos ahora la clasificación de las gramáticas en función de los tipos de reglas definidos.

Definición 2.58: *Gramática con estructura de frase (tipo 0) (GEF)*
Diremos que una gramática es con estructura de frase sii todas sus reglas son de tipo *0* (con estructura de frase).

Nota: Toda gramática es de tipo *0*.

Definición 2.59: *Gramática sensible al contexto (tipo1) (GSC)*
Diremos que una gramática es sensible al contexto sii todas sus reglas son de tipo *1* (sensible al contexto).

Definición 2.60: *Gramática de contexto libre (tipo 2) (GCL)*
Diremos que una gramática es de contexto libre sii todas sus reglas son de tipo *2* (de contexto libre).

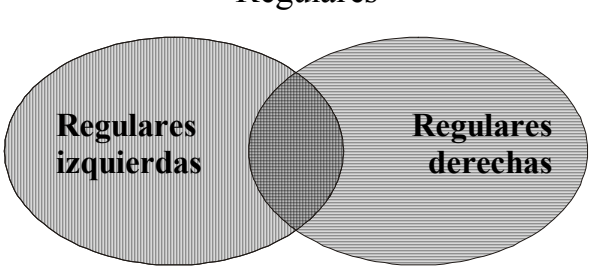
Nota: Denominarla *"independiente del contexto"* sería más correcto, pero no es lo usual.

Definición 2.61: *Gramática regular (tipo 3) (GR)*
Diremos que una gramática es regular sii es *regular izquierda* o *regular derecha*.

Diremos que una gramática es regular izquierda *(GRI)* sii cada una de sus reglas es regular izquierda o regular terminal.

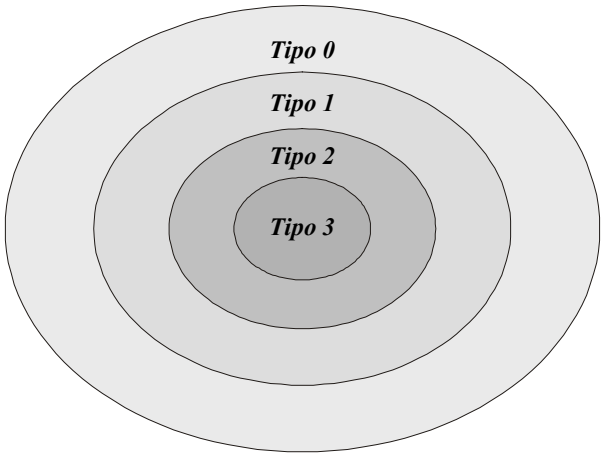
Diremos que una gramática es regular derecha *(GRD)* sii cada una de sus reglas es regular derecha o regular terminal.

Notas: $G = (\{A,B\}, \{a\}, \{ A \rightarrow aB, A \rightarrow Ba \}, A)$ no es una gramática regular.
$G = (\{A,B\}, \{a\}, \{ A \rightarrow a \}, A)$ es regular derecha y regular izquierda.



La *Jerarquía de Chomsky* de las gramáticas establece las siguientes inclusiones:

$$Tipo\ 3 \subset Tipo\ 2 \subset Tipo\ 1 \subset Tipo\ 0$$



Ejemplo 2.62:

Sea una gramática $G = (N, T, P, S)$ con $N = \{S, A, B\}$ y $T = \{a, b, c\}$:

Si $P = \{ S \to a, S \to aS \}$ entonces G es regular izquierda, y por tanto regular (tipo *3*).

Si $P = \{ S \to b, B \to Bc \}$ entonces G es regular derecha, y por tanto regular (tipo *3*).

Si $P = \{ S \to c, S \to aA, S \to bB, A \to aA, A \to a, B \to bB \}$ entonces G es regular izquierda, y por tanto regular (tipo *3*).

Si $P = \{ S \to c, A \to a, B \to b \}$ entonces G es regular izquierda y regular derecha, y por tanto regular (tipo *3*).

Si $P = \{ S \to a, S \to SS \}$ entonces G es de contexto libre (tipo *2*) y no es regular (tipo *3*).

Si $P = \{ A \to ccc \}$ entonces G es de contexto libre (tipo *2*) y no es regular (tipo *3*).

Si $P = \{ S \to A, A \to a \}$ entonces G es de contexto libre (tipo *2*) y no es regular (tipo *3*).

Si $P = \{ S \to aaScc, S \to b \}$ entonces G es de contexto libre (tipo *2*) y no es regular (tipo *3*).

Si $P = \{ S \to a, S \to SS, SS \to SSS \}$ entonces G es sensible al contexto (tipo *1*) y no es de contexto libre (tipo *2*).

Si $P = \{ S \rightarrow b,\ aBAc \rightarrow aBAcc \}$ entonces G es sensible al contexto (tipo *1*) y no es de contexto libre (tipo *2*).

Si $P = \{ A \rightarrow aAb,\ aA \rightarrow aaA,\ aA \rightarrow aa,\ Ab \rightarrow Abb,\ aAb \rightarrow aabb \}$ entonces G es sensible al contexto (tipo *1*) y no es de contexto libre (tipo *2*).

Si $P = \{ S \rightarrow a,\ S \rightarrow SS,\ SS \rightarrow S \}$ entonces G es con estructura de frase (tipo *0*) y no es sensible al contexto (tipo *1*).

Si $P = \{ S \rightarrow a,\ a \rightarrow b \}$ entonces G es con estructura de frase (tipo *0*) y no es sensible al contexto (tipo *1*).

Si $P = \{ S \rightarrow a,\ a \rightarrow aa,\ a \rightarrow b,\ b \rightarrow c \}$ entonces G es con estructura de frase (tipo *0*) y no es sensible al contexto (tipo *1*).

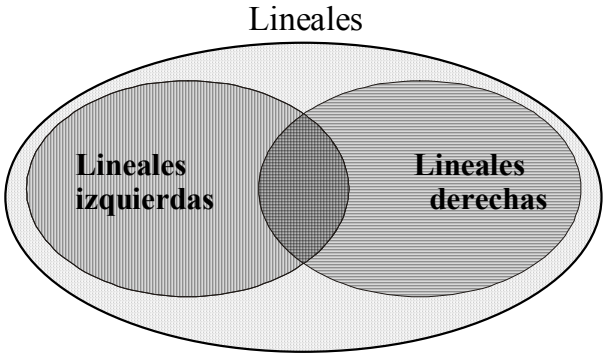
Si $P = \{ S \rightarrow a,\ aB \rightarrow Ba,\ SSS \rightarrow BBB \}$ entonces G es con estructura de frase (tipo *0*) y no es sensible al contexto (tipo *1*). □

Definición 2.63: *Gramática lineal (GL)*
Diremos que una gramática es lineal sii todas sus reglas son lineales.

Definición 2.64: *Gramática lineal izquierda (GLI)*
Diremos que una gramática es lineal izquierda sii cada una de sus reglas es lineal izquierda o lineal terminal.

Definición 2.65: *Gramática lineal derecha (GLD)*
Diremos que una gramática es lineal derecha sii cada una de sus reglas es lineal derecha o lineal terminal.

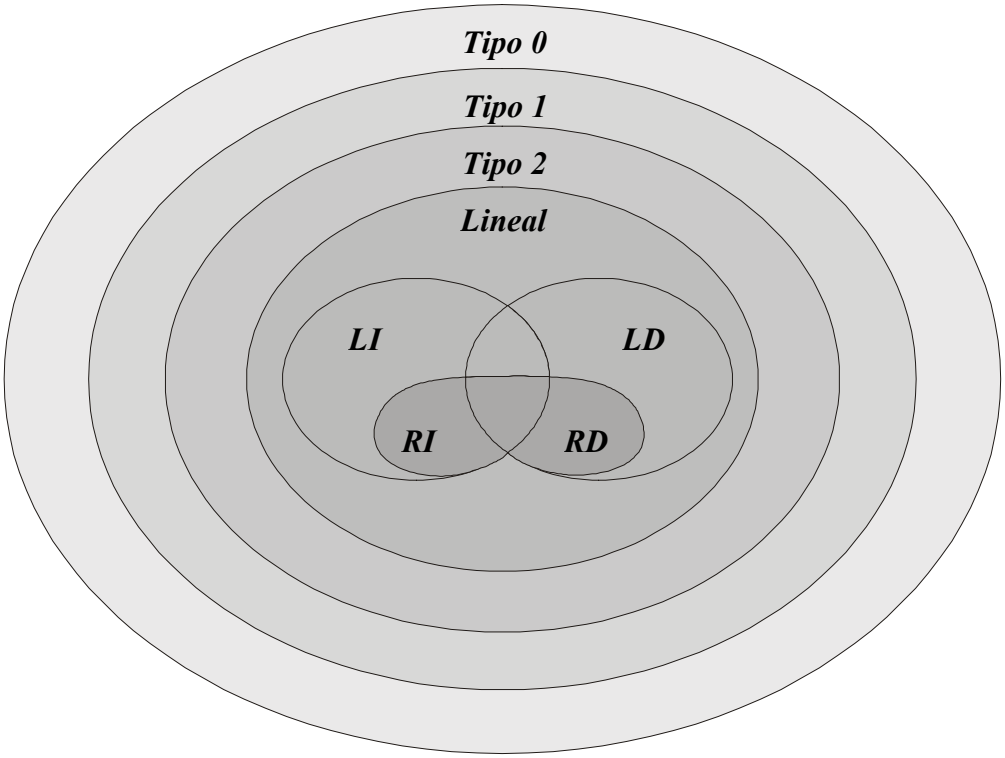
Nota: Una gramática puede ser lineal sin ser lineal izquierda ni lineal derecha.



Si tenemos en cuenta las gramáticas lineales además de las asociadas a la Jerarquía de Chomsky, tenemos que:

$$Tipo\ 3 \subset Lineales \subset Tipo\ 2 \subset Tipo\ 1 \subset Tipo\ 0$$

Una representación gráfica de dicha jerarquía, incluyendo los tipos de gramáticas lineales y regulares, es:



Ejemplo 2.66:
Sea una gramática $G = (N, T, P, S)$ con $N = \{S, A, B\}$ y $T = \{a, b, c\}$:

Si $P = \{ S \to aaa, S \to bbb \}$ entonces G es lineal izquierda y lineal derecha, y por tanto lineal.

Si $P = \{ S \to a, S \to aaS \}$ entonces G es lineal izquierda (y por tanto lineal), y no es lineal derecha.

Si $P = \{ S \to a, S \to aS, S \to aaS \}$ entonces G es lineal izquierda (y por tanto lineal), y no es lineal derecha.

Si $P = \{ S \to a, S \to Saaaa, S \to Saa \}$ entonces G es lineal derecha (y por tanto lineal), y no es lineal izquierda.

Si $P = \{ S \to a,\ S \to Sa \}$ entonces G es lineal derecha (y por tanto lineal), y no es lineal izquierda.

Si $P = \{ S \to bbc,\ S \to Bbbc,\ B \to bbcS \}$ entonces G es lineal, y no es ni lineal izquierda ni lineal derecha.

Si $P = \{ S \to aab,\ S \to aaSb \}$ entonces G es lineal, y no es ni lineal izquierda ni lineal derecha. □

Definición 2.67: *Regla épsilon*
Dada una gramática $G = (N, T, P, S)$ diremos que una regla que pertenece a P es una regla épsilon sii es de la forma $A \to \varepsilon$ con $A \in N$.

Definición 2.68: *Gramática ε-regular (GεR)*
Diremos que una gramática es ε-regular sii es *ε-regular izquierda* o *ε-regular derecha*.

Diremos que una gramática es ε-regular izquierda *(GεRI)* sii cada una de sus reglas es regular izquierda, o regular terminal, o regla épsilon.

Diremos que una gramática es ε-regular derecha *(GεRD)* sii cada una de sus reglas es regular derecha, o regular terminal, o regla épsilon.

Definición 2.69: *Gramática ε-contexto libre (GεCL)*
Diremos que una gramática es ε-contexto libre sii cada una de sus reglas es de contexto libre, o regla épsilon.

Ejemplo 2.70:
Sea una gramática $G = (N, T, P, S)$ con $N = \{S, A, B\}$ y $T = \{a, b, c\}$:

Si $P = \{ S \to a,\ S \to aS,\ A \to \varepsilon \}$ entonces G es ε-regular izquierda, y por tanto ε-regular.

Si $P = \{ S \to b,\ B \to Bc,\ B \to \varepsilon \}$ entonces G es ε-regular derecha, y por tanto ε-regular.

Si $P = \{ S \to c,\ S \to aA,\ S \to bB,\ A \to aA,\ A \to a,\ B \to bB,\ S \to \varepsilon \}$ entonces G es ε-regular izquierda, y por tanto ε-regular.

Si $P = \{ S \to c,\ A \to a,\ B \to b,\ S \to \varepsilon \}$ entonces G es ε-regular izquierda y ε-regular derecha, y por tanto ε-regular.

Si $P = \{ S \to a, S \to SS, S \to \varepsilon \}$ entonces G es ε-contexto libre y no es ε-regular.

Si $P = \{ A \to ccc, A \to \varepsilon \}$ entonces G es ε-contexto libre y no es ε-regular.

Si $P = \{ S \to A, A \to a, B \to \varepsilon \}$ entonces G es ε-contexto libre y no es ε-regular.

Si $P = \{ S \to aaScc, S \to b, S \to \varepsilon \}$ entonces G es ε-contexto libre y no es ε-regular.

Si $P = \{ S \to a, S \to aS, B \to Bc, A \to \varepsilon \}$ entonces G es ε-contexto libre y no es ε-regular. □

Proposición 2.71:
Sea G una gramática ε-regular izquierda, entonces existe G' gramática regular izquierda tal que $L(G') = L(G) - \{\varepsilon\}$.

Proposición 2.72:
Sea G una gramática regular izquierda, entonces existe G' gramática ε-regular izquierda tal que $L(G') = L(G) \cup \{\varepsilon\}$.

Proposición 2.73:
Sea G una gramática ε-contexto libre, entonces existe G' gramática de contexto libre tal que $L(G') = L(G) - \{\varepsilon\}$.

Proposición 2.74:
Sea G una gramática de contexto libre, entonces existe G' gramática ε-contexto libre tal que $L(G') = L(G) \cup \{\varepsilon\}$.

2.7 Notación

Aunque el concepto de gramática ya ha sido definido, a continuación enunciamos el sistema de notación completo que usaremos en el ámbito de las gramáticas:

$T \equiv$ *Alfabeto Terminal*.
Notación de elementos de T:

- Letras minúsculas del principio del alfabeto (subíndices permitidos): a, b, c, ...
- Operadores: $+$, $-$, $*$, $/$
- Caracteres especiales: $@$, $($, $)$, $[$, $]$, $\&$, $\#$
- Dígitos : $0, 1, \ldots, 9$
- Palabras en minúsculas subrayadas: $\underline{if}$, $\underline{then}$, ...

$N \equiv$ *Alfabeto No Terminal*.
Notación de elementos de N:

- Letras mayúsculas del principio del alfabeto (subíndices permitidos): A, B, C, ...
- Palabras en mayúsculas subrayadas: $\underline{DÍGITO}$, $\underline{EXPRESIÓN}$, ...
- Palabras entre paréntesis angulares: $<dígito>$, $<EXPRESIÓN>$, ...

Se ha de cumplir que $N \cap T = \varnothing$.

$V = N \cup T$.
Notaremos los elementos de V con letras mayúsculas del final del alfabeto ($\ldots, X, Y, Z$).

Notaremos los elementos de T^* con letras minúsculas del final del alfabeto (t, u, v, ...).
Notaremos los elementos de V^* con letras griegas minúsculas del principio del alfabeto (subíndices permitidos), a excepción de ε (α, β, γ, ...).

$\varepsilon \equiv$ *Cadena vacía*.

$S \equiv$ *Axioma*, con $S \in N$ (subíndices permitidos).

Usaremos el símbolo " | " para abreviar la notación de las reglas, de forma que:

$$A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \quad \equiv \quad \begin{cases} A \to \alpha_1 \\ A \to \alpha_2 \\ \quad \ldots \\ A \to \alpha_n \end{cases}$$

2.8 Clasificación de Lenguajes

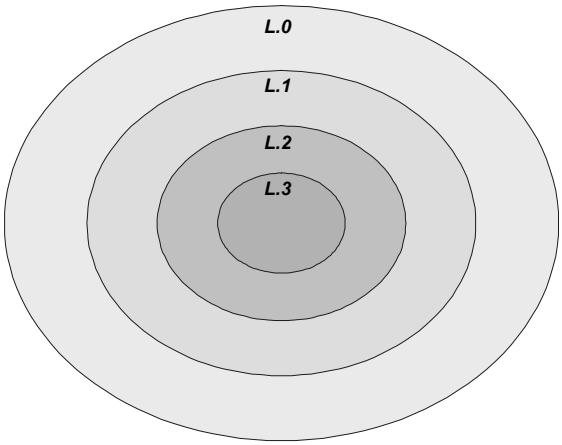
Definición 2.75: *Lenguaje de tipo i (L.i)*

Diremos que un lenguaje L es de tipo i, con $i \in \{0,1,2,3\}$, si existe una gramática G de tipo i tal que $L(G) = L - \{\varepsilon\}$.

Nota: También se le denomina por el nombre correspondiente al tipo i de la gramática.

La Jerarquía de Chomsky de los lenguajes establece las siguientes inclusiones:

$$L.3 \subset L.2 \subset L.1 \subset L.0$$



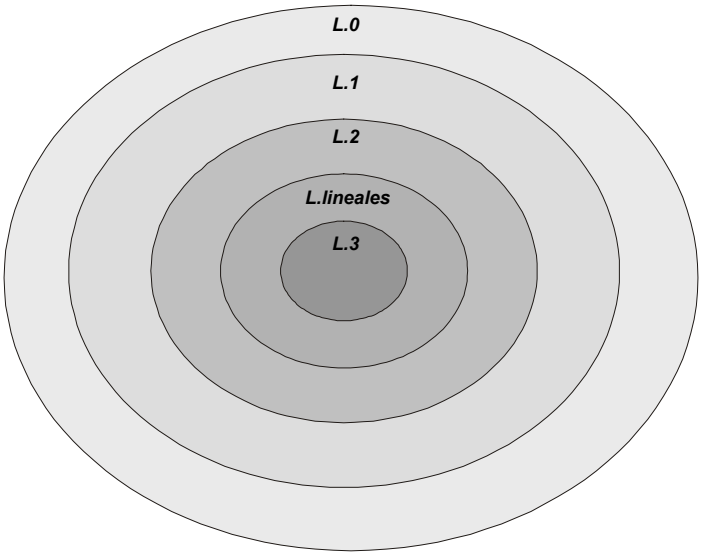
Definición 2.76: *Lenguaje lineal (L.lineales)*

Diremos que un lenguaje L es lineal si existe una gramática lineal G tal que $L(G) = L - \{\varepsilon\}$.

Nota: Si G es una gramática lineal derecha o lineal izquierda entonces $L(G)$ es regular.

Si tenemos en cuenta los lenguajes lineales además de los asociados a la Jerarquía de Chomsky, tenemos que:

$$L.3 \subset L.lineales \subset L.2 \subset L.1 \subset L.0$$



Como se deduce de las definiciones anteriores, un mismo lenguaje puede ser generado por gramáticas de distinto tipo (véanse ejemplos **2.62**, **2.66** y **2.68**).

Ejemplo 2.77:

Sea $\Sigma = \{ a, b, c \}$.

$L = \{ w \in \Sigma^* \mid w = a^n \text{ con } n \geq 0 \} \in L.3$.

$L = \{ w \in \Sigma^* \mid w = a^n b^n \text{ con } n \geq 0 \} \in (L.lineales - L.3)$.

$L = \{ w \in \Sigma^* \mid w = a^n b^n c a^m b^m \text{ con } n \geq 0, \ m \geq 0 \} \in (L.2 - L.lineales)$.

$L = \{ w \in \Sigma^* \mid w = a^n b^n c^n \text{ con } n \geq 0 \} \in (L.1 - L.2)$.

$L = \{ w \in \Sigma^* \mid w \notin L(g(f(w))) \} \in (L.0 - L.1)$,

donde:

$f : \Sigma^* \rightarrow \mathbb{N}$

$g : \mathbb{N} \rightarrow \{ G \text{ sensibles al contexto} \mid T_G = \Sigma \}$

son biyecciones (que se pueden calcular). □

2.9 Preguntas básicas sobre los Lenguajes

Dadas dos gramáticas G_1 y G_2 , de un mismo tipo, se plantean las siguientes preguntas:

Equivalencia: ¿ $G_1 \equiv G_2$?
Inclusión: ¿ $L(G_1) \subseteq L(G_2)$?
Pertenencia: ¿ $x \in L(G_1)$?
Vacuidad: ¿ $L(G_1) = \varnothing$?
Finitud: ¿ $\| L(G_1) \| \in \mathsf{N}$?
Regularidad: ¿ $L(G_1) \in L.3$?

En la siguiente tabla mostramos si existe un algoritmo conclusivo para responder a dichas preguntas según el tipo de las gramáticas:

	Estructura de Frase	Sensible al contexto	Contexto Libre	Lineales	Regulares
Equivalencia	No	No	No	No	Sí
Inclusión	No	No	No	No	Sí
Pertenencia	No	Sí	Sí	Sí	Sí
Vacuidad	No	No	Sí	Sí	Sí
Finitud	No	No	Sí	Sí	Sí
Regularidad	No	No	No	No	Trivial

2.10 Operaciones sobre Lenguajes

Al ser los lenguajes conjuntos de cadenas las operaciones genéricas para conjuntos también son aplicables a los lenguajes, así pues suponemos conocidas las operaciones de *unión*, *intersección*, *diferencia* y *complemento* ($\overline{L} = \Sigma^* - L$).

Además de dichas operaciones genéricas, definiremos otras que son específicas para los lenguajes, y que se basan en la primera que veremos, la concatenación de lenguajes, la cual a su vez se basa en la concatenación de cadenas.

Definición 2.78: *Concatenación de lenguajes*

Si L_1 , L_2 son lenguajes sobre Σ, definimos L_1 concatenado con L_2 , notado $L_1 \cdot L_2$ o simplemente L_1L_2 , como:

$$L_1L_2 = \{ w \in \Sigma^* \mid w = x\,y\,, \text{ con } x \in L_1 \text{ e } y \in L_2 \} .$$

Notas: $L\varnothing = \varnothing L = \varnothing \quad \forall L \subseteq \Sigma^*$

$L\{\varepsilon\} = \{\varepsilon\}L = L \quad \forall L \subseteq \Sigma^*$

$L_1 (L_2L_3) = (L_1L_2)L_3 \quad \forall L_1 , L_2 , L_3 \subseteq \Sigma^*$

$L_1 (L_2 \cup L_3) = L_1L_2 \cup L_1L_3 \quad \forall L_1 , L_2 , L_3 \subseteq \Sigma^*$

$(L_1 \cup L_2)L_3 = L_1L_3 \cup L_2L_3 \quad \forall L_1 , L_2 , L_3 \subseteq \Sigma^*$

Ejemplo 2.79:

Sea $\Sigma = \{ 0, 1 \}$ y $L_1 , L_2 \subseteq \Sigma^*$

Si $L_1 = \{ 010, 00 \}$ y $L_2 = \{ 111, 10 \}$ entonces

$L_1L_2 = \{ 010111, 01010, 00111, 0010 \}$

$L_2L_1 = \{ 111010, 11100, 10010, 1000 \}$

Si $L_1 = \{ w \in \Sigma^* \mid w=01^n \text{ con } n \geq 0 \}$ y $L_2 = \{ w \in \Sigma^* \mid |w|_0 = 2n+1 \text{ con } n \geq 0 \}$ entonces

$L_1L_2 = \{ w \in \Sigma^* \mid w(1)=0 \wedge |w|_0 = 2n \text{ con } n \geq 1 \}$

$L_2L_1 = \{ w \in \Sigma^* \mid |w|_0 = 2n \text{ con } n \geq 1 \}$ □

Definición 2.80: *Potencia de un lenguaje (L^n)*

Sea L un lenguaje sobre Σ, definimos la *n*-ésima potencia de L , notado L^n , como:

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

Ejemplo 2.81:

Sea $\Sigma = \{ a, b \}$ y $L \subseteq \Sigma^*$ con $L = \{ aa , b \}$

$L^3 = LL^2 = LLL^1 = LLLL^0 = LLL\{\varepsilon\} = LLL = \{ aa , b \} \{ aa , b \} \{ aa , b \} =$

$\{ aaaaaa, aaaab, aabaa, aabb, baaaa, baab, bbaa, bbb \}$ □

Definición 2.82: *Cierre (o Estrella de Kleene)*

Dado $L \subseteq \Sigma^*$ definimos el cierre o estrella de Kleene del lenguaje L, notado L^*, como:

$$L^* = L^{\bullet e}$$

donde $L^{\bullet e}$ es el cierre amplio del conjunto L para la operación concatenación de cadenas, en el monoide $(\Sigma^*, \cdot)$.

Notas: $L^* = \{ w \in \Sigma^* \mid w = w_1 w_2 ... w_k \text{ con } w_1, w_2, ..., w_k \in L \text{ y } k \geq 0 \}$.

$L^* = \bigcup_{n \geq 0} L^n$

Según la definición tenemos que $\varepsilon \in L^* \quad \forall L \subseteq \Sigma^*$.

Ejemplo 2.83:

$\varnothing^* = \{\varepsilon\}$

$\{\varepsilon\}^* = \{\varepsilon\}$

Si $L = \{ 01, 1, 100 \}$ tenemos que:

$110001110011 \in L^*$

$100001 \notin L^*$

$\varepsilon \in L^*$

Si $L = \{ \varepsilon, a \}$ tenemos que:

$L^* = \{ \varepsilon, a, aa, aaa, ... \}$

Sea $\Sigma = \{ a, b \}$ y $L \subseteq \Sigma^*$ con $L = \{ bbb \}$

$L^* = \{ w \in \{ a, b \}^* \mid w = (bbb)^n \text{ con } n \geq 0 \}$ □

Definición 2.84: *Cierre estricto*

Dado $L \subseteq \Sigma^*$ definimos el cierre estricto del lenguaje L, notado L^+, como:

$$L^+ = L^{\bullet}$$

donde $L^{\bullet}$ es el cierre estricto del conjunto L para la operación concatenación de cadenas.

Notas: $L^+ = \{ w \in \Sigma^* \mid w = w_1 w_2 ... w_k \text{ con } w_1, w_2, ..., w_k \in L \text{ y } k \geq 1 \}$.

$L^+ = \bigcup_{n \geq 1} L^n$

$L^+ = LL^*$.

Si $\varepsilon \in L$ entonces $L^+ = L^*$.

Si $\varepsilon \notin L$ entonces $L^+ = L^* - \{\varepsilon\}$.

Ejemplo 2.85:

$\varnothing^+ = \varnothing$

$\{\varepsilon\}^+ = \{\varepsilon\}$

Si $L = \{ 01 , 1 , 100 \}$ tenemos que:

$110001110011 \in L^+$

$100001 \notin L^+$

$\varepsilon \notin L^+$

Si $L = \{ \varepsilon , a \}$ tenemos que:

$L^+ = \{ \varepsilon , a, aa, aaa, ... \}$

Sea $\Sigma = \{ a , b \}$ y $L \subseteq \Sigma^*$ con $L = \{ bbb \}$

$L^+ = \{ w \in \{ a, b \}^* \mid w = (bbb)^n \text{ con } n \geq 1 \}$ □

Definición 2.86: *Inverso de un lenguaje*

Sea $L \subseteq \Sigma^*$, definimos el inverso de L, notado L^R, como $L^R = \{ x^R \mid x \in L \}$.

Ejemplo 2.87:

Sea $\Sigma = \{ a, b \}$ y $L \subseteq \Sigma^*$ con $L = \{ w \in \{ a, b \}^* \mid w = a^n b^n \text{ con } n \geq 0 \}$

$L^R = \{ w \in \{ a, b \}^* \mid w = b^n a^n \text{ con } n \geq 0 \}$ □

2.11 Cierre de los Tipos de Lenguaje

En la siguiente tabla se muestra si cada tipo de lenguaje es o no cerrado para cada una de las operaciones vistas anteriormente.

	Estructura de frase	Sensible al contexto	Contexto libre	Lineales	Regulares
Unión	Sí	Sí	Sí	Sí	Sí
Intersección	Sí	Sí	No	No	Sí
Intersección con $L.3$	Sí	Sí	Sí	Sí	Sí
Complemento	No	Sí	No	No	Sí
Concatenación	Sí	Sí	Sí	No	Sí
Potencia	Sí	Sí	Sí	No	Sí
Cierre	Sí	Sí	Sí	No	Sí
Cierre estricto	Sí	Sí	Sí	No	Sí
Inverso	Sí	Sí	Sí	Sí	Sí

El hecho de que $L.0$ no sea cerrado para el complemento, junto con los tipos de lenguajes ya definidos, nos lleva, a modo de resumen, a la siguiente cadena de inclusiones propias:

$$L.finitos \subset L.3 \subset L.lineales \subset L.2 \subset L.1 \subset L.0 \subset L.REP \subset 2^{\Sigma^*}$$

3
Expresiones Regulares

Las *expresiones regulares* (*ER*) son un método de representación de lenguajes. Aunque su potencia expresiva es limitada, haciendo que sólo los lenguajes regulares puedan representarse con ellas, tienen la virtud de una gran sencillez en su formulación.

3.1 Definiciones

Definición 3.1: *Expresión regular (ER)*

Dado un alfabeto Σ definimos las expresiones regulares sobre Σ como las cadenas sobre el alfabeto $\Sigma' = \Sigma \cup \{ \ ) \ , \ (\ , \ \varnothing \ , \ + \ , \ ^* \ \}$ que pueden construirse con las siguientes reglas:

a) $\varnothing$ es *ER*.
b) Si $a \in \Sigma$ entonces a es *ER*.
c) Si α , β son *ER* entonces $(\alpha\beta)$ es *ER*.
d) Si α , β son *ER* entonces $(\alpha+\beta)$ es *ER*.
e) Si α es *ER* entonces α^* es *ER*.
f) Ninguna cadena sobre Σ' que no se pueda construir con las reglas anteriores es *ER*.

Ejemplo 3.2:

Sea $\Sigma = \{ a , b \}$

Nuestro Σ' será $\Sigma' = \{ a , b ,) , (, \varnothing , + , ^* \}$

Son expresiones regulares sobre Σ:

$\varnothing$, a , b , a^*, $(\varnothing\varnothing)$, $(\varnothing\ a)$, (aa) , (ba) , $(ba)^*$, $((a\varnothing)(bb))$, $((ab)+(\varnothing\ b))$, ...

□

Definición 3.3: *Aplicación L*

La aplicación L establece una relación formal entre las expresiones regulares y los lenguajes que éstas representan, definiéndose como sigue:

$$L: ER \rightarrow 2^{\Sigma^*}$$
$$er \rightarrow L(er)$$

a) $L(\varnothing) = \varnothing$

b) $L(a) = \{a\} \quad \forall a \in \Sigma$

c) Si $\alpha, \beta \in ER$ entonces $L((\alpha\beta)) = L(\alpha)L(\beta)$

d) Si $\alpha, \beta \in ER$ entonces $L((\alpha+\beta)) = L(\alpha) \cup L(\beta)$

e) Si $\alpha \in ER$ entonces $L(\alpha^*) = L(\alpha)^*$

Notación habitual

$\varepsilon \equiv \varnothing^*$

$\alpha^* = \varepsilon + \alpha + \alpha\alpha + \alpha\alpha\alpha + \ldots = \varepsilon + \alpha + \alpha^2 + \ldots + \alpha^i + \ldots$

$L(\alpha^*) = \{\varepsilon\} \cup L(\alpha) \cup L(\alpha)^2 \cup \ldots \cup L(\alpha)^i \cup \ldots$

Ejemplo 3.4:

Vamos a ver el significado de la siguiente expresión regular: $((b+a)^*b)$

$$L(((b+a)^*b)) = L((b+a)^*)L(b) = L((b+a)^*)\{b\} = L((b+a))^*\{b\} =$$
$$\uparrow \text{por c)} \qquad \uparrow \text{por b)} \qquad \uparrow \text{por e)} \qquad \uparrow \text{por d)}$$

$$(L(b) \cup L(a))^*\{b\} = (\{b\} \cup \{a\})^*\{b\} = \{b,a\}^*\{b\}$$
$$\uparrow \text{por b)} \qquad \uparrow \text{por la unión}$$

Nota: $\{b,a\}^*\{b\} = \{ w \in \{a,b\}^* \mid w \text{ termina en } b \}$ □

Proposición 3.5:

Todo lenguaje que puede ser representado por una expresión regular puede ser representado por infinitas expresiones regulares.

Demostración:

Sea α una expresión regular sobre Σ y $L(\alpha)$ el lenguaje que representa.

Entonces todas las expresiones regulares distintas entre sí (distintas cadenas sobre el alfabeto $\Sigma' = \Sigma \cup \{ (,), \varnothing, +, ^* \}$) de la forma

α , $(\alpha+\alpha)$, $((\alpha+\alpha)+\alpha)$, $(((\alpha+\alpha)+\alpha)+\alpha)$, ... cumplen que
$L(\alpha) = L((\alpha+\alpha)) = L(((\alpha+\alpha)+\alpha)) = L ((((\alpha+\alpha)+\alpha)+\alpha)) = ...$ □

Convenio de eliminación de paréntesis de las expresiones regulares

a) Podemos sustituir los pares de paréntesis por corchetes para una mayor legibilidad.
b) Eliminamos, si los hubiera, los paréntesis que son primer y último símbolo de la expresión regular (sólo aplicable una vez).
c) Jerarquización de las prioridades de las operaciones, de mayor a menor:
cierre > concatenación > unión
d) En caso de igualdad de prioridades se asociaría por la izquierda, aunque es indiferente puesto que la unión y la concatenación son asociativas.

Ejemplo 3.6:
$(\alpha+\beta) = \alpha+\beta$
$(\alpha\beta)^* \neq \alpha\beta^*$
$(\alpha\beta)+\gamma = \alpha\beta+\gamma$
$\alpha(\beta\gamma) = (\alpha\beta)\gamma = \alpha\beta\gamma$
$\alpha+(\beta+\gamma) = (\alpha+\beta)+\gamma = \alpha+\beta+\gamma$ □

Nota: De aquí en adelante, una expresión regular simplemente representará un lenguaje, y no nos fijaremos en ella como una cadena sobre un alfabeto ampliado.
Por ejemplo: $a^*b^* = (a^*b^*) = \{a\}^*\{b\}^* = a^*b^* + \varnothing$

Definición 3.7: *Conjunto de lenguajes representados por expresiones regulares* *(L(ER))*

$$L(ER) = Rg(L)$$

donde L es la aplicación definida anteriormente.

Debido a que $L(ER)$ se define en función de L , y esta última está relacionada con la definición de *ER*, el conjunto de los lenguajes representados por expresiones regulares cumple las siguientes condiciones:

a) $\varnothing \in L(ER)$

b) $\{a\} \in L(ER) \ \forall a \in \Sigma$

c) Si $L_1, L_2 \in L(ER) \Rightarrow L_1 \cup L_2 , L_1 L_2 , L_1^* \in L(ER)$

d) $\forall C$ que verifique las anteriores condiciones tenemos que $L(ER) \subseteq C$

Proposición 3.8:
$L(ER) \subset L.REP$

Ejemplo 3.9:
Sea $L = \{ w \in \{0,1\}^* \mid w = 0^n 1^n$, con $n \geq 1 \}$
$L \notin L(ER)$ (ver ejemplo **5.5**)

Nota: Posteriormente veremos (en el corolario **4.48**), como ya hemos comentado, que $L(ER) = L.3$. □

3.2 Propiedades de las Expresiones Regulares

Las expresiones regulares cumplen un buen número de propiedades, de las cuales veremos las veinte más importantes.

Proposición 3.10:
Si α , β y γ son expresiones regulares entonces se cumple:

1) $\alpha + \varnothing = \alpha$
2) $\alpha + \alpha = \alpha$
3) $\alpha + \beta = \beta + \alpha$
4) $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$
5) $\alpha(\beta\gamma) = (\alpha\beta)\gamma$
6) $\alpha\varepsilon = \varepsilon\alpha = \alpha$
7) $(\alpha + \beta)\gamma = \alpha\gamma + \beta\gamma$
8) $\gamma(\alpha + \beta) = \gamma\alpha + \gamma\beta$
9) $\alpha^* \alpha^* = \alpha^*$
10) $(\alpha^*)^* = \alpha^*$
11) $\alpha\alpha^* = \alpha^*\alpha$

12) $\alpha^* = \varepsilon + \alpha + \alpha^2 + \alpha^3 + ... + \alpha^k + \alpha^{k+1}\alpha^* \qquad \forall k \geq 0$

13) $\alpha^* = \varepsilon + \alpha\alpha^*$

14) $(\alpha^* + \beta^*)^* = (\alpha + \beta)^*$

15) $(\alpha^*\beta^*)^* = (\alpha + \beta)^*$

16) $(\alpha\beta)^*\alpha = \alpha(\beta\alpha)^*$

17) $(\alpha^*\beta)^*\alpha^* = (\alpha + \beta)^*$

18) $\alpha^*(\beta\alpha^*)^* = (\alpha + \beta)^*$

19) $(\alpha^*\beta)^* = (\alpha + \beta)^*\beta + \varepsilon$

20) $f(\alpha_1, \alpha_2, ..., \alpha_n) \subseteq (\alpha_1 + \alpha_2 + ... + \alpha_n)^*$ donde las imágenes obtenidas por medio de f son combinaciones de las expresiones regulares α_i utilizando sólo operaciones regulares: unión (+), concatenación (·) y estrella de Kleene (*).

Demostración:

Para hacer las demostraciones haremos uso de la aplicación L, de las propiedades de las operaciones de conjuntos y de lenguajes, y de las propiedades de las *ER* que ya hayan sido demostradas.

1) $\alpha + \varnothing = \alpha$

$$L((\alpha+\varnothing)) = L(\alpha) \cup L(\varnothing) = L(\alpha) \cup \varnothing = L(\alpha)$$

$\uparrow$ *por d)* $\qquad$ $\uparrow$ *por a)*

□

2) $\alpha + \alpha = \alpha$

$$L((\alpha+\alpha)) = L(\alpha) \cup L(\alpha) = L(\alpha)$$

$\uparrow$ *por d)*

□

3) $\alpha + \beta = \beta + \alpha$

$$L((\alpha + \beta)) = L(\alpha) \cup L(\beta) = L(\beta) \cup L(\alpha) = L((\beta + \alpha))$$

$\uparrow$ *por d)* $\qquad$ $\uparrow$ *por d)*

□

4) $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$

$$L((\alpha + (\beta + \gamma))) = L(\alpha) \cup L((\beta + \gamma)) = L(\alpha) \cup (L(\beta) \cup L(\gamma)) =$$

$\uparrow$ *por d)* $\qquad$ $\uparrow$ *por d)*

$L(\alpha) \cup L(\beta) \cup L(\gamma) = (L(\alpha) \cup L(\beta)) \cup L(\gamma) = L((\alpha + \beta)) \cup L(\gamma) =$

por d) *por d)*

$L(((\alpha + \beta) + \gamma)$ □

5) $\alpha(\beta\gamma) = (\alpha\beta)\gamma$

$L((\alpha(\beta\gamma))) = L(\alpha)\, L((\beta\gamma)) = L(\alpha)(L(\beta)L(\gamma)) = L(\alpha)L(\beta)L(\gamma) = (L(\alpha)L(\beta))L(\gamma) =$

por c) *por c)* *por c)*

$L((\alpha\beta))L(\gamma) = L(((\alpha\beta)\gamma))$

por c)

□

6) $\alpha\varepsilon = \varepsilon\alpha = \alpha$

Como $\varepsilon = \varnothing^*$ tenemos:

$L((\alpha\varnothing^*)) = L(\alpha)L(\varnothing^*) = L(\alpha)L(\varnothing)^* = L(\alpha)\varnothing^* = L(\alpha)\,\{\varepsilon\} = L(\alpha) =$

por c) *por e)* *por a)*

$\{\varepsilon\}L(\alpha) = \varnothing^* L(\alpha) = L(\varnothing)^*\, L(\alpha) = L(\varnothing^*)L(\alpha) = L((\varnothing^*\, \alpha))$

por a) *por e)* *por c)*

□

7) $(\alpha + \beta)\gamma = \alpha\gamma + \beta\gamma$

$L(((\alpha + \beta)\gamma)) = L((\alpha + \beta))L(\gamma) = (L(\alpha) \cup L(\beta))L(\gamma) = L(\alpha)L(\gamma) \cup L(\beta)L(\gamma) =$

por c) *por d)* *por c)*

$L((\alpha\gamma)) \cup L((\beta\gamma)) = L((\alpha\gamma + \beta\gamma))$

por d)

□

8) $\gamma(\alpha + \beta) = \gamma\alpha + \gamma\beta$

$L((\gamma\,(\alpha + \beta))) = L(\gamma)L((\alpha + \beta)) = L(\gamma)(L(\alpha) \cup L(\beta)) = L(\gamma)L(\alpha) \cup L(\gamma)L(\beta) =$

por c) *por d)* *por c)*

$L((\gamma\alpha)) \cup L((\gamma\beta)) = L((\gamma\alpha + \gamma\beta))$

↑

por d)

□

9) $\alpha^* \alpha^* = \alpha^*$

$\alpha^* = \varepsilon + \alpha + \alpha^2 + ... + \alpha^i + ...$

$\alpha^*\alpha^* = (\varepsilon + \alpha + \alpha^2 + ... + \alpha^i + ...)\ \alpha^* = \varepsilon\alpha^* + \alpha\alpha^* + \alpha^2\alpha^* + ... + \alpha^i\alpha^* + ...$

↑

por 7)

Vemos cada sumando:

$\varepsilon\alpha^* = \alpha^*$

$\alpha\alpha^* = \alpha(\varepsilon + \alpha + \alpha^2 + ... + \alpha^i + ...) = \alpha + \alpha^2 + \alpha^3 + ... \subseteq \alpha^*$

↑

por 8)

$\alpha^2\alpha^* = \alpha^2(\varepsilon + \alpha + \alpha^2 + ... + \alpha^i + ...) = \alpha^2 + \alpha^3 + ... \subseteq \alpha^*$

↑

por 8)

⋮

luego tenemos que $\forall i \geq 1,\ \alpha^i\alpha^* \subseteq \alpha^*$, y por absorción:

$\alpha^* + \alpha\alpha^* + \alpha^2\alpha^* + ... + \alpha^i\alpha + ... = \alpha^*$

□

10) $(\alpha^*)^* = \alpha^*$

$(\alpha^*)^* = (\alpha^*)^0 + (\alpha^*)^1 + (\alpha^*)^2 + (\alpha^*)^3 + ...$

Veamos cada sumando:

$(\alpha^*)^0 = \{\varepsilon\} \subseteq \alpha^*$

$(\alpha^*)^1 = \alpha^*$

$(\alpha^*)^2 = \alpha^*\alpha^* = \alpha^*$

↑

por 9)

$(\alpha^*)^3 = (\alpha^*)^2\alpha^* = (\alpha^*\alpha^*)\alpha^* = \alpha^*\alpha^* = \alpha^*$

↑ ↑

por 9) por 9)

⋮

$(\alpha^*)^i = \alpha^* \quad \forall i \geq 1$

Y por absorción e idempotencia tenemos que $(\alpha^*)^* = \alpha^*$

□

11) $\alpha\alpha^* = \alpha^*\alpha$

$$\left.\begin{array}{l} \alpha\alpha^* = \alpha(\varepsilon + \alpha + \alpha^2 + \ldots + \alpha^i + \ldots) = \alpha + \alpha^2 + \alpha^3 + \ldots + \alpha^{i+1} + \ldots \\ \alpha^*\alpha = (\varepsilon + \alpha + \alpha^2 + \ldots + \alpha^i + \ldots)\alpha = \alpha + \alpha^2 + \alpha^3 + \ldots + \alpha^{i+1} + \ldots \end{array}\right\} \Rightarrow$$

$\alpha\alpha^* = \alpha^*\alpha$ □

12) $\alpha^* = \varepsilon + \alpha + \alpha^2 + \alpha^3 + \ldots + \alpha^k + \alpha^{k+1}\alpha^* \qquad \forall k \geq 0$

$\alpha^{k+1}\alpha^* = \alpha^{k+1}(\varepsilon + \alpha + \alpha^2 + \ldots + \alpha^i + \ldots) =$

$\alpha^{k+1} + \alpha^{k+2} + \alpha^{k+3} + \ldots + \alpha^{i+k+1} + \ldots \Rightarrow$

$\varepsilon + \alpha + \alpha^2 + \alpha^3 + \ldots + \alpha^k + \alpha^{k+1}\alpha^* =$

$\varepsilon + \alpha + \alpha^2 + \alpha^3 + \ldots + \alpha^k + \alpha^{k+1} + \alpha^{k+2} + \alpha^{k+3} + \ldots + \alpha^{i+k+1} + \ldots = \alpha^*$

□

13) $\alpha^* = \varepsilon + \alpha\alpha^*$

$\alpha^* = \varepsilon + \alpha\alpha^*$

↑

por 12) para k=0

□

14) $(\alpha^* + \beta^*)^* = (\alpha + \beta)^*$

$\alpha + \beta \subseteq \alpha^* + \beta^* \Rightarrow (\alpha + \beta)^* \subseteq (\alpha^* + \beta^*)^*$

$$\left.\begin{array}{l} \alpha^* \subseteq (\alpha + \beta)^* \\ \alpha^* \subseteq (\alpha + \beta)^* \end{array}\right\} \Rightarrow \left.\begin{array}{l} \alpha^* + \beta^* \subseteq (\alpha + \beta)^* \\ ((\alpha + \beta)^*)^* = (\alpha + \beta)^* \end{array}\right\} \Rightarrow (\alpha^* + \beta^*)^* \subseteq (\alpha + \beta)^*$$

↑

por 10)

$$\left.\begin{array}{l} (\alpha + \beta)^* \subseteq (\alpha^* + \beta^*)^* \\ (\alpha^* + \beta^*)^* \subseteq (\alpha + \beta)^* \end{array}\right\} \Rightarrow (\alpha^* + \beta^*)^* = (\alpha + \beta)^*$$

□

15) $(\alpha^*\beta^*)^* = (\alpha + \beta)^*$

$$\left.\begin{array}{l} \alpha \subseteq \alpha^*\beta^* \\ \alpha_2 \subseteq \alpha^*\beta^* \end{array}\right\} \Rightarrow \alpha + \beta \subseteq \alpha^*\beta^* \Rightarrow (\alpha + \beta)^* \subseteq (\alpha^*\beta^*)^*$$

$$\left.\begin{array}{l}\alpha^* \subseteq (\alpha+\beta)^* \\ \alpha^*\beta \subseteq (\alpha+\beta)^* \\ \alpha^*\beta^2 \subseteq (\alpha+\beta)^* \\ \quad\vdots\end{array}\right\} \Rightarrow \alpha^*\beta^* \subseteq (\alpha+\beta)^* \underset{\text{por 10)}}{\Rightarrow} (\alpha^*\beta^*)^* \subseteq (\alpha+\beta)^*$$

$$\left.\begin{array}{l}(\alpha+\beta)^* \subseteq (\alpha^*\beta^*)^* \\ (\alpha^*\beta^*)^* \subseteq (\alpha+\beta)^*\end{array}\right\} \Rightarrow (\alpha^*\beta^*)^* = (\alpha+\beta)^* \qquad \square$$

16) $(\alpha\beta)^*\alpha = \alpha(\beta\alpha)^*$

$$(\alpha\beta)^*\alpha = [(\alpha\beta)^0 + (\alpha\beta)^1 + (\alpha\beta)^2 + \ldots + (\alpha\beta)^k + \ldots]\,\alpha =$$

$$[\varepsilon + (\alpha\beta) + (\alpha\beta)(\alpha\beta) + \ldots + \underbrace{(\alpha\beta)\ldots(\alpha\beta)}_{\text{k-veces}} + \ldots]\,\alpha = \quad \text{(por 7)}$$

$$\varepsilon\alpha + (\alpha\beta)\alpha + (\alpha\beta)(\alpha\beta)\alpha + \ldots + \underbrace{(\alpha\beta)\ldots(\alpha\beta)}_{\text{k-veces}}\alpha + \ldots = \quad \text{(por 5) y 6)}$$

$$\alpha + \alpha\beta\alpha + \alpha\beta\alpha\beta\alpha + \ldots + \underbrace{\alpha\beta\ldots\alpha\beta}_{\text{k-veces}}\alpha + \ldots = \quad \text{(por 5) y 6)}$$

$$\alpha\varepsilon + \alpha(\beta\alpha) + \alpha(\beta\alpha)(\beta\alpha) + \ldots + \alpha\underbrace{(\alpha\beta)\ldots(\alpha\beta)}_{\text{k-veces}} + \ldots = \quad \text{(por 8)}$$

$$\alpha[\varepsilon + (\beta\alpha) + (\beta\alpha)(\beta\alpha) + \ldots + \underbrace{(\beta\alpha)\ldots(\beta\alpha)}_{\text{k-veces}} + \ldots] =$$

$$\alpha[(\beta\alpha)^0 + (\beta\alpha)^1 + (\beta\alpha)^2 + \ldots + (\beta\alpha)^k + \ldots] = \alpha(\beta\alpha)^* \qquad \square$$

17) $(\alpha^*\beta)^*\alpha^* = (\alpha+\beta)^*$

Veamos en primer lugar $(\alpha+\beta)^* \subseteq (\alpha^*\beta)^*\alpha^*$:

Para ello demostraremos que cualquier cadena genérica del primer conjunto también pertenece al segundo conjunto, por lo que se cumple la inclusión.

Sea $w \in (\alpha + \beta)^*$, $w = w_1 w_2 \ldots w_n$ con $w_i \in \alpha+\beta$ e $i = 0, 1, 2, \ldots, n$
Sean $w_{i_1}, \ldots, w_{i_p}$ con $w_{i_k} \in \beta$ y $k = 0, \ldots, p$ con $p \leq n$ entonces:

$$w = \underbrace{w_1 \ldots w_{i_1-1} w_{i_1}}_{\in (\alpha^*\beta)} \underbrace{w_{i_1+1} \ldots w_{i_2-1} w_{i_2}}_{\in (\alpha^*\beta)} \underbrace{w_{i_2+1} \ldots w_{i_3-1} w_{i_3}}_{\in (\alpha^*\beta)} \ldots \underbrace{w_{i_{(p-1)}+1} \ldots w_{i_p-1} w_{i_p}}_{\in (\alpha^*\beta)} \underbrace{w_{i_p+1} \ldots w_n}_{\in \alpha^*}$$

Por lo tanto, $w \in (\alpha^*\beta)^*\alpha^* \Rightarrow (\alpha + \beta)^* \subseteq (\alpha^*\beta)^*\alpha^*$

$$\left.\begin{array}{l} (\alpha^*\beta)^*\,\alpha^* \subseteq (\alpha + \beta)^* \\ (\alpha + \beta)^* \subseteq (\alpha^*\beta)^*\alpha^* \end{array}\right\} \Rightarrow (\alpha^*\beta)^*\,\alpha^* = (\alpha + \beta)^*$$

Para $(\alpha^*\beta)^*\,\alpha^* \subseteq (\alpha + \beta)^*$ se realizaría por el mismo procedimiento (si bien en este caso también se podría argumentar que se cumple por la propia definición de $(\alpha + \beta)^*$). □

18) $\alpha^*(\beta\alpha^*)^* = (\alpha + \beta)^*$

$$(\alpha + \beta)^* = (\alpha^*\beta)^*\,\alpha^* = \alpha^*(\beta\alpha^*)^*$$

por 17) *por 16)* □

19) $(\alpha^*\beta)^* = (\alpha + \beta)^*\beta + \varepsilon$

$$(\alpha^*\beta)^* = \varepsilon + \alpha^*\beta\,(\alpha^*\beta)^* = \varepsilon + (\alpha^*\beta)^*\alpha^*\beta = \varepsilon + (\alpha + \beta)^*\beta = (\alpha + \beta)^*\beta + \varepsilon$$

por 13) *por 11)* *por 17)* *por 3)* □

20) $f(\alpha_1, \alpha_2, \ldots, \alpha_n) \subseteq (\alpha_1 + \alpha_2 + \ldots + \alpha_n)^*$ **donde las imágenes obtenidas por medio de** f **son combinaciones de las expresiones regulares** α_i **utilizando sólo operaciones regulares: unión (+), concatenación (·) y estrella de Kleene (*).**

Podríamos hacer un demostración genérica, pero para no extendernos sólo haremos una reflexión: se cumple ya que $(\alpha_1 + \alpha_2 + \ldots + \alpha_n)^*$ incluye precisamente todas las posibles combinaciones de las expresiones regulares α_i mediante $+$, $\cdot$, $*$, y en particular la combinación f. □

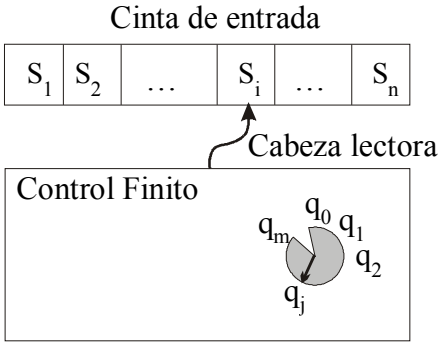
4
Autómatas Finitos

Los *autómatas finitos* (*AF*) no son sino otra forma de representar los lenguajes regulares, en este caso dispositivos reconocedores de los mismos. En este tema veremos los deterministas, posteriormente los no deterministas, para terminar comprobando la equivalencia entre ellos y cómo se obtiene uno mínimo.

4.1 Autómatas Finitos Deterministas (*AFD*)

Un *autómata finito determinista* (*AFD*) es un modelo matemático de un sistema discreto con un número finito de posibles estados internos, y donde ese estado interno en un momento dado es la única información disponible sobre las entradas pasadas. El comportamiento de un *AFD* vendrá por tanto determinado, exclusivamente, por el estado actual y las futuras entradas.

Antes de ver la definición formal de *AFD*, vamos a comentar su estructura y funcionamiento. Para ello nos lo imaginaremos como una máquina ideal, según muestra la figura siguiente.



Al ser un *AFD* un tipo de autómata, sabemos que es un dispositivo reconocedor de lenguajes, concretamente de lenguajes regulares. Así, la cadena a reconocer se le suministra al *AFD* por medio de una *cinta de entrada*, dividida en cuadrados, con un símbolo en cada cuadrado.

El *control finito* es una "caja negra" que contiene un número finito de *estados* y sólo puede estar en uno de ellos. Cada estado está marcado como *final* o como *no final*, y siempre hay uno denominado *estado inicial* (normalmente el q_0) que es en el que el control finito está al principio.

La *cabeza lectora*, que es controlada por el control finito e informa a éste del contenido del *cuadrado escrutado*, sólo puede moverse un único cuadrado cada vez hacia la derecha, y empieza a la izquierda de la cinta de entrada.

El control finito, dependiendo del símbolo escrutado por la cabeza lectora y del estado en el que se encuentra, pasa a otro estado y mueve la cabeza lectora. Cuando alcanza el final de la cinta el *AFD* se detiene. Si ha terminado en un estado final la cadena se considera *aceptada*, y *no aceptada* en caso contrario.

El *lenguaje aceptado* por el *AFD* es el conjunto de cadenas que acepta.

A continuación vamos a definir formalmente los conceptos anteriormente comentados, empezando por la definición de *AFD*.

Definición 4.1: *Autómata Finito Determinista (AFD)*

Un autómata finito determinista (*AFD*) es una quíntupla $M = (K , \Sigma , \delta , s , F)$ donde:

- K es un conjunto finito no vacío de estados;
- Σ es un alfabeto;
- s es el estado inicial $(s \in K)$;
- F es el conjunto de estados finales $(F \subseteq K)$;
- δ es una función de transición definida como $\delta: K \times \Sigma \to K$.

Nota: Si M está en el estado $q \in K$ y el símbolo leído en la cinta de entrada es $\sigma \in \Sigma$, entonces el siguiente estado al que pasa M es $\delta(q,\sigma) \in K$, que es único al ser δ una función.

Definición 4.2: *Configuración*

Dado $M = (K, \Sigma, \delta, s, F)$ *AFD*, una configuración de M es cualquier par $(q,w) \in K \times \Sigma^*$.

Nota: Representa el estado en el que está el *AFD* y la cadena que queda por leer.

Definición 4.3: *Configuración inicial*
Dado $M = (K, \Sigma, \delta, s, F)$ *AFD*, definimos configuración inicial de M como toda configuración de la forma (s,w) .

Definición 4.4: *Configuración terminal*
Dado $M = (K, \Sigma, \delta, s, F)$ *AFD*, definimos configuración terminal de M como toda configuración de la forma (q,ε) .

Definición 4.5: *Transitar directamente*
Sea M un *AFD*, y sean (q,w) y (q',w') configuraciones de M .
Diremos que (q,w) transita directamente a (q',w') , notado $(q,w) \vdash (q',w')$, sii:

$$\exists \sigma \in \Sigma \mid (w = \sigma w') \wedge (\delta (q,\sigma) = q')$$

Nota: Transitar directamente es una función, definida como:

$$\vdash : K \times \Sigma^{+} \rightarrow K \times \Sigma^{*}$$
$$(q,\sigma w') \rightarrow (\delta (q,\sigma),w')$$

Definición 4.6: *Transitar en n pasos*
Sea M un *AFD*, y sean (q,w) y (q',w') configuraciones de M .

Diremos que (q,w) transita en n pasos a (q',w') , notado $(q,w) \vdash^{n} (q',w')$, con $n>1$, sii:
$\exists\ C_1, C_2, \ldots, C_{n-1}$ configuraciones de M |

$$(q,w) \vdash C_1 , \ C_1 \vdash C_2 , \ \ldots , \ C_{n-1} \vdash (q',w')$$

Diremos que (q,w) transita en 1 paso a (q',w') , notado $(q,w) \vdash^{1} (q',w')$, sii:

$$(q,w) \vdash (q',w')$$

Diremos que (q,w) transita en 0 pasos a (q',w'), notado $(q,w) \vdash^{0} (q',w')$, sii:

$$(q,w) = (q',w')$$

Definición 4.7: *Transitar en al menos un paso*
Sea M un AFD, y sean (q,w) y (q',w') configuraciones de M.
Diremos que (q,w) transita en al menos un paso a (q',w'), notado $(q,w) \vdash^{+} (q',w')$, sii:

$$\exists n>0 \mid (q,w) \vdash^{n} (q',w')$$

Definición 4.8: *Transitar*
Sea M un AFD, y sean (q,w) y (q',w') configuraciones de M.
Diremos que (q,w) transita a (q',w'), notado $(q,w) \vdash^{*} (q',w')$, sii:

$$\exists n\geq 0 \mid (q,w) \vdash^{n} (q',w')$$

En resumen:

$\vdash$ es una relación binaria entre configuraciones (sobre $K \times \Sigma^{*}$)

$\vdash^{+}$ es su cierre transitivo

$\vdash^{*}$ es su cierre reflexivo y transitivo

Definición 4.9: *Computación, Longitud de una computación*
Sea M un AFD, definimos computación de M como cualquier secuencia finita de configuraciones $C_0, C_1, C_2, \ldots, C_n$ tal que $C_i \vdash C_{i+1}$ (siendo C_{i+1} la *configuración siguiente* de la C_i), donde $i = 0, \ldots, n-1$ con $n\geq 0$.
La notaremos $C_0 \vdash C_1 \vdash C_2 \vdash \ldots \vdash C_n$.
La longitud de dicha computación es n.

Notas: En un AFD, para toda configuración que no sea terminal, siempre existe una y sólo una configuración siguiente.
En un AFD, para toda configuración terminal no existe configuración siguiente.

Definición 4.10: *Computación bien iniciada / terminada / completa*
Computación bien iniciada es cualquier computación cuya primera configuración es inicial.
Computación terminada es cualquier computación cuya última configuración es terminal.
Una computación es completa si es una computación bien iniciada y terminada.

Definición 4.11: *Estado terminal*
Estado terminal es el estado de la última configuración de una computación terminada.

Definición 4.12: *Cadena aceptada*
Sea $M = (K, \Sigma, \delta, s, F)$ un *AFD* y $w \in \Sigma^*$.
Diremos que w es una cadena aceptada por M sii $\exists q \in F \mid (s,w) \vdash^* (q, \varepsilon)$.

Definición 4.13: *Lenguaje aceptado*
Sea M un *AFD*, definimos el lenguaje aceptado por M , notado $L(M)$, como:

$$L(M) = \{ w \in \Sigma^* \mid w \text{ es aceptada por } M \}$$

Nota: Las cadenas y los lenguajes aceptados también suelen denominarse *reconocidos*.

Definición 4.14: *Conjunto de lenguajes aceptados por AFD*
El conjunto de los lenguajes aceptados por *AFD*, notado $\mathcal{L}(AFD)$, se define como:

$$\mathcal{L}(AFD) = \{ L \mid \exists M \ AFD \text{ con } L(M) = L \}$$

Ejemplo 4.15:
Sea $M = (K , \Sigma , \delta , s , F)$ un *AFD* donde:
$K = \{ q_0 , q_1 \}$
$\Sigma = \{ 0 , 1 \}$

$\delta(q,\sigma)$	0	1
q_0	q_0	q_1
q_1	q_1	q_0

$s = q_0$
$F = \{ q_0 \}$

$L(M) = \{ w \in \{0,1\}^* \mid |w|_1 = 2n , \text{ con } n \geq 0 \}$

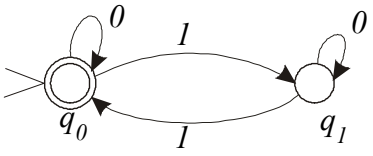
Veamos qué hace el autómata con la cadena *00110* :

$(q_0,00110) \vdash (q_0,0110) \vdash (q_0,110) \vdash (q_1,10) \vdash (q_0,0) \vdash (q_0,\varepsilon) \Rightarrow$
$(q_0,00110) \vdash^* (q_0,\varepsilon)$, $q_0 \in F$ $\Rightarrow$ $00110 \in L(M)$

□

Definición 4.16: *Diagrama de estados*
El diagrama de estados de un *AFD* es un grafo dirigido y etiquetado donde los nodos son los estados.
Los estados finales se marcan con dobles círculos y el estado inicial se indica con "$>$".
Los arcos de q a q' se etiquetarán con σ siempre que $\delta(q,\sigma) = q'$.

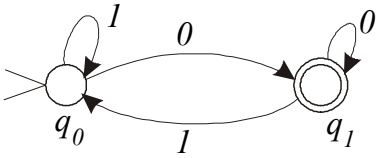
Ejemplo 4.17:
El diagrama de estados del autómata propuesto en el ejemplo anterior sería el siguiente:



□

Ejemplo 4.18:
Sea $M = (K, \Sigma, \delta, s, F)$ con:
$K = \{ q_0, q_1 \}$
$\Sigma = \{ 0, 1 \}$

$\delta(q,\sigma)$	0	1
q_0	q_1	q_0
q_1	q_1	q_0



$s = q_0$
$F = \{ q_1 \}$

$L(M) = \{ w \in \{0,1\}^* \mid 0$ es sufijo de $w \}$ □

Ejemplo 4.19:

Sea $M = (K , \Sigma , \delta , s , F)$ con:

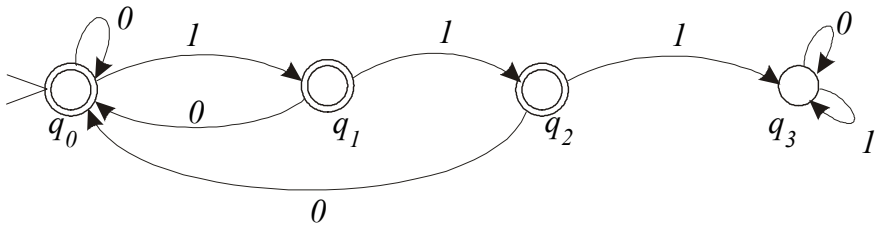
$K = \{ q_0 , q_1 , q_2 , q_3 \}$

$\Sigma = \{ 0 , 1 \}$

$\delta(q,\sigma)$	0	1
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_0	q_3
q_3	q_3	q_3

$s = q_0$

$F = \{ q_0 , q_1 , q_2 \}$



$L(M) = \{ w \in \{0,1\}^* \mid w$ no contiene la subcadena $111 \}$

Nota: A los estados como el q_3 se les denomina *estados pozo*, por razones obvias. □

Definición 4.20: *Estado accesible, Estado inaccesible*

Sea M un *AF*. Un estado $q \in K$ es accesible sii $\exists x \in \Sigma^* \mid (s,x) \vdash^* (q , \varepsilon)$.

Un estado $q \in K$ es inaccesible sii no es accesible.

Nota: Esta definición es válida tanto para los autómatas deterministas como para los no deterministas que veremos en el siguiente apartado.

4.2 Autómatas Finitos No Deterministas (*AFND*)

En este apartado introducimos el no determinismo en los *AF*, obteniendo los *AFND*. La introducción del no determinismo supone varias posibilidades que antes no teníamos en los *AFD*. Así, para una entrada y un estado, pueden haber varios próximos estados (incluso ninguno), por tanto pueden haber varias próximas configuraciones (incluso ninguna) para una dada. Además se permite consumir más de un símbolo en un solo paso, e incluso transitar entre estados consumiendo la cadena vacía, es decir, ningún símbolo.

Este no determinismo permite que para una misma cadena puedan haber varias computaciones, e incluso que en alguna de ellas puedan darse *bloqueos* del autómata, que se producen cuando sin haber terminado de leer la cadena de entrada no hay configuración siguiente. Las computaciones que terminan en bloqueos, se produzcan en estados finales o no, se consideran computaciones de no aceptación. Una cadena se considera aceptada por un *AFND* si a partir de ella hay alguna computación que la acepta.

Como veremos, los *AFND* aceptan los mismos lenguajes que los *AFD*, los regulares. Sin embargo es útil su introducción por comodidad, ya que normalmente un *AFND* tiene menos estados y transiciones que un *AFD* que acepte el mismo lenguaje.

A continuación vamos a definir formalmente *AFND* y los conceptos relacionados. Habrá conceptos que si bien se denominan igual para los *AFD* son realmente distintos pues aquí se refieren a *AFND*. Esta sobrecarga se mantendrá y se diferenciarán unos conceptos de otros según el tipo de autómata al que nos estemos refiriendo.

Definición 4.21: *Autómata Finito No Determinista (AFND)*
Un autómata finito no determinista (*AFND*) es una quíntupla $M = (K, \Sigma, \Delta, s, F)$ donde:

- K es un conjunto finito no vacío de estados;
- Σ es un alfabeto;
- s es el estado inicial ($s \in K$);
- F es el conjunto de estados finales ($F \subseteq K$);
- Δ es una relación de transición definida como un subconjunto finito de $K \times \Sigma^* \times K$.

Definición 4.22: *Transición*
Sea $M = (K , \Sigma , \Delta , s , F)$ un *AFND*. Cada terna $(q, u, p) \in \Delta$ se denomina transición de M .

Notas: Por tanto, Δ es un conjunto finito de transiciones.
Si $(q, u, p) \in \Delta$ y M está en el estado q y M puede consumir u de la cadena de entrada, entonces al hacerlo pasa al estado p .

Definición 4.23: *Configuración*
Dado $M = (K, \Sigma, \Delta, s, F)$ *AFND*, una configuración de M es cualquier par $(q,w) \in K \times \Sigma^*$.

Definición 4.24: *Configuración inicial*
Dado $M = (K , \Sigma , \Delta , s , F)$ *AFND*, definimos configuración inicial de M como toda configuración de la forma (s,w) .

Definición 4.25: *Configuración terminal*
Dado $M = (K , \Sigma , \Delta , s , F)$ *AFND*, definimos configuración terminal de M como toda configuración de la forma (q,ε) .

Definición 4.26: *Transitar directamente*
Sea M un *AFND*, y sean (q,w) y (q',w') configuraciones de M .
Diremos que (q,w) transita directamente a (q',w') , notado $(q,w) \vdash (q',w')$, sii:

$$\exists u \in \Sigma^* \mid (w = uw') \wedge ((q,u,q') \in \Delta)$$

Notas: Transitar directamente no es una función, ya que para algunas configuraciones (q,w) puede haber varias (o ninguna) (q',w') tal que $(q,w) \vdash (q',w')$.
Pero transitar directamente sí es una relación de $(K \times \Sigma^*) \times (K \times \Sigma^*)$ definida como $((q,uw') , (q',w'))$ con $(q,u,q') \in \Delta$.

Definición 4.27: *Configuración de bloqueo*
Sea $M = (K , \Sigma , \Delta , s , F)$ un *AFND* y (q,w) una configuración de M.
Diremos que (q,w) es una configuración de bloqueo si no es terminal y $\nexists\ (q',w') \in K \times \Sigma^* \mid (q,w) \vdash (q',w')$.

Definición 4.28: *Transitar en n pasos*
Sea M un *AFND*, y sean (q,w) y (q',w') configuraciones de M.

Diremos que (q,w) transita en n pasos a (q',w'), notado $(q,w) \vdash^n (q',w')$, con $n>1$, sii:

$\exists\ C_1 , C_2 , \ldots , C_{n-1}$ configuraciones de M $\mid$
$$(q,w) \vdash C_1 , \ C_1 \vdash C_2 , \ \ldots , \ C_{n-1} \vdash (q',w')$$

Diremos que (q,w) transita en 1 paso a (q',w'), notado $(q,w) \vdash^1 (q',w')$, sii:
$$(q,w) \vdash (q',w')$$

Diremos que (q,w) transita en 0 pasos a (q',w'), notado $(q,w) \vdash^0 (q',w')$, sii:
$$(q,w)=(q',w')$$

Definición 4.29: *Transitar en al menos un paso*
Diremos que (q,w) transita en al menos un paso a (q',w'), notado $(q,w) \vdash^+ (q',w')$, sii:
$$\exists n>0 \mid (q,w) \vdash^n (q',w')$$

Definición 4.30: *Transitar*
Sea M un *AFND*, y sean (q,w) y (q',w') configuraciones de M.
Diremos que (q,w) transita a (q',w'), notado $(q,w) \vdash^* (q',w')$, sii:
$$\exists n\geq 0 \mid (q,w) \vdash^n (q',w')$$

En resumen:
$\vdash$ es una relación binaria entre configuraciones (sobre $K \times \Sigma^*$)
$\vdash^+$ es su cierre transitivo
$\vdash^*$ es su cierre reflexivo y transitivo

Definición 4.31: *Computación, Longitud de una computación*
Sea M un *AFND*, definimos computación de M como cualquier secuencia finita de configuraciones C_0 , C_1 , C_2 , ... , C_n tal que $C_i \vdash C_{i+1}$ (siendo C_{i+1} la *configuración siguiente* de la C_i), donde $i = 0$, ... , $n-1$ con $n \geq 0$.
La notaremos $C_0 \vdash C_1 \vdash C_2 \vdash ... \vdash C_n$.
La longitud de dicha computación es n .

Nota: En un *AFND*, para toda configuración, ya sea terminal o no terminal, pueden existir cero, una o más configuraciones siguientes.

Definición 4.32: *Computación bien iniciada / terminada / completa / bloqueada*
Computación bien iniciada es cualquier computación cuya primera configuración es inicial.
Computación terminada es cualquier computación cuya última configuración es terminal.
Una computación es completa si es una computación bien iniciada y terminada.
Una computación está bloqueada si su última configuración es de bloqueo.

Definición 4.33: *Estado terminal*
Estado terminal es el estado de la última configuración de una computación terminada.

Definición 4.34: *Cadena aceptada*
Sea $M = (K, \Sigma, \delta, s, F)$ un *AFD* y $w \in \Sigma^*$.
Diremos que w es una cadena aceptada por M sii $\exists q \in F \mid (s,w) \vdash^* (q,\varepsilon)$.

Definición 4.35: *Lenguaje aceptado*
Sea M un *AFND*, definimos el lenguaje aceptado por M , notado $L(M)$, como:

$$L(M) = \{ w \in \Sigma^* \mid w \text{ es aceptada por } M \}$$

Nota: Las cadenas y los lenguajes aceptados también suelen denominarse *reconocidos*.

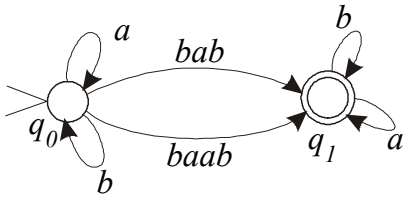
Definición 4.36: *Conjunto de lenguajes aceptados por AFND*
El conjunto de los lenguajes aceptados por *AFND*, notado $\mathcal{L}(AFND)$, se define como:

$$\mathcal{L}(AFND) = \{ L \mid \exists M \ AFND \ \text{ con } \ L(M) = L \}$$

Notas: Utilizaremos $\mathcal{L}(AF)$ para referirnos al conjunto de lenguajes aceptados por los *AF*.
Como ya hemos comentado, $\mathcal{L}(AFND) = \mathcal{L}(AFD) = \mathcal{L}(AF) = \mathcal{L}_3$.

Definición 4.37: *Diagrama de estados*
El diagrama de estados de un *AFND* es un grafo dirigido y etiquetado donde los nodos son los estados.
Los estados finales se marcan con dobles círculos y el estado inicial se indica con "$>$".
Los arcos de q a q' se etiquetarán con w siempre que $(q,w,q') \in \Delta$.

Ejemplo 4.38:
Sea $M = (K , \Sigma , \Delta , s , F)$ un *AFND* donde:
$K = \{ q_0 , q_1 \}$
$\Sigma = \{ a , b \}$
$\Delta = \{ (q_0 , a , q_0) , (q_0 , b , q_0) , (q_0 , bab , q_1) , (q_0 , baab , q_1) ,$
$(q_1 , a , q_1) , (q_1 , b , q_1) \}$
$s = q_0$
$F = \{ q_1 \}$



□

Ejemplo 4.39:

Sea $M = (K , \Sigma , \Delta , s , F)$ con:

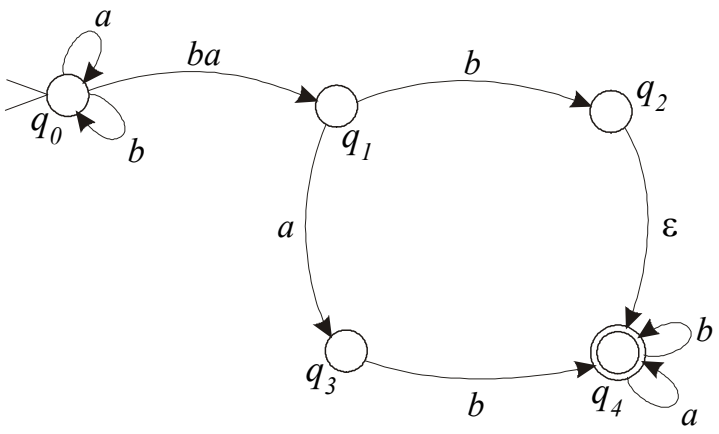
$K = \{ q_0 , q_1 , q_2 , q_3 , q_4 \}$

$\Sigma = \{ a , b \}$

$\Delta = \{ (q_0 , a , q_0) , (q_0 , b , q_0) , (q_0 , ba , q_1) ,$
$(q_1 , b , q_2) , (q_1 , a , q_3) ,$
$(q_2 , \varepsilon , q_4) ,$
$(q_3 , b , q_4) ,$
$(q_4 , a , q_4) , (q_4 , b , q_4) \}$

$s = q_0$

$F = \{ q_4 \}$



□

Ejemplo 4.40:

Los autómatas de los dos ejemplos anteriores (**4.38** y **4.39**) aceptan el mismo lenguaje:

$L(M) = \{ w \in \{a,b\}^* \mid w$ contiene la subcadena bab o $baab \}$

□

Ejemplo 4.41:

Vemos ahora todas las posibles computaciones que puede realizar el autómata del ejemplo **4.39** con la cadena $baababaaa$.

1ª) $(q_0 , baababaaa) \vdash (q_0 , aababaaa) \vdash (q_0 , ababaaa) \vdash$
$(q_0 , babaaa) \vdash (q_0 , abaaa) \vdash (q_0 , baaa) \vdash (q_0 , aaa) \vdash$
$(q_0 , aa) \vdash (q_0 , a) \vdash (q_0 , \varepsilon)$

Esta computación no acepta la cadena.

2ª) $(q_0, baababaaa) \vdash (q_0, aababaaa) \vdash (q_0, ababaaa) \vdash (q_0, babaaa) \vdash (q_0, abaaa) \vdash (q_0, baaa) \vdash (q_1, aa) \vdash (q_3, a)$

Esta computación se queda bloqueada, y por tanto no acepta la cadena.

3ª) $(q_0, baababaaa) \vdash (q_0, aababaaa) \vdash (q_0, ababaaa) \vdash (q_0, babaaa) \vdash (q_1, baaa) \vdash (q_2, aaa) \vdash (q_4, aaa) \vdash (q_4, aa) \vdash (q_4, a) \vdash (q_4, \varepsilon)$

Esta computación acepta la cadena.

4ª) $(q_0, baababaaa) \vdash (q_1, ababaaa) \vdash (q_3, babaaa) \vdash (q_4, abaaa) \vdash (q_4, baaa) \vdash (q_4, aaa) \vdash (q_4, aa) \vdash (q_4, a) \vdash (q_4, \varepsilon)$

Esta computación acepta la cadena.

Podemos concluir que:

$$\left.\begin{array}{l} (q_0, baababaaa) \vdash^* (q_4, \varepsilon) \\ q_4 \in F \end{array}\right\} \Rightarrow baababaaa \in L(M)$$

□

Ejemplo 4.42:

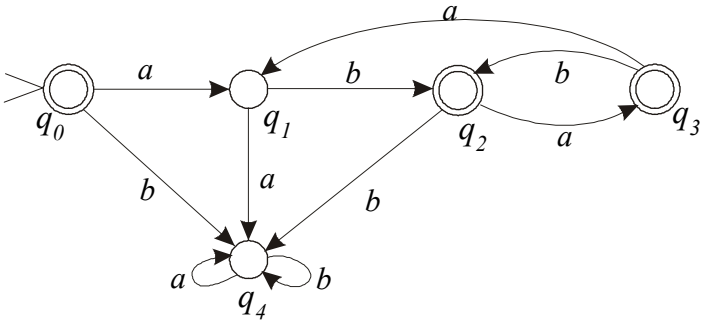
Sea $L = (ab + aba)^*$, veremos varios autómatas que reconocen este lenguaje:

$M_1 = (K, \Sigma, \delta, s, F)$ con:

$K = \{ q_0, q_1, q_2, q_3, q_4 \}$

$\Sigma = \{ a, b \}$

$\delta(q,\sigma)$	a	B
q_0	q_1	q_4
q_1	q_4	q_2
q_2	q_3	q_4
q_3	q_1	q_2
q_4	q_4	q_4



$s = q_0$

$F = \{ q_0, q_2, q_3 \}$

Sea $M_2 = (K , \Sigma , \Delta , s , F)$ *AFND* con:

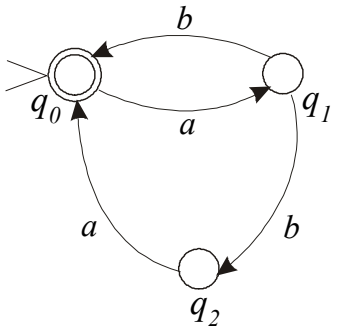
$K = \{ q_0 , q_1 , q_2 \}$

$\Sigma = \{ a , b \}$

$\Delta = \{ (q_0 , a , q_1) ,$
$(q_1 , b , q_0) , (q_1 , b , q_2) ,$
$(q_2 , a , q_0) \}$

$s = q_0$

$F = \{ q_0 \}$



Sea $M_3 = (K , \Sigma , \Delta , s , F)$ *AFND* con:

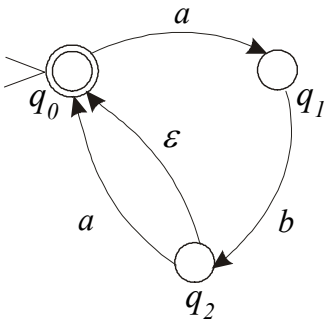
$K = \{ q_0 , q_1 , q_2 \}$

$\Sigma = \{ a , b \}$

$\Delta = \{ (q_0 , a , q_1) ,$
$(q_1 , b , q_2) ,$
$(q_2 , a , q_0) , (q_2 , \varepsilon , q_0) \}$

$s = q_0$

$F = \{ q_0 \}$



Sea $M_4 = (K , \Sigma , \Delta , s , F)$ *AFND* con:

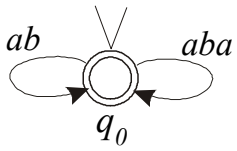
$K = \{ q_0 \}$

$\Sigma = \{ a , b \}$

$\Delta = \{ (q_0 , ab , q_0) , (q_0 , aba , q_0) \}$

$s = q_0$

$F = \{ q_0 \}$



$L(M_1) = L(M_2) = L(M_3) = L(M_4) = L$ □

En realidad un *AFD* no es más que un tipo especial de *AFND* donde $\Delta \subset K \times \Sigma^* \times K$ es una función $\delta: K \times \Sigma \rightarrow K$. De esta forma, un *AFND* $M = (K , \Sigma , \Delta , s , F)$ es un *AFD* sii:

$[\ (q,u,q') \in \Delta \ \Rightarrow \ |u|=1 \ ] \ \wedge \ [\ \forall (q,\sigma) \in K \times \Sigma \ \ \exists! \, q' \in K \ \mid \ (q,\sigma,q') \in \Delta \ ]$

Definición 4.43: *Autómatas finitos equivalentes ($M_1 \equiv M_2$)*

Sean M_1 y M_2 dos autómatas finitos, decimos que son equivalentes sii:

$$L(M_1) = L(M_2)$$

4.3 Autómata Finito Determinista Mínimo (*AFDM*)

En este apartado veremos el concepto de *Autómata Finito Determinista Mínimo* (*AFDM*), el cual es importante, ya que nos permite tener un *AF* (salvo renumeración de estados) representante de cada lenguaje regular.

Definición 4.44: *Autómata Finito Determinista Mínimo (AFDM)*
Sea $M = (K, \Sigma, \delta, s, F)$ un *AFD*.
Diremos que M es un *AFDM* sii $\forall M' = (K', \Sigma, \delta', s', F')$ se cumple que:

$$M' \equiv M \Rightarrow ||K'|| \geq ||K||$$

Proposición 4.45:
$\forall M\ AFD\ \ \exists M'\ AFDM\ \mid M \equiv M'$

4.4 Equivalencias

Si bien no vamos a ver los distintos algoritmos de conversión entre las distintas formas de representar un lenguaje regular, sí al menos presentamos dos teoremas y un corolario al respecto.

Teorema 4.46:
$L \in \mathcal{L}(ER) \Leftrightarrow L \in \mathcal{L}(AF)$

Teorema 4.47:
$L \in \mathcal{L}.3 \Leftrightarrow L \in \mathcal{L}(AF)$

Corolario 4.48:
$\mathcal{L}.3 = \mathcal{L}(ER) = \mathcal{L}(AF)$

5
Condiciones de Regularidad

Hasta ahora si queremos mostrar que un lenguaje es regular sólo tenemos que buscar una representación "regular" (*AF, ER, GR*) para él. Pero el problema surge cuando queremos demostrar que un lenguaje dado no es regular, ya que tendríamos que demostrar que no existe ningún *AF* (o *ER*, o *GR*) que lo represente.

Dado que esto es difícil lo que haremos es establecer una serie de condiciones de regularidad (condiciones necesarias) que cumple todo lenguaje regular. Así, si demostramos que un lenguaje no cumple alguna de estas condiciones (necesarias) hemos demostrado que no es regular.

5.1 Myhill - Nerode

En este apartado vamos a ver una condición que es necesaria y suficiente, pero que habitualmente se utiliza como condición necesaria para demostrar que un lenguaje no es regular.

Definición 5.1: *Cadenas distinguibles, Cadenas indistinguibles*

Sea $L \subseteq \Sigma^*$ y sea $x,y \in \Sigma^*$.

Diremos que x e y son cadenas distinguibles con respecto a L sii

$$\exists z \in \Sigma^* \mid (xz \in L \wedge yz \notin L) \vee (xz \notin L \wedge yz \in L)$$

Diremos que x e y son cadenas indistinguibles con respecto a L sii

$$\forall z \in \Sigma^* \ (xz \in L \wedge yz \in L) \vee (xz \notin L \wedge yz \notin L)$$

Ejemplo 5.2:

Sea $L = \{ w \in \{0,1\}^* \mid w = y10 \text{ con } y \in \{0,1\}^* \}$

Las cadenas 01011 y 100 son distinguibles con respecto a L ya que

$$\exists z=0 \mid 010110 \in L \wedge 1000 \notin L$$

Las cadenas 0 y 100 son indistinguibles con respecto a L ya que $0z$ y $100z$ pertenecerán o no ambas a la vez a L según que z termine o no en 10. □

Definición 5.3: *Relación de indistinguibilidad*
Dado Σ, definimos la relación de indistinguibilidad sobre Σ^*, notada I_L, como:

$$I_L = \{ (x,y) \in \Sigma^* \times \Sigma^* \mid x \text{ e } y \text{ son indistinguibles con respecto a } L \}$$

Proposición 5.4:
I_L es una relación de equivalencia sobre Σ^*.

Demostración:
Trivialmente es reflexiva y simétrica.
También es transitiva ya que:

$$[[\forall z \in \Sigma^* \ (xz \in L \ \wedge \ yz \in L) \vee (xz \notin L \ \wedge \ yz \notin L)] \wedge [\forall z \in \Sigma^* \ (yz \in L \ \wedge \ wz \in L) \vee (yz \notin L \ \wedge \ wz \notin L)]]$$

$$\Downarrow$$

$$\forall z \in \Sigma^* \ (xz \in L \ \wedge \ wz \in L) \vee (xz \notin L \ \wedge \ wz \notin L)$$ □

Por ser I_L una relación de equivalencia sobre Σ^*, establece unas clases de equivalencia en Σ^* que constituyen una partición de Σ^*. Llamaremos Π_L a dicha partición.

Teorema de Myhill – Nerode:
$L \in L.3 \Leftrightarrow \| \Pi_L \| \in \mathbb{N}$

Demostración:
a) $L \in L.3 \Rightarrow \| \Pi_L \| \in \mathbb{N}$:
$L \in L.3 \Rightarrow L \in L(AFD) \Rightarrow \exists M \ AFD$ sin estados inaccesibles $\mid L(M) = L$.
Sea $M = (K , \Sigma , \delta , s , F)$ dicho autómata, con $K = \{ q_0 , q_1 , \ldots , q_n \}$ y $s = q_0$.
Para este autómata definimos la función δ^* como:
$\delta^* : K \times \Sigma^* \rightarrow K$
$\delta^*(q_i , x) = q_j$ sii $(q_i , x) \vdash^* (q_j , \varepsilon)$
Sea $A_i = \{ x \in \Sigma^* \mid \delta^*(q_0 , x) = q_i \}$ con $i = 0, 1, \ldots , n$.

Sea $\Pi = \{ A_0 , A_1 , \dots , A_n \}$.
Al ser M un *AFD* se cumple que Π es una partición de Σ^* y además:

$$\forall x,y \in A_i \ (x,y) \in I_L \text{ , con } i = 0, 1, \dots , n$$

Sea $\Pi' = \{ B \subseteq \Sigma^* \mid B = A_{i_1} \cup A_{i_2} \cup \dots \cup A_{i_k} \text{ con } A_{i_j} \in \Pi \ \wedge$

$$\forall x,y \in B \ (x,y) \in I_L \ \wedge$$

$$(x \in B \ \wedge \ y \notin B) \Rightarrow (x,y) \notin I_L \ \}$$

Por definición tenemos que Π' es una partición de Σ^*, que además cumple:
$(\Pi' = \Pi_L \ \wedge \ \| \Pi' \| \leq \| \Pi \| = n+1) \ \Rightarrow \ \| \Pi_L \| \leq n+1 \ \Rightarrow \ \| \Pi_L \| \in \mathbb{N}$

a) □

b) $\| \Pi_L \| \in \mathbb{N} \Rightarrow L \in L.3$:
$\| \Pi_L \| \in \mathbb{N} \Rightarrow \| \Pi_L \| = m$ con $m \geq 1 \Rightarrow \Pi_L = \{ A_0 , A_1 , \dots , A_{m-1} \}$
↑
por ser Π_L una partición de Σ^*

Sea $\Sigma = \{ a_1 , a_2 , \dots , a_n \}$ el alfabeto sobre el que está definido L .
Sea $M = (K , \Sigma , \delta , s , F)$ un *AFD* con:

$K = \{ q_0 , q_1 , \dots , q_{m-1} \}$
$\Sigma = \{ a_1 , a_2 , \dots , a_n \}$
$\delta (q_i , a_k) = q_j$ sii $x \in A_i \Rightarrow x a_k \in A_j$
$s = q_j$ sii $\varepsilon \in A_j$
$F = \{ q_i \in K \mid x \in A_i \Rightarrow x \in L \}$

Por construcción tenemos que $L(M) = L \ \Rightarrow \ L \in L.3$

b)□

Nota: M es el *AFDM* (salvo renumeración de estados) de L .

Los dos enunciados siguientes son equivalentes al del teorema de Myhill – Nerode:

"Un lenguaje L sobre Σ es no regular sii hay un subconjunto infinito de Σ^* tal que cualesquiera dos elementos suyos son distinguibles con respecto a L ."

"Sea $L \subseteq \Sigma^*$. Si hay un subconjunto de Σ^* de cardinal n tal que cualesquiera dos elementos suyos son distinguibles con respecto a L, entonces cualquier *AFD* que reconozca a L tiene al menos n estados."

El teorema, como ya hemos comentado, lo usaremos para demostrar que un lenguaje no es regular. Para ello, dado un lenguaje, buscaremos un conjunto infinito de cadenas tal que cualesquiera dos elementos suyos sean distinguibles con respecto a L, ya que la existencia de dicho conjunto implica que el conjunto de clases de equivalencia de la relación I_L no es finito, y por tanto el lenguaje no es regular.

Ejemplo 5.5:
Sea $L = \{ 0^n 1^n \mid n \geq 1 \}$.
Definimos $S = \{ 0^n \mid n \geq 1 \}$.
Cualesquiera dos elementos de S son distinguibles respecto a L, ya que $0^i, 0^k \in S$ son distinguibles con respecto a L por medio de la cadena 1^i:

$$(0^i 1^i \in L) \wedge (0^k 1^i \notin L) \Rightarrow L \text{ no es regular.}$$

Nota: También son distinguibles con respecto a L por medio de la cadena 1^k.

□

5.2 Bombeo

El lema del bombeo que vamos a ver es el más usado como condición necesaria para demostrar que un lenguaje no es regular.

Teorema 5.6:
Dado $M = (K, \Sigma, \delta, s, F)$ *AFD* con $\| K \| = n$, sea $L = L(M)$.
Entonces $\forall x \in L$, $|x| \geq n$, $\exists u,v,w \in \Sigma^*$ tales que:

1) $x = uvw$
2) $|uv| \leq n$
3) $|v| > 0$
4) $\forall m \geq 0 \quad uv^m w \in L$

Demostración:
Dado M un *AFD*, definimos la función δ^* como:

$$\delta^* : K \times \Sigma^* \rightarrow K$$
$$\delta^*(q_i, x) = q_j \quad \text{sii} \quad (q_i, x) \vdash^* (q_j, \varepsilon)$$

Sea $x \in L$ tal que $|x| \geq n$ con $n = \| K \|$
Así, $x = a_{x_1} a_{x_2} \ldots a_{x_n} y$ donde $y \in \Sigma^*$ y $a_{x_k} \in \Sigma$ con $k = 1, \ldots, n$
Veamos la secuencia de estados, con $s = q_{0)}$:

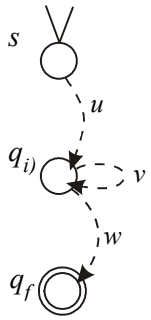
$$q_{0)}$$
$$q_{1)} = \delta^*(q_{0)}, a_{x_1})$$
$$q_{2)} = \delta^*(q_{0)}, a_{x_1} a_{x_2})$$
$$q_{3)} = \delta^*(q_{0)}, a_{x_1} a_{x_2} a_{x_3})$$
$$\vdots$$
$$q_{j)} = \delta^*(q_{0)}, a_{x_1} a_{x_2} \ldots a_{x_j})$$
$$\vdots$$
$$q_{n)} = \delta^*(q_{0)}, a_{x_1} a_{x_2} \ldots a_{x_n})$$

Esta secuencia tiene $n+1$ estados y $|| K ||=n$, por el principio de los casilleros algún estado en ella aparece al menos dos veces (bucle), así, sea $q_{i)} = q_{i+p)}$ con $0 \le i < i+p \le n$, entonces:

$$\delta^*(q_{0)}, a_{x_1} \dots a_{x_i}) = q_{i)}$$
$$\delta^*(q_{i)}, a_{x_{i+1}} \dots a_{x_{i+p}}) = q_{i)} = q_{i+p)}$$
$$\delta^*(q_{i)}, a_{x_{i+p+1}} \dots a_{x_n} y) = q_f \in F$$

Si denominamos:

$$u = a_{x_1} \dots a_{x_i}$$
$$v = a_{x_{i+1}} \dots a_{x_{i+p}}$$
$$w = a_{x_{i+p+1}} \dots a_{x_n} y$$



tenemos que:

- $|u| = i$, $|v| = p$
- $x = uvw$ (*1*)
- $|uv| = i+p \le n$ (*2*)
- $i < i+p \quad \Rightarrow \quad p > 0 \quad \Rightarrow \quad |v| > 0$ (*3*)
- $\delta^*(q_{i)}, v) = q_{i)} \Rightarrow \delta^*(q_{i)}, v^m) = q_{i)}, \forall m \ge 0 \Rightarrow$
  $\delta^*(q_{0)}, uv^m w) = q_f, \forall m \ge 0 \Rightarrow uv^m w \in L, \forall m \ge 0$ (*4*) □

Definición 5.7: *Condición de bombeo regular (CBR)*

Sea $L \subseteq \Sigma^*$.

Diremos que L cumple la condición de bombeo regular (cumple *CBR*) sii

$\exists n \in \mathbb{N} \mid \forall x \in L$ con $|x| \ge n$, $\exists u,v,w \in \Sigma^*$ tales que:

1) $x = uvw$
2) $|uv| \le n$
3) $|v| > 0$
4) $\forall m \ge 0 \quad uv^m w \in L$

Lema del bombeo para *L.3* :
$L \in L.3 \Rightarrow L$ cumple *CBR*

Para utilizar el lema del bombeo para demostrar que un lenguaje no es regular tenemos que demostrar que el lenguaje no cumple la *CBR*, es decir, que:

$\forall n \in \mathbb{N}$, $\exists x \in L$ con $|x| \geq n$ | $\forall u,v,w \in \Sigma^*$ no se cumplen conjuntamente las cuatro condiciones de la *CBR*.

La demostración de lo anterior se hace por reducción al absurdo:

Suponemos que $\exists u,v,w \in \Sigma^*$ tales que cumplen conjuntamente las cuatro condiciones de la *CBR*, y llegamos a una contradicción.

Lo único que tenemos que hacer es elegir adecuadamente un $x \in L$ que nos lleve a dicha contradicción.

Ejemplo 5.8:
Sea $L=\{ 0^n 1^n \mid n \geq 0 \}$.
Demostraremos que no es un lenguaje regular.

Para ello debemos demostrar que:
$\forall n \in \mathbb{N}$, $\exists x \in L$ con $|x| \geq n$ | $\forall u,v,w \in \Sigma^*$ no se cumplen conjuntamente las cuatro condiciones de la *CBR*.

Así, sea $n \in \mathbb{N}$, y sea $x = 0^n 1^n \in L$, $|x| = 2n \geq n$
Supongamos que $\exists u,v,w \in \Sigma^*$ | cumplen conjuntamente las cuatro condiciones de la *CBR*:

1) $x = uvw$

2) $|uv| \leq n \Rightarrow uv = 0^k$ con $k \leq n \Rightarrow w = 0^{n-k} 1^n$
($\uparrow$ *por 1)*; ($\uparrow$ *por 1)*)

3) $|v| > 0 \Rightarrow v = 0^j$ con $0 < j \leq k$
($\uparrow$ *por 2)*)

4) $\forall m \geq 0 \quad uv^m w \in L \Rightarrow \forall m > 1 \quad uv^m w \in L$

Pero $uv^m w = (uv)v^{m-1}w = 0^k (0^j)^{m-1} 0^{n-k} 1^n = 0^{k+j(m-1)+(n-k)} 1^n \notin L$ porque $j(m-1) > 0$ al ser $j > 0$ y $m > 1$ $\Rightarrow$ La *CBR* no se cumple $\Rightarrow$ L no es regular.

También podemos demostrarlo si en el paso *4)* usamos $m=0$:

4) $\forall m \geq 0 \quad uv^m w \in L \Rightarrow \forall m=0, \; uv^m w \in L$
Así, $uv^m w = uw = 0^{k-j} 0^{n-k} 1^n = 0^{(k-j)+(n-k)} 1^n = 0^{(n-j)} 1^n \notin L$ porque $n-j < n$ al ser $0 < j$ $\Rightarrow$ La *CBR* no se cumple $\Rightarrow$ L no es regular. □

6
Lenguajes de Contexto Libre

Los lenguajes con paréntesis balanceados no son regulares (son *LCL*) y tienen una gran relevancia debido a que la mayoría de los lenguajes de programación son de ese tipo. De aquí la importancia de los *LCL*, de las *GCL* que son la forma más habitual de representarlos, y de los *APND*, que son sus dispositivos reconocedores.

6.1 Derivaciones y Ambigüedad

Definición 6.1: *Subárbol-A de derivación*
Sea $G = (N, T, P, S)$ una *GCL*.
Un árbol es subárbol-*A* de derivación asociado a G, con $A \in N$, si:

1) Cada nodo tiene una etiqueta que es un símbolo de V.
2) La etiqueta del nodo raíz es A.
3) Si un nodo n tiene un símbolo B y $n_1, n_2, \ldots, n_k$ son los hijos del nodo n de izquierda a derecha con etiquetas $X_1, X_2, \ldots, X_k$ respectivamente, entonces:

$$(B \rightarrow X_1 X_2 \ldots X_k) \in P$$

Definición 6.2: *Producto de un subárbol-A de derivación*
El producto de un subárbol-*A* de derivación es el recorrido en preorden de las etiquetas de las hojas.

Nota: El producto siempre será una cadena $\alpha \in V^+$.

Definición 6.3: *A-derivación*
Dado un subárbol-*A* de derivación, una *A*-derivación asociada a él es una secuencia finita de producciones directas, obtenida a partir del símbolo A, aplicando las reglas según se indica en el subárbol-*A*.

Definición 6.4: *Árbol de derivación*
Un árbol de derivación es un subárbol-S de derivación siendo S el axioma de la gramática.

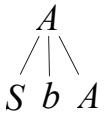
Definición 6.5: *Derivación*
Una derivación es una S-derivación donde S es el axioma de la gramática.

Definición 6.6: *Derivación extrema izquierda, Derivación extrema derecha*
Si en cada paso de una derivación se aplica una regla al no terminal más a la izquierda/derecha entonces la derivación es extrema izquierda/derecha , notada *DEI* / *DED* .

Ejemplo 6.7:
Sea $G = (N , T , P , S)$ una *GCL* con:

$N = \{ S , A \}$
$T = \{ a , b \}$
$P = \{ S \to aAS \mid a ,$
$A \to SbA \mid SS \mid ba \}$

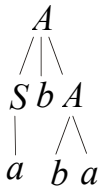
Veamos algunos subárboles-A de derivación, sus productos y sus A-derivaciones asociadas:



Producto: *SbA*
A-derivación:
$A \Rightarrow SbA$

Producto: *SS*
A-derivación:
$A \Rightarrow SS$

Producto: *ba*
A-derivación:
$A \Rightarrow ba$

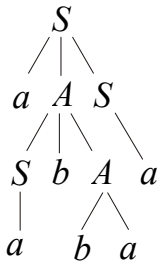


Producto: *abba*
A-derivaciones:
$A \Rightarrow SbA \Rightarrow Sbba \Rightarrow abba$
$A \Rightarrow SbA \Rightarrow abA \Rightarrow abba$

Veamos ahora algunos árboles de derivación (subárboles-S), sus productos y algunas de sus derivaciones asociadas:

Producto: a
Derivación:
$S \Rightarrow a$

Producto: aAS
Derivación:
$S \Rightarrow aAS$



Producto: $aabbaa$
Derivaciones:

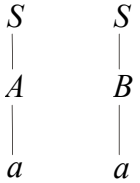
$$S \Rightarrow a\boldsymbol{A}S \Rightarrow aSb\boldsymbol{A}S \Rightarrow aSbba\boldsymbol{S} \Rightarrow a\boldsymbol{S}bbaa \Rightarrow aabbaa$$
$$S \Rightarrow a\boldsymbol{A}S \Rightarrow a\boldsymbol{S}bAS \Rightarrow aab\boldsymbol{A}S \Rightarrow aabba\boldsymbol{S} \Rightarrow aabbaa \quad (\text{DEI})$$
$$S \Rightarrow aA\boldsymbol{S} \Rightarrow a\boldsymbol{A}a \Rightarrow aSb\boldsymbol{A}a \Rightarrow a\boldsymbol{S}bbaa \Rightarrow aabbaa \quad (\text{DED})$$

Nota: Derivación es distinto de árbol de derivación □

Definición 6.8: *Gramática ambigua*
Sea $G = (N, T, P, S)$ una *GCL*, diremos que G es ambigua sii
$\exists w \in L(G) \mid w$ es producto de más de un árbol de derivación.

Ejemplo 6.9:
Sea $G = (N, T, P, S)$ con:

$N = \{ A , B \}$
$T = \{ a \}$
$P = \{ S \to A \mid B \ ,$
$\quad A \to a \ ,$
$\quad B \to a \ \}$



La cadena a es producto de dos árboles distintos, por tanto G es ambigua.

Nota: La ambigüedad es una propiedad indecidible, es decir, no existe ningún algoritmo conclusivo que para cualquier *GCL* dada me conteste en un tiempo finito si es o no ambigua. □

Definición 6.10: *Lenguaje inherentemente ambiguo*
Un lenguaje de contexto libre L es inherentemente ambiguo sii
$L = L(G) \Rightarrow G$ es ambigua.

Ejemplo 6.11:
Sea $\Sigma = \{ a, b, c, d \}$.
Sea $L_1 = \{ w \in \Sigma^* \mid w = a^n b^n c^m d^m \text{ con } n,m \geq 1 \}$
Sea $L_2 = \{ w \in \Sigma^* \mid w = a^n b^m c^m d^n \text{ con } n,m \geq 1 \}$
El lenguaje $L = L_1 \cup L_2$ es un lenguaje inherentemente ambiguo. □

6.2 Recursividad

En este apartado veremos el concepto de recursividad y sus tipos, así como algunas proposiciones relacionadas.

Definición 6.12: *Recursividad*
Sea $G = (N, T, P, S)$ una gramática de contexto libre.
Diremos que:

G es *recursiva directa por la izquierda* sii
$$\exists A \in N \text{ tal que } A \Rightarrow A\alpha \text{ , con } \alpha \in V^*$$

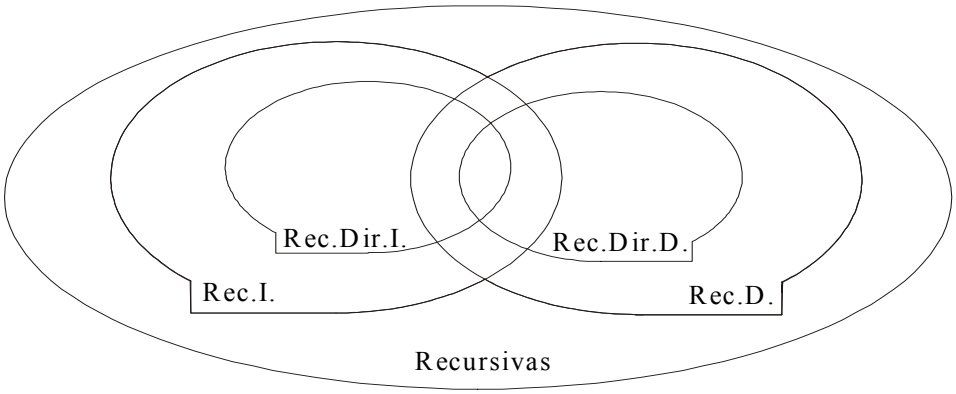
G es *recursiva directa por la derecha* sii
$$\exists A \in N \text{ tal que } A \Rightarrow \alpha A \text{ , con } \alpha \in V^*$$

G es *recursiva por la izquierda* sii
$$\exists A \in N \text{ tal que } A \Rightarrow^+ A\alpha \text{ , con } \alpha \in V^*$$

G es *recursiva por la derecha* sii
$$\exists A \in N \text{ tal que } A \Rightarrow^+ \alpha A \text{ , con } \alpha \in V^*$$

G es *recursiva* sii
$$\exists A \in N \text{ tal que } A \Rightarrow^+ \alpha A \beta \text{ , con } \alpha,\beta \in V^*$$

En el siguiente diagrama se observa la relación entre los distintos tipos de recursividad:



Rec.Dir.I. = recursivas directas por la izquierda
Rec.I. = recursivas por la izquierda
Rec.Dir.D. = recursivas directas por la derecha
Rec.D. = recursivas por la derecha

Nota: Las gramáticas que son recursivas por la derecha e izquierda a la vez no son regulares.

Ejemplo 6.13:

Sea $G = (N, T, P, S)$ con:

$N = \{ S , A , B \}$

$T = \{ a , b , c \}$

$P = \{ S \to AB ,$

$A \to Aa \mid a ,$

$B \to bBc \mid bc \}$

G es recursiva directa por la izquierda, recursiva por la izquierda y recursiva.

□

Proposición 6.14:

Sea G una gramática de contexto libre, entonces existe G' gramática de contexto libre no recursiva directa por la izquierda tal que $G \equiv G'$.

Proposición 6.15:

Sea G una gramática de contexto libre, entonces existe G' gramática de contexto libre no recursiva por la izquierda tal que $G \equiv G'$.

6.3 Simplificación de *GCL*

Veremos en este apartado varios conceptos relacionados con la simplificación de las *GCL*, incluyendo el de símbolo inútil y el de gramática propia, así como algunas proposiciones relacionadas.

Definición 6.16: *Símbolo útil, Símbolo inútil*
Sea $G = (N , T , P , S)$ una *GCL* y sea $X \in V$, diremos que X es un símbolo útil sii $\exists \alpha, \beta \in V^*$ y $w \in T^+ \mid S \Rightarrow^* \alpha X \beta \Rightarrow^* w$
En caso contrario diremos que X es un símbolo inútil.

Definición 6.17: *Símbolo terminable, Símbolo no terminable*
Diremos que X es un símbolo terminable sii $\exists w \in T^+ \mid X \Rightarrow^* w$
En caso contrario diremos que X es un símbolo no terminable.

Definición 6.18: *Símbolo accesible, Símbolo inaccesible*
Diremos que X es un símbolo accesible sii $\exists \alpha, \beta \in V^* \mid S \Rightarrow^* \alpha X \beta$
En caso contrario diremos X que es un símbolo inaccesible.

Nota: Que X sea terminable y accesible no implica que sea útil.

Definición 6.19: *Gramática propia*
Una *GCL* diremos que es propia sii no es recursiva izquierda y no tiene símbolos inútiles.

Proposición 6.20:
Sea G una gramática de contexto libre que genera un lenguaje distinto del vacío, entonces existe G' gramática de contexto libre propia tal que $G \equiv G'$.

Proposición 6.21:
Sea G una gramática de contexto libre, entonces existe G' gramática de contexto libre sin reglas unitarias tal que $G \equiv G'$.

6.4 Formas Normales

Veremos dos formas normalizadas, la de Chomsky y la de Greibach, para las *GCL*.

Definición 6.22: *Forma normal de Chomsky (FNC)*

Sea $G = (N, T, P, S)$ una *GCL*. Diremos que G está en forma normal de Chomsky sii

$$\forall (A \to \alpha) \in P \quad \alpha = BC \ \vee \ \alpha = a \ , \text{ con } B,C \in N \text{ y } a \in T$$

Proposición 6.23:

$\forall G \ GCL \quad \exists G'$ en $FNC \mid G \equiv G'$

Definición 6.24: *Forma normal de Greibach (FNG)*

Sea $G = (N, T, P, S)$ una *GCL*. Diremos que G está en forma normal de Greibach sii

$$\forall (A \to \alpha) \in P \quad \alpha = a\beta \ , \text{ con } a \in T \text{ y } \beta \in N^*$$

Proposición 6.25:

$\forall G \ GCL \quad \exists G'$ en $FNG \mid G \equiv G'$

6.5 Propiedades de Cierre

Teorema 6.26:

$L.2$ es cerrado para las operaciones de unión, concatenación y estrella de Kleene.

Proposición 6.27:

$L.2$ no es cerrado para las operaciones de intersección y complemento.

Demostración:

Sea $L_1 = \{ a^n b^n c^m \mid n,m \geq 1 \}$.

$L_1 \in L.2$ ya que puede ser generado por una gramática de tipo dos, como por ejemplo $G = (N, T, P, S)$ con:

$N = \{ S , A , B \}$
$T = \{ a, b, c \}$
$P = \{ \ S \to AB ,$
$A \to aAb \mid ab ,$
$B \to cB \mid c \}$

Sea $L_2 = \{ a^m b^n c^n \mid n,m \geq 1 \}$.
$L_2 \in L.2$ (véase ejemplo **6.13**)

$L_1 \cap L_2 = \{ a^n b^n c^n \mid n \geq 1 \} \notin L.2$ (véase ejemplo **6.49**)
por tanto $L.2$ no es cerrado para la intersección.

Dado que $L.2$ es cerrado para la unión, y puesto que $L_1 \cap L_2 = \overline{(\overline{L_1} \cup \overline{L_2})}$, tenemos por tanto que $L.2$ no es cerrado para el complemento. □

6.6 Autómatas con Pila No Deterministas (*APND*)

Definición 6.28: *Autómata con pila no determinista (APND)*
Un autómata con pila no determinista es una séxtupla $M = (K , \Sigma , \Gamma , \Delta , s , F)$ donde:

K es un conjunto finito de estados
Σ es un alfabeto (símbolos de entrada)
Γ es un alfabeto (símbolos de la pila)
$s \in K$ es el estado inicial
$F \subseteq K$ es el conjunto de estados finales
Δ es una relación de transición definida como un subconjunto finito de $(K \times \Sigma^* \times \Gamma^*) \times (K \times \Gamma^*)$

Definición 6.29: *Transición*
Sea $M = (K , \Sigma , \Gamma , \Delta , s , F)$ un *APND*. Se denomina transición del autómata M a cada par $((p, u, \beta), (q, \gamma)) \in \Delta$.
Así, Δ es un conjunto finito de transiciones.

Nota: Si $((p, u, \beta), (q, \gamma)) \in \Delta$ entonces, si M está en el estado p con β en la cima de la pila y u es prefijo de la cadena de entrada, M puede consumir u de la cadena de entrada, reemplazar β por γ en la cima de la pila, y pasar al estado q .

Definición 6.30: *Configuración*
C es una configuración de $M = (K, \Sigma, \Gamma, \Delta, s, F)$ sii $C \in K \times \Sigma^* \times \Gamma^*$.

Definición 6.31: *Configuración inicial*
Dado $M = (K, \Sigma, \Gamma, \Delta, s, F)$ un *APND*, definimos configuración inicial de M como toda configuración de la forma (s, w, ε) .

Definición 6.32: *Configuración terminal*
Dado $M = (K, \Sigma, \Gamma, \Delta, s, F)$ un *APND*, definimos configuración terminal de M como toda configuración de la forma $(q, \varepsilon, \varepsilon)$.

Definición 6.33: *Transitar directamente*
Sean (q, u, α) y (q', u', α') dos configuraciones del autómata $M = (K, \Sigma, \Gamma, \Delta, s, F)$. Diremos que (q, u, α) transita directamente a (q', u', α') , notado $(q, u, \alpha) \vdash (q', u', \alpha')$, sii:

$\exists v \in \Sigma^*$ y $\beta, \beta', \gamma \in \Gamma^*$ |

$$(u = vu') \wedge (\alpha = \beta\gamma) \wedge (\alpha' = \beta'\gamma) \wedge ((q,v,\beta), (q',\beta')) \in \Delta$$

Definición 6.34: *Configuración de bloqueo*
Sea $M = (K, \Sigma, \Gamma, \Delta, s, F)$ *APND* y (q, u, α) una configuración de M . Diremos que (q, u, α) es una configuración de bloqueo si no es terminal y $\nexists\ (q', u', \alpha') \in K \times \Sigma^* \times \Gamma^*$ | $(q, u, \alpha) \vdash (q', u', \alpha')$.

Definición 6.35: *Transitar en n pasos*
Sea M un *APND*, y sean (q, u, α) y (q', u', α') configuraciones de M .

Diremos que (q, u, α) transita en n pasos a (q', u', α') , con $n>1$, notado $(q, u, \alpha) \vdash^n (q', u', \alpha')$, sii:
$\exists\ C_1, ..., C_{n-1}$ configuraciones de M |

$$(q, u, \alpha) \vdash C_1 , C_1 \vdash C_2 , ..., C_{n-1} \vdash (q', u', \alpha')$$

Diremos que (q, u, α) transita en 1 paso a (q', u', α'), notado $(q, u, \alpha) \vdash^{1} (q', u', \alpha')$, sii:

$$(q, u, \alpha) \vdash (q', u', \alpha')$$

Diremos que (q, u, α) transita en 0 pasos a (q', u', α'), notado $(q, u, \alpha) \vdash^{0} (q', u', \alpha')$, sii:

$$(q, u, \alpha) = (q', u', \alpha')$$

Definición 6.36: *Transitar en al menos un paso*
Sea M un *APND*, y sean (q, u, α) y (q', u', α') configuraciones de M.
Diremos que (q, u, α) transita en al menos un paso a (q', u', α'), notado $(q, u, \alpha) \vdash^{+} (q', u', \alpha')$, sii:

$$\exists n>0 \mid (q, u, \alpha) \vdash^{n} (q', u', \alpha')$$

Definición 6.37: *Transitar*
Sea M un *APND*, y sean (q, u, α) y (q', u', α') configuraciones de M.
Diremos que (q, u, α) transita a (q', u', α'), notado
$(q, u, \alpha) \vdash^{*} (q', u', \alpha')$, sii:

$$\exists n\geq 0 \mid (q, u, \alpha) \vdash^{n} (q', u', \alpha')$$

Definición 6.38: *Computación, Longitud de una computación*
Sea M un *APND*, definimos computación de M como cualquier secuencia finita de configuraciones $C_0, C_1, \ldots, C_n$ tal que $C_i \vdash C_{i+1}$ (siendo C_{i+1} la *configuración siguiente* de la C_i), donde $i = 0, \ldots, n-1$ con $n\geq 0$.
La notaremos $C_0 \vdash C_1 \vdash C_2 \vdash \ldots \vdash C_n$.
La longitud de dicha computación es n.

Definición 6.39: *Computación bien iniciada / terminada / completa / bloqueada*
Computación bien iniciada es cualquier computación cuya primera configuración es inicial.
Computación terminada es cualquier computación cuya última configuración es terminal.
Una computación es completa si es una computación bien iniciada y terminada.
Una computación está bloqueada si su última configuración es de bloqueo.

Definición 6.40: *Estado terminal*
Estado terminal es el estado de la última configuración de una computación terminada.

Definición 6.41: *Cadena aceptada*
Sea $w \in \Sigma^*$ y M un *APND*.
Diremos que w es aceptada por M sii $\exists q \in F \mid (s, w, \varepsilon) \vdash^* (q, \varepsilon, \varepsilon)$.

Definición 6.42: *Lenguaje aceptado*
Sea M un *APND*, definimos el lenguaje aceptado por M, notado $L(M)$, como: $L(M) = \{ w \in \Sigma^* \mid w$ es aceptada por $M \}$

Definición 6.43: *Conjunto de lenguajes aceptados por APND*
El conjunto de los lenguajes aceptados por *APND*, notado $\mathcal{L}(APND)$, se define como:
$\mathcal{L}(APND) = \{ L \mid \exists M \ APND$ con $L(M) = L \}$

Ejemplo 6.44:
Cualquier autómata finito se puede considerar como un *APND* que nunca opera en su pila.
Sea $M = (K, \Sigma, \Delta, s, F)$ un *AF*. Sea $M' = (K, \Sigma, \Gamma, \Delta', s, F)$ un *APND* con $\Delta' = \{ ((p, u, \varepsilon), (q, \varepsilon)) \mid (p, u, q) \in \Delta \}$.
Entonces $M \equiv M'$. □

Ejemplo 6.45:
Sea $L = \{ wcw^R \mid w \in \{a, b\}^* \}$.
Sea $M = (K, \Sigma, \Gamma, \Delta, s, F)$ con:

$K = \{ s, f \}$
$\Sigma = \{ a, b, c \}$
$\Gamma = \{ a, b \}$
$F = \{ f \}$
$\Delta = \{ ((s, a, \varepsilon), (s, a)),$
$((s, b, \varepsilon), (s, b)),$
$((s, c, \varepsilon), (f, \varepsilon)),$
$((f, a, a), (f, \varepsilon)),$
$((f, b, b), (f, \varepsilon)) \ \}$

Tenemos que $L(M) = L$.

Veamos una computación de aceptación del autómata para la cadena de entrada *abbcbba* :

Estado	Por leer	Pila	Aplicamos
s	*abbcbba*	ε	*((s, a, ε), (s, a))*
s	*bbcbba*	*a*	*((s, b, ε), (s, b))*
s	*bcbba*	*ba*	*((s, b, ε), (s, b))*
s	*cbba*	*bba*	*((s, c, ε), (f, ε))*
f	*bba*	*bba*	*((f, b, b), (f, ε))*
f	*ba*	*ba*	*((f, b, b), (f, ε))*
f	*a*	*a*	*((f, a, a), (f, ε))*
f	ε	ε	-

□

Ejemplo 6.46:

Sea $L = \{ ww^R \mid w \in \{a, b\}^* \}$.

Sea $M = (K, \Sigma, \Gamma, \Delta, s, F)$ donde:

$K = \{ s, f \}$

$\Sigma = \{ a, b \}$

$\Gamma = \{ a, b \}$

$F = \{ f \}$

$\Delta = \{ ((s, a, \varepsilon), (s, a)) ,$
$((s, b, \varepsilon), (s, b)) ,$
$((s, \varepsilon, \varepsilon), (f, \varepsilon)) ,$
$((f, a, a), (f, \varepsilon)) ,$
$((f, b, b), (f, \varepsilon)) \}$

Tenemos que $L(M) = L$.

Veamos una computación de aceptación del autómata para la cadena de entrada *abbbba* :

Estado	Por leer	Pila	Aplicamos
s	*abbbba*	*ε*	*((s, a, ε), (s, a))*
s	*bbbba*	*a*	*((s, b, ε), (s, b))*
s	*bbba*	*ba*	*((s, b, ε), (s, b))*
s	*bba*	*bba*	*((s, ε, ε), (f, ε))*
f	*bba*	*bba*	*((f, b, b), (f, ε))*
f	*ba*	*ba*	*((f, b, b), (f, ε))*
f	*a*	*a*	*((f, a, a), (f, ε))*
f	*ε*	*ε*	-

Veamos una computación de rechazo del autómata para la cadena de entrada *abbbba* :

Estado	Por leer	Pila	Aplicamos
s	*abbbba*	*ε*	*((s, a, ε), (s, a))*
s	*bbbba*	*a*	*((s, ε, ε), (f, ε))*
f	*bbbba*	*a*	-

□

Proposición 6.47:
$L \in L.2 \Leftrightarrow L \in L(APND)$

6.7 Bombeo

Hasta ahora si queremos mostrar que un lenguaje es de contexto libre sólo tenemos que buscar una *GCL* que lo genere o un *APND* que lo acepte. Pero el problema surge cuando queremos demostrar que un lenguaje dado no es de contexto libre, ya que tendríamos que demostrar que no existe ninguna *GCL* (o *APND*) que lo represente.

Dado que esto es difícil lo que haremos es establecer alguna condición (necesaria) que cumpla todo lenguaje de contexto libre. Así, si demostramos que un lenguaje no cumple dicha condición (de contexto libre) hemos demostrado que no es de contexto libre.

El lema del bombeo que vamos a ver es el más usado como condición necesaria para demostrar que un lenguaje no es de contexto libre.

En primer lugar, de forma similar a como hicimos para los lenguajes regulares, realizamos una definición previa.

Definición 6.48: *Condición de bombeo de contexto libre (CBCL)*

Sea $L \subseteq \Sigma^*$.

Diremos que L cumple la condición de bombeo de contexto libre (cumple *CBCL*) sii $\exists n \in \mathbb{N} \mid \forall x \in L$ con $|x| \geq n$, $\exists u,v,w,y,z \in \Sigma^*$ tales que:

1) $x = uvwyz$
2) $|vwy| < n$
3) $|vy| > 0$
4) $\forall m \geq 0 \quad uv^m wy^m z \in L$

Lema del bombeo para $L.2$:

$L \in L.2 \Rightarrow L$ cumple *CBCL*

La demostración de este lema es similar a la del bombeo para $L.3$, sólo que se realiza con *GCL* (árboles de derivación) en lugar de con *AFD*.

Para utilizar este lema del bombeo para demostrar que un lenguaje no es de contexto libre tenemos que demostrar que el lenguaje no cumple la *CBCL*, es decir, que:

$\forall n \in \mathbb{N}$, $\exists x \in L$ con $|x| \geq n \mid \forall u,v,w,y,z \in \Sigma^*$ no se cumplen conjuntamente las cuatro condiciones de la *CBCL*.

La demostración de lo anterior se hace por reducción al absurdo:

Suponemos que $\exists u,v,w,y,z \in \Sigma^*$ tales que cumplen conjuntamente las cuatro condiciones de la *CBCL*, y llegamos a una contradicción.

Lo único que tenemos que hacer es elegir adecuadamente una $x \in L$ que nos lleve a dicha contradicción.

Ejemplo 6.49:

Sea $L = \{ a^n b^n c^n \mid n \geq 0 \}$.

Demostraremos que no es un lenguaje de contexto libre.

Para ello debemos demostrar que:

$\forall n \in \mathbb{N}$, $\exists x \in L$ con $|x| \geq n$ | $\forall u,v,w,y,z \in \Sigma^*$ no se cumplen conjuntamente las cuatro condiciones de la *CBCL*.

Así, sea $n \in \mathbb{N}$, y sea $x = a^n b^n c^n \in L$, $|x| = 3n \geq n$

Supongamos que $\exists u,v,w,y,z \in \{a,b,c\}^*$ | cumplen conjuntamente las cuatro condiciones de la *CBCL*:

1) $x = uvwyz$

2) $|vwy| < n \Rightarrow \exists \sigma \in \{a,b,c\} \mid |vwy|_\sigma = 0$
(↑ *por 1)*)

3) $|vy| > 0 \Rightarrow |v| > 0 \vee |y| > 0$

4) $\forall m \geq 0,\ uv^m wy^m z \in L \Rightarrow uv^0 wy^0 z \in L$

Pero $uv^0 wy^0 z = uwz \notin L$ ya que (por *3)*) $|v| > 0 \vee |y| > 0 \Rightarrow$ La *CBCL* no se cumple $\Rightarrow$ L no es de contexto libre.

Nota: También podemos demostrarlo si en el paso *4)* usamos $m>1$. □

CAPÍTULO II

Computabilidad

Tema 7: Introducción

Tema 8: La Máquina de Turing

Tema 9: Funciones Recursivas

Tema 10: El lenguaje *WHILE*

Tema 11: Teorema de Equivalencia

Tema 12: Universalidad

Tema 13: Limitaciones formales de la computación

7
Introducción

7.1 Antecedentes históricos

En este apartado realizaremos un breve viaje por los antecedentes históricos del concepto de algoritmo, desde los primeros que hoy consideraríamos como tales hasta su definición formal como objeto de estudio ya en el siglo XX.

No incluiremos en este viaje los antecedentes de los objetos físicos (artilugios, máquinas y posteriormente computadores) que han ido creándose a lo largo del tiempo para ayudarnos a aplicar dichos algoritmos (es decir, la habitualmente denominada "historia de los computadores").

Primeros algoritmos

El reconocimiento de que ciertos grupos de cosas cotidianas tienen en común una propiedad abstracta, a la que nosotros denominamos número, puede considerarse ya una idea matemática. Hay quienes sostienen que dicho descubrimiento debió de ser muy temprano, una especie de conciencia gradual que pudo haberse producido dentro del desarrollo cultural humano tan tempranamente al menos como el uso del fuego hace unos 400.000 años. Si para apoyar dicha afirmación sólo hay ligeros indicios antropológicos, lo que es indudable es que la idea de número es muy anterior a la escritura (la cual, en el sentido moderno, surgió hace aproximadamente 5.200 años) ya que, por ejemplo, se han encontrado huesos de animales de hace unos 30.000 años con muescas realizadas por el hombre, en distintas series, agrupadas de cinco en cinco. Ello demuestra que la idea de número (y su representación) estaba ya asentada en aquel tiempo.

Si aceptamos (con las reservas oportunas y de modo informal) que una serie de actos encaminados a realizar un cálculo es un algoritmo, entonces, las acciones de correspondencia biunívoca utilizadas para establecer el número de cosas de un grupo desconocido utilizando otro grupo de número conocido, se podría decir que fueron los primeros algoritmos de la humanidad. Curiosamente, muchos milenios después, Cantor utilizó (como algunos lectores ya conocen) el

concepto de biyección para su teoría de los cardinales (transfinitos) de los conjuntos infinitos (en una especie de vuelta a los orígenes).

Avanzando en el tiempo hasta los primeros algoritmos de los cuales hay constancia documental nos encontramos con el nacimiento de las civilizaciones que se caracterizaron por el uso de los metales (bronce), la rueda y la escritura. Dicho nacimiento tuvo lugar en los grandes valles fluviales de los ríos Eúfrates y Tigris (Mesopotamia), Nilo (Egipto), Indo (India) y Yangtzé (China), siendo al parecer Mesopotamia el primer lugar, durante el cuarto milenio antes de Cristo, donde una civilización de este tipo surgió.

Hace 4.500 años ya hay constancia de que en Mesopotamia los Babilonios (término genérico para varios pueblos de la zona) utilizaban un sistema de numeración. Éste evolucionó hasta llegar a ser un sistema de numeración posicional, sexagesimal (base 60) y sin el cero (si bien muy posteriormente fue introducido), teniendo también "decimales" sexagesimales.

Conocían ya, al menos, las cuatro operaciones básicas. Se ha conservado en escritura cuneiforme una división que calcula 1.152.000 entre 7 y da como resultado 164.571 y de resto de 3. La tablilla de arcilla se remonta al año 2650 a.C., siendo el testimonio escrito de la división (y por tanto, la aplicación de un algoritmo) más antigua que se conoce en la historia. Es una división de cierta complejidad que se realizó en Mesopotamia hace más de 4.500 años (casi tan antigua como el diluvio mesopotámico).

También conocían el algoritmo para el cálculo por aproximaciones sucesivas de raíces cuadradas, posteriormente conocido como algoritmo de Arquitas, de Herón de Alejandría o de Newton (si bien quizás deberíamos llamarlo algoritmo babilónico). Utilizándolo calcularon (en sexagesimal) un valor para la raíz cuadrada de dos de 1,414222, el cual difiere del valor exacto menos de 0,000009. Incluso hay constancia de que hace 4.000 años ya resolvían ecuaciones cuadráticas e incluso algunas cúbicas. En algunos aspectos, sobre todo el algebrista, fue la matemática más avanzada hasta el renacimiento.

Desplazándonos hacia el oeste nos encontramos que hace cerca de 5.000 años en Egipto ya habían desarrollado un sistema de numeración no posicional, sin el cero, y estructurado en una escala numérica basada en el diez. Hace 4.000 años usaban fracciones unitarias (es decir, inversos de naturales), y la fracción 2/3, para descomponer otras fracciones en ellas. Su uso llegaría hasta la edad media. Por medio de duplicaciones y fracciones unitarias realizaban ya entonces operaciones de suma, resta, multiplicación y división. Utilizaban la regla de tres

para resolver algunos problemas y llegaron a resolver algunos casos de ecuaciones lineales de primer orden.

Otras civilizaciones fueron tomando el testigo y realizando sus propias aportaciones, si bien la historia de las matemáticas (y por tanto la de los algoritmos) no es continua, ni temporal ni espacialmente, y muy a menudo poco documentada. Veamos sólo dos pinceladas relacionadas con algoritmos. Hace algo más de 1.500 años en China, Tsu Ch´ung-Chih, calculó (usando un algoritmo) la siguiente acotación del valor de π : $3{,}1415927 > \pi > 3{,}1415926$. Hace aproximadamente 1.100 años en la India ya estaba asentado un sistema de numeración que aunaba tres características que lo convertían en la mayor contribución de la India al desarrollo de la matemática: era un sistema de numeración en base diez, era posicional y tenía el cero (ésta contribución superaba a la que la India había realizado 400 años antes, con la elaboración de las tablas de la función trigonométrica seno). Este sistema de numeración sería heredado por los árabes y posteriormente por Occidente, facilitando en gran medida el cálculo (aplicación de un algoritmo) con lápiz y papel.

Origen del vocablo algoritmo

En el siglo VIII, el califa de Bagdad Hârûn al-Rasîd fundó “La casa de la Sabiduría”, impulsada luego por su hijo el califa al-Ma'mûn (786-833), como un lugar para el desarrollo y protección del saber y los sabios. Durante muchos años fue el foco intelectual del mundo, pasando por ella la mayoría de los pensadores de aquella época. Entre ellos estuvo uno con gran importancia desde el punto de vista algorítmico: Abu Ja'far Mohammed ibn Musa al-Khowârizmî (780-850) (padre de Ja´far, Mohammed, hijo de Moisés, natural de Khowârizm; pequeña ciudad antes persa y en la actualidad rusa). De su nombre procede la palabra algoritmo cuya forma original era algorismo, que se vio corrompida por la influencia de la palabra aritmética (también de aquí procede la palabra guarismo, sinónimo de cifra).

Polifacético como sus compañeros de “La casa de la Sabiduría” tiene contribuciones en astronomía, geografía e historia. Entre ellas está un libro que dio origen a la palabra álgebra: “Kitab al jabr w'alal-muqabalah” (reglas de restauración y reducción). La finalidad del álgebra de Al-Khowârizmî era presentar los métodos más útiles y generales para resolver ciertos problemas algebraicos. Sus métodos para hacer las operaciones elementales son

simplificaciones de métodos anteriores, pero aún así son difíciles de hacer con lápiz y papel. Dos siglos después, Al-Uqlîdisî, matemático de Damasco, adaptó los algoritmos de Al-Khowârizmî a métodos más apropiados para realizarlos con lápiz y papel.

Como curiosidad señalar que el significado de álgebra como restauración y reducción "de miembros" era de uso común todavía en el siglo XVI en España, así en El Quijote se usa al término algebrista en el sentido de traumatólogo (reductor de miembros).

Procedimientos generales

Sabemos que la matemática mesopotámica influyó en los árabes y a través de ellos llegó a España y desde aquí al resto de Europa. De esta forma, los árabes inspiraron a Raimundo Lulio (1235-?) a partir sin embargo de una motivación religiosa. Raimundo fue en peregrinación a Tierra Santa. A su vuelta decidió dedicarse a predicar la doctrina cristiana entre los árabes para lo cual pasó nueve años estudiando lengua árabe y debió entrar en contacto con la obra de los matemáticos árabes.

Los métodos algorítmicos de dichas obras le inspiraron, hacia el 1300, su obra "Ars Magna", en la que describe un procedimiento general de base combinatoria para hallar todas las "verdades" (la motivación del libro era religiosa: la búsqueda del algoritmo universal que resolvería todos los problemas y por tanto acercaría a Dios). Considerados desde un punto de vista sensato, los procedimientos aducidos por Lulio no tienen mucho valor. Lo importante sin embargo es que concibió una idea ciertamente espléndida, la de procedimiento. El trabajo de Lulio tuvo una gran influencia en la matemática posterior, y así, doscientos años después, Cardano (1545) titula su libro sobre algoritmos algebraicos remedando el título de la obra de Lulio: "Artis magnae seu de regulis algebraicis liber unus".

El mismo objetivo de Raimundo Lulio de reducir a cálculos la determinación de todas las verdades, en este caso lógicas, fue perseguido también por Gottfried Leibnitz (1646-1716), bien conocido por sus aportaciones al Análisis Matemático.

Del estudio de los trabajos de Lulio determinó que había dos conceptos que se deberían considerar por separado: El "Ars inveniendi" o procedimiento de generación y el "Ars iudicandi" o procedimiento de decisión (nótese que, al

menos informalmente, estos dos conceptos se pueden asociar con los de enumerable y decidible que veremos más adelante). Pretendía desarrollar un "calculus ratiocinator" para decidir las verdades lógicas. Éste perseguía dos objetivos: Uno era construir un lenguaje formal lo más amplio posible para expresar cualquier enunciado, incluyendo por supuesto a la aritmética; el otro era llegar a un procedimiento tal que si una persona opina que el enunciado *A* es verdad y otra que el enunciado *no-A* es verdad, para ponerse de acuerdo sólo necesitarían realizar cálculos que darían la razón a uno de los dos.

El primer objetivo, como veremos a continuación, fue más tarde alcanzado con los desarrollo de la lógica y el primer lenguaje formal construido por G. Frege en 1879. El segundo objetivo se mostró inalcanzable con los trabajos de Gödel y Turing entre 1931 y 1937.

La lógica

Para poder determinar qué se puede resolver de forma algorítmica y qué se puede resolver eficientemente se necesitaba de una expresión más precisa, que vino de la mano de la lógica, la cual podemos decir de manera informal que se dedica al estudio del razonamiento válido.

La silogística de Aristóteles (384 a.C.-322 a.C.) fue un primer intento en esta dirección, pero es en el siglo XIX cuando se produce el desarrollo de la lógica moderna. Los métodos de cálculo de George Boole (1815-1869) se constituyeron, en gran medida, de acuerdo con los métodos usuales del álgebra, siendo estos los primeros pasos hacia una formalización de la lógica. Tras diversos desarrollos a lo largo del siglo XIX, a finales de siglo Gottlob Frege (1848-1925) desarrolla un sistema suficientemente amplio como para proceder a una formalización que es base para el desarrollo posterior. En su artículo "Begriffschrift" de 1879 presenta el que se puede considerar primer lenguaje formal, en el cual se pueden expresar enunciados tales como completitud (si todos los enunciados son demostrables) o decidibilidad (si existen algoritmos para demostrar los enunciados). B. Russell fue quién reconoció su importancia y divulgó la obra de Frege. Su libro "Principia Mathematica" (Whitehead & Russell, 3 vols. 1910-1913) está basado en la obra de Frege.

En paralelo a Frege, Guiseppe Peano (1858-1932) escribió "Arithmeticae Principia. Nova methodo exposita", donde axiomatiza la aritmética de primer orden y de donde se deduce toda la teoría de números naturales. Este fue el

modelo que eligió Kleene, vía Gödel, para la formalización de la teoría de la recursividad (como veremos más adelante).

Al final de esta época está bien establecido que gran parte de la matemática puede ser deducida con la ayuda de un cálculo lógico. Se puede observar que todos estos desarrollos perseguían reducir los procesos de demostración a meros cálculos o procedimientos mecánicos que no dependan del ejecutor.

En el entorno científico de la época, tras la gran cantidad de desarrollos de la matemática en los siglos anteriores, y la creación de la teoría de conjuntos, aparecen tres grandes escuelas de pensamiento sobre la fundamentación de la matemática: la logicista, encabezada por los matemáticos ingleses Russell y Whitehead, la intuicionista, con el holandés Brower a la cabeza y la escuela formalista o axiomática de Hilbert.

La tesis logicista es que la Matemática es una rama de la lógica. Las nociones matemáticas han de ser definidas en términos de nociones lógicas, y los teoremas de la matemática han de ser demostrados como teoremas de lógica.

La escuela intuicionista defiende la no aceptación de la ley del tercero excluido para conjuntos infinitos (así por ejemplo, si en un dominio infinito D al suponer que todo miembro de D no tiene la propiedad P se alcanza una contradicción, entonces se puede deducir que existe un elemento de D que tiene dicha propiedad. Este razonamiento de tipo existencial no es aceptado por la escuela intuicionista).

En tercer lugar, el programa de la escuela formalista fue esbozado por Hilbert (1862-1943) en 1904, donde se propone "formular la matemática clásica como una teoría axiomática formal, y deberá demostrarse que esta teoría es consistente, esto es, libre de contradicción". Este programa fue acometido por Hilbert y su equipo desde 1920. El problema más grave era la demostración de la no contradicción.

En este ámbito se sitúa el problema de la decisión o "Entscheidungsproblem" como uno de los más ambiciosos de la metamatemática. En matemáticas, hay cuestiones generales tales que para cualquier instancia particular se puede responder mediante un método efectivo (es decir, objetivo y con unos requerimientos finitos de espacio y tiempo) que aplicado a ese caso nos dará una respuesta correcta: SÍ o NO. Así por ejemplo, hay métodos para saber, dados dos polinomios con coeficientes enteros, si uno es factor del otro. Basta realizar la división y observar si el resto es cero. Esto se puede hacer de manera mecánica y en un número finito de pasos. Un método de

este tipo se denomina un procedimiento de decisión. El problema de la decisión es determinar si existen tales métodos para saber si una fórmula es válida o no en un lenguaje formalizado. Este problema se asocia habitualmente con Hilbert por los grandes esfuerzos que dedicó a su resolución, pero aparece anteriormente en Schöder y en Löwenheim.

Hay que tener en cuenta que "método" o "procedimiento" era aún un concepto informal. En esa época había cierto consenso en lo que se podía considerar como método, en el sentido de usar un espacio finito, usar un tiempo finito y no dependencia de quién aplica el método (objetivo).

Siguiendo con el programa de trabajo, Hilbert y Ackermann delimitaron, hacia 1928, de un modo claro la lógica de primer orden y presentaron un cálculo deductivo para ella. Se plantearon la cuestión de si ese cálculo deductivo era semánticamente suficiente, esto es, si permitía deducir todas las fórmulas válidas. En su libro "Elementos de Lógica Teórica" (1928) manifiestan explícitamente que aún no habían encontrado respuesta a esa pregunta.

Dos años después de la publicación del libro de Hilbert y Ackermann, el austriaco Kurt Gödel (1906-1978) publica su trabajo sobre "La suficiencia de los axiomas del cálculo lógico de primer orden" (1930), donde demuestra que, mediante los axiomas y las reglas de inferencia de la lógica, en el ámbito de la lógica de primer orden, toda fórmula válida es deducible (es decir, es un conjunto recursivamente enumerable).

Con esto el programa de Hilbert obtenía un primer y esperanzador éxito. Sin embargo Gödel continuó sus trabajos y al año siguiente publicó su trabajo "Sobre las sentencias formalmente indecidibles de los Principia Mathematica y sistemas afines" en el cual demostraba la imposibilidad de llevar a cabo el programa de Hilbert (recuérdese que el objetivo era desarrollar sistemas formales y demostrar su consistencia). Gödel demostró que todos los sistemas formales de la matemática clásica (incluidos el sistema de los Principia Mathematica, la aritmética de Peano, la teoría axiomática de conjuntos, y en general, cualquier sistema formal que cumpliese ciertas condiciones básicas de aceptabilidad) son incompletos o contradictorios (es decir, el conjunto anterior de fórmulas válidas, aunque enumerable, no es recursivamente decidible). Además esta incompletitud no tiene remedio: por muchos axiomas que añadamos, el sistema sigue siendo incompleto. De aquí se deduce también la indecidibilidad de toda lógica de orden mayor que uno.

También demostró que es imposible probar la consistencia de un sistema formal dentro del propio sistema (se puede probar desde una teoría mas potente que el propio sistema formal, pero eso es de dudosa utilidad).

En este trabajo, Gödel define y utiliza las funciones recursivas primitivas (que él llama recursivas) de las que afirma que: "... tienen la importante propiedad de que para cada conjunto dado de argumentos el valor de la función puede computarse mediante un procedimiento finito." (nótese la referencia a procedimiento; ya nos vamos acercando a la formalización del concepto de algoritmo). Peter y Ackermann demuestran que las funciones recursivas (primitivas) definidas por Gödel son insuficientes para formalizar el concepto de "procedimiento efectivo".

1936: formalización del concepto de algoritmo

En abril de 1935, Alonzo Church presentó una comunicación donde formalizaba las funciones recursivas en términos del λ-cálculo y enunció la equivalencia entre su clase de funciones y las calculables por procedimientos efectivos (hoy conocida como tesis de Church-Turing). Cuando Church expuso su tesis, Stephen C. Kleene se propuso demostrar su falsedad mediante diagonalización (técnica que algunos lectores ya conocen) sobre toda la clase de las funciones definibles por medio del λ-cálculo. Al ver la imposibilidad de conseguirlo, cambió de opinión y fue Kleene quién le dio el nombre de tesis de Church.

En septiembre de 1935 Kleene presentó en una comunicación los resultados fundamentales de su tesis doctoral en la cual, siguiendo las funciones que usó Gödel en su demostración del teorema de incompletitud, definió las funciones recursivas primitivas y las funciones recursivas generales (también denominadas μ-recursivas). Demostraba la existencia de conjuntos indecidibles, definía los recursivamente enumerables y demostraba el teorema de la Forma Normal que lleva su nombre (este teorema tenía un aspecto formal difícil, pero en terminología actual se podría enunciar como sigue: cada programa admite uno equivalente usando bucles FOR y un sólo bucle WHILE).

En 1936 las dos comunicaciones se publicaron en forma de artículos. En ambos casos dieron caracterizaciones formales de la noción de función efectivamente calculable y demostraron la existencia de problemas indecidibles pero sin exhibir ninguno.

En 1936 Alan Turing (1912-1954) da una nueva formalización definiendo lo que hoy se conoce como máquina de Turing. Demostró la indecidibilidad del problema de la parada y del problema de la decisión por métodos constructivos (usando lo que hoy se conoce como máquina de Turing Universal); y, en un trabajo posterior, la equivalencia de su modelo con los modelos de calculabilidad vía λ-cálculo. También definió máquina con oráculo y la reducibilidad entre problemas para comparar dificultades de resolución. Lo distintivo de este modelo es que define el concepto de "paso de cómputo", el cual es fundamental para el desarrollo de la teoría de la complejidad.

El mismo año, Emil Post (1847-1954) define la máquina que lleva su nombre. Muy parecido al modelo de Turing, lo publicó en un artículo con pocos meses de diferencia, pero escrito de manera totalmente independiente (lo cual tuvo que justificar ante la comunidad científica).

Las evidencias a favor de la tesis (de Church) de que todos estos modelos capturan la idea intuitiva de algoritmo son de diversos tipos:

- Equivalencia entre los distintos modelos (que ya hemos citado).
- El concepto de máquina de Turing está diseñado explícitamente para reproducir las operaciones que un calculador humano puede realizar.
- Evidencia heurística:
  -- Toda función particular efectivamente calculable y toda operación para definir de manera efectiva una función a partir de otras, han resultado calculables por máquinas de Turing.
  -- La exploración de varios métodos de los que se podría esperar que condujesen a una función fuera de la clase de las Turing-calculables han mostrado, o bien que el método no conduce fuera de la clase, o bien que la nueva función obtenida no puede ser considerada como efectivamente calculable.

Otros modelos formales

Posteriormente a los modelos de Church, Kleene, Turing y Post, han aparecido otros con diferentes orientaciones. Observamos como la aparición de estos modelos ha ido paralela al desarrollo de los ordenadores y los lenguajes de programación. En primer lugar citaremos el modelo URM (Unlimited Registers Machine) definido por Shepherdson y Sturgis (1963). En este modelo se evitan los tediosos recorridos por la cinta de la máquina de Turing mediante la

introducción de registros en los que se almacenan datos, sin importar su tamaño. Junto con esta forma de almacenamiento está un adecuado juego de instrucciones que incluye sumar uno, restar uno, copiar de un registro a otro, salto incondicional y salto condicional. Los modelos derivados de URM, como el conocido modelo RAM (Random Access Machine) de 1974, tienen un juego de instrucciones más amplio, que se parecen mucho a las de un lenguaje ensamblador.

El siguiente paso en los modelos fue pasar de registros a variables, de forma que una "máquina" pasa a tener aspecto de programa. Inicialmente se usaron lenguajes no estructurados, con variables de entrada, variables de uso interno y variables de salida. Posteriormente la programación estructurada llegó al ámbito de la Informática Teórica, surgiendo modelos de cómputo basados en ella. Uno de ellos es el lenguaje "*WHILE*", que tiene sentencias de asignación y una sola de control que le da nombre al lenguaje, el bucle *WHILE*.

De todos los modelos citados en esta introducción histórica, tres son los que veremos en los siguientes temas: la máquina de Turing, las funciones recursivas, y el lenguaje *WHILE*.

7.2 Los conceptos intuitivos de la computabilidad

Uno de los objetivos al definir un modelo de cómputo es poder definir formalmente los conceptos intuitivos de función computable, conjunto decidible y conjunto enumerable. En los tres temas siguientes veremos estos conceptos formalizados según cada uno de los tres modelos que iremos definiendo. Para una mejor comprensión de estas formalizaciones es conveniente tener claros los conceptos intuitivos que queremos formalizar.

Necesitamos en primer lugar una aproximación a la idea de algoritmo. Así, podemos decir que un *algoritmo* es una secuencia finita de instrucciones precisas (no ambiguas) que, para un cierto tipo de entrada, o bien nos devuelve (computa en un tiempo finito) una salida o bien no termina nunca. Si un algoritmo siempre, para su tipo de entrada, nos devuelve en un tiempo finito una salida, entonces decimos que es un *algoritmo conclusivo*.

Diremos que una función es computable si existe un algoritmo que para cualquier argumento de la función como entrada nos devuelve el valor de la función como salida. Si la función para cierto argumento no toma valor (es

decir, es una función parcial) entonces el algoritmo para ese argumento como entrada no termina nunca. De aquí se deduce que si una función computable es total entonces el algoritmo que la computa es un algoritmo conclusivo.

Sea A un conjunto de cadenas sobre un alfabeto y B un subconjunto de A. Obsérvese que entre otros muchos conjuntos, A podría ser el conjunto de los naturales, o el conjunto de los vectores de naturales de cierto tamaño, o incluso el conjunto de todos los vectores de naturales.

Diremos que B es un conjunto decidible si existe un algoritmo conclusivo que dado un elemento de A como entrada nos dice siempre (en un tiempo finito al ser el algoritmo conclusivo) si dicho elemento pertenece o no al conjunto B.

Diremos que B es un conjunto enumerable si existe un algoritmo que dado un elemento de A como entrada nos devuelve una respuesta afirmativa si y sólo si dicho elemento pertenece al conjunto B.

La diferencia entre decidible y enumerable es que en el primer caso siempre tenemos respuesta del algoritmo, mientras que en el segundo sólo podemos asegurar que el algoritmo terminará si el elemento en cuestión pertenece al conjunto. De aquí se deduce que todo conjunto decidible es enumerable pero no a la inversa.

Habrá lectores que ya han tenido contacto con estos dos conceptos (conjunto decidible y conjunto enumerable), aunque en su momento no los definieran de ese modo (si el lector no ha tenido dicho contacto puede tenerlo consultando el libro *Teoría de Autómatas y Lenguajes Formales I*, Gonzalo Ramos Jiménez, 2005). Así, cuando uno se formula preguntas básicas sobre los tipos de lenguajes de la Jerarquía de Chomsky, entre ellas la de pertenencia y si hay algún algoritmo conclusivo que la responda, lo que está preguntándose es si los lenguajes de un tipo dado son conjuntos decidibles o no.

Por ejemplo, todo lenguaje regular es decidible, ya que existe un autómata finito determinista (es decir, un algoritmo conclusivo) que lo decide (también llamado dispositivo reconocedor). Lo mismo sucede con los lenguajes de contexto libre y con los lenguajes sensibles al contexto, ya que tienen sus respectivos dispositivos reconocedores (algoritmos conclusivos).

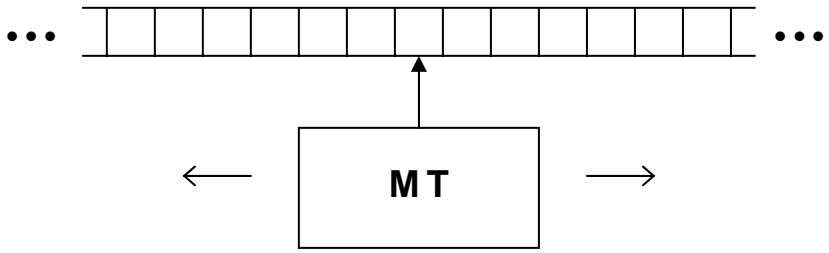
Sin embargo, sólo podemos asegurar que todo lenguaje con estructura de frase es enumerable, ya que sabemos que para dicho tipo de lenguajes no existe ningún tipo de dispositivo reconocedor. Es decir, no tenemos seguridad de poder encontrar un algoritmo conclusivo para cada lenguaje con estructura de frase que lo decida, pero si un algoritmo que nos permita afirmar que es enumerable.

8
La Máquina de Turing

8.1 Concepto informal

Podemos imaginar una *Máquina de Turing* (MT) como una máquina ideal que opera con el material de computación que se describe a continuación.

Por una parte tenemos una *cinta* infinita dividida en cuadrados, con sentidos izquierdo y derecho diferenciados. En cada cuadrado de la cinta sólo puede haber un símbolo de los que forman el llamado *alfabeto* de la MT, o bien un símbolo especial para señalar que el cuadrado está vacío. Además siempre habrá sólo un número finito de cuadrados ocupados en la cinta. Al contenido completo de la cinta se le denomina *expresión de cinta*. También tenemos un puntero que indica el *cuadrado escrutado* de la cinta, es decir, el único cuadrado cuyo contenido en un momento dado "ve" la MT.



Las únicas *instrucciones elementales* con las que trabaja una MT son:

- Escribir un símbolo del alfabeto de la MT en el cuadrado escrutado.
- Mover un cuadrado a la derecha el puntero.
- Mover un cuadrado a la izquierda el puntero.
- Parar.

Una MT está constituida por una serie de *estados* (uno de ellos destacado como *estado inicial*). En cada momento la MT está en uno de sus estados, y sólo en uno. Cada estado, cuando la MT está en él, tiene información para decidir, dado el contenido del cuadrado escrutado, qué instrucción elemental debe realizar y a qué nuevo estado debe transitar la MT.

De esta forma, antes de un *paso de cómputo*, el cual se produce siempre que nos encontramos con una instrucción elemental distinta de "parar", tenemos en el material de computación:

- una expresión de cinta
- un cuadrado escrutado

y en la MT:

- un estado actual (y por tanto una instrucción elemental).

Después del paso de cómputo tenemos en el material de computación:

- una nueva expresión de cinta (que puede ser la misma)
- un nuevo cuadrado escrutado (que puede ser el mismo)

y en la MT:

- un nuevo estado actual (y por tanto una nueva instrucción elemental, pudiendo ser los mismos tanto el uno como la otra).

8.2 Definición formal

Hay varias formas (todas ellas equivalentes) de definir una Máquina de Turing. Nosotros diferenciaremos entre la definición formal y la forma de representar una MT. Con ello pretendemos conseguir por un lado una definición rigurosa y por otro una representación clara y sencilla de una MT, por lo que la representaremos como una tabla, que es la forma más extendida y quizás también la de más fácil comprensión.

Definición 8.1: *Máquina de Turing (MT)*

Sea un alfabeto $\Sigma = \{ a_1, a_2, a_3, \ldots, a_n \}$ con $n \geq 1$ y con $r, l, h \notin \Sigma$.

Sea a_0 el símbolo ideal vacío, con $a_0 \notin \Sigma$ (habitualmente $a_0 = *$)

Sea $\Sigma' = \Sigma \cup \{ a_0, r, l, h \}$.

Una Máquina de Turing (MT) es una quíntupla $M = (K, q_0, \Sigma, \gamma, \delta)$ donde:

K es un conjunto finito no vacío de estados, con $K \subset \mathbb{N}$;

q_0 es el estado inicial ($q_0 \in K$);

Σ es un alfabeto (es decir, un conjunto finito no vacío de símbolos);

γ es una función de instrucción definida como $\gamma : K \times \Sigma \cup \{ a_0 \} \rightarrow \Sigma'$;

δ es una función de transición definida como $\delta : K \times \Sigma \cup \{ a_0 \} \rightarrow K$.

Nota: Si M está en el estado $q_j \in K$ y el símbolo leído es $a_i \in \Sigma$, entonces la instrucción elemental que realiza M es $\gamma(q_j, a_i) \in \Sigma'$, y el siguiente estado al que pasa M es $\delta(q_j, a_i) \in K$ (γ y δ son funciones totales).

Una MT $M = (K, q_0, \Sigma, \gamma, \delta)$ la representaremos con una matriz de cuatro columnas y $||K||(n+1)$ filas de la forma:

q_0	a_0	$\gamma(q_0, a_0)$	$\delta(q_0, a_0)$
q_0	a_1	$\gamma(q_0, a_1)$	$\delta(q_0, a_1)$
...	...	...	...
q_0	a_n	$\gamma(q_0, a_n)$	$\delta(q_0, a_n)$
q_1	a_0	$\gamma(q_1, a_0)$	$\delta(q_1, a_0)$
q_1	a_1	$\gamma(q_1, a_1)$	$\delta(q_1, a_1)$
...	...	...	...
q_1	a_n	$\gamma(q_1, a_n)$	$\delta(q_1, a_n)$
...	...	...	...
...	...	...	...
$q_{\|\|K\|\|-1}$	a_0	$\gamma(q_{\|\|K\|\|-1}, a_0)$	$\delta(q_{\|\|K\|\|-1}, a_0)$
$q_{\|\|K\|\|-1}$	a_1	$\gamma(q_{\|\|K\|\|-1}, a_1)$	$\delta(q_{\|\|K\|\|-1}, a_1)$
...	...	...	...
$q_{\|\|K\|\|-1}$	a_n	$\gamma(q_{\|\|K\|\|-1}, a_n)$	$\delta(q_{\|\|K\|\|-1}, a_n)$

donde:

$q_j \in K \subset \mathbb{N}$ con $j = 0, ..., ||K||-1$ cumpliéndose que $j \neq i \Rightarrow q_j \neq q_i$
$a_i \in \Sigma \cup \{ a_0 \}$ con $i = 0, ..., n$
$\gamma(q_j, a_i) \in \Sigma'$
$\delta(q_j, a_i) \in K$

Notas: El primer estado de la tabla (q_0) siempre es el inicial de la MT.

Tradicionalmente si $\gamma(q_j, a_i) = h$ entonces $\delta(q_j, a_i) = q_j$, aunque es por puro sentido estético, ya que en este caso (como veremos más adelante) un valor distinto no afectaría al funcionamiento de la MT.

Definición 8.2: *Expresión de cinta*
Una expresión de cinta de una MT $M = (K , q_0 , \Sigma , \gamma , \delta)$ es toda aplicación $E : \mathsf{Z} \to \Sigma \cup \{ a_0 \}$ que cumple que $\| \{ z \in \mathsf{Z} \mid E(z) \neq a_0 \} \| \in \mathsf{N}$.

Notas: Por convención se considera que el cuadrado z es el inmediato a la izquierda del $z+1$, por lo que tenemos una cinta de cálculo numerada e infinita a derecha e izquierda, donde $E(z)$ es el contenido del cuadrado de número z .
La condición impuesta a la aplicación E para ser expresión de cinta significa que, aunque la cinta es infinita en ambos sentidos, sólo hay un número finito de cuadrados que no están vacíos. Esta condición implica que la aplicación E siempre es representable.

Definición 8.3: *Configuración*
Dada una MT $M = (K , q_0 , \Sigma , \gamma , \delta)$, una configuración de M es toda terna (q, E, z) donde:
$q \in K$;
E es una expresión de cinta de M ;
$z \in \mathsf{z}$.

Nota: Representa el estado actual (q) en el que está la MT, el contenido completo actual de la cinta (E), y el *cuadrado escrutado* actual (z).

Definición 8.4: *Configuración inicial*
Dada una MT $M = (K , q_0 , \Sigma , \gamma , \delta)$, una configuración inicial de M es toda configuración de la forma (q_0 , E, z) .

Definición 8.5: *Configuración terminal, Configuración no terminal*
Dada una MT $M = (K , q_0 , \Sigma , \gamma , \delta)$, una configuración (q, E, z) es terminal sii $\gamma(q , E(z)) = h$.
En caso contrario decimos que es una configuración no terminal.

Nota: Una configuración puede ser a la vez inicial y terminal.

Definición 8.6: *Transitar directamente*

Sea $M = (K , q_0 , \Sigma , \gamma , \delta)$ una MT, y sean (q, E, z) y (q', E', z') configuraciones de M.

Diremos que M transita directamente de (q, E, z) a (q', E', z'), notado $(q, E, z) \vdash (q', E', z')$, sii:

$$
\begin{aligned}
& [\ q' = \delta(q , E(z))\ ] \\
& \qquad \wedge \\
& [\ [\ z' = z\text{-}1 \ \wedge \ \gamma(q , E(z)) = l\ ] \ \vee \\
& \quad [\ z' = z+1 \ \wedge \ \gamma(q , E(z)) = r\ ] \ \vee \\
& \quad [\ z' = z \ \wedge \ \gamma(q , E(z)) \in \Sigma \cup \{ a_0 \}\ ]\ ] \\
& \qquad \wedge \\
& E'(x) = \begin{cases} E(x) & \text{si } x \neq z \\ E(x) & \text{si } [\ x = z \ \wedge \ \gamma(q , E(z)) \in \{ l , r \}\ ] \\ \gamma(q , E(z)) & \text{si } [\ x = z \ \wedge \ \gamma(q , E(z)) \in \Sigma \cup \{ a_0 \}\ ] \end{cases}
\end{aligned}
$$

Notas: Si $(q, E, z) \vdash (q', E', z')$ entonces decimos que (q', E', z') es la *configuración siguiente* de la configuración (q, E, z).

En una MT para toda configuración no terminal siempre existe una y sólo una configuración siguiente.

En una MT para toda configuración terminal no existe configuración siguiente.

Definición 8.7: *Transitar en n pasos*

Sea $M = (K , q_0 , \Sigma , \gamma , \delta)$ una MT, y sean (q, E, z) y (q', E', z') configuraciones de M.

Diremos que M transita en n pasos de (q, E, z) a (q', E', z'), con $n>1$, notado $(q, E, z) \vdash^n (q', E', z')$, sii:

$\exists\ C_1 , C_2 , \ldots , C_{n-1}$ configuraciones de M |

$$(q, E, z) \vdash C_1 , \ C_1 \vdash C_2 , \ \ldots , \ C_{n-1} \vdash (q', E', z')$$

Diremos que M transita en 1 pasos de (q, E, z) a (q', E', z'), notado $(q, E, z) \vdash^1 (q', E', z')$, sii:

$$(q, E, z) \vdash (q', E', z')$$

Diremos que M transita en 0 pasos de (q, E, z) a (q', E', z') , notado $(q, E, z) \vdash^0 (q', E', z')$, sii:

$$(q, E, z) = (q', E', z')$$

Definición 8.8: *Transitar en al menos un paso*

Sea $M = (K , q_0 , \Sigma , \gamma , \delta)$ una MT, y sean (q, E, z) y (q', E', z') configuraciones de M .

Diremos que M transita en al menos un paso de (q, E, z) a (q', E', z') , notado $(q, E, z) \vdash^+ (q', E', z')$, sii:

$$\exists n>0 \mid (q, E, z) \vdash^n (q', E', z')$$

Definición 8.9: *Transitar*

Sea $M = (K , q_0 , \Sigma , \gamma , \delta)$ una MT, y sean (q, E, z) y (q', E', z') configuraciones de M .

Diremos que M transita de la configuración (q, E, z) a la (q', E', z') , notado $(q, E, z) \vdash^* (q', E', z')$, sii:

$$\exists n\geq 0 \mid (q, E, z) \vdash^n (q', E', z')$$

Definición 8.10: *Computación, Longitud de una computación*

Definimos computación de una MT, notada $C_0 \vdash C_1 \vdash C_2 \vdash ... \vdash C_n$, como cualquier secuencia finita de configuraciones $C_0 , C_1 , C_2 , ... , C_n$ tal que $C_i \vdash C_{i+1}$ donde $i = 0 , ... , n-1$ con $n\geq 0$.

La longitud de dicha computación es n .

Definición 8.11: *Computación bien iniciada, Computación terminada*

Diremos que una computación de una MT es una computación bien iniciada si su primera configuración es inicial.

Diremos que una computación de una MT es una computación terminada si su última configuración es terminal.

Definición 8.12: *Computación completa*

Una computación de una MT es completa sii es bien iniciada y terminada.

Antes de ver algunos ejemplos de MT y de su funcionamiento vamos a definir algunos conceptos relacionados con la descripción genérica de dicho funcionamiento.

Definición 8.13: *No pararse a partir de una configuración*
Sea M una MT, y sea C una configuración de M.
Diremos que M no se para a partir de C sii:

$$\forall n \geq 0 \quad \exists C' \text{ configuración de } M \mid C \vdash^{n} C'$$

Definición 8.14: *Pararse a partir de una configuración*
Sea M una MT, y sea C una configuración de M.
Diremos que M se para a partir de C sii:

$$\exists C' \text{ configuración terminal de } M \mid C \vdash^{*} C'$$

Concretamente diremos que M, a partir de C, se para en la configuración C'.

Esta última definición da lugar a otras tres para las cuales necesitamos además recordar algunos conceptos básicos sobre lenguajes formales.

Ya hemos mencionado que un alfabeto es un conjunto finito no vacío de símbolos. Una cadena (también llamada palabra) sobre un alfabeto es cualquier secuencia finita de símbolos del mismo, y su longitud (que se nota entre barras simples) es el número de símbolos de que está compuesta.

Dada una expresión de cinta E de una MT $M = (K, q_0, \Sigma, \gamma, \delta)$, y un cuadrado z que cumple que $E(z) \neq a_0$, llamaremos $w_{(z)}$ a la cadena sobre Σ a la que pertenece $E(z)$, es decir:

$$w_{(z)} = E(x)E(x+1)...E(z-1)E(z)E(z+1)...E(y-1)E(y)$$

donde:

$$x = max\{ n \leq z \mid E(n) = a_0 \} + 1$$
$$y = min\{ n \geq z \mid E(n) = a_0 \} - 1$$

y diremos que $w_{(z)}$ está situada detrás del cuadrado $x-1$, o delante del $y+1$.

Veamos ahora esas tres definiciones derivadas de la **8.14**.

Definición 8.15: *Pararse sobre una cadena*
Sea $M = (K, q_0, \Sigma, \gamma, \delta)$ una MT, y sea C una configuración de M.
Sea $C' = (q, E, z)$ una configuración terminal de M tal que $C \vdash^{*} C'$.
Si $E(z) \neq a_0$ entonces diremos que M, a partir de la configuración C, se para sobre la cadena $w_{(z)}$.

Definición 8.16: *Pararse detrás de una cadena*
Sea $M = (K , q_0 , \Sigma , \gamma , \delta)$ una MT, y sea C una configuración de M.
Sea $C' = (q, E, z)$ una configuración terminal de M tal que $C \vdash^* C'$.
Si $E(z) = a_0$ y $E(z-1) \neq a_0$ diremos que M, a partir de C, se para detras de la cadena $w_{(z-1)}$.

Definición 8.17: *Pararse detrás de la cadena vacía*
Sea $M = (K , q_0 , \Sigma , \gamma , \delta)$ una MT, y sea C una configuración de M.
Sea $C' = (q, E, z)$ una configuración terminal de M tal que $C \vdash^* C'$.
Si $E(n) = a_0 \ \forall n \leq z$ diremos que M, a partir de C, se para detrás de la cadena vacía.

Los últimos conceptos definidos están relacionados con la posible finalización de una computación. Sólo nos falta hacerlo en relación al inicio de la misma.

Definición 8.18: *Colocar una MT en un cuadrado con una expresión de cinta*
Sea $M = (K , q_0 , \Sigma , \gamma , \delta)$ una MT.
Sea E una expresión de cinta de M y z un entero.
Colocar M en el cuadrado z con la expresión de cinta E equivale a hacer la configuración $C_0 = (q_0 , E, z)$.

Definición 8.19: *Colocar una MT detrás de una cadena*
Sea $M = (K , q_0 , \Sigma , \gamma , \delta)$ una MT.
Sea $w_{(y)} = E(x)E(x+1)...E(y-1)E(y)$ una cadena sobre Σ.
Colocar M detrás de $w_{(y)}$ equivale a colocarla en el cuadrado $y+1$ con la expresión de cinta $E(z) = a_0 \ \forall z \in \mathbb{Z} - [x,y]$ (es decir, a excepción de la cadena $w_{(y)}$ la cinta está vacía).

Nota: Cual es en concreto el valor del entero y no es relevante, ya que una variación en dicho valor sólo supone un desplazamiento en la cinta, de manera que una computación sólo se diferenciará de otra desplazada en que las terceras componentes de sus correspondientes configuraciones se diferenciaran unas de otras en una constante entera (el desplazamiento).

8.3 Ejemplos de MT

Ejemplo 8.20:
Sea M la MT dada por la tabla:

0	*	|	*0*
0	|	*h*	*0*

Por tanto $M = (K, q_0, \Sigma, \gamma, \delta)$ con:

$K = \{ 0 \}$
$q_0 = 0$
$\Sigma = \{ | \}$ ($a_0 = *$, que es lo habitual)
γ: $\gamma(0, *) = |$
$\gamma(0, |) = h$
δ: $\delta(0, *) = 0$
$\delta(0, |) = 0$ □

Hemos comprobado el hecho de que la representación de una MT como una tabla nos proporciona su definición formal, por lo que de ahora en adelante cuando tengamos que dar una MT concreta lo haremos como una tabla.

Ejemplo 8.21:
Sea M la MT del ejemplo anterior.
Veamos el funcionamiento de M en algunos casos.

Si colocamos M en el cuadrado 13 con la expresión de cinta $E(z) = * \ \forall z \in \mathbb{Z}$ tenemos la siguiente computación:

$$(0, E, 13) \vdash\!\!\!- (0, E', 13)$$

donde E' es:

$E'(z) = * \quad \forall z \neq 13$
$E'(13) = |$

En esta computación la configuración $C_0 = (0, E, 13)$ es inicial, ya que $q_0 = 0$, y la $C_1 = (0, E', 13)$ es terminal, ya que $\gamma(0, E'(13)) = h$ al ser $E'(13) = |$ y $\gamma(0, |) = h$ (también es inicial, por la misma razón que lo es la C_0).
Por tanto la computación obtenida, de longitud uno, es bien iniciada (ya que C_0 es inicial), terminada (ya que su última configuración, la C_1, es terminal) y por tanto completa (ya que es bien iniciada y terminada).

Si ahora tenemos $w = |\,|\,|\,|$ situada delante del cuadrado -5, y colocamos M detrás de w tenemos la siguiente computación:

$$(0, E, -5) \vdash\!\!\!- (0, E', -5)$$

donde E es:

$E(z) = |\quad \forall n \in [-9,-6]$

$E(z) = *\quad \forall n \in \mathbb{Z} - [-9,-6]$

y E' es:

$E'(z) = |\quad \forall n \in [-9,-5]$

$E'(z) = *\quad \forall n \in \mathbb{Z} - [-9,-5]$

En esta computación la configuración $C_0 = (0, E, -5)$ es inicial, ya que $q_0 = 0$, y la $C_1 = (0, E', -5)$ es terminal, ya que $\gamma(0, E'(-5)) = h$ al ser $E'(-5) = |$ y $\gamma(0, |) = h$ (también es inicial, por la misma razón que lo es la C_0).

Por tanto la computación obtenida, al igual que la anterior, es de longitud uno, y también es bien iniciada (ya que C_0 es inicial), terminada (ya que su última configuración, la C_1, es terminal) y por tanto completa (ya que es bien iniciada y terminada).

Si al igual que antes tenemos $w = |\,|\,|\,|$ situada delante del cuadrado -5, y colocamos M en el cuadrado -6 tenemos la siguiente computación:

$$(0, E, -6)$$

donde E es:

$E(z) = |\quad \forall n \in [-9,-6]$

$E(z) = *\quad \forall n \in \mathbb{Z} - [-9,-6]$

La computación obtenida, de longitud cero, también es completa (ya que la única configuración que la compone es inicial y terminal). □

Ejemplo 8.22:

Sea M la MT dada por la tabla:

0	*	*l*	*0*
0	|	*r*	*1*
1	*	|	*2*
1	|	*r*	*1*
2	*	*	*2*
2	|	*r*	*3*
3	*	*h*	*3*
3	|	*r*	*3*

Si tenemos $w = |\,|\,|$ situada delante del cuadrado 0, y colocamos M detrás de la cadena w tenemos la siguiente computación:

$$(0, E, 0) \vdash\!\!\!- (0, E', -1) \vdash\!\!\!- (1, E^{2'}, 0) \vdash\!\!\!- (2, E^{3'}, 0) \vdash\!\!\!- (3, E^{4'}, 1)$$

donde E es:

$E(z) = |\quad \forall n \in [-3,-1]$

$E(z) = *\quad \forall n \in \mathbb{Z} - [-3,-1]$

$E' = E^{2'} = E$

$E^{3'}$ es:

$E^{3'}(z) = E^{2'}(z) \quad \forall z \neq 0$

$E^{3'}(0) = |$

$E^{4'} = E^{3'}$

En esta computación la configuración $C_0 = (0, E, 0)$ es inicial, ya que $q_0 = 0$, y la $C_4 = (3, E^{4'}, 1)$ es terminal, ya que $\gamma(3, E^{4'}(1)) = h$ al ser $E^{4'}(1) = *$ y $\gamma(3, *) = h$.

Por tanto la computación obtenida, de longitud cuatro, también es completa, ya que es bien iniciada (a ser C_0 inicial) y terminada (a ser C_4, su última configuración, terminal). □

Por la propia definición de una MT, en un paso de cómputo sólo puede o moverse (un cuadrado a la derecha o a la izquierda) o escribir en el cuadrado escrutado, pero no puede realizar las dos cosas a la vez en un solo paso. Esto significa que en una computación de una MT entre una configuración y su siguiente, si la expresión de cinta cambia (lo hará sólo en un cuadrado) entonces el cuadrado escrutado se mantiene el mismo, y si el cuadrado escrutado cambia (se incrementa o decrementa en una unidad) entonces la expresión de cinta se mantiene igual (véase el ejemplo anterior). Puede darse que se mantengan iguales la expresión de cinta y el cuadrado escrutado si la instrucción elemental de escritura coincide con el contenido anterior de dicho cuadrado.

Es fácil apreciar en el último ejemplo (**8.22**) que, a pesar de la simplicidad de la MT y de la computación presentadas, la representación de las configuraciones se ve dificultada por lo engorroso de la definición formal de las sucesivas expresiones de cinta. Para solucionarlo, al igual que hemos facilitado la representación de las MT por medio de una tabla, vamos a representar las expresiones de cinta (no muy complejas) como una pequeña sucesión de símbolos según el siguiente convenio de representación:

a_i	un cuadrado que contiene el símbolo $a_i \in \Sigma$
*	un cuadrado vacío (o lo que es lo mismo, que contiene a_0)
a	un cuadrado que contiene un símbolo (no vacío) cualquiera de Σ
~	un cuadrado, vacío o no
...	una secuencia finita de cuadrados vacíos (al menos uno)
*...	una secuencia infinita hacia la derecha de cuadrados vacíos
...*	una secuencia infinita hacia la izquierda de cuadrados vacíos
w	una secuencia finita de cuadrados no vacíos que forma la cadena w

Para no tener que numerar todos los cuadrados de la expresión de cinta simplemente distinguiremos el cuadrado escrutado subrayando el símbolo que representa su contenido.

De ahora en adelante las expresiones de cinta las representaremos usando este convenio.

A modo de comparación, veamos como queda la computación del ejemplo **8.22** cuando usamos el convenio.

Ejemplo 8.23:
Sea M la MT del ejemplo **8.22**.
Si de nuevo tenemos $w = |\,|\,|$ situada delante del cuadrado 0 , y colocamos M detrás de la cadena w tenemos la siguiente computación:

$$(0, ...*\,|\,|\,|\,\underline{*}...\,, 0) \vdash (0, ...*\,|\,|\,\underline{|}\,*...\,, -1) \vdash (1, ...*\,|\,|\,|\,\underline{*}...\,, 0) \vdash$$
$$(2, ...*\,|\,|\,\underline{|}\,*...\,, 0) \vdash (3, ...*\,|\,|\,|\,|\,\underline{*}...\,, 1)$$ □

Ya comentamos en el apartado anterior que en realidad el número concreto del cuadrado escrutado no es relevante (desplazamiento), sino que sólo es importante cual es el cuadrado escrutado en relación con el contenido de la cinta, es decir, la expresión de cinta.

Por ello en ocasiones, por simplicidad y siempre que mantengamos distinguido en la expresión de cinta el cuadrado escrutado, nos permitiremos eliminar la tercera componente de las configuraciones (es decir, el número del cuadrado escrutado).

De esta forma la computación anterior quedaría aún más simplificada:

$$(0, ...*\,|\,|\,|\,\underline{*}...) \vdash (0, ...*\,|\,|\,\underline{|}\,*...) \vdash (1, ...*\,|\,|\,|\,\underline{*}...) \vdash$$
$$(2, ...*\,|\,|\,\underline{|}\,*...) \vdash (3, ...*\,|\,|\,|\,|\,\underline{*}...)$$

Ejemplo 8.24:
Retomemos la MT M del ejemplo **8.22**, que recordamos viene dada por la tabla:

0	*	*l*	*0*
0	|	*r*	*1*
1	*	|	*2*
1	|	*r*	*1*
2	*	*	*2*
2	|	*r*	*3*
3	*	*h*	*3*
3	|	*r*	*3*

Veamos algunas computaciones más de esta MT.

Iniciada con la siguiente configuración: $(\ 0, ...*\underline{\perp}|\ |\ *...\ )$, tenemos por ejemplo la computación:

$$(\ 0, ...*\underline{\perp}|\ |\ *...\ ) \vdash (\ 1, ...*\ |\underline{\perp}|\ *...\ ) \vdash (\ 1, ...*\ ||\underline{\perp}*...\ ) \vdash$$
$$(\ 1, ...*\ |||\ \underline{*}...\ ) \vdash (\ 2, ...*\ |||\underline{\perp}*...\ ) \vdash (\ 3, ...*\ ||||\ \underline{*}...\ )$$

Esta computación, de longitud cinco, es completa.
Se podría resumir escribiendo:

$$(\ 0, ...*\underline{\perp}|\ |\ *...\ ) \vdash^{5} (\ 3, ...*\ ||||\ \underline{*}...\ )$$

o bien:

$$(\ 0, ...*\underline{\perp}|\ |\ *...\ ) \vdash^{+} (\ 3, ...*\ ||||\ \underline{*}...\ )$$

o también:

$$(\ 0, ...*\underline{\perp}|\ |\ *...\ ) \vdash^{*} (\ 3, ...*\ ||||\ \underline{*}...\ )$$

Iniciada con la configuración $(\ 0, ...*\ |||\ ****\underline{\ }...\ )$ tenemos por ejemplo:

$$(\ 0, ...*\ |||\ ****\underline{\ }...\ ) \vdash (\ 0, ...*\ |||\ ***\underline{\ }...\ ) \vdash (\ 0, ...*\ |||\ **\underline{\ }...\ ) \vdash$$
$$(\ 0, ...*\ |||\ \underline{*}...\ ) \vdash (\ 0, ...*\ ||\underline{\perp}*...\ ) \vdash (\ 1, ...*\ |||\ \underline{*}...\ ) \vdash$$
$$(\ 2, ...*\ |||\underline{\perp}*...\ ) \vdash (\ 3, ...*\ ||||\ \underline{*}...\ )$$

Esta computación, de longitud siete, también es completa.

Iniciada con la configuración $(\ 2, ...*\ |||\ \underline{*}...\ ,\ 0\ )$ tenemos por ejemplo:

$$(\ 2, ...*\ |||\ \underline{*}...\ ,\ 0\ ) \vdash (\ 2, ...*\ |||\ \underline{*}...\ ,\ 0\ )$$

Esta computación, de longitud uno, no es completa, ya que ni es bien iniciada (su primera configuración, C_0 , no es inicial) ni es terminada (su última configuración, C_1 , no es terminal).

Podemos afirmar que M no se para a partir de $(\ 2,\ ...*\ |\ |\ |\ \underline{*}...\ ,\ 0\ )$, ya que hemos encontrado una computación que la tiene como primera configuración en la cual aparece una configuración repetida (en este caso ella misma), es decir, hemos encontrado un bucle (en este caso de longitud uno) en dicha computación. Esto significa que dada cualquier longitud siempre podemos encontrar una computación que partiendo de la citada configuración tenga dicha longitud (que corresponde precisamente con la definición de no pararse una MT a partir de una configuración).

Nota: La línea quinta de la tabla de M ($2\ *\ *\ 2$) tiene esta forma (tan poco útil) debido a que es la habitual cuando estamos seguros de que nunca "pasaremos" por ella en una computación bien iniciada. Téngase en cuenta que en M sólo llegamos al estado 2 "a través" de la tercera línea de M ($1\ *\ |\ 2$) tras haber escrito "|".

Iniciada con la configuración $(\ 0,\ ...\underline{*}\ |\ |\ |\ *...\ )$ tenemos por ejemplo:

$$(\ 0,\ ...\underline{*}\ |\ |\ |\ *...\ )\ \vdash\ (\ 0,\ ...\underline{**}\ |\ |\ |\ *...\ )\ \vdash\ (\ 0,\ ...\underline{***}\ |\ |\ |\ *...\ )$$

Esta computación, de longitud dos, es bien iniciada pero no es completa, ya que no es una computación terminada.

Podemos afirmar que M no se para a partir de cualquier configuración inicial de la forma $(\ 0,\ ...\underline{*}\ |\ |\ |\ *...\ ,\ z\ )$, pero esta afirmación no se basa, como antes, en la detección de un bucle en una computación que tiene dicha configuración como primera (es decir, en una computación bien iniciada "en ella"), sino en un estudio del funcionamiento de esta MT en concreto y de la configuración inicial dada, que tiene como resultado el darnos cuenta de que no existe, para dicha configuración, ningún cuadrado no vacío a la izquierda del cuadrado escrutado, y por tanto nunca "saldremos" del estado cero.

Nota: La pregunta de si una MT concreta parará o no a partir de una configuración inicial dada no es ni mucho menos fácil de contestar de forma genérica. Es lo que se conoce como "Problema de la Parada", el cual estudiaremos más adelante. □

Cuando uno diseña una MT, las computaciones que nos interesan son las bien iniciadas. Según nuestras definiciones, a partir de una configuración inicial de una MT sólo podemos obtener, a lo sumo, una computación completa. Si no

podemos obtenerla significa que dicha MT no se para a partir de esa configuración inicial.

No es raro que en muchas ocasiones, de una computación completa de una MT lo que nos interese sea sólo la configuración inicial y final. De hecho, muchas veces realmente no nos importa cuales son en concreto los enteros de los cuadrados escrutados de dichas configuraciones ni tampoco los naturales concretos de los estados de las mismas, sino que sólo nos interesa el contenido de la cinta y cual es el cuadrado escrutado dentro de ese contenido al inicio y al final del cómputo. Es decir, estamos interesados solamente en "qué" hace de forma genérica la MT y no en los detalles de "cómo" lo hace.

En aquellos casos en que tenemos una MT y una expresión de cinta sencillas, podemos resumir aún más una computación completa si además de simplemente señalar que la configuración inicial transita a la terminal, eliminamos de dichas configuraciones los estados y los cuadrados escrutados (primera y tercera componentes de las mismas respectivamente) y las expresiones de cinta las representamos con el convenio ya introducido anteriormente (incluido el subrayado para el cuadrado escrutado).

Lo que obtenemos se suele denominar "método de operación" de la MT, y solo es posible hacerlo cuando la descripción de lo "qué" hace de forma genérica una MT es una descripción bastante sencilla. En realidad un método de operación describe muchas computaciones completas, y por lo general no describe otras muchas (por lo que no es único para una MT).

Veamos a continuación un ejemplo de método de operación de una MT.

Ejemplo 8.25:
Sea M la MT dada por la tabla:

7	*	|	*3*
7	|	*r*	*7*
3	*	*r*	*5*
3	|	*r*	*5*
5	*	*h*	*5*
5	|	*h*	*5*

Iniciada con la configuración $(\ 7,\ ...*\ |\ |\ |\ \underline{*}...\ ,\ 0\ )$ tenemos la computación completa:

$$(\ 7,\ ...*\ |\ |\ |\ \underline{*}...\ ,\ 0\ ) \vdash (\ 3,\ ...*\ |\ |\ |\ \underline{|}*...\ ,\ 0\ ) \vdash (\ 5,\ ...*\ |\ |\ |\ |\ \underline{*}...\ ,\ 1\ )$$

Esta computación, de longitud dos, se podría resumir escribiéndola así:

$$(7, ...* | | | \underline{*}... , 0) \ |\!\!-\!\!\!-^{*} \ (5, ...* | | | | \underline{*}... , 1)$$

Por tanto, según lo expuesto anteriormente, de aquí obtendríamos el siguiente método de operación de esta MT:

$$(...* | | | \underline{*}...) \ |\!\!-\!\!\!-^{*} \ (...* | | | | \underline{*}...)$$

Este método de operación muestra que esta MT colocada detrás de una cadena de longitud tres (ya que Σ solo tiene un símbolo) se para a la derecha de una cadena de longitud cuatro manteniendo el resto de la cinta vacía. Puesto que este funcionamiento es similar sea cual sea la cadena detrás de la que coloquemos la MT, podemos generalizar aún más el método de operación de la siguiente forma:

$$(...* w \underline{*}...) \ |\!\!-\!\!\!-^{*} \ (...* w | \underline{*}...)$$

Es habitual también, para dejar si cabe aún más claro que lo que tenemos ya no son configuraciones ni una computación, sino algo más general y menos concreto, representar los métodos de operación eliminando los paréntesis y sustituyendo el símbolo de transitar por el símbolo "$\Rightarrow$". Aplicando esto al caso que nos ocupa tendríamos finalmente el siguiente método de operación:

$$...* w \underline{*}... \ \Rightarrow \ ...* w | \underline{*}...$$

Notas: El método de operación expuesto describe muchas computaciones completas, pero hay otras muchas que no describe, por ejemplo aquellas que inician el cómputo no detrás de una cadena sino "sobre la cadena" o "delante" de ella. Señalar también que un método de operación no nos indica el grado de complejidad de la computación realizada por la MT, sino sólo una descripción general del inicio y del final de dicha computación (como muestra véase que el modo de operación anterior es aplicable a la MT del ejemplo **8.23**, a pesar de que la computación en este caso para una misma cadena es más larga).

La MT presentada en este ejemplo tiene sus estados no consecutivos y el inicial es el siete. Si bien esto es formalmente correcto, lo habitual es que los estados de una MT sean naturales consecutivos a partir del cero y que sea dicho estado cero el inicial. □

Para terminar este apartado a continuación veremos algunos ejemplos más de MT, los cuales estarán menos comentados que los precedentes con el objeto de que el lector reflexione sobre ellos.

Ejemplo 8.26:
Sea M la MT dada por la tabla:

0	*	*r*	*0*
0	|	*l*	*0*

Por tanto a partir de cualquier configuración M no se para. □

Ejemplo 8.27:
La tabla:

0	*	*r*	*0*
0	|	*r*	*1*

no corresponde con ninguna MT. □

Ejemplo 8.28:
Sea M_1 la MT dada por la tabla:

0	*	*l*	*1*
0	|	*l*	*1*
1	*	*h*	*1*
1	|	*l*	*1*

y M_2 la MT dada por la tabla:

10	*	*l*	*20*
10	☺	*l*	*20*
10	♀	*l*	*20*
10	♂	*l*	*20*
10	€	*l*	*20*
20	*	*h*	*20*
20	☺	*l*	*20*
20	♀	*l*	*20*
20	♂	*l*	*20*
20	€	*l*	*20*

El siguiente método de operación:

$$\sim^* w \underline{^*}\sim \;\Rightarrow\; \sim\underline{^*} w\, ^*\sim$$

lo es tanto de M_1 como de M_2. □

Ejemplo 8.29:
Sea M la MT dada por la tabla:

0	*	*l*	*1*
0	|	*l*	*1*
1	*	|	*2*
1	|	*l*	*1*
2	*	*l*	*3*
2	|	*r*	*2*
3	*	*l*	*4*
3	|	*	*3*
4	*	*h*	*4*
4	|	*	*4*

M tiene el siguiente método de operación:

$$\sim * \; w \,|\, | \; \underline{*} \sim \;\; \Rightarrow \;\; \sim | \; w \; \underline{*} ** \sim$$ □

8.4 Computabilidad, decidibilidad y enumerabilidad con MT

Para definir formalmente estos conceptos, como ya señalamos en el tema anterior, vamos a ceñirnos a funciones que trabajan con naturales, es decir, funciones de la forma $f: \mathbb{N}^n \to \mathbb{N}$ con $n \geq 0$.

Puesto que vamos a definir estos conceptos con MT necesitamos poder representar cualquier natural en la cinta de una MT. Así, si tenemos una MT $M = (K, q_0, \Sigma, \gamma, \delta)$, cada natural n lo representaremos como una cadena de longitud $n+1$ formada exclusivamente por ocurrencias del símbolo $a_1 \in \Sigma$. Sobrecargando el lenguaje, llamaremos n a la cadena que representa al natural n (la cual tiene longitud $n+1$). Es habitual trabajar en este contexto con el alfabeto $\Sigma = \{ | \}$ (a veces denominado alfabeto "palote"), de modo que por ejemplo el dos lo representaremos con la cadena "| | |" (a la que denominaremos cadena *2*) y el cero con la cadena "|" (a la que denominaremos cadena *0*).

Una vez que hemos establecido como representar un natural, y puesto que vamos a trabajar con funciones *n*-arias con $n \geq 0$, debemos especificar cómo proporcionarle a la MT los n argumentos de una función *n*-aria.

Definición 8.30: *Colocar una MT detrás de unas cadenas*

Sea $M = (K , q_0 , \Sigma , \gamma , \delta)$ una MT.

Sean $w_1 , w_2 , ..., w_n$ cadenas sobre Σ, con longitudes $|w_1| , |w_2| , ..., |w_n|$ respectivamente. Sea $w = w_1 a_0 w_2 a_0 \ ... \ a_0 w_n$ una cadena sobre $\Sigma \cup \{ a_0 \}$, la cual tiene por tanto longitud $| w | = |w_1| + |w_2| + ... + |w_n| + (n-1)$.

Colocar M detrás de las cadenas $w_1 , w_2 , ..., w_n$ (en ese orden) equivale a colocarla detrás de la cadena w.

Nota: En esta definición estamos sobrecargando el término colocarse detrás de una cadena, ya que la cadena w de la definición tiene ocurrencias del símbolo ideal vacío (a_0) dentro de ella. Esta sobrecarga se obvia diciendo que colocar M detrás de las cadenas $w_1 , w_2 , ..., w_n$ (en ese orden) equivale a colocarla en el cuadrado y con la siguiente expresión de cinta:

$E(z) = a_0 \quad \forall z \in \mathbb{Z} - [y-((\sum_{i=1}^{i=n} | w_i |)+ (n-1)) , y-1]$

$E(z) = a_0$ si $z = y-p$ donde $p = (\sum_{i=0}^{i=k} | w_{n-i} |) + (k+1)$ con $k \in [0, n-2]$

$E(z) = w_{n-k}(j)$ si $z = y-p$ donde:

$p = (\sum_{i=0}^{i=k} | w_{n-i} |) + (k+1) - j$ con $k \in [0, n-2]$ y $j \in [1, |w_{n-k}|]$

siendo $w_{n-k}(j) \in \Sigma$ el símbolo j-ésimo de la cadena w_{n-k}.

Llegados aquí estamos en disposición de dar una definición formal de los conceptos intuitivos, descritos en el tema anterior, de función computable, conjunto decidible y conjunto enumerable (o generable). Puesto que la formalización se hace con MT, estos conceptos formales se denominarán anteponiendo a su nombre la palabra "Turing" seguida de un guión.

Definición 8.31: *Función Turing-computable, F(MT)*

Sea la función $f : \mathbb{N}^n \rightarrow \mathbb{N}$ con $n \geq 0$.

Diremos que f es una función Turing-computable sii existe una MT M tal que para todo vector $(m_1 , m_2 , ... , m_n) \in \mathbb{N}^n$, M colocada detrás de las cadenas $m_1 , m_2 , ... , m_n$ se para detrás de la cadena $f(m_1 , m_2 , ... , m_n)$.

En ese caso diremos que dicha MT Turing-computa la función f.

Llamaremos $F(MT)$ al conjunto de todas las funciones Turing-computables.

Nota: Si f no está definida para algunos valores (es decir, no es total) entonces para esos valores la MT no se para (en **9.8** se encuentra una definición formal de función total y parcial).

Definición 8.32: *Conjunto Turing-decidible, DEC(MT)*
Sea $A \subseteq \mathbb{N}^n$.
Diremos que A es un conjunto Turing-decidible sii existe una MT M tal que para todo vector $(m_1, m_2, \ldots, m_n) \in \mathbb{N}^n$, M colocada detrás de las cadenas $m_1, m_2, \ldots, m_n$ se para en una configuración terminal $C = (q, E, z)$ cumpliéndose que: $E(z) \neq a_0 \Leftrightarrow (m_1, m_2, \ldots, m_n) \in A$
En ese caso diremos que dicha MT Turing-decide el conjunto A.
Llamaremos $DEC(MT)$ al conjunto de todos los conjuntos Turing-decidibles.

Definición 8.33: *Conjunto Turing-enumerable, ENU(MT)*
Sea $A \subseteq \mathbb{N}^n$.
Diremos que A es un conjunto Turing-enumerable sii existe una MT M tal que para todo vector $(m_1, m_2, \ldots, m_n) \in \mathbb{N}^n$, M colocada detrás de las cadenas $m_1, m_2, \ldots, m_n$ cumple que:

$$\left.\begin{array}{c}\text{se para en una configuración terminal } C = (q, E, z) \\ \wedge \\ E(z) \neq a_0\end{array}\right\} \Leftrightarrow (m_1, m_2, \ldots, m_n) \in A$$

En ese caso diremos que dicha MT Turing-enumera el conjunto A.
Llamaremos $ENU(MT)$ al conjunto de todos los conjuntos Turing-enumerables.

Nota: $DEC(MT) \subset ENU(MT)$

Definición 8.34: *Conjunto Turing-generable*
Sea $A \subseteq \mathbb{N}^n$.
Diremos que A es un conjunto Turing-generable sii A es Turing-enumerable.

Ejemplo 8.35:
Sea la función $sucesor : \mathbb{N} \to \mathbb{N}$ donde $sucesor(n) = n+1$.
La función $sucesor$ es una función Turing-computable, ya que la MT M del ejemplo **8.25** la Turing-computa (es decir, $sucesor \in F(MT)$). □

Ejemplo 8.36:
Sea la función $suma : \mathsf{N}^2 \rightarrow \mathsf{N}$ donde $suma(n,m) = n+m$.
La función *suma* es una función Turíng-computable, ya que la MT M del ejemplo **8.29** la Turing-computa (es decir, $suma \in F(MT)$). □

Se puede demostrar que las operaciones aritméticas habituales, así como todas las que podemos obtener a partir de ellas componiéndolas (de manera finita) son todas ellas Turing-computables. En realidad se asume como cierta la tesis (que es parte de la de Church) de que toda función computable en el sentido intuitivo es Turing-computable, y viceversa. De hecho esta asunción se extiende no sólo a $F(MT)$, sino también a $DEC(MT)$ y $ENU(MT)$ en relación con sus correspondientes conceptos intuitivos (decidible y enumerable respectivamente).

Antes de terminar este apartado recordar que, como ya se comentó en el primer tema, es posible definir estos últimos conceptos (los definidos a partir del de función Turing-computable) no sólo para funciones y conjuntos de naturales sino, de una forma más general, para funciones y conjuntos de cadenas sobre un alfabeto.

Señalar también que hay otras formas similares pero equivalentes de definir los citados conceptos. Por ejemplo, al definir función Turing-computable se puede hacer diciendo que la MT se para "a la derecha" en lugar de "detrás" del resultado, o bien iniciar la computación en cualquier cuadrado de la cinta en lugar de "detrás" de los argumentos. De la misma forma, al definir conjunto Turing-decidible podemos hacerlo destacando dos símbolos concretos sobre los que se parará la MT para decidir en lugar de simplemente de si es o no el símbolo ideal vacío. Estas variaciones en las definiciones no cambian lo definido (es decir, son definiciones equivalentes), por lo que cualquiera de ellas es válida.

8.5 Algunas consideraciones

En este apartado vamos a destacar una serie de hechos genéricos (algunos ya comentados) sobre las MT los cuales consideramos que en un tema sobre ellas deben de, al menos, ser mencionados, si bien (por razones de espacio y propósitos del presente libro) no los demostraremos.

En primer lugar ya hemos visto que ni la posición del cuadrado de número cero ni la numeración concreta de los estados es determinante. Tampoco lo es el alfabeto utilizado, ya que por medio de un isomorfismo podemos usar otro cualquiera (en este caso del mismo cardinal). De hecho tampoco es determinante cuantos símbolos tiene el alfabeto de la MT, ya que podemos computar cualquier función perteneciente a *F(MT)* con un alfabeto de un sólo símbolo (p.e. el alfabeto "palote"). Para ello sólo hay que representar los símbolos del alfabeto de mayor cardinal como cadenas del único símbolo del otro alfabeto. La MT resultante de esta "traducción" tendrá más estados que de la de partida pero también Turing-computará la función.

Destacable es también el hecho de que una MT, a pesar de ser finita por definición, puede realizar procesos infinitos no periódicos. Con "no periódicos" nos referimos a que nunca se repite una configuración por muy larga que escojamos la computación. Esto es posible debido a que su funcionamiento viene determinado, además de por su tabla, por el contenido de la cinta, la cual no está acotada. No es difícil, por ejemplo, construir una MT que, a partir de una cinta vacía, no se pare nunca mientras escribe la secuencia infinita: * | * | | * | | | * | | | * ... , la cual es claramente no periódica (ya que aunque es "predecible" nunca repite configuración).

Es fácil apreciar que por su simplicidad estructural es engorroso crear una MT en cuanto la función que queremos Turing-computar no es trivial. Para paliar esto en la medida de lo posible se definen unos grafos llamados "diagramas de MT" mediante los cuales es posible definir gráficamente, a partir de unas MT elementales, MT complejas sin necesidad de dar sus extensas tablas (si bien estas se pueden construir a partir de dichos diagramas). Aunque esto no añade nada desde el punto de vista formal, nos permite no sólo definir, utilizando incluso varios niveles, MT complejas, sino además seguir (y entender) el funcionamiento de dichas máquinas, lo cual es importante a la hora de hacer demostraciones con MT.

En definitiva, hemos presentado en este tema el primer modelo de cómputo, de los tres que veremos, que formaliza el concepto intuitivo de algoritmo: la Máquina de Turing. Históricamente fué de los primeros en desarrollarse e indudablemente es el más conocido de todos. Basándonos en él hemos formalizado también los conceptos intuitivos de computable, decidible y enumerable.

9
Funciones Recursivas

9.1 Concepto informal

Si en el tema anterior para formalizar el concepto de función computable (Turing-computable) hemos definido previamente un modelo de cómputo (MT), aquí accederemos directamente al concepto de función computable (en este caso función recursiva) quedando el modelo de cómputo implícito en la definición de dicho concepto.

La idea de *función recursiva* (también denominada inicialmente *μ-recursiva*) es muy sencilla. Se escogen algunas funciones muy simples, a las que denominaremos *funciones iniciales* (las cuales, por su extrema simplicidad, podríamos considerar que son "trivialmente" computables). Posteriormente se definen varios operadores sobre funciones (en concreto tres: composición, recursión primitiva y minimización no acotada). Llegados a este punto diremos que una función es recursiva si es una función inicial o se puede generar a partir de funciones iniciales mediante alguna secuencia finita de los operadores mencionados.

De esta forma, el conjunto de todas las funciones recursivas (al que denominaremos *REC*) coincide con la mínima clase de funciones que contiene a las funciones iniciales y es cerrada para la composición, la recursión primitiva y la minimización no acotada.

Nota: No confundir las funciones recursivas (μ-recursivas) con las funciones *recursivas primitivas*. Estas últimas forman un conjunto (que se suele denominar *PRIM*) que es un subconjunto propio de *REC*, es decir, todas las funciones recursivas primitivas también son recursivas, pero hay funciones recursivas que no son recursivas primitivas. Su definición es similar a las *REC* sólo que no incluye el operador de minimización, por lo que son todas funciones totales.

9.2 Definición formal

Definición 9.1: *Función cero (θ)*

$\theta : \mathbb{N}^0 \rightarrow \mathbb{N}$

$\theta() = 0$

Definición 9.2: *Función sucesor (σ)*

$\sigma : \mathbb{N} \rightarrow \mathbb{N}$

$\sigma(n) = n+1$

Definición 9.3: *Función proyección i-ésima de k argumentos (π_i^k, π)*

Para cada par $(k,i) \in \mathbb{N}^2$ con $k \geq 1$ y $1 \leq i \leq k$ se define la función proyección i-ésima de k argumentos, π_i^k, como:

$\pi_i^k : \mathbb{N}^k \rightarrow \mathbb{N}$

$\pi_i^k(\underline{n}) = n_i$ donde $\underline{n} = (n_1, n_2, ..., n_k)$

Al conjunto de todas las proyecciones i-ésimas de k argumentos lo denominaremos π (proyecciones), es decir:

$\pi = \{ \pi_i^k \mid (k,i) \in \mathbb{N}^2 \text{ con } k \geq 1 \wedge 1 \leq i \leq k \}$

Definición 9.4: *Funciones iniciales (INI)*

Definimos el conjunto de las funciones iniciales, al que denominaremos *INI*, como:

$INI = \pi \cup \{ \theta, \sigma \}$

Nota: Es decir, las funciones iniciales son la función cero, la función sucesor y las proyecciones.

Una vez que hemos definido las funciones iniciales, las cuales usaremos posteriormente, pasamos a continuación a definir tres operadores sobre funciones que nos van a permitir crear nuevas funciones a partir de otras ya definidas. Dichos operadores son los ya mencionados de composición, recursión primitiva y minimización no acotada.

Definición 9.5: *Composición*

Sean $m > 0$, $k \geq 0$ y las funciones:

$g : \mathsf{N}^m \to \mathsf{N}$

$h_1 , h_2 , ..., h_m : \mathsf{N}^k \to \mathsf{N}$

Si la función $f : \mathsf{N}^k \to \mathsf{N}$ es:

$$f(\underline{n}) = g(h_1(\underline{n}) , h_2(\underline{n}) , ..., h_m(\underline{n}))$$

entonces decimos que f se obtiene a partir de g , h_1 , h_2 , ..., h_m por composición.

Lo notaremos $f(\underline{n}) = g(h_1 , h_2 , ..., h_m)(\underline{n})$, o simplemente $f = g(h_1 , h_2 , ..., h_m)$.

Definición 9.6: *Recursión primitiva*

Sea $k \geq 0$ y las funciones:

$g : \mathsf{N}^k \to \mathsf{N}$

$h : \mathsf{N}^{k+2} \to \mathsf{N}$

Si la función $f : \mathsf{N}^{k+1} \to \mathsf{N}$ es:

$$f(\underline{n},m) = \begin{cases} g(\underline{n}) & \text{si } m = 0 \\ h(\underline{n},m-1,f(\underline{n},m-1)) & \text{si } m > 0 \end{cases}$$

entonces decimos que f se obtiene a partir de g y h por recursión primitiva.

Lo notaremos $f(\underline{x}) = \langle g|h \rangle(\underline{x})$, o simplemente $f = \langle g|h \rangle$.

Definición 9.7: *Minimización no acotada*

Sea $k \geq 0$ y la función:

$g : \mathsf{N}^{k+1} \to \mathsf{N}$

Si la función $f : \mathsf{N}^k \to \mathsf{N}$ es:

$$f(\underline{n}) = \begin{cases} mínimo(A) & \text{si } A \neq \varnothing \ \wedge \ \forall t \leq mínimo(A) \ \ g(\underline{n},t) \in \mathsf{N} \\ \uparrow & \text{en otro caso} \end{cases}$$

$$\text{donde } A = \{ t \in \mathsf{N} \mid g(\underline{n},t) = 0 \} \text{ y } \underline{n} \in \mathsf{N}^k$$

entonces decimos que f se obtiene a partir de g por minimización no acotada.

Lo notaremos $f(\underline{n}) = \mu[g](\underline{n})$, o simplemente $f = \mu[g]$.

Nota: El símbolo "$\uparrow$" significa que la función, para ese vector de entrada ($\underline{n}$), cumple que: $\underline{n} \notin Dom(f)$. Es decir, la función para dicha entrada *diverge* (a veces se utiliza el término "no está definida").

Veamos ahora algunos aspectos destacables de estas tres definiciones.

En todas ellas es fundamental que los tamaños de los argumentos de las funciones que intervienen sean exactamente los definidos.

Así, en la composición, han de haber tantas funciones h (habitualmente denominadas *funciones internas*) como argumentos tiene la función g (habitualmente denominada *función externa*) que tiene al menos uno. Además todas las funciones h tienen el mismo número de argumentos, coincidiendo éste con los de la función obtenida por composición (es decir, la función f).

En la recursión primitiva, la función obtenida (es decir, la función f) tiene un argumento más que la función g (habitualmente denominada *función base*) y uno menos que la función h (habitualmente denominada *función de inducción*).

En la minimización no acotada, la función obtenida (es decir, la función f) tiene un argumento menos que la función g (la cual tiene al menos uno).

Para poder comentar otro aspecto destacable de estos operadores necesitamos previamente definir formalmente los conceptos de función total y función parcial.

Definición 9.8: *Función total, Función parcial*

Sea la función $f: \mathbb{N}^k \to \mathbb{N}$ con $k > 0$.

Diremos que f es una función total si $Dom(f) = \mathbb{N}^k$.

Sea la función $g: \mathbb{N}^0 \to \mathbb{N}$.

Diremos que g es una función total si $g() \in \mathbb{N}$.

Diremos que una función es parcial sii no es total.

Nota: Recordar que se define el *dominio* de una función $f: A \to B$, notado $Dom(f)$, como: $Dom(f) = \{ a \in A \mid f(a) = b \text{ con } b \in B \}$.

Nótese que si los operadores de composición y recursión primitiva trabajan con funciones totales la función obtenida siempre es total. Sin embargo, el operador de minimización no acotada puede obtener una función parcial a partir de una total, lo que, junto con el hecho de que la minimización es "no acotada", amplía considerablemente el conjunto de funciones que podemos obtener con él.

De hecho, es este último operador, como veremos a continuación, el que nos va a permitir obtener todo el conjunto de las funciones computables (que aquí llamaremos recursivas) a partir de las iniciales y de los otros dos operadores.

Definición 9.9: *Función recursiva, REC*
Definimos el conjunto *REC* como:

1/ $INI \subset REC$

2/ $g, h_1, h_2, ..., h_m \in REC \ \wedge \ \exists\, g(h_1, h_2, ..., h_m) \ \Rightarrow \ g(h_1, h_2, ..., h_m) \in REC$

3/ $g, h \in REC \ \wedge \ \exists\, \langle g|h \rangle \ \Rightarrow \ \langle g|h \rangle \in REC$

4/ $g \in REC \ \wedge \ \exists\, \mu[g] \ \Rightarrow \ \mu[g] \in REC$

5/ Ninguna otra función pertenece al conjunto *REC*.

Diremos que f es una función recursiva sii $f \in REC$.

Notas: *REC*, por tanto, es el conjunto de todas las funciones recursivas (también llamadas μ-recursivas).

La definición dada del conjunto *REC* es una definición constructiva, de hecho es una definición inductiva con un apartado base, tres de inducción, y uno de exclusión (para garantizar que el conjunto sea mínimo).

Cuando se está familiarizado con el concepto de "conjunto cerrado para una operación" también se puede definir el conjunto *REC* de manera equivalente como: "*REC* es la mínima clase de funciones que contiene a las funciones iniciales y es cerrada para la composición, la recursión primitiva y la minimización no acotada".

Señalar por último que según hemos definido, una función recursiva o bien es una función inicial, o bien se puede generar a partir de las funciones iniciales mediante alguna secuencia finita de operaciones de composición, recursión primitiva y minimización no acotada. Es precisamente el hecho de que la secuencia ha de ser finita lo que nos asegura aquí que toda función recursiva es representable (sin embargo, como veremos más adelante, toda función representable no es recursiva).

Una vez que tenemos definidos los conceptos de función recursiva y función total podemos definir el conjunto *TREC*, el cual es subconjunto propio de *REC*.

Definición 9.10: *TREC*

$TREC = \{ f \in REC \mid f \text{ es total} \}$

Para la evaluación de los operadores sobre un vector de argumentos sólo tenemos que aplicar sus respectivas definiciones.

Así, en la composición, dado un vector de argumentos ($\underline{n}$), se evalúan primero las funciones internas sobre dicho vector, y a continuación, a partir de los valores obtenidos, se evalúa la función externa.

$$f = g(h_1, h_2, ..., h_m)$$

$$\Downarrow$$

$$f(\underline{n}) = g(h_1, h_2, ..., h_m)(\underline{n}) = g(h_1(\underline{n}), h_2(\underline{n}), ..., h_m(\underline{n}))$$

En la recursión primitiva, dado un vector de argumentos ($\underline{x} = (\underline{n}, m)$), la recursión se realiza sobre la última componente (m), de forma que se van haciendo llamadas al operador disminuyendo en uno cada vez la última componente, hasta que dicha componente alcanza el valor cero. En ese momento se evalúa la función base (sobre el vector $(\underline{n},0)$), y a partir de ahí las distintas llamadas a la función de inducción hasta llegar al valor pedido.

$$f = \langle g|h \rangle$$

$$\Downarrow$$

$$f(\underline{n}, m) = \langle g|h \rangle(\underline{n}, m) = h(\underline{n}, m-1, f(\underline{n}, m-1))$$

$$f(\underline{n}, m-1) = \langle g|h \rangle(\underline{n}, m-1) = h(\underline{n}, m-2, f(\underline{n}, m-2))$$

$$f(\underline{n}, m-2) = \langle g|h \rangle(\underline{n}, m-2) = h(\underline{n}, m-3, f(\underline{n}, m-3))$$

$$\vdots$$

$$f(\underline{n}, m-(m-3)) = \langle g|h \rangle(\underline{n}, 3) = h(\underline{n}, 2, f(\underline{n}, 2))$$

$$f(\underline{n}, 2) = \langle g|h \rangle(\underline{n}, 2) = h(\underline{n}, 1, f(\underline{n}, 1))$$

$$f(\underline{n}, 1) = \langle g|h \rangle(\underline{n}, 1) = h(\underline{n}, 0, f(\underline{n}, 0))$$

$$f(\underline{n}, 0) = \langle g|h \rangle(\underline{n}, 0) = g(\underline{n})$$

Obsérvese que puesto que la última componente del vector de argumentos inicial es un número natural el proceso de recursión siempre es finito (y de profundidad igual a dicho número). Así, si sustituimos las sucesivas llamadas tendremos una expresión finita de la forma:

$$\langle g|h\rangle(\underline{n},m) = h(\underline{n}, m\text{-}1, h(\underline{n}, m\text{-}2, h(\underline{n}, m\text{-}3, \cdots h(\underline{n}, 2, h(\underline{n}, 1, h(\underline{n}, 0, g(\underline{n})))) \cdots)))$$

Para terminar este apartado, señalar que, a pesar de su definición estática, la evaluación de la minimización no acotada, dado un vector de argumentos ($\underline{n}$), se puede asimilar a un proceso por el cual probamos, sucesivamente, a partir de $t = 0$, si $g(\underline{n},t) = 0$. El primer t que cumpla la condición es el valor buscado de $\mu[g](\underline{n})$, siempre y cuando para los valores menores que t la función g no diverja. Si ese t no existiera, o si para algún valor menor que t la función g diverge, entonces el proceso descrito no pararía nunca, lo que significaría que $\mu[g](\underline{n}) = \uparrow$ (diverge).

Nota: Destacar la relación entre los conceptos de recursión primitiva y bucle definido, y entre los de minimización no acotada y bucle indefinido.

9.3 Ejemplos de funciones recursivas

Ejemplo 9.11:
La función *sumados* definida como:
$sumados : \mathbb{N} \to \mathbb{N}$
$sumados(n) = n + 2$
es una función recursiva, ya que se puede definir como:

$$sumados(n) = \sigma(\sigma(n))$$

o de forma más abreviada:

$$sumados = \sigma(\sigma)$$

Es decir, se puede definir con el operador composición con:

$m = 1$
$k = 1$
$g = \sigma$ ($g : \mathbb{N} \to \mathbb{N}$)
$h_1 = \sigma$ ($h_1 : \mathbb{N} \to \mathbb{N}$)

Nota: A la hora de intentar dar la expresión recursiva de una función es conveniente fijar primero la idea sobre la que se va a construir dicha expresión. En esta función tan sencilla la idea también es muy simple: $sumados(n) = (n+1)+1$. □

Ejemplo 9.12:

La función *suma* definida como:

$suma : \mathbb{N}^2 \to \mathbb{N}$

$suma(n_1, n_2) = n_1 + n_2$

es una función recursiva, ya que se puede definir como:

$$suma(\underline{n}) = \langle \pi_1^1 \mid sucesor_3 \rangle(\underline{n})$$

o de forma más abreviada:

$$suma = \langle \pi_1^1 \mid sucesor_3 \rangle$$

Es decir, se puede definir con el operador recursión primitiva con:

$k = 1$

$g = \pi_1^1$ ($g : \mathbb{N} \to \mathbb{N}$) ($\pi_1^1 = I$, identidad)

$h = sucesor_3$ ($h : \mathbb{N}^3 \to \mathbb{N}$)

donde

$sucesor_3 : \mathbb{N}^3 \to \mathbb{N}$

$sucesor_3(n_1, n_2, n_3) = n_3 + 1$

también es una función recursiva, ya que se puede definir como:

$$sucesor_3(\underline{n}) = \sigma(\pi_3^3(\underline{n}))$$

o de forma más abreviada:

$$sucesor_3 = \sigma(\pi_3^3)$$

Es decir, se puede definir con el operador composición con:

$m = 1$

$k = 3$

$g = \sigma$ ($g : \mathbb{N} \to \mathbb{N}$)

$h_1 = \pi_3^3$ ($h_1 : \mathbb{N}^3 \to \mathbb{N}$)

Notas: La función *suma* no se puede definir con el operador composición ya que los valores m y k de dicho operador son naturales fijos, no variables.

La idea para definir recursivamente la función *suma* es:

$$\begin{cases} suma(n_1, 0) = n_1 \\ suma(n_1, n_2) = suma(n_1, n_2 - 1) + 1 \end{cases}$$

y la idea para la función $sucesor_3$ es:

$sucesor_3(\underline{n}) = \pi_3^3(\underline{n}) + 1$ □

Cuando en la definición recursiva de una función aparecen otras funciones recursivas distintas de las iniciales decimos que es una *expresión abreviada* de la función. Cuando por el contrario en la definición de la función sólo aparecen funciones iniciales (y operadores) decimos que es una *expresión completa* de la función.

Una expresión completa se puede obtener de una abreviada simplemente sustituyendo en ella las funciones que no son iniciales por sus respectivas expresiones completas.

En el ejemplo anterior la expresión abreviada de la función *suma* es:

$suma = \langle \pi_1^1 \mid sucesor_3 \rangle$

y la expresión completa:

$suma = \langle \pi_1^1 \mid \sigma(\pi_3^3) \rangle$

Veamos ahora un ejemplo de evaluación de una función recursiva, tanto en su expresión abreviada como en su expresión completa.

Ejemplo 9.13:

Vamos a evaluar, paso a paso, la función *suma* definida en el ejemplo anterior, tanto en su expresión abreviada como en su expresión completa, para el vector de argumentos $(4, 3)$.

$$\begin{aligned}
&suma(4, 3) = \langle \pi_1^1 \mid sucesor_3 \rangle(4, 3) = \\
&\quad sucesor_3(4, 2, \langle \pi_1^1 \mid sucesor_3 \rangle(4, 2)) = \\
&\quad sucesor_3(4, 2, sucesor_3(4, 1, \langle \pi_1^1 \mid sucesor_3 \rangle(4, 1))) = \\
&\quad sucesor_3(4, 2, sucesor_3(4, 1, sucesor_3(4, 0, \langle \pi_1^1 \mid sucesor_3 \rangle(4, 0)))) = \\
&\quad sucesor_3(4, 2, sucesor_3(4, 1, sucesor_3(4, 0, \pi_1^1(4)))) = \\
&\quad sucesor_3(4, 2, sucesor_3(4, 1, sucesor_3(4, 0, 4))) = \\
&\quad sucesor_3(4, 2, sucesor_3(4, 1, 5)) = \\
&\quad sucesor_3(4, 2, 6) = \\
&\quad 7
\end{aligned}$$

$suma(4\ ,\ 3) = \langle \pi_1^1 \mid \sigma(\pi_3^3) \rangle(4\ ,\ 3) =$

$\sigma(\pi_3^3)(4\ ,\ 2\ ,\ \langle \pi_1^1 \mid \sigma(\pi_3^3) \rangle(4\ ,\ 2)) =$

$\sigma(\pi_3^3)(4\ ,\ 2\ ,\ \sigma(\pi_3^3)(4\ ,\ 1\ ,\ \langle \pi_1^1 \mid \sigma(\pi_3^3) \rangle(4\ ,\ 1))) =$

$\sigma(\pi_3^3)(4\ ,\ 2\ ,\ \sigma(\pi_3^3)(4\ ,\ 1\ ,\ \sigma(\pi_3^3)(4\ ,\ 0\ ,\ \langle \pi_1^1 \mid \sigma(\pi_3^3) \rangle(4\ ,\ 0)))) =$

$\sigma(\pi_3^3)(4\ ,\ 2\ ,\ \sigma(\pi_3^3)(4\ ,\ 1\ ,\ \sigma(\pi_3^3)(4\ ,\ 0\ ,\ \pi_1^1(4)))) =$

$\sigma(\pi_3^3)(4\ ,\ 2\ ,\ \sigma(\pi_3^3)(4\ ,\ 1\ ,\ \sigma(\pi_3^3)(4\ ,\ 0\ ,\ 4))) =$

$\sigma(\pi_3^3)(4\ ,\ 2\ ,\ \sigma(\pi_3^3)(4\ ,\ 1\ ,\ \sigma(\pi_3^3(4\ ,\ 0\ ,\ 4)))) =$

$\sigma(\pi_3^3)(4\ ,\ 2\ ,\ \sigma(\pi_3^3)(4\ ,\ 1\ ,\ \sigma(4))) =$

$\sigma(\pi_3^3)(4\ ,\ 2\ ,\ \sigma(\pi_3^3)(4\ ,\ 1\ ,\ 5)) =$

$\sigma(\pi_3^3)(4\ ,\ 2\ ,\ \sigma(\pi_3^3(4\ ,\ 1\ ,\ 5))) =$

$\sigma(\pi_3^3)(4\ ,\ 2\ ,\ \sigma(5)) =$

$\sigma(\pi_3^3)(4\ ,\ 2\ ,\ 6) =$

$\sigma(\pi_3^3(4\ ,\ 2\ ,\ 6)) =$

$\sigma(6) =$

7 □

Ejemplo 9.14:

La función *uncero* definida como:

$uncero : \mathbb{N} \rightarrow \mathbb{N}$

$$\begin{cases} uncero(0) = 0 \\ uncero(n) = \uparrow \quad \text{si} \quad n > 0 \end{cases}$$

es una función recursiva, ya que se puede definir como:

$uncero(n) = \mu[suma](n)$

o de forma más abreviada:

$uncero = \mu[suma]$

Es decir, se puede definir con el operador minimización no acotada con:

$k = 1$

$g = suma \qquad (\ g : \mathbb{N}^2 \rightarrow \mathbb{N}\ )$

donde *suma* es la función recursiva ya definida.

Su expresión completa es:

$uncero = \mu[\langle \pi_1^1 \mid \sigma(\pi_3^3) \rangle]$ □

Ejemplo 9.15:
Vamos a evaluar la función *uncero* definida en el ejemplo anterior para diversos argumentos.

$uncero(0) = \mu[suma](0) = 0$
ya que: $\{ t \in \mathbb{N} \mid suma(0,t) = 0 \} \neq \varnothing$
y por tanto: $uncero(0) = mínimo\{ t \in \mathbb{N} \mid suma(0,t) = 0 \} = mínimo\{ 0 \} = 0$

$uncero(n) = \mu[suma](n) = \uparrow$ si $n > 0$
ya que: $\{ t \in \mathbb{N} \mid suma(n,t) = 0 \} = \varnothing$ si $n > 0$
y por tanto: $uncero(n) = \uparrow$ si $n > 0$

En este caso la evaluación con la expresión completa sería igual. □

9.4 Computabilidad, decidibilidad y enumerabilidad con *REC*

Para definir formalmente estos conceptos, como ya señalamos en los temas anteriores, vamos a ceñirnos a funciones que trabajan con naturales, es decir, funciones de la forma $f: \mathbb{N}^n \rightarrow \mathbb{N}$ con $n \geq 0$.

Señalar en primer lugar que los predicados se notarán con letras mayúsculas y los argumentos entre paréntesis, aunque se permitirá la notación infija para los más comunes. Así mismo, dentro de esta notación, para representar el valor de verdad o falsedad de un predicado se utilizará el convenio:

$P(\underline{x}) \equiv cierto$
$\overline{P(\underline{x})} \equiv falso$

Definición 9.16: *Conjunto de valores de verdad de un predicado*
Sea $k \geq 0$ y P un predicado de k argumentos.
El conjunto de valores de verdad de un predicado P, notado V_P, se define como:

$$V_P = \{ \underline{x} \in \mathbb{N}^k \mid P(\underline{x}) \}$$

Definición 9.17: *Predicado asociado a una función*
Sea $k \geq 0$ y f una función de k argumentos.
El predicado asociado a la función f, notado P_f, se define como:

$$P_f(\underline{x}) = \begin{cases} cierto & \text{si } f(\underline{x}) > 0 \\ falso & \text{en otro caso} \end{cases}$$

Nota: La opción "en otro caso" significa que o bien $f(\underline{x})=0$, o bien $f(\underline{x})=\uparrow$. Así, un mismo predicado puede estar asociado a funciones distintas.

Definición 9.18: *Función característica de un conjunto*
Sea $k \geq 0$ y $V \subseteq \mathbb{N}^k$.
La función característica de V, notada X_V, se define como:

$$X_V : \mathbb{N}^k \rightarrow \mathbb{N}$$

$$X_V(\underline{x}) = \begin{cases} 1 & \text{si } \underline{x} \in V \\ 0 & \text{si } \underline{x} \notin V \end{cases}$$

Nota: Si f es total se cumple que $X_{V_{P_f}} = sg(f)$, siendo sg la función signo (es decir, $sg(0)=0$ y $sg(n)=1\ \forall n>0$).

Definición 9.19: *Función característica de un predicado*
Dado un predicado P, llamaremos a X_{V_P} función característica del predicado P.

Nota: Cuando el contexto elimina la ambigüedad, es habitual sobrecargar el nombre de un predicado llamando a su función característica de la misma forma.

El concepto intuitivo de función computable introducido en el primer tema corresponde aquí exactamente con el concepto formal ya definido de función recursiva. Respecto a los conceptos intuitivos de conjunto decidible y conjunto enumerable (o generable), las denominaciones de los correspondientes conceptos formales incluirán la palabra "recursivamente" para indicar que la formalización se hace con funciones recursivas.

Comenzaremos definiendo los conceptos de predicado recursivamente decidible y predicado recursivamente enumerable.

Definición 9.20: *Predicado recursivamente decidible, PRED(TREC)*
Definimos el conjunto *PRED(TREC)* como:

$$PRED(TREC) = \{ P_f \mid f \in TREC \}$$

Diremos que P es un predicado recursivamente decidible sii $P \in PRED(TREC)$.

Notas: Un predicado es recursivamente decidible si es el predicado asociado de alguna función recursiva total.
PRED(TREC), por tanto, es el conjunto de todos los predicados recursivamente decidibles.

Definición 9.21: *Predicado recursivamente enumerable, PRED(REC)*
Definimos el conjunto *PRED(REC)* como:

$$PRED(REC) = \{ P_f \mid f \in REC \}$$

Diremos que P es un predicado recursivamente enumerable sii $P \in PRED(REC)$.

Notas: Un predicado es recursivamente enumerable si es el predicado asociado de alguna función recursiva.
PRED(REC), por tanto, es el conjunto de todos los predicados recursivamente enumerables.
Se cumple que $PRED(TREC) \subset PRED(REC)$ ya que $TREC \subset REC$.

Definición 9.22: *Conjunto recursivamente decidible, DECI*
Definimos el conjunto *DECI* como:

$$DECI = \{ V_P \mid P \in PRED(TREC) \}$$

Diremos que $V \subseteq \mathbb{N}^k$ es un conjunto recursivamente decidible sii $V \in DECI$.

Notas: Un conjunto es recursivamente decidible si es el conjunto de valores de verdad de algún predicado recursivamente decidible.
DECI, por tanto, es el conjunto de todos los conjuntos recursivamente decidibles.

Definición 9.23: *Conjunto recursivamente enumerable, ENUM*
Definimos el conjunto *ENUM* como:

$$ENUM = \{ V_P \mid P \in PRED(REC) \}$$

Diremos que $V \subseteq \mathbb{N}^k$ es un conjunto recursivamente enumerable sii $V \in ENUM$.

Notas: Un conjunto es recursivamente enumerable si es el conjunto de valores de verdad de algún predicado recursivamente enumerable.
ENUM, por tanto, es el conjunto de todos los conjuntos recursivamente enumerables.
Se cumple que $DECI \subset ENUM$ ya que $PRED(TREC) \subset PRED(REC)$.

Definición 9.24: *Conjunto recursivamente generable*
Diremos que V es un conjunto recursivamente generable sii V es recursivamente enumerable.

Se puede demostrar que las operaciones aritméticas habituales, así como todas las que podemos obtener a partir de ellas componiéndolas (de manera finita) son todas ellas recursivas. En realidad se asume como cierta la tesis (que es parte de la de Church) de que toda función computable en el sentido intuitivo es recursiva, y viceversa. De hecho esta asunción se extiende no sólo a REC , sino también a $DECI$ y $ENUM$ en relación con sus correspondientes conceptos intuitivos (decidible y enumerable respectivamente).

Antes de terminar este apartado recordar que, como ya se comentó en los temas anteriores, es posible definir estos últimos conceptos (los definidos a partir del de función recursiva) no sólo para funciones y conjuntos de vectores de naturales sino, de una forma más general, para funciones y conjuntos de cadenas sobre un alfabeto.

9.5 Algunas consideraciones

Se puede demostrar, aunque no lo hagamos aquí (por razones de espacio y propósitos del presente libro), que $REC = F(MT)$, y derivado de ello también que $DECI = DEC(MT)$ y $ENUM = ENU(MT)$.

Sobre este asunto volveremos más adelante al inicio del capítulo siguiente, cuando hablemos del Teorema de Equivalencia.

Señalar por último, a modo de resumen, que en este tema hemos presentado el segundo modelo de cómputo, de los tres que veremos, que en este caso formaliza directamente el concepto intuitivo de función computable: la función recursiva. Históricamente también fué de los primeros en desarrollarse, y aunque a nivel general es menos conocido, ha sido fundamental en el ámbito matemático. Basándonos en él hemos formalizado también los conceptos intuitivos de decidible y enumerable.

10
El lenguaje *WHILE*

10.1 Concepto informal

En este tema, al igual que en el dedicado a Máquinas de Turing, para formalizar el concepto de función computable (*WHILE*-computable) hemos definido previamente un modelo de cómputo (el lenguaje *WHILE*).

El *lenguaje WHILE* se puede entender como un lenguaje de programación muy simplificado. Sólo trabaja con naturales, y únicamente tiene cuatro tipos de *sentencias de asignación*, y un tipo de *sentencia de control*. Las de asignación son: asignar a una variable la constante cero, asignar a una variable el valor de otra variable, asignar a una variable el valor de otra variable incrementada en una unidad, y asignar a una variable el valor de otra variable decrementada en una unidad. La de control es una sentencia de tipo bucle indefinido (sentencia "while"), la cual además tiene siempre una condición de control muy simple: comprobar si el valor de una variable (denominada de control) es distinto de cero.

A continuación vamos a describir de manera informal la sintaxis del lenguaje *WHILE*, y después su semántica, o lo que es lo mismo, el funcionamiento de cualquier programa escrito en dicho lenguaje. Posteriormente en el apartado siguiente formalizaremos estos conceptos.

En un programa escrito en el lenguaje *WHILE* (abreviadamente *programa WHILE*) los identificadores de variables tienen la forma:

$X_1, X_2, X_3, \ldots, X_i, \ldots$

Las sentencias de asignación tienen la forma:

$X_i := 0$
$X_i := X_j$
$X_i := X_j + 1$
$X_i := X_j - 1$

La sentencia de control es de tipo bucle indefinido, con la forma:

while $X_i \neq 0$ *do*	(es la llamada *cabecera* del bucle)
código	(es el llamado *cuerpo* del bucle)
od	(es la llamada *cola* del bucle)

Un *código* es una secuencia finita no vacía de sentencias separadas por el símbolo ";".

Nota: Es habitual utilizar de forma indistinta los términos "bucle", "bucle *while*" o simplemente "*while*" para referirse a la sentencia de control.

Esta es la sencilla sintaxis del código de un programa *WHILE*. Decimos del código ya que un programa para nosotros es algo más que un código. Un programa *WHILE* es una terna (n, p, s) donde n y p son naturales con $p \geq n$ y $p > 0$; y s es un código según la sintaxis anteriormente descrita.

Si un programa *WHILE* es una terna (n, p, s) esto significa que el programa tiene p *variables de uso* en total (X_1 , X_2 , X_3 , ... , X_p) que toman valores naturales. De ellas n son *variables de entrada* (X_1 , X_2 , X_3 , ... , X_n), denominándose a las restantes *variables auxiliares*. Estas últimas variables (un total de $p - n$) se inicializan implícitamente a cero al inicio de la ejecución del programa. No hay instrucciones de entrada ni de salida. La salida del programa es el valor de la variable X_1 cuando el programa termina.

El funcionamiento de un programa *WHILE* consiste, dado que es un modelo basado en un lenguaje estructurado, en ejecutar secuencialmente las sentencias que componen su código, empezando por la primera del mismo. Esto supone que se ejecuta una sentencia cuando la anterior ha sido terminada. Cómo se ejecutan las sentencias de asignación, no necesita más aclaración salvo señalar que la sentencia de decremento ($X_i := X_j - 1$) asigna el valor cero a la variable resultante en el caso de que la variable decrementada valiera cero antes de la sentencia. Respecto a la sentencia de control (*while*), ésta primero comprueba si la variable de control que aparece en su cabecera es distinta de cero; si es así ejecutará el código que compone el cuerpo del bucle, tras lo cual volverá a comprobar la variable. Esto se repetirá mientras dicha variable sea distinta de cero cada vez que sea comprobada. Si en una de las comprobaciones vale cero la sentencia *while* termina.

Podemos por tanto asociar una única función de $\mathbb{N}^n$ en $\mathbb{N}$ a cada programa *WHILE*. Esta función, que es la que más adelante definiremos como función calculada por un programa *WHILE*, puede ser parcial, ya que puede haber entradas para las cuales el programa no termina (diverge).

10.2 Definición formal

Para realizar la definición formal de este modelo usaremos algunos conceptos relacionados con los lenguajes formales, como los de gramática y lenguaje generado, entre otros. En el primer capítulo se definen los conceptos que aquí usaremos. Si el lector quiere ampliar sus conocimientos sobre ellos puede consultar el libro *Teoría de Autómatas y Lenguajes Formales I* (Gonzalo Ramos Jiménez, 2005).

Para definir algunas gramáticas auxiliares que vamos a necesitar, en primer lugar introducimos dos alfabetos que usaremos en ellas, como son:

$\Sigma_d = \{ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 \}$
$\Sigma_1 = \{ X , := , + , - , 1 , while , \neq , 0 , do , od , ; \}$

Notas: En los alfabetos definidos los símbolos van separados por comas. Esto quiere decir que por ejemplo "*while*" es un solo símbolo perteneciente al alfabeto Σ_1 , que tiene por tanto once símbolos.
El alfabeto Σ_d está formado por los dígitos en subíndice.

Sea $G_1 = (N_1 , T_1 , P_1 , \langle número \rangle)$ una gramática con:

$N_1 = \{ \langle número \rangle , \langle p\_dígito \rangle , \langle r\_número \rangle , \langle dígito \rangle \}$
$T_1 = \Sigma_d$
$P_1 = \{ \langle número \rangle \rightarrow \langle p\_dígito \rangle$
$\quad | \langle p\_dígito \rangle \langle r\_número \rangle ,$
$\langle p\_dígito \rangle \rightarrow 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ,$
$\langle r\_número \rangle \rightarrow \langle dígito \rangle$
$\quad | \langle dígito \rangle \langle r\_número \rangle ,$
$\langle dígito \rangle \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \}$

Notas: Las letras "*p*" y "*r*" corresponderían a las palabras "*primer*" y "*resto*" respectivamente.
Esta gramática genera (en subíndice) los naturales mayores que cero en notación decimal (sin ceros no significativos por la izquierda).

Sea $G_2 = (N_2 , T_2 , P_2 , \langle código \rangle)$ una gramática con:

$N_2 = \{ \langle código \rangle , \langle sentencia \rangle , \langle asignación \rangle , \langle bucle \rangle ,$
$\langle identificador \rangle \} \cup N_1$

$T_2 = \Sigma_d \cup \Sigma_1$

$P_2 = \{ \langle código \rangle \rightarrow \langle sentencia \rangle$
$\quad | \langle código \rangle ; \langle sentencia \rangle ,$
$\langle sentencia \rangle \rightarrow \langle asignación \rangle$
$\quad | \langle bucle \rangle ,$
$\langle asignación \rangle \rightarrow \langle identificador \rangle := 0$
$\quad | \langle identificador \rangle := \langle identificador \rangle$
$\quad | \langle identificador \rangle := \langle identificador \rangle + 1$
$\quad | \langle identificador \rangle := \langle identificador \rangle - 1 ,$
$\langle bucle \rangle \rightarrow while \langle identificador \rangle \neq 0 \; do \langle código \rangle \; od ,$
$\langle identificador \rangle \rightarrow X \langle número \rangle \} \cup P_1$

Definición 10.1: *Código While, Conjunto de códigos While (Cód_While)*
Sea $s \in T_2^+$. Diremos que s es un código *While* sii $s \in L(G_2)$.
El conjunto de códigos *While*, notado *Cód_While* , se define como:

$$Cód\_While = L(G_2)$$

Nota: El conjunto *Cód_While* está formado por todos los códigos en lenguaje *WHILE* sintácticamente correctos.

Antes de definir el concepto de programa *WHILE* señalar que, de ahora en adelante, asumiremos que una cadena de dígitos (sin ceros no significativos por la izquierda) tiene como valor numérico el natural cuya representación decimal coincide con ella.

Definición 10.2: *Programa WHILE, Conjunto de programas WHILE (WHILE)*
El conjunto de programas *WHILE*, notado $WHILE$, se define como:

$$WHILE = \{ (n, p, s) \in \mathbb{N} \times \mathbb{N} \times Cód\_While \mid p \geq n \wedge p > 0 \wedge (X_m \prec s \text{ con } m \in \Sigma_d^+ \Rightarrow m \leq p) \}$$

Diremos que Q es un programa *WHILE* sii $Q \in WHILE$.

Nota: La expresión "$X_m \prec s$" significa que la cadena "X_m" es subcadena de la cadena "s". En adelante usaremos el símbolo "$\prec$" para indicar que una cadena es subcadena de otra.

Hasta aquí la definición formal de lo que es un programa *WHILE*, es decir, su sintaxis. Para definir formalmente su funcionamiento, es decir, su semántica, tenemos que introducir algunos conceptos adicionales.

Definición 10.3: *Tamaño de un código While (tam)*
Sea $s \in Cód\_While$, definimos el tamaño de s , notado $tam(s)$, como:

a) Si en G_2 se cumple que $<asignación> \Rightarrow^* s$, entonces:
$$tam(s) = 1$$

b) Si $s = while\ X_i \neq 0\ do\ s_1\ od$, con $i \in \Sigma_d^+$ y $s_1 \in Cód\_While$, entonces:
$$tam(s) = tam(s_1) + 2$$

c) Si $s = s_1 ; s_2$, con $s_1 , s_2 \in Cód\_While$, entonces:
$$tam(s) = tam(s_1) + tam(s_2)$$

Definición 10.4: *Tamaño de un programa WHILE (tam)*
Sea $Q \in WHILE$, con $Q = (n, p, s)$.
Definimos el tamaño de Q , notado $tam(Q)$, como:
$$tam(Q) = tam(s)$$

Nota: La sobrecarga definida anteriormente para la función tam no representa ningún problema de legibilidad, ya que siempre podemos distinguir el tipo de argumento al que se aplica.

Definición 10.5: *Función línea*
Definimos la función $línea : Cód\_While \times \mathbb{N} \to T_2^*$ como:

a) Si en G_2 se cumple que $<asignación> \Rightarrow^* s$, entonces:

$$línea(s,n) = \begin{cases} s & si \quad n = 1 \\ \varepsilon & en\ otro\ caso \end{cases}$$

b) Si $s = while\ X_i \neq 0\ do\ s_1\ od$, con $i \in \Sigma_d^+$ y $s_1 \in Cód\_While$, entonces:

$$línea(s,n) = \begin{cases} while\ Xi \neq 0\ do & si \quad n = 1 \\ od & si \quad n = tam(s) \\ línea(s_1, n-1) & si \quad 1 < n < tam(s) \\ \varepsilon & en\ otro\ caso \end{cases}$$

c) Si $s = s_1 ; s_2$, con $s_1 , s_2 \in Cód\_While$, entonces:

$$línea(s,n) = \begin{cases} línea(s_1, n) & si \quad 0 < n \leq tam(s_1) \\ línea(s_2, n - tam(s_1)) & si \quad tam(s_1) < n \leq tam(s) \\ \varepsilon & en\ otro\ caso \end{cases}$$

Nota: La función *línea* devuelve el texto de una línea concreta de un código *While*, o bien la cadena vacía si el número de línea no es válido.

Definición 10.6: *Función salto*
Definimos la función $salto : Cód\_While \times \mathbb{N} \to \mathbb{N}$ como:

a) Si en G_2 se cumple que $<asignación> \Rightarrow^* s$, entonces:

$$salto(s,n) = 0$$

b) Si $s = while\ X_i \neq 0\ do\ s_1\ od$, con $i \in \Sigma_d^+$ y $s_1 \in Cód\_While$, entonces:

$$salto(s,n) = \begin{cases} tam(s)+1 & si \quad n = 1 \\ 1 & si \quad n = tam(s) \\ salto(s_1, n-1)+1 & si \quad (1 < n < tam(s)) \wedge (salto(s_1, n-1) \neq 0) \\ 0 & en\ otro\ caso \end{cases}$$

c) Si $s = s_1 ; s_2$, con $s_1 , s_2 \in Cód\_While$, entonces:

$$salto(s,n) = \begin{cases} salto(s_1,n) \quad si \quad 0 < n \le tam(s_1) \\ salto(s_2, n - tam(s_1)) + tam(s_1) \quad si \\ \qquad (tam(s_1) < n \le tam(s)) \wedge (salto(s_2, n - tam(s_1)) \ne 0) \\ 0 \quad en\ otro\ caso \end{cases}$$

Nota: La función *salto* devuelve el número de línea no consecutiva al que se podría saltar desde una línea dada de un código *While*, o bien cero si desde la línea especificada no se puede saltar a una línea no consecutiva.

Definición 10.7: *Configuración, Conjunto de configuraciones*
Sea $Q \in WHILE$, con $Q = (n, p, s)$.
Definimos el conjunto de configuraciones de Q , notado C_Q , como:
$C_Q = \{ \underline{c} \in \mathbb{N}^{p+1} \mid \underline{c} = (m, \underline{x}) \quad \text{con} \quad 1 \le m \le tam(Q)+1 \quad \text{y} \quad \underline{x} \in \mathbb{N}^p \}$

Diremos que $\underline{c}$ es una configuración de Q sii $c \in C_Q$.

Definición 10.8: *Configuración inicial*
Sea $Q \in WHILE$, con $Q = (n, p, s)$, y sea $\underline{c} \in C_Q$, con $\underline{c} = (m, \underline{x})$.
Diremos que $\underline{c}$ es configuración inicial de Q sii

$$m = 1 \quad \wedge \quad x_{n+1} = \ldots = x_p = 0$$

Definición 10.9: *Configuración terminal, Configuración no terminal*
Sea $Q \in WHILE$, y sea $\underline{c} \in C_Q$, con $\underline{c} = (m, \underline{x})$.
Diremos que $\underline{c}$ es configuración terminal de Q sii $m = tam(Q)+1$
En caso contrario decimos que es una configuración no terminal.

Nota: Una configuración no puede ser a la vez inicial y terminal.

Como en toda máquina abstracta, que es lo que en definitiva es un programa *WHILE*, el concepto de configuración representa todo lo que necesitamos saber acerca de la situación de dicha máquina (línea y contenido de las variables) para poder determinar la nueva situación después de que la máquina "dé un paso" (hablamos de máquinas discretas y deterministas). Así pues, a continuación vamos a definir en qué consiste "dar un paso" en un programa *WHILE*.

La denominación formal aquí, al igual que en el modelo de MT, al concepto "dar un paso" es "transitar directamente", y define en su nivel más atómico la semántica de un programa *WHILE*.

Definición 10.10: *Transitar directamente*

Sea $Q \in WHILE$, con $Q = (n, p, s)$.

Sean $\underline{c}_1, \underline{c}_2 \in C_Q$, con $\underline{c}_1 = (m, \underline{x})$ y $\underline{c}_2 = (t, \underline{z})$.

Diremos que Q transita directamente de $\underline{c}_1$ a $\underline{c}_2$, notado $\underline{c}_1 \vdash\!\!\!- \underline{c}_2$, sii

$$línea(s, m) = X_i{:=}0 \quad \text{con} \quad i \in \Sigma_d^+$$
$$\Rightarrow \quad t = m+1 \quad \wedge \quad z_i = 0 \quad \wedge \quad (z_r = x_r \quad \forall r \neq i)$$
$$\wedge$$
$$línea(s, m) = X_i{:=}X_j \quad \text{con} \quad i, j \in \Sigma_d^+$$
$$\Rightarrow \quad t = m+1 \quad \wedge \quad z_i = x_j \quad \wedge \quad (z_r = x_r \quad \forall r \neq i)$$
$$\wedge$$
$$línea(s, m) = X_i{:=}X_j + 1 \quad \text{con} \quad i, j \in \Sigma_d^+$$
$$\Rightarrow \quad t = m+1 \quad \wedge \quad z_i = x_j + 1 \quad \wedge \quad (z_r = x_r \quad \forall r \neq i)$$
$$\wedge$$
$$línea(s, m) = X_i{:=}X_j - 1 \quad \text{con} \quad i, j \in \Sigma_d^+$$
$$\Rightarrow \quad t = m+1 \quad \wedge \quad z_i = x_j - 1 \quad \wedge \quad (z_r = x_r \quad \forall r \neq i)$$
$$\wedge$$
$$línea(s, m) = while\ X_i \neq 0\ do \quad \text{con} \quad i \in \Sigma_d^+$$
$$\Rightarrow \quad (x_i \neq 0 \ \Rightarrow \ t = m+1) \quad \wedge \quad (x_i = 0 \ \Rightarrow \ t = salto(s,m)) \quad \wedge \quad \underline{z} = \underline{x}$$
$$\wedge$$
$$línea(s, m) = od$$
$$\Rightarrow \quad t = salto(s,m) \quad \wedge \quad \underline{z} = \underline{x}$$

Notas: Las asignaciones realizan la asignación pedida sobre la variable adecuada y pasan a la línea siguiente.

La cabecera de bucle no toca el contenido de las variables y pasa a la línea siguiente si la variable de control del bucle es distinta de cero, y a la línea siguiente a su cola si la variable de control del bucle es cero.

La cola de bucle no toca el contenido de las variables y pasa a la línea de su cabecera.

A continuación vamos a definir una serie de funciones asociadas a cada programa *WHILE*, que nos van a permitir pasar del concepto "transitar directamente" al de "función calculada" por un programa *WHILE*.

Definición 10.11: *Función siguiente (SIG)*
Sea $Q \in WHILE$, con $Q = (n, p, s)$.
Sea C_Q el conjunto de configuraciones de Q.
Definimos la función siguiente de Q, notada SIG_Q, como:

$$SIG_Q : \mathbb{N}^{p+1} \to \mathbb{N}^{p+1}$$

$$SIG_Q(\underline{c}) = \begin{cases} \underline{c}' & \text{si } \underline{c} \vdash \underline{c}' \text{ con } \underline{c}, \underline{c}' \in C_Q \\ \underline{c} & \text{si } \nexists\, \underline{c}' \in C_Q \mid \underline{c} \vdash \underline{c}' \text{ con } \underline{c} \in C_Q \\ \underline{c} & \text{si } \underline{c} \notin C_Q \end{cases}$$

Nota: Amplía el concepto "transitar directamente" dándonos una función total.

Definición 10.12: *Función cálculo (CAL)*
Sea $Q \in WHILE$, con $Q = (n, p, s)$.
Definimos la función cálculo de Q, notada CAL_Q, como:

$$CAL_Q : \mathbb{N}^{n+1} \to \mathbb{N}^{p+1}$$

$$CAL_Q(\underline{x}, i) = \begin{cases} (1, \underline{x}, \underline{0}) & \text{si } i = 0 \\ SIG_Q(CAL_Q(\underline{x}, i-1)) & \text{si } i > 0 \end{cases}$$

Nota: $i \in \mathbb{N}$, $\underline{x} \in \mathbb{N}^n$ y $\underline{0} \in \mathbb{N}^{p-n}$.

Definición 10.13: *Función complejidad temporal (T)*
Sea $Q \in WHILE$, con $Q = (n, p, s)$.
Definimos la función complejidad temporal de Q, notada T_Q, como:

$$T_Q : \mathbb{N}^n \to \mathbb{N}$$

$$T_Q(\underline{x}) = \begin{cases} mínimo(A) & \text{si } A \neq \varnothing \\ \uparrow & \text{si } A = \varnothing \end{cases}$$

donde $A = \{ j \in \mathbb{N} \mid \pi_1^{p+1}(CAL_Q(\underline{x}, j)) = tam(Q)+1 \}$

Nota: $\underline{x} \in \mathbb{N}^n$.

Definición 10.14: *Función calculada (F)*
Sea $Q \in WHILE$, con $Q = (n, p, s)$.
Definimos la función calculada de Q , notada F_Q , como:
$F_Q : \mathbb{N}^n \to \mathbb{N}$
$F_Q(\underline{x}) = \pi_2^{p+1}(CAL_Q(\underline{x}, T_Q(\underline{x})))$

Nota: $\underline{x} \in \mathbb{N}^n$ y $j \in \mathbb{N}$.

10.3 Ejemplos de programas *WHILE*

Ejemplo 10.15:
Sea $Q = (1, 2, s)$ con s :

$X_2 := X_1$;
while $X_2 \neq 0$ *do*
$X_1 := X_1 + 1$;
$X_2 := X_2 - 1$
od

$Q \in WHILE$, es decir, es un programa *WHILE*, ya que se cumple que:

$Q \in \mathbb{N} \times \mathbb{N} \times Cód\_While \quad \wedge$
$2 \geq 1 \quad \wedge$
$2 > 0 \quad \wedge$
$(\ X_m \prec s$ con $m \in \Sigma_d^+ \ \Rightarrow \ m \leq 2 \ )$

El hecho de que la tercera componente de Q , es decir, la cadena s , pertenezca a $Cód\_While$ significa que s es un código en lenguaje *WHILE* sintácticamente correcto, cumpliéndose que $tam(Q)=tam(s)=5$.
Veamos a continuación una serie de afirmaciones acerca del programa Q .

El vector $(1,1,1)$ es una configuración de Q , es decir, $(1,1,1) \in C_Q$. No es una configuración inicial, ya que la tercera componente no es cero, ni terminal, puesto que la primera componente no es seis.
El vector $(0,1,1)$ no es una configuración de Q , es decir, $(0,1,1) \notin C_Q$, ya que $0 \notin [1, 6]$.
El vector $(7,1,1)$ no es una configuración de Q , es decir, $(7,1,1) \notin C_Q$, ya que $7 \notin [1, 6]$.

El vector $(1,10,0) \in C_Q$, siendo además configuración inicial.
El vector $(6,6,6) \in C_Q$, siendo además configuración terminal.

Se cumple que $(1,1,1) \vdash (2,1,1)$, es decir, Q transita directamente de $(1,1,1)$ a $(2,1,1)$, y también que $(1,10,0) \vdash (2,10,10)$.
Sin embargo el vector $(0,1,1)$ no transita directamente a ninguna configuración al no serlo él, lo mismo que el vector $(7,1,1)$.
La configuración $(6,6,6)$, al ser terminal, no transita directamente a ninguna otra.

Si aplicamos la función siguiente a estos vectores tenemos:
$SIG_Q(1,1,1)=(2,1,1)$.
$SIG_Q(0,1,1)=(0,1,1)$.
$SIG_Q(7,1,1)=(7,1,1)$.
$SIG_Q(1,10,0)=(2,10,10)$.
$SIG_Q(6,6,6)=(6,6,6)$.

Si aplicamos la función cálculo al vector $(2,4)$ tenemos:
$CAL_Q(2,4) =$
$SIG_Q(CAL_Q(2,3)) =$
$SIG_Q(SIG_Q(CAL_Q(2,2))) =$
$SIG_Q(SIG_Q(SIG_Q(CAL_Q(2,1)))) =$
$SIG_Q(SIG_Q(SIG_Q(SIG_Q(CAL_Q(2,0))))) =$
$SIG_Q(SIG_Q(SIG_Q(SIG_Q(1,2,0)))) =$
$SIG_Q(SIG_Q(SIG_Q(2,2,2))) =$
$SIG_Q(SIG_Q(3,2,2)) =$
$SIG_Q(4,3,2) = (5,3,1)$

Si aplicamos la función complejidad temporal al vector (2) tenemos:
$T_Q(2) = 10$
puesto que:
$mínimo\{ j \in \mathbb{N} \mid \pi_1^3(CAL_Q(2,j)) = 6 \} = 10$

Si aplicamos la función calculada al vector (2) tenemos:
$F_Q(2) =$
$\pi_2^3(CAL_Q(2, T_Q(2))) =$

$\pi_2^3(CAL_Q(2,10)) =$
$\pi_2^3(SIG_Q(CAL_Q(2,9))) =$
$\pi_2^3(SIG_Q(SIG_Q(CAL_Q(2,8)))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(CAL_Q(2,7))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(CAL_Q(2,6)))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(CAL_Q(2,5))))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(CAL_Q(2,4)))))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(CAL_Q(2,3))))))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(CAL_Q(2,2)))))))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(CAL_Q(2,1))))))))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(CAL_Q(2,0)))))))))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(1,2,0)))))))))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(2,2,2))))))))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(3,2,2)))))))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(4,3,2))))))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(5,3,1)))))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(2,3,1))))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(SIG_Q(3,3,1)))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(SIG_Q(4,4,1))))) =$
$\pi_2^3(SIG_Q(SIG_Q(SIG_Q(5,4,0)))) =$
$\pi_2^3(SIG_Q(SIG_Q(2,4,0))) = \pi_2^3(6,4,0) = 4$

Este cálculo también se puede expresar de forma más abreviada como:
(1,2,0) |— (2,2,2) |— (3,2,2) |— (4,3,2) |— (5,3,1) |— (2,3,1) |— (3,3,1) |— (4,4,1) |— (5,4,0) |— (2,4,0) |— (6,4,0)
siendo por tanto el resultado *4* , ya que es el valor de la segunda componente (es decir, de la variable uno) de la configuración terminal.

Dado que el programa es sencillo podemos hacer un estudio genérico de su comportamiento.

$$SIG_Q(a,b,c) = \begin{cases} (0,b,c) & \text{si } a = 0 \\ (2,b,b) & \text{si } a = 1 \\ (3,b,c) & \text{si } a = 2 \wedge c \neq 0 \\ (6,b,c) & \text{si } a = 2 \wedge c = 0 \\ (4,b+1,c) & \text{si } a = 3 \\ (5,b,c-1) & \text{si } a = 4 \\ (2,b,c) & \text{si } a = 5 \\ (a,b,c) & \text{si } a > 5 \end{cases}$$

Si comenzamos el cómputo con la configuración inicial $(1, n, 0)$, con $n > 0$, tenemos:

$(1, n, 0) \vdash (2, n, n) \vdash (3, n, n) \vdash$
$(4, n+1, n) \vdash (5, n+1, n-1) \vdash (2, n+1, n-1) \vdash (3, n+1, n-1) \vdash$
$(4, n+2, n-1) \vdash (5, n+2, n-2) \vdash (2, n+2, n-2) \vdash (3, n+2, n-2) \vdash$
$\cdots$
$(4, 2n-1, 2) \vdash (5, 2n-1, 1) \vdash (2, 2n-1, 1) \vdash (3, 2n-1, 1) \vdash$
$(4, 2n, 1) \vdash (5, 2n, 0) \vdash (2, 2n, 0) \vdash (6, 2n, 0)$

y para $n = 0$ tenemos:
$(1, 0, 0) \vdash (2, 0, 0) \vdash (6, 0, 0)$

Por tanto:
$T_Q(n) = 4n+2 \quad \forall n \in \mathbb{N}$
$F_Q(n) = 2n \quad \forall n \in \mathbb{N}$ □

El análisis último que hemos realizado en el ejemplo anterior para obtener una expresión aritmética de la función complejidad temporal y de la función calculada de un programa, sólo es posible cuando tenemos un programa sencillo, ya que sino en última instancia lo que obtendríamos sería una expresión tan compleja como el programa.

Tampoco hay una forma genérica de saber cuanto vale la función complejidad temporal de un programa para un argumento dado. Para evaluarla debemos ejecutar el programa con dicho argumento como entrada hasta que se alcance (si es el caso) una configuración terminal, siendo el valor de la función complejidad temporal el número de transiciones directas realizadas durante el cómputo. En el caso de que no se pueda alcanzar una configuración terminal

tanto la función complejidad temporal como la función calculada divergen para ese argumento.

Ejemplo 10.16:

Sea $Q = (1, 1, s)$ con s :

$X_1 := X_1 + 1$;
while $X_1 \neq 0$ *do*
$X_1 := X_1$
od

$Q \in$ *WHILE* , es decir, es un programa *WHILE*, ya que se cumple que:

$Q \in \mathbb{N} \times \mathbb{N} \times$ *Cód_While* $\wedge$
$1 \geq 1 \quad \wedge$
$1 > 0 \quad \wedge$
$(X_m \prec s$ con $m \in \Sigma_d^+ \Rightarrow m \leq 1)$

El hecho de que la tercera componente de Q , es decir, la cadena s , pertenezca a *Cód_While* significa que s es un código en lenguaje *WHILE* sintácticamente correcto, cumpliéndose que $tam(Q)=tam(s)=4$.

Veamos a continuación una serie de afirmaciones acerca del programa Q .

El vector $(2,0)$ es una configuración de Q , es decir, $(2,0) \in C_Q$. No es una configuración inicial, ya que la primera componente no es uno, ni terminal, puesto que la primera componente no es cinco.

El vector $(0,1)$ no es una configuración de Q , es decir, $(0,1) \notin C_Q$, ya que $0 \notin [1, 5]$.

El vector $(6,1)$ no es una configuración de Q , es decir, $(6,1) \notin C_Q$, ya que $6 \notin [1, 5]$.

El vector $(1,10) \in C_Q$, siendo además configuración inicial.

El vector $(5,5) \in C_Q$, siendo además configuración terminal.

Se cumple que $(2,0) \vdash (5,0)$, es decir, Q transita directamente de $(2,0)$ a $(5,0)$, y también que $(1,10) \vdash (2,11)$.

Sin embargo el vector $(0,1)$ no transita directamente a ninguna configuración al no serlo él, lo mismo que el vector $(6,1)$.

La configuración $(5,5)$, al ser terminal, no transita directamente a ninguna otra.

Si aplicamos la función siguiente a estos vectores tenemos:
$SIG_Q(2,0)=(5,0)$.
$SIG_Q(0,1)=(0,1)$.
$SIG_Q(6,1)=(6,1)$.
$SIG_Q(1,10)=(2,11)$.
$SIG_Q(5,5)=(5,5)$.

Si aplicamos la función cálculo al vector $(2,4)$ tenemos:
$CAL_Q(2,4) =$
$SIG_Q(CAL_Q(2,3)) =$
$SIG_Q(SIG_Q(CAL_Q(2,2))) =$
$SIG_Q(SIG_Q(SIG_Q(CAL_Q(2,1)))) =$
$SIG_Q(SIG_Q(SIG_Q(SIG_Q(CAL_Q(2,0))))) =$
$SIG_Q(SIG_Q(SIG_Q(SIG_Q(1,2)))) =$
$SIG_Q(SIG_Q(SIG_Q(2,3))) =$
$SIG_Q(SIG_Q(3,3)) =$
$SIG_Q(4,3) = (2,3)$

Si aplicamos la función complejidad temporal al vector (2) tenemos:
$T_Q(2) = \uparrow$
ya que en este caso, si nos fijamos, la configuración $(2,3)$ apararece dos veces en el cálculo anterior, lo que indica la existencia de un bucle, y por tanto:
$\{ j \in \mathbb{N} \mid \pi_1^2(CAL_Q(2,j)) = 5 \} = \varnothing$
También, por ello, si aplicamos la función calculada al mismo vector tenemos:
$F_Q(2) = \uparrow$

Dado que este programa también es sencillo podemos hacer un estudio genérico de su comportamiento.

$$SIG_Q(a,b) = \begin{cases} (0,b) & \text{si } a = 0 \\ (2,b+1) & \text{si } a = 1 \\ (3,b) & \text{si } a = 2 \wedge b \neq 0 \\ (5,b) & \text{si } a = 2 \wedge b = 0 \\ (4,b) & \text{si } a = 3 \\ (2,b) & \text{si } a = 4 \\ (a,b) & \text{si } a > 4 \end{cases}$$

Si comenzamos el cómputo con la configuración inicial $(1, n)$ tenemos:
$(1, n) \vdash (2, n+1) \vdash (3, n+1) \vdash (4, n+1) \vdash (2, n+1) \vdash \cdots$
y puesto que la configuración $(2, n+1)$ apararece dos veces tenemos un bucle, y por tanto:
$T_Q(n) = \uparrow \quad \forall n \in \mathbb{N}$
$F_Q(n) = \uparrow \quad \forall n \in \mathbb{N}$ □

A continuación veremos algunos ejemplos más de programas *WHILE* mostrando sólo el programa y su función calculada.

Ejemplo 10.17:
Sea el programa *WHILE* $Q = (1, 1, s)$ con s :

$X_1 := X_1 + 1$;
$X_1 := X_1 + 1$

$F_Q(n) = n+2 \quad \forall n \in \mathbb{N}$ (es la función *sumados* del ejemplo **9.11**) □

Ejemplo 10.18:
Sea el programa *WHILE* $Suma = (2, 2, s)$ con s :

while $X_2 \neq 0$ *do*
$X_1 := X_1 + 1$;
$X_2 := X_2 - 1$
od

$F_{Suma} : \mathbb{N}^2 \to \mathbb{N}$
$F_{Suma}(n,m) = n+m \quad \forall n,m \in \mathbb{N}$ (es la función *suma* del ejemplo **9.12**) □

Ejemplo 10.19:
Sea el programa *WHILE* $Uncero = (1, 1, s)$ con s :

while $X_1 \neq 0$ *do*
$X_1 := X_1$
od

$F_{Uncero} : \mathbb{N} \to \mathbb{N}$
$\begin{cases} F_{Uncero}(0) = 0 \\ F_{Uncero}(n) = \uparrow \quad \text{si} \quad n > 0 \end{cases}$ (es la función *uncero* del ejemplo **9.14**) □

10.4 Computabilidad, decidibilidad y enumerabilidad con *WHILE*

De nuevo, como ya señalamos en los temas anteriores, para definir formalmente estos conceptos vamos a ceñirnos a funciones que trabajan con naturales, es decir, funciones de la forma $f: \mathbb{N}^n \to \mathbb{N}$ con $n \geq 0$.

Este modelo, a diferencia de la Máquina de Turing, no necesita definiciones previas, ya que los programas *WHILE* tienen como entrada vectores de naturales, y como salida un natural. Por tanto estamos en disposición de dar una definición formal de los conceptos intuitivos, descritos en el primer tema, de función computable, conjunto decidible y conjunto enumerable (o generable).

Puesto que la formalización se hace con programas *WHILE*, estos conceptos formales se denominarán anteponiendo a su nombre la palabra "*WHILE*" seguida de un guión.

Definición 10.20: *Función WHILE-computable, F(WHILE)*
Sea la función $f: \mathbb{N}^n \to \mathbb{N}$ con $n \geq 0$.
Diremos que f es una función *WHILE*-computable sii existe un programa *WHILE* Q tal que $F_Q = f$.
En ese caso diremos que dicho programa *WHILE*-computa la función f.
Al conjunto de todas las funciones *WHILE*-computables lo llamaremos *F(WHILE)* .

Nota: Al conjunto de todas las funciones *WHILE*-computables $f: \mathbb{N}^n \to \mathbb{N}$ lo llamaremos $F^n(WHILE)$.

Definición 10.21: *T-WHILE*
$T\text{-}WHILE = \{ f \in F(WHILE) \mid f \text{ es total} \}$

Definición 10.22: *Predicado WHILE-decidible, PRED(T-WHILE)*
Definimos el conjunto *PRED(T-WHILE)* como:

$$PRED(T\text{-}WHILE) = \{ P_f \mid f \in T\text{-}WHILE \}$$

Diremos que P es un predicado *WHILE*-decidible sii $P \in PRED(T\text{-}WHILE)$.

Nota: Un predicado es *WHILE*-decidible si es el predicado asociado de alguna función *WHILE*-computable total. *PRED(T-WHILE)*, por tanto, es el conjunto de todos los predicados *WHILE*-decidibles.

Definición 10.23: *Predicado WHILE-enumerable, PRED(WHILE)*
Definimos el conjunto *PRED(WHILE)* como:

$$PRED(WHILE) = \{ P_f \mid f \in F(WHILE) \}$$

Diremos que P es un predicado *WHILE*-enumerable sii $P \in PRED(WHILE)$.

Notas: Un predicado es *WHILE*-enumerable si es el predicado asociado de alguna función *WHILE*-computable. *PRED(WHILE)*, por tanto, es el conjunto de todos los predicados *WHILE*-enumerables.
$PRED(T\text{-}WHILE) \subset PRED(WHILE)$ ya que $T\text{-}WHILE \subset F(WHILE)$.

Definición 10.24: *Conjunto WHILE-decidible, DEC(WHILE)*
Definimos el conjunto *DEC(WHILE)* como:

$$DEC(WHILE) = \{ V_P \mid P \in PRED(T\text{-}WHILE) \}$$

Diremos que $V \subseteq \mathbb{N}^k$ es un conjunto *WHILE*-decidible sii $V \in DEC(WHILE)$.

Nota: Un conjunto es *WHILE*-decidible si es el conjunto de valores de verdad de algún predicado *WHILE*-decidible. *DEC(WHILE)*, por tanto, es el conjunto de todos los conjuntos *WHILE*-decidibles.

Definición 10.25: *Conjunto WHILE-enumerable, ENU(WHILE)*
Definimos el conjunto *ENU(WHILE)* como:

$$ENU(WHILE) = \{ V_P \mid P \in PRED(WHILE) \}$$

Diremos que $V \subseteq \mathbb{N}^k$ es un conjunto *WHILE*-enumerable sii $V \in ENU(WHILE)$.

Notas: Un conjunto es *WHILE*-enumerable si es el conjunto de valores de verdad de algún predicado *WHILE*-enumerable. *ENU(WHILE)*, por tanto, es el conjunto de todos los conjuntos *WHILE*-enumerables.
Al igual que para los predicados, $DEC(WHILE) \subset ENU(WHILE)$ ya que $PRED(T\text{-}WHILE) \subset PRED(WHILE)$.

Se puede demostrar que las operaciones aritméticas habituales, así como todas las que podemos obtener a partir de ellas componiéndolas (de manera finita) son todas ellas *WHILE*-computables. En realidad se asume como cierta la

tesis (que es parte de la de Church) de que toda función computable en el sentido intuitivo es *WHILE*-computable, y viceversa. De hecho esta asunción se extiende no sólo a *F(WHILE)* , sino también a *DEC(WHILE)* y *ENU(WHILE)* en relación con sus correspondientes conceptos intuitivos (decidible y enumerable respectivamente).

Antes de terminar este apartado recordar que, como ya se comentó en los temas anteriores, es posible definir estos últimos conceptos (los definidos a partir del de función *WHILE*-computable) no sólo para funciones y conjuntos de vectores de naturales sino, de una forma más general, para funciones y conjuntos de cadenas sobre un alfabeto.

10.5 Algunas consideraciones

En el siguiente tema demostraremos que *F(WHILE) = REC* (y por tanto también *F(WHILE) = F(MT)*) , y derivado de ello tenemos a su vez que *DEC(WHILE) = DECI* y *ENU(WHILE) = ENUM* (y por tanto también *DEC(WHILE) = DEC(MT)* y *ENU(WHILE) = ENU(MT)*).

Si realizamos una tabla mostrando las denominaciones dentro de cada modelo para estos y otros conceptos comunes tenemos:

F(MT)	*REC*	*F(WHILE)*
DEC(MT)	*DECI*	*DEC(WHILE)*
ENU(MT)	*ENUM*	*ENU(WHILE)*
funciones Turing-computables totales	*TREC*	*T-WHILE*
predicados Turing-decidibles	*PRED(TREC)*	*PRED(T-WHILE)*
predicados Turing-enumerables	*PRED(REC)*	*PRED(WHILE)*

Obsérvese que los conjuntos *funciones Turing-computables totales*, *predicados Turing-decidibles* y *predicados Turing-enumerables* no fueron definidos específicamente en su momento en el tema dos debido a que no hicieron falta para definir los conceptos de conjunto Turing-decidible y conjunto

Turing-enumerable. De todas formas, por completitud, damos a continuación aquí sus definiciones y denominaciones.

Definición 10.26: *T-MT*

$T\text{-}MT = \{ f \in F(MT) \mid f \text{ es total} \}$

Definición 10.27: *Predicado Turing-decidible, PRED(T-MT)*

Definimos el conjunto *PRED(T-MT)* como:

$$PRED(T\text{-}MT) = \{ P_f \mid f \in T\text{-}MT \}$$

Diremos que P es un predicado Turing-decidible sii $P \in PRED(T\text{-}MT)$.

Nota: Un predicado es Turing-decidible si es el predicado asociado de alguna función Turing-computable total. *PRED(T-MT)*, por tanto, es el conjunto de todos los predicados Turing-decidibles.

Definición 10.28: *Predicado Turing-enumerable, PRED(MT)*

Definimos el conjunto *PRED(MT)* como:

$$PRED(MT) = \{ P_f \mid f \in F(MT) \}$$

Diremos que P es un predicado Turing-enumerable sii $P \in PRED(MT)$.

Notas: Un predicado es Turing-enumerable si es el predicado asociado de alguna función Turing-computable. *PRED(MT)*, por tanto, es el conjunto de todos los predicados Turing-enumerables.

$PRED(T\text{-}MT) \subset PRED(MT)$ ya que $T\text{-}MT \subset F(MT)$.

Si ahora completamos la tabla anterior con estas nuevas definiciones tenemos:

F(MT)	*REC*	*F(WHILE)*
DEC(MT)	*DECI*	*DEC(WHILE)*
ENU(MT)	*ENUM*	*ENU(WHILE)*
T-MT	*TREC*	*T-WHILE*
PRED(T-MT)	*PRED(TREC)*	*PRED(T-WHILE)*
PRED(MT)	*PRED(REC)*	*PRED(WHILE)*

Señalar para terminar, que en este tema hemos presentado el tercer y último modelo de cómputo de los que íbamos a ver en este texto: el lenguaje *WHILE*. Históricamente es el modelo más reciente de los tres que hemos visto, y aún no es el más extendido; pero debido a lo que podríamos llamar "sus cualidades docentes" es el que usaremos en todo el capítulo siguiente, ya que las demostraciones son mucho más cortas y legibles usando este modelo que si usamos cualquiera de los otros dos. Además este modelo, a pesar de que al igual que las funciones recursivas no es adecuado para el estudio de la complejidad espacial, que lo haríamos con máquinas de Turing, es el mejor de los tres para abordar el estudio de la complejidad temporal, aunque ambos estudios quedan fuera de los propósitos de este texto.

11
Teorema de Equivalencia

11.1 Lenguaje *WHILE* ampliado

En cualquier modelo de cómputo con el que se trabaje es conveniente disponer de expresiones abreviadas, además de las completas, para las funciones representadas, o lo que es lo mismo, para los algoritmos que las computan. Así, las MT tienen los ya citados "diagramas de MT" que permiten representar MT extensas sin tener que dar sus tablas completas (aunque se pueden deducir de dichos diagramas). Las funciones recursivas, en su propia definición, incorporan la posibilidad de expresiones abreviadas para representar funciones. Además, como ya vimos, la obtención de la expresión completa a partir de una abreviada se realiza directamente por sustitución.

El modelo de lenguaje *WHILE* también tiene la posibilidad de usar expresiones abreviadas. Para ello necesitamos ampliar el lenguaje *WHILE*, obteniendo el que denominaremos *lenguaje WHILE ampliado*.

La primera ampliación consiste en permitir en los programas en lenguaje *WHILE* ampliado el uso de *macrosentencias*, que son sentencias de asignación cuyo miembro de la derecha es una llamada a una función *WHILE*-calculable. A dichos programas se les suele denominar *macroprogramas*.

Ejemplo 11.1:
La función *suma* ($suma(n,m) = n+m \quad \forall n,m \in \mathbb{N}$) sabemos que es una función *WHILE*-calculable, ya que es calculada por el programa *Suma* del ejemplo **10.18** ($F_{Suma} = suma$), por lo que podemos usarla dentro de una macrosentencia de otro programa.
Sea, por ejemplo, el programa *(2, 3, s)* con *s* :

$X_3 := X_1$;
$X_1 := 0$;
while $X_2 \neq 0$ *do*
    $X_1 := suma(X_1, X_3)$;
    $X_2 := X_2 - 1$
od □

Esto permite reutilizar cualquier función para la cual ya tengamos un programa *WHILE*. Además, si la función tiene una representación infija habitual, podremos utilizarla; así como más de una función *WHILE*-computable en una macroinstrucción (p.e. $X_3 := X_1 + (X_1 \times (X_3 - X_2))$).

La segunda ampliación consiste en la denominación libre de las variables, lo que conlleva que se sustituya el programa como terna por sólo un código, que tiene al principio la descripción de qué variables son de entrada, y cual es la de salida. Las variables auxiliares son todas aquellas que aparecen en el código del programa y que no se han indicado como de entrada al principio.

Ejemplo 11.2:
Sea *Doble* el programa:

```
Entradas:   dato
Salida:     doble
Código:
            doble := dato ;
            while  dato ≠ 0  do
                   doble := doble + 1 ;
                   dato := dato – 1
            od                                                    □
```

El programa anterior calcula la función *doble* (al igual que el del ejemplo de **10.15** si bien en este caso la varible de entrada y de salida son distintas). Para poder insertar esa información, o cualquier otra que deseemos, dentro del propio programa, la tercera ampliación consiste en permitir incluir comentarios. Éstos irán siempre al final de las líneas de código, o bien en líneas vacías, iniciados por los símbolos "(*" y terminados por los símbolos "*)".

Ejemplo 11.3:
Sea *Sumados* el programa:

```
Entradas:   dato
Salida:     másdos          (*  = dato + 2  *)
Código:
        másdos := dato + 1 ;          (* éste es el segundo comentario *)
        (* éste es el último comentario de este programa *)
        másdos := másdos + 1                                      □
```

La cuarta y última ampliación consiste en permitir el uso de un mayor número de sentencias de control. Así, podremos usar las siguientes cuatro sentencias:

while C do
  s
od

do f(x) times
  s
od

if C then
  s
fi

if C then
  s_1
else
  s_2
fi

donde *C* es una condición booleana siempre y cuando corresponda con un predicado *WHILE*-enumerable, y *f(x)* es una función *WHILE*-computable; siendo la semántica de estas sentencias la habitual.

Llamaremos *WHILE_A* al conjunto de todos lo programas en lenguaje *WHILE* ampliado, y *F(WHILE_A)* al conjunto de todas las funciones representadas por los programas pertenecientes a *WHILE_A* .

Aunque $WHILE \subset WHILE\_A$ se cumple que $F(WHILE\_A) = F(WHILE)$, ya que a partir de cualquier programa en lenguaje *WHILE* ampliado podemos obtener otro en lenguaje *WHILE* que sea equivalente.

11.2 Recursiva ⇒ *WHILE*-calculable

El teorema de equivalencia podríamos decir que es la parte formalizable de la tesis de Church-Turing. Dicho teorema afirma que las distintas formalizaciones que históricamente se han realizado del concepto de algoritmo y función computable son equivalentes, es decir, representan lo mismo sólo que de distinta forma. Para su demostración completa tendríamos que añadir algunos modelos de cómputo a los tres ya vistos, y realizar por ejemplo una serie circular de demostraciones de inclusiones con todos los modelos. Obviamente esto excede a los objetivos del presente texto, por lo que nos limitaremos a demostrar una de las inclusiones de la equivalencia entre dos de los modelos vistos: funciones recursivas y lenguaje *WHILE*.

Teorema 11.4:
REC = F(WHILE)

Este apartado vamos a demostrar una de las inclusiones necesarias para la demostración completa de este teorema. La demostración, ya que se trata de una igualdad entre dos conjuntos, se haría por doble inclusión, y nosotros veremos la demostración de una de dichas inclusiones; concretamente la primera de las dos proposiciones siguientes.

Proposición 11.5:
$REC \subseteq F(WHILE)$

Proposición 11.6:
$F(WHILE) \subseteq REC$

Para demostrar la proposición **11.5** demostraremos que todas las funciones iniciales son *WHILE*-calculables, y que las funciones obtenidas por composición, recursión primitiva y minimización no acotada de funciones *WHILE*-calculables son también *WHILE*-calculables.

Proposición 11.7:
$INI \subseteq F(WHILE)$

Demostración:

a) $\theta \in F(WHILE)$

Sea el programa $Q = (0\ ,\ 1\ ,\ X_1 := 0\ )$.

$F_Q = \theta \ \Rightarrow\ \theta \in F(WHILE)$ a) □

b) $\sigma \in F(WHILE)$

Sea el programa $Q = (1\ ,\ 1\ ,\ X_1 := X_1 + 1\ )$.

$F_Q = \sigma \ \Rightarrow\ \sigma \in F(WHILE)$ b) □

c) $\pi \subseteq F(WHILE)$

Sea el programa $Q = (k\ ,\ k\ ,\ X_1 := X_i\ )$, con $k \geq 1$ y $1 \leq i \leq k$.

$F_Q = \pi_i^k \ \Rightarrow\ \pi_i^k \in F(WHILE) \ \Rightarrow\ \pi \subseteq F(WHILE)$ c) □

$INI = \pi \cup \{\ \theta\ ,\ \sigma\ \} \ \Rightarrow\ INI \subseteq F(WHILE)$

↑

por a), b) y c) □

Proposición 11.8:
$f = g(h_1\ ,\ h_2\ ,\ ...,\ h_m) \ \wedge\ g\ ,\ h_1\ ,\ h_2\ ,\ ...,\ h_m \in F(WHILE) \ \Rightarrow\ f \in F(WHILE)$

Demostración:

Sean $m > 0$, $k \geq 0$ y las funciones:

$g : \mathbb{N}^m \rightarrow \mathbb{N}$

$h_1\ ,\ h_2\ ,\ ...,\ h_m : \mathbb{N}^k \rightarrow \mathbb{N}$

y sea $f = g(h_1\ ,\ h_2\ ,\ ...,\ h_m)$.

Sea el macroprograma $Q = (k\ ,\ k+m\ ,\ s)$ con s :

$X_{k+1} := h_1(X_1\ ,\ X_2\ ,\ ...,\ X_k)$;

$X_{k+2} := h_2(X_1\ ,\ X_2\ ,\ ...,\ X_k)$;

. . .

$X_{k+m} := h_m(X_1\ ,\ X_2\ ,\ ...,\ X_k)$;

$X_1 := g(X_{k+1}\ ,\ X_{k+2}\ ,\ ...,\ X_{k+m})$

$F_Q = f \ \Rightarrow\ f \in F(WHILE)$ □

Proposición 11.9:

$f = \langle g|h \rangle \ \wedge \ g, h \in F(WHILE) \ \Rightarrow \ f \in F(WHILE)$

Demostración:

Sea $k \geq 0$ y las funciones:

$g : \mathbb{N}^k \rightarrow \mathbb{N}$

$h : \mathbb{N}^{k+2} \rightarrow \mathbb{N}$

y sea $f = \langle g|h \rangle$.

Sea el macroprograma $Q = (k+1, k+3, s)$ con s :

$X_{k+2} := g(X_1, X_2, ..., X_k)$;
do X_{k+1} *times*
    $X_{k+2} := h(X_1, X_2, ..., X_k, X_{k+3}, X_{k+2})$;
    $X_{k+3} := X_{k+3} + 1$
od ;
$X_1 := X_{k+2}$

$F_Q = f \ \Rightarrow \ f \in F(WHILE)$ □

Proposición 11.10:

$f = \mu[g] \ \wedge \ g \in F(WHILE) \ \Rightarrow \ f \in F(WHILE)$

Demostración:

Sea $k \geq 0$ y la función:

$g : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$

y sea $f = \mu[g]$.

Sea el macroprograma $Q = (k, k+1, s)$ con s :

while $g(X_1, X_2, ..., X_k, X_{k+1}) \neq 0$ *do*
    $X_{k+1} := X_{k+1} + 1$
od ;
$X_1 := X_{k+1}$

$F_Q = f \ \Rightarrow \ f \in F(WHILE)$ □

Al demostrar estas cuatro proposiciones (**11.7**, **11.8**, **11.9**, **11.10**) queda demostrada la proposición **11.5**. Para terminar la demostración del teorema **11.4** sólo tendríamos que demostrar la proposición **11.6**, si bien, como ya hemos comentado, esto queda fuera el objetivo del presente libro. El lector puede encontrar dicha demostración completa en el libro *Teoría de Autómatas y Lenguajes Formales I* (Gonzalo Ramos Jiménez, 2005).

12
Universalidad

12.1 Codificación de programas *WHILE*

En este apartado primero veremos el concepto de indexación de funciones, para a continuación abordar cómo se codifican (o gödelizan) los programas *WHILE*. Esa codificación será nuestra indexación de REC^n (funciones recursivas de n argumentos) para cada $n \geq 0$. Esto nos permitirá definir en el apartado siguiente el concepto de función universal, donde veremos el teorema que motiva este tema, el cual terminaremos con el programa universal que computa dicha función universal.

Definición 12.1: *Indexación de funciones, Índice de una función*
Sea F un conjunto numerable de funciones.
Denominaremos indexación de F a toda función sobreyectiva $h: \mathbb{N} \to F$.
Un índice de una función $f \in F$ bajo una indexación h es cualquier número natural i tal que $h(i) = f$.

Nota: Bajo una indexación toda función de F tiene al menos un índice, pero una función puede tener más de un índice, ya que para ser indexación sólo se exige que h sea sobreyectiva.

Ejemplo 12.2:
Sea $F = \{ f: \mathbb{N} \to \mathbb{N} \mid f(x) = p \cdot x + q \text{ con } p,q \in \mathbb{N} \}$.
Una indexación de F es:
$h: \mathbb{N} \to F$
$h(i)(x) = \sigma_{2,1}^{1}(i) \cdot x + \sigma_{2,2}^{1}(i)$

Nota: En este caso la indexación h es biyectiva, si bien esto no es necesario para ser indexación □

Para abordar la codificación de programas necesitamos antes introducir los conceptos de codificación y decodificación de Gödel, también llamados *gödelización* y *degödelización* respectivamente.

Estas funciones trabajan sobre el conjunto de todos los vectores de naturales (o sucesiones finitas de naturales), al que denominaremos $\mathbb{N}^*$, es decir:

$$\mathbb{N}^* = \bigcup_{n=0}^{\infty} \mathbb{N}^n \quad \text{con} \quad \mathbb{N}^0 = \varepsilon$$

Dado que el cardinal de $\mathbb{N}^*$ es igual que el de $\mathbb{N}$ podemos establecer una función biyectiva entre estos dos conjuntos.

Sea $god : \mathbb{N}^* \to \mathbb{N}$ una función biyectiva y recursiva, a la que llamaremos gödelización. La definición de dicha función se puede consultar en el anexo titulado "Codificación de Gödel", si bien en este apartado sólo necesitamos saber que dicha función biyectiva recursiva existe. Esta función nos va a permitir "codificar" un vector de naturales (de tamaño variable) en un solo natural.

Por otra parte, sea $degod : \mathbb{N}^2 \to \mathbb{N}$ una función recursiva y total que nos permite obtener de una gödelización la componente que deseemos, es decir:

$$degod(z,i) = \begin{cases} 0 & \text{si } z = 0 \ \vee \ k > l(z) \\ x_i & \text{si } god(x_1, x_2, \ldots, x_i, \ldots, x_{l(z)}) = z \end{cases}$$

donde $l : \mathbb{N} \to \mathbb{N}$ es una función recursiva y total que devuelve el tamaño del vector gödelizado por un natural dado, es decir:

$l(z) = m$ sii $god(\underline{x}) = z \ \wedge \ \underline{x} \in \mathbb{N}^m$

La definición de la función *degod*, a la que llamaremos degödelización, así como la de la función l, se puede consultar en el anexo titulado "Codificación de Gödel", si bien en este apartado sólo necesitamos saber que existe dicha función recursiva y total. Esta función nos va a permitir "decodificar" un natural, obteniendo la componente que deseemos del vector gödelizado por dicho natural. Si la componente pedida es mayor que el tamaño del vector entonces *degod* devuelve cero.

A diferencia de la cantorización, en la cual hay una función distinta para cada tamaño de vector, la gödelización es una única función, ya que trabaja sobre el conjunto de todos los vectores de naturales.

Ya estamos en disposición de introducir la codificación (o gödelización) de programas *WHILE*. Ésta consiste en definir una biyección recursiva entre el

conjunto de los programas *WHILE* y los naturales, la cual asignará biunívocamente a cada programa un número (su número).

Para ello primero veremos cómo *codificar sentencias* (función *codi*), después cómo *codificar códigos* (función *Codi*), y finalmente cómo *codificar programas* (función *CODI*).

Definición 12.3: *Función codificación de sentencias (codi)*
Definimos la función codificación de sentencias, notada *codi* , como:
$codi : Cód\_While \to \mathbb{N}$

$$codi(s) = \begin{cases} 5(i\text{-}1) & \text{si } s = X_i := 0 \\ 5\sigma_1^2(i\text{-}1, j\text{-}1)+1 & \text{si } s = X_i := X_j \\ 5\sigma_1^2(i\text{-}1, j\text{-}1)+2 & \text{si } s = X_i := X_j + 1 \\ 5\sigma_1^2(i\text{-}1, j\text{-}1)+3 & \text{si } s = X_i := X_j - 1 \\ 5\sigma_1^2(i\text{-}1, Codi(s_1))+4 & \text{si } s = while\ X_i \neq 0\ do\ s_1\ od \end{cases}$$

donde *Codi* es la función codificación de códigos que definimos a continuación.

Definición 12.4: *Función codificación de códigos (Codi)*
Si s es un código *While* con $s = s_1 ; s_2 ; ...; s_m$ donde cada s_i es una sentencia, entonces:
$Codi(s) = god(codi(s_1), codi(s_2), ..., codi(s_m)) - 1$

Notas: Restamos uno para que *Codi* sea biyectiva.
Si $m=1$ entonces, siguiendo la definición, $Codi(s) = god(codi(s_1)) - 1$.

Definición 12.5: *Función codificación de programas (CODI)*
Sea $Q \in WHILE$, con $Q = (n, p, s)$.
Definimos la función codificación de programas, notada *CODI* , como:
$CODI : WHILE \to \mathbb{N}$
$CODI(Q) = \sigma_1^3(n, p - max\{n,k\}, Codi(s))$
donde $k = max\{ m \in \Sigma_d^+ \mid X_m \infty s \}$

Notas: k es el mayor índice de variable que aparece en el código s .
Siempre se cumple que $p \geq max\{n,k\}$.

Una vez que tenemos la función codificación de programas, para definir su inversa requerimos también de tres decodificaciones distintas. No obstante, antes de definir las decodificaciones necesitamos introducir dos funciones auxiliares, la función *tipo de sentencia*, y la función *extraer*.

Definición 12.6: *Función tipo de sentencia (tipo)*
Definimos la función tipo de sentencia, notada *tipo* , como:
$codi : \mathbb{N} \rightarrow \{0, 1, 2, 3, 4\}$
$codi(z) = z \bmod 5$
donde *mod* es la función módulo (en este caso módulo cinco).

Nota: Esta función nos indica, con un número entre cero y cuatro, qué tipo de sentencia codifica el argumento (z) según el orden establecido en la definición **12.3**.

Definición 12.7: *Función extraer (extr)*
Definimos la función extraer, notada *extr* , como:
$extr : \mathbb{N} \times \{1,2\} \rightarrow \mathbb{N}$

$$extr(z,i) = \begin{cases} 1 + z/5 & \text{si } tipo(z) = 0 \\ 1 + \sigma_{2,i}^{1}((z - tipo(z))/5) & \text{si } tipo(z) \neq 0 \end{cases}$$

Notas: i señala qué variable de la sentencia codificada por z deseamos, si la 1^a (izq.) o la 2^a (der.).
Cuando $tipo(z) = 4$, $extr(z,2)$ nos devuelve $Codi(s_1)+1$, es decir, la gödelización del cuerpo del bucle.

Para ver las decodificaciones seguiremos el mismo orden que para las codificaciones. Así, primero veremos cómo *decodificar sentencias* (función *decodi*), después cómo *decodificar códigos* (función *DeCodi*), y finalmente cómo *decodificar programas* (función *DECODI*).

Definición 12.8: *Función decodificación de sentencias (decodi)*
Definimos la función decodificación de sentencias, notada *decodi* , como:
$decodi : \mathbb{N} \rightarrow Cód\_While$

$$decodi(z) = \begin{cases} X_i := 0 & \text{si } tipo(z) = 0 \\ X_i := X_j & \text{si } tipo(z) = 1 \\ X_i := X_j + 1 & \text{si } tipo(z) = 2 \\ X_i := X_j - 1 & \text{si } tipo(z) = 3 \\ \textit{while } X_i \neq 0 \textit{ do } DeCodi(j-1) \textit{ od} & \text{si } tipo(z) = 4 \end{cases}$$

donde $i = extr(z,1)$, $j = extr(z,2)$ y $DeCodi$ es la función decodificación de códigos que definimos a continuación.

Definición 12.9: *Función decodificación de códigos (DeCodi)*
Definimos la función decodificación de codigos, notada $DeCodi$, como:
$DeCodi : \mathbb{N} \to Cód\_While$
$DeCodi(z) =$
$\quad decodi(degod(z+1,1)) ; decodi(degod(z+1,2)) ; ... ; decodi(degod(z+1,l(z)))$

Definición 12.10: *Función decodificación de programas (DECODI)*
Definimos la función decodificación de programas, notada $DECODI$, como:
$DECODI : \mathbb{N} \to WHILE$
$DECODI(z) = (\sigma_{3,1}^{1}(z) , \sigma_{3,2}^{1}(z) + max\{\sigma_{3,1}^{1}(z), k\} , DeCodi(\sigma_{3,3}^{1}(z)))$
donde $k = max\{ m \in \Sigma_d^+ \mid X_m \prec DeCodi(\sigma_{3,3}^{1}(z)) \}$

Dado que para toda función *WHILE*-computable hay al menos un programa *WHILE* que la computa (de hecho hay infinitos), nuestra decodificación de programas ($DECODI$) nos va a permitir definir una indexación de $F(WHILE)$; o lo que es lo mismo según el teorema de equivalencia, una indexación de REC .

De esta forma tenemos que:

$h : \mathbb{N} \to F(WHILE)$
$h(z) = F_{DECODI(z)}$

es una indexación de REC .

Por otra parte, puesto que se cumple que:

$F_{(n,\ max\{n,k\},\ s)} = F_{(n,\ max\{n,k\}+i,\ s)}$ con $k = max\{ m \in \Sigma_d^+ \mid X_m \prec s \}$ e $i \in \mathbb{N}$

nuestra decodificación de códigos ($DeCodi$) nos va a permitir definir una indexación de $F^n(WHILE)$; o lo que es lo mismo según el teorema de equivalencia, una indexación de REC^n .

De esta forma tenemos que:

$h : \mathbb{N} \to F^n(WHILE)$

$h(z) = F_{(n,\ max\{n,k\},\ DeCodi(z))}$ donde $k = max\{\ m \in \Sigma_d^+ \mid X_m \prec DeCodi(z)\ \}$

es una indexación de REC^n .

12.2 Función universal

Definición 12.11: *Función universal (U)*

Sea F un conjunto numerable de funciones de $\mathbb{N}^n$ en $\mathbb{N}$.

Diremos que la función $U[F] : \mathbb{N}^{n+1} \to \mathbb{N}$ es universal para la clase F sii:

$\exists\, h : \mathbb{N} \to F$ indexación de F | $U[F](i, \underline{x}) = h(i)(\underline{x})$ $\forall i \in \mathbb{N} \wedge \forall \underline{x} \in \mathbb{N}^n$

Notas: $h(i) \in F$

Cuando escogemos una indexación concreta h para $U[F]$, decimos que $U[F]$ es la función universal para F bajo la indexación h .

En adelante, nuestra función universal para REC^n ($U[REC^n]$) lo será bajo la indexación de REC^n vista anteriormente.

Ahora estamos en disposición de enunciar el teorema que motiva este tema.

Teorema 12.12:
$U[REC^n] \in REC^{n+1}$

Demostración:

Supongamos que: $\exists U \in WHILE \mid F_U = U[REC^n]$

$$\Downarrow$$

$$U[REC^n] \in F^{n+1}(WHILE)$$

$$\Downarrow$$

$$U[REC^n] \in REC^{n+1}$$

Así pues, sólo tenemos que demostrar que $\exists U \in WHILE \mid F_U = U[REC^n]$, lo cual haremos en el siguiente apartado.

Nota: A dicho programa U lo denominaremos programa universal. □

12.3 Programa universal

Por definición, el programa universal U calcula la función universal $U[REC^n]$, por lo que U tendrá $n+1$ variables de entrada, habiendo un programa universal distinto para cada valor de n .

Según nuestra indexación, un índice de una función recursiva de n variables de entrada es el número del código de cualquier programa *WHILE* que la calcule. Así pues tenemos que $U(z, \underline{x})$ devuelve la salida del programa $(n, max\{n,k\}, DeCodi(z))$ donde $k = max\{ m \in \Sigma_d^+ \mid X_m \prec DeCodi(z) \}$ para el vector de entrada $\underline{x}$.

En adelante, para una mayor legibilidad, al describir un macroprograma en lenguaje *WHILE* ampliado, en lugar de escribir:

Nombre
Entradas: $\underline{x}$
Salida: X_1
Código:
s

escribiremos:

Nombre($\underline{x}$)
s

Nota: Todo macroprograma descrito mediante esta abreviatura tendrá como variable de salida X_1 .

En las sucesivas definiciones presentamos, de manera descendente, el programa universal.

Definición 12.13: *Programa universal (U)*

$U(z, \underline{x})$

$m := god(\underline{x})$; (* inicializa m , la variable de la memoria *)
$m := Simular(z, m)$; (* simula el programa z con la memoria m *)
$X_1 := degod(m,1)$ (* obtiene de la memoria la salida de z *)

Tanto la función *god* como la función *degod* son recursivas, por lo que pertenecen a *F(WHILE)*. El programa *Simular* , que simula el funcionamiento del programa z con la memoria m (gödelización del vector de entrada $\underline{x}$), se presenta a continuación.

Definición 12.14: *Programa Simular*

```
Simular(z, m)
    z := z+1 ;                          (* sumamos uno ya que al codificar restamos uno *)
    while  l(z) ≠ 0  do                 (* mientras haya sentencias pendientes *)
        s := degod(z,1) ;                   (* obtengo el número de la primera sentencia *)
        if  tipo(s) ≤ 3  then               (* si es una asignación *)
            m := Ejecutar(s, m) ;               (* ejecutarla *)
            z := Reducir(z) ;                   (* y eliminar la sentencia de z *)
        else                                (* si es una sentencia while *)
            if  degod(m, extr(s,1)) ≠ 0  do     (* si la variable de control no es nula *)
                z := Añadir(z, extr(s,2))           (* añade a z el cuerpo del bucle *)
            else                                (* si la variable de control es nula *)
                z := Reducir(z)                     (* eliminar la sentencia de z *)
            fi
        fi
    od ;
    X1 := m
```

Antes de ver los programas *Ejecutar* , *Reducir* y *Añadir* , necesitamos introducir una función que será utilizada más adelante.

Definición 12.15: *Función reemplazar (reem)*

Definimos la función reemplazar, notada *reem* , como:

$reem : \mathbb{N}^3 \rightarrow \mathbb{N}$

$reem(z,k,x) =$

$$\begin{cases} god(degod(z,1), ..., degod(z,l(z)), 0, ..., 0, x) & \text{si} \quad k > l(z) \ \wedge \ l(z) > 0 \\ god(0, ..., 0, x) & \text{si} \quad k > l(z) \ \wedge \ l(z) = 0 \\ z & \text{si} \quad k = 0 \\ god(degod(z,1), ..., degod(z,k-1), x, degod(z,k+1), ..., degod(z,l(z))) & \\ & \text{si} \quad k \leq l(z) \ \wedge \ k \neq 0 \end{cases}$$

Notas: *reem(z,k,x)* es el código del vector obtenido tras reemplazar, en el vector codificado por z , la componente k por el valor x .
A diferencia de la función *reem* para Cantor, aquí puede aumentar la longitud del vector tras aplicar la función, ya que esta función es única y no una para cada tamaño de vector.

El programa *Ejecutar* , que ejecuta las asignaciones; el programa *Reducir* , que elimina la primera instrucción pendiente (por la cima) de z ; y el programa *Añadir* , que añade por la cima a z (que funciona como una pila) la secuencia de instrucciones (sus números) del cuerpo de un bucle; se presentan a continuación.

Definición 12.16: *Programa Ejecutar*

```
Ejecutar(s, m)
    if tipo(s) = 0 then
        m := reem(m, extr(s,1), 0)
    fi ;
    if tipo(s) = 1 then
        m := reem(m, extr(s,1), degod(m, extr(s,2)))
    fi ;
    if tipo(s) = 2 then
        m := reem(m, extr(s,1), degod(m, extr(s,2))+1)
    fi ;
    if tipo(s) = 3 then
        m := reem(m, extr(s,1), degod(m, extr(s,2))-1)
    fi ;
    X1 := m
```

Definición 12.17: *Programa Reducir*

```
Reducir(z)
    if l(z) < 2 then z := 0
    else z := god(degod(z,2), degod(z,3), ..., degod(z,l(z)))
    fi ;
    X1 := z
```

Definición 12.18: *Programa Añadir*

Añadir(z,s)

$X_1 := god(degod(s,1), \ldots, degod(s,l(s)), degod(z,1), \ldots, degod(z,l(z)))$

Nota: Lo que añadimos a z (por la cima) no es el código del cuerpo del bucle (s), sino la secuencia de instrucciones codificadas por s . Ese es el motivo de que degödelicemos s antes de volver a gödelizar z .

13
Limitaciones formales de la computación

13.1 Problema de la parada

Veamos en primer lugar en qué consiste un problema asociado a un predicado.

Dado un predicado P de n argumentos, y un vector $\underline{a} \in \mathbb{N}^n$, el problema P asociado a dicho predicado (lo notaremos de la misma forma), es la siguiente pregunta:

¿Se cumple el predicado P para el vector $\underline{a}$?

Es decir: ¿ $P(\underline{a})$ es verdadero ?

O más abreviadamente: ¿ $P(\underline{a})$?

Al vector $\underline{a}$ se le denomina instancia del problema P .

Dado que los problemas van asociados a predicados, la clasificación de los primeros estará en función de la de los segundos. En la siguiente tabla se muestra la denomiación del problema, y el tipo de predicado, y de función, del que es asociado.

problema	predicado	función
resoluble	decidible	$\in$ *TREC*
parcialmente resoluble	enumerable	$\in$ *REC*
no resoluble	no decidible	$\notin$ *TREC*
totalmente no resoluble	no enumerable	$\notin$ *REC*

Definición 13.1: *Predicado H*

Sea g el número de un programa *WHILE*.

Sea $\underline{x}$ un vector de entrada para g .

$H(g, \underline{x})$ es verdadero sii el programa g con la entrada $\underline{x}$ termina.

Definición 13.2: *Predicado* H^n

Sea g el número de un programa *WHILE* de n entradas.

Sea $\underline{x}$ un vector de entrada para g ($\underline{x} \in \mathbb{N}^n$).

$H^n(g, \underline{x})$ es verdadero sii el programa g con la entrada $\underline{x}$ termina.

El *problema de la parada* para el caso general es el problema asociado al predicado H (lo notaremos igual).

El problema de la parada para programas con n entradas es el problema asociado al predicado H^n (lo notaremos igual).

Teorema 13.3:
H^l es no resoluble.

Demostración:
Lo demostraremos por reducción al absurdo.
Supongamos que: H^l es resoluble

$$\Downarrow$$

H^l es decidible

$$\Downarrow$$

$\exists H^l \in TREC \mid H^l$ es la función característica del predicado H^l

Es decir, la función H^l definida como:

$$H^l(g, x) = \begin{cases} 1 & \text{si el programa de número } g \text{ con la entrada } x \text{ termina} \\ 0 & \text{si el programa de número } g \text{ con la entrada } x \text{ no termina} \end{cases}$$

pertenece a $TREC$.

Consideremos ahora la función *mala* definida como:

$$mala(x) = \begin{cases} 1 & \text{si } H^l(x,x) = 0 \\ \uparrow & \text{si } H^l(x,x) = 1 \end{cases}$$

La función *mala* la calcula el macroprograma $Mala = (1,2,c)$ con c:

$X_2 := X_2 + 1$;
While $H^l(X_1, X_1) \neq 0$ *do*
    $X_2 := 0$
od ;
$X_1 := X_2$

Sea m el número del programa *Mala*.

¿Cuál es el valor de $H^l(m, m)$?

- Si $H^l(m, m) = 1 \Rightarrow$ el programa m con la entrada m acaba $\Rightarrow$

  $\uparrow$ por definición de H^l — $\uparrow$ por definición de *mala*

$mala(m) = \uparrow \Rightarrow H^1(m, m) = 0$ ➔➔

por ser m el número de *Mala* y por definición de H^1

- Si $H^1(m, m) = 0 \Rightarrow$ el programa m con la entrada m no acaba $\Rightarrow$

por definición de H^1 por definición de *mala*

$mala(m) = 1 \Rightarrow H^1(m, m) = 1$ ➔➔

por ser m el número de *Mala* y por definición de H^1

En ambos casos hemos llegado a una contradicción, por lo que el problema H^1 es no resoluble.

Nota: Este resultado es extensible a H^n y a H. □

Proposición 13.4:
H^1 es parcialmente resoluble.

Demostración:
Sea la función f: $\mathbb{N}^2 \rightarrow \mathbb{N}$ definida como:

$$f = \sigma(U[REC^1])$$

$P_f = H^1 \Rightarrow H^1 \in PRED(REC) \Rightarrow H^1$ es parcialmente resoluble

$f \in REC$

Notas: $f(g, x) > 0$ sii el programa g con la entrada x termina.
Este resultado es extensible a H^n y a H. □

13.2 Función castor afanoso

En este apartado vamos a ver una función no computable; es decir, una función que no es *WHILE*-computable, y por tanto no es Turing-computable ni recursiva. Para ello necesitamos primero una definición previa.

Definición 13.5: *Longitud de un programa WHILE (long),* P^n

Sea $Q \in WHILE$, con $Q = (n, p, s)$.

Definimos la función longitud de un programa *WHILE*, notada *long* , como:

$long : WHILE \rightarrow \mathbb{N}$

$long(Q) = |s|_{do} + |s|_{:=}$

Llamaremos P^n al conjunto de todos los programas de longitud n , es decir:

$P^n = \{ Q \in WHILE \mid long(Q) = n \}$

Nota: La longitud de un programa es el número de sentencias que tiene su código, mientras que el tamaño (definiciones **10.3** y **10.4**) es el número de líneas de dicho código.

Ejemplo 13.6:

Sea $Q = (1, 2, s)$ con s :

$X_2 := X_2 + 1$;
While $X_1 \neq 0$ *do*
$X_2 := 0$
od ;
$X_1 := X_2$

Tenemos que $Q \in P^4$ ya que $long(Q) = 4$ □

Ahora estamos en disposición de definir la función no computable que da título a este apartado.

Definición 13.7: *Función castor afanoso (Σ)*

Definimos la función castor afanoso, notada Σ, como:

$\Sigma : \mathbb{N} \rightarrow \mathbb{N}$

$\Sigma(n) = max\{ Q(0) \mid Q \in WHILE^1 \wedge Q \in P^n \wedge Q(0) \in \mathbb{N} \}$

Nota: $\Sigma(n)$ es el mayor valor, de todos los que devuelven (en X_1 al terminar) los programas *WHILE* de una entrada ($WHILE^1$) de longitud exactamente n , cuando dicha entrada vale cero.

Ahora vamos a ver dos proposiciones previas que necesitamos antes de ver el teorema.

Proposición 13.8:
$\Sigma(n+1) > \Sigma(n) \quad \forall n \in \mathbb{N}$

Demostración:
Sea $Q = (1, p, s)$ un programa con $long(Q) = n$ y que $Q(0) = \Sigma(n)$ (es seguro que al menos hay uno de dicha longitud que para el cero como entrada devuelve $\Sigma(n)$).
Sea $Q' = (1, p, s')$ un programa, con s':

s ;
$X_1 := X_1 + 1$

Se cumple que: $long(Q') = n+1 \ \wedge \ Q'(0) = \Sigma(n)+1$

$$\Downarrow$$

$$\Sigma(n+1) \ \geq \ \Sigma(n)+1 \ > \ \Sigma(n)$$

$$\Downarrow$$

$$\Sigma(n+1) > \Sigma(n)$$

□

Proposición 13.9:
$\exists Q \in P^k \mid f = F_Q \ \Rightarrow \ \Sigma(n+k) \geq f(n) \quad \forall n \in \mathbb{N}$

Demostración:
Sea $Q = (1, p, s)$ un programa con $long(Q) = k$ y $F_Q = f$.
Sea $Q' = (1, p, s')$ un programa, con s':

$X_1 := X_1 + 1$;
$\cdots$
$X_1 := X_1 + 1$;
} n veces
s

Se cumple que: $long(Q') = n+k \ \wedge \ Q'(0) = f(n)$

$$\Downarrow$$

$$\Sigma(n+k) \geq f(n)$$

□

Ahora estamos en disposición de ver el teorema que motiva este apartado.

Teorema 13.10:
$\Sigma \notin F(WHILE)$

Demostración:
Lo demostraremos por reducción al absurdo.
Supongamos que: $\Sigma \in F(WHILE)$
Sea $g(n)=2n$; $g \in F(WHILE)$
Sea $f(n)=\Sigma(2n)$

$\Rightarrow f \in F(WHILE) \Rightarrow$

la composición de funciones computables es una función computable

$\exists Q \in P^k \mid F_Q = f \Rightarrow \Sigma(n+k) \geq f(n) \quad \forall n \in \mathbb{N} \Rightarrow \Sigma(n+k) \geq \Sigma(2n) \quad \forall n \in \mathbb{N} \Rightarrow$

por **13.9** $f(n)=\Sigma(2n)$ *para* $n=k+1$

$\Sigma(2k+1) \geq \Sigma(2k+2)$ ➔➔ $\Sigma(2k+1) < \Sigma(2k+2)$

por **13.8**

Por tanto $\Sigma \notin F(WHILE)$. □

Hay una estrecha relación entre la función castor afanoso y el problema de la parada. Dicha relación queda expuesta en la siguiente proposición.

Proposición 13.11:
$\Sigma \notin F(WHILE) \Rightarrow H^1$ es no resoluble

Demostración:
Lo demostraremos por reducción al absurdo.
Supongamos, con $\Sigma \notin F(WHILE)$, que H^1 es resoluble.
En ese caso puedo seleccionar, para cada longitud dada, sólo los programas que terminan para el cero como entrada. Tras ejecutarlos para el cero, escojo la mayor salida.
Esto implica que: $\Sigma \in F(WHILE)$ ➔➔

Por tanto:

$$\Sigma \notin F(WHILE) \Rightarrow H^1 \text{ es no resoluble}$$

Nota: Para cualquier longitud, el número de programas que calculan funciones distintas siempre es finito, ya que los restantes son el resultado de renumerar variables. □

ANEXOS

Anexo I: Máquinas de Mealy y de Moore

Anexo II: Codificación de Cantor

Anexo III: Codificación de Gödel

Anexo I
Máquinas de Mealy y de Moore

En este anexo presentamos los dos tipos de traductores finitos más conocidos: el de Mealy y el de Moore. Si bien formalmente no son más que autómatas finitos a los que se añade una función de salida, y no aportan nada a la jerarquía de Chomsky, hemos creído conveniente al menos definirlos aquí para que el lector conozca el significado del término traductor finito dentro de la teoría de autómatas.

Definición: *Máquina de Mealy*

Una máquina de Mealy es una séxtupla $M = (K , \Sigma , \Lambda , \delta , \phi , s)$ donde:

K es un conjunto finito no vacío de estados;

Σ es un alfabeto, el de entrada;

Λ es un alfabeto, el de salida;

δ es una función, la de transición, definida como $\delta: K \times \Sigma \to K$;

ϕ es una función, la de salida, definida como $\phi: K \times \Sigma \to \Lambda$;

s es el estado inicial $(s \in K)$.

Definición: *Máquina de Moore*

Una máquina de Moore es una séptupla $M = (K , \Sigma , \Lambda , \delta , \varphi , s , F)$ donde:

K es un conjunto finito no vacío de estados;

Σ es un alfabeto, el de entrada;

Λ es un alfabeto, el de salida;

δ es una función, la de transición, definida como $\delta: K \times \Sigma \to K$;

φ es una función, la de salida, definida como $\varphi: K \to \Lambda$;

s es el estado inicial $(s \in K)$;

F es el conjunto de estados finales $(F \subseteq K)$.

Anexo II
Codificación de Cantor

Definimos la codificación de Cantor, también llamada cantorización, de tamaño dos, notada σ_1^2 , como:

$\sigma_1^2 : \mathbb{N}^2 \to \mathbb{N}$
$\sigma_1^2(x,y) = (x+y)(x+y+1)/2 + y$

Generalizando, definimos la codificación de Cantor (o cantorización) de tamaño $k+1$, con $k > 1$, como:

$\sigma_1^{k+1} : \mathbb{N}^{k+1} \to \mathbb{N}$
$\sigma_1^{k+1}(x_1, x_2, \ldots, x_{k+1}) = \sigma_1^2(x_1, \sigma_1^k(x_2, \ldots, x_{k+1}))$

Por completitud, diremos que la función σ_1^1 es la función identidad para los naturales.

Definimos la decodificación de Cantor, también llamada decantorización, de tamaño dos, de la segunda componente, notada $\sigma^1_{2,2}$, como:

$\sigma^1_{2,2} : \mathbb{N} \to \mathbb{N}$
$\sigma^1_{2,2}(z) = z - \sigma_1^2(d(z), 0)$

donde $d(z) = min\{ t \in \mathbb{N} \mid \sigma_1^2(t,0) > z \} - 1$

(aritméticamente: $d(z) = \lfloor (\sqrt{8z+1} - 1) / 2 \rfloor$)

Definimos la decodificación de Cantor (o decantorización), de tamaño dos, de la primera componente, notada $\sigma^1_{2,1}$, como:

$\sigma^1_{2,1} : \mathbb{N} \to \mathbb{N}$
$\sigma^1_{2,1}(z) = d(z) - \sigma^1_{2,2}(z)$

Generalizando, definimos decodificación de Cantor (o decantorización), de tamaño $k+1$, de la i-ésima componente, con $k > 1$ y $1 \leq i \leq k+1$, como:

$$\sigma^{1}_{k+1,i} : \mathsf{N} \rightarrow \mathsf{N}$$

$$\sigma^{1}_{k+1,i}(z) = \begin{cases} \sigma^{1}_{2,1}(z) & \text{si } i=1 \\ \sigma^{1}_{k,i-1}(\sigma^{1}_{2,2}(z)) & \text{si } 2 \leq i \leq k+1 \end{cases}$$

Por completitud, diremos que la función $\sigma^{1}_{1,1}$ es la función identidad para los naturales.

Anexo III
Codificación de Gödel

Definimos la función longitud de un vector, notada L , como:

$L : \mathbb{N}^* \to \mathbb{N}$
$L(\varepsilon) = 0$
$L(\underline{x}) = m \quad \forall(\underline{x}) \in \mathbb{N}^m \qquad$ (con $m > 0$)

Definimos la codificación de Gödel, también llamada gödelización, notada *god* , como:

$god : \mathbb{N}^* \to \mathbb{N}$

$$god(\underline{x}) = \begin{cases} 0 & \text{si } L(\underline{x}) = 0 \\ \sigma_1^{\,2}(L(\underline{x}) - 1\ ,\ \sigma_1^{\,L(\underline{x})}(\underline{x})) + 1 & \text{si } L(\underline{x}) \neq 0 \end{cases}$$

Definimos la función longitud, notada l , como:

$l : \mathbb{N} \to \mathbb{N}$
$l(0) = 0$
$l(z) = \sigma^1_{2,1}(z\text{-}1) + 1 \qquad$ (para $z > 0$)

Definimos la decodificación de Gödel, también llamada degödelización, notada *degod* , como:

$degod : \mathbb{N}^2 \to \mathbb{N}$

$$degod(z,k) = \begin{cases} 0 & \text{si } z = 0 \ \vee\ k > l(z) \\ \sigma^1_{l(z),k}(\sigma^1_{2,2}(z\text{-}1)) & \text{resto} \end{cases}$$

Bibliografía

Hemos incluido en esta bibliografía algunos de los (escasos) títulos en español sobre el tema, así como aquellos en inglés que por más conocidos deben de poder encontrarse con facilidad en cualquier biblioteca universitaria.

Alfonseca Cubero, Enrique. *Teoría de autómatas y lenguajes formales.* McGraw-Hill (2007).

Brookshear, J. Glenn. *Teoría de la Computación. Lenguajes formales, autómatas y complejidad.* Addison-Wesley (1993).

Floyd, Robert W.; Beigel, Richard. *The Language of Machines. An Introduction to Computability and Formal Languages.* W. H. Freeman and Company (1994).

Harrison, M. A. *Introduction to Formal Language Theory.* Addison-Wesley (1978).

Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley (2001).

Hopcroft, John E.; Ullman, Jeffrey D.. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley (1979).

Isasi Viñuela, Pedro; Martínez Fernández, Paloma; Borrajo Millán, Daniel. *Lenguajes, Gramáticas y Autómatas. Un enfoque práctico.* Addison-Wesley (1997).

Kozen, Dexter C. *Automata and Computability.* Springer-Verlag, Inc. (1997).

Lewis, Harry R.; Papadimitriou, Christos H.. *Elements of the theory of computation.* Prentice-Hall, Inc. (1981). Second edition (1998).

Linz, Peter. *An introduction to formal languages and automata.* D. C. Heath and Company (1996).

Martin, John C. *Introduction to languages and the theory of computation.* McGraw-Hill (1991).

McNaughton, Robert. *Elementary Computability, Formal Languages, and Automata.* Z B Publishing Industries (1993).

Morales Bueno, Rafael; Ramos Jiménez, Gonzalo. *Modelos de Cómputo.* Edición de Autor (2007).

Ramos Jiménez, Gonzalo. *Teoría de Autómatas y Lenguajes Formales I.* Edición de Autor (2005).

Salomaa, Arto. *Formal Languages.* Academic Press, Inc. Ltd. (1973).

Sommerhalder, R.; Westrhenen, S. C. van. *The theory of computability. Programs, Machines, Effectiveness and Feasibility.* Addison-Wesley (1988).